Technical Report

UCAM-CL-TR-83
ISSN 1476-2986

Number 83

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Operating system design for large personal workstations

Ian David Wilson

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

© Ian David Wilson

This technical report is based on a dissertation submitted July 1985 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Darwin College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Summary

With the advent of personal computers in the mid 1970s, the design of operating systems has had to change in order to take account of the new machines. Traditional problems such as accounting and protection are no longer relevant, but compactness, efficiency and portability call have all `become important issues as the number of these small systems has grown.

Since that time, due to the reductions in the costs of computer components and manufacture, personal workstations have become much more common, with not only the number of machines having increased, but also their CPU power and memory capacity. The work on software for the new machines has not kept pace with the improvements in hardware design, and this is particularly true in the area of operating systems, where there is a tendency to treat the new machines as small, inferior mainframes.

This thesis investigates the possibility of enhancing work done on the original personal computer operating systems, so that better utilisation of the new machines can be obtained. The work concentrates on two main areas of improvement: the working environment as perceived by the user, and the underlying primitives and algorithms used by the operating system kernel.

The work is illustrated by two case studies. The user environment of the TRIPOS operating system is described, along with a new command line interpreter and command programming language, and a series of techniques to make better use of the available hardware facilities is discussed. The kernel of the TRIPOS operating system is examined critically, particularly with the respect to the way that machine resources are used, and finally, a new set of kernel primitives and algorithms is suggested, with reference to an experimental kernel for the real time implementation of network protocol software.

Contents

1. Introduction 1
- 1.1 Background 3
- 1.2 Motivation and aims 4
- 1.3 Work done 5
- 1.4 Layout of thesis 5
- 1.5 Extent of collaboration 6

2. Related Work 7
- 2.1 Operating system design 7
  - 2.1.1 OS6 8
  - 2.1.2 UNIX 9
  - 2.1.3 SOLO 11
  - 2.1.4 MUSS 13
  - 2.1.5 THOTH 14
  - 2.1.6 Lampson and Sproull 15
  - 2.1.7 PILOT 16
  - 2.1.8 LILITH 18
  - 2.1.9 TRIX 19
  - 2.1.10 SPICE 20
  - 2.1.11 MAYFLOWER 22
  - 2.1.12 Commercial operating systems 23
- 2.2 Command languages 23
  - 2.2.1 Stephenson's EXEC 24
  - 2.2.2 KRONOS and MULTICS 25
  - 2.2.3 Programmer's Workbench and the C Shell 26
  - 2.2.4 BASIC and LISP 29
  - 2.2.5 MXEC 30
  - 2.2.6 REXX 30

3. TRIPOS 32
- 3.1 Introduction 32
- 3.2 BCPL 32
- 3.3 The operating system 33
  - 3.3.1 The system tasks 34
  - 3.3.2 Input/Output 35
- 3.4 The command language interpreter 36
- 3.5 Arguments to commands 37
- 3.6 Programming conventions 38
  - 3.6.1 Storage allocation 39
  - 3.6.2 I/O streams 39
  - 3.6.3 Break conditions 39
- 3.7 The CLI environment 40
- 3.8 CLI commands 41

3.8.1 The PROMPT command 41
3.8.2 The RUN command 41
3.8.3 The IF command 41
3.8.4 The REPEAT command 42
3.9 Spooling 42
3.9.1 The SPOOL command 42
3.10 Command sequences 42
3.10.1 The T command 43
3.10.2 The LAB and SKIP commands 43
3.10.3 Parameters 44
3.10.4 The C command 44
3.10.5 Temporary command files 44
3.11 Return codes 45
3.12 Evaluation of the CLI 46

4. Enhancing the User Interface 48
4.1 Introduction 48
4.2 Areas of enhancement 48
4.3 Pipes 49
4.3.1 UNIX pipes 50
4.3.2 TRIPOS pipes 50
4.3.3 Pipe expressions 51
4.4 Command history mechanism 52
4.5 Command execution 54
4.5.1 The command executor task 54
4.5.2 Command arguments 55
4.5.3 Type ahead 56
4.5.4 Existing TRIPOS commands 57
4.5.5 Conclusions 57
4.6 Implementation of the Shell 58
4.6.1 Background execution 58
4.6.2 Foreground execution 60
4.6.3 Shell search paths 62
4.6.4 Built-in commands 62
4.6.5 Command line parsing 63
4.6.6 Global vector allocation 64
4.6.7 Other built-in commands 65
4.7 Summary 65

5. A Command Programming Language 67
5.1 Introduction 67
5.2 Motivation 68
5.3 Requirements 68
5.4 General philosophy 69
5.5 The REX language 70
5.5.1 Simple command sequence 70
5.5.2 Towers of Hanoi 70

5.6 Language issues 71
5.6.1 Data typing 71
5.6.2 Arrays and array elements 73
5.6.3 Records and other data structures 74
5.6.4 The necessity of NIL 75
5.6.5 Procedures 76
5.6.6 Parameter passing and variable scope 77
5.6.7 Dynamic type coercion 80
5.6.8 Error recovery and debugging 83
5.7 Implementation issues 83
5.7.1 Representation of REX programs 84
5.7.2 Storage allocation and garbage collection 84
5.8 Interface to the Shell 86
5.8.1 The OBEY statement 87
5.8.2 The QUEUE statement 89
5.8.3 I/O and the EX: device 90
5.8.4 The system variables 91
5.8.5 Execution of REX programs from the Shell 92
5.9 Portability and the REXSHELL module 92
5.10 Performance 93
5.10.1 REX performance test 94
5.10.2 Ackermann's function 95
5.11 Summary 96
5.12 Evaluation of REX 96

6. Additions to the User Environment 99
6.1 Introduction 99
6.2 Motivation 99
6.3 Winchester discs 100
6.4 The CORE: device 101
6.5 The PRELOAD system 103
6.5.1 The SEGLIB library 104
6.5.2 The PRELOAD: device 104
6.5.3 Requests to PRELOAD: 105
6.5.4 The pre-load list 107
6.5.5 Naming of pre-loaded programs 107
6.5.6 Problems with PRELOAD: 109
6.5.7 Alternative methods 109
6.5.8 Performance 110
6.6 Other TRIPOS devices 111
6.7 The WINDOW: device 111
6.7.1 Process buffers 112
6.7.2 The implementation of WINDOW: 113
6.7.3 The COHAND console handler 114
6.7.4 Summary 115
6.8 The EX: and EXALL: devices 116
6.9 The DIR: device 118

6.10 Summary 119

7. Kernel Issues 121
7.1 Introduction 121
7.2 Language issues 121
7.2.1 Operating system language 122
7.2.2 User languages 126
7.3 Processes and coroutines 128
7.3.1 Processes 129
7.3.2 Coroutines 129
7.4 Scheduling 131
7.4.1 Unique priority 132
7.4.2 Non-unique priority 134
7.4.3 Round robin 134
7.4.4 Which scheduling algorithm? 135
7.4.5 A compromise solution 136
7.5 Calling conventions 137
7.6 Inter-process communication 140
7.6.1 Messages 140
7.6.2 Monitors 143
7.7 Naming and location of services 144
7.8 Storage allocation 145
7.8.1 Dual list first fit 147
7.8.2 Buddy 150
7.8.3 Cartesian tree 151
7.8.4 Which storage allocation algorithm? 152
7.9 User Interface 152
7.10 Portability issues 156
7.10.1 Overall portability 156
7.10.2 User level portability 157
7.10.3 Same processor portability 158
7.11 Case study: the GMK kernel 159
7.11.1 GMK tasks 159
7.11.2 GMK messages 160
7.11.3 GMK flags 160
7.11.4 Kernel interfaces 160
7.11.5 Free store allocation 161
7.11.6 Task creation and deletion 162
7.11.7 Task re-scheduling 162
7.11.8 Message manipulation 162
7.11.9 Flag manipulation 162
7.11.10 Critical code sections 162
7.11.11 Device and interrupt handling 163
7.11.12 Evaluation of the GMK kernel 163
7.12 Summary and conclusions 164

8. Conclusion 167

References 175

Appendix 1: Shell details 184
Command history mechanism 184
Line editing 184
Shell commands 185
The HISTORY command 185
The SWITCH command 185
The CHAR command 186
Directory commands 186
The SET command 186
The UNSET command 186
The PWD, PUSHD and POPD commands 187

Appendix 2: REX details 188
REX programs 188
Comments 188
Numbers 188
Strings 188
Booleans 189
Variables 189
Nil 189
Operators 189
Arithmetic operators 189
Relational operators 190
String operators 191
Boolean operators 191
System variables 191
The ARGS variable 192
The LEN function 192
The SUBSTR function 192
Explicit coercions 192
Precedence of operators 193
Synonyms 193
Arrays 193
Array elements 193
The DIM statement 194
The TABLE statement 194
General control statements 195
The assignment statement 195
The IF statement 195
The DO statement 196
The BREAK statement 196
The LOOP statement 197
The SKIP statement 197
The EXIT statement 197
Procedures 197

Procedure calling 198
The PROC statement 198
The RETURN statement 198
The RESULT statement 198
The DUMPSTACK statement 199
Interface to the Shell 199
The SAY statement 199
The QUEUE statement 200
The OBEY statement 200
The PARSE statement 200
The RDARGS function 201
The PROMPT statement 201
Input/output 201
The OPENIN and OPENOUT functions 202
The READ statement 202
The WRITE statement 202
The CLOSE statement 202
The EOF function 203

1. Introduction

When the first production computers became available in the 1960s, they were so expensive to build and maintain that the primary consideration in their use was the amount of work they could do, and the number of users they could serve. Originally, access was through "batch jobs" where the user constructed his program, either on paper tape or punched cards, and then submitted his job to the operators for execution, collecting the results some time later.

As computers became larger and faster, so time-sharing became possible, with many users accessing a central mainframe interactively, being able to edit and develop their programs on disc rather than on paper tape or punched cards. The large multi-user operating systems were pre-occupied with accounting and protection, attempting to ensure fair and equal access for all their clients, and preventing users from interfering with each other and with the operating system itself. Significant advances were made with virtual memory and address translation mechanisms, since users could be kept apart by putting them in separate address spaces where they could harm no-one but themselves. True independence came with virtual machine operating systems, which created an environment as though each user had his own personal workstation.

At the same time, mini-computers such as the Data General NOVA, DEC PDP-11 and Computer Automation LSI4, were becoming cheaper and more readily available, and by the mid 1970s it was financially viable to give each user a real personal workstation. This trend was continued by the development and gradual refinement of the 8 bit micro-computers, such as the Motorola MC6800, the Signetics 6502, the Intel 8080 and the Zilog Z80. As micro-processors became more powerful and the cost of memory fell, it became even more attractive to have personal computers rather than shared access to a mainframe, since the interactive response time was better, and the cost of hardware and maintenance much lower.

The operating systems which ran on the mini- and micro-computers were, on the whole, very simple, and in many ways similar to the batch monitors written for the earliest mainframes. Because the machines had small amounts of memory, the system software had to be compact and uncomplicated, and hence usually single threaded. Even if multi-tasking was provided, it was of a primitive sort with very little practical application.

During the 1970s as the diversity of the small machines increased, it was realised that, for the first time, portability in operating system design was a

major issue, and several successful systems were produced out of this approach. One such operating system was TRIPOS [Richards79a], which forms the basis of the work described in the rest of this thesis.

At the same time as the development of portable, single user operating systems, there was an increase in the interest shown in high speed communication, and its application in local area networks. One of the networks built was the Cambridge Ring [Wilkes75, Hopper78], which provided a 10 megabit/second communication medium for a variety of heterogeneous computers. This provoked a great deal of work in designing and implementing a distributed working environment [Needham82], and as part of this exercise, TRIPOS was modified so that it would handle remote, virtual peripherals, accessed via the network [Knight82]. The distributed version of TRIPOS ran on a "processor bank" of Computer Automation LSI4/10 and LSI4/30 machines, and was used as the multi-tasking environment for the implementation of the Cambridge File Server [Dion80].

By the late 1970s and early 1980s, there had been sufficient advances in VLSI techniques that it was possible to have 16 and 32 bit micro-processors, for example the Motorola MC68000, the Zilog Z8000 and the Intel 8086 and 8088. More ambitious processors such as the Intel iAPX-432 were planned, but never came to fruition. At the time of writing, not only have DEC managed to implement a version of the PDP-11 in VLSI, but also the MicroVax, a version of the 32 bit VAX-11. With the improvement in the type of processor came the production of 64K and 256K dynamic memory chips at a fraction of their previous cost, making it possible to have a personal 32 bit workstation with several megabytes of memory at a cost of well under £10,000.

Apart from the obvious application of building new machines with larger amounts of memory, it was possible to develop high-resolution bit-mapped screens, which enabled much more information to be displayed than on a standard VDU. The technique of splitting the screen into several "windows", with different processes attached to different windows, enabled concurrency to be used in a convenient and simple manner. Also, the use of a "pointing" device (as well as the normal keyboard) meant that areas of the screen could be selected quickly and accurately, with menus being used to obtain different facilities. The pioneers of this type of personal computing were the Xerox Corporation, and descriptions of this type of computing can be found in [Thacker78, Redell80, Lauer81, XEROX82, XEROX84]. At the time of writing, even though computers with high-resolution screens are available, they are still relatively expensive compared to other systems, and as a result are rather in the minority.

With the exception of the Xerox systems described above, the development of software for the new machines tended to lag behind the hardware innovations, with two main approaches being adopted. There were those who treated the new machines as large micro-processors, and so "ported up" the small, simple operating systems such as CP/M [DigitalResearch82]. Alternatively, there were those who treated the new machines as small mainframes, and so "ported down" large and complicated operating systems such as UNIX [Ritchie74]. Some operating systems were designed from scratch, but these were mainly aimed at the business and hobby markets, and tended to fall into the first category.

1.1 Background

At the end of 1980, a local company was employed to design a new machine, based on the MC68000, which was to fulfil two functions. Firstly, it was to replace three rather aged NOVA computers, and take over from them the task of teaching assembly language programming to undergraduates. Secondly, it was intended as the second generation of machine to be installed in the processor bank, to augment the stock of LSI4s.

Originally, there were two basic types of system envisaged: one with 64K bytes of memory to be used by students, and one with 256K bytes of memory to be used by the rest of the research department. It soon became clear that this distinction was unnecessary, since effort would have to be exerted to support two different sets of system software, and in any case, the cost of 64K dynamic RAM chips had dropped to an insignificant part of the machine's price. It was decided therefore that all systems should have a minimum of 256K bytes, with the option of upgrading them to 512K bytes if necessary.

The first 68000 prototype was available in May 1981, and the first printed circuit board version was produced in the following October. At the time that the hardware development was being done, a small group of people were preparing the system software for the new machines. It was decided that TRIPOS was the operating system which should be run on these machines. At the time, this made great sense, since TRIPOS was designed to be portable, and as the 68000 processor itself was only in the prototype stage, there was as yet no alternative commercial product. TRIPOS was already running on the LSI4 machines, and as a result, a large amount of both system and application software was already available.

A macro assembler [Wilson81] and interactive debugger [Wilson82a] for the 68000 were written, and an undergraduate student produced a prototype

TRIPOS kernel as part of his final year's work. This was taken by M. Richards, and from it he produced a working version of the TRIPOS kernel, along with a BCPL code generator for the 68000.

With an assembler, a BCPL compiler and a kernel, the task of implementing the rest of TRIPOS proved to be fairly straightforward. After all, TRIPOS had been ported onto new machines many times, and the only difference between the 68000 and other TRIPOS implementations was the BCPL word size. Unfortunately, much of the ring handling software had word length dependencies (the ring word size up to then had been the same as the machine word size), and B. J. Knight had the job of removing the 16 bit dependencies.

One piece of work which did have to be done from scratch was the software for the 68000's ring interface. The LSI4 computers had an interface based on the 8X300 micro-processor [Gibbons80]. The 68000 machines had been designed with a 6809 based interface, and although the functionality of the two processors was roughly the same, the interface code itself had to be totally re-written. This was done by N. H. Garnett, and his initial work on the 6809 interface later turned into the *supermace* byte stream handler [Garnett83].

The debugging of the hardware, and the initial testing of the software was primarily done by myself and the original design engineer. By the end of 1981, we had a rack of three working 68000 processors, all running ring-based TRIPOS. By that time, the cost of memory had come down even further, and it was decided to upgrade all the machines to 512K bytes. All subsequent machines would have at least this much, and at the time of writing, the majority have been upgraded to 1M byte.

1.2 Motivation and aims

The purpose of the work described here was to make better use of the 68000 machines, by investigating how they were used and what could be done to improve performance. Out of the work done, it was hoped that certain design principles for the new machines would emerge, and that a set of guidelines could be drawn up for the use of future implementations.

The motivation for doing the work was that neither of the two types of operating system described earlier were capable of taking full advantage of the new hardware, and only by investigating different techniques would it be possible to use these machines efficiently. The small operating systems were inefficient, since they did not provide multi-tasking, and could not hope to make full use of the CPU and memory facilities available. The large operating systems were also

inefficient, not because they could not make full use of the hardware, but because they relied too much on fast discs for paging and swapping. The peripheral technology has not kept pace with the micro-processor advances in terms of speed, and the result is that these operating systems spend much of their time idle, waiting for disc I/O transfers to be completed.

The approach adopted here was to start with a simple operating system which did have multi-tasking, but did not require fast peripherals. The operating system in question—TRIPOS—also had the advantage that that it hàd been developed in Cambridge, and hence the sources were freely available. It was also written almost exclusively in a high level language, BCPL [Richards69], making it ideal for experimental modification.

1.3 Work done

The work done on TRIPOS falls neatly into two separate areas. Firstly, the user interface was investigated, and a new command language interpreter and command/programming language were designed and implemented. Secondly, different ways of improving the user environment and making use of the extra memory and CPU power were tried, and several are described along with a discussion of their relative merits.

Working on TRIPOS was fine for the development of user level software, but because of the problems of compatibility, it was never possible to perform useful experiments on the TRIPOS kernel itself. An opportunity did arise, however, in the form of a requirement for a light-weight multi-tasking kernel for small IBM/370 machines, and many of the potential modifications to the TRIPOS kernel were incorporated into this design instead.

1.4 Layout of thesis

Chapter 2 investigates the work done by others in the fields of single user operating system design and command programming languages.

The next two chapters set the scene for the rest of the work. Chapter 3 describes the relevant parts of the TRIPOS operating system, and in particular, its command language interpreter and the programs which form its primitive command language. Chapter 4 describes how this simple command environment was enhanced so as to improve the user interface, and the facilities which were added to enable the work on a new command and programming language to proceed.

Chapter 5 introduces the command and programming language REX, and gives examples of programs and command sequences written in the language. The important differences between the REX language and other programming languages are discussed, and some issues of the TRIPOS implementation of REX are described.

Chapter 6 investigates how the user environment of an operating system can be improved by providing extra facilities for doing specific tasks. Techniques to utilise large amounts of memory by keeping programs pre-loaded are discussed, along with a simple window handler interface to a full screen editor and two different facilities for directory management.

Chapter 7 investigates issues associated with the design of operating system kernels. Different techniques for operating system implementation, structuring, scheduling and memory allocation are discussed, and a set of guidelines for future operating system designers is drawn up. As an illustration of some of the ideas presented, the GMK multi-tasking kernel is described, and contrasted with its TRIPOS predecessor.

In conclusion, chapter 8 sums up the work done and discusses some of the results.

1.5 Extent of collaboration

Chapter 3 describes the original version of TRIPOS, and is included as an introduction to the rest of the thesis—none of the work described in it is my own. Chapter 4 describes the TRIPOS Shell which, although entirely my work, is based heavily on the design of the UNIX C Shell.

Some of the TRIPOS facilities described are the work of other people, and where this is the case, it is stated explicitly in the text. Particular mention should be made of the "PIPE:" and "CORE:" pseudo devices, which are both the work of M. F. Richardson, and the "WORCESTAR" full screen editor, which is the work of W. R. Stoye. I am also indebted to many of the members of the Systems Research Group, too numerous to mention individually, who have helped to produce the many utilities which comprise the TRIPOS environment. Without them, this work would not have been possible.

Apart from the exceptions mentioned above, the rest of the work described in this thesis is entirely my own, as are all the comments, discussions and conclusions.

2. Related Work

There are two main areas of research which are related to the work described in this thesis, and they are treated separately here. The first is the area of operating system design for single user computers, and the relationships between the requirements, the techniques used and the facilities provided. The second is the design of user command languages, and their relationships to more orthodox programming languages.

2.1 Operating system design

Although much work has been done on operating system design in the past, single user computing is a relatively new phenomenon, since only recently has it become financially viable to provide individuals with their own personal workstations. Initially, these came in the form of mini- and micro-computers, usually small, which supported simple operating systems, very similar in concept to those produced for the old batch mainframe systems. In the early 1970s, there was much research into the implementation of small operating systems in high level languages, and designing them so that they would be portable onto many different types of hardware.

As the cost of computers fell and the corresponding cost in manpower rose, it became clear that simple, portable single user operating systems were financially advantageous, and towards the end of the 1970s many such systems were developed. With the arrival of cheap memory and fast 16 or 32 bit micro-processors in the early 1980s, single user computing became even more attractive, but unlike their 8 bit predecessors, very little work was done on how to make full use of them.

Much of the work described in this thesis is based on the TRIPOS operating system [Richards79a], which was developed at Cambridge as a piece of research into portable operating system design. Many other systems were produced at about the same time as TRIPOS, and through looking at the different implementations, it is possible, with the benefit of hindsight, to draw conclusions as to how future single user operating systems should be designed.

At the same time as the work done on single user operating systems, there was a large increase in the number and variety of high bandwidth local area networks. The result has been that, more recently, research has tended to be in the area of distribution, with the underlying operating system design being rather

neglected. Although networking is a highly important area of computer science research, it is outside the scope of this thesis, and on the whole unrelated to the topics discussed here.

2.1.1 OS6

One of the first experiments in single user operating system design was OS6 [Stoy72], designed by J. E. Stoy and C. Strachey at the University of Oxford, and implemented initially on a CTL Modular One. OS6 introduced many new concepts which were to be adopted by the other systems which followed, and many of its ideas still have much relevance, even though the nature of computing has changed greatly in the past 15 years.

OS6 is a single user, single threaded operating system, and thus removes many of the complexities of more orthodox systems. Through the single user simplification, the necessity for accounting and protection have been removed, and the fact that OS6 is single threaded removes the problems of inter-process communication and process synchronisation. It was the designers' view that, before generalisations could be made regarding multi-user working or concurrency, it was necessary to solve the simple single user problem first. It is this approach which has enabled subsequent implementations to benefit from their results.

OS6 is written almost entirely in one high level language, BCPL [Richards69], since it was considered that doing so would ease implementation and increase portability. Stoy and Strachey chose a typeless language because they believed that the most important aspect of an operating system language should be control structure, with matters such as storage and object representation being left to the programmer. The portability of OS6 came from the fact that the BCPL source was compiled into a compact code for a virtual machine, which was then interpreted by a 250 line program written in Modular One assembly code. Because of the use of an interpreter, the resident part of the system was much less bulky than it would have been had it been compiled directly into machine code. As a result, much more of the operating system could afford to be written in a high level language. It also meant that many of the peculiarities of the underlying hardware could be hidden from the operating system, and even though interpretation reduced the effective machine speed by a factor of 15, this was considered a small price to pay for generality. Apart from the interpreter, the only parts of the system not written in BCPL were a few small procedures executing I/O instructions which could not be compiled.

OS6 has no special command language, with BCPL being used as the interface between the user and the program loader. Programs may call each other recursively, with the store containing the code of loaded programs being managed as a "last in, first out" stack. The local variables of the programs are also held on a stack, which is unwound whenever a program finishes or an error occurs. There is no distinction between what is operating system code and what is user code, and nothing in OS6 is inherently more privileged than anything else. There is no mechanism to protect the operating system from accidental over-writing (the problem of malicious over-writing being ignored in the context of the single user operation), but the authors claim that such problems are rare, and they attribute this fact to the banning from the system of assembly language programs.

One of the most important techniques to come out of the work on OS6 was the concept of "stream functions" enabling I/O streams to be handled in a device independent manner, and compound streams to be built for "on the fly" data transformation. The effect of an OS6 stream function is to take one I/O stream as an argument, and yield a different one as its result. Through this technique, OS6 has a standard "internal code" for the representation of characters, with the ASCII parity bit to be used to indicate underlined characters, and a special code to mean "4 spaces". The conversion between this code and the external representation is done in the individual device handlers. Although the use of an internal code is not especially important, the concept of event driven streams, with data processing being done at each level in the stream as the data passes, is extremely powerful and has been adopted by many later systems, including TRIPOS.

2.1.2 UNIX

In the early 1970s, mini-computers were still fairly expensive, and hence there was still the necessity to use them to provide time-sharing services. An operating system designed for this purpose was UNIX [Ritchie74], originally implemented on DEC PDP-7 and PDP-9 computers, with later versions running on PDP-11/40 and PDP-11/45 machines. UNIX is designed to bring a high degree of generality and power to a small, cheap ($40,000 in 1974) computer system, and provides many novel facilities which were not available in larger operating systems.

UNIX is implemented almost entirely in C [Kernighan78] (a derivative of BCPL), with only a few hundred lines being written in assembly code, either out of necessity or for efficiency. In order to ease the implementation and portability of UNIX, the C language was extended to make it more general and less machine

dependent. The major addition was the "TYPEDEF" construct, which enabled C data types to be parameterised.

The UNIX filing system is hierarchical, with mountable volumes grafted as directories in the filing system structure. In an attempt to rationalise access to files and devices, they both appear as entries in the filing system, with device handlers being marked as "special" files. Both are handled in exactly the same way, giving the user a high degree of uniformity, but meaning that it is impossible to access a raw device directly. In a similar way to files and devices, processes can be connected together by means of "pipes", with the output of one process being sent as the input of another. In this way, a program can take a stream of characters as its input, process them, and send another stream of characters as the output—the effect being that of a "filter". It is the UNIX philosophy that every program should concentrate on doing only one job well, and because of this, many complex operations (such as compilation) are implemented as a whole set of programs connected together by pipes.

One of the original design decisions of UNIX was that it should provide a convenient and powerful programming environment, which was clean and simple to use. This has been accomplished by providing a program called the "Shell" [Bourne78], which acts as the entire user interface to the operating system. The purpose of the Shell is to take a series of command lines and execute them—just as with any other command language interpreter. The Shell does rather more though, in that it holds the entire user environment, and allows the setting up of multiple processes with pipe connections between them. The syntax used is simple and concise, making it easy to build up complex sequences of commands and run background processes.

The authors attribute the success of UNIX to three main factors. Firstly, it was developed "in house" to aid the writing and testing of programs, and was designed from the start to be a highly interactive system. Secondly, because the machines on which it was implemented were small, there were fairly severe size constraints on the system and its resident software. This constraint encouraged conciseness and economy, and resulted in an elegant design. Thirdly, very soon after its initial conception, the operating system was fully self-supporting. This meant that the designers of the system were themselves forced to use it, and they quickly became aware of any functional deficiencies. As all the program sources were on-line, it was possible for modifications to be made quickly and easily, with the result that the system continued to evolve on demand.

There is a certain irony about UNIX in that, even though it was capable of supporting multiple users through time-sharing, it was originally built as a simple

single user programming environment for mini-computers. Through its portability and popularity, it has been moved onto many different types of machine, and several versions now exist. The University of California at Berkeley has produced a much extended version of UNIX [Berkeley81], incorporating virtual memory and inter-process communication—items which had been omitted from the original design. Some of the machines on which UNIX has been implemented (such as the DEC VAX-11) are too expensive for single user operation, and so support many users in a time-sharing environment. The irony comes from the fact that, as micro-computers have become cheaper and more powerful, it has been possible to port UNIX onto them, but these implementations revert to supporting just a single user. Certain commercial versions of UNIX are now available: XENIX [Microsoft82] which has attempted to standardise the operating system, CROMIX [Cromemco82] which runs on Z80 and 68000 machines, and more recently, ULTRIX [DEC84] (a derivative of the Berkeley version of UNIX) for DEC MicroVax machines.

In his retrospective look at UNIX [Ritchie78], D. M. Ritchie examines the reasons for its success, and highlights some of its shortcomings. He believes that it is simple enough to be comprehended by most people, and powerful enough to meet most of its users' requirements. This is particularly true because of the sheer number of utilities and applications programs now available to run under UNIX. He also believes that the user interface is clean and simple, but perhaps rather cryptic and difficult to learn by newcomers to the system. The shortcomings of UNIX arise from the fact that there is no inter-process communication mechanism (making multi-event processes impossible to implement), and since processes cannot be locked in real memory, a guaranteed real-time response is difficult to achieve. This is worsened by the fact that processes cannot access devices directly, and that although I/O appears synchronous to the user, the system performs read-ahead and write-behind on his behalf.

2.1.3 SOLO

In the mid 1970s, there was an interest in adding concurrency primitives to orthodox programming languages, and such a language (concurrent PASCAL) was used by P. Brinch Hansen to implement SOLO [BrinchHansen76]. In many ways, SOLO is very similar to OS6 in that it was intended for single user operation, with no special facilities for accounting and protection. SOLO was implemented originally on a PDP-11/45, and like OS6, it is written almost entirely in a high level language (concurrent and sequential PASCAL). The only

assembly language in the entire system is a 4K word kernel, which represents only 4% of the whole system.

In order to ease implementation and increase portability, SOLO runs on a virtual machine which removes low level programming features such as registers, addresses, interrupts and so on. Through the use of a virtual machine, and the protection which arises from the PASCAL compile time checking mechanism, the entire SOLO system (100K words of code, including two compilers) was implemented by two people in less than a year.

SOLO has a fixed number of processes, with input, output and the user job being executed concurrently. All other programs in the system are sequential. The system itself has no built-in I/O drivers for different devices, but simply provides a mechanism for data to be produced and consumed by sequential PASCAL programs loaded from disc. In this way, the user can add more devices and perform complex operations on I/O data (in a similar way to stream functions), without having to modify the underlying operating system.

The user interface in SOLO is very similar in concept to that of OS6, in that the command language is simply an extension of the implementation language (in this case PASCAL rather than BCPL), and the user can enter programs which themselves call each other recursively, with arbitrary nesting. Unlike OS6, programs which are not active are swapped out to disc, with only the data of these programs being kept on a stack in memory.

The SOLO filing system is held on a slow disc with removable packs. Each user has his own pack which contains his private files, and because of this the authors did not think it necessary to provide a hierarchical filing system. Each pack contains a catalogue of the files on it, and each file on the disc has an associated data type. All programs check the type of their input files before accessing them, and associate types with files when writing to them. The possible types are *scratch* for compiler work files, *ascii* for normal text files, with *seqcode* and *concode* for compiled sequential and concurrent PASCAL respectively. Files can be protected against accidental over-writing, but since each user has his own private disc pack, protection against malicious alteration or deletion is not necessary.

In summing up his system, Brinch Hansen attributes the success of SOLO to the choice of implementation language. The whole development cost was under 2 man-years, and he estimates an equivalent cost of 20 to 30 man-years if the system had been implemented in assembly code. He also believes that, as an added benefit, writing in a high level structured language leads to a system which is much more understandable and easy to maintain than would have been

possible before.

2.1.4 MUSS

The MUSS operating system [Frank79] produced at the University of Manchester is rather different to the others discussed here, in that it was never specifically designed for single user operation. Instead, it was implemented simultaneously on six different types of machine, ranging from the MU5 main user job processor, through an ICL 1905E front end processor for batch work, to a DEC PDP-11/10 front end processor for interactive work. Rather than having a single, portable, general purpose operating system for all these applications, several compatible systems were designed to cover a wide range of applications. The systems were built so that the processes running in one machine could cooperate with the processes running in another, and so the whole system structure was standardised over the different machines.

In order to achieve this goal, all operating system modules run in different virtual machines, and communicate with one another by sending and receiving messages. On each machine, a small highly privileged kernel is responsible for the multi-programming of the different virtual machines. The designers chose messages rather than shared data areas as the means of inter-process communication for two main reasons. Firstly, it preserves the independence of modules, with virtual machines only interacting through the exchange of messages. Secondly, since there is separation between virtual machines, it is a simple extension to allow message passing between different physical machines.

The most relevant part of the MUSS project is the design and operation of the operating system kernel. Its job is simply to schedule a small, fixed number of system processes, which perform more complicated scheduling, memory management, and so on. The main design aim for the kernel was simplicity, because of the desire for very fast process switching. The technique used is very simple and rather elegant. If the number of processes is fixed and small, then each process can be represented by a single bit in a machine word. It is therefore possible, rather than maintaining a queue of runnable processes, waiting processes and so on, to have single word "sets" of processes for this purpose (where each process is represented by a single bit) so that adding or removing a process from a set is simply a logical *or* or *and* operation.

The main use of MUSS was to provide a powerful computing environment for main users on the central University mainframe, but it is an impressive design achievement for the system to run on such a machine, and at the same time be fully compatible with one running on a 16K word PDP-11. The technique

vindicates the view expressed later in this thesis that for an operating system to be truly general purpose, it must be designed as a series of separate layers, with each layer only being included if required.

2.1.5 THOTH

At about the same time that TRIPOS was being designed and implemented at Cambridge, the other operating system which most resembles it, THOTH [Cheriton79], was being being produced at the University of Waterloo. The two main design aims of THOTH were portability and real-time performance—two of the targets of TRIPOS—but unlike TRIPOS, THOTH is designed to support many users in separate virtual address spaces.

THOTH has been implemented on the Texas Instruments 990 and Data General NOVA mini-computers—machines with totally different architectures. The portability is achieved by designing the operating system for an ideal "THOTH machine", and by writing most of the software in a high level language, in this case EH [Braga76], another derivative of BCPL. Like BCPL and C, EH allows the programmer to write in a structured and portable manner, at the same time avoiding acting as a barrier between him and the hardware. In order to ease implementation, the EH language supports a "`twit`" statement for the in-line inclusion of assembly code.

THOTH aims for an efficient inter-process communication mechanism, and attempts to make it easy to solve problems using small concurrent processes. Normally, each process runs in a separate address space, but it is possible to set up *teams* of processes which share a common address space, and hence can share data. Inter-process communication is by means of fixed length 8 word messages, and there are primitives to send, receive and reply to messages. Because processes are held in separate address spaces, all messages must be copied—something which enforces simple semantics since everything must be passed by value, but which can cause a great deal of inefficiency if copying occurs too often. One feature of THOTH is that, after a message has been sent, the sending process is blocked until the message has been received by the recipient process. This means that for true, multi-event programming, a team of several processes must be used in order to ensure continuity of service.

THOTH uses a simple priority based scheduling algorithm, where high priority (system) processes can pre-empt low priority (user) processes, thus ensuring speedy response to real-time stimuli. Processes of equal priority are handled in a "first come, first served" manner, but no time slicing is done. Even so, THOTH is used to support multiple users, primarily in the teaching of real-

time operating system principles to undergraduate students.

I/O in THOTH attempts, like UNIX, to present a uniform interface between the user and peripheral devices. All streams are represented by general "file control blocks", which hold the current byte position in the file. There are functions provided to access the stream in a random way (if applicable) by investigating and setting the byte position value. Although the mechanism is clean and simple (being very similar to that provided by UNIX), it does not have the expandability of the stream function method, and is not as general.

In their conclusion, the designers of THOTH believe that the project was successful, as it demonstrated that it was possible to write a portable operating system for a particular class of machines. The implementation (like TRIPOS) supported only one language, making portability of other programs onto THOTH a more difficult proposition. The system has been used for undergraduate teaching and as a research tool, and is also in use industrially for real-time control applications.

2.1.6 Lampson and Sproull

As a piece of research into single user operating system structure, B. W. Lampson and R. F. Sproull produced an "open" system [Lampson79]. Rather than protecting each user process in its own virtual machine where the realities of the outside world could be hidden, the principle was that the operating system would be very open, with no well defined boundaries between it and its user programs. All parts of the system would be accessible to the user, and available for modification—absolutely nothing would be hidden from him.

The operating system is written in BCPL, and runs on a Xerox ALTO computer [Thacker78]. There are just two processes—the keyboard process which handles characters typed by the user, and another process which handles everything else. There is no scheduler, and no synchronisation is necessary, since the keyboard is purely interrupt driven. It is more reasonable to treat this system as a run time library providing access to useful abstract objects, such as I/O streams, files, discs and so on. Each of the abstractions is represented as a single BCPL word.

Because there is only one active process, there are no communication primitives, and so if two programs wish to cooperate, they must do so by using disc as the intermediate medium. There is a mechanism for saving and restoring the processor state from a disc file, and through this it is possible to switch between different contexts in the single process. This is effectively a coroutine switch, and programs which take over the processor in this way are called

"juntas".

The authors attempted to standardise access to the disc filing system and to the local area network. This facility proved convenient for the user, but was difficult to maintain because much of the functionality was duplicated in different operating system modules. They do not recommend this type of standardisation in future systems. The "open" approach, although interesting, makes it difficult to change the internal representation of objects (such as files), since there is no way of doing this without affecting assumptions made by the user. This makes this method less general purpose, and because the system hides none of the hardware from the user, it also makes it less portable.

2.1.7 PILOT

By the early 1980s, the cost of hardware had come down enough to increase the power of the machine which could be given to a single user. One of the companies to pioneer this approach to computing was Xerox, and a product of their PARC research laboratory was the PILOT operating system [Redell80].

PILOT provides a single language environment for high level programming on a powerful single user computer. PILOT is implemented entirely in MESA [Mitchell79], and the two are very closely coupled. In many ways, it is convenient to think of PILOT as being a multi-programming environment for MESA, in very much the same way as TRIPOS is for BCPL. Not only is the personal computer used extremely powerful, but it is also micro-coded specially for the MESA language, and since concurrency is built into the language, absolutely none of the system need be written in anything other than MESA.

One of the facilities provided by MESA is module interface checking, making it easy to split the implementation into many separate modules, with all the consistency checking being done by the MESA compiler. The design team attribute the success of PILOT to this fact, since the 24,000 lines of MESA which comprise the system were split into around 160 modules, with each module being roughly 150 lines. Each module could be defined absolutely in terms of its interface, and then implemented by any of the 8 strong team independently. Since PILOT does not rely on hardware mechanisms for protection, system integrity (so called "defensive protection") is provided by MESA's type and interface checking.

The PILOT filing system is large and flat, with the *file* and *volume* modules providing the basic structures for disc storage. File names are 64 bit unique identifiers or *uids*, which are guaranteed unique in both space and time. This is essential, since removable volumes are the standard way of transporting files from

one PILOT machine to another, and any chance of name clashes would make this impossible. Each file has several attributes associated with it, on the assumption that a higher level module will provide more complicated features, such as naming, date stamping and so on. One of the attributes is an uninterpreted 16 bit "type" value, which aids in recovery of the filing system after a disc failure.

PILOT has a simple linear virtual address space of 2^{32} 16 bit words, and all processes run in this address space. As mentioned earlier, the protection is provided by means of MESA type checking. There is also a "world swap" mechanism to enable the entire virtual memory of a machine to be changed, in order to bring in a totally new environment. This is used by the PILOT debugger "CoPilot" whenever a system error occurs.

I/O streams under PILOT are very similar to those in OS6 and TRIPOS, in that it is possible to build compound streams from individual stream components, with each component performing "on the fly" transformations on the data flowing though them. Combining this facility and the file "type" attribute, it is possible to implement a fully typed filing system, supporting many different file structures.

As with other concurrent languages, inter-process communication in MESA is through use of shared memory, and facilities are provided such as *fork* (to create a new process), monitors, condition variables, coroutines and so on, as well as simple recursive procedure calls. There is also a *signal* operation—a special type of procedure call used in exception handling. MESA also supports communication between loosely coupled processes (in other words, those on different machines), through a family of packet communication protocols called Internet [XEROX81a].

H. C. Lauer [Lauer81] in his review of the PILOT operating system, comments on some of the design decisions and their relative success. The virtual memory system was implemented to take account of discs with slow access times, with heavy use being made of concurrency and queueing. He says that, given the actual speed of the discs used, it would have been better and simpler to treat them as synchronous devices. He also comments on the fact that the PILOT stream mechanism was not used much for communication between machines, since even with loosely coupled processes, it is easier in MESA to use a procedure call interface (through the Courier [XEROX81b] remote procedure call mechanism).

2.1.8 LILITH

With the advent of the Am2901 bit-slice processor, it has become more convenient to construct micro-programmable hardware which is tailored to a specific task. One such example of this is the LILITH computer [Wirth81] designed and built at ETH Zürich. The LILITH is a personal computer whose architecture is tailored to the language MODULA-2 [Wirth80], with a memory of 128K 16 bit words (half of which is used by a high resolution display), a disc, a local area network, a keyboard and a mouse. The processor is micro-programmed to interpret "M-code", the intermediate code of the MODULA-2 compiler.

All the LILITH software is written in MODULA-2, including the operating system "MEDOS". The reason for this is to take advantage of MODULA-2's module interface checking, so that a relatively secure system can be built without the requirement of hardware memory protection. The approach followed is very similar to that of PILOT, with the operating system forming a powerful run time environment for a single type-checked language.

For simplicity, LILITH is built with a single processor, so as to remove the problems of processor synchronisation. The designers also believe that concurrency in processes plays a minor role in operating system implementation, and so MEDOS uses coroutines exclusively. Transfer of control between coroutine processes is implied in statements which send signals or wait to receive "signals", where a signal is represented as a MODULA-2 data type. There is a data type "PROCESS", with primitives to create a process given an entry point, and transfer control to a process. Using these mechanisms, MEDOS schedules its processes in a simple "round robin" manner. Interrupts are handled by having device driver coroutines to which control is given whenever a device interrupt occurs.

In their conclusions, the LILITH team consider that their approach to personal computing has been successful, in that the specialised hardware enables systems programs to be written in a high level language, with little problems of code compactness or overall efficiency. The LILITH machines present a pleasant interface to their users and permit a high degree of interaction, without the restrictions of mainframe computing. They believe that their choice of language, as with PILOT, has enabled them to build a large system in a simple and clean manner, with much of the burden being removed by the MODULA-2 interface checking facility. They also believe that, through using a single high level language, it is possible to design a computer architecture without regard to its suitability for assembly language programming. The resulting computer can be geared towards a language, and take advantage of such factors as variable length

addresses in order to provide a high density of code.

2.1.9 TRIX

As local area networks became more common, so did the tendency to design operating systems around them, concentrating rather more on the distribution aspects than on the operating system design *per se*. One such approach was taken at the MIT [Ward79], and encompassed both the hardware and software design for a distributed system.

The basic items of hardware used are "NU" computers, connected together via the "NUBUS". Each computer is built out of several standard components (such as processors, memory, keyboard, screen and so on), with the number and type of components used depending on the nature of the particular application and the facilities required. Several CPU cards are available, including those for the Motorola MC68000, Zilog Z8001 and the "RHO" processor, a 32 bit Am2901 based micro-codable machine. Memory comes in units of 128K bytes with 16K DRAM chips, and 512K bytes with 64K DRAM chips. There is also an interface to an 8 megabit/second local area network.

The software work resulted in the TRIX operating system [Ward80], which was designed to allow transparent communication over the local area network. In order to achieve this goal, TRIX removes abstractions such as files, devices and so on, and replaces them by "streams" which connect different processes. Streams are simply full-duplex communication paths between processes, down which messages are sent. The important factor is that the semantics of the communication are associated with the stream, not with the mechanism at the end of the stream, meaning that it is possible to implement simple operations (such as "read", "write" and so on) in a device independent way. On the whole, it is impossible to tell whether a stream is connected to a device, another process, or in fact to another machine via a network handler.

In order to achieve efficiency, the TRIX system handles static objects such as files by means of "system processes", which remove some of the overhead of the more general stream mechanism. The user cannot tell that this is taking place though, since the same set of standard opcodes are used by the system processes as are used by everything else. Each stream has a name associated with it, and it is possible to open a stream by quoting its name. There are certain specialised "directory" processes, which allow the association of names with streams, and they provide a generalised naming convention which can be used to specify a "path" through a series of directory nodes, just like directories in a conventional filing system. Using the stream representation, as with stream functions, it is

possible to perform data transformations "on the fly", for example, the user should not be able to tell whether his files are each separate and being handled by separate processes, or stored in a compact archive format and handled by only one process. This approach has the effect of making network access totally transparent, with access to remote services and files appearing no different to the user than simple access to local versions.

The stream mechanism is simple and elegant, but suffers from four major drawbacks. Firstly, the efficiency of simple objects sometimes suffers, since they have to be represented in a complicated way. This can be avoided to some extent by having system processes, but the real time delay in passing a message through a chain of processes may be quite large. Secondly, since the filing system is represented by a set of processes and streams (which are transient objects), problems arise when the machine crashes, in that it is impossible to restore the structure to exactly what it was before the crash. Thirdly, since streams are fixed objects, it is difficult to handle a filing system with removable disc packs. Fourthly, since streams can be set up between arbitrary processes, there is nothing to stop cyclic structures. These have the same problem as with memory management, since a simple "use count" method is not enough to ensure that streams are freed when a connection is broken, and some type of garbage collection is required as well.

2.1.10 SPICE

Another project similar to that at MIT is the SPICE system [CMU80] being developed at Carnegie-Mellon University. SPICE (Scientific Personal Integrated Computing Environment) is intended to provide a high performance network of over 100 personal computers, with shared facilities such as printers and filing systems. SPICE is an ambitious piece of research, with operating system design, resource distribution and programming environments being just three of the areas covered. Rather than designing specialised hardware like the MIT project, SPICE uses commercially available computer and network systems. Most of the work has been done on DEC VAX-11 and Three Rivers Corporation PERQ machines, with ETHERNET being the communication medium.

The piece of work most relevant to this thesis which came out of the SPICE project is the ACCENT operating system kernel [Rashid81], which has been designed from the start for network operation. One of the major design aims of the ACCENT team was to build an operating system which would access the network transparently, and allow decomposition of modules so as to allow their distribution between different processes on the same machine, or onto different

processors on a network. Multiple languages are supported by ACCENT, with each program being separated into its own virtual address space for protection.

The ACCENT kernel provides two types of protection. Firstly, there is the address space protection mentioned above, which ensures that no process can affect any other, except through use of the inter-process communication facility. Secondly, access protection is included in the form of capabilities, which prevent unauthorised communication between processes. The authors contrast this to the PILOT approach, where all processes run in a single virtual address space, with protection being obtained through the use of a single, heavily checked programming language.

The whole ACCENT system is communication oriented, with simple messages being used both to communicate between processes and different computers. The messages are structured in such a way that intermediary processes, for example debuggers, protocol converters or network communication servers, can intercept, modify and then retransmit messages in a transparent manner. Because all processes exist in different virtual address spaces or on different machines, messages are always copied thus simplifying the communication semantics. This approach is very similar to that adopted by THOTH, but ACCENT attempts to optimise the copying of messages from one address space to another by simply placing data pages into the address map of the receiving process.

One interesting feature of ACCENT is that messages are not sent to processes but to "ports" (mailboxes), and in order to send a message to a port, a capability for that port must be held. The "send" capability can be owned by many processes, and passed to a child process by a parent when it is created. Only one "receive" capability for a port can exist at any one time, and this capability can be passed between processes which provide the same service. Associated with each port is a "backlog" value, which defines the maximum length of the message queue which can build up for that port. Once the backlog threshold is reached, sending processes are suspended until their message can be placed in the queue properly. If a CPU bound process does not wish to block waiting for a reply to a message, then it can use the "pseudo interrupt" facility, in which a software interrupt is generated by the kernel whenever a message arrives on a particular port.

The ACCENT kernel has 16 different priority levels, and performs time slicing between processes at the same priority level. The concept of "aging" is used to ensure some degree of fairness in scheduling. The kernel attempts to provide a high speed context switch, by allowing each language to have support in micro-code for the saving and restoring of its state. In most cases, the cost of

a context switch is only twice that of a normal language procedure call.

In their conclusions, the authors believe that the ACCENT kernel is well designed, and provides a clean and efficient environment in which to perform research into program distribution. They defend their use of messages and independent address spaces by using machines for which micro-code support for these functions can be provided. They believe that the mechanisms they have chosen give a much more secure and general environment than equivalent systems such as PILOT, and allow for the introduction of powerful debugging and monitoring facilities, as well as virtually total network transparency.

2.1.11 MAYFLOWER

The MAYFLOWER operating system [Hamilton84] came out of a research project at Cambridge University, investigating aspects of distribution and remote procedure call mechanisms. Of the systems described so far, the closest in philosophy to MAYFLOWER is PILOT, in that they are both written in a single concurrent language, but in this case, CLU [Liskov81] rather than MESA. The extensions to CLU to support concurrency were implemented at Cambridge by K. G. Hamilton. Unlike PILOT, MAYFLOWER will support other programming languages, since its interface is via a language independent "trap" mechanism, but so far this has not been done.

The MAYFLOWER kernel was originally implemented on a Motorola MC68000 without memory mapping, and its primary aim was to provide a clean and efficient programming environment for CLU. The kernel separates processes into domains, but allows groups of processes to be run as part of the same domain, in very much the same way as THOTH's teams. Unlike THOTH though, there is no separation into different address spaces, and all the protection is through the type checking mechanism of CLU. There are two simple synchronisation primitives which can be used to communicate between different processes running in the same domain. Each process can run at one of eight priority levels, with time slicing being performed between processes at the same priority level.

The main thrust of the MAYFLOWER work is to build a distributed system around different processes all running the MAYFLOWER kernel, with remote procedure call being used to communicate between machines. It is currently being used to implement a resource management system, and distributed compilation and debugging systems.

2.1.12 Commercial operating systems

Over the past few years, commercial operating systems for the new generation of personal computers have come in two distinct sorts: UNIX and everything else. Because of the portability of the UNIX system, and the increasing number of software packages available for it, more and more implementations have appeared on many different types of hardware. Examples are CROMIX, XENIX and ULTRIX, all of which are based on UNIX, or are UNIX compatible.

The other types of operating system available commercially tend to be simple and single threaded, based on those for the 8 bit microcomputers of the mid 1970s. An obvious example is CP/M [DigitalResearch82], originally implemented on the Intel 8080, with more recent versions for the Intel 8086 (CP/M-86) and Motorola MC68000 (CP/M-68K). Another example is MS-DOS [Microsoft83], which has been produced for the IBM PC computer, running on the Intel 8088. The only multi-tasking performed is the spooling of printer output, and apart from that, the facilities provided are very simple. An attempt to produce something more ambitious came with the Sinclair QL in the form of QDOS [SinclairResearch84], which supports screen windowing, with a different task running in each window.

2.2 Command languages

In the past when operating systems have been designed, the emphasis has been on how many users they can support, and how much work they can perform. This attitude arose from the time when only batch access to computers was possible, and machines were so expensive that their utilisation had to be maximised. When time sharing systems became available, the user interface was rather neglected, with users having to type cryptic incantations in order to get their programs to execute.

One step on from typing commands directly at a terminal is the ability to write *command sequences*, in other words, files containing commands which would have been typed at the terminal. Such a mechanism could be used for running a series of commands repeatedly, or if parameter substitution were possible, then for providing templates to define the execution of frequently used sets of commands. For such a language to be properly usable though, it must be possible to test for the success (or failure) of commands, and to be able to execute commands conditionally. It should also provide some sort of looping construct, and perhaps simple variables.

The problem is that most systems have evolved as described above, with facilities being added when it seemed appropriate. The result is that, on the whole, command languages are clumsy and awkward, usually with an archaic or cryptic syntax, making sequences written in them totally unreadable to anyone other than the original author. Surprisingly, very little has been done to improve this area. New command languages have been produced, but like their predecessors, the facility has tended to be built into the command language interpreter, and even though the results are more powerful, they are just as clumsy.

Some people have attempted to combine the idea of an interpretive programming language with an interactive command language, in order to clean up the area of command syntax and program execution. The approach has proved fruitful, in that the user of such a system is faced with learning only one language for both his computation and command execution. The work referenced here is all by people who have treated command languages as programming languages, but with various different types of result.

2.2.1 Stephenson's EXEC

One of the first command languages to be designed with programming in mind was IBM's EXEC program for the Conversational Monitor System [IBM72], and the one based on it by C. J. Stephenson [Stephenson73]. In his view, command control languages are generally rather primitive and awkward, with the language interpreter being an integral part of shared computer systems. The advent of virtual machines each supporting a single user has meant that experimentation in the command control area is now possible.

Stephenson describes the traditional view of a command line, where the first word on the line is taken as the name of a command, and the remainder of the line is then passed to the command as a parameter list. He stresses the importance of the recursive view of command environments, where commands can be executed from within the editor for example. In his view, a command control language is not a true programming language, since it lacks the notation of evaluation, and in general it lacks variables and the ability to execute sequences of commands conditionally, depending on the result of some evaluation.

He then lists the three main design decisions taken regarding the command language. Firstly, it is a special purpose language, designed for the controlling of commands only. The only data object is a character string, and the interpreter handles only a limited type of expression evaluation. Only integer arithmetic is supported. Secondly, the language is interpretive, since compilation would have

been difficult, and at best partial. Thirdly, commands written in the command control language are written exactly as if they had been typed at the terminal. There is no program prologue or epilogue, and control statements are distinguished syntactically from data by being prefixed by the "&" character.

Simple variable assignment is provided, with all values being held textually. There is a conditional clause in the form "&IF ... &ELSE", and a looping construct in the form of "&DO ... &WHILE". Labels are provided by prefixing a name with the "-" character, and may be jumped to using "&GOTO". Primitive terminal I/O is provided through "&READ" and "&PRINT". Terminal input is handled in the form of a buffer, which also acts as a LIFO stack. In this way, command and data records can be stacked in the terminal buffer, so that they can be read by the command interpreter or a user program. There are functions provided to stack and unstack terminal records, and to rotate the stack. Simple string manipulation functions such as "&SUBSTR OF" and "&CONCAT OF" are provided, along with relational operators for comparing strings. Arguments are passed from the command line, and are available as the variables "&1" onwards. The interface to the command environment is provided through functions like "&EXIT" to abandon a program, and variables such as "&RETCODE", which holds the return code from the most recently executed command.

In conclusion, Stephenson challenges the decision to use a stack to hold terminal records, since it would have been more natural to use a FIFO queue. The only reservation he has about using a queue is that the wrong effect is obtained if a queue is not empty before further records are added. He therefore suggests that both a LIFO and FIFO facility should be provided, and that the queue should be flushed after the execution of each command. He also says that the command language described is fine for interactive working, but suggests that more powerful facilities (such as macro processing) would be useful if the language were to be used for anything more complicated than simple command sequences.

2.2.2 KRONOS and MULTICS

Although many of the command languages for large operating systems were primarily intended only for program control, some do have facilities found in real programming languages, and these features can sometimes be exploited. A. W. Colijn [Colijn76, Colijn81] has experimented with KRONOS and MULTICS, encoding the classic problems of "Ackermann's function" and "Towers of Hanoi" in the two different command languages. Through this, he investigates the facilities provided by the languages, and comments on the restrictions which they

impose.

The KRONOS command language, KCL, has features taken from high level languages such as FORTRAN and ALGOL60, which makes it a programming language in its own right. It allows arbitrary integer arithmetic, and provides simple variable assignment through "SET" statements. Only three variables are allowed—the pseudo registers "R1" "R2" and "R3". Conditional execution is possible through the "IF" statement (similar to the logical IF statement in FORTRAN), and output can be sent to the terminal by means of the "DISPLAY" statement. There is also a "CALL" statement, enabling another KCL command file to be executed recursively. The overall result is something which looks remarkably like FORTRAN, but has the restriction of just three variables and (like FORTRAN) no string manipulation primitives.

The MULTICS command language is different from KCL, and much more closely linked to the CMS EXEC language described earlier. MULTICS supports two types of command: loaded commands and special control commands, whose names are prefixed by the "&" character. Variables are supported, and can be assigned using the "value$set" statement. Normally, objects are held in string representation, but are converted to numeric values when necessary. Expression evaluation is possible using LISP like prefix notation, for example "[&plus A 1]". Conditional execution is provided by means of single line "&if ... &then ... &else" statements, with iteration possible through recursive entry of the same command file. Simple console output is available through "&print".

Although Colijn accepts that the applications chosen are rather frivolous, he does believe that the fact that what he has done is possible implies that command control languages are actually more powerful than most people assume. He says that command languages should not be dismissed for programming applications, since it is often more efficient to use an interpretive command language than to go through the steps of compilation and linking with a more traditional programming language.

2.2.3 Programmer's Workbench and the C Shell

Rather than designing a separate command execution language, another possibility is to extend the standard command language interpreter so that it incorporates programming language features. This approach was taken by the people working in UNIX, and two examples exist in the the form of the Programmer's Workbench Command Language [Mashey76] and the C Shell [Joy80]. Both take advantage of the original UNIX Shell, which provides a clean and well defined interface to the operating system execution primitives. The

work in both cases involves an extension of the Shell syntax to encompass programming language features, to enable simple programs to be constructed, as well as complex command sequences.

In his description of the Programmer's Workbench (PWB), J. R. Mashey describes the modifications made to the standard UNIX Shell, with the result being the PWB Command Language, or PWB CL. He investigated nearly 2,000 CL procedures in order to determine frequency and type of use, and the facilities most used by different applications. He lists all the important facilities which a CL should have.

It must be easy to use interactively, and should emphasize simplicity and avoid the need for redundant typing. It must be convenient to use CL both as an on-line command language, and as a programming language, and it must be possible for CL procedures and compiled programs to be used interchangeably, with the user being unaware which type he is executing. CL procedures should be easy to create and simple to maintain, and it must be possible for many users to share libraries of these procedures. The language should provide simple arithmetic, pattern matching and string manipulation facilities, with the emphasis being on characters rather than numbers.

Mashey also believes that the CL interpreter should not be viewed as part of the operating system, but as a user program which communicates with it. In this way, it is possible for users with different tastes or requirements to tailor the CL to their own liking if they want. His view is that, although many different versions will appear initially, there will be a tendency towards the "survival of the fittest", and only those with useful modifications will actually be adopted by the user community.

The PWB Shell provides 26 string variables, "`$a`" to "`$z`", of which the first half are guaranteed to be initialised to the null string. The rest are initialised to special values, for example "`$r`" holds the current return code. Arguments are passed to CL procedures as the variables "`$1`" to "`$9`", with "`$0`" being the name of the procedure. The "`shift`" command is provided for accessing argument 10 onwards. Assignment is possible through the "`= a b`" construct, where the value "`b`" is assigned to the variable "`$a`". Variables can be set to the results of loaded commands by "piping" the output into a variable, for example "`date | = d`". Loops are possible using the "`goto`" statement, and labels are introduced by the "`:`" character. Simple operators such as "`=`" are provided for comparing strings, but a different set of operators must be used when comparing numbers ("`-eq`" instead of "`=`" for instance), otherwise the comparison would be lexicographic rather than numeric. Commands can be issued conditionally through the "`if` ...

then ... else ... endif" construct, and a "**switch ... endsw**" statement (similar to that in C) is provided.

In his assessment of the PWB CL, Mashey suggests a series of things which should be done to improve the command language. Firstly, some sort of looping construct (for example "**while**") should be added. Secondly, the Shell should be modified so that the majority of the control code is resident, to make execution of CL programs much faster. Thirdly, the syntax should be tidied up, as it has many ugly aspects. Finally, the argument passing should be changed so that the decoding of parameters does not require shifting within a loop.

The C Shell was produced by W. Joy at the University of California at Berkeley, and is part of the Berkeley distribution of UNIX. Joy has attempted to copy as closely as possible the syntax of the C language, at the same time adding facilities such as a command history and command line editing. The C Shell attempts to provide an entire programming environment, with directives for job control, scheduling, command execution and simple programming.

C Shell variables have names prefixed with the "**$**" character, and may be up to 20 characters in length. Arguments passed from the Shell are available as the variables "**$1**" onwards, with "**$0**" being the name of the Shell procedure. Unlike the PWB CL, the C Shell syntax is such that numbers of greater than one digit can be written without ambiguity, and so there is no need for a "**shift**" command. Within a C Shell program, lines can be read from the terminal by using "**$<**" in the same way as a simple variable.

There is a series of built-in commands which evaluate arithmetic expressions, and the operators used are equivalent to those in the C language. There is a conditional statement "**if ... then ... else ... endif**", and a C like "switch" statement of the form "**switch ... endsw**". There is a simple looping construct "**foreach**", which iterates for each "word" in a string. More general looping is available through the use of labels (a label being a name followed by a "**:**" character), with the "goto" statement providing transfer of control.

In many ways the PWB CL and C Shell are remarkably similar in their outlook, with each taking the same program as their starting point, and producing very much the same results. Both suffer from the fact that the syntax is ugly and difficult to understand, and although the languages are fine for writing command sequences and driving programs for sets of commands, they are neither general nor efficient enough for true programming.

2.2.4 BASIC and LISP

Another approach to the problem of command languages is to use a programming language which already exists, and add control features to it. The obvious example is BASIC [Kurtz78] (Beginner's All-purpose Symbolic Instruction Code), a language originally designed in 1963 by T. E. Kurtz and J. G. Kemeny at Dartmouth College. The main purpose of the language was, as the name implies, to teach simple programming to people who were new to computers. BASIC is an interpretive language with variables, conditional statements, FOR loops and GOTOs, and provides simple terminal and file I/O. Because of its small size, BASIC has been implemented on many microcomputers, and on these machines it forms not only the programming language, but the command language as well. This is analogous to the OS6 and SOLO operating systems described earlier, since they also used a simplified form of their programming language as the user interface.

As a more serious suggestion, J. Levine [Levine80] asks why a functional language such as LISP should not be used as a command language. He looks at older command languages, such as IBM's JCL for catalogued procedures, and at the UNIX Shell and CMS EXEC languages, examining the features they have in common. He believes that treating single commands as tools (such as editors to perform text processing and data manipulation) means that very powerful compound utilities can be built up, with the driving program being written in a command language.

Levine thinks that command languages such as the Shell and EXEC are good for certain applications, but he believes that in many ways they are totally deficient. The facilities for handling data structures are primitive, with only strings and integers normally being supported. I/O is usually of a rudimentary sort, and most importantly, programs written in such languages tend to be ugly and unstructured, with large Shell and EXEC programs typically being totally unreadable to anyone other than their authors.

He suggests that what is required is a language where execution is immediate and easy, procedures are easily constructed and modified, data is symbol oriented (rather than number or string oriented), and where procedures can be used just like built-in commands. In his view, the LISP language, although over 20 years old, has all these properties, and would be ideal for the command language application. In his view, there is no distinction to be drawn between command and programming languages, and future implementations should attempt, where possible, to combine the two.

2.2.5 MXEC

As an alternative to a command programming language, another approach is to use a macro processor built on top of the standard command language interpreter. The MXEC system [Ash81], produced at Bolt Beranek and Newman Inc., is a sophisticated macro processor with the ability to create multiple parallel tasks. The aims of MXEC were to provide something which had powerful facilities and which enabled the user to customise his environment to his own tastes.

Instead of defining executable programs, MXEC uses a macro facility as a pre-processor for normal command lines. Through its structure, not only is it possible to produce complex command sequences, but also to perform file editing functions and string processing. MXEC is built as a user program which calls the DEC TOPS-20 EXEC program to execute commands. The TOPS-20 operating system was not modified in any way to support MXEC—a fact which aids its portability and hence usefulness.

MXEC allows multiple parallel processes to be created, and provides a facility to have "secondary I/O streams", which can be used to link separate jobs together, in a similar way to UNIX pipes. MXEC also provides a "daemon"—an asynchronous process which watches for a number of conditions, and informs the user when they occur. Services it provides are notification of new mail, completion of line printer listings, date and time, and so on.

The MXEC syntax is rather opaque, but Ash claims that, through the facilities provided to debug macros, even relative newcomers are soon able to construct their own command sequences. Like other macro processors, programs written in MXEC tend to be ugly and difficult to read, and as a result are usually short. As a system to tailor an environment to a particular user's needs, macro facilities are extremely useful and MXEC has achieved popularity at its place of creation. The fact that macro processing for the command language application has not caught on elsewhere implies that, as a general facility, it is less convenient and more difficult to learn than an executable programming language.

2.2.6 REXX

Of the languages described here, the only one to have been designed for both programming *and* command execution is the replacement for IBM's CMS EXEC language, REXX [Cowlishaw84]. REXX was designed primarily at the IBM UK Laboratories at Hursley, and was then continued at the Yorktown Heights Research Center. It is based heavily on PL/1 but is interpretive, and unlike conventional programming languages, its prime objective is to be easy to use.

The authors of REXX say that it has three major functions. Firstly, it is a language for personal programming, which is easy to teach and quick to learn. In this aspect, the language is very similar to BASIC and LOGO [Harvey82] in its outlook, and is now in heavy use as a tool for teaching computer newcomers. Secondly, it is a language for tailoring user commands, and so is very similar in this way to the UNIX Shell and MXEC. Thirdly, because the language provides powerful string manipulation and pattern matching facilities, it can act as a replacement for a general macro processor. As a language, it is easier to write in than most macro languages, and is more understandable by people who are not computer professionals.

The language attempts to be small, having only a few powerful constructs. Procedures, loops and conditional statements are almost identical to PL/1, with the exception that the syntax is less strict, and unnecessary punctuation can be omitted. All variables are held internally as strings of potentially infinite length, with operators provided for simple string and numeric calculations. One of the primary design aims of REXX was readability, and unlike other IBM languages, there is no restriction on the case of characters, or on the layout of the program. In order to make the language easy to use, variables are not declared explicitly, and all variable scoping is dynamic.

Cowlishaw believes that one of the most interesting features of REXX was that it was documented before it was implemented, and the documentation was distributed over IBM's VNET to potential users, in order to obtain feedback before any programming effort had been expended. He attributes the simplicity and ease of use of the language to this factor, and say that the comments of "real" users (as opposed to computer scientists) had meant that it was possible to draw the line between usability and complexity, and to create a language which was concise and simple without being cryptic.

3. TRIPOS

3.1 Introduction

All of the work described in this thesis is either implemented under, or heavily influenced by the TRIPOS operating system [Richards79a]. TRIPOS was developed at Cambridge by M. Richards and a group of his research students, and was intended to be a small, efficient, portable operating system for 16 bit mini-computers. Much of its portability comes from the fact that it is implemented almost entirely in BCPL [Richards69]. In an attempt to achieve simplicity, one of the facilities which was sacrificed was language independence, and in many ways, it is better to think of TRIPOS as a powerful multi-tasking run time system for BCPL. All the software described here which runs under TRIPOS is, without exception, implemented in BCPL.

The information included in this chapter is only that which is actually relevant to the work described later. A more comprehensive description of the techniques and data structures used can be found in [Knight82].

3.2 BCPL

BCPL is a block structured programming language of the ALGOL family, whose design came out of the work done on CPL [Barron63], from which it takes its name. The structure of BCPL programs is very similar to that in any other ALGOL-like language, but BCPL differs from the others in having only one data type—the machine word. The size of the BCPL word is defined by the nature of the underlying hardware, and BCPL achieves its portability by treating its memory space as a contiguous vector of these words, with each word having an address one different to its neighbours. All kinds of abstraction—integers, boolean values, character constants and so on—are represented in terms of simple values stored in a single BCPL word. As a BCPL memory address is just an offset in the vector of words, pointers can easily be stored, and operators are provided for obtaining the address of an object, and for indirecting on an address. Since there is no data type checking, pointer arithmetic is easy, but also prone to user error—it is perfectly possible to write syntactically correct BCPL programs which are semantically meaningless.

BCPL has four data storage areas available to the programmer. The *stack* is used to store procedure parameters and local variables, but unlike other ALGOL languages, dynamic free variables are not allowed. The *code area* is used to store

static variables (similar to ALGOL60 "own" variables). Static variables are rarely used though, since they have names which can only have scope local to a single code section, and their use tends to cause the inherent re-entrancy of a stack based language to be lost. Variables which have truly global scope are held in the *global vector*, and are represented simply as names which correspond to offsets within this vector. Unlike static variables, globals can be accessed by more than one section, and since each program has its own global vector, their use does not stop re-entrancy. The fourth area is the *heap*, but since access to the heap is via procedures in the run time library rather than built-in to the BCPL language, it is only possible to reference heap items via pointer variables.

BCPL allows programs to be compiled in separate sections, but since only the global vector is common to different sections, each global variable or procedure must be assigned a unique offset in the global vector. This is usually accomplished by having a single *header* file containing nothing but global declarations, which is then included by all separately compiled sections of a program. There is no interface checking though, and care must be taken to avoid global name clashes, and to ensure that when a header file is updated, all the BCPL sections which rely on it are re-compiled.

BCPL is intended as a language for systems programmers, and is ideal for writing compilers, editors and other utilities. It provides a high degree of power and flexibility to the experienced programmer, and in many ways it is convenient to think of BCPL as a structured assembly code, while having the portability of a high level language. Because of the lack of checking and the ability to write nonsensical programs, BCPL is not a beginner's language, and a great deal of self discipline is required when writing in it.

3.3 The operating system

The TRIPOS operating system is implemented as a simple kernel, on top of which runs a set of *tasks* and *devices*. Each TRIPOS task is assigned a *task id*, a positive integer, and a unique *priority*, also a positive integer, which defines the task's importance compared to the other tasks in the system. Since TRIPOS is a single language operating system, associated with each task is its BCPL stack and global vector, pointers to which are stored in the *task control block*. Each TRIPOS device is assigned a *device id*, a negative integer, and comprises five different pieces of code which are called during different stages of the device's lifetime.

Communication between tasks and devices is by means of ***packets***, which are blocks of memory, at least two but conventially five words in length. One of the fields in the packet defines the identifier (***task id*** or ***device id***) of the recipient. When the packet is sent, this field is altered so that it holds the identifier of the sender, and in this way, the packet can be returned to its owner.

Communication between tasks and the kernel is by means of a machine code library called *klib*, in which each kernel primitive is defined as a BCPL callable procedure. Some primitives need atomic access to certain data structures, and when this is the case, mutual exclusion is obtained by running with interrupts disabled. If a program is written in any language other than BCPL, then in order to call the kernel primitives, it must must use the standard BCPL calling conventions.

Scheduling is pre-emptive, with the highest priority runnable task always being the one which is given the CPU. No time-slicing is done, and so the task priorities have to be arranged so that starvation of important tasks does not occur. Devices, although they look like tasks to the user, are always called synchronously whenever a packet is sent to them, and device interrupt handlers must use a special set of kernel primitives to handle packets, since the standard primitives cannot be used from anywhere other than a normal BCPL environment.

Storage is allocated from a single pool, and no track is kept of the memory allocated by a particular task. A single store chain "first fit" algorithm is used, since it is easy to encode, and on a 16 bit mini-computer, the number of allocated blocks is typically fairly small.

3.3.1 The system tasks

The first four tasks in any TRIPOS system are fixed, to enable them to be located in a standard manner without the necessity of some kind of lookup. The system tasks are:

- Task 1 is always the main *user* task, and runs the standard command language interpreter described later in this chapter.
- Task 2 is always the *debug* task, and it is to this task that a packet is sent whenever some sort of exception occurs. On machines where memory is scarce, the resident debug module may be very small, with a command to enable the rest of the debugger to be loaded when required.
- Task 3 is always the *console handler* task, which is responsible for the handling of the console device, and providing console I/O streams for user programs.
- Task 4 is always the main *file handler* task, which is responsible for

handling the disc device, and providing file I/O streams for user programs.

The tasks mentioned above are guaranteed to have fixed task identifiers, but dynamic services are available through the use of the ***assignments list***, which keeps a mapping between the name and the task identifier of a handler. New devices and handlers can be loaded and unloaded explicitly by using the ***mount*** command.

TRIPOS was originally intended to run on stand-alone mini-computers, but with the development of the Cambridge Ring, the standard system tasks were replaced by equivalent code which handled virtual rather than real devices [Knight82]. The TRIPOS kernel was also used in this distribution process, since it provided a light-weight multi-tasking environment for the development of Ring services, notably the File Server [Dion80] and the Ring-Ring Bridge [Leslie83].

TRIPOS was therefore used for two totally different applications. Firstly, it provided a portable, single user operating system for mini-computers, with a command language, a filing system and many utilities. At the same time, the TRIPOS kernel was used as a multi-tasking run time system for the BCPL language, for the implementation of real time network servers.

3.3.2 Input/Output

The TRIPOS input/output system is extremely general, and borrows an idea originally used in OS6 [Stoy72]. Since every program under TRIPOS is written in BCPL, input/output is through BCPL *streams*, and each open stream is represented by a *stream control block*. Associated with the stream control block is all the control information for the stream, and all streams are handled in an identical manner, no matter what kind of device they represent. The relevant fields in the TRIPOS stream control block are:

link Not used by the I/O system, but is available for the user to allow him to keep a chain of open streams.

id Used to differentiate between input streams, output streams and update (both input and output) streams.

type Misnamed field used to hold the task identifier of the handler for the stream device. If negative, it is assumed that the stream is interactive.

buf Used to point to a buffer for this stream. For input streams, characters are read from the buffer, and for output streams, characters are written to it.

pos Used to keep the byte offset of the position in the buffer of the current character.

end Used to keep the byte offset of the end of the buffer.

func1 Used to hold a pointer to the stream ***replenish*** function. Whenever an input stream runs out of characters in the buffer, the stream replenish function is called to fill the buffer. If zero, this stream cannot be replenished.

func2 Used to hold a pointer to the stream ***deplete*** function. Whenever an output stream fills the buffer, the stream deplete function is called to empty the buffer. If zero, this stream cannot be depleted.

func3 Used to hold a pointer to the stream *close* function. Whenever either of the BCPL procedures ***endread*** or ***endwrite*** is called, the stream close function is called to free any resources allocated to the stream (for example, the buffer). If zero, then no special action is taken when the stream is closed.

There are also two extra user fields which can be used for special effects. The above mechanism is ideal for input streams, since it is possible to generate data to be read "on the fly", which can cut out an intermediate disc stage. This is particularly useful for such things as parameter substitution, queueing of input lines and so on, and is heavily used by much of the software described later in this thesis.

3.4 The command language interpreter

For an operating system to be at all usable on a small, 16 bit mini-computer, a compromise must be reached between the facilities offered by the operating system, and the amount of free storage available to the user. There is no point in having a system which provides powerful facilities if there is so much resident code that the user finds himself restricted when running his own programs.

The TRIPOS command language interpreter was conceived very much with this in mind, and the original CLI was extremely simple, with the main execution loop being only a handful of lines of BCPL. It was always intended that it should be easy to replace the simple CLI if its facilities proved inadequate, and the expected result was that each TRIPOS site would develop its own command language interpreter depending on the requirements of its users and the facilities provided by its hardware. In fact, what happened was that the original CLI was adopted as the standard one, and many utilities were written which relied on its particular environment. It was unfortunate that this happened, since although the original CLI was small and simple as intended, it provided no room for expansion, and after the introduction of the utilities which relied on it, further development was hindered.

The majority of the CLI code is involved with the general housekeeping of task creation and deletion, and with the handling of the CLI I/O streams. At the heart of the CLI there is a simple, infinite loop, which does the following:

- Prints out a prompt
- Reads a single item
- Treating the item as a filename, attempts to open it, looking first in the current directory, and then in the system command directory
- Loads the file into memory performing any necessary relocation
- Re-initialises the global vector, setting up all the globals defined in the module
- Creates a coroutine corresponding to the global routine *start*
- Calls the coroutine, giving it the same I/O streams as the CLI
- Deletes the coroutine and unloads the module

3.5 Arguments to commands

As mentioned above, the first argument on the command line is assumed to be the name of the file containing the command. After it is read, the remainder of the command line is untouched by the CLI, and so all argument reading and parsing is done by the command itself. Since the CLI's I/O streams are passed to the command intact, the command can read its arguments from the current input stream. It can also leave the input stream in whatever state it desires, knowing that the CLI will continue reading its commands from that stream, when it finishes.

In virtually all cases, command arguments are read using the BCPL procedure *rdargs*, which reads items from the current input stream, and parses them with respect to a user defined pattern. The pattern is defined as being a set of *keywords*, each with an optional set of *qualifiers*. The qualifiers are as follows:

`/A` The `/A` qualifier means that this argument must *always* be present (hence cannot be omitted)

`/K` The `/K` qualifier means that, if this argument is present, then it must be introduced by its *keyword* (hence cannot be positional)

`/S` The `/S` qualifier means that this keyword is a *switch* and does not take an argument

A typical *rdargs* pattern might be:

```
"FROM/A,TO/K,QUIET/S"
```

The first argument FROM is not optional (the /A qualifier) but need not have its keyword specified (absence of the /K qualifier). Hence, if the FROM keyword is omitted, the first unkeyed item will be taken by default. The second argument TO, must have its keyword specified (the /K qualifier) but is optional (absence of the /A qualifier). Hence, if the TO keyword is omitted, no positional parameter will be taken by default. The third argument QUIET is a switch (the /S qualifier) and is also optional (absence of the /A qualifier). Hence, the keyword QUIET is either present or not present—it does not imply an argument.

The following are examples of valid command arguments which would match the above pattern:

```
file1 to file2
file1 quiet
to file2 file1
quiet to file2 from file1
```

The *rdargs* procedure reads from the current input stream, stopping either on end of line or at the delimiter ";". It also handles the parameter "?" specially, treating it as an enquiry as to the parsing pattern, printing out the pattern and then prompting for further input. Because virtually all commands use *rdargs*, the TRIPOS user sees a simple, coherent command syntax, with well defined command separation.

There are some commands however which do not use this well defined interface to the user. They read characters from the input stream, performing the parsing themselves rather than relying on *rdargs*. Commands like this can therefore ignore command separators, and provide a totally different command syntax. They can also leave the input stream in whatever state they wish, which can have the effect of the CLI repeating or skipping commands, or even executing arguments as though they were commands. It is using this, rather primitive, mechanism that many of the so called CLI commands work, such as **input**, **run**, c and so on.

3.6 Programming conventions

Since virtually everything in TRIPOS is under the programmer's control, certain conventions must be adhered to if total chaos is to be avoided. One of these conventions—the use of *rdargs* to read and parse arguments to commands—has already been mentioned, but there are many others. The three

most important ones are described in detail here.

3.6.1 Storage allocation

All store in TRIPOS is allocated from the same pool using the BCPL routine *getvec*. When a block is allocated, no attempt is made to record which task allocated it, and no restriction is placed on how that store is used—for example, there is nothing stopping a piece of store allocated by one task from being passed as an argument to another task, and freed by it rather than by the allocator. As a result of this, it is the duty of each command to free all the memory which it has allocated, otherwise store will gradually be used up.

3.6.2 I/O streams

There are two conventions associated with I/O streams.

The first one ties in with "storage allocation", since the same rules apply to stream control blocks as to allocated store—namely, files which have been opened must also be closed. If not, then the files remain open and cannot be accessed again, and the store for their stream control block and data buffer remains allocated, causing store to be used up.

The second one relates to the I/O streams which are shared between the loaded command and the CLI. Just as the arguments to a command are read from the *current input* stream, it is conventional to read any other console input also from this stream. This is to make the writing of command sequences easy. Similarly, it is conventional to send any console output from the command to the *current output* stream. This is so that the output from a series of commands can be spooled to a file rather than sent to the console. In addition to the first point, neither of the current CLI streams must be closed, since the CLI is incapable of coping with this case.

3.6.3 Break conditions

Since TRIPOS does not keep lists of the resources obtained by a command, it cannot in any clean way stop that command running in response to a user break. This problem is exacerbated by the fact that TRIPOS uses messages as its inter-process communication mechanism, and since the messages are not copied when they are sent, there is the possibility that stopping a command may cause deallocation of resources which are already in use by another task.

As a result of this, the command itself must periodically poll for a break condition (using the BCPL routine *testflags*), and take appropriate action when one is detected. Appropriate action in this context means freeing all allocated store, closing all open files, and generally returning the environment to a

standard state before stopping. All commands must therefore perform this test regularly enough to give the impression of "forced termination", with the inevitable effect that they are slowed down because of this. There must also be a conscious effort to insert tests for the break conditions at strategic points in a program, otherwise it may be unstoppable.

3.7 The CLI environment

The CLI environment is stored entirely in a set of 20 CLI global variables which are shared between the CLI itself and the programs which it loads. In this way, loaded programs can give the impression of being "built in" since they alter the CLI environment in exactly the way that native code would.

The loaded command is therefore run using the same *global vector* as the CLI, but using a different *stack*, since the command is run as a coroutine of the CLI. The reason for this is that coroutine stacks can be created and deleted dynamically, and hence the size of the stack given to a loaded command can be set by the user to be appropriate for that command. This is particularly beneficial on machines with a small amount of memory, since the store can be split efficiently into code, stack and heap areas.

The CLI has two sets of input and output streams: the *standard* and *current* streams. In normal, interactive running, these two sets of streams are identical. The streams differ in one of two cases—the *output* streams differ when the output from commands is being spooled to somewhere other than the console; the *input* streams differ when commands are being read from somewhere other than the console.

Information as to the user's current position in the filing system is also held as part of the CLI environment. Locks (pointers to filing system descriptors) are held on both the *current directory* and the *system command directory.*

Apart from I/O streams and directory locks, the CLI holds all the information needed to run commands and store return code information after command completion. Information as to whether the CLI is in foreground or background mode is kept, along with information as to whether the current CLI input stream is interactive or not. In short, the twenty system globals reserved for the CLI are treated as an environment vector—one which holds all the information necessary to enable it to run.

3.8 CLI commands

There are a whole series of commands which alter the environment of the CLI. They are each fairly short and simple, but make explicit assumptions about the nature of the CLI under which they run. What follows is a small selection of these commands, with an indication of how they use the peculiarities of the CLI environment to perform their function.

3.8.1 The PROMPT command

The prompt given to the user when the CLI expects input can be changed by means of the `prompt` command. `prompt`, unlike the other examples in this section, actually uses *rdargs* to read its argument, and does nothing special with the CLI input stream. It does, however, update the buffer pointed to by *cli.prompt* (one of the CLI globals), which is the source of the prompt printed out by the CLI at the beginning of its main loop. For example:

```
prompt "Tripos> "
```

3.8.2 The RUN command

Commands can be run in the background by means of the `run` command. `run` just reads from the CLI input stream, copying the entire command line into a buffer, ignoring command separators. This buffer is then incorporated into a stream control block, and passed to a newly created CLI task as its standard input stream. The new task runs at a lower priority than the main one, and in "background" mode. In this mode, the printing of prompts is inhibited, and the task deletes itself when its main input stream is exhausted. For example:

```
run copy file1 to file2; delete file1; echo "Finished"
```

Note that, since the parsing of the command line is done by `run` rather than by the CLI, the command separators are ignored, and the whole command line is passed to a background task for execution. If the implementation of the CLI changes, then the effect of the `run` command may also change, and a discussion of this problem can be found in chapter 4.

3.8.3 The IF command

Commands can be run conditionally by means of the `if` command. `if` treats the first set of arguments on the line (up to the command separator) as being its own, and the rest of the line as being the commands to be obeyed conditionally. It evaluates its own arguments, yielding a boolean result. On *true* nothing is done, leaving the CLI to read and execute the rest of the command line normally.

On *false*, the rest of the command line is ignored (by repeatedly calling the BCPL routine *rdch*), causing the CLI to start reading at the next input line. For example:

```
if mctype 68000;  type 68000-info
```

3.8.4 The REPEAT command

Commands can be run repeatedly by means of the `repeat` command. `repeat` relies on a feature of the main CLI input stream—that a single command line is stored as a single buffer. It is therefore possible to reset the buffer pointer back to the beginning of the input line (by repeatedly calling the BCPL routine *unrdch*), causing the CLI to re-read the entire command line. For example:

```
wait 10 mins;  echo "Waiting ...";  repeat
```

3.9 Spooling

Spooling is simply the re-direction of program output (which would normally be sent to the console) to some other device, for instance a disc file or a printer. Spooling is possible under TRIPOS because of the two CLI output streams, and the conventions adopted by most programs.

Under normal circumstances, the *current output* is the same as the *standard output*, i.e. directed to the console. When spooling, the *current output* stream is updated to point to the new stream—so long as each command follows the convention that all output is sent to the CLI *current output*, it will all be redirected to the device. After spooling, the *current output* stream is closed and then set to be the same as the *standard output* again, restoring normality.

3.9.1 The SPOOL command

The `spool` command performs the function described above. Its argument is the name of a file to which output should be redirected. This file is opened, and selected as the CLI *current output* (after closing the previous one, if it was already different to the *standard output*). `spool` without any arguments simply closes the spool stream, and re-selects the *standard output*.

3.10 Command sequences

For the execution of a series of commands repetitively, command sequences are required. Command sequences (or command commands) are simply files containing lists of commands which would normally be typed at the console.

When executing a command sequence, the CLI takes successive commands from the file rather than from the console, but with similar effects. It is here that the two CLI input streams play their part. Under normal circumstances, the two streams are identical, and refer to console input. When executing commands from a file rather than from the console, the *current input* is set up to be the file stream. The CLI continues executing command lines from the file until the stream is exhausted, at which point the stream is closed, and the *current input* once more becomes the same as the *standard input*.

3.10.1 The T command

The t command has the effect described above. Its argument is the name of the file from which commands are to be taken. The file is opened, and set up as the CLI *current input*, carefully closing the previous one if it was different to the *standard input*. The commands are then taken from the file exactly as if they were typed at the console. As with the spool command, for t to work properly, the command input conventions must be adhered to—as long as this is the case though, what is contained in the file should be exactly the same as that which would have been typed at the console. For example, to compile a file, the following command sequence could be constructed:

```
echo "Starting to compile"
bcpl file to obj.file
join obj.file obj.library as command
echo "Finished compiling"
```

3.10.2 The LAB and SKIP commands

Because of the intimate coexistence of the CLI and its commands, it is possible to set labels within a command file, and skip forward to them. The lab command simply ignores its arguments and returns—a null operation. Its function is textual, since it is used by the skip command. The argument to skip is the name of a label; the action of skip is to read the command file, searching for a line beginning with the letters "lab" followed by the name of the required label. This facility, in conjunction with the if command mentioned earlier, makes it possible to construct command sequences which will work under different sets of circumstances. For example:

```
if mctype lsi4;   skip lsi4
if mctype 68000;  skip 68000
....
lab lsi4
....
```

```
lab 68000
....
```

3.10.3 Parameters

One of the primary uses of command sequences is to be able to use *parameters*, in other words, to write a general command sequence with the ability to have the "circumstance dependent" parts filled in dynamically. For example:

```
bcpl bcpl.<file> to obj.<file>
```

The above line defines a general command, which compiles a source file kept in the directory bcpl, storing the resultant object code in the directory obj. The name of the file is left as the parameter <file> whose value is not defined until the command sequence is executed.

3.10.4 The C command

The c command is like the t command, except that it allows the use of parameters and parameterised command sequences like the one described above. c, like t, takes its first argument as being the name of the file which is to be executed as a command sequence. The rest of the arguments are taken as the parameters which will be used to substitute the "gaps" in the command file. Within the file itself, there are control records which are interpreted specially by the c command itself, and which are not passed to the CLI for execution. One of these is the ".k" control record, which defines the format of parameters expected by the command file. The argument to the ".k" record is a *rdargs* pattern. For example:

```
.k file/a
bcpl bcpl.<file> to obj.<file>
echo "File <file> recompiled"
```

3.10.5 Temporary command files

Apart from the obvious matter of parameter passing, there is one other major difference between the c and t commands. The t command transfers control, initially from the console to a file, and then from one file to another. t never returns, and so the following example would not actually work:

```
bcpl bcpl.file to obj.file
t otherfile
join obj.file obj.library as command
```

The c command however is defined to return, and this poses a problem. To

implement this sort of recursive call properly, a *stack* of input streams would be required, whereas in reality there are only two—the *standard input* and the *current input.*

The way that the call and return is implemented is as follows. When c is applied to a file, it creates a temporary work file (in the directory "t:"), and makes a copy of the file being commanded there, incorporating any parameter substitutions. Included in the name of the temporary file is the task identifier of the commanding task, so as to avoid name clashes with any other CLIs which might also be doing the same thing. When copied, the new file is opened, and selected as the CLI *current input*, and execution continues as with the t command. The "returning" effect is obtained by the following ploy—after making a copy of the file being commanded, it continues to copy into the temporary file until the old file is exhausted. The effect is that after executing the commands which were in the command file, any remaining commands which were left from the old file are then executed as well.

If c is called twice, the temporary file into which the new copy should be made is *also* the current file from which commands are being taken. It can be seen therefore that two temporary files are necessary, if the problem of the same file being required for both reading and writing at the same time is to be avoided.

An extra piece of CLI assistance is required for the c command which is not required by t. To avoid temporary files being left in the "t:" directory after the execution of a command sequence, the CLI has not only to close down the command file when it is exhausted, but to delete it as well. To this end, there is a buffer (pointed to by one of the CLI globals, *cli.commandfile*) which holds the name of the current command file. This file is deleted when no longer required.

3.11 Return codes

When a loaded command returns, it can pass two return codes back to the CLI. One is a "severity" code, and is used by the CLI to determine whether to stop a command sequence prematurely. The second is a "reason" code, which can be used, in conjunction with the why command, to print out a meaningful message when an error occurs. If, for example, the command:

```
type bcpl.file
```

is executed, and the file bcpl.file does not exist, the type command stops with return code 10 (meaning "hard error") and reason code 5156 (meaning "file does not exist"). The two return codes are stored in CLI globals *cli.returncode* and *cli.result2* respectively. As would be expected, the if command can be used to

test the values of these globals. The conventional return codes are 5 (soft error), 10 (hard error) and 20 (total failure); these can be tested using `if warn`, `if error` and `if fail` respectively. A return code of zero implies success. The `quit` command treats its argument as an integer, and simply returns to the CLI, setting the return code to the value given. If run in non-interactive mode, `quit` also skips the rest of a command sequence to ensure that the return is immediate.

Return codes have little meaning when in an interactive environment, but when executing a command sequence, unless the user explicitly requests it, there really is little point in continuing after one of the commands in the sequence fails. The user can set a threshold at which the command sequence stops by using the `failat` command. `failat`, like the other CLI commands already mentioned, simply sets the value of one of the CLI globals, this time *cli.userfaillevel*. Alternatively, using the `if` command, errors can be detected within the command sequence. The following command sequence uses many of the facilities of the CLI which have been described above, and shows what is possible using the very simple facilities available:

```
.k file/a
failat 10
if mctype lsi4;  skip lsi4
if mctype 68000; skip 68000
echo "Unknown machine type!"; quit 20
lab lsi4;  bcpl bcpl.<file> to lsi4.<file>;  skip check
lab 68000; bcpl bcpl.<file> to 68000.<file>
lab check
if error; echo "Error in compilation"; quit 20
c link <file>
if error; echo "Error in linking"; quit 20
echo "Compilation and linking complete"
```

3.12 Evaluation of the CLI

It cannot be denied that the CLI is small, but its interface is far from simple. From the user's point of view, the facilities provided are crude and, although usable, certainly not ideal. Most commands have the same kind of user interface and usually work in the same sort of way, but there are exceptions which may cause confusion, particularly to novices. Because the arguments are read from the input stream by the command, rather than being processed by the CLI, there are ambiguities which are hard to resolve. For instance, consider the following:

```
input to file1;  input to file2
blah1 blah1 blah1
```

```
/*
blah2 blah2 blah2
/*
```

Simplistically, one might expect the blah1 line to be stored in file1, and the blah2 line to be stored in file2. This is not the case, since the input command first reads its arguments using *rdargs*, and then continues to read until the terminator /* is found. The result is that file1 contains:

```
  input to file2
blah1 blah1 blah1
```

with the result that an attempt is made to load blah2 and /* as commands.

From the programmer's point of view, the interface to the command language interpreter is also far from satisfactory. Although he has total control, he must exercise a great deal of self discipline and keep to the TRIPOS conventions, or the results will be at best unpredictable, and at worst, disastrous. Ironically, although each command has full access to the CLI environment, there exists no interface for a command to call the CLI recursively in order to execute a command line from within a program. This means that it is impossible to write a *new* command language interpreter simply as a sophisticated pre-processor which passes command lines to the simple CLI for execution. Anything which wishes to act as a CLI in fact must totally replace the original, and because of the CLI commands described earlier in this chapter, any new version must either re-implement these commands or provide all the same interfaces to enable all the old ones to continue to work.

4. Enhancing the User Interface

4.1 Introduction

This chapter investigates the areas in which the command language interpreter described in chapter 3 can be improved, and in particular, how facilities which make use of large amounts of memory can be incorporated, in order to enhance the user interface. It shows how an existing command syntax can be altered in such a way as to allow constructive program concurrency, and discusses some of the problems associated with introducing enhancements to an existing system.

Some of the work described in this chapter is quite heavily based on the UNIX Shell [Bourne78] and C Shell [Joy80], since they have already tackled many of the relevant problems. Because of its superficial similarity with the UNIX Shell, the enhanced TRIPOS CLI has been called the TRIPOS Shell. This is probably a misnomer, since the TRIPOS Shell is merely a sophisticated command language interpreter, and not the all-embracing working environment of its UNIX counterpart. In particular, the TRIPOS Shell does not include any sort of programming language, as this facility is handled by the REX system, described in chapter 5. Since the facilities provided by UNIX have proved so popular, there was little incentive to design something new. In fact, providing a high degree of compatibility actually has a great advantage in an establishment such as the Cambridge Computer Laboratory, since many of its inhabitants spend much of their time swapping between different systems in order to get their work done.

4.2 Areas of enhancement

Given the new generation of 68000 TRIPOS machines, some of the design decisions, taken with respect to the smaller systems were beginning to look rather conservative, to say the least. If memory is plentiful, then it is no longer important to save store by having an extremely simple CLI, with the more complex facilities being provided by loaded commands. The size of resident programs becomes insignificant compared to the total amount of memory, and so it is more reasonable to provide built-in facilities. It is also possible to keep certain programs pre-loaded ready for use, and this technique is discussed in chapter 6.

Because of the extra memory it is possible to have more resident tasks, enabling much better use to be made of the available CPU power. Being able to

have many resident tasks also aids the process of program design, since logically distinct activities can be split, giving a higher degree of modularity. The ability to have many resident tasks is only useful if the command language interpreter makes it easy to run many jobs in parallel. Similarly, the command language interpreter must make it possible to connect tasks together, so that the output of one becomes the input of another.

One way of improving a user interface to a command language interpreter is to cut down the amount of typing necessary. The time when this is most important is the case when a command line has been mistyped, since it is irritating and inconvenient to have to type the whole line in again, just to correct a small mistake. A command history and line editing facility can be added simply to a command language interpreter, so long as the relevant interfaces are present.

Another advantage which arises from the definition of a command history interface is the ability to execute command lines which have not been typed directly at the console. This means that it is possible for programs running under the command language interpreter to call it recursively, passing to it a command line to be executed. The existence of this interface makes it reasonable for the user to provide a command pre-processor to handle any desired command syntax, and means that it is possible to design a language which incorporates a command execution facility.

4.3 Pipes

Experience with UNIX has shown that, given sufficient memory and CPU power, with two tasks which are essentially serial in nature, much time can be saved simply by connecting the *output* of one process to the *input* of the other. Using this method, maximum parallelism is achieved, without the added overhead of spooling intermediate results to disc. A typical application is in the case of a compiler and its language pre-processor. Here, the pre-processor is run on the raw source, and the result is a file which is acceptable to the compiler. The compiler uses the intermediate file, producing compiled code as the result.

The mechanism used to send the output of one program to the input of another program is the *pipe*. A pipe is essentially a pair of I/O streams, one input and one output, which feed into and out of a buffer. The pipe *writer* continues to write into the pipe until the buffer is full, at which point it is suspended. The pipe *reader* continues to read from the pipe until the buffer is empty, at which point it is suspended. The two tasks are therefore self-

regulating, in that if the writer produces too much output, then it is suspended until the reader catches up. Similarly, if the reader consumes all its input, then it is suspended until more output is produced.

4.3.1 UNIX pipes

The UNIX pipe system works well because of two, related factors.

Firstly, there is a simple programming convention that all programs, by default, take input from the *primary input* stream, and send their output to the *primary output* stream. (These streams are directly analogous to the TRIPOS *current input* and *current output* streams described in chapter 3). In the UNIX case, I/O streams are normally presented to a program implicitly and already opened, whereas with TRIPOS, input and output files are presented to a program textually on the command line, and therefore unopened.

Secondly, and as a result of the previous point, UNIX is able to make use of a simple and powerful command syntax, where single characters can be used to define primary input and output streams, and pipes connecting two processes. `<file` means "take the primary input from `file`", `>file` means "send the primary output to `file`", and `command1|command2` means "pipe the primary output of `command1` into the primary input of `command2`". For example:

```
preproc <file1 | compile >file2
```

runs the pre-processor command `preproc` which takes input from `file1`, and pipes the output into the compiler command `compile` which sends its output to `file2`.

4.3.2 TRIPOS pipes

The TRIPOS Pipe Handler was written originally by M. F. Richardson, and like many TRIPOS facilities is implemented as a device, "`pipe:`". Unlike UNIX, the pipes are not anonymous, and must be given names by the user. Pipes of the same name can then be opened for *reading* by one task, and for *writing* by another task, at which point they work in exactly the same way as UNIX pipes, with the output of one task being sent as input to the other task.

The `pipe:` device pre-dates the TRIPOS Shell by three years or so, but was not in common use because there was no simple way of using pipes implicitly. All pipe naming had to be explicit, as did the use of background tasks. For example:

```
run preproc file1 to pipe:name1
compile pipe:name1 to file2
```

When compared with the clarity and succinctness of the UNIX example above, it is not difficult to see why TRIPOS pipes were little used.

4.3.3 Pipe expressions

Given that pipes must have textual names, and that these names must appear in their rightful position on the command line, a completely different syntax to that of UNIX is appropriate. The most important point, is that all arguments to TRIPOS commands are textual and are normally read using *rdargs*, hence they must match some predetermined pattern. This means that, unlike UNIX, pipe symbols cannot be treated as command separators (since the position of the pipe symbol in the line is fixed) and so some other scheme must be devised.

Taking the previous example, the first stage in rationalisation is to remove the need for the explicit call of *run*, and to replace the pipe name given by an anonymous pipe symbol. The two command lines then become:

```
preproc file1 to |
compile | to file2
```

This cuts down on the amount of typing necessary, but it is still of no use since the two commands have not yet been joined in any way. To the user, sending output to a *program* rather than to a *file* is easy to conceptualise—it is simply the replacement of a *file name* by a *program name* instead. Under TRIPOS, one way of doing this is to use syntax of the form:

```
preproc file1 to (compile | to file2)
```

Here, the pipe symbol "|" of the first command line is just replaced by the whole of the second command line. Note that delimiters, in this case parentheses, are required in order to remove possible ambiguity over textual binding. Using the same logic as above, the command line can also be re-written as:

```
compile (preproc file1 to |) to file2
```

which is similar, except that the substitution of the pipe symbol has been done on the second command line.

The commands within parentheses containing pipe symbols are called *pipe expressions*. The TRIPOS Shell reads the single command line, and treats pipe expressions specially. Firstly, the pipe symbol is replaced textually by a Shell generated pipe name. The whole pipe expression is then extracted from the command line, and is replaced by the *same* pipe name as that used within the expression. Thus, taking the original example again, out of:

```
preproc file1 to (compile | to file2)
```

the two resulting command lines are:

```
preproc file1 to pipe:xxxx
```

```
compile pipe:xxxx to file2
```

Pipe expressions are run as background tasks, with the main command running in the foreground.

The syntax chosen has a couple of drawbacks. Firstly, because a pipe expression can return at the most one result, only one pipe symbol per pipe expression is allowed. This means that UNIX command lines like:

```
preproc <file1 | compile | assemble >file2
```

cannot be represented as succinctly under TRIPOS, with the result of the translation being:

```
preproc file1 to (compile | to (assemble | to file2))
```

which is more complicated, and certainly less obvious. Secondly, because it is impossible for the Shell to determine whether a command will open a pipe for input or output (or just treat it as a piece of text), it cannot check for clashes, and mistyping of a command line can cause total deadlock. Command lines such as:

```
copy file1 to (copy file2 to |)
```

are syntactically correct, but otherwise meaningless, since the same pipe name would be opened for output twice. Having mentioned the problems, users have not found the single pipe name to be a restriction, since TRIPOS commands were never designed to make use of piping in the first place. Similarly, pipe deadlocks are few and far between, and can be corrected by killing the pipes concerned, and starting again.

One pleasing outcome of the chosen syntax is that it is possible to have pipe expressions which do not contain any pipe symbols. The result is that the command within parentheses is executed in the background as before, but since no pipe has been used, it is replaced by the null string in the original command line. A simple command within parentheses is therefore identical to using *run*. For example:

```
(copy file1 to file2)
```

4.4 Command history mechanism

The command history mechanism is one of the most useful features of the UNIX C Shell. In a conventional environment where commands are typed interactively to a command language interpreter, there is always the very real possibility that a typing mistake will be made, and the line will have to be re-

typed. Also, in such an environment it is quite likely that having executed one command line, the next one will be very similar if not identical.

Many intelligent terminals keep a stack of recently typed lines, and provide simple line editing facilities. This copes with both the above cases. Command lines which are in error can be recalled, edited to correct the error, and then re-transmitted to the host computer for execution. Similarly, command lines which differ only superficially from recent ones, can also be recalled, edited and then executed. An obvious example of this type of terminal is the IBM 3270 series, which provides not only local line editing, but local screen editing as well.

The principle of keeping a command history is to make use of the two simple facts outlined above, and to provide a mechanism for their exploitation without the need for an intelligent terminal. A command history is, as the name implies, a collection of the most recently executed command lines, stored within the command language interpreter. Special commands are then provided so that the history may be interrogated, and individual command lines extracted for modification and re-execution. For this to be a worthwhile feature, it must obviously be less work to recall and edit a command line than to type it in again from scratch. The character sequence to recall a specific command line must therefore be short and accurate, and editing commands must be simple and easy to use.

The syntax chosen to recall old command lines is exactly that used by the UNIX C Shell. There were two main reasons for using a specification which already existed, rather than inventing something original. Firstly, many members of the Computer Laboratory were either using UNIX heavily in Cambridge, or had used it heavily at their previous locations. For their sake, it seemed perverse to produce a system which did the same thing, but in a different way. Secondly, there would have been an argument for choosing another method if the original one had proved inadequate, over-complicated or cumbersome. Many of the C Shell facilities provided are indeed very complicated, but what arose from talking to many C Shell users was that only a subset of the features were actually in common use, and those which were did exactly what was wanted.

In defining the syntax to be used for the TRIPOS Shell, as with any re-design, those features of the old system which were deemed to be good were kept, and those which were either never used or considered too complicated to use were discarded. Having recalled a previous command line, it is possible to apply simple editing commands to it before it is executed. Again, the syntax chosen is a subset of that used by UNIX, but very little of the power is lost because of the

simplification. A description of the TRIPOS Shell command history mechanism can be found in appendix 1.

4.5 Command execution

For the pipe expression and command history mechanisms to work, it must be possible to take a series of command lines, and execute one in the foreground and the rest in the background. It was at this point that the first real restriction of the TRIPOS CLI was recognised, since it has no well defined program interface—there is no way of being able to call the CLI from a program in order to execute another command line recursively. Executing command lines in the background is easy, since the mechanism of the ***run*** command is perfectly adequate for them, but the foreground command presents something of a problem.

There were three possible solutions to the problem, depending on the required amount of compatibility with the old system. Firstly, if the Shell were to remain as a program running under the standard CLI, then some other way of executing the command lines would have to be found. Secondly, as an alternative to the previous point, the standard CLI could be altered to add the recursive call facility which would be required by the Shell. The third possibility, and the one which was finally adopted because of problems with the other methods, was to remove the old CLI completely, replacing it with something which both has the required program interfaces and also handles the ***pipe expression*** syntax.

4.5.1 The command executor task

In an attempt to clean up the interface to the TRIPOS CLI, the *command executor task*, or *cet*, was born [Wilson83]. The principle of the *cet* was simple—to provide a TRIPOS task with a well defined, packet level interface, which would execute a command line, and on completion of the command, would yield the final return codes. In other words, to abstract from the CLI the part of the code which was involved with command execution, leaving just those parts which deal with syntax analysis and command decoding.

This was exactly the interface required by the Shell—after parsing the main input line, each resulting command line could then be passed to the *cet* for execution, with an indication as to whether it should be run in foreground or background mode. The two functions provided by the *cet* are as follows:

```
(id, priority)  :=  setup( command )
(rc, result2)   :=  execute( id, environment, foreground )
```

In the first call, the argument to the *cet* function **setup** is simply the name of a command. The *cet* loads the command into memory, and creates a separate task in which the command will eventually run. The two results of the call are the `id` of the newly created task, and the `priority` at which the task will run.

In the second call, the arguments of the *cet* function **execute** are the `id` of the task to be executed (one of the results of the **setup** function), a vector containing the environment in which the command should be executed, and a boolean flag stating whether the command should be run in foreground mode. The two results of the call are return codes **rc** and **result2**, which resulted from the execution of the command. The environment vector has the following entries:

- Stacksize—the size of the stack which the command should be given
- Arguments—the arguments to the command (the command line without the command name)
- Input Stream—the *current input* for the command
- Output Stream—the *current output* for the command

The environment to each new command is set up afresh just before command execution, and any changes in the environment are not passed on to subsequent commands. The concept of the *cet* is therefore much cleaner than the original CLI, and in an ideal world, would have worked well in practice. So what went wrong? The *cet* did work, after a fashion, but much of the existing system was designed in such a way that its performance was never acceptable.

4.5.2 Command arguments

The first problem is one of command arguments. In chapter 3, the principle of reading command arguments from the CLI *current input* stream was explained. This in itself does not present too much of a problem to the *cet*—it is easy under TRIPOS to construct an input stream control block which turns an arbitrary string (e.g. the command arguments) into a stream which can be read using *rdch* or *rdargs.* What does present a problem though, is that commands are entitled to continue reading this stream in order to obtain the remainder of their command input. In fact, as explained before, not only are they *entitled* to do so, but TRIPOS programming convention defines that they *should.*

The new stream control block which must be constructed is now rather more complicated. Not only must the command arguments be packaged up into the buffer of the stream control block, but the *replenish* function must be set up so that, when the command line is exhausted, further characters are read from the true *current input* stream instead.

4.5.3 Type ahead

In order to minimise confusion when many tasks wish to read from the console, TRIPOS has the concept of a *current input task.* This is the task to which input typed at the console is sent, and can be changed by typing the "@Snn" command to the console handler, or from a program by sending a *set current input task* request to the console handler task. All tasks other than the currently selected one are suspended if they attempt to read from the console, and are only re-activated when they themselves become the current input task.

It can be seen at once that the *cet* runs into difficulties here. To read the initial command line, the current input task must be the Shell. When running the foreground command, the current input task must be the task which is running that command. This in itself is not a problem, since the console handler provides the interface to switch input tasks.

The real problem arises as follows. The current input task cannot be changed until the code for the command has been loaded and the new task corresponding to it has been created. With large commands, there can be a delay of several seconds between typing the name of the command, and the command being loaded and ready to accept input. The result is that, after typing the original command line, the user has no idea of how long he must wait in order to make sure that items typed at the console will go to the *command* task rather than the *Shell* task. Most programs which require console input write out a prompt when they are ready (e.g. the `edit` command), but others (e.g. `input`) do not, so with these commands, the user can never be sure whether what he is typing is having the correct effect.

This is not the only place where the switch of tasks causes a problem. The other major area which is affected is that of "type ahead", that is the ability to type future commands while the current command is still executing. This no longer works if the current input task is altered, and the reason is simple. While a command is executing, the current input task is the task which is handling the command. Anything which is typed during the execution time of that command is queued up for that task, remaining unread. When the command finally completes, the current input task is set back to the original CLI task, for which the type ahead was originally intended. The effect to the user is that the extra commands he typed have been "swallowed" by the system, but the actual result is more subtle than that. The *next* time that a command task is created with the same task identifier as the one which had the type ahead input queued for it, those lines will appear as though they were typed as data to that command.

One way round this problem would have been, rather then keeping the queues of input lines on a "per task" basis, to have just one queue, with only the current input task being entitled to read the queued lines. For this mechanism to work properly, the switching of current input tasks must be done synchronously (with console commands being queued in the same way as other input). This removes the benefit of being able to have multiple tasks, since a new task cannot be activated until the old one has read all its input lines, and so was rejected.

4.5.4 Existing TRIPOS commands

Because the many TRIPOS CLI commands have such an intimate relationship with the CLI itself, they tend to stop working when they are taken out of their original context and put somewhere new. The *cet* brought out many problems of this nature. For most commands, it is possible to imitate the CLI environment enough for them to continue to work as before—the majority of commands do not touch the CLI environment at all, and if they do, it is simply to test the CLI "background" or "interactive" flags.

The problem arises with commands like `prompt`, `t`, `spool` and so on, which rely on altering the CLI environment in order to work properly. If the commands are executed in a task other than the main CLI, the results are harmless but do not have the desired effect. Take for example the `prompt` command. This alters the the buffer pointed to by *cli.prompt*, which is one of the CLI globals. This buffer is local to each task, and so running the `prompt` command therefore alters the prompt in the *command* task, not the CLI task, and so would appear to the user to have no effect. More seriously, commands like `t` and `spool` cease to work, since their effect relies on altering the CLI's current and standard I/O streams—these streams are also local to the command task, and so the real CLI environment remains unaffected.

4.5.5 Conclusions

Although the command executor task is, in principle, a clean and simple way of solving some of the CLI problems, many factors make its operation far from ideal. Many of the problems described above can be overcome by introducing a set of built-in commands, that is a set of commands which are interpreted directly in the Shell itself rather than being executed by the *cet*; in this way, commands such as `prompt`, `t` and `spool` can be made to work. Even when this has been done though, the problems of not knowing which the current input task is, and being unable to use type ahead, both add up to a system which is frustrating to use and unpredictable in its results.

4.6 Implementation of the Shell

Because of the problems outlined above, it is essential under TRIPOS for foreground commands to execute as part of the CLI task. This, combined with the fact that the old CLI provides completely the wrong interface to support a new one built on top of it, leaves only one other possibility—to replace the old CLI with a new one—the Shell. As will be seen later though, this replacement must in essence be an augmentation, since there are so many existing commands which make assumptions about the environment in which they run. Unfortunately, in order to maintain compatibility for the user, these commands must be accommodated in any new implementation.

In the original implementation of the CLI, there was the concept that all CLI input—both command lines and data for the commands—came from the same input stream. What was left unread on the command line by the CLI was considered as data for the command, and vice versa. This effect is very difficult to simulate if the main input stream is treated in a different way. With the pipe expression and command history mechanisms described above, the Shell must treat each command line as a whole. Consider the following example:

```
compare (typehex file1 to |) (typehex file2 to |); echo "Done"
```

This is a single command line which would originally have been read from the console, and possibly later extracted from the command history. When parsed, no less than four separate command lines remain to be executed, two in the background and two in the foreground.

4.6.1 Background execution

The mechanism of the *run* command is used when executing a command line in background mode. Basically, console input is a meaningless concept for background commands (since console input is always directed towards the foreground task), and so it need not be provided. This fact simplifies matters immensely. Having established the command line to be executed in the background, the next step is to create a *pseudo* input stream from which the background task will read its command line. Under TRIPOS, this is straightforward—a stream control block is constructed with the following parameters:

buf Buffer containing the command arguments

pos Zero (beginning of line)

end Number of characters in the buffer

func1 Zero (no *replenish* function)

func2 Zero (no *deplete* function)

func3 Zero (no *close* function)

The effect of such a stream is that the command line is read as though it had been typed at the console, but since the *replenish* function is missing, "end of line" is also treated as "end of stream". After executing the single command line, the background Shell will therefore detect "end of stream", at which point it closes down. The pseudo stream is closed (a null operation since the *close* function is also missing), and the task then deletes itself.

To ensure that the storage used by the command line buffer is freed when the pseudo stream is closed, two solutions exist. Firstly, a *close* function could be defined which freed the buffer when it was called. Secondly, the storage for the stream control block and the command line buffer could be allocated together as one contiguous area of memory. No *close* function would now be required, since when the stream control block was freed, the buffer (being part of the same piece of storage) would be freed also. The latter is the simpler solution, and was the one chosen.

Having constructed a dummy input stream, a new Shell task is created to deal with it. All parameters from the main Shell are copied across to the new one by a variety of means. The stack size for the Shell itself is a parameter to the BCPL function *createtask*, and hence is passed implicitly when the task is created. The size of the command stack and other dynamic parameters are passed to the new Shell in the *startup* packet, that is, the first packet to be sent to the new task just after its creation.

Fixing the global vector size for the newly created task is not quite as simple. When a task is activated (when it receives its very first packet following initial creation), the task segment list is scanned, and the global number of the highest referenced global is found. This number determines the exact size of the global vector for the new task, and must be set artificially high for a Shell, if it is to cope with as many loadable commands as possible. The way the global vector size is set dynamically is to add an extra entry to the new task's segment list—one which contains no executable code, but one which has its "highest referenced global field" set to the desired value. Hence, when the task is activated, it is this value which determines the size of the global vector, not the highest referenced global of the Shell itself.

All of the Shell environment—the current working directory, the prompt, and so on—are all passed to the task either as simple arguments in the task *startup*

packet, or as entries in the Shell environment vector, which is also passed. The new Shell is, as nearly as possible, a clone of its parent.

4.6.2 Foreground execution

Unfortunately, execution of foreground commands is not so simple as the background case. The main problem comes from the original design decision—that command names, command arguments and command data should all be read from the same input stream. Before describing some of the contortions which must be employed in order to solve this problem, it is worth saying how it might have been done differently, and why this solution was not adopted instead.

By changing the definition of *rdargs*, it would be possible to take input from a Shell buffer rather than from the current input stream. This buffer could either be passed to *rdargs* as a parameter, or be accessible as a Shell global variable. Adding an extra parameter to *rdargs* is, although the more general of the two possibilities, not a good idea, since all programs would have to be edited to take account of the new version. Using a global Shell buffer solves the simple case where *rdargs* is used just to parse the arguments to a command, but does not take into account the utilities which use *rdargs* to parse data files or interactive input lines. These programs would have to be altered so that they stored the relevant data characters in the Shell buffer, rather than relying on them being read from the current input stream. Both possibilities require existing programs to be modified, and would mean that compatibility with other TRIPOS installations would be lost.

Given that the simple solution was inappropriate, another solution had to be found. The requirements were as follows: it had to be possible to construct an input stream for each command, which both contained the arguments to that command and provided access to the main Shell input stream, in case the command needed to read data from it. This in itself was not particularly complicated, after all, the *command executor task* employed just such a method in order to solve the same problem. Unfortunately though, this is not the solution to the whole problem.

The general principle is the same as that originally designed for the *cet.* A dummy stream control block is constructed with the following properties:

buf Buffer containing the command arguments

pos Zero (beginning of line)

end Number of characters in the buffer

func1 Special ***replenish*** function

func2 Zero (no *deplete* function)

func3 Special *close* function

arg1 Pointer to the main Shell input stream control block

The properties of such a stream control block are that the command line will be read normally (from the buffer provided), but if the command continues to read from the stream, requiring input data, then the special ***replenish*** function is called. The result of this call is to over-write the dummy stream control block with the information held in the stored main stream control block. After doing so, a character is read from the saved stream, causing the real ***replenish*** function to be called. Should the special *close* function be called, which is possible with the `c` and `t` commands, then the main stream control block is copied over the dummy stream control block, as with ***replenish***, and then the real *close* function is called in order to free any allocated store. The command buffer used in the dummy stream control block is also freed.

If the special ***replenish*** or *close* functions are called, then nothing need be done to restore the *status quo* after execution of the command. If, however, the dummy stream is not over-written by the standard stream, then this must be done explicitly by the Shell. There are problems associated with this process, in that it is possible for one of two things to have happened:

- If the `endcli` command had been executed, then the input stream parameters would have been patched, so that *end of file* would be encountered on the next read operation.
- If the `t` or `c` commands had been executed, then a new Shell current input stream would have been created.

Either way, the stored stream control block cannot simply be copied back over the dummy stream control block indiscriminately, and the algorithm adopted must depend on the state of the current and standard input streams before and after command execution. On closer inspection, it is clear that only two situations are actually possible.

If the Shell were running in "normal" mode, in other words, not running a command file, then before execution of the command, both the standard and current input streams will be the same—they will point to the dummy stream. After execution, then, assuming the standard stream has not been patched by `endcli`, the dummy stream should be copied over it. The only possible change is that the current stream will have been replaced with a new stream, by the `t` or `c` commands.

If the Shell were running a command file, then before execution of the command, the standard and current input stream will be different, with only the current stream pointing to the dummy stream. After execution, there cannot have been any change of state in the streams, since this would have been reflected by a call of either the special *replenish* or *close* functions. Assuming that the current stream has not been patched by **endcli**, the dummy stream should be copied over it.

4.6.3 Shell search paths

One of the inadequacies of the old TRIPOS CLI was that it only searched two command directories in order to locate loadable commands. First, the current directory was searched, followed by the main system command directory, usually "sys:c". The Shell searches two other directories as well, thus allowing user command libraries to be set up.

The mechanism used is the *directory assignment*, in other words, an entry in the assignments list which refers to a disc directory rather than to a mounted device. The advantages of using directory assignments are that, firstly, access to the assignments list is quick, and secondly, directory assignments can be set up and changed easily by means of the **assign** command. The order in which the directories are searched by the Shell is:

- The current working directory
- The directory referenced by "COM1:", if any
- The system command directory
- The directory referenced by "COM2:", if any

It is therefore possible to have user command libraries which either do or do not over-ride the system command directory. Also, by taking advantage of the "DIR:" device (see chapter 6), assignments can be made to groups of concatenated directories, thus "COM1:" is in fact a *set* of directories which are searched before the system command directory, and similarly for "COM2:".

As part of its interface with the REX interpreter, the Shell also searches the current working directory, and the assignment "REX:", in order to find REX programs. This mechanism is discussed in more detail in chapter 5.

4.6.4 Built-in commands

As described in chapter 3, the original TRIPOS CLI had no built in commands, but achieved the same effect by allowing loaded commands to update its data structures. When the original design was done, 20 global slots were

reserved for CLI data structures, and then each global was allocated to its specific task, leaving no room for expansion.

The Shell has many more data structures than the old CLI, but in order to keep compatibility with programs written for the old system, it must maintain the CLI globals as before. This means that some of the Shell data structures are accessible via these system globals, but other structures are only accessible through the Shell's own private globals. As explained above, the Shell saves its entire environment while executing a loaded command. The result of this is that the commands which alter the Shell data structures *not* in the system area, must therefore be local to the Shell itself. Those which fall into this category are termed "built-in commands".

The number of built-in commands has been kept to an absolute minimum. Those which are provided either manipulate the Shell data structures, or override loaded commands which make invalid assumptions about the Shell environment. These assumptions are where particular details of the old CLI environment have been exploited in order to give a special effect. There are two areas where this is the case—command line parsing, and global vector allocation.

4.6.5 Command line parsing

One of the major differences between the CLI and the Shell is the place where the splitting of the command lines into their constituent commands is done. When running under the CLI, it is up to each individual command to leave the main CLI input stream in a state fit for the next command. Hence there are no enforced breaks in command lines, and there is nothing to stop individual commands reading beyond command separators. Three commands which rely on this fact in order to function properly are `run`, `repeat` and `if`. Specifications of these commands can be found in chapter 3.

Under the Shell, command lines are read as whole entities (so that they might be stored in the command history), and then subsequently parsed, and split into their individual commands. Taking an example, the difference in the effect of the two systems can easily be seen:

```
run copy file1 to file2;  delete file1;  echo "Finished"
```

Under the CLI, the effect is to create a background task which runs the `copy`, `delete` and `echo` commands in sequence. Under the Shell, the whole command line is split into three separate commands: `run`, `delete` and `echo`, which would then be executed in sequence. The side effect of the `run` command is to create a background task which runs the `copy` command. Note here that, not only are the semantics of the command line different in the CLI and Shell versions, but the

overall effect may be catastrophic. If the **delete** command begins execution after the background copy command, then the delete will fail, since the object **file1** would already be in use. This is not the effect wanted, but is however, harmless. Alternatively, given that the background task is of lower priority than the foreground Shell, it is likely that the **delete** command will actually begin execution before the background copy command—the result being that **file1** is deleted before it is copied, and any information held in **file1** is lost.

The **repeat** and **if** commands suffer the same problems as the **run** command, and all three must be handled specially. For that reason, they are included as built-in commands, where it is possible for them to access the command line before it is split into its constituent commands.

4.6.6 Global vector allocation

One of the reasons that loaded commands are executed as coroutines of the CLI or Shell is so that the size of the BCPL stack given to the command can be set dynamically. Unfortunately, things are not quite so simple with respect to the global vector. The loaded command and its parent CLI or Shell share the same global vector, the size of which is determined by the TRIPOS kernel at task creation time. Although the global vector size is set to be fairly large (currently 600) there is always the possibility that certain commands will require a global vector larger than this. The Cambridge Rainbow Group, for example, use a modified version of the BCPL run time library [Wilkes82] which calls for a global vector which is double the default size.

Under the old CLI, the problem of changing the global vector size was tackled in a cunning, if rather inelegant manner, by means of the **globals** command. The command worked as follows. First, all the relevant CLI parameters stored in the global vector were extracted, and put into a TRIPOS packet. This packet was then sent to the CLI task, leaving it enqueued on its work queue. After saving this information, the "highest referenced global" field of the CLI code segment was patched to reflect the new required global vector size, and the main CLI input stream was closed prematurely. The CLI, being unable to read commands from the closed stream, shut itself down. The stack and the global vector were freed, and the task state was set to "dead". However, since the CLI had sent itself a packet before committing suicide, it was then immediately re-activated by the kernel scheduler, with a new stack and global vector. This time, the global vector size was the value patched into the "highest referenced global" field, and so the **globals** command effectively returned, having fulfilled its purpose.

For this mechanism to work properly, an intimate knowledge of the the CLI function and data structures was required, and this knowledge is unfortunately no longer valid when running under the Shell. The **globals** command must therefore be over-ridden by a built-in equivalent, which works in a slightly cleaner, if less machine-independent way. The Shell version of the **globals** command simply allocates a new vector of the required size, and then copies the values from the old global vector into it. A small machine code subroutine is then called, which has the effect of updating the BCPL "G" register (the processor register used for global references) to point to the new global vector rather than the old. The old global vector is then released to the free pool, and execution continues with the new global vector of the required size.

4.6.7 Other built-in commands

The rest of the built-in commands exist because they manipulate Shell data structures, and hence cannot be implemented as loaded commands. These commands fall into three main categories. The general commands deal with the local Shell environment, allowing the command history to be investigated and Shell options to be altered. The directory commands deal with the manipulation of the directory stack mechanism, and allow directory environments to be saved and restored. A description of the general and directory commnds can be found in appendix 1. The third category of command, in other words those which interact with the REX language system, can be found in chapter 5.

4.7 Summary

The TRIPOS Shell is much enhanced version of the old command language interpreter, and incorporates many of the good ideas from the UNIX Shell and C Shell. More importantly, the TRIPOS Shell has tidied up many of the earlier design inconsistencies, and now provides a much cleaner user interface. As far as possible, compatibility between the Shell and the CLI have been maintained, but where this is impossible, built-in commands have been provided to over-ride the old commands which would not work under the new system.

An extension to the TRIPOS syntax has been defined, which provides a simple and convenient way of using concurrency, with pipes being used to connect different tasks. As a result of the change of syntax, a different technique for the execution of foreground and background tasks was developed, and through this it became possible to execute command lines without requiring them to be typed at the console. The command history mechanism uses this technique to keep a set of recently executed command lines, allowing them to be recalled and

edited.

Because of the work on command execution techniques, it became possible for programs to call the Shell recursively. The effect of this is to allow the user to provide a command pre-processor which handles any desired syntax, without the necessity for the whole of the command environment to be re-implemented. It is this interface which the REX command and programming language uses, and more details can be found in chapter 5.

In conclusion, the TRIPOS Shell is, although more complicated than the CLI, more convenient to use, and much more productive for the experienced TRIPOS user. Furthermore, in conjunction with the REX language system and the devices described in chapter 6, it has been possible for the user to take advantage of the more powerful CPU and much larger memories of the new TRIPOS machines.

5. A Command Programming Language

5.1 Introduction

In traditional operating systems, much work has been expended to add simple programming language features to command language interpreters, so as to enable the user to construct command sequences. These languages tend to be unstructured and ugly and provide only very basic facilities, such as the conditional execution of commands and simple variables. If looping constructs are provided, then they are generally of a very crude nature. Above all, the programs written in traditional command languages are virtually unreadable by anyone other than the original author. Examples of this type of command language are those for IBM's Conversational Monitor System [IBM72, Stephenson73] and UNIX [Mashey76, Bourne78, Joy80].

As an alternative to a simple command language, it is possible to use a macro language as a pre-processor for command lines, and this was the approach adopted in MXEC [Ash81]. There is nothing wrong with the idea of using macros to tailor a command environment or to execute groups of commands, but macro languages tend to be even less readable than command languages, and it is difficult to include a sensible control structure which includes conditional execution and iteration.

This chapter describes the REX command programming language, which tackles the problem in a somewhat different manner. Rather than adding programming language features to a simple command language, it turns the problem around, and adds command execution features to a powerful interpretive programming language. By adopting this approach, it is possible to have all the advantages of a language designed for general programming, with the added benefit of having a clean and well defined interface to the command language interpreter, enabling command sequences to be constructed simply and easily. This approach is similar to that taken by IBM's REXX [Cowlishaw84] except that the IBM language is designed primarily for teaching, and is very heavily based on PL/1.

The main aim of this work was to investigate the features which would be required in any language to be used for both command sequences and general programming. The Cambridge TRIPOS environment was ideal for this type of work for two main reasons. Firstly, through the enhancements made to the command language interpreter (described in chapter 4), there already existed a

system which possessed the necessary interfaces for the development of a command language. Secondly, the Cambridge TRIPOS community was large and active, with many sophisticated users who were themselves involved in research. This meant that there were many people who were prepared to try out new ideas, and more importantly, to give intelligent feedback and make constructive criticisms.

5.2 Motivation

The motivation for the work described in this chapter came from two distinct areas.

Firstly, TRIPOS lacked a language in which small "throw-away" programs could be written easily. The term "throw-away" as used here should not be confused with the technique of throw-away compilation [Brown79], where parts of a program are compiled when required, and the compiled code discarded when memory becomes scarce. It is used to describe the type of program, usually very short, which is written quickly for a single application and then discarded when no longer required. The only language available on TRIPOS for such work was BCPL, which although ideal for systems programming, was hardly the right language to use if it was wished to make a program correct first time. It is easy to write nonsensical programs, and there is little in either the language or the run time library to help with such things as string manipulation—one of the facilities which would be required when writing throw-away programs. Also, BCPL is a compiled language, making the "edit, compile, debug" loop fairly slow.

Secondly, TRIPOS lacked a decent language in which to write command sequences. Chapter 3 has shown that, with clever use of the `c`, `t`, `lab`, `skip` and `if` commands, it is possible to write reasonably complex command sequences. Unfortunately, the language is unstructured, difficult to read, and since everything is implemented using disc I/O and loaded commands, it is relatively slow to execute. Even though it is possible to speed up the execution of command sequences by keeping the relevant commands pre-loaded (see chapter 6), this does not help the structure and readability of the language.

5.3 Requirements

Before designing designing any new language, particularly one which must fulfil two functions, it is necessary to investigate the requirements of the language. In fact, even though conventional systems separate command and throw-away programming languages, there is very little difference between the

two types of application, and it is possible to draw up a list of requirements for a language which can satisfy both.

- It must be easy to learn by beginners and experts alike
- It must be possible to write programs quickly
- It must be easy to read by someone other than the programmer
- It must have good input/output facilities
- It must have a good control structure
- It must have good string handling facilities
- It must have a clean and simple interface to the operating system
- Above all, it must be easy to get programs right first time

The aims of the REX project were to design and implement such a language, and then to incorporate it into the TRIPOS Shell in such a way as to make command sequences and throw-away programs easy to produce and fast to execute.

5.4 General philosophy

The list of requirements given above already defines many of the facilities which the language should have. The fact that it should be easy to learn and fast to write in implies that the language must be small, with only a few powerful constructs. The fact that it must be easy to read implies free format input layout, long variable names, character case equating wherever possible, and the ability to use procedures in order to modularise programs. The fact that it must be possible to get programs right first time implies a tolerance of user error which is not normally seen in other programming languages. For ease of use, it should be possible to use variable names without declaring them explicitly, and variables must be capable of having a data type which might change dynamically during the running of the program.

The first major design decision to be taken was whether the language should be compiled or interpreted. With a compiled system, the programs would run fast, but things like error handling and dynamic data type coercion would be difficult to implement, and would require a large support library. Conversely, with an interpreted system, execution speed could be sacrificed in order to improve the ease with which error handling and data type coercion could be implemented. In addition, with an interpreted system, it would be easier to trace the execution of a program, and hence improve the debugging properties. Since

execution speed is not a major issue in either of the two types of language, an interpreted system was chosen, with the added bonus that the interpreter would be portable onto other TRIPOS machines.

5.5 The REX language

The syntax of REX has many similarities to other languages, in that it has variables, arrays, procedures, conditional statements, loops and so on. Many of the facilities provided are new, and these are discussed in detail later in this chapter. The language itself is described fully in appendix 2.

As an indication of the power of the REX language, included here are two short examples showing how it can be used. The first example is a command sequence, and corresponds to the rather ungainly example in chapter 3. The second example is less practical, and shows how the classic "Towers of Hanoi" problem can be programmed in REX. Although they are both rather contrived, they give an insight into what REX programs actually look like, and what they capable of in practice.

5.5.1 Simple command sequence

```
$  Example 1:  Command sequence from chapter 3

proc run( command ):
    obey command
    if  rc ~= 0  then
        exit( rc, command )
    fi
corp

if  ~rdargs( args, "file/a", file )  then
    exit( 20, "Bad arguments:" args )
fi

if  mctype = "lsi4"   |  mctype = "68000"  then
    run( "bcpl bcpl."||file "to" mctype||"."||file )
    run( "link" file )

    say "Compilation and linking complete"
  else
    exit( 20, "Unknown machine type:" mctype )
fi
```

5.5.2 Towers of Hanoi

```
$  Example 2:  Towers of Hanoi

proc move( disc, peg1, peg2, peg3 ):
    if  disc > 0  then
        move( disc-1, peg1, peg3, peg2 )
        say "Move disc" disc "from peg" peg1 "to peg" peg2
        move( disc-1, peg3, peg2, peg1 )
    fi
corp

move( 4, "A", "B", "C" )
```

5.6 Language issues

On the face of it, the REX language appears very similar to many in the ALGOL family. This is true to a certain extent, since the purpose of this research was not to investigate syntax design *per se*, but more to look at the way in which the language was used in practice, and what facilities it must have in order to meet its requirements. Because of that, relevant keywords and constructs have been adopted from other languages—for example, "`break`" and "`loop`" come from BCPL, "`skip`" from ALGOL68, "`dim`" from BASIC, and "`||`" from PL/1.

Apart from the obvious superficial similarities with other languages, REX introduces many new features which were designed specifically for throw-away programming and the writing of command sequences. It is in these areas where REX diverges from both command and programming languages, and it is the principal differences which are discussed in this section.

5.6.1 Data typing

REX is, like many other programming languages, strongly data typed. Languages such as ALGOL68 [VanWijngaarden76] , PASCAL [Wirth74] and so on insist that every variable carries with it an associated data type, the usage of which is checked for consistency by the compiler, and faulted if there is a data type mismatch. In such orthodox languages, it should be noted that all data type checking is performed at compile time, with each object's data type being fixed once declared. In the case of arrays, all elements of the array are assumed to have the same data type, and the concept of *records* is needed to encompass array-like structures which employ a selection of different data types.

REX differs from conventional languages on all these important points. Firstly, variables do not have to be declared explicitly—they simply come into existence the first time they are given a value. Secondly, once a variable has been given a value (along with its associated data type), not only may the value of the variable change throughout the running of the program, but the data type may change also. Thirdly, the data type of an array variable is simply *array*, not *array of X*, which is what would be required in other languages. Each element of an array acts in an identical manner to a simple variable, being capable of taking on any value and data type. Fourthly, since the data type of variables, array elements and so on, are not actually known until run time, all data typing and coercion is done then, rather than when the program is loaded from disc.

REX has eight possible data types, each of which are used in certain specific circumstances. The data types are:

- *number*
- *string*
- *bool*
- *proc*
- *array*
- *input channel*
- *output channel*
- *nil*

Some of the values are capable of being coerced into other data types with little problem—the number 1234, for example, can easily be be coerced into the string "1234", and vice versa. Other data types are more specialised, and cannot be coerced so easily. For instance, there is no obvious way of coercing *input channel* to anything else.

The internal representation of each of the data types is quite heavily determined by the underlying BCPL implementation language. The precision of objects of type *number*, for instance, is determined by the size of the BCPL word—32 bits on a 68000. Similarly, REX strings are implemented internally as BCPL strings, and hence are represented as blocks of bytes in memory, with the first byte holding the string length. The maximum length of objects of type *string* is therefore determined by the size of the BCPL byte—normally 8 bits, giving strings of up to 255 characters. Objects of type *bool* are simply represented by the BCPL truth values `true` and `false`.

The other data types, excepting *nil*, which has no associated value, are represented as pointers to more complicated data structures. Data type *proc* is represented by a pointer to a procedure definition block, which holds pointers to the list of formal parameters for the procedure, and a pointer to the main procedure body. Data type *array* is represented by a pointer to an array control block, holding the dimensionality of the array, the subscript lower and upper bound values, and pointers to the constituent array elements. Finally, data types *input channel* and *output channel* are represented by pointers to I/O control blocks, which hold flags giving the channel's type and status, and a pointer to the BCPL stream control block representing the I/O stream itself.

5.6.2 Arrays and array elements

Arrays in REX are simply matrices of up to five dimensions, where each element of the array is entirely independent and acts in exactly the same way as a normal variable. This means that each array element has its own associated value and data type, which can change dynamically throughout the running of a program.

Arrays themselves are handled in a rather unconventional manner. In orthodox languages, allocating an array of, for example, 100 by 100 elements, would cause 10,000 units of memory to be allocated, where each unit was capable of holding an object of the array data type. REX approaches the problem somewhat differently, in that it works on the assumption that most arrays are either very small or very sparse, and that there is no point in allocating storage for array elements which are not going to be used.

Because of the decision to use sparse arrays, the strategy adopted for accessing array elements cannot simply be direct lookup, with the address of an element being calculated by multiplying the relevant dimension bounds and array subscripts. The method used in fact is one of hashing. On each array access, first the dimensionality of the array is checked against the number of subscripts given, and an error flagged if there is no match. Next, each of the subscripts is compared in turn with the lower and upper bounds of the relevant dimension, and if out of range, the access is rejected. Having validated the number and values of the subscripts, a hashing function is applied to them, and the resulting value is used to index a hash table of array elements. Each array element with the same hash value is linked on a chain from the hash table, and this chain is then scanned for an element with exactly the right subscript values.

The size of the hash table is set according to the number of elements in the array. If the number of elements is less than some pre-defined maximum

(currently 100), then a hash table of that size is chosen. Arrays with greater than this number of elements have a hash table of the maximum size. It should be noted here that, since only those array elements which are actually defined have an entry in the hash table, access speed is defined not by the size of the array, but by the density of its utilisation. Obviously, if a 10,000 element array had all of its elements defined, then the chains of array elements hanging from the hash tables would, on average, be 100 items long. Access time would indeed be slow on such an occasion, but it is doubtful whether a program which required such facilities should be written in REX anyway. In practice, arrays are typically either small or sparse, and so the problem does not arise. For most applications, where the number of defined elements is less than the maximum size of the hash table, the correct array element is usually found as the only item on its hash chain.

It is arguable that, for small arrays, it would be better to arrange to use a linear vector rather than a hash table, as this would improve performance. By observation, it was found that the majority of array access time was being taken not in chaining down the hash table list, but in checking the dimensionality and values of the array subscripts before chaining even began. These checks would be just as expensive, no matter what the method used to access the array data structure, and so the single hashing method remained the one used.

5.6.3 Records and other data structures

One of the principal differences between REX and other data typed languages, is that individual array elements can each have a different data type. This fact makes REX arrays much more like multi-dimensional *records* in a conventional sense, and like records, REX arrays can be used to build arbitrary data structures. The fact which makes this possible is that the `dim` and `table` statements each have the property of allocating a *new* array from the heap. Since array pointers can be copied by using assignment statements, or by returning them as results from procedures, `dim` and `table` can be treated exactly like the "new" operation in PASCAL, or the "HEAP" generator in ALGOL68.

Since each element in an array can have any value or data type, there is nothing stopping arrays from referencing themselves or other arrays, in order to create chains, trees, directed graphs and so on. The following fragment of code creates a chain of LISP like nodes, where the first node element is a number, and the second node element is a pointer to a chain of other numbers smaller than it.

```
proc makechain( number ):
    if  number > 0
        then  result  table number, makechain( number-1 )
        else  result  nil
    fi
corp

chain  :=  makechain( 10 )
```

Just as chains can be created and manipulated easily, so can trees and other complex data structures. The following fragment of code prints out a tree of sorted items in ascending order.

```
proc printtree( tree ):
    if  tree = nil
        then  return
        else  printtree( tree[ 1 ] )
              say tree[ 2 ]
              printtree( tree[ 3 ] )
    fi
corp

printtree( root )
```

In the above examples, it has been shown that it is easy to create relatively complex data structures, and to manipulate them in a simple and clean manner, just by using arrays. The user is unaware of the problems of storage allocation and pointer arithmetic, and since all data type and array bound checking is performed at run time, the interpreter can pick up any error state, such as following a non-existent pointer or using an array subscript which is out of bounds.

5.6.4 The necessity of NIL

The data type *nil* is necessary because there must be some way of representing the "null pointer", and other special values. It would have been possible to implement null pointers in a different way, simply by re-defining the equality operators "=" and "~=", so that they would work for objects of type *array*, as well as for numbers and strings. In this way, the user could easily define an array which he uses to represent the null pointer, and then using the newly extended comparison operators, check for this special value.

This solution was not adopted, since there are several other occasions when special values are required, usually to indicate some sort of error condition, and in those cases it would not be quite so easy to have a user defined value. One example of this is the result returned by the I/O functions openin and openout,

if they fail to open the file given to them. Using the same principle as put forward for the *array* case, it should be possible to enhance the equality operators still further, so that they work for data types *input channel* and *output channel* as well. This is of little use though, since there is no simple way of generating the special value with which to do the comparison later. The only way it could be done would be to have two variables, one an *input channel*, and the other an *output channel*, referring to streams, the opening of which is guaranteed to fail. This is not only untidy and inelegant, it is also prone to error, since operations very rarely fail when this is required of them!

The solution adopted is much more general, and applicable to the two cases outlined above, along with more besides. If a new data type, *nil*, is created, for which the equality operators are defined, then it is possible to provide null pointers and other special values easily. The major difference between *nil* and other data types is that *nil* has no associated value, and the comparison performed by the equality operators is a comparison of data type, not of data value. *nil* can, therefore, safely be compared with any object of any data type, and will only return `true` when compared with itself. The `openin` and `openout` functions return *nil* when the "open" operation fails, and the `rdargs` function returns *nil* for undefined data values, to distinguish these from the null string. *nil* is not only necessary in order to implement REX data structures properly, but also simplifies many of the other situations when a special value is required.

5.6.5 Procedures

One of the design requirements of REX was the ability to write readable, modular programs. One way of improving both readability and modularity, and at the same time, increasing the power of the language enormously, is through the use of procedures. As with the data type and array mechanisms described earlier, REX implements procedures in a rather unorthadox way.

In other strongly typed languages, such as PASCAL or ALGOL, because the data type checking is performed at compile time, it is necessary for the syntax of these languages to reflect, when a procedure is defined, whether the procedure returns a result, and if so, what the data type of that result is. This introduces the distinction between subroutines and functions which all major programming languages have. Apart from unchecked languages like BCPL, it is impossible normally to treat a procedure on one occasion as a subroutine, and on another occasion as a function.

REX removes the distinction of subroutines and functions from the declaration of a procedure—all procedures have data type *proc*, and are declared using the

proc statement. There are three ways of returning from a procedure. It is simply possible to complete execution of all the statements in the procedure, at which point the procedure returns without passing back a result. Explicit return from a procedure is obtained by using the **return** or **result** statements. **return** has the same effect as before, in other words, returning to the caller without passing back a result. **result**, on the other hand, takes as its argument an arbitrary expression, which is evaluated and passed back to the caller, along with its associated data type.

Because of the fact that procedures may or may not return a result, procedure calls are valid, either as simple statements, or as parts of expressions. When a procedure call is made, the REX interpreter makes note of the context of the call, which defines whether a result is required or not. On return from the procedure, either implicitly or explicitly, the interpreter compares the requirement for a result against the fact of whether or not a result has actually been provided, trapping the error condition if there is a mismatch.

5.6.6 Parameter passing and variable scope

When a procedure call is made, each of the parameters in the parameter list is evaluated, and the value and data type are assigned to the corresponding formal parameter variable. A check is made at this point that the number of parameters given matches exactly the number of parameters expected, and if not, the operation is aborted. As with the definition of the procedure itself, there is no necessity to define the data type of the formal parameters in the syntax of the declaration, since they will be taken dynamically, depending on the values and data types of the actual parameters.

All variables in REX which are defined, are given entries on the *variable stack*. When executing the main part of a REX program, all variables share the root stack. When a variable is used in an expression, the stack is scanned from top to bottom, and if the relevant variable is found, its value and data type are returned. If the variable is not found on the stack, then it does not have a defined value, and hence the condition should be flagged as an error. When updating a variable, either explicitly by using an assignment statement, or implicitly by using **prompt**, **read** etc., again the variable stack is scanned in a top to bottom manner, and if the variable is found, its value and data type are updated accordingly. In this case, however, it is not an error to reference an undefined variable, since it is through assignment that variables become defined in the first place. If a variable does not already have an entry on the variable stack, then a new one is added at the top of the stack, in which is stored the new

value and data type.

In this way, variables become defined as they are used, and the amount of time taken to look up the value of a variable is proportional to the number of items on the variable stack, which for short programs is fairly small. All variables defined on the root stack, therefore, have global scope while executing anywhere in the main program, with the only restriction being that a variable must be defined before it is used. Note that, because of the lack of a variable declaration syntax, it is impossible for `if` and `do` clauses to have variables of local scope, but owing to the nature of programs written in REX, in practice this is not a restriction.

When it comes to procedures, the variable stack is handled slightly differently. The current top of the variable stack is stored internally by the interpreter, and this point now becomes the base of the ***procedure stack frame.*** Each of the formal parameters of the procedure is then assigned the value and data type of the corresponding actual parameter, the effect of which is to create new items on the stack for these variables, within the current stack frame. If one of the formal parameters has a name identical to a variable defined in a previous stack frame or the root stack, then the previous entry remains untouched.

During the running of the procedure, variables used in expressions are handled in exactly the same way as before. The entire stack is scanned from top to bottom, and the first entry found which matches the variable is the one used. In this way, formal parameters are found before any variables in previous stack frames which might have the same name. Dynamic free variables, in other words, variables which have been defined in previous stack frames, are available to this procedure, and because of the way the variable stack is scanned, the effect is what one might expect, with the most recently defined incarnation of a variable being the one actually used.

It should be noted here that this mechanism is different to the one adopted by languages which perform data type and variable scope checking at compile time. In languages such as PASCAL and ALGOL, the scope of dynamic free variables is set during compilation, and is restricted by the nesting of procedure declarations. Unless one procedure is declared as being local to another, there is no way that the variables of the outer procedure can be touched by the other. The only exception to this rule is the "main program", which acts as though it were a large procedure within which all others are declared. The result is that variables declared within the main program are therefore accessible to all other procedures, and therefore have the effect of being global.

The mechanism adopted by the REX system is rather different, and is much more akin to the LISP *association list* strategy. The scope of variables is defined, not at compile (or in this case, syntax analysis) time, but when the program is actually executed. The result of this is to make the scoping of variables dynamic, and depend on the order that procedures are called, rather than the way in which they are declared. This is subtly different to the standard practice, and is only possible, as in LISP, because all the variable declaration and data type checking are performed at run time.

Assignment to variables while within a procedure is handled in a rather different manner. Formal parameters to the procedure are defined as being local to that procedure implicitly, since new entries for them are always created at the base of the new stack frame. Given that REX has no primitives to enable variables to be declared explicitly (rather than simply being assigned to), only the formal parameter variables would be local in scope and could be updated without fear of corrupting variables defined in previous stack frames. In order to alleviate this matter, when a variable has a value assigned to it within a procedure, rather than scanning the entire stack to find a variable entry, only the current stack frame is scanned, with a new entry being added at the top of the current frame if necessary. In this way, any variable which is assigned to within a procedure, either implicitly in the case of formal parameters or explicitly in the case of assignment statements, is always local to that procedure, and hence cannot corrupt variables of the same name defined in previous stack frames.

From the user's point of view, this mechanism is simple to use, and is almost always what is wanted. He sees it as though the most recent version of all defined variables are copied into the new stack frame whenever a procedure starts, and from then on, all variable access within that procedure is local to it. Since actual parameters are evaluated before being passed to a procedure, and altering the formal parameters while in the procedure affects only the local environment, the effect to the user is that of simple "call by value", with read only access to all dynamic free variables.

Because of the strategy employed here, it is not possible to have global variables, since each variable acts as if it were local to its particular stack frame, including those defined on the root stack. There are, in fact, only two ways in which procedures can affect the external environment. Firstly, they can of course return a result to the caller, and no restriction is placed on the value and data type of this result. Secondly, they can update external array elements.

Arrays are represented simply as pointers to array control blocks, and so finding the root of an array is simply a matter of looking up the array name in

the variable stack—a read only operation. Having once found the root of the array, elements of the array can be updated easily, and since an array element update does not go through the "variable stack" mechanism, the side effects are not restricted to the local environment. The following code fragment shows how this mechanism can be used in very much the same way as the BCPL global vector:

```
dim globals[ 150 ]
result2  :=  10

proc procedure( a, b, c ):
    ....
    ....
    globals[ result2 ]  :=  200
    result 100
corp

res1  :=  procedure( 1, 2, 3 )
res2  :=  globals[ result2 ]
```

Alternatively, since it is possible to set up single dimension arrays which have defined values (see the `table` statement), it is easy to arrange that more than one result is returned from a procedure, without resorting to side effects. Using this method, the above example becomes:

```
proc procedure( a, b, c ):
    ....
    ....
    result  table 100, 200
corp

res   :=  procedure( 1, 2, 3 )
res1  :=  res[ 1 ]
res2  :=  res[ 2 ]
```

5.6.7 Dynamic type coercion

One of the factors which makes REX such an easy language to write programs in is that, if the wrong data type is used in a particular context, then the interpreter attempts to coerce the value given to the correct data type, so as to avoid an error situation if this is at all possible. Take, for example, the following code fragment:

```
prompt "How many types round the loop?": howmany

for i to howmany
do
    ....
od
```

In this example, the **prompt** statement is used to print out a message to the main Shell output stream, and then read the user's response into a variable called howmany. The data type of howmany at this point is *string*, since there has been no interpretation of the characters typed by the user. When **howmany** is used as the argument to **to** in a **do** statement, the data type required is not *string*, but *number*. Since this fact is obvious from the context, the interpreter attempts to parse the string as a number, using the same syntax as for REX source code. If the coercion is possible, in other words, the user has typed in a valid number, then the coercion takes place implicitly, and the **do** statement is obeyed normally. Only if the user's response to the **prompt** statement is invalid will the type coercion system complain.

Examples such as this are common in the writing of REX programs, and it is very rare that the user actually has to know what data type a variable or an array item has at any particular moment. There are, however, cases where the dynamic type coercion mechanism may have totally the wrong effect, and to this end, several functions are provided which enable the user to perform explicit data type coercion. Before describing the functions themselves, it is worth looking at an example where the implicit data type coercion system is inadequate, and where user intervention is required.

The problem arises primarily from the relational operators, which are defined for both numeric and lexicographic comparisons. If both arguments to a relational operator are of the same data type—both *number* or *string*—then it is obvious which sort of comparison should be done. If, however, the data types are mixed—one *number* and one *string*—then it is not clear what should be done. Should the *string* be coerced to *number*, and a numeric comparison done? Alternatively, should the *number* be coerced to *string*, and a lexicographic comparison done? In order to avoid this ambiguity, the functions **num** and **str** are provided, in order that the data type of a particular variable or expression can be determined explicitly by the user, so as to obtain the desired effect. The relational operators are defined such that, if either of the operands is of data type *number*, then the other is also coerced to *number*, and a numeric comparison is done. If not, then both operands are coerced to data type *string*, and a

lexicographic comparison is done. Consider the following code fragment, which illustrates the need for explicit data type coercion:

```
prompt "Type two numbers:": a, b

if  a < b  then  say "a < b"  fi
if  num(a) < num(b)  then  say "num(a) < num(b)"  fi
```

When the two variables a and b are initially defined by the prompt statement, they both have data type *string*, and so in the first if statement, a lexicographic comparison is done. In the second if statement, the operands are explicitly coerced to data type *number* before the comparison is done, and so the operation performed is numeric. If the two values entered were "5" and "10", then the ambiguity is clearly seen—5 is less than 10 numerically, but not lexicographically, and so, only by using explicit coercion will the correct result be obtained.

Along with the functions num and str described above, there are four others which are provided for similar purposes. The ustr and lstr functions are exactly equivalent to str, except that the coercion involves, not only a conversion to data type *string*, but the upper or lower casing of the consituent characters as well. These functions could be written in REX, but were included as built-in functions for efficiency. The chr and asc functions were added for a rather different reason, and provide services which could not be written in REX itself. They are exactly analogous to the BASIC functions of the same name, in that they provide a mapping between single characters and their ASCII code representations. Characters in this context are objects of type *string* and length one, such as "a". These functions are not strictly needed, but enable the REX programmer to use the special facilities available in his terminal, which are only accessible through using control characters and escape sequences.

As an alternative to having multiple data types and providing coercion between them, it is possible to have just one data type—the character string—and perform all other operations using it. This is the approach adopted by REXX [Cowlishaw84] where all variables are capable of holding strings of potentially infinite length. Before any ambiguous operation (such as a comparison) can be performed, the operands must be parsed to see which type of operation is appropriate. This removes the problem mentioned above where numbers can be compared wrongly, simply because they are being held in variables with the wrong data type. It does, however, add a distinct overhead to every arithmetic operation, since there must be a conversion from the string representation of a number, each time it is used in a numeric context.

5.6.8 Error recovery and debugging

Even though REX is designed to be an easy language to write in, it is inevitable that mistakes will be made. Simple syntax errors (such as the omission of "then" after "if") are corrected as a program is loaded from disc, and a warning message is printed when this happens. More serious syntax errors cause the program to be rejected, again with messages being printed out giving the probable cause of the problem. Given that REX is a dynamically typed language with full run time checking, the majority of errors are detected not as incorrect syntax, but as undefined variables, array subscripts out of bounds, type coercion problems and so on.

Whenever a run time error is detected, a message is printed out indicating the cause of the error and giving the line number in the original source where the error occurred. The set of statements being executed is abandoned, and the interpreter starts again at the next well defined point in the user program. This means at the head of a loop, or at the end of a program block such as a procedure or an "if" statement. If no sensible error recovery point exists, the program is aborted.

The REX interpreter also has two tracing modes, which enable an erroneous program to be debugged. The first mode, designed primarily for the debugging of command sequences, simply traces the execution of "obey" statements, printing out each command line as it is passed to the Shell for execution. Combined with the Shell option to inhibit the execution of command lines, a command sequence can be checked quickly so as to find out where it is in error. The second mode is designed for the tracing of whole REX programs, and before executing each statement, the interpreter prints it out along with the source line number. Because there is little overhead in loading a REX program again, the debugging loop is fast, and with the two tracing facilities described above, the effort involved in correcting a program is usually very small.

5.7 Implementation issues

The previous section discussed some of the language issues which make REX different from conventional languages, and which help to make it easy to learn and simple to use. The following section looks in more detail at the REX interpreter itself, and examines how it works, and how it is integrated into the TRIPOS Shell environment.

5.7.1 Representation of REX programs

That which has been described so far is the representation of REX programs as files on disc, which is how they are perceived by the user. In order for a program to be interpreted, it must first be read into memory and parsed, before it is in a fit state for the interpreter to use.

Since REX is essentially an ALGOL-like block-structured language, the most convenient way to parse it is by means of recursive descent syntax analysis, resulting in the *applicative expression* or AE tree, and it is the AE tree representation which the interpreter uses to execute the program. Normally, using an AE tree is a bulky way in which to store a program, and the tendency is to "flatten" it into some more compact, linear representation. Since it is assumed that memory is in plentiful supply, and the typical REX program is less than 100 lines long, the amount of store wasted by using the AE tree rather than a linear code is negligible.

The advantages of using the AE tree as the structure for interpretation are twofold. Firstly, since the input format for the interpreter is the output format for the syntax analyser, the extra translation stage is removed from the loading process. Secondly, tree-walking techniques such as those employed by optimising compilers and so on, are well understood, and the AE tree structure enables the recursive nature of the interpreted language to be implemented conveniently using the recursion facilities of the interpreting language. Constructs such as do loops, `if` statements and so on, can easily be represented at interpretation time by a procedure call with local variables. The nesting of such statements is possible simply through the use of recursive procedure calls.

The overhead of loading a REX program is, on the whole, fairly small, with the majority of the time being spent in opening the file containing the source. However, once loaded, there is little point in unloading a REX program unless memory is particularly scarce, and so programs are loaded once when first referenced, and from then on, interpreted directly from memory, with the initial loading operation being omitted.

5.7.2 Storage allocation and garbage collection

As with any other large, complex program, the REX system requires the use of dynamic storage allocation. The implementation language, BCPL, provides only the very primitive *getvec* and *freevec* functions, and does not have any sort of garbage collection mechanism. Also, in its standard state, storage allocation under TRIPOS can be a slow business in which the whole store chain is scanned for every piece of storage allocated (see chapter 7). It is therefore reasonable for

the REX interpreter to provide its own dynamic storage allocation scheme, along with any necessary garbage collection mechanism.

There are three main areas in which the REX interpreter requires dynamic storage, each of which have very different requirements, and are handled in different ways. They are:

- Programs
- Arrays
- Strings

The REX program is, as explained before, represented in memory by its AE tree. The memory from which this tree is built is allocated from the *program pool* by the syntax analyser module, as and when it is required. The property of program storage is that it is allocated in very small units—typically 4 or 5 words—and once allocated, not freed until the whole program is unloaded. In order to speed up allocation of program memory, chunks of 1,000 words are allocated from the system pool, with the program nodes being allocated from these chunks on demand. The fact that the nodes cannot be freed individually is of no consequence, since this never happens in practice, and because of the relatively slow access to the system pool, allocating memory in larger units speeds up the syntax analysis operation.

Arrays are created dynamically during the running of a REX program by the `dim` and `table` statements. In order to be able to build data structures from within REX, arrays once allocated, must not be freed, except when no references to them exist, in other words, when they become garbage. This only happens when an array pointer is over-written by another value during the running of a program, which in practice is virtually never. On observing techniques adopted by other REX programmers, it was found that arrays tend to be defined at the beginning of a program, have their elements updated during the running of a program, and are not discarded until the run is over. Because of this fact, array memory is allocated from the *array pool*, which is handled in a similar way to the *program pool*. Large chunks are allocated from the system pool, which are then subdivided and allocated to array items on demand. On completion of the REX program, all the memory in the array pool is released back to the system, unlike program memory which is kept until the program is unloaded.

REX strings, on the other hand, are an entirely different matter. Because of frequent use of the juxtaposition and concatenation operators, combined with the way that string parsing and other string manipulation functions work, strings are often generated implicitly as part of simple expression evaluation, and then

discarded as garbage later. All this happens behind the user's back, and he is totally unaware of the amount of string garbage being created on his behalf, over and above that which is is creating himself, simply by over-writing string variables or array elements.

Since REX strings are defined to be implemented internally as BCPL strings which have a fairly short maximum length, it is a reasonable proposition to allocate fixed size string buffers, each of maximum length, which come initially from the *string pool*. Each string in a REX program is marked as being either *temporary* or *permanent*. Temporary strings are those which are yielded as intermediate results from string concatenation operations and so on, and are not referenced elsewhere. Permanent strings, on the other hand, are referenced either from variables on the variable stack, or from array elements. Whenever a variable or array element is updated, the item being over-written is checked to see if it is of data type *string*, and if so, the string's state is changed from *permanent* to *temporary*. Similarly, the object which is being written is checked, and if it is of data type *string*, its state is changed from *temporary* to *permanent*.

Before each new REX statement is executed, the list of allocated strings is scanned, and those found to be in *temporary* state are moved from the "allocated" chain to the "free" chain. During execution of the statement, whenever a string buffer is required, it is first allocated from the "free" chain (if there is one spare), and from the main string pool if not. The string is then added onto the "allocated" chain, and execution continues. This kind of garbage collection must be done synchronously, since at times other than between execution of statements, it is impossible to judge whether an allocated *temporary* string is actually referenced or not, since it may still be in use as part of some intermediate result.

5.8 Interface to the Shell

The feature of the REX language which makes it ideal for writing command sequences, as well as just simple throw-away programs, is its interface to the command environment via the Shell. The interface as seen by the user is extremely simple, with just two statements affecting the Shell, and a handful of "system" variables, which allow access to the command environment.

The most useful feature of the interface between the Shell and the REX system is that it is bi-directional, with either program being capable of calling the other via the interface module REXSHELL. Not only can the Shell and REX programs call each other, but they can each call themselves recursively using the same

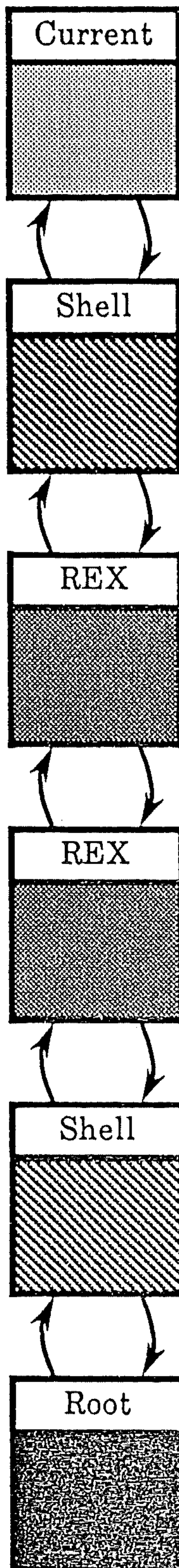
interface. The effect of this is that, at run time, there is no discernable difference between loaded commands and REX programs, with either being capable of being in control at any one time. Using the recursion facility to save and restore environments, it is possible for loaded commands to execute other loaded commands or REX programs, and for REX programs to execute loaded commands or other REX programs, with the depth of recursion being restricted only by the amount of memory available.

In order to implement this recursion, it is clear that some sort of stack mechanism is required. Using a single program stack is insufficient, since its size would either have to be set to that required by the maximum recursion depth, or would have to be expandable dynamically. One way of obtaining the effect of an expanding stack is to use the BCPL coroutine mechanism, with each coroutine representing a new "frame" in the calling sequence. Figure 5.1 shows the three different types of coroutine used, and their relationship to one other. In each case, the environment of the current execution level is saved on the coroutine stack, before the next coroutine is called. On return from an execution level, the coroutine used to implement that level is deleted, and the old environment is restored from the previous coroutine stack. The structure can therefore be imagined as a stack of environments, where each environment is represented by a coroutine stack. This technique is discussed further in chapter 7.

5.8.1 The OBEY statement

One of the great benefits of the work done previously on the command executor task, pipe expressions and command histories (see chapter 4), was that the problem of splitting a command line from its input stream had already been tackled. Indeed, the Shell already possessed the necessary procedural interface to enable free standing command lines to be executed in a given environment, as this was required as part of the command history and pipe expression mechanisms.

This interface was designed with the possibility of adding a sophisticated command language to the Shell at some later time, and hence fitted exactly the interface required by the REX "obey" statement. The effect of obey is very simple—its argument is coerced to data type *string*, and then treated as a command line, being passed to the Shell for execution in the current Shell environment, exactly as if the command had been typed directly at the console, or extracted from the command history. The command line is then executed, with control being returned to the REX program as soon as the command completes execution.



Current Coroutine

Either a Shell or a REX coroutine, depending on whether a loaded command or a REX program is executing. This coroutine can call another of either type before it returns.

Second Shell Coroutine

Called from the second REX coroutine via an "obey" statement as before, but this time causing a loaded command to be executed.

Second REX Coroutine

Called from the first REX coroutine. This represents a REX program executing an "obey" statement, causing another REX program to be executed.

REX Coroutine

Used to handle the execution of a REX program. All information local to the REX program is held in this stack.

Main Shell Coroutine

Base of the Shell calling structure. Used to store the Shell environment whenever a loaded command or REX program is executed.

Root Stack of the Shell Task

Allocated by the kernel when the Shell is created, with a small fixed size. Deallocated only when the Shell Task is deleted.

REX/Shell Coroutine Stack Structure

Figure 5.1

The interface is also general enough that it is possible for loaded commands to call their parent Shell recursively, in order to execute further commands or start up new Shell sessions from within other programs. An example of this is the TRIPOS MAIL system [Wilson84] which allows the user to switch trivially between the MAIL environment and the Shell environment, without the necessity of having a separate Shell task and selecting it explicitly.

5.8.2 The QUEUE statement

Since many programs prompt the console for parameters which are not given on the command line, a facility is provided to enable input lines to be enqueued in such a way that they are read by these programs directly, without the necessity for them to go to the console. The **queue** statement works asynchronously, adding the line given as its argument to the queue of lines which are waiting to be read by the main Shell input stream.

As with the command executor task and Shell foreground command execution, the effect works because of the generality of the TRIPOS stream handling mechanism. The stream control block for the queued input stream has the following structure:

buf	Buffer containing the current input line
pos	Character position in line buffer
end	Number of characters in line buffer
func1	Special *replenish* function
func2	Zero (no *deplete* function)
func3	Special *close* function
arg1	Pointer to real (unqueued) Shell input stream
arg2	Pointer to queue of input buffers

The queueing mechanism works through the special *replenish* function, which is called whenever the line buffer is empty, and the Shell requires further input characters. The *replenish* function first looks at the queue of buffers to see if there are any, and if so, copies the contents of the first queued buffer into the line buffer, unchains it from the queue and returns, giving the effect that the Shell had read the queued line as though it had been typed at the console. If no queued lines are available, then the *replenish* function of the real Shell input stream is called in order to refill the line buffer.

It is impossible for the Shell to tell whether the characters it is reading have been typed at the console by a user, or added onto its input queue by a REX

program. This mechanism is useful not only for queueing ***data*** lines which are to be read by loaded commands, but for queueing ***command*** lines which are to be read by the Shell as though they had been typed directly. It should be noted that the obey statement works synchronously, thus avoiding the queueing mechanism, and enabling commands to be called from a REX program which can then read any queued input lines. On the whole, it is the case that **queue** is only used to enqueue data lines, but it is a rather pleasing side effect of the mechanism that enqueued command lines work as well.

The ability to enqueue a set of lines to be read by a command language interpreter or the programs which it loads is not new, and was used in the IBM EXEC [IBM72] language and its derivative [Stephenson73]. The mechanism used by EXEC is slightly different though, in that the lines are kept on a LIFO stack rather than a FIFO queue. Both mechanisms have their disadvantages. The stack is confusing for the user, since the lines must be stored in the reverse order to that in which they will eventually be read. The queue is not confusing, since lines are stored in the natural order, but the mechanism is prone to error. This is because it is impossible to guarantee that the queue is empty when a line is added, and so any program executed synchronously may pick up data lines which were not intended for it.

5.8.3 I/O and the EX: device

I/O from within a REX program is possible in one of two ways. Firstly, there are the "interactive" statements, **say** and **prompt**, which work by using the main Shell channels, and can be used for all console I/O. Secondly, there is the more general mechanism of being able to open arbitrary files for input or output, and then performing I/O on these files. The **openin** and **openout** functions are provided for opening files, and these correspond exactly to the *findinput* and *findoutput* functions of the underlying BCPL system. The form of the arguments to openin and openout therefore depend on what is required by the BCPL system—on TRIPOS, file names are used.

All input and output is performed on a record by record basis, with the splitting of BCPL streams into their individual records being done in a machine independent manner by the REX interpreter. Additionally, when reading a record from an input channel, there is the option to apply the **parse** operation on the line as it is read, allowing individual words to be extracted. When the BCPL end of file marker *endstreamch* is encountered, a flag is set in the relevant I/O channel control block, and this flag can be tested using the **eof** function.

Being able to open arbitrary files for I/O means that the full power of TRIPOS pseudo devices is available to the REX programmer. The most useful of these devices by far is "EX:" (see chapter 6), which enables filenames to be extracted from their parent directories. Using this device, it is possible to write REX programs which act on some or all of the files within a directory. As an example of this facility, it is possible to re-write the TRIPOS command scratch—a program which deletes all the files within a directory—in REX.

```
$  Tripos SCRATCH command

if  ~rdargs( args, "dir", dir )  then
    exit( 20, "Bad args" )
fi

if  dir = nil  then  dir  :=  "t:"  fi

stream  :=  openin( "ex:" || dir )

if  stream = nil  then
    exit( 20, "Directory" dir "not found" )
fi

until  eof( stream )
do
    read stream: file
    obey "delete" file
od

close( stream )
```

5.8.4 The system variables

In order to provide access to certain TRIPOS and Shell parameters, it is necessary for REX to have system variables. There are five of these:

- rc
- mcname
- mctype
- systype
- args

The variable rc always evaluates to the current Shell return code, which is set by loaded commands when they call the BCPL procedure *stop*, and by REX programs when they call **exit**. The return code has data type *number*, and can

be treated as a number in all calculations. It can trivially be checked after an **obey** statement, in order to detect whether the command line executed successfully. The variables **mcname**, **mctype** and **systype** correspond exactly to fields in the TRIPOS "information" data structure, which gives details of the local machine environment. They are all of data type *string*, and refer to the current machine name, machine type and system type respectively. Given that these variables are special (being operating system dependent), they are handled by procedures in the REXSHELL module, so as to maintain portability.

The **args** variable is set up by the REX interpreter to be of type *string* and to have the value of the arguments given by the user to the REX program being obeyed. There are two different facilities provided for the user to interpret the arguments string. Firstly, the **parse** statement splits up strings into "words" in very much the same way as the UNIX Shell. Secondly, **rdargs** enables strings to be parsed in a more complex way, with a generalised pattern being used to split strings into keywords and arguments.

5.8.5 Execution of REX programs from the Shell

The Shell provides the ability to execute REX programs implicitly, or to interact directly with the REX system. It has already been described in chapter 4 how the Shell searches for loaded commands in a series of directories. This is not in fact the whole story, since the Shell can also handle the loading and execution of REX programs.

The new sequence of events is as follows. When a command name is typed in, the REX system is called to find out whether this is the name of a REX program which is already loaded in memory—if so, then the REX program is entered. If not, then the current directory, COM1:, system command directory and COM2: are all scanned in sequence, in an attempt to load the command name as a program. If this fails, then the current directory and REX: are searched to see if the name corresponds to a REX program. Only if that fails is the whole search abandoned.

5.9 Portability and the REXSHELL module

There are two levels of portability which are worth investigating here. Firstly, the portability of programs written in the REX language, and secondly, the portability of the REX language system itself.

Programs written in REX vary in portability depending on how much of the operating system interface is used. For programs which do not perform file I/O and do not execute command lines, the only factors which affect portability are the precision to which arithmetic calculations are done, and the maximum length

of string variables. For those programs which do use the operating system interface, the form in which arguments to the **openin** and **openout** functions are given varies depending on the underlying BCPL run time library and the syntax of file names, and the nature of command lines, return codes and so on, depends on the nature of the operating system. These problems do not arise between different TRIPOS systems though, and so provided no unreasonable assumptions are made about numerical precision or the maximum length of strings, REX programs should be portable between TRIPOS systems, with no modification necessary.

The REX language system itself achieves portability by being written entirely in BCPL, and by avoiding the use of any of the BCPL language extensions. This means that it is portable not just to other TRIPOS installations, but to almost anywhere which runs a BCPL system. The REX syntax analyser and interpreter make no assumptions about their BCPL run time system, in that they take great care over allocation and deallocation of resources, explicitly freeing anything which has been obtained. Also, since all the free store management is implemented as part of the REX system, only the most primitive of BCPL environments is required for the system to run.

Even though REX and the TRIPOS Shell were designed to run together, no assumptions are made about the operating system interface. The Shell and REX modules both communicate with each other and with the operating system by using a library of procedures in a third module, REXSHELL. By simple re-definition of the procedures in this module, it is possible to run either the Shell or the REX system in isolation. Similarly, it is possible to use the REXSHELL module to provide the REX language system with an interface to any other BCPL implementation, and as an illustration of the portability of the REX system, it was moved successfully onto an IBM 3081/D under MVS, with the only modifications being to the procedures in REXSHELL.

5.10 Performance

The following measurements were taken to give an indication as to the performance of programs written in REX. The first is a set of measurements of different parts of the language in isolation, giving some indication of their relative execution times. The second is an implementation of Ackermann's function in REX, whose performance can be compared with similar programs written in other command languages. All measurements were taken under TRIPOS on a Motorola MC68000 processor running at 8MHz.

5.10.1 REX performance test

The following test statements were executed to give some idea of the efficiency of the different language constructs. The tests include simple assignment, arithmetic and string operations, subroutine and function calls with different numbers of parameters, array lookup and assignment, and conditional and looping statements. Each of the tests were executed 1,000 times, and the timings taken from the TRIPOS real time clock, which is accurate to 0·02 seconds. The timing for 1,000 executions of the null statement **skip** was then subtracted from each of the measurements, giving the timing for the operation itself. The result is the timing for 1,000 operations measured in seconds, in other words, the timing for a single operation measured in milliseconds.

Statement	Time (mS)
`x := 100`	0·5
`x := 100 + 200`	0·9
`x := "100"`	1·1
`x := "100" "200"`	2·2
`x := 100 + "200"`	1·6
`x := "100" 200`	3·7
`sub0()`	1·9
`sub1(100)`	2·4
`sub2(100,200)`	2·8
`sub3(100,200,300)`	3·3
`x := fun0()`	2·4
`x := fun1(100)`	2·8
`x := fun2(100,200)`	3·3
`x := fun3(100,200,300)`	3·7
`x := array1[100]`	1·7
`x := array2[100,200]`	2·1
`x := array3[100,200,300]`	2·4
`x := array4[100,200,300,400]`	2·8
`x := array5[100,200,300,400,500]`	3·1
`array1[100] := 1`	1·6
`array2[100,200] := 2`	1·9
`array3[100,200,300] := 3`	2·2
`array4[100,200,300,400] := 4`	2·6
`array5[100,200,300,400,500] := 5`	2·9
`if true then skip fi`	1·2
`if false then skip fi`	0·2
`if true then skip else skip fi`	1·2
`if false then skip else skip fi`	1·2
`to 10 do skip od`	9·5
`for i to 10 do skip od`	11·9

5.10.2 Ackermann's function

As an indication of the performance of REX as opposed to other command languages, measurements were taken for different arguments of Ackermann's function, and then compared to similar implementations on other systems. A. W. Colijn [Colijn76, Colijn81] has taken timings for Ackermann's function written in the KRONOS and MULTICS command languages, and uses the results to discuss the relative power of the languages, and the limitations of the different systems. The KRONOS language is not recursive and has only three distinct variables, so it is impossible to implement anything more ambitious than "`ackermann(2,2)`". The MULTICS command language is more general, but runs into performance difficulties above "`ackermann(2,3)`".

```
$  Ackermann's function programmed in REX for comparison
$  with KRONOS and MULTICS command languages.

proc ackermann( m, n ):
    if  m = 0
        then  result  n + 1
    elif  n = 0
        then  result  ackermann( m-1, 1 )
        else  result  ackermann( m-1, ackermann( m, n-1 ) )
    fi
corp

x  :=  ackermann( 2, 4 )
```

The KRONOS measurements were taken on a CDC 6400, the MULTICS measurements on a Honeywell level 68 DPS. The REX measurements were again taken under TRIPOS on a Motorola MC68000 running at 8MHz. The timings are all in seconds.

Function	KRONOS	MULTICS	REX
2,0	0·318	0·861	0·0332
2,1	1·364	2·362	0·0880
2,2	3·849	4·501	0·1676
2,3	—	7·454	0·2734
2,4	—	—	0·4048

5.11 Summary

The REX system was designed as the result of an investigation into two different types of language which, although separate on traditional systems, have very much the same sorts of requirements. The first is the simple interactive programming language, which enables small throw-away programs to be constructed quickly and easily. The second is a well structured, easy to read language for the writing of command sequences.

The REX language itself is simple, with very few constructs which must be learned by the programmer. It is strongly typed, but all type checking is dynamic and performed at run time, with as much implicit type coercion being performed by the interpreter as possible. Arrays are handled in a rather novel way in REX since they resemble multi-dimensional records much more than conventional matrices. Each element of a REX array is entirely independent, and has its own value and data type. Since arrays can contain references to other arrays, complex data structures can be built and manipulated. Not only is the data type checking performed dynamically at run time, but so is the scope checking of variables. The effect is much more like that experienced in LISP by means of the association list, than it is in more conventional block structured languages.

The REX system has been implemented under TRIPOS, and runs alongside the Shell. One of the most important aspects of the language is the nature of its interface to the Shell and other parts of the underlying operating system. This is achieved through the use of a module of procedure libraries, REXSHELL, which acts as a link between the two halves of the system and an interface to those parts of the REX system which are TRIPOS specific. Through the use of this interface module, it is possible to run either the Shell or the REX system in isolation. Similarly, by re-defining the procedures which are operating system specific, it is possible to achieve a high degree of portability.

5.12 Evaluation of REX

Although it is possible to measure the performance of REX by timing the execution of programs, the real value of the work is very difficult to assess. It is not easy to say how much programmer productivity improves with the introduction of a new facility, and the only possible measurement is the amount the facility is used, and the type of feedback from its users. Of the 60 or so registered TRIPOS users at Cambridge, almost all have written at least one REX program, and over half are regular REX programmers. The language is often used

for text processing and file management, but the most common use is for simple command sequences.

When writing a short command sequence, there is little to choose between different systems, and simple macro processing such as that used in the TRIPOS c command or the MXEC system are entirely adequate. Since the command sequence is so small, neither efficiency nor readability are important issues, and the vast majority of command languages are capable of supporting this kind of application. For this type of sequence, a command programming language can actually be a hindrance, since it is likely that unnecessary keywords, string concatenation operators and so on will have to be used, when all that is required is simple text substitution.

For more complex command sequences involving conditional execution of commands, looping and so on, a more powerful type of command language is required. Variables are needed as well as simple parameter substitution, and there must be a convenient way to interrogate the success or failure of command execution through return codes. Macro processing is less suited for such an application, since a better control structure is required. For this sort of command sequence, a command programming language is ideal, since it has the sort of control structure and operating system interface needed. However, it is also possible to use a command language interpreter which has had programming language features added to it. IBM's EXEC languages, the UNIX Shell and C Shell, and the command languages of KRONOS and MULTICS are examples of systems based on this principle, and all are powerful enough for this type of application.

When it comes to writing throw-away programs, more facilities are required, and it is here that the command programming language comes into its own. Even though conditional statements, looping constructs and simple I/O are provided in the powerful command language interpreters mentioned above, there are often many restrictions. For example, it is possible that looping and conditional statements cannot be nested, or that the language only provides string variables with no means for performing arithmetic evaluations, or that it is impossible to perform I/O on disc files.

Because a command programming language is designed with structure, simplicity and flexibility in mind, it provides the necessary facilities to enable either simple or complex programs to be written quickly and accurately. At the same time, since its design is based on "real" programming languages (as opposed to being an augmented command language interpreter), it has a clean and simple syntax, and provides all the expected I/O, arithmetic and string manipulation

facilities. The benefit of such a system can be seen both from the point of view of the user and the system designer. The user benefits, since he must learn only one language rather than two, and the language he has to learn is designed specifically to be simple to use and easy to get right. The system designer also benefits, since it is possible to combine two tasks into one, and this reduces the amount of implementation and maintenance effort which he must expend.

6. Additions to the User Environment

6.1 Introduction

In the previous two chapters, the emphasis has been on enhancing the user environment by providing a powerful command language interpreter and command programming language. The user tends to view these facilities as part of the operating system, since they appear to him to be built-in and unchangeable. This chapter investigates how the user environment can be improved, not by enhancing the resident operating system, but by adding user-level services which either improve the overall utilisation of the hardware or provide powerful facilities which enable complex operations to be performed quickly and easily.

The factor which makes this kind of work desirable and necessary is the increase in the amount of memory which is available on the new generation of personal workstations. Although there are some programming languages and general applications which are capable of making use of large amounts of memory, the majority cannot, and as a result, techniques must be developed to take account of this. As stated in chapter 4, one of the ways of utilising large amounts of memory is to have many resident tasks which provide useful services. Memory can also be used to cut down on the amount of disc traffic, and since modern cheap peripherals are usually fairly slow, doing so is advantageous.

The facilities described in this chapter fall into two categories, both of which help to enhance the user environment. Those in the first category improve the utilisation of the whole system, either by performing certain operations faster than was possible before, or by making more efficient use of the available hardware facilities, particularly memory. The facilities in the second category do not actually cause any improvement in performance or hardware utilisation themselves, but make the system much more convenient to use, and so have the effect of altering the way in which certain types of computation are performed.

6.2 Motivation

The motivation for the first part of this work was the wish to improve the performance of TRIPOS running on the processor bank 68000 computers. The conditions experienced on these machines were typical of those found on other systems, in that memory was in plentiful supply and the speed of most computations was limited by the speed of access to the filing system. In the

specific case of the Cambridge environment, the reason for the slow filing system access was that the discs were managed by a remote fileserver which was running into performance difficulties because of the growing number of clients it had to support. The problem, though, is a more general one, and is experienced by any system whose general operation is I/O bound rather than CPU bound. The second part of this work was prompted by the desire to improve the facilities available to those people using TRIPOS for program development to support their research. The aim was to investigate those areas where services were either not provided or provided only in a very inconvenient form, and then to see how they could be added in such a way as to improve the user working environment.

As with the REX system described in chapter 5, TRIPOS proved an ideal context for this work, since it supported users who were prepared to try out new ideas and give useful feedback. Under TRIPOS, it is also very easy to add dynamic services in the form of *pseudo devices.* The mechanism is identical to that used by peripheral handlers, for example the console and disc drivers, but as the name implies, no real device is actually handled. Using pseudo devices, it is possible to add new features to the user environment without altering the operating system in any way, and because of the generality of the TRIPOS stream mechanism, they appear to the user simply as an extension of the normal filing system.

6.3 Winchester discs

Because access to the disc filing system was rather slow, there have been many ideas as to how the situation could be improved. One suggestion [Wilson82b] was to supply each 68000 workstation with its own 10 megabyte Winchester disc drive. Unfortunately, due to a design error on the part of Motorola and the lack of a convenient Memory Management Unit for the 68000 processor, it was not feasible to use such a disc as a paging device for virtual memory. It would have been possible, however, to use the disc as a local cache, which could either be built up implicitly, with the filing system keeping recently accessed items on the local disc rather than the remote fileserver, or explicitly, with the user copying his working environment onto the Winchester disc at the beginning of a session, and copying it back at the end.

The Winchester disc idea was never implemented, for several reasons. Firstly, Winchester discs, disc controllers, 68000 disc interfaces and mounting racks were all rather expensive—it was far more cost effective to give each 68000 machine another half megabyte of memory than to supply it with discs.

Secondly, because of the Cambridge "processor bank" philosophy of machine allocation (rather than the Xerox "one per person" strategy), when a machine crashed it would have been impossible to guarantee that the same machine would be available afterwards in order to restore the session. Thirdly, machines in the processor bank are available to any authorised user, and there is no restriction as to the range of software which can be run on the machines. Because of this generality, each operating system is entitled to make full use of the connected hardware, and hence no assumptions can be made about the state of that hardware when a new operating system starts up. This meant that it would not have been practical to keep a permanent cache on the Winchester disc, since its integrity could not be guaranteed between sessions. Finally, since TRIPOS itself is a totally unprotected operating system, it is very likely, particularly during program development, that the machine would crash before such time as any modified files could be copied back to the fileserver. Combined with the fact that it would not even have been possible to guarantee the re-allocation of the same machine in order to pick up the pieces, the result is that a Winchester disc was not a reasonable proposition.

6.4 The CORE: device

Given that memory was cheaper to purchase than disc, this is what was actually done, and all but four of the processor bank 68000 machines were upgraded to at least one megabyte each. TRIPOS, being originally designed for small, 16 bit mini-computers, had never run in so much memory before, and was unable to use it sensibly. There existed, therefore, the ridiculous situation of TRIPOS only using about one quarter of the memory available, and at the same time, often being idle due to the slow speed of access to the filing system.

The first attempt to improve this situation came in the form of the `CORE:` device, implemented by M. F. Richardson. The aim of `CORE:` was to build a virtual disc drive in memory, with disc blocks being replaced by memory pages. The advantage of this approach was that there already existed a TRIPOS file handler "FH3" [Richardson80] for physically connected discs, and it was simply a matter of altering the disc device driver code to access memory pages rather than physical disc blocks. As expected, `CORE:` was quick to implement, and since all data structures were held in memory, access to files held in `CORE:` was fast. There was also the added advantage that, because `CORE:` acted exactly like a real disc, nothing needed modifying in order to make use of it.

The CORE: device had several uses, two of which were more important than the others. Firstly, it was useful for keeping small, temporary files, for which there is no necessity to keep a record on disc. Falling into this category were simple, throw-away programs, listing output from compilers and assemblers, and intermediate object modules used for linking and loading. Secondly, it was useful for keeping the binaries of frequently used programs, so that loading these programs did not require access to the main disc filing system. The TRIPOS CLI could not make use of this facility, since it only looked for commands in the current directory and the system command directory. However, the TRIPOS Shell enabled CORE: to be used to its full advantage in that, through the assignments "COM1:" and "COM2:", it was possible to have user command libraries which were searched either before or after the system command directory. Simply by executing:

```
assign com1: core:
```

it was possible to copy frequently used commands from disc into CORE:, and from then on, to have the cached versions over-ride those on disc.

Using CORE: to keep cached commands had an enormous effect on programmer productivity and general morale, since it made it possible to enter an editor without the frustrating delay waiting for it to be loaded from disc. Additionally, other programs could be made to run faster. For example, the speed of the BCPL compiler could be increased by a factor of two in some circumstances, simply by keeping the intermediate OCODE file in CORE: rather than on disc.

There were, however, several disadvantages to the CORE: approach. Firstly, the handler itself was quite large (since in its "FH3" form, it was capable of handling an entire filing system), with only a small proportion of its facilities being used. Although the size of the handler in memory terms is not particularly important, the CORE: handler had to be loaded from disc at the beginning of each session, and hence the larger it was, the longer it took to load. Secondly, since CORE: was simply a disc filing system handler with its blocks in memory rather than on disc, there was the overhead of keeping directory blocks, file root blocks, checksums, block pointers and so on. Similarly, the unit of allocation was the "block", and so memory was wasted if the last block of a file was not full. Thirdly, CORE: was simply another filing system from which programs had to be loaded, and no account was made of the fact that programs in CORE: were in memory already. The effect of this is that there were always *two* copies of a program in memory—one in CORE: and one actually being executed. If the

program being loaded was large, say an editor, then double the amount of memory was required, and the amount of copying involved was enough to give a noticeable delay between typing the command, and the command starting to execute.

6.5 The PRELOAD system

From experience with `CORE:`, it was obvious that it was beneficial to utilise memory to keep pre-loaded programs. Unfortunately, `CORE:` was too general a solution to the problem, and due to its generality, could not make use of two simple facts specific to loaded programs.

The first is that loaded modules (in other words, files loaded using the BCPL procedure *loadseg*), have a well defined format in memory, which is different from that used in `CORE:` blocks. Any system which attempts to improve the handling of pre-loaded programs must use this format if it is to avoid large amounts of copying and duplication.

The second is that, under TRIPOS, nearly all the programs to be loaded are BCPL modules, and that the vast majority of these contain code which is pure. In other words, most programs which would be pre-loaded do not modify their own code, and hence are capable of being shared. On the whole, those which are not pure, are in fact serially re-usable, in other words, their code cannot be shared, but once finished with, can be used again. This simple piece of information about TRIPOS programs has a very important implication—in most cases, it is not actually necessary to give each task its own copy of code to be executed, and when execution has completed, since the code will not have been corrupted by its execution, there is no need to discard it.

Given the facts outlined above, it became clear that it was possible to design a system for keeping pre-loaded programs, where the code itself was kept in a form ready for use, with no copying necessary. If the loaded code were pure, then it was also possible to allow multiple tasks to use the code simultaneously, with no need to load a copy for each new incarnation.

In order to implement a scheme which worked in this manner, it was necessary to determine *where* in the loading sequence the pre-loaded code should be used in preference to the disc version. With `CORE:`, it was necessary to employ the `COM1:` facility of the TRIPOS Shell in order to give the desired effect, since the pre-loaded programs were simply held as files in a memory filing system. For a more specialised implementation, where the *only* thing capable of being loaded into memory is a program, it is clear that the correct place to intercept

the loading of programs is in the BCPL procedure *loadseg*, through which all loading requests must pass.

6.5.1 The SEGLIB library

Over-riding system procedures under TRIPOS is a straightforward job, and requires use of the `library` command [Evans81]. The function of the `library` command is to load a segment into memory, and then to append this to the segment list referenced in the task control block. The effect is that whenever a new command is executed or a subtask created, this segment list is scanned in order to initialise the globally defined procedures held within the segments. The segment list is scanned in a specified order, and hence globals defined in later segments will over-ride globals defined in earlier segments.

In order to implement the pre-loading system, a library segment `SEGLIB` was produced, in which re-definitions of the system procedures *loadseg* and *unloadseg* were held. These new procedures would, before going to disc in order to load a segment, investigate whether the `PRELOAD:` device was resident, and if so, pass over the responsibility of loading the program to it.

The reason for using a separate task, rather than a global data structure shared between tasks, is primarily one of synchronisation. It would obviously have been possible to employ some global data structure, linked into the TRIPOS rootnode in a similar way to the assignments list, in which all the information about pre-loaded programs was stored. This solution has one major drawback, and that is the problem of updating the data structure in an atomic manner. It would have been necessary for the updating task to have a critical code section which would run at maximum possible priority (effectively uninterruptable) while it was manipulating the pre-load data structure. Given that there is no shortage of memory, and that inter-process communication is cheap under TRIPOS, it is much better to handle important data structures within a task, so that the synchronisation problems are handled by the kernel's message passing system. It is also better from a programming point of view, since the pre-load data structure becomes an abstraction, hidden behind a well defined message interface, thus allowing details of the implementation to be transparent to the user of the service.

6.5.2 The PRELOAD: device

The pre-load handler task is implemented as a pseudo device, which is mounted using the `mount` command. The reason for using a device is that TRIPOS does not associate a name with a task, simply a number—the *task id*. There is no way normally of finding out the identity of a task simply by knowing

its function, since the allocation of identifiers is dynamic, and may change from occasion to occasion. The four main system tasks—the primary Shell, Debug, the Console Handler and the File Handler—all have fixed identifiers, which are defined to be the same on all systems. This is satisfactory for a small number of tasks, since the identifiers can be fixed at system generation time, but for tasks which are created dynamically, it would mean reserving a free task slot for every possible loadable handler.

The "mounted device" scheme avoids this problem by associating with each handler a *name*, which is kept in the *assignments list*. Along with the name is kept the *task id* of the handler, and a secondary value which is device dependent. The BCPL procedure *devicetask* is provided for interrogating the assignments list, and effectively provides a mapping between the task name, and its identifier and secondary value. Rather than binding the pre-load handler to a specific task id, it is much more convenient to bind it to a name—PRELOAD:—which can then be looked up in the assignments list whenever required. It is through this mechanism that the SEGLIB library determines whether the pre-load handler is resident.

Within the PRELOAD: device itself, the loading and unloading of programs is handled, along with the maintaining of the main pre-load data structure. Calls come into the device from all tasks which have the SEGLIB library loaded, and each request is handled in turn. Because several disc accesses can be performed in parallel, it is advantageous to have a degree of parallelism within the PRELOAD: device itself, so that loading a large program from disc does not have the effect of blocking other requests, possibly for other programs which are already pre-loaded.

To this end, the PRELOAD: device is organised as a master scheduler and a set of slave coroutines, each of which is capable of handling any of the pre-load functions. When a new request arrives, unless the operation requested is something which can be done quickly and synchronously, the table of coroutine pointers is scanned for one which is currently unoccupied, and it is then activated to handle the request. If all the coroutines prove active, then the request is added onto an internal queue, and processed when the next coroutine becomes free.

6.5.3 Requests to PRELOAD:

The possible requests to the PRELOAD: device are:

- Load
- Unload
- Loadseg

- Unloadseg
- List
- Monitor

The *load* request causes a file to be loaded from disc, and added to the list of pre-loaded programs. This operation returns to the caller immediately, and is then handled asynchronously by the device. The reason for this is to enable a series of pre-load operations to be executed quickly in a user's initialisation sequence, without the necessity of using background tasks. The effect to the user is the same, but the result is to remove the problem of memory fragmentation which occurred due to the alternation of permanent pre-load code areas and transient Shell stacks and global vectors.

The *unload* request causes a file which is in the list of pre-loaded programs to be removed from the list, and unloaded from memory. If the program is already in use, then it is not unloaded immediately, but only when its use-count next drops to zero.

The *loadseg* request is a call from the *loadseg* procedure in the SEGLIB library. The list of pre-loaded programs is scanned to see if the file being loaded corresponds to a pre-loaded program. If found, then the use-count is incremented, and a pointer to the pre-loaded segment is returned. If the program is not found, then it is loaded from disc in the normal manner.

The *unloadseg* request is a call from the *unloadseg* procedure in the SEGLIB library. The list of pre-loaded programs is scanned to see if the segment being unloaded corresponds to a pre-loaded program. If found, then the use-count is decremented, and if required, the entry removed and the program unloaded. If the program is not found, then it is unloaded from memory in the normal manner.

The *list* request causes a copy to be made of the pre-load list, which is then passed back to the caller, so that it can be printed out. The copy is made synchronously, so as to avoid strange timing effects with the updating of the main list.

The *monitor* request enables or disables the PRELOAD: handler monitoring function. When enabled, each *loadseg* attempt is logged to the console, along with the name of the program being loaded, and information as to whether the program was found in the pre-load list, or had to be loaded from disc. Using the *monitor* facility, it is possible to detect program naming problems, and to investigate whether optimisations are possible by pre-loading something which is used frequently, for example, the handler for the "NIL:" device which provides

access to dummy streams.

6.5.4 The pre-load list

The pre-load list is the main data structure employed by the PRELOAD: device. As its name implies, it is a linked list of pre-load entries, each of which holds information about a single pre-loaded program. The fields in the pre-load list entry are:

- *segment*—Pointer to the loaded program
- *segname*—Name of the loaded program
- *usecount*—Use-count for re-entrant (pure) programs
- *unloading*—Flag saying whether the program should be unloaded
- *reentrant*—Flag saying whether the program is re-entrant
- *checksum*—Binary checksum of the program

6.5.5 Naming of pre-loaded programs

The naming of pre-loaded programs presents a problem which is worth discussing. Ideally, a pre-loaded program should be recognised as such, no matter which path name is used to access it. For example, the files:

```
status
:c.status
sys:c.status
```

are all the same file in the filing system, except that they have different names, and *loadseg* called with any of the three arguments should always load the same program.

Under TRIPOS, it is possible to identify a file uniquely, by using its *key*. On a system with a local disc, this is the *block id* of the file header block; on a fileserver based system, this is the *puid* (the fileserver's unique identifier) of the file. Both these numbers are guaranteed to be unique, and hence can be used to identify the file unambiguously. It would be possible, therefore, to associate with each loaded module its key, so that the naming ambiguities could be resolved. This, however, defeats the object of having a pre-loaded program, since to obtain the key of a file, it is necessary to consult the filing system, which immediately re-introduces the delay due to disc access which is supposed to be avoided.

As a compromise, rather than keeping the file key, the file *name* is kept in the form in which it is expected to be used. The command **preload**, which acts as an interface between the user and the PRELOAD: device, allows an optional alternative name to be given to the program, which is the name it is to be known

by when it is pre-loaded. For example, the **status** command could be pre-loaded by using the following command:

```
preload sys:c.status as status
```

In this case, "`sys:c.status`" is the full path name of the program, and is used to identify the file unambiguously for loading purposes. The name the program is known by when loaded is "**status**", since this is what would be typed by the user as the command name. It is also possible for this mechanism to provide alternative names for programs, should a user prefer this.

There are, unfortunately, times when the name presented to the PRELOAD: device by *loadseg* does not match the name by which the program is known in the pre-load list, and hence unnecessary disc accesses are performed. This is, however, harmless, and does not happen very often. When it does happen, the effect is fairly obvious and can be investigated by using the PRELOAD: *monitor* option.

When a program is pre-loaded, one of the pieces of information passed by the user to the PRELOAD: device is a flag saying whether the program is re-entrant or not. The value of this flag affects the action of PRELOAD: whenever that program is involved in a *loadseg* or *unloadseg* call.

For re-entrant programs, when a *loadseg* call is made, the program's use-count field is incremented, and a pointer to the loaded code is returned. When an *unloadseg* call is made, the program's use count field is decremented, and the binary checksum of the program is re-calculated and compared with its original value, in order to detect whether the program has corrupted itself. If it has, then a warning is printed out, and the program is unloaded at the first possible opportunity.

For non re-entrant programs, when a *loadseg* call is made, the pre-loaded version is *only* used if the program's use-count field is zero. If not, implying that the code is already in use, then a new copy is loaded from disc, so as to prevent an impure program from being used by two tasks simultaneously. When an *unloadseg* call is made, no attempt is made to detect program corruption by re-calculating the binary checksum—the onus is on the author of the non re-entrant programs to make them serially re-usable, if advantage is to the taken of the pre-load system.

Whenever an *unload* request is made, or an unexpected program corruption is detected, the program's *unloading* flag is set in the pre-load list entry, and whenever the use-count reaches zero, the program is unloaded, and the entry removed from the pre-load list.

6.5.6 Problems with PRELOAD:

The `PRELOAD:` system as described works well in practice, and is simple to use, even for relative newcomers to the system. It does, however, have drawbacks, and is certainly not the only way of achieving the desired effect. There are two main problems associated with the `PRELOAD:` system as it stands.

Firstly, as mentioned before, in order to cut down on unnecessary disc accesses, the name of the pre-loaded program must match exactly the name used in a *loadseg* request, otherwise the effect is lost. An example of this is the "`dat-to-strings`" overlay, which is loaded by many programs, to provide mappings between TRIPOS internal date format, and printable strings. Some programs reference the overlay as "`:l.dat-to-strings`", and others as "`sys:l.dat-to-strings`". Inconsistencies such as these must be detected by the user, and corrected manually by altering and re-compiling the offending programs.

The second problem, and one which is much more difficult to solve, is the case when the system procedures *loadseg* and *unloadseg* have already been re-defined by something other than `SEGLIB`. This is the case, for instance, whenever a language other than BCPL is run under TRIPOS—ALGOL68C and MODULA-2 each have their own linking loaders, which are implemented as libraries, in a similar way to `SEGLIB`. Unfortunately, in order to avoid infinite recursion, both `SEGLIB` and `PRELOAD:` must each have their own definitions of *loadseg* and *unloadseg*, and those definitions must be fixed at the compile time of the two programs. This means that anyone wishing to use a version of *loadseg* which is different to the default, must have their own, modified versions of `PRELOAD:` and `SEGLIB`.

This is highly unsatisfactory, since it makes the installing of updates to the pre-load system a slow and tedious process, and there is always the possibility of missing one of the private versions. Thus, a mechanism is required which saves the value of a global which is about to be over-ridden, and then makes it available to the program which is doing the over-riding. This is impractical, and at the time of writing, no simple, clean solution to the problem has been found.

6.5.7 Alternative methods

The mechanism described above is only one way of solving the problem of pre-loaded programs. With `PRELOAD:`, programs are pre-loaded at the behest of the user, and fully under his control. Only those programs which are explicitly pre-loaded will ever be kept in memory, and then only until the user decides explicitly to unload them.

It is possible to imagine a system where the only thing under the control of the user is the amount of memory available for pre-loaded programs, and that some, more automated way could be found for deciding when to load and unload pre-loaded programs. One could no doubt define some suitable algorithm which employed a "least recently used" technique for deciding what to unload when storage in the pre-load area was scarce. For the system to work in a reasonable way, given the normal "edit, compile, debug" cycle, the algorithm would also have to take into account frequency of use, size of program and length of time taken to re-load it, should it have to be brought in again from disc.

The problem with such a scheme, as with any paging algorithm, is that there are always situations when the system will work pessimally, and very few when it will work exactly as the designer had intended. In effect, such a scheme would require "hints" from the user—such as what to load when the system started up initially—in order that more sensible decisions about loading and unloading could be taken. When taken to its logical extreme, "hints" become direct commands to the pre-loading system, and the manipulation of pre-loaded programs reverts entirely to total user control.

Experience has shown that the user is almost always the best judge of what is best to be kept pre-loaded, and that even though it would, theoretically, be possible to automate the process to some extent, there is little incentive for doing so.

6.5.8 Performance

As with the REX system described in chapter 5, it is difficult to measure the improvement in programmer productivity due to the inclusion of a new facility. Obviously, it is possible to measure the amount of time taken to load a program from disc, and given that the equivalent operation for pre-loaded program is effectively instantaneous, thus calculate the amount of time saved. The absolute figures are small: 10 seconds on average for entering a screen editor and 15 seconds for performing a BCPL compilation are typical figures. This may seem insignificant compared to the amount of time actually spent in an editor or performing a compilation, but the advantage of the extra speed is of psychological importance to the user.

If entering an editor entails a delay during which the programmer is forced to be idle, then frustration occurs, and after a while there is a growing reluctance to leave the editor, given the delay involved in re-entering it. Similarly, if compiling a program means that the compiler and its overlays must be loaded from disc, then there is a similar reluctance to use it because of the time taken. On the

other hand, if entering the editor and compiling a program are both fast operations, then it is possible to adopt the "fix one problem at a time" approach to debugging, rather than spending a large amount of time staring at a terminal or a program listing.

The benefit of a pre-load system is, therefore, not in the small amount of absolute time saved in loading the programs from disc, but in the change of attitude taken by the programmers, and the methods they use for program development. Although it is difficult to quantify this benefit, many people (including the author) believe it to be high.

6.6 Other TRIPOS devices

Keeping programs cached in store is only one way of using the large amount of memory available on the new 68000 TRIPOS machines. Another method, which has just as big an effect on programmer productivity, is to increase the number and quality of the services available to him. The biggest impact in this area was no doubt the TRIPOS Shell and REX system, but there are other smaller programs which, each in their own way, have managed to improve the working environment significantly.

As mentioned before, the most convenient way of encapsulating a new service under TRIPOS is to use a task. Synchronisation problems are then handled by the operating system kernel, and the details of the implementation can be hidden from the user behind a well-defined, message level interface. `PRELOAD:` is an example of a device, which is loaded into memory by means of the `mount` command—originally intended for loading disc handlers and so on. Using `mount` is only one of three ways of taking advantage of pseudo devices though, and the examples which follow give an indication as to how the other methods work.

6.7 The WINDOW: device

The "`WINDOW:`" device falls, along with `PRELOAD:`, into the first category of pseudo devices—those which are mounted. The TRIPOS `mount` command takes a description of the device from a file, including the filename of the handler program, task priority, stack size and so on, and constructs the device in memory, adding the device's name to the assignments list.

The default TRIPOS console handler communicates with the Terminal Concentrator [Ody84] on Ring based systems, and directly with the physical console device on stand-alone systems. If a command line interpreter or Shell is required which uses a console different to the default, then a new handler must

be provided. The `WINDOW:` device provides a console handler which has the required interface to enable a full screen editor to implement *process buffers.*

6.7.1 Process buffers

The first editor to use the concept of process buffers was EMACS [Gosling82] which runs under UNIX. Process buffers are full screen editor windows which, instead of representing a file being edited, represent a console session. Everything which would be written to the console, instead gets written into the editor window, and anything typed into the window is handled as though it had been typed at the console.

The benefits of such a mechanism are easy to see. No longer are artificial command histories and command line editing required, since a transcript of the entire console session is available in the editor window, and commands, data and program output can all be edited, copied and moved as though they were simple text, with all the power of the full screen editor at the disposal of the user. If process buffers are so much more powerful than simple console sessions and command histories, then why are they not always used in preference? There are two main reasons for this.

Firstly, and most importantly, full screen editors tend to be slow at re-drawing areas of the screen which have changed. In order to work properly, they must keep an internal map of what the screen should look like and what it actually does look like, so that the necessary updates can be made. What a full screen editor is unable to do properly is to make use of the hardware assisted "scroll" facility of the terminal itself. This does not matter whilst editing a file, since a user will tend to step through the file one page at a time, in which case the whole screen must be re-drawn anyway, or simply type characters, in which case the speed of update is not critical. When "editing" a console session though, since output from commands cause the window contents to change suddenly and without warning, the screen must be re-drawn fairly often, which can be a slow process. There is also a problem with programs which use control characters or escape sequences to access facilities provided by the terminal (such as highlighting, cursor positioning and so on), since these are handled by the editor and are not passed through to the terminal.

Secondly, full screen editors tend to use non-printing control characters as editor commands. This means that these characters cannot be typed directly into a process buffer, since they would be interpreted by the editor, and never be received by the underlying program. These characters must therefore be "escaped" by using some sort of special character sequence.

Despite their problems, process buffers are extremely useful objects in practice, since there are many times when it is convenient to be able to edit a file and run a console session simultaneously. Under TRIPOS, there was only one editor which was capable of being able to perform this task, and so the WINDOW: device was implemented specifically with this editor in mind. The interface to WINDOW: is, however, extremely general, and could be used by any other TRIPOS editor to perform the same function, with little or no modification.

6.7.2 The implementation of WINDOW:

The editor in question is WORCESTAR [Stoye83] which was implemented by W. R. Stoye, and the original encouragement to provide a process buffer handler came from him and other WORCESTAR users. One of the major differences between other TRIPOS editors and WORCESTAR is the ability to split the screen between two different files. The mechanism is by no means as general as the EMACS loose binding between buffers and windows, but is enough for most applications, and enables the introduction of process buffers.

In order to implement one process buffer, four separate tasks are required. The editor task handles the real console I/O, and deals with screen layout and general screen management. The WINDOW: task handles requests from the editor task, and in turn, communicates with the COHAND (console handler) tasks, of which there is one per window. Each COHAND task has at least one Shell task, which actually executes the commands typed by the user.

Because WORCESTAR is written in a single threaded manner, its interface to the WINDOW: device is polled. Each window has its own *twid*—a task window identifier—and every transaction associated with a window is flagged with its *twid*. The editor calls WINDOW: with one of the following requests:

- Create Window
- Delete Window
- Read Buffer
- Write Buffer
- Write Escape

The *create window* request causes WINDOW: to allocate a new window slot, and if necessary, to create a new COHAND and Shell task to go with it. The result of the request is the *twid* of the new window. The *delete window* request causes WINDOW: to cancel the relevant window slot, and to invalidate the *twid* given, so that it cannot be used again. The *read buffer* request causes WINDOW: to call the relevant COHAND task for this window, in order to pick up any buffered output

which should be written to the window. The ***write buffer*** request causes WINDOW: to call the relevant COHAND task for this window, in order to write a line of input typed into the window to the main input task for that window. The ***write escape*** request causes WINDOW: to interpret a line of input typed into a window in a special way, passing control information to the relevant COHAND task.

The WINDOW: device has the effect of acting as a "go between", providing a clean interface between the editor task, and the tasks using the process buffer. The *create window* and *delete window* primitives are relatively self explanatory, in that they deal with the creation and deletion of windows, along with the housekeeping entailed in such actions. The *read buffer* and *write buffer* primitives are somewhat confusing, in that "read" and "write" are taken from the point of view of the editor task, not from the point of view of the program in the process buffer. Essentially, all that the WINDOW: device does is to multiplex output coming from the process buffer tasks, giving it to the editor when required, and to demultiplex output coming from the editor, sending it to the process buffer tasks. The *write escape* primitive is necessary, since WORCESTAR uses many of the control characters which TRIPOS requires in order to give commands to the console handler. Examples of this are CTRL-B, which is used to communicate a "break" condition to an executing program, and CTRL-S which is used to select the current input task. Both these control characters have different meanings to the editor. The solution to the problem is to enable the user to type a line into a process window, flagged in such a way that it is interpreted by the console handler, rather than being transmitted to the program running in the window.

6.7.3 The COHAND console handler

The WINDOW: console handler, COHAND, must be an exact replacement for the standard "*" device. Unfortunately, the RMVTHand implementation of this device [Knight82] has grown so much that many of the functions it provides are either inappropriate to process buffers, or impossible to implement. Those which are relevant have been implemented, with the others being ignored.

By far the most common usage of the "*" device is simple, line-by-line input and output, using standard stream control blocks created by *findinput* or *findoutput*, and with input/output being performed using *rdch* and *wrch*. This is easy to implement, since each line of input or output corresponds exactly to the "buffer" which it required by the editor and WINDOW: handler. Other console handler functions, for example "set current input task", are also easy to implement, and so have been included.

What is not easy to implement is the entire set of console handler functions which are involved with single character I/O. These functions were added because RMVTHand needed to know whether to drive the virtual terminal protocol stream in *line* mode or *character* mode [Ody84], and were necessary to support full screen editors. In order to deal with this problem sensibly, functions were added to enter and leave "single character mode", and perform "single character read" and "single character write" operations. On top of that, extra functions were included in an attempt to speed up the execution of those programs which used single character mode—mainly full screen editors. An example of this sort of function is the "how much input" request, which enables a program to find out how many characters have been buffered, waiting to be read.

Given that COHAND runs under the WORCESTAR editor, which itself runs in single character mode, it is inappropriate to implement another level of single character I/O. The reason for this is that programs which switch RMVTHand into single character mode, implicitly take over control of the entire console, and in particular, the layout of the screen. When running in a window, a program must only be allowed to take over its own window, and cannot be allowed to touch the rest of the screen. Unfortunately, RMVTHand does not provide primitives like "clear screen", "position cursor" or "set highlight mode", and so any program which handles the entire screen, does so by sending control characters or escape sequences directly to the terminal itself. Consequently, since the single character I/O could not be provided in a satisfactory manner, it was omitted.

6.7.4 Summary

The process buffer scheme described above works well, and has only one major disadvantage, and that comes from an inadequacy of the TRIPOS system itself. Once a task has been created, it cannot be deleted explicitly until such time as it runs to completion. This, in the case of a task running a Shell as part of a process buffer, is never. The effect is that, whenever a window is deleted, it is impossible to clean up the tasks which were created for that window.

In practice, this does not actually present a problem, since deleting a window simply suspends the polling of its COHAND task. As a result of this, any program using the console handler which requires some form of console I/O, will suspend waiting for it. Everything then remains in a suspended state until a new window is created, at which point, rather than create a new COHAND for the window, the old one is used in preference. This has the effect of waking up the program waiting for I/O, and everything then starts up again. The create and delete

functions would more accurately be described as *connect* and *disconnect.*

6.8 The EX: and EXALL: devices

The second category of pseudo devices are those which are loaded on demand. Like mounted devices, they are represented as names followed by a ":" character, but unlike mounted devices, the names are never included in the assignments list. Devices loaded in this manner tend to be single threaded in structure, and hence are only capable of handling one client at once. The handlers of such devices are loaded in response to *findinput* or *findoutput* requests, where the name given as the argument is prefixed by the name of the handler. When presented with such a name, the assignments list is scanned for it, but if this operation fails, the system handler directory "sys:h" is scanned for a handler of the same name. If one is found, then it is loaded into memory, and everything then proceeds as if the device had been mounted.

An example of such a device is the Byte Stream Handler, "BSP:", written by B. J. Knight. Here, advantage is taken of the fact that running a Ring byte stream connection is essentially a two stage process. Firstly, the *open* block must be sent out, and its *openack* reply awaited. This part is essentially synchronous, and is accomplished as part of the task which is opening the byte stream connection. Secondly, once the connection is established, the byte stream is run as a finite state machine with an associated byte stream control block, the code for which is shared by all byte stream connections. This is implemented, either as a separate task in the same machine in the case of standard TRIPOS [Knight82], or as code in the Ring interface processor in the case of *supermace* [Garnett83]. The code to establish the connection is loaded on demand from the file "sys:h.bsp", and once established, the byte stream is then handled asynchronously, with the BSP: device code being discarded.

An alternative mechanism to the loaded device calling a task which already exists, is for the device to create a new task specially. It is this mechanism which the "EX:" and "EXALL:" devices adopt. The motivation for the work on these devices was that many programs require as their input a file containing a list of filenames on which the program should act. In order to obtain such a list of filenames, it was necessary either to type the list in by hand (which was slow, tedious and inaccurate), or to run the "ex" program, directing the output to a file, and then editing the file to remove the unwanted information.

There was obviously a need for a device which would, when given the name of a directory, return a file containing the names of the files within that directory.

There is no reason why the file produced needs to be stored in the filing system, because the TRIPOS stream mechanism is general enough for the file to be generated on the fly, as required by the calling program. The EX: and EXALL: devices perform exactly this function, with the only difference between them being that EXALL: searches for files in sub-directories, as well as in the named directory.

The EX: device is loaded by calling the BCPL procedure *findinput*, with argument:

```
ex:<directory name>
```

This loads the code of the handler (from the file "sys:h.ex") which obtains a lock on the directory, and sets up a stream control block for the newly created input stream. The effect of the operation is that, whenever the stream's *replenish* function is called, the directory is examined until the next file entry is found, at which point the name of the file is stored in the stream's buffer (with the *pos* and *end* fields being set up accordingly), and the calling program consequently reads this file name as data from the stream. When the stream's *close* function is called, the directory lock is freed, and the EX: handler deletes itself.

Being able to handle directories in this way is extremely powerful, and is made more so by an extra level of pattern matching which has been included. The name of the directory given as the argument to EX: or EXALL: may contain "wildcard" characters—"?" which matches any single character, and "*" which matches any number of characters. Thus:

```
ex:*-obj.???
```

would match all files with three character names, in directories whose names ended in "-obj". So, for example, the above pattern would match:

```
68000-obj.abc
lsi4-obj.def
pdp11-obj.ghi
```

and so on. The wildcard facility is implemented using the regular expression pattern matcher [Richards79b], with a suitable translation being performed between the "?" and "*" notation, and the more general and complex notation used by the pattern matching system.

The ability to have wild cards is particularly useful for those people working under TRIPOS who, either by choice or necessity, do not separate source and binary files into different sub-directories. An example of this is the MODULA-2

system which forces all files to be in the same directory, and using the UNIX philosophy, differentiates files by using suffices: "-mod" for MODULA-2 source, "-def" for MODULA-2 definitions, and so on.

6.9 The DIR: device

The third category of pseudo device occurs when a handler is accessed, not by its name in the assignments list, nor by implicit loading from the system handler directory, but via a filing system *lock*. A lock is represented as a small control block in memory, in which is stored all the information about the lock, such as its owning task, and whether the lock is exclusive or shared. This method is very similar to the assignments list system, except that the information about the handler is stored in the lock, rather than in the assignments list entry.

An example of a device which uses this mechanism is "DIR:", which enables multiple directories to be concatenated. A set of concatenated directories can be thought of as being a single, shared lock on a virtual directory, which is simply a table of other, shared locks on real directories. The property of a lock is that, whenever a filing system operation is attempted on an object whose owning directory is represented by a lock, a message is sent to the task which owns that lock, and it is up to that task to perform the operation. In this way, whenever access is attempted to any object in the set of concatenated directories, the DIR: device is called to handle it. Having taken control, DIR: can then attempt the operation on each of the stored directory locks in turn, until one which succeeds is found. The effect seen by the user is that the set of concatenated directories appear as one single directory, and hence a group of directories can be used anywhere that a single directory would be valid.

The actions described above are only appropriate for certain filing system functions. The operations *findinput* and *locateobj* are both valid, since the filing system is not updated, and even if the operation is ambiguous, it is reasonable for the first object which matches to be the one which is used. Operations such as *findoutput*, *deleteobj* and so on, are potentially destructive in their nature, and since any ambiguities cannot be resolved in a reasonable manner, these operations are faulted.

The main application for concatenated directories is where a "search path" is required, in other words, where objects which have the same purpose are actually held in different directories. The most striking example of where concatenated directories are of help is in the area of the Shell, and its handling of loaded commands and REX programs. The Shell provides the facility where the directory

assignments "COM1:" and "COM2:" are scanned for loadable commands, and the directory assignment "REX:" is scanned for REX programs. Using concatenated directories, it is possible to have a whole series of directories which are scanned in order to pick up the required program. This means that it is possible for groups of people, working together on the same project, to have shared and private command directories, in addition to any standard system directories, without ever having to specify explicit path names.

Since the DIR: device is only accessible while there is a copy of its lock in existence, the most convenient way to keep and access that lock is to give it a name, and store the name in the assignments list. The command responsible for manipulating directory assignments is **assign**, and with a slight change of syntax from the original version, it can easily handle concatenated directories. Normally, the command:

```
assign com1: 68000-obj
```

causes a lock to be obtained on the directory "68000-obj", and a pointer to this lock to be included in the assignments list under the name "com1:". For concatenated directories, the command:

```
assign com1: 68000-obj+:bjk.68000-obj+:njo.68000-obj
```

causes the DIR: device to be loaded and called. This in turn sets up a table of three entries, one for each directory in the list, and returns as its result a newly generated lock which refers to DIR: rather than the main file handler.

The DIR: task for a set of concatenated directories stays in existence until the number of locks which reference it drops to zero. At this point, it releases the locks it holds on the constituent directories, and then deletes itself.

6.10 Summary

The original implementation of TRIPOS, when moved onto 68000 machines with large amounts of memory, was incapable of making efficient use of the facilities provided by the hardware. Because of the speed of access to remote fileservers or slow local discs, much of the CPU power was lost waiting for disc transactions. This can be improved by using memory pages rather than disc blocks to provide a resident filing system. Alternatively, it is possible to pre-load frequently used programs so that the overheads of reading the programs from disc can be avoided, and the code can be shared between different tasks.

Another way of making use of the large amounts of memory available is to provide extra services which improve the user environment. Although not

strictly necessary, these facilities increase the convenience of the system to those who use it, and alter the philosophies adopted to solve everyday problems. Examples given above are a process buffer handler which makes it possible to conduct a console session from within a full screen editor, and two different facilities which ease the handling of filing system directories.

The additions suggested in this chapter are just some of those possible, and the examples given illustrate how the modifications may be made to an existing system. In many ways TRIPOS was ideal for this work since it was easy to add new facilities in the form of pseudo devices. It is the principles involved which are important though, and these are applicable to other systems which, like the original version of TRIPOS, are unable to harness the power of the machines on which they are run.

7. Kernel Issues

7.1 Introduction

The chapters up to now have tended to concentrate on how to adapt existing system software to take advantage of larger and more powerful personal computers. In contrast, this chapter concentrates on the issues involved in the design of operating system kernels, and aims to draw up a set of guidelines for anybody faced with the task of producing one for the new generation of machines. In order to reach the set of guidelines, many of the techniques adopted by previous designs are described, and then discussed with respect to different types of application. To give an indication of the success of a previous design, different aspects of TRIPOS are examined critically, and performance measurements are taken in order to suggest better techniques.

Various different areas of operating system design are explored in an attempt to reach conclusions. The whole area of language choice and utilisation is discussed, with reference to the operating system kernel itself and the programs which use it. The different uses of coroutines within an operating system are investigated, especially the choice which the implementor of such a system has to make—that between coroutines and processes. The topic of scheduling is examined, along with the relative merits of pre-emptive and time slicing techniques. The issue of storage allocation takes on new importance on large memory machines, and different methods are compared with respect to their relative convenience, simplicity and efficiency. Finally, more general issues such as structure, portability and the nature of the user interface are discussed.

As an illustration of some of the ideas presented here, an experimental light-weight multi-tasking kernel for network protocol software is introduced, and contrasted with its TRIPOS predecessor.

7.2 Language issues

There are two different levels at which programming language decisions must be taken when designing any new operating system. Firstly, there is the decision as to which language should for the operating system implementation—should it be a high level language for portability, or assembly code for efficiency? Secondly, what restrictions, if any, are to be placed on the user of the operating system, with respect to the languages he may use? These two issues are essentially independent, and so can be examined separately.

7.2.1 Operating system language

Before the introduction of the large personal workstation, there were essentially two sorts of computer for which operating systems were available. In simple terms, these could be classified as "large" and "small".

Large systems, on the whole, tended to be physically large, requiring a large amount of human effort to keep them running, and were financially expensive. Typically, the amount of memory available was measured in megabytes, with the rate at which instructions could be executed being greater—sometimes much greater—than 1 MIPS. Large systems, because of their capacity and expense, tended to support multiple users, either in the form of batch processing or time sharing. One thing which distinguished large systems from others was the complexity and speed of their peripherals. Elaborate channel and I/O processors took the function of handling peripheral devices away from the CPU, leaving it to perform users' calculations for the maximum amount of time.

Small systems, on the other hand, tended to be desk-top computers, with each user having his own personal machine. Their memory size was severely limited, being measured in tens of kilobytes rather than megabytes, and the processor speed was almost always around 1 MIPS, as this was the limit of the NMOS and CMOS technology out of which these systems were built. No elaborate I/O processors were available on these machines, and since peripherals were often polled rather than using DMA, much of the CPU's time was spent in handling devices rather than in performing computation.

Even though the types of system described above are so different, there was always the tendency to implement operating systems for both in assembly code. The reasons for this decision reflect the requirements of each. Multi-user operation combined with large numbers of processes mean that efficiency is of great importance to large systems, with the compactness of the resident software being the over-riding factor for small systems.

As the number of both types of system increased, so did the philosophy that portability was more important than efficiency, and as the small systems grew larger in capacity, so the need for compactness was reduced. In order to achieve this portability, it became increasingly common for operating systems to be written in a high level language. Examples of such systems are common: OS6 [Stoy72] was written in BCPL, SOLO [BrinchHansen76] in concurrent PASCAL, THOTH [Cheriton79] in EH (a derivative of BCPL), PILOT [Redell80] in MESA, MEDOS [Wirth81] in MODULA-2, and for larger machines, UNIX [Ritchie74] in C.

There are obvious advantages to writing operating system code in a high level language. The portability issue has already been raised, but this is only one reason for the choice of such a language. One of the main advantages is the ability to define algorithms and manipulate data in a structured manner, which has the effect of increasing software reliability and improving the system programmer's productivity. Using a high level language also has the effect of making program interfaces much more easy to define, and in certain circumstances, the language system can be used to check those interfaces for consistency. Add to that the fact the software written in a high level language is, on the whole, easier to read by someone other than the original implementor, and making modifications and fixing bugs is much easier than with assembly code.

Having listed the advantages of using a high level language for this type of software implementation, it is worth looking at some of the disadvantages. Firstly, no matter how good a compiler is, the output produced by it is never as fast nor compact as hand crafted assembly code. Secondly, high level languages, by their very nature, cushion the user from the harsh realities of the underlying hardware, and although this helps with portability from one machine to another, it does hinder the task of the system designer. This is because it is the job of the operating system to control and manipulate the hardware of the machine on which it is running, (accessing memory, peripherals and so on), and this job is made more difficult if the language makes it impossible to deal with the hardware directly.

The problems of compactness and efficiency are much less of a problem, for the simple reason that the distinction between large and small systems described earlier is becoming less and less. As micro-processors become more powerful and the cost of memory falls, it is now possible to have computers which fall into both categories. The systems are large, because of their power and memory size, but still small because of their physical size and because they operate in a single user manner. One factor which puts these new machines definitely into the "small systems" category, despite their capacity, is the fact that access to peripherals is still handled by the main processor, and since there is no point in attaching expensive, fast devices to cheap, single user computers, access to peripherals tends to be slow. Because of this, it is clear that the efficiency of the operating system kernel (in terms of CPU cycles) is not the major factor which determines the overall utilisation of the machine. There is little point in having a set of finely tuned systems procedures, which spend most of their time idle waiting for I/O to occur. Compactness is no longer an issue either, since the cost of memory is now a very small fraction of the price of the whole machine. Code which

appeared bulky in 64K bytes is of no importance if several megabytes are available.

With the arguments given above, there is no advantage at all in writing a new operating system in assembly code, and only one minor disadvantage to using a high level language—the problem of accessing the hardware directly. There are two ways of tackling this problem. One is to choose an implementation language which allows full and free access to the underlying hardware, with no data typing restrictions stopping this. OS6, TRIPOS, THOTH and UNIX were all designed in this way, with OS6 and TRIPOS being written in BCPL, and the others being written in EH and C respectively, which are both derivatives of BCPL. The other approach is to use a strongly typed language, such as PASCAL for SOLO, MESA for PILOT and MODULA-2 for MEDOS—all languages which hide the hardware behind a level of abstraction—and then use either micro-code or assembly code sections to provide loopholes in the typing mechanism to enable the hardware to be accessed.

Whatever the choice of implementation language, there is always a certain amount of any operating system which will always be machine dependent. Falling into this category is the code to handle bootstrapping and general initialisation, the section which handles the processor register dumping and loading while switching between processes, and the code which initiates, and then handles interrupts which come from device transactions. Often, the hardware can help in this area by, for example, providing a single instruction to perform a complete context switch.

The amount of software which must be written in assembly code depends very much on the nature of the operating system, and more particularly on the "ideal machine" it was designed for. Certain systems, notably PILOT and MEDOS, require very little, if any, since the operating systems run on hardware which has been tailored and micro-coded for their particular needs. Unless this is the case though, there will always be discrepancies between the ideal and actual machines, and when porting an operating system onto a new computer, there will always be some programming to be done at the assembly code level.

Given the arguments outlined above, there is no reasons why systems should not be designed so that the amount of programming at this level is kept to an absolute minimum. Since memory is plentiful, compactness is not an issue, and because of the relatively slow access to peripherals, efficiency is not really at stake. The operating system which follows this ideal to its logical conclusion is UNIX, with only the tiniest part of the operating system kernel and I/O device drivers being written in assembly code, with the rest of the kernel, device drivers

and other operating system functions being written entirely in C.

Even though much of TRIPOS is written in a high level language, the entire operating system kernel, a BCPL run time library and all the device drivers to be written in assembly code—typically 3,000 to 4,000 lines in all, or about three man-months' effort. The arguments used for this are necessity first, and efficiency second. Certainly, operations such as those mentioned before (initialisation, interrupt handling and so on) must be programmed in assembly code, and so these sections are inevitable. The efficiency argument is less easy to justify, unless the kernel is in use in a situation where it must have a fast real time response. For normal operating system running, where the client is a user, the efficiency issues are much less important, and a great deal of time and effort could be removed from the job of porting TRIPOS, simply by recoding most of the operating system kernel in BCPL.

Whatever the language chosen to implement the operating system, there are very strong reasons for writing all of it in that particular language. It would obviously be possible to implement different parts of the operating system in different languages, indeed this should not be prohibited by the designer, but there are many advantages to a single language approach.

Firstly, there is the issue of consistency between different operating system modules, and the definition of the interface between them. If two modules are written in different languages, then the task of defining and formalising the interface is made much more difficult. Automatic interface consistency checking (such as that provided by MESA or MODULA-2) can only be used conveniently if both sides of an interface are programmed in that language, otherwise the mechanism is useless.

Secondly, there is the issue of simplicity when it comes to accessing operating system data structures, such as process control blocks or the free store chain. If more than one language is used to access these data structures, then decisions must be taken in order to define the physical layout of these structures in memory, whether addresses should be byte or word values, and so on. Since no one representation is ideal for all languages, there is always a compromise involved when designing such systems, and so it is clear that, for at least one of the languages involved, access to these data structures would be in an unnatural manner.

Thirdly, there is the issue of portability, and the amount of effort required to move an operating system onto new machines. It is already the case that an assembler and a compiler must be written for the primary implementation languages. If, added to that, there is the requirement that other compilers must

be written for each of the other implementation languages, then the amount of work involved in these will increase the task of porting the operating system to such an extent that it becomes unfeasible.

For these three reasons, it is therefore advantageous to keep to a single language for the implementation of the entire operating system. In this way, interfaces between the modules can be checked by the language compiler (if this is possible), data structures can be handled in the most natural way for the language, and the task of porting the operating system onto other machines is reduced, since only one compiler is actually necessary. Also, if the amount of assembly code required by the operating system is small and its structure simple, then the assembler required need not be at all sophisticated, with the result that the language implementation is the only major obstacle to portability.

7.2.2 User languages

Even though a single language operating system has many attractions, it is not at all clear that this philosophy should be enforced on the users of such a system. On the whole, operating systems fall into three distinct categories with respect to user languages.

Firstly, there are the *single language* systems, where the user of the operating system is forced to write in the same language in which the operating system itself is implemented. Examples of such systems are TRIPOS where the language is BCPL, PILOT where the language is MESA, and MEDOS where the language is MODULA-2. With single language systems, the whole gearing of the operating system is towards this language, with system calls being represented by language procedure calls. Even if it is possible to implement other languages on such systems, this is often difficult and inefficient. R. D. Evans [Evans81] describes some of the problems of implementing PASCAL and ALGOL68C on TRIPOS, which is heavily geared towards every program being written in BCPL.

Secondly, there are the *preferred language* systems, where the user of the operating system is encouraged to write in the same language as that used by the system itself, but sufficient "hooks" are included to make it possible for other languages to be used in a clean and efficient manner. The major difference between this type of system and the single language version is that the interface to the operating system kernel is much less language specific, usually being implemented using "traps" or "supervisor call" instructions rather than by language procedure calls. Examples of preferred language systems are UNIX, where the language is C, MAYFLOWER [Hamilton84], where the language is CLU and CAP [Wilkes79], where the language is ALGOL68C.

Thirdly, there are the truly *language independent* systems, which allow the user to write in any language, neither enforcing nor encouraging his decision. On the whole, language independent systems tend to be large, and support multiple users each with different requirements. In such circumstances, it is not reasonable to restrict the choice of implementation language, and so all interfaces are sufficiently general to allow for complete language independence.

Given the power and generality of the new generation of personal workstations, there is no reason why they should not support implementations of many different languages. Similarly, as these machines become more common, so the requirements of their users diversify, making the ability of an operating system to support many different languages essential.

There have been arguments in the past about whether language independence causes a decrease in efficiency, simply because the interface between the user and the operating system must be more general. The generality means that there must be a well defined calling convention between the user programs and the kernel, implemented either through a subroutine call or a processor "trap" instruction. Indeed, the interface does have to be more general, with more dumping and loading of processor registers, and on the whole, trap instructions are slower than simple subroutine calls, since part or all of the machine state must be saved. Also, there is often a certain amount of decoding necessary when a trap is executed, whereas subroutine calls go to the correct entry point immediately.

The advantage of executing a trap instruction is that there is usually an implicit change of processor privilege state, which may be necessary in order to access peripherals, protected memory and so on. Single language systems, such as TRIPOS, force the processor to operate always at its highest possible privilege state, since there is no simple way of determining where the user code finishes and the operating system code starts.

Because of the inability to draw a clean line between what is in the user domain and what is in the operating system domain, it is very easy to become confused about what aspects of a design are operating system issues and what are language issues. TRIPOS, for example, as well as allocating each process a task control block, also allocates a stack and global vector as well, on the grounds that the task code, being written in BCPL, will require them. This is, in a sense, confusing two completely separate issues—the creation of a new task and its initial activation, and the allocation of resources to that task which are language specific. In fact, TRIPOS actually suffers because of this, since activation of a task happens within the kernel, with interrupts disabled. The activation process,

requiring two areas of store to be allocated and initialised, also runs with interrupts disabled, and the whole operation may take several milliseconds, during which time no interrupts can take place.

This ambiguity of definition is removed in a language independent system, since the interface between processes and the operating system kernel is clear cut, with a definite domain transition taking place when passing from one to the other. The interfaces between the kernel and the rest of the world should be well defined, and wherever possible consistent. There is a great advantage if the calling conventions of processes are the same as those for exception handlers, interrupt handlers and so on, since this allows any language to be used anywhere in the system. The topic of calling conventions is discussed in more detail later in this chapter.

7.3 Processes and coroutines

Using the process as the unit of operating system work is a concept that has been around for many years. More recently, there has been an increase in the interest shown about coroutines [Moody80], and their use in operating system design. B. J. Knight [Knight82] describes how coroutines can be used to subdivide work within processes into more manageable units, and discusses their use in helping to implement a multi-threaded remote debugger [Atkins83]. Some operating systems, notably MEDOS, have done away with the concept of independent concurrent processes altogether, with coroutines being used exclusively for scheduling purposes.

Before examining processes and coroutines in more detail, it is worth comparing their properties and limitations. Both processes and coroutines can be treated as individual units of "work" whose execution can be suspended at any time, while the processor performs some other function, and then resumed at some later time. They both require an environment which is saved when suspension occurs and is restored on resumption. The differences between processes and coroutines lie in two distinct areas.

Firstly, processes are capable of being pre-empted, in other words, if some event occurs which enables a more important process to start execution, then the current process is suspended forcibly, and is only resumed when the more important process has no more work to do. Coroutines, on the other hand, cannot be pre-empted, in that they continue to run until such time as they suspend themselves voluntarily.

Secondly, each process is run in a separate environment, and may be written in a different programming language. This means that a process context switch involves the dumping of the current machine state, (represented by the processor registers, program counter, status word and so on), and the loading of the new state. Unless the processor provides instructions to help with this, the job of dumping and restoring of machine states can be fairly lengthy, particularly if memory page tables must be updated as well. Sets of coroutines, on the other hand, tend to be written in the same programming language, and execute as part of the same process. A coroutine context switch is, therefore, much cheaper than its process equivalent, since much less of the environment needs to be changed.

7.3.1 Processes

Processes have several uses within an operating system, but by far the most important feature they provide is that they can be pre-empted, should this prove necessary. An example of this type of pre-emption is when the completion of a disc transfer or a character being typed on a keyboard causes a user process to be suspended in favour of a more important system device handler. In this way, it is possible both to handle devices, servicing them within their critical access times, and to allow the user to run programs which are heavily CPU intensive. Pre-emption is normally implemented using a priority based scheme, in which each process is assigned a value which enables its relative importance with respect to other processes to be calculated. In such a system, a process continues to execute until such time as it suspends itself voluntarily, or a process of higher priority is given work to do. This means that, with a pre-emptive system, it is always the most important process which is capable of running which is actually given the CPU.

Processes also have other uses, in that a process is often the unit of accounting and protection on multi-user systems, and so the process context switch can be used in order to control the resources being used and to protect users from one another. Similarly, since a process is a self contained unit, it is possible to treat processes as abstractions, holding sensitive or important data as part of their environments. In this way, data can be confined to the area in which it is used, with a well defined interface at which checks for consistency or authentication can be made.

7.3.2 Coroutines

Coroutines have similar properties to processes, except that once running, a coroutine cannot be pre-empted by another in the same process, and suspension is always voluntary and explicit. Coroutines are also simpler in concept, since

normally they are defined in terms of a programming language rather than an operating system. Context switches are therefore done at a language level, and hence tend to be not much more expensive than a simple procedure call. As aids to implementation, coroutines have four main uses.

Firstly, they are useful for separating the different parts of a multi-threaded program into several, single threaded units, with each unit being represented by a coroutine. For TRIPOS, B. J. Knight [Knight82] explains how, by careful redefinition of the BCPL function *pktwait* in terms of *cowait* (the coroutine suspension primitive), it is possible to encode a complicated multi-threaded program as a batch of simple, single threaded coroutines. The benefit here is a human one, since on the whole, people are used to thinking serially, and are very bad at imagining operations happening in parallel. There is a direct comparison to be drawn between coroutines and the use of "control blocks", which is another way of implementing multi-threaded programs. Take, for example, a protocol handler which deals with a set of I/O streams. Using the coroutine method, each stream would have an associated coroutine, and whenever a request came to the handler for a particular stream, it would be handled by the relevant coroutine. Using the control block method, whenever a request arrived for a stream, the relevant control block for that stream would be located, and the parameters stored in the block would be adjusted accordingly. In each case, the same operation is done, with the state held on the coroutine stack as local variables being exactly equivalent to the state held in the control block. The only difference is one of human conception, with coroutines making multi-threading easier to understand.

Secondly, since coroutines can never be pre-empted by other coroutines, their operation is guaranteed to be mutually exclusive, with the execution of one coroutine stopping the execution of another. Because of this, coroutines can handle shared data structures with impunity, since they need not suspend themselves until the operation on the structure is complete. Using coroutines rather than processes therefore has the advantage, in some cases, of removing the need for an interlocking operation, such as a message queue or a semaphore. For light-weight applications, mutual exclusion is much more efficient than either of the other synchronisation methods.

Thirdly, since the size of a coroutine stack frame is determined dynamically, there is the ability to use several coroutine stack frames to represent a stack whose size increases and decreases dynamically. Since entire environments can be saved on coroutine stacks, it is convenient to use these stacks chained together, as though they were "frames" on a much larger stack, with each frame

representing an execution level. In this way, control can be passed from one level to another, with the environment being saved before the call, and restored afterwards. An example of this kind of use is given in chapter 5.

Fourthly, it is possible to use coroutines to implement the interface between different programming languages. This seems strange on the face of it, since one of the properties of coroutines is that they are programming language specific, and this is one of the factors which makes them so efficient. This is true in most traditional implementations, but for a small amount of extra overhead, it is possible to design a set of coroutine primitives which are programming language independent. The approach suggested here was adopted by the GMK kernel (described at the end of this chapter), which unifies all kernel calling sequences, including those for coroutine manipulation. So long as the calling sequence used by the coroutine primitives is simple and consistent, it can be called from any language, and if the sequence used by the coroutine package is the same as that used by the operating system to enter newly created processes, there is no reason why each coroutine should not be implemented in a different language. Obviously, since the coroutine package claims to be language independent, it must save the context of each coroutine before it calls another. Because the other coroutine may be programmed in a different language, it is no longer possible to implement coroutine switching quite so efficiently as before, since the saving of the processor registers cannot be optimised. Having said that, the time taken to switch between coroutines of different languages is still much less than the time taken in a process context switch, and so using coroutines as the interface between different languages is an extremely elegant and attractive proposition.

7.4 Scheduling

One of the requirements of any operating system which has multiple processes is that these processes should be scheduled in such a way as to share out the CPU resources as the application dictates. Much has been written about the problems of scheduling for large systems, particularly those which support many interactive users. In such cases, what is important is the maximum utilisation of the CPU, and the sharing out of CPU resources to each of the user processes in a fair and equal manner. On a single user machine, all these parameters change, and the issue of scheduling is much simplified.

The first, and most important point is that on single user machines, the whole of the CPU power is potentially available to service the needs of that user,

unlike a multi-user time shared system, where the CPU must be shared by many competing users and batch jobs. The concept of "fairness" in scheduling is therefore removed, since unless the CPU is idle, it is bound to be executing at least one of the user's processes, and then only in preference to others belonging to him. The fact than a user has the entire CPU available means that the factors important to him are the overall throughput of his system and the amount of time he has to wait for operations to complete. Scheduling on a single user system is therefore not a matter of accounting, but one of sensible resource management in order to maximise convenience and minimise real time delay.

The second point is that, on a single user system, the number of resident processes is usually very small (on TRIPOS, typically less than ten), and so simple techniques which would be inefficient on large systems become more attractive on personal computers.

Of the many different scheduling algorithms available, there are three which have particular relevance to single user systems. The important observation is that no one algorithm performs perfectly under all possible conditions, and so the type of scheduler chosen for an operating system depends heavily on the application involved. The two applications which have the most diverse requirements are also the two which are the most common. Firstly, there is the light-weight, multi-tasking kernel, which is the basis for the real time servicing of devices, such as discs or networks. Secondly, there is the operating system which provides a service to a user, allowing him to edit files, compile and run programs, interrogate databases and so on. The requirements of these two types of system require very different techniques, in order to maximise their efficiency and performance.

7.4.1 Unique priority

The first type of scheduling which is worth investigating is the pre-emptive "unique priority" system, as adopted by TRIPOS. In many ways, this is the simplest algorithm to apply, and one of the best to use in a general purpose operating system. Each process is given a unique priority, usually a positive integer, which defines its importance in relation to all the other processes in the system. The scheduling algorithm itself is very simple—the runnable process with the highest priority is always the one which is given the CPU. This process continues to run until either it blocks, or another process with a higher priority becomes runnable. This may happen in response to a device interrupt, or the active process giving work to an inactive, higher priority process. When this happens, the active process is effectively pre-empted by the higher priority

process, and so important jobs (such as handling fast devices) can be accomplished, even if the user processes are CPU intensive.

The simplicity of this system means that only one queue is required, and that is a queue of all the process control blocks, sorted with respect to their priority. In order to activate a process, the chain of process control blocks is scanned from high to low priority, with the first "runnable" process being the one chosen. Since blocked processes are never activated and runnable processes are only ever activated in a priority determined order, there is no need to keep a separate queue of runnable process control blocks.

The system is indeed simple, but has one major drawback, which arises if any of the processes go into a CPU loop, blocking either never or very infrequently. It has already been stated that this does not affect higher priority processes, since these will pre-empt a CPU bound process whenever necessary. This is not the case for processes of lower priority though, since they will never be able to pre-empt the active process, and since the process very rarely blocks, the lower priority processes suffer from partial or total CPU starvation.

This is not a problem to the system processes, since they can arrange to have priorities greater than those of the user processes. The user may run into difficulties though if he has more than one active process, since for the scheduler to work properly, he must only ever have one CPU bound process, and he must arrange for that process to have the lowest priority of all processes in the system, otherwise starvation will occur. Apart from the obvious case of the user running CPU bound programs, there are other occasions when the single priority pre-emptive algorithm performs badly.

The first case occurs when there is more than one incarnation of a single process, each performing the same function. This will be the case if, for instance, there are two file handler processes, each handling different physical discs. It will also happen in the case where the operating system kernel is being run without a user, performing a real time service, such as a network fileserver. Within such a server, there are likely to be several processes all of which perform exactly the same function—handling disc objects or network transactions, for example. With this sort of arrangement, it is artificial to give each of the different incarnations a different priority, since no single one is inherently more important than any other. In fact, being forced to assign unique priorities actually hinders such systems, since the incarnation which happens to have the highest priority will be the one which is given the CPU in preference, potentially causing partial starvation to the others. The same is also true for the incarnation with the next lowest priority, as it will pre-empt any lower ones, and the incarnation with the

lowest priority may never see the CPU at all. The problem can be solved, either by juggling the process priorities every so often, or by running each of the processes as coroutines within the *same* process.

The second case is where it is necessary to support more than one user on the same machine. This may seem a rather odd requirement for a "single user" operating system, but it is often the case that it is convenient to allow one user to be logged on via the intimately connected console, and another via a local area network. Another example of where multi-user operation is desirable is to allow many people to access a central resource, a database for instance, either from a series of consoles, or from network connections. This is in fact a special case of the previous problem, but more important in reality, since the clients which are kept waiting due to the CPU starvation are not computers but people. If two people are actively working on a system then, simply by running a CPU intensive program, one can effectively lock out the other.

7.4.2 Non-unique priority

The second type of scheduling is a variation of the previous algorithm, with the extension that processes of equal priority are allowed. Pre-emption of low priority processes by high priority processes occurs as before, but processes of equal priority are handled in a "first come, first served" order.

This method of scheduling removes one of the problems of the unique priority algorithm, since multiple incarnations of the same process can each be given the same priority, and so will be handled in the order that events actually occur, rather than in an artificial priority ordering. Using equal priorities does not solve the multiple user problem, since processes which have the same priority are handled in order, and until the active process releases the CPU by blocking, the next one awaiting execution (with either the same or a lower priority) suffers from starvation.

For systems which have to respond well to real time events, such as network servers, this type of scheduling has much to recommend it. An example of a system which uses this method of scheduling is THOTH, which is designed specifically for real time operation.

7.4.3 Round robin

The third type of scheduling which is worth investigating for use on single user systems is "round robin". This is very simply the sharing of the CPU between processes of effectively equal priority, by partitioning the CPU time available into slots, or *time slices*, and then giving the CPU to each process in turn, changing to a new one whenever a process blocks or a time slice expires. As

with the previous algorithm, processes are either blocked or runnable, with the runnable processes being held on a separate queue. When a process is wakened after being blocked, it is added to the end of the runnable queue so that it can be activated during the next available time slice. When the active process either blocks or is timed out, the next item on the queue is activated, and if the outgoing process is still runnable, it is re-inserted at the end of the queue. In this way, the CPU is shared out in a fair manner between all the processes which are ready to run.

This algorithm is precisely what is required for time sharing, and has been adopted by many multi-user operating systems, such as IBM's OS/360 and UNIX. What the round robin algorithm fails to take into account is the fact that certain processes have more importance than others. These processes have to wait their turn, which may be a long time in real terms, depending on the number of runnable processes and the length of the time slice. Devices with critical access times are difficult to handle when round robin scheduling is used, and operating systems which employ this algorithm are typically fairly bad at handling real time events. This problem can be solved by promoting processes to the head of the runnable queue rather than adding them to the end, and by allocating different time slice values to individual processes, where the length of the slice is inversely proportional to the process's importance.

7.4.4 Which scheduling algorithm?

As explained above, each of the three algorithms have their own particular advantages and disadvantages. The one chosen by an operating system designer should very much depend on the application which that system is to be used for, but if a general purpose system is required, then some compromise is necessary. The factors which affect the choice of algorithm are the number of users a system must support, the number of processes and its required real time response.

The "unique priority" system is the simplest of the three to encode, but is particularly bad if either the number of processes is large (since the length of the chain of process control blocks to be scanned for each scheduling operation would be large), or if there is a possibility of one process causing CPU starvation for others. In other cases, the behaviour of this algorithm is good, and real time response can usually be guaranteed.

The "non-unique priority" system is unaffected by the number of processes, since a separate queue of runnable processes is kept. The algorithm is easier to encode than round robin, since it requires no real time clock, but even though equal priority processes are allowed, they are handled in the order in which they

are ready to be run, which does not remove the possibility of CPU starvation. Real time response from such a system is good, since it is always the most needy process which is executed at any one time.

The "round robin" system is also unaffected by the number of processes, since there is a separate queue for those processes which are runnable. It is more complex then either of the priority systems to encode though, since a real time clock is required in order to define the time slices. Round robin is ideal for systems supporting multiple users, since each user is guaranteed a fair proportion of the available CPU time. What the round robin system is particularly bad at is responding to real time events, which have to wait their turn in the time slicing queue. This can be improved by careful queue manipulation and time slice tuning, but is unlikely to be adequate in situations where a regular, fast real time response is required.

7.4.5 A compromise solution

Unless one of the very basic methods described above is adequate for a particular application, a compromise algorithm can be used to combine the advantages of more than one. This should only be done if a truly general purpose system is required, because unless that is the case, it is almost always better to sacrifice a small amount of performance in order to maintain simplicity.

The following solution is a combination of the "non-unique priority" and "round robin" algorithms, and provides a good real time response, along with the ability to share time between processes of the same priority. The algorithm requires a queue of processes which are ready to run. At any one time, it is always the process at the head of this runnable queue which is given the CPU. The process continues to run until one of three events occurs. Firstly, if the process blocks, then it is removed from the runnable queue, and the next item in the queue is executed instead. Secondly, if something occurs which causes a higher priority process to become runnable, then this process is added to the head of the runnable queue, and then activated. Thirdly, each time a new process is activated, a "countdown" value is set in its process control block, and each time the processor is interrupted by the real time clock, this value is decremented. If this value reaches zero, the priorities of the active process and the next process on the queue are compared, and if they are the same, the active process relinquishes the CPU. Since the suspended process is still runnable, it is re-queued so that it will run again at some later time.

A variation on this algorithm is adopted by the ACCENT [Rashid81] and MAYFLOWER kernels, which allow a small number of priority levels, and then

apply a time-slicing scheduling algorithm for processes at the same priority level. Restricting the number of levels may enable the implementation to optimise performance by splitting the queue of runnable processes into several queues, and thus reduce the overhead of inserting a process which has been woken up.

7.5 Calling conventions

One of the most important aspects of operating system design is the nature of the interfaces between different parts of the system. The most important feature about all operating system interfaces is that they should be *consistent*. This may seem an obvious statement, but it is surprising how many system designers ignore the consistency issue in an attempt to define specialised interfaces for specialised jobs.

There are many levels of operating system interface, and the ones investigated here apply to the operating system designers and system programmers, hence the interfaces in question are those at the lowest level. It is the case, however, that if the design at the lowest level is clean and consistent, then this philosophy tends to percolate through the rest of the system. This is particularly important for two main reasons. Firstly, these low level interfaces tend to involve assembly code sections, and so having a consistent calling sequence simplifies their implementation. Secondly, so long as the interfaces are consistent, the code can be written to be context independent, with the same calling sequence being used no matter what the occasion. There are four main areas where it is imperative to use the same calling sequence, if the operating system is to be truly general.

Firstly, there is the calling sequence for newly created processes. When a process is created, parameters to the kernel *create process* primitive should include an entry point to the process, and an argument to be passed to that process when it is activated. On activation, the new process should effectively be called as a subroutine of the operating system, given its argument and return address in a standard way. The process then runs to completion, and on returning from the subroutine call, should then be deleted by the system. If a result (return code) is passed back from the process to the operating system, then this should also be done in a standard manner.

Secondly, there is the calling sequence for device interrupt handlers. Rather than having each interrupt handler programmed in an *ad hoc* manner, it is better for the interrupt to be trapped by the operating system, and for the kernel to call the interrupt handler for that device as a subroutine, in exactly the same way as

it calls newly created processes. There should be kernel primitives to create and delete interrupt handlers, and when an interrupt handler is created, an argument should be given to the kernel which is then passed on to the interrupt handler each time it is activated. Just as before, the calling sequence should be a subroutine call, with the argument and return address being passed in a standard way.

Thirdly, there is the calling sequence for general exception handlers. These are very similar to device interrupt handlers, but are called whenever some sort of error occurs, such as division by zero, addressing non-existent memory and so on. Unlike interrupt handlers, exception handlers should be activated on a "per task" basis, so that individual errors can be trapped and treated cleanly. In this way, it is possible to implement a clean debugging environment, with the exception handler (debugger) being called whenever an error occurs. As with the previous two examples, the calling sequence should be consistent, and exception handlers should again be called as subroutines of the operating system. Using the exception handler mechanism, such things as inter-process signalling and asynchronous break handling can easily be implemented.

Fourthly, in order to make the consistency of these calling sequences especially useful, coroutines should be called in exactly the same way as processes, interrupt handlers and exception handlers. If this is the case, then it is possible to take advantage of the mechanism discussed earlier, in other words, the use of coroutines as interfaces between programs written in different languages.

Obviously, the exact calling mechanism to be used depends very heavily on the facilities provided by the underlying hardware, but no matter what the instruction set is, the principles are the same. All programs, whether they are processes, coroutines or handlers, should all be called as subroutines of the operating system, with the only valid means of exit being the return address passed by the subroutine call. In this way, the kernel can arrange to handle such things as machine state dumping and restoration, interrupt enabling and disabling, and so on, in fact all the things which would be common to all systems programs can be handled centrally by the kernel.

Having decided always to call these programs as subroutines, it is essential to use the same conventions for register allocation, workspace pointers and return link addresses, as well as the passing of arguments and results. The general rule is that a program should not have to know whether it was called as a process, a coroutine, or in response to an error or a device interrupt.

As an illustration of the points made here, included are two calling sequence conventions for different machines, which obey the consistency rules. The first is

a real example, and taken from the GMK kernel for IBM 370 machines described at the end of this chapter. The other is hypothetical, and is the suggested equivalent of the 370 version for the Motorola MC68000. The processors have different architectures, but it can be seen that the principles involved in the design of the calling sequence are exactly the same. The IBM 370 conventions are:

R15 Base address of called program

R14 Return link to operating system

R13 Pointer to workspace for register dump

R1 R0 Argument and result registers

Thus, each 370 program looks like:

```
        USING  PROGRAM,R15     Establish addressability
PROGRAM ST     R1,....         Save argument
        ....
        ....
        L      R1,....         Load result
        BR     R14             Return to the operating system
        DROP   R15             Discard base register
```

For the 68000, the calling sequence would be just as simple, but since the processor provides a stack and "program counter relative" addressing, there are fewer processor registers involved. The 68000 conventions might be:

A7 Pointer to stack, holding operating system return link. The stack is at least big enough for one entire register dump.

D1 D0 Argument and result registers

Thus, each 68000 program would look like:

```
PROGRAM MOVE.L D1,....         Save argument
        ....
        ....
        MOVE.L ....,D1         Load result
        RTS                    Return to the operating system
```

The calling sequences suggested are simple to implement and convenient to use, but the most important fact about them is that they should be adhered to in all possible circumstances. If this is done, then the result is total language independence throughout the whole system, with inter-language communication thoroughly feasible.

7.6 Inter-process communication

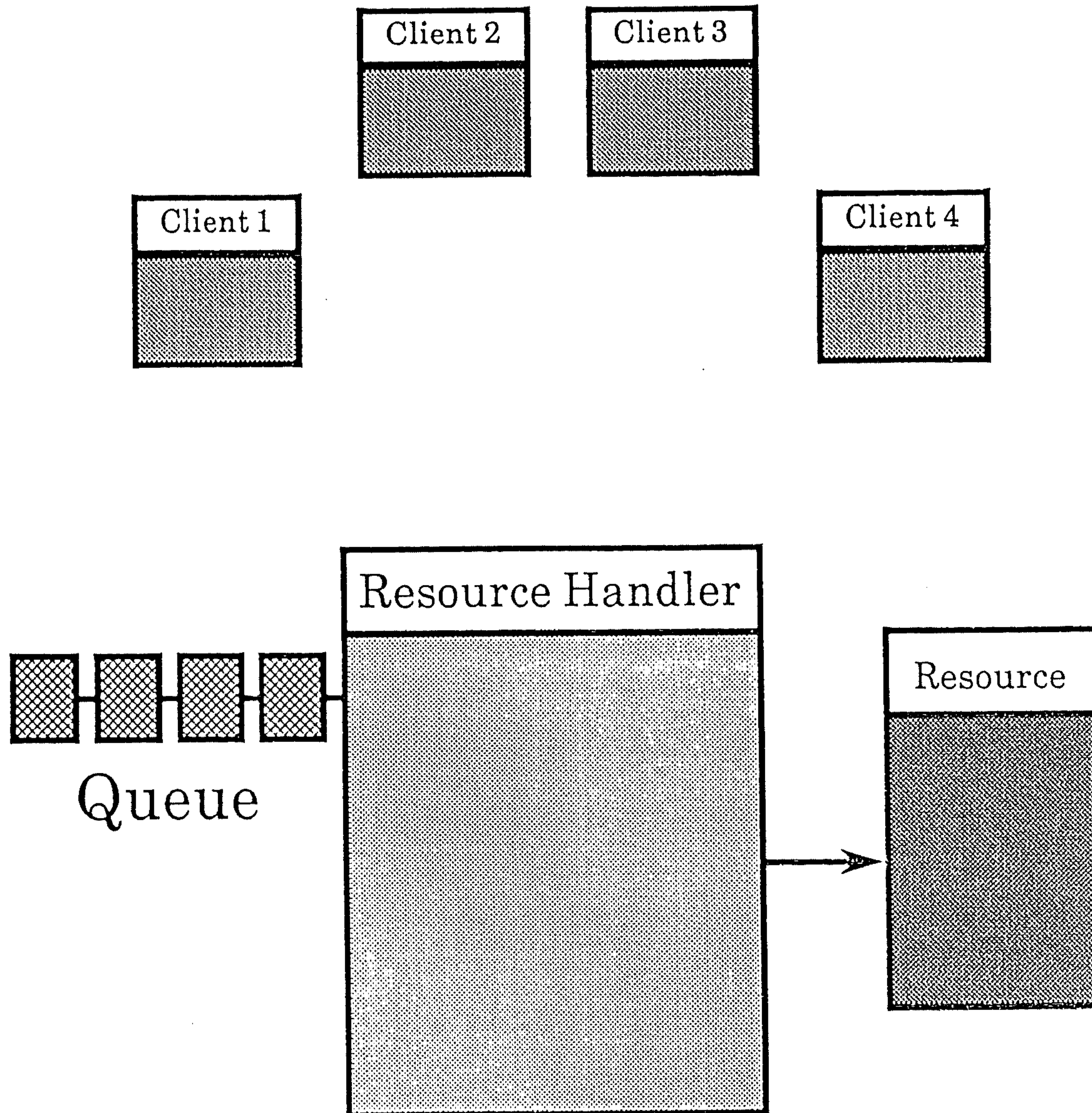
As with the topic of scheduling, much has been written about different types of data sharing and inter-process communication. There are two main schools of thought. Firstly, there is the group of people which believes that inter-process communication should be performed using *messages* which are sent and received by processes. These messages are held on queues, and processed usually in a "first come, first served" order. Secondly, there is the group of people which believes that inter-process communication should be performed using shared data, which is handled by special procedures called *monitors* [Hoare74], which can only be entered by one process at a time. Other processes waiting to use the monitor must be queued, and as with messages, are usually handled in the order in which they are queued. H. C. Lauer and R. M. Needham [Lauer78] put forward the view that there is no practical difference between the two methods, with both functionality and performance being exactly equivalent. This view is illustrated simply by looking at figure 7.1, which is a diagrammatic representation of both different types of system. If the handler is a process, then its attached queue is a set of *messages* waiting to be received, dealt with, and returned. If the handler is a *monitor* then its queue is a set of process control blocks, representing the processes which are waiting for the monitor lock to become free. In either case, the effect is the same, with the only difference being how the mechanisms are conceptualised by their human users.

7.6.1 Messages

Messages are simple blocks of memory containing information to be passed from one process to another. Message primitives allow messages to be sent and received, and since message return is asynchronous, having once sent a message, there is no need to block waiting for the reply.

The message passing system is easy to conceptualise, since it has many analogues in every day life. Each person working with operating systems is used to the idea of posting letters which cause some form of work to be done on his behalf (while he goes off and does something else), with the reply arriving some time later. This ease of conceptualisation is one of the major advantages which message systems have over their monitor counterparts, since programmers find it natural to think in terms of messages and replies. This point cannot be stressed too highly—a system which is easy to understand is also one which is easy and convenient to use.

Using message passing as the basic primitive in inter-process communication has one extra advantage over the use of monitors and procedures—that the



General Synchronised Access to a Resource

Figure 7.1

mechanism is extendable to cover communication over computer networks. Two examples of systems which use this fact to make program distribution easier are MUSS [Frank79] and ACCENT, both of which use small, structured messages for communication between virtual and real machines.

Many systems, both large and small, have used message passing as their inter-process communication mechanism. Examples of systems which employ this technique are IBM's OS/360, the University of Waterloo's THOTH and Cambridge's own TRIPOS. The factor which unites these systems is that they are implemented in untyped, sequential languages (either assembly code, BCPL or one of its derivatives), and that all checking, if any, is performed at run time. Using messages has the advantage that inter-process communication is language independent, but adds the overhead of run time parameter checking. Two of the examples cited above, THOTH and TRIPOS, have been used extensively for implementation of network services, and the logical extension of operating system messages into network packets has made their use highly successful. Because the message passing system is so easy to comprehend by beginners, both systems have been used in the undergraduate teaching of real time operating system principles.

Messages have two major problems when it comes to implementation. Firstly, as with network packets, the amount of information held in a message depends heavily on the application involved. Some systems, for example TRIPOS, allow messages to be of arbitrary length, thus removing any restriction. TRIPOS, however, is only able to do this because messages are objects owned by the user rather than by the system, and are allocated from the user's memory space. If some protection is required (through copying) or if allocation of messages is from a system pool, then it is necessary to enforce a maximum message length. THOTH, for instance, restricts messages to 8 words. The result of this restriction is that objects, rather than being passed by value within the message, tend to be passed by reference, with messages containing pointers to objects. When this happens, there is the possibility that, by the time the message is processed, it will be in a different virtual address space to the data which it references. Not only that, but the extendability of inter-process messages into network packets is also lost. Secondly, since the values passed in messages are simply bit patterns, any data typing must be performed dynamically at run time. There is also the problem that, since processes may be implemented in different languages, there may be disagreement as to how certain data types (for example, strings) are represented.

7.6.2 Monitors

A monitor is a block of program code, usually a set of procedures, which operates on a shared data structure or device. Such a concept is not new, since the language SIMULA [Dahl66] effectively had this with its *class* mechanism, and more recently, object-oriented languages such as CLU [Liskov81] have become popular. The difference between a monitor and a class in SIMULA or a cluster in CLU is that procedures within a monitor may be defined to be "mutually exclusive", in other words, only one of the procedures may access the shared data structure at once. Before being able to execute such a procedure, a process must first obtain a monitor *lock*, that is a token of entitlement to enter a procedure within a monitor. If the monitor is in use, then the process is held on a queue until the lock is freed, at which point it is dequeued and re-activated.

Operations like this have analogues in every day life, and so are easy to understand. Visiting the doctor, for instance, involves either going straight into his surgery to see him if he is free, or waiting in a queue if he is not. Unfortunately, as it stands, the monitor mechanism is not quite as general as the message passing case, since entering a monitor always causes a process to block if the monitor is in use. In order to achieve the asynchronous operation provided by the message queueing mechanism, the operations *fork* and *join* are required, to enable "half" the process to be blocked waiting for the monitor while the other continues to work, followed by a joining up and re-synchronisation sometime later. Taking the "doctor" analogy further, this would imply a cloning operation at the surgery, with one clone waiting in the queue, and the other clone continuing to work. The two clones would then meet up some time later, and reform into one person again.

The analogy is far fetched, and since there are no *fork* and *join* operations in real life, it can be difficult for programmers to think in this way. Compared to the simplicity of message passing, monitors seem unduly complicated. There is no logical extension of monitors which encompasses networks either, since remote execution of procedures [Birrell84, Hamilton84], although possible, is less easy to achieve.

Having said this, monitors have an advantage in type checked programming languages which have concurrency built in, such as MESA. With such languages, since inter-process communication is by means of shared, type checked data through type checked procedures, it is possible to perform all the consistency checking at compile time, and thus cut run time overheads. Because of this, operating systems which use monitors tend to be written in a single, concurrent programming language which has a high degree of compile time type checking.

The PILOT and MAYFLOWER operating systems are implemented in exactly this way, with the type checking being performed by the language system, and the run time implementation of queues and monitor locks being implemented by the underlying multi-tasking mechanism.

Even though they seem less natural than messages, monitors and the use of procedures are extremely attractive propositions when it comes to operating system design. This is primarily because of the *semaphore* mechanism [Dijkstra68] for ensuring single access to monitor procedures. Semaphores are simple and cheap to implement, meaning that monitors can be included easily, either as built-in additions to a programming language as in MESA, or as a procedure library attached to a language run time system, such as the monitor implementation for BCPL [Lister76].

One of the reasons why semaphores are so attractive is that they allow uncontested access to a resource, for example the free store chain in a store manager. Without semaphores, access to such a structure must be controlled, either by having a store manager process (which guarantees mutual exclusion at the cost of two process switches), or by running the primitives which access the structure always with interrupts disabled. There are many operating system and user structures for which it would be convenient to control access using semaphores and monitors.

The same rules would apply as to the other types of program discussed with regard to calling conventions. In this way, monitors can look like, and be implemented as if they were processes, coroutines and so on.

7.7 Naming and location of services

In many ways, operating systems suffer from the same problems as networks when it comes to location of services, particularly those which are in any way dynamic. Each process (or monitor) must be given some identifier, so that it can be located whenever it is required. This problem is not so important if the operating system is written in a single, concurrent programming language (usually with a monitor facility), since the naming of services is handled at compile time. For other types of system though, there must be some way of locating a service within a machine.

One way of doing this is to allocate to each service a unique identifier, which would correspond to a message *mailbox*, in other words, to fix the place to which a message should be sent in order for a particular function to be performed. The operating system could then keep tables of these identifiers, with each available

service having an entry in them. An example of a system which uses such a mechanism is ACCENT, where messages are sent to *ports* (mailboxes), and can then be handled by any process which has a "receive" capability for the relevant port.

In order to ease implementation, rather than sending a message to a general mailbox, it is convenient to arrange for messages to be sent directly to processes, with a function code within the message to allow demultiplexing to take place in the destination process. This cuts the possible number of message destinations down enormously, but does not remove the problem of locating the relevant process in the first place.

For static services, for example the console handler and filing system processes, it is reasonable to used fixed identifiers which are globally available, and defined to be the same on all machines. This, however, is not satisfactory when it comes to dynamic services, since this would force fixed slots to be allocated for every possible service process. A solution to this problem is to give each process a unique *name*, as with network services, and provide mapping functions to convert between process names and process identifiers dynamically, in very much the same way that a nameserver provides such a mapping service for a network. Even though it was never designed as such, the TRIPOS assignments list provides exactly this mapping, but in a rather *ad hoc* manner. A better solution is to associate with each process a name which defines that process uniquely, and is assigned to the process when it is created. It is then possible to put the name in each message, and for the kernel to process the name mapping on the fly whenever a message is sent. In practice, this is an inefficient way of proceeding, and it is better for the kernel to provide explicit mapping primitives, with the name being used to *locate* the service and the process identifier to *access* the service. Associating a name with a process carries little overhead, and experience with TRIPOS has shown that expansion would have been made easier had this facility been available.

7.8 Storage allocation

As with many of the other issues raised in this chapter, storage allocation is a subject which has been well researched before, but mainly in the context of large, multi-user operating systems. Many of the new algorithms could not be applied to small single user machines though, because they often wasted areas of store, which were sacrificed in order to improve allocation time. This was impractical on such small machines, as memory was scarce and so had to be conserved at all

costs. The other issue which was important on small sytems was the size of the largest contiguous area of free store—fragmentation is a problem on any system, but badly allocated store would make these small machines unusable.

Since program space was also at a premium, the simplicity of the free store allocation system was also important, with the result that the simplest possible algorithms were used. Two simple methods suggested by D. Knuth [Knuth73] are *first fit* and *best fit*. With both algorithms, a single chain of allocated and free store is kept, and scanned linearly. If a block of size n is requested, then first fit returns the first block in the chain which is at least size n, and best fit returns the block which is closest to size n while still being large enough. The first fit algorithm is simpler, and in most cases, performs better. Since the best fit algorithm must always scan the entire store chain before a decision can be made, first fit is bound to win, since it is likely that a free block of the required size will be found by only scanning part of the chain. Also, since best fit always minimises the wasted store when a block is split, it tends to create large numbers of small, useless blocks which are never coalesced, causing store fragmentation.

The choice of simple algorithms is only reasonable while two conditions hold. Firstly, the length of the store chain must be short enough that the act of scanning it is not a disastrously slow process. Secondly, only while memory is scarce is it necessary that the algorithm used should be simple—as soon as more store is available, code size and wasted memory both become less important.

TRIPOS uses a single chain "first fit" approach, and this worked well on the machines that the operating system was originally designed for. When TRIPOS was moved onto newer, larger machines, the scaling up of the memory size had an accompanying effect on the length of the store chain, with the average length of the chain increasing from around 120 blocks on LSI4 machines to around 400 blocks on 68000 machines. Since the free store chain is a data structure which must be updated atomically, most of the scanning is performed with interrupts disabled. In an attempt to ensure the real time handling of peripheral interrupts, each time around the scanning loop, interrupts are enabled for one instruction, allowing interrupt events to be handled. If the interrupt causes a re-schedule (which is quite likely) and the newly scheduled process *also* attempts to access the store chain, then the interrupted process must start its scan at the beginning again. To measure how often this happens in practice, a monitoring version of the TRIPOS kernel was put into service, which logged each time a store chain scan had to be re-started. The results showed that restarts happened very rarely in normal running—the maximum recorded total was 29 for a machine which had been running for a whole day—with the average number of restarts being around

7, most of which happened while the system was in its initialisation phase.

Even though restarting the scanning of the store chain is a rare occurrence, the allocation of store is still not an efficient process. Figure 7.2 shows the measurements carried out on around one hundred 68000 TRIPOS sessions, whose store chains were inspected remotely while the systems ran normally. The graphs show the number of links in the store chain which must be followed in order to allocate 1, 10 or 100 blocks of various different sizes, with the frequencies being the average number of links scanned per block allocated.

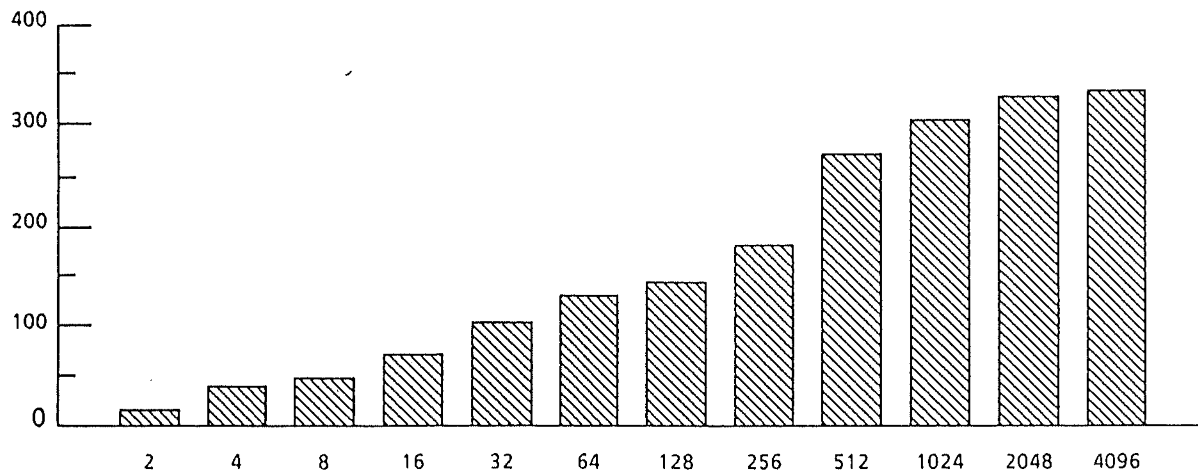
The interesting points to note are the discontinuities at block size 256-512 for 1 allocation, and 32-64 for 10 allocations. These indicate the points at which the free blocks of a required size within the store chain are used up. Before the discontinuity, blocks are allocated from free areas within the store chain itself. After the discontinuity, blocks are always allocated at the end of the store chain, and so the rate determining factor is the overall length of the chain itself.

Figure 7.3 shows the relative frequencies of blocks of different sizes within the store chains, which are either free or allocated. The interesting points on this graph are the peaks in the number of allocated blocks of size 8 and 512, which correspond to stream control blocks and I/O buffers respectively. Another point worthy of note are relative discrepancies between the numbers of free and allocated blocks of size 2, implying that these small blocks are created as the result of breaking up larger ones, and are then never used or coalesced.

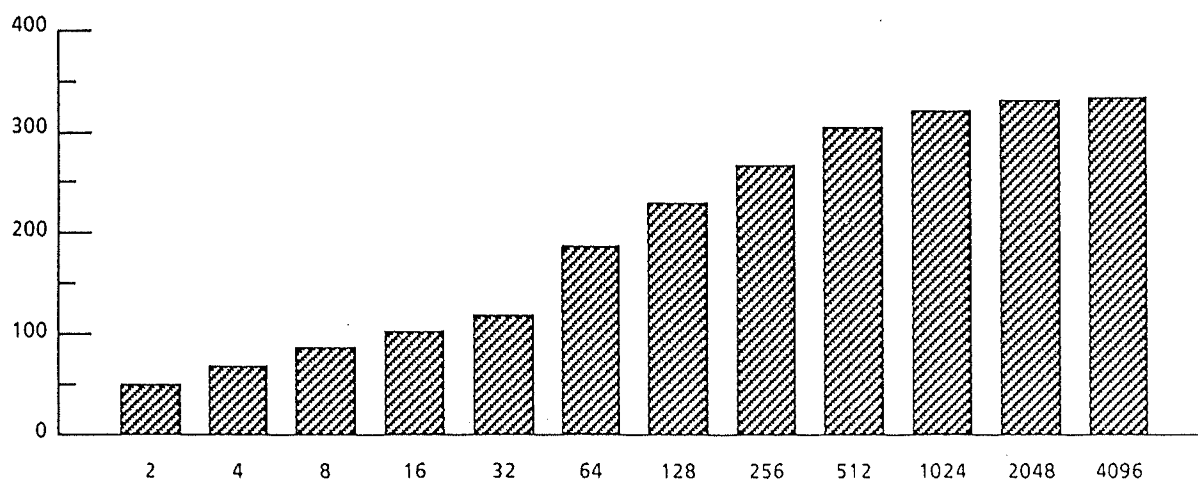
The reason for performing these measurements on the simple TRIPOS first fit algorithm is that, out of the results obtained, suggestions can be made as to how this simple algorithm can be improved, or how a new one could be used to better effect.

7.8.1 Dual list first fit

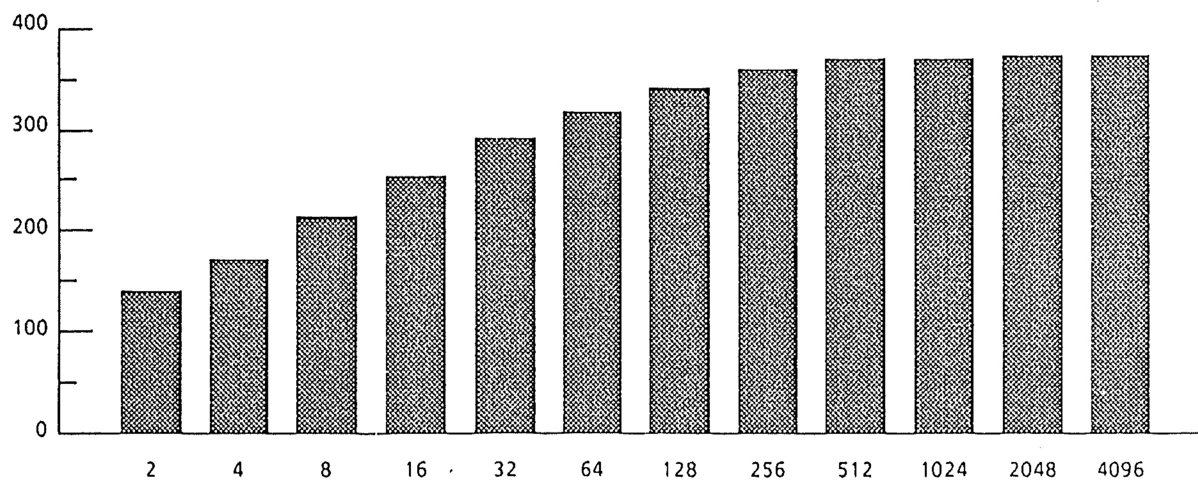
The "dual list first fit" algorithm [Wilson85], is a variation of the standard "first fit" approach, except that the free store is kept on a separate chain to the allocated store. In fact, this is a direct extension to the TRIPOS algorithm, and can easily be used to replace the standard one. The dual list method makes use of two important factors. Firstly, it assumes that the number of free blocks in the store chain is small, compared to the total number of blocks. (The observed distribution under TRIPOS is that 8% to 15% of the store chain is made up of free blocks). Secondly, it works on the assumption that, since the number of words allocated is always even (due to the least significant "size" bit being used as a flag), there is always at least one spare word available in every free block.



1 Block

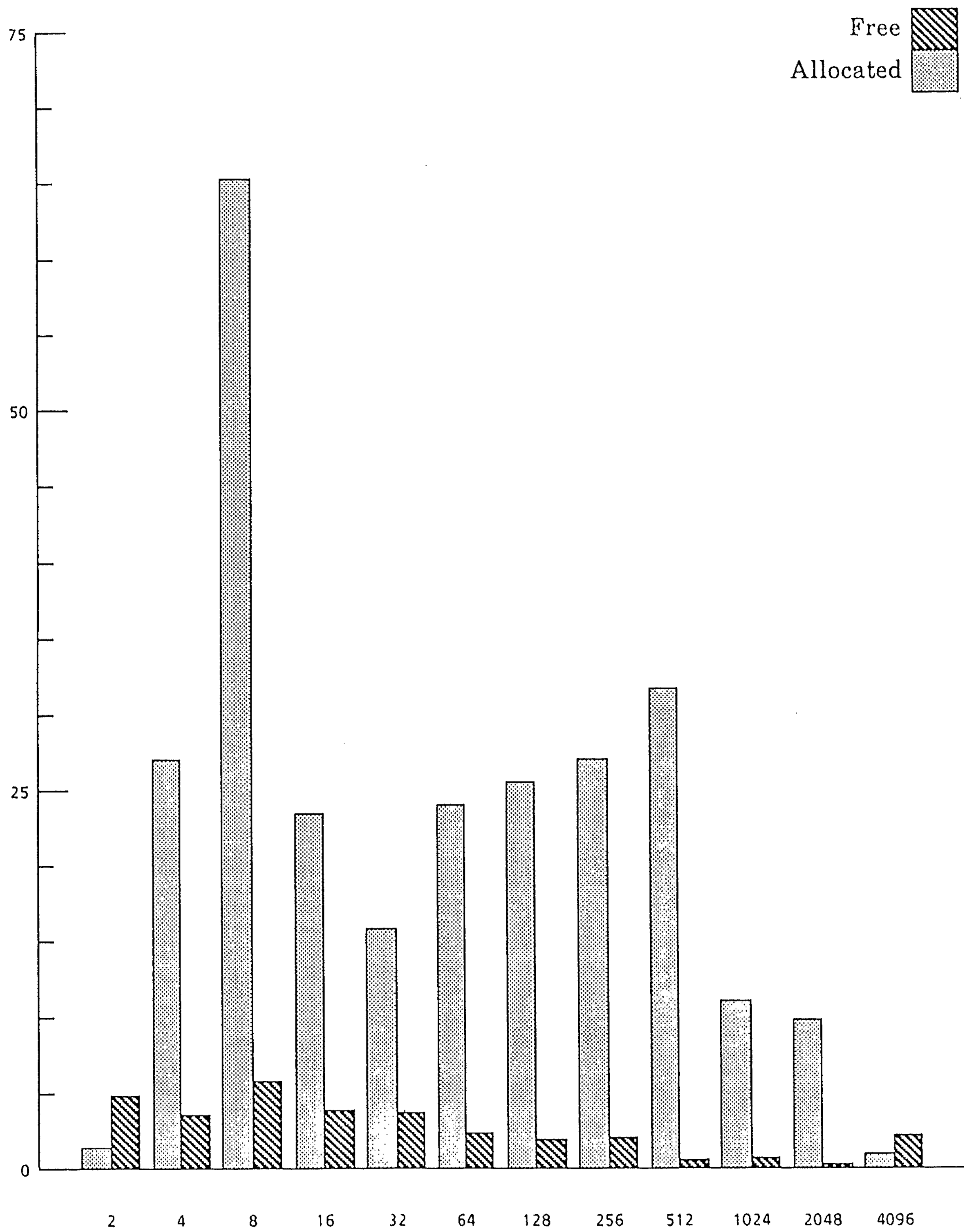


10 Blocks



100 Blocks

Figure 7.2



Distribution of Allocated and Free Store

Figure 7.3

Using these two pieces of information, the second word of every free block is used to chain these blocks together, with the root of the free chain being kept separately from the root of the whole chain. The effect of arranging the blocks in this manner is twofold. Firstly, since the free chain is much shorter than the whole store chain (and contains only free blocks anyway), the number of chain items to be scanned is much smaller than before. Through cutting out unnecessary scanning, the cost of allocating blocks can be reduced to around 10% of its original value. Secondly, since allocating a block from the free chain can never actually increase the length of chain, the speed of allocation is not determined by the total number of allocated blocks. In fact, if a block of exactly the right size is found in the free chain, the length of the chain actually decreases.

The main disadvantage of such an algorithm is in freeing allocated blocks, which must be re-inserted into the free chain, as well as being flagged as "free". This involves a scan of, on average, half the free chain, in order to find the correct place to re-insert the block. There is a hidden advantage here though, in that to re-insert the block properly, the previous and following blocks must be obtained, and so coalescing adjacent blocks can be done trivially, thus minimising the length of the free chain.

Overall performance of the dual list system is difficult to measure, since it is determined by the number of blocks being freed, rather than by the allocation time. Working on worst case figures, the whole free chain must be scanned both for allocation and deallocation, so if the total length of the store chain is n blocks of which 10% are free, then $0{\cdot}2n$ operations must be performed in order to allocate and free a block. For the simple first fit algorithm to match this performance, a block of the correct size would always have to be found within the first 20% of the store chain, which is not the case in practice. In fact, when the store chain itself is saturated (the point of the allocation discontinuity) it is the total length of the chain which determines performance, and so n operations must always be performed for each allocation. Even though deallocation is simpler with the single list algorithm (being effectively a null operation compared to allocation), this still gives the dual list algorithm a factor of five advantage at worst case.

7.8.2 Buddy

If the amount of memory is large, and the speed of allocation is more important than memory utilisation, the "buddy" method [Knuth73] is extremely attractive. In the buddy system, only blocks whose size is a power of two are allocated. When allocating a particular block, its size is always rounded up to

the next power of two. The beauty of this system is that, if the memory size is 2^n, then only n possible different sizes of block exist, and so the free blocks of each size can be kept on one of n lists. Allocation is therefore straightforward, since it is likely that a block of the correct size will already be free. If not, then a block of the next size up is split into two, with one of these blocks being allocated, and the other added to the free chain. The two blocks created in this process are called "buddies", and when both buddies have been freed, they are then coalesced back into the original sized block.

The buddy method is highly attractive, so long as the size of the memory is such that it does not matter if up to half of it remains unused. If that is the case, and very fast allocation of memory is required (for instance, any kind of real time service), the buddy system has much to recommend it. There are many modified "buddy" systems in operation, where block sizes other than the simple power of two are allowed. If certain block sizes are prevalent in a particular application, it is reasonable to restrict allocation only to blocks of these sizes—this is not strictly a "buddy" method, but has the same effect since it involves the small fixed number of block queues.

7.8.3 Cartesian tree

The cartesian tree method for dynamic storage allocation is proposed by C. J. Stephenson [Stephenson83]. Rather than using a simple chain of blocks linked together in address order, he suggests that a sorted *tree* of free blocks should be kept, with the following being true for each node of the tree.

- The descendants on the left (if any) have addresses less than that of the current node, and the descendants on the right (if any).
- The size of the current node is greater than that of all its descendants.

In other words, the blocks are held in a tree, sorted first on size, and then on address. In this way, the largest block of free store is always at the top of the tree, with all other blocks appearing as leaves. There are two possible allocation strategies with such a tree.

"Leftmost fit" is roughly equivalent to first fit, in that the leftmost node in the tree which is of sufficient size is allocated. This has the effect of allocating blocks in the low area of memory, leaving the large blocks in the high area of memory. This is fast and cuts down fragmentation, but tends to make the tree lop-sided.

"Better fit" is roughly equivalent to best fit, in that for each node of the tree, the better of the two descendants is chosen, with the search stopping when both descendants are too small. This too preserves the largest block until last, and

also has the effect of keeping the tree more balanced.

The number of blocks to be considered on allocation is on average $\log_2 n$, where n is the number of nodes in the tree. The searching of the tree is fast, but whenever a block is allocated or freed, the tree must be re-built in order to maintain the sorting criteria. Coalescing adjacent blocks is easy though, since the blocks are sorted in address order. Stephenson claims a space performance comparable to first fit (with no store being wasted), and a time performance which lies somewhere between first fit and buddy, depending on the balancing of the tree.

7.8.4 Which storage allocation algorithm?

Of the four different algorithms presented here, the choice of which one to use depends heavily on the nature of the application. First fit performs adequately on small systems with short store chains, and is simple to program. Dual list first fit removes many of the inadequacies of first fit, and is a reasonable choice if a simple algorithm is required, but the number of blocks in the store chain is large. The buddy method performs best of the four time-wise, and is hence a good method for systems which should have a fast real time response. It is, however, wasteful of space, with an average of 25% of allocated store being unused. The final method, using a cartesian tree, is perhaps the most elegant of the four, but since the tree must be re-built each time a block is allocated or freed, it is more complex to program than the others.

If a general purpose system is required which wastes no store and performs better than first fit, then the dual list method is a good compromise. It is simple to program, and although slower than the buddy method in execution time, it does not sacrifice store utilisation in order to obtain improved efficiency.

7.9 User Interface

The nature of the operating system interface to its user is probably the most difficult to define, because not only must the kernel present a clean, simple and consistent interface, but so must all the utilities and system processes which use it. This consistency is not so easy to arrange as with the kernel, since the amount of software work required is much larger, and the effort will be spread over several, if not hundreds of individuals. Unfortunately, when this happens, the style of the individual programmer tends to show in his product, with the result that different programs have subtly different user interfaces. This is particularly true if the operating system is in use at many different sites or allows ease of portability from other machines, since then not only is there a difference

in personal programming style to contend with, but possibly a totally different "house style" as well.

Two systems which have suffered badly from this syndrome are UNIX and TRIPOS. Their histories are very similar, since they were both originally designed and implemented by small teams of people in a research environment, which allowed a large amount of consistency in the kernels and low level software. They were then "enhanced" by many different people from many different sites, each with their own particular views about style and user interface.

The only way to avoid this particular problem is for one individual—either a person or a corporation—to act as a dictator, and define a uniform interface and set of rules which all other people must follow. This job is made easier on a single language system, since the interface can be defined in a concrete manner in terms of interface procedures which all programmers must use. The Xerox corporation have been very successful using this ploy, and the benefit to the user of having a totally consistent set of interfaces is enormous. The process of learning the system is made much easier, since there are no niggling special cases or pieces of "folklore" to be understood.

There is one major part of the operating system interface, which is often overlooked when designing single user operating systems, usually on the grounds of complexity. This is the ability of the operating system to "clean up" after a user program finishes, and being able to kill a program which is running asynchronously.

The first, and most important decision which an operating system designer must take is whether the clients of his system are to be *users* or *programs*, as the requirements are totally different. When designing a system to support multi-tasking facilities, for example, for a network fileserver, the most important factors are efficiency and the real time response to requests. Some sort of priority scheduling algorithm is needed, and since the processes running in the server tend to be static, there is no need to worry about cleaning up, since this would be a wasted facility and could cause inefficiency.

If, on the other hand, the operating system is being designed specifically to support a user environment, then different factors apply. Firstly, real time response is only necessary for the handling of devices such as discs, and user programs should have the rest of the available CPU time divided between them in a "time slice" fashion. Secondly, it is likely that such a system would be used to develop programs, which in their course of development will no doubt fail. The operating system must therefore be capable of debugging those programs and

also tidying up afterwards, and similarly it must be capable of killing programs which are in some sort of uncontrollable state.

It can be seen that the two sets of requirements are almost exactly opposite, and so the decision as to which facilities to provide when designing a general purpose operating system is extremely difficult. For this reason, the best way of tackling the problem is by building the operating system in two distinct layers. The kernel layer has already been discussed, and deals with such factors as simple memory management, process scheduling, inter-process communication and interfaces to the physical hardware (such as interrupt handlers). This type of kernel is small, efficient and simple, and is all that is required by programs which just need a multi-tasking environment in which to work. This would be sufficient for such things as the network fileserver mentioned earlier, and would also cope with the running of debugged applications programs, for instance, a database or information retrieval system. It can be seen though, that this kind of environment is not sufficient to support a user programming environment, and so an extra level built on top of the kernel is required.

It is worth mentioning at this point the experience with the TRIPOS operating system which attempts to get away without having this intermediate layer of user-oriented software. One of the basic design philosophies of TRIPOS was simplicity, and that it was the user's (not the operating system's) job to clean up at the end of a program run, deallocating everything which had been allocated. This meant that it was the responsibility of the individual programmer to remember to free every single memory block, close every file and free every lock, because otherwise those resources would remain allocated, and hence inaccessible.

This strategy is perfectly acceptable for network servers, but users (particularly beginners), find it unnatural and difficult to use. Certainly, it is only a matter of discipline, and it is perfectly possible to write programs in such a way that all resources are released when a program finishes. This discipline takes time to learn though, and even experienced programmers occasionally forget to deallocate something, with frustrating and sometimes unfortunate results. Because of the discipline which is imposed on TRIPOS programs, it is usually the case that they can be exported to other sites with little difficulty, since there is no actual *harm* in freeing resources explicitly, even if the underlying operating system does not require it. Unfortunately, portability is a one-way operation, since programs written at other sites which *do* rely on their systems to tidy up after them, require heavy modification if they are ever to run under TRIPOS.

Now that memory is more plentiful and efficiency much less important, there is no reason for not providing this intermediate user control layer, since the benefits to the individual using the system are so great. The suggestion that the user control software is added as an extra level on top of the kernel rather than designing a single integrated system has two main advantages.

Firstly, the "small is beautiful" philosophy, when applied to operating system design, pays dividends, since each small unit is capable of being defined absolutely, with a published interface. This is particularly useful if the language being used to implement the system allows for the checking of these interfaces, since correctness and integrity are then much easier goals to attain. It is also important that, unless the whole job is to be undertaken by one person, it must be possible to split the implementation effort between many individuals so that each of the operating system modules can be developed in parallel. It also aids comprehension of the entire system if it can be broken down into small, manageable units.

Secondly, unless a two-layered approach is followed, it is difficult to design a system which is usable both by programs and by people, given that their requirements are so different. Splitting the functionality into two parts not only extends the applicability of the product, but since the user level software has a well defined interface to the kernel, there is no reason why it should not be portable to other, similar kernels which provide the same sort of primitives.

Built on top of the user control layer is the user interface, which the person using the system actually sees. This includes the file and compiler utilities, and all the other applications programs. As a rough guide to the demarcation of functionality between the different layers, the following table suggests the facilities which should be put in each.

The *kernel* layer should contain:

- Basic memory management
- Process creation and scheduling
- Inter-process communication
- Bootstrapping
- Device and interrupt handling

The *user control* layer should contain:

- Allocation/deallocation of memory
- Opening/closing of I/O streams
- Obtaining/freeing interlocks

- Console I/O and attentions generated from the keyboard
- Asynchronous program deletion

The *user* layer should contain everything else, but some of the items which fall into this category are:

- Compilers and assemblers
- File and other utilities
- Mail
- Applications

The command language interpreter and command programming language are normally considered by most people to be integral parts of the operating system. This is true in most traditional implementations, but it is actually better to provide a simple "execute command" primitive in the user control layer, and then run the command interpreter as a normal user program. If this is done, then it is much easier to modify the command environment, either to add new facilities or to tailor it for a particular requirement. It also means that it is possible to produce totally new command interpreters, allowing for experimentation and, perhaps, emulation of other systems.

7.10 Portability issues

One of the arguments for encoding an operating system in a high level language is to make the code more portable between different processors. This is only one of the levels of portability which must be considered though, since in some sense, portability of user programs is much more important than that of the underlying operating system, as is portability of the operating system to another machine with the same processor but a different peripheral architecture.

7.10.1 Overall portability

The type of operating system portability which has been investigated in the past, tended to concentrate on overall portability, in other words, the portability of the entire operating system and everything running under it between machines with different processors. The standard way of achieving this goal was to design an operating system for its *ideal machine*, that is, the hardware (usually mythical) which would run that operating system best. This is the approach used for THOTH, which is designed for an abstract "THOTH machine" rather than for a specific piece of hardware. TRIPOS was also designed with overall portability in mind, and through this approach, has been ported successfully onto

half a dozen or so different machines.

In order to achieve overall portability, it is important that the ideal machine chosen for the implementation should not make assumptions about the hardware which would restrict the set of real machines which could be used. This normally means choosing the "lowest common denominator" of the facilities provided by different pieces of hardware, and only using a particular facility if it is available on a large proportion of the possible machines.

If overall portability is a design goal, then the safest definition of an ideal machine is as follows. Assume a large, contiguous address space, which is not segmented. It is important to treat the memory either as a vector of words or a vector of bytes, since mixing the two tends to build in dependencies of word size and byte ordering, which although they are different on different machines, need not affect the operating system design. It is not safe to assume any sort of memory management or memory protection facility, since the number of machines which provide these is relatively small, and when the facilities are provided, they are different on each type of machine. If protection is required, then for portability reasons, it is much better to use a type-safe programming language than to rely on hardware assistance.

To sum up, a simple machine with no fancy features should be assumed, with the result that the portability of the operating system is determined by the portability of the implementation language. This technique leads to under-utilisation of certain hardware features, but in most cases, this is not of great importance compared with the convenience of portability.

7.10.2 User level portability

Since the amount of operating system software on single user machines tends to be small compared to the amount of applications software, there is an argument for the portability to be at this level, rather than to aim for the overall portability of the entire system. From the user point of view there is no difference, since he sees the same interface no matter which machine he uses, and hence he can move his programs easily from site to site.

The advantage of such an approach is that the operating system on each type of hardware can attempt to make more use of the facilities available. For instance, if memory management is available, there is no reason why different applications should not be split into different address spaces, or given virtual rather than real memory to work with. Similarly, if the hardware allows areas of memory to be protected from accidental over-writing, then this can be used to enhance the integrity of the system. There are many examples of facilities which

would improve the utilisation of individual pieces of hardware, and these facilities would be used to their best advantage with this approach.

7.10.3 Same processor portability

This is a type of portability which is rather neglected academically, but is exploited heavily in commercial environments. Same processor portability implies that the operating system is portable onto other machines which have the same instruction set, but which may have a different overall architecture, particularly with respect to peripherals. The portability of such a system comes, not from the language in which it is written, but from the abstractions it uses for items which are architecture dependent.

An example of this type of portability is CP/M [DigitalResearch82], which is implemented entirely in 8080 assembly code. Part of CP/M's success came from the fact that the Z80 processor, although made by a different manufacturer, is compatible with the 8080. Even though the instruction sets of the different processors were compatible, the peripherals connected to them were very different. CP/M achieves its portability by assuming a set of simple facts about the machine on which it runs—akin to the "ideal machine" approach discussed earlier, but much more specific. The amount of memory is unimportant, so long as it is contiguous and starts at location zero. The nature of processor interrupts is irrelevant, since CP/M runs permanently with interrupts disabled, polling all its peripherals. The rest of the portability arises from the separation of the logical parts of peripheral access from the actual details of bytes, I/O commands and peripheral addresses. Each new implementation of CP/M requires a new section to be added to the BIOS (Basic Input Output System) module, which deals with the actions of reading and writing disc blocks, the handling of terminals and printers, and so on. This section is typically only a couple of hundred lines of 8080 assembly code, meaning that very little effort is required to port CP/M onto a new machine—a fact which has contributed to its popularity and adoption as an industry standard.

If large numbers of the same processor become available, there there is a good argument (particularly financially) for designing an operating system which makes use of all the facilities provided by that processor, and then restricting portability to other machines which share that processor. This is particularly true if the designer of the software is also the manufacturer of the hardware, since it is possible to acquire captive users. Examples of this type of system are IBM's OS/360 and DEC's VMS, both of which have dedicated clients who find it financially impossible to patronise anyone other than the original manufacturer.

7.11 Case study: the GMK kernel

GMK is an experimental operating system kernel for IBM 370 processors, and was designed and implemented at the IBM Zürich Research Laboratory [1]. Since the time of the initial work, significant improvements have been made, and the current version is the result of a large amount of development work done by many different people. It should be stressed that the kernel described here is the original version of 1983, and hence contains none of the modifications which have been made since that date.

In the production of GMK, the principal design aim was to provide a light-weight, multi-tasking environment for the running of real time network protocol handling software. It was never intended to support a user programming environment, and so falls into one of the two categories of operating system kernel discussed earlier. Language independence was important, since at the time the kernel was written, the implementation language or languages for the rest of the system had not been decided upon. It was also a conscious design aim to provide the maximum possible power and generality with the simplest possible set of primitives, and hence many of the mechanisms used are "half way houses", or a compromise between generality and complexity.

The design of the kernel was heavily influenced by the author's own experience with the TRIPOS operating system, but GMK was by no means just another TRIPOS implementation. With hindsight, it was possible to examine the original design of TRIPOS critically, and so to decide how many of the ideas involved had stood the test of time. Some of the features of GMK have been adopted directly from TRIPOS, for example the task structure, scheduling, communication using messages and free store management. On the other hand, features such as programming language independence, flag signalling and the whole area of device handling, are new, and totally unrelated to TRIPOS.

7.11.1 GMK tasks

The unit of program encapsulation in GMK is the *task*, and tasks communicate with each other by means of *messages* or single bit *flags*. Each task is given a unique *task name* and *task identifier*, which are used to identify that task. The task name is up to 8 characters in length, and through the kernel primitive *mapname*, it is possible for the binding between task function and task identifier to be dynamic. The task identifier is a positive integer, and is used in

---

[1] The initial design and implementation was done while the author worked at the Laboratory during the Autumn of 1983.

the kernel primitives to identify a task unambiguously. Thus, the mapping from task name to task identifier must be done explicitly by the user before any task communication can occur.

Each GMK task has a unique priority which gives that task's relative importance compared to all others in the system. Since no CPU bound software would run under GMK, there was no need to provide any form of time slicing. Processes of equal importance are implemented as separate coroutines within the same task, enabling a "first come, first served" strategy to be adhered to.

7.11.2 GMK messages

A GMK message is a block of memory, at least two words long, of which the first word is used to chain messages together, and the second word is used to hold the task identifier of the source or destination task (depending on the state of the message). GMK messages are of arbitrary length, and their contents are uninterpreted by the kernel. Messages are always sent directly to tasks rather than to mailboxes, and the allocation of store for a message is the responsibility of the task sending that message.

7.11.3 GMK flags

Each task has a set of single bit flags occupying a word of memory, each of which can be set and tested independently. The difference between the TRIPOS and GMK concept of flags is that, under GMK, it is possible for a task to suspend itself, waiting for one or more of its flags to be set by another task, or by an asynchronous handler. In this way, flags can be treated as zero-length messages, and since setting a flag avoids the message queueing mechanism, it is possible to use flags to signal extraordinary events.

The biggest advantage of flags is that no memory is required to set a flag, and so they can be used in situations where memory allocation would be a problem. One of these circumstances where this is the case is in the signalling of an interrupt condition from an interrupt handler (called asynchronously by the kernel) to its parent task.

7.11.4 Kernel interfaces

Programs running under GMK communicate with the kernel by means of the 370 SVC (supervisor call) instruction. This provides a complete context switch to one of 256 possible SVC handlers, of which only 10% or so are defined by the kernel itself. The others remain available, and the *setsvc* primitive is provided for adding new SVC handlers dynamically. Given the nature of the applications which use GMK, memory protection is not an issue, and hence all programs run

in privileged state. Even so, it is still convenient to use the SVC instruction, since it enables the entry to the kernel to be clean and well defined, and provides the mechanism to enable kernel routines to run with interrupts disabled.

Interrupt handlers can be added by using the *setext* and *setint* primitives, which set up handlers for 370 external and I/O interrupts respectively. When the relevant interrupt condition occurs, the kernel calls the handler which has been set up for that interruption code. Interrupt handlers, unlike SVC handlers, run asynchronously, and so they must use the normal kernel primitives in order to re-synchronise with their parent tasks. When an interrupt handler is created, a value is given to the kernel which is then passed as an argument to the handler each time it is activated. Because of the 370 architecture, this argument is likely to be a pointer to a piece of workspace which is being shared between the parent task and the handler. In this way, the interrupt handler code can be pure, and hence shared by similar devices.

The calling conventions used by the kernel are general, and are exactly the same in all circumstances. This gives a high degree of consistency, and makes it possible to write tasks, interrupt handlers or coroutines in any of the available languages. The kernel also takes responsibility for saving the processor registers and status word whenever an interrupt occurs, and so handlers always appear as simple subroutines of the kernel.

7.11.5 Free store allocation

Free store is allocated in a "first fit" manner, with a single chain of free and allocated blocks. With hindsight, the first fit algorithm was the wrong one to choose, since (as explained earlier in this chapter) its performance becomes linearly worse as the length of the store chain increases. There was also the added problem that, although the kernel was designed to run native on a raw 370 processor, more often than not, it actually ran on a virtual machine under VM/SP [IBM83]. In this environment, the memory given to the kernel is virtual, and so scanning along a single store chain has the effect of requiring a whole sequence of memory pages to be brought from backing store, which are used once and then discarded.

The primitives available are implementation independent though, making it possible for the underlying algorithms to change, without affecting the software which uses them. *getvec* and *freevec* are provided to allocate and deallocate blocks of store, with *shrinkvec* allowing an allocated vector to have some of its store released back to the system.

7.11.6 Task creation and deletion

The primitives to create and delete tasks are *createtask* and *deletetask*, but the GMK philosophy is rather different to that of TRIPOS. Under TRIPOS, it is defined that a task, once created, remains in "dead" state until a message is sent to it. This is not the case under GMK, and when a task is created, the parameters to the kernel primitive are simply the new task's entry point and a parameter to be passed to it when it is activated. The task is then called as a kernel subroutine, with the argument passed to it on startup being the one given by the creator. When created, a task is given a unique *name* (assigned by the creator) and *identifier* (assigned by the system), with the *mapname* and *mapid* primitives providing a mapping between the two task representations.

7.11.7 Task re-scheduling

The primitives provided by TRIPOS for task re-scheduling have proved entirely adequate, and so have been adopted by GMK. A task's priority can be changed by using the *changepri* primitive, and the *hold* and *release* primitives can stop and start a task being activated. There is also the extra primitive *die* which can be called to kill the current task prematurely, and which is called whenever a task runs to completion.

7.11.8 Message manipulation

As with TRIPOS, the primitives provided are *qpkt* to send a message to another task, *dqpkt* to remove a message from another task's message queue, and *taskwait* to suspend a task, waiting for a message to arrive. Messages can also be awaited using the more general *wait* primitive, which incorporates waiting for flags as well.

7.11.9 Flag manipulation

As already described, flags take on much more significance under GMK than they did under TRIPOS. The primitives *setflags* and *testflags* are provided to manipulate flags synchronously, and in addition, *flagwait* is provided enabling a task to suspend itself waiting for a flag event. The general *wait* primitive allows suspension, waiting for either flags or messages.

7.11.10 Critical code sections

To enable atomic updating of data structures to take place outside the GMK kernel, the primitives *crit* and *uncrit* are privided to enter and leave critical sections of code. This is implemented simplistically by disabling and enabling interrupts, but even so, it is a very useful facility. An alternative to this method would have been to provide semaphores and monitors, although it is dubious

whether the advantages would warrant the extra kernel complexity.

7.11.11 Device and interrupt handling

It is in the area of interrupts and device handling where GMK is totally different from TRIPOS. Devices under TRIPOS, although simple in concept, are complicated and difficult to encode, and tend to be larger than the design team originally hoped. The main problem is that TRIPOS tries, to its credit, to make devices look to the user exactly like tasks, in that communication with devices is also through the use of messages. GMK takes the approach that only two very small parts of a device driver actually need to be written in assembly code, and that the rest can easily be written in a high level language. The result is that, rather than having the distinction between tasks and devices which exists under TRIPOS, GMK only has tasks, and provides primitives to enable a task to handle devices directly.

The two pieces of assembly code mentioned above are the device *start* routine, which handles the initiation of an I/O operation, and the device *interrupt* routine, which handles the completion of an I/O operation. The action of starting an I/O operation is synchronous, and can easily be handled by an assembly code subroutine of the handler task. The kernel primitive *startio* is provided, to enable the 370 channel address word to be updated and the I/O operation to be started in an atomic manner. External and I/O interrupts are trapped by the kernel, and then passed on to interrupt handlers, which can be set up using the *setext* and *setint* primitives. Because of their nature, interrupt handlers are called asynchronously, and hence must use one of the kernel message or flag primitives to communicate with their parent tasks.

7.11.12 Evaluation of the GMK kernel

The GMK kernel was written with a particular application in mind, but one of the main design aims was to produce something clean and simple, which was general enough for other applications as well. The two main areas where GMK definitely improves on TRIPOS are those of language independence and device handling.

All the GMK interfaces and calling conventions were designed carefully, with the result that there is a high degree of consistency. This usually means that, unlike TRIPOS, any task or coroutine can be written in any language, so long as care is taken to choose a sensible format for any structure which is shared between programs in different languages. Even though the calling conventions used for interrupt handlers are the same as the others, it is possible that certain languages may be difficult to use for this purpose, simply because of the

complexity of the run time environment which they require.

The area of device handling has been completely re-designed, and the result is much simpler and more convenient to use. It should be pointed out here that the 370 is an extremely good machine for I/O, with every device being handled in exactly the same way. This meant that devices could simply be represented by their device codes, and these codes could be used to identify device handlers whenever an interrupt occurred. Again, interrupts are clean and simple on a 370, with the results always being placed in the same memory location. Given that I/O is so simple on a 370, it was decided to make maximum use of this fact, which means that the mechanism is not easily portable to processors with a slightly less ideal architecture.

The initial version of GMK was by no means perfect—storage allocation using "first fit" and the exclusion of semaphores are just two of the decisions which would be taken differently now. It was a significant improvement on TRIPOS though, and many of the guidelines proposed in this chapter came directly out of experience with GMK and its implementation.

7.12 Summary and conclusions

In this chapter, many different aspects of operating system design have been discussed, and it is now possible to summarise the points raised, and provide guidelines for designers of future systems. The most important point to note is that there are two totally different applications for any multi-tasking operating system. The first, which is the topic of earlier chapters, is the single user workstation, where the operating system provides a convenient working environment for one user to perform program development, word processing, database interrogation, and so on. The second application is when the operating system in question is used, not to serve users, but to serve programs or other computers. This is particularly true in the area of computer networks, for example fileservers and inter-network bridges, which are primarily involved with giving a real time service to other network nodes.

These two types of application demand very different solutions, but through clever design, it is possible to provide a general purpose system which goes most of the way to solving the requirements of both applications. The general guidelines which should be adhered to when designing a new system are as follows.

The operating system should be split into two parts—the *kernel* layer, which is involved with basic hardware management, and the *user control* layer, which is

involved with the user's working environment. The interface between the two layers should be clean and well defined, with the kernel capable of being run in isolation if no user is attached, and efficiency is more important than convenience.

The kernel of the operating system should, where possible, be written in a high level language. This is not primarily for portability, although portability is a useful side effect, but to enable algorithms to be encoded in such a way that they are easy to read, verify and debug. On the whole, raw efficiency of compiled code as compared to hand crafted assembly code is not important, and the emphasis should be placed on efficient algorithms rather than efficient implementation. The kernel itself should be written in a single language, since this simplifies the definition of module interfaces, allows data structures to be represented in a consistent and natural way, and may allow compiler checking of interface definitions. The choice of language is not particularly important, except that untyped languages tend to allow easier access to the underlying hardware than typed languages, which usually require "loopholes" for this.

The interface between the kernel and its clients should be via a processor "trap" mechanism, since this provides a well defined barrier behind which all the kernel data structures can be protected. Using a trap mechanism also allows language independence outside the kernel, and gives an easy way of distinguishing kernel issues from language issues within the operating system itself.

The choice of scheduling algorithm which the kernel should use is determined by the nature of the requirements. A priority system should be used for applications where real time response is important, with time slicing being used where multiple processes of equal importance are present, or where a multi-user capability is required. A system which has process priorities and which time slices processes of equal priority is a convenient compromise, should both facilities be required.

Where possible, the interfaces and calling sequences used by the operating system should be consistent throughout, and should not depend on context. In this way, processes, coroutines, monitors, interrupt handlers and so on, can all be written in the same way, with language independence being guaranteed.

The choices between processes and coroutines, and messages and monitors are very much up to the designer of the system, and depend on his own personal preferences. The main issue here is one of expandability, in that message systems fit more cleanly into network architectures, with process messages and network packets being exactly analogous. This is still a personal decision though, since remote procedure calls are possible to implement, and have been shown to work

well in practice. Coroutines are useful both as sub-processes within the same process and, so long as the coroutine system is language independent, as interfaces between programs written in different languages. For a truly general purpose system, there is no reason why both messages and monitors should not be provided. Monitors and the use of semaphores would, in some circumstances, speed up access to kernel data structures (such as a free store chain) since through their use, it is possible to ensure atomic update without having to disable processor interrupts.

The storage allocation strategy used by the kernel should depend on the nature of the application, and on the number of store blocks being handled. For systems with large amounts of memory where a fast real time response is required, the buddy method is the one to choose, otherwise the dual list first fit method is a reasonable compromise between allocation speed and store wastage.

All the parts of the operating system responsible for cleaning up after programs, handling asynchronous program killing and so on, should be implemented in a user control layer, thus allowing the convenience it provides to be an "added extra" if the corresponding reduced efficiency is not a problem.

Finally, if a portable system is required, then it should be decided at which level the portability should be. Overall portability enables the entire operating system to be moved, but precludes the use of special hardware features such as memory management. User level portability ensures the portability of user programs, and since the operating system itself need not be portable, this allows optional use of such things as virtual memory and storage protection. A third level of portability is possible if there is a wide range of machines based on a single type of processor, in that a system can be designed to be portable, but only between machines which use that processor.

8. Conclusion

The work described in this thesis is an attempt to decrease the widening gap between the hardware available for large personal workstations, and the operating system software which makes use of them. Through the development of fast, single chip 32 bit micro-processors (typified by the Motorola MC68000), and the massive reduction in the cost of 64K and 256K dynamic memory chips, it is now possible for an individual to have on his desk a machine which, only a few years ago, would have supported many tens of users.

Until the introduction of this type of machine, it was possible to classify computers into two distinct categories. Firstly, there were the "small" machines: cheap mini- and micro-computers with small amounts of memory, 8 or 16 bit processors, and low performance peripherals. Secondly, there were the "large" machines: expensive mainframes with large amounts of memory, fast 32 (or more) bit processors, with peripherals controlled by I/O channels, each of which was a reasonably powerful computer it its own right. The new generation of machines did not fit conveniently into either category, having features which were characteristic of both. Because of this confusion, two different approaches were taken to operating system design.

The first approach was that the new machines were large micro-computers, and as such, should be treated as larger versions of the "small" category machines. This approach was taken principally by commercial establishments, and out of it came operating systems like CP/M-86, MS/DOS and QDOS, written for the 8086, 8088 and 68008 respectively. Because of the "small computer" attitude, there was a tendency for operating systems to be single threaded, with the only multi-tasking (if any) being in the form of spooling to a printer.

The second approach was that the new machines were small mainframes, and hence should be treated as smaller versions of the "large" category machines. This is perhaps the better of the two approaches, since it aimed to utilise the hardware to a greater extent. Also, much effort had been expended in the design of the mainframe operating systems, even though many of the facilities were inapplicable to the single user case. Unfortunately, mainframe operating systems tend to be large and complicated, requiring fast discs to implement such things as paging and swapping. It is this fact which sometimes makes using a mainframe operating system on the new machines unsatisfactory, since not only are the discs themselves slow, but the main processor must handle all its own I/O operations,

and often access to peripherals is through polling rather than by DMA.

Both approaches have produced usable working environments, but neither actually tackled the real problem—how to get the maximum possible out of the new type of machine. The approach taken here was rather different to the two described above, in that it took as its starting point a small multi-tasking operating system which was already designed for single user operation, and investigated the changes necessary to enable this type of system to be enhanced in order to make better use of the available hardware facilities. Out of the work came a series of ideas, techniques and guidelines which, if applied elsewhere, should improve overall performance and increase hardware utilisation.

The operating system taken as the starting point was TRIPOS, which had the double advantage of being written in a high level language and also available locally. TRIPOS was written in a portable manner, and before the work described here, it had been implemented on half a dozen or so different 16 bit mini-computers. The event which sparked this work was the porting of TRIPOS onto a new range of 68000 based machines, each with at least half a megabyte of memory—something for which TRIPOS was never originally designed.

The overall environment where the work was carried out was also important, in that the only people using TRIPOS and relying on it to fulfil their computing requirements were themselves involved in research. These people, who were themselves used to trying out new ideas, could fairly easily be persuaded to take part in experiments, and at the same time, would cooperate by making helpful suggestions and providing intelligent feedback.

The fact that the work was carried out in a distributed computing environment had little effect on the results, except that it was easier to release new versions of software for testing purposes. The use of remote peripherals (even fast expensive ones, such as the fileserver discs) had the effect of making most computations I/O bound, and simulated closely what would have happened if the work had been carried out on a stand-alone machine with locally attached, but relatively slow peripherals.

There was one aim to the whole project, and that was to find out how the new machines could be used to their full advantage. The work involved can be split conveniently into four separate areas, and it is possible to draw conclusions from each of these individually.

The first part of the work (described in chapter 4 and appendix 1), involved the design and implementation of the TRIPOS Shell—an enhanced replacement

for the original command language interpreter. Although this work was a pre-requisite for that which followed, it is likely that the days of the command language interpreter are numbered, and in the future, facilities such as command histories will be obsolete. This is because the techniques involved aim to optimise the use of the conventional computer terminal device—a dumb VDU or teletype. With such a device, the only way of sending information to the host computer is through the use of a keyboard, and because so many keystrokes are required to send any particular command, the chance of error is always high.

The reason for the use of this sort of a terminal is that, in the past, many users shared a computer, and since the machine was remote and had to handle many terminal lines, only a low bandwidth connection was possible. Also, through the use of serial lines, standardisation was achieved, making it possible to connect any terminal to any computer, and also to connect computers to each other, enabling file transfer and so on. Serial lines have the advantage that they are simple, require very few wires, and the data transfer rate is slow enough that long distance communication using telephone lines is possible.

As the cost of computer components, particularly memory, has come down, there has been the tendency to give each user his own personal workstation, rather than sharing a single mainframe. Since the physical size of the machine is usually fairly small, it is possible, by placing a machine in the office of its owner, to shorten the distance between terminal and computer in such a way as to allow much higher bandwidth communication. Combine this with the cheapness of memory, and it becomes possible to give each user his own high resolution bit-mapped terminal, on which can be displayed many VDU-like screens as separate windows. Accompanying such a terminal is a pointing device, usually a mouse, which enables characters, words or whole areas of the screen to be selected with ease.

In such an environment, where the act of pointing replaces the less accurate typing at a keyboard, the need for command language interpreters is reduced, since it is possible to point at commands, rather than type their names. Some operating systems have already done away with their command language interpreters, with obvious examples being those for the Xerox personal workstations [Redell80, XEROX82, XEROX84] and the Apple Macintosh [Apple84]. No doubt, as memory becomes even cheaper, so bit-mapped terminals will become more common, and future work on user interfaces should really be concentrated in this area.

As a piece of future work, a "virtual screen" protocol should be designed, so as to standardise the way in which bit-mapped terminals and pointing devices are

used. Only when this is done will it be possible to write software which handles these devices in a consistent manner, and until it exists, a series of *ad hoc* implementations will appear, making life confusing for the user and portability impossible for the operating system designer. This protocol should enable bit-mapped terminals to be used to access computers where only a low bandwidth connection (such as a serial line) is available. In this way, even multi-user operating systems and wide-area computer networks can make use of the new technology.

There are still many occasions, though, when access to a computer must be via a traditional terminal, and hence where a command language interpreter is necessary. The main advantage that typing commands at a keyboard has over pointing at a screen, is that it requires simpler equipment, and a much lower bandwidth connection between the terminal and the host computer. This factor is particularly important for systems which support many users, or are accessed in some remote manner. Even local area networks with high point-to-point bandwidths take several seconds to transfer a bit-map from one machine to another. If a command language interpreter is required, then there are three important facilities which it should provide if it is to be of maximum use.

Firstly, it should enable the user to have several tasks (both foreground and background) executing in parallel, and provide some convenient way of switching between them. It should also provide some way of being able to direct the output of one program as the input to another program, so that the maximum parallelism can be achieved. The syntax should be easy to learn and convenient to use, with a reasonable compromise between being too verbose on the one hand, and too cryptic on the other.

Secondly, it should provide some sort of command history mechanism, so that recently typed command lines can be recalled, edited, and then re-executed. The methods used to identify previous command lines should be concise and easy to use, as should the editing commands. It goes without saying that it must be easier to edit and re-execute a previous command line, than it is to type it in again.

Thirdly, it should have a simple and clean interface to programs which run under it, and it must be possible for these programs to call the command language interpreter recursively, should they wish to execute a command line or start up a new command session. So long as the interface is available, it is possible for the user to switch between command environments at will, and also to customise his command environment, simply by adding an extra layer of software above the command language interpreter to handle the required syntax.

In any computer system, there will always be the situation where a certain set of commands must be executed repeatedly (with or without parameter substitution), and for this application, some sort of command language is required. In the past, such languages have been modelled on, but not actually been, true programming languages, and have tended to have an archaic and clumsy syntax. As a second requirement of any command environment, there should also be a language in which the user can write small throw-away programs. It is often the case that a column of numbers must be added up, or a set of files must be re-formatted, or some other job which is too complicated for one of the standard utilities (such as an editor) to perform. For this sort of application a simple program is required, but traditional compiled programming languages are far from simple to use, and programs written in them are almost impossible to get right first time.

Assuming that there is a clean interface to the command language interpreter, there is no reason why the two requirements cannot be fulfilled by the same language. In fact, there is a distinct advantage in doing so, since firstly, the user need learn only one language rather than two, and secondly, there is often the desire to combine command execution with some form of computation—surprisingly difficult in traditional systems where the command and programming languages are separate.

To show that it was possible to design a language which was useful for both command sequences and simple programming, the REX system (described in chapter 5 and appendix 2) was designed and implemented. The language is strongly data typed, but the interpreter, where possible, coerces from one data type to another in order to avoid error. There is an emphasis on being able to make a program work, and as a result the parser tries very hard to recover from syntax errors while giving useful diagnostics, and the interpreter allows programs to be traced as they are executed, in order to find out where they are going wrong. The variety of applications is immense, but the most popular use by far is for the implementation of short command sequences, for example, to compile, link and then run a program. As well as simple command sequences, there are examples of REX programs to re-format files, to sort and re-number the globals in a BCPL header file, and to re-initialise a disc directory structure.

The third area of work, described in chapter 6, involved the investigation of how simple techniques can be used to improve the convenience and performance of a user's working environment. Under TRIPOS, adding the extra facilities proved relatively straightforward, owing to the way in which pseudo

devices—handlers with no associated peripheral—could be created. In other environments, adding such facilities may not be quite so simple, but the principles involved are the same, no matter what the nature of the underlying system.

One of the techniques, command pre-loading, demonstrates how it is possible to improve the access to slow discs, simply by making use of memory to store frequently used programs. Slow discs have always been a problem in computer systems, and because of this, the technique of keeping in memory a cache store of recently used disc blocks was developed. Although keeping a cache of disc blocks in memory still has much to recommend it, it does not take account of files as a whole, and so the effect of keeping a file in memory may be lost if one of its constituent blocks has been lost from the cache. The type of file for which this is important is a program load module, because the whole file is needed at once, and even one disc transfer can slow the loading operation down. There is also the important fact that programs, once loaded, are stored in a different format in memory than they were on disc.

The experience with TRIPOS has shown that pre-loaded programs should be handled in a different way to normal cached disc blocks. Both systems are required, because it is still likely that, for instance a file will be read after it has been written, or a directory block used after it has been examined. With both mechanisms, better utilisation can be made of memory, and the overall improvement in response which the user sees can be dramatic. It is likely that the absolute amount of time gained from such a system is very small if measured over a long period, but that productivity is improved because using large programs (such as an editor or a compiler) no longer involves the long delay while they are brought into memory from disc, and so the user's attitude to program development changes accordingly.

The fourth area of work, described in chapter 7, investigates the issues associated with operating system kernels, in an attempt to draw up a set of guidelines for future designers.

With the new type of machines, large amounts of memory are available, and since access to I/O devices is slow, the size and efficiency of the operating system kernel are no longer relevant issues. What is important though, and will become more important as the cost of software development increases, is that the kernel should be written in a single high level language, and provide as general an interface as possible to its clients.

There are two main reasons for writing an operating system kernel in a high level language. Firstly, by their very nature, high level languages are more portable than assembly code, and although using a high level language does not ensure portability, it certainly aids it. Secondly, it is easier and quicker to express algorithms in a language which has a decent control structure, and because programs written in a high level language must be checked by a compiler before being executed, many potential problems can be detected at compile time. If the language chosen provides the facility, it is also possible to use it to check the interfaces between different operating system modules—something with helps ensure correctness, and allows for the easy distribution of implementation jobs between different individuals.

Even though the kernel itself should be written in a single high level language, it should provide a general and language independent interface to its clients. If possible, hardware traps should be used to enter the kernel domain, as this allows non-kernel processes to execute at a lower level of privilege. It also defines an exact point where the transition from user code to kernel code occurs, and so defines a protection boundary behind which the kernel abstractions can be hidden. Whenever the kernel passes control to user code—on entry to a process, when an interrupt occurs and so on—it must do so in a simple and consistent manner, preferably through means of a subroutine call. So long as all different calling sequences obey this same convention, it is possible to write software which is both language independent and context free.

If possible, an operating system should be designed in a two-layered manner, so that the same kernel can be used for both real time programming work, and to provide a user working environment. Where possible, all the user specific code should be kept out of the kernel, so that if an application requires efficiency rather than convenience, no CPU time is lost performing unnecessary work. Designing the operating system in this way gives greater generality, and removes the necessity of having separate implementations for the two different kinds of application.

There are three different types of portability which the operating system designer must bear in mind. Firstly, there is overall portability, where the whole operating system, and everything which runs under it is designed to be portable. Unfortunately, for an operating system to be portable to this extent, it cannot afford to rely on facilities which are only available on a few machines. Secondly, there is user level portability, where only the portability of applications software is guaranteed. This approach has much to recommend it, since the user sees no difference between this and overall portability, but it does allow more specialised

use of the hardware available, for example virtual memory or a bit-mapped terminal. Thirdly, there is same processor portability, where the operating system is portable between different computers which the same type of processor. As an example of this last case, IBM/370 machines are so common that there was no point in making the GMK kernel portable onto other types of processor; to do so would only have added complications.

To sum up, the new generation of computers have requirements which are very different from either of their two types of predecessor, and a separate set of techniques should be used in order to utilise them to their full capacity. They have many weaknesses which require further work in order to improve performance. The two obvious examples are the speed of and method of access to peripherals, particularly discs, and the cleaning up of the micro-processor instruction sets to make them more orthogonal, and easier for compilers to generate code. Given that the new machines have so much memory, it would be useful to have some standard form of memory management unit, built in VLSI, which would enable operating systems designers to implement protection through address space separation without restricting the portability of his product.

This thesis has attempted to show the subtle change of emphasis which is required when designing system software for the new type of machines, from the techniques to be used in the operating system kernel to the facilities which should be provided in the user interface. The problem of slow access to discs combined with the availability of large amounts of memory mean that it is possible to have a great deal of resident software, providing functions which are specifically geared to optimising the user environment. The memory size also means that it is possible to treat as resident programs which are frequently used, since they can be pre-loaded in such a way as to appear part of the operating system. The combination of all the ideas results in a simple, unified design, which can be applied to give a powerful working environment which takes full advantage of the new type of large personal workstation.

References

[Apple84] Apple Computer Inc.
Apple Macintosh User's Manual
Apple Computer Inc., Cupertino, California, 1984

[Ash81] W. L. Ash
Mxec: Parallel Processing with an Advanced Macro Facility
Communications of the ACM 24 No. 8 pp. 502-509, 1981

[Atkins83] M. S. Atkins and B. J. Knight
Experience with Coroutines in BCPL
Software Practice and Experience 13 No. 8 pp. 765-769, 1983

[Barron63] D. W. Barron, J. N. Buxton, D. F. Hartley, E. Nixon and C. Strachey
The main features of CPL
The Computer Journal 6 pp. 134-143, 1963

[Birrell84] A. D. Birrell and B. J. Nelson
Implementing Remote Procedure Calls
ACM Transactions on Computer Systems 2 No. 1 pp. 39-59, 1984

[Berkeley81] University of California at Berkeley
UNIX Programmer's Manual (4th Berkeley Distribution)
Computer Science Division, University of California at Berkeley. Seventh Edition, 1981

[Bourne78] S. R. Bourne
The UNIX Shell
Bell System Technical Journal 57 No. 6 pp. 1971-1990, 1978

[Braga76] R. S. C. Braga
Eh Reference Manual
Department of Computer Science, University of Waterloo. Rep. CS-76-45, 1976

[BrinchHansen76] P. Brinch Hansen
The SOLO Operating System: A Concurrent Pascal Program
The SOLO Operating System: Job Interface
The SOLO Operating System: Processes, Monitors and Classes
Software Practice and Experience 6 No. 2 pp. 141-200, 1976

[Brown79] P. J. Brown
Writing Interactive Compilers and Interpreters
John Wiley and Sons, Chichester, pp. 253-256, 1979

[Cheriton79] D. R. Cheriton, M. A. Malcolm, L. S. Melen and G. R. Sager
THOTH: A Portable Real-Time Operating System
Communications of the ACM 22 No. 2 pp. 105-114, 1979

[CMU80] Carnegie-Mellon University
Research in Personal Computing at Carnegie-Mellon University
CMU Internal Planning Document, 1980

[Colijn76] A. W. Colijn
Experiments with the KRONOS Control Language
Software Practice and Experience 6 No. 1 pp. 133-136, 1976

[Colijn81] A. W. Colijn
A Note on the MULTICS Command Language
Software Practice and Experience 11 No. 7 pp. 741-744, 1981

[Cowlishaw84] M. F. Cowlishaw
The Design of the REXX Language
IBM Systems Journal 23 No. 4 pp. 326-335, 1984

[Cromemco82] Cromemco Inc.
CROMIX: Multi-User Multi-Tasking Disk Operating System
Cromemco Inc., California. Part No. 023-4022, 1982

[Dahl66] O.-J. Dahl and K. Nygaard
Simula, an Algol based Simulation Language
Communications of the ACM 9 No. 9 pp. 671-678, 1966

[DEC84] Digital Equipment Corporation
ULTRIX-32 Programmer's Manual
Digital Equipment Corporation, Merrimack, New Hampshire. Order No. AA-BG52A-TE, 1984

[DigitalResearch82] Digital Research
CP/M Operating System Manual
Digital Research, Pacific Grove, California, 1982

[Dijkstra68] E. W. Dijkstra
Cooperating Sequential Processes
In *Programming Languages*, Ed. F. Genuys, Academic Press, New York, 1968

[Dion80] J. Dion
Reliable Storage in a Local Network
Ph.D. Thesis, published as Technical Report 16, University of Cambridge, 1980

[Evans81] R. D. Evans
Language Implementation in a Portable Operating System
Ph.D. Thesis, University of Cambridge, 1981

[Frank79] G. R. Frank and C. J. Theaker
The Design of the MUSS Operating System
MUSS: The User Interface
MUSS: A Portable Operating System
An Assessment of the MUSS Operating System
Software Practice and Experience 9 No. 8 pp. 599-670, 1979

[Garnett83] N. H. Garnett
Intelligent Network Interfaces
Ph.D. Thesis, published as Technical Report 46, University of Cambridge, 1983

[Gibbons80] J. J. Gibbons
The Design of Interfaces for the Cambridge Ring
Ph.D. Thesis, University of Cambridge, 1980

[Gosling82] J. Gosling
UNIX Emacs
Carnegie-Mellon University, 1982. Republished, University of Cambridge, 1984

[Hamilton84] K. G. Hamilton
A Remote Procedure Call System
Ph.D. Thesis, published as Technical Report 70, University of Cambridge, 1984

[Harvey82] B. Harvey
Why LOGO?
BYTE 7 No. 8 pp. 163-193, 1982

[Hoare74] C. A. R. Hoare
Monitors: An Operating System Structuring Concept
Communications of the ACM 17 No. 10 pp. 549-557, 1974

[Hopper78] A. Hopper
Local Area Computer Communication Networks
Ph.D. Thesis, published as Technical Report 7, University of Cambridge, 1978

[IBM72] IBM Corporation
IBM VM/370: Command Language User's Guide
IBM Corporation, Order No. GC20-1804-0, 1972

[IBM83] IBM Corporation
IBM VM/SP Release 3: Introduction
IBM Corporation, Order No. GC19-6200-2, 1983

[Joy80] W. Joy
CSH: A Shell (command interpreter) with C-like Syntax
UNIX Programmer's Manual, 4th Berkeley Distribution, 1980

[Knight82] B. J. Knight
Portable System Software for Personal Computers on a Network
Ph.D. Thesis, published as Technical Report 26, University of Cambridge, 1982

[Kernighan78] B. W. Kernighan and D. M. Ritchie
The C Programming Language
Prentice Hall, Englewood Cliffs, New Jersey, 1978

[Knuth73] D. Knuth
The Art of Computer Programming: Fundamental Algorithms
Addison Wesley, London, pp. 435-451, 1973

[Kurtz78] T. E. Kurtz
BASIC Language Summary
ACM SIGPLAN Notices 13 No. 8 pp. 103-118, 1978

[Lampson79] B. W. Lampson and R. F. Sproull
An Open Operating System for a Single User Machine
Proceedings of the 7th Symposium on Operating System Principles, pp. 98-105, 1979

[Lauer78] H. C. Lauer and R. M. Needham
On the Duality of Operating System Structures
Proceedings of the 2nd International Symposium on Operating Systems, IRIA, 1978

[Lauer81] H. C. Lauer
Observations on the Development of an Operating System
ACM Operating Systems Review 15 No. 5 pp. 30-36, 1981

[Leslie83] I. M. Leslie
Extending the Local Area Network
Ph.D. Thesis, published as Technical Report 43, University of Cambridge, 1983

[Levine80] J. Levine
Why a LISP based command language?
ACM SIGPLAN Notices 15 No. 5 pp. 49-53, 1980

[Liskov81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. Schaffert, R. Scheiffler and A. Snyder
CLU Reference Manual
Springer Verlag, Berlin, 1981

[Lister76] A. M. Lister and K. J. Maynard
An Implementation of Monitors
Software Practice and Experience 6 No. 3 pp. 377-385, 1976

[Mashey76] J. R. Mashey
Using a Command Language as a High Level Programming Language
IEEE Proceedings of the 2nd International Conference on Software Engineering pp. 169-176, 1976

[Microsoft82] Microsoft Corporation
XENIX Programmer's Manual
Microsoft Corporation, Bellevue, Washington State, 1982

[Microsoft83] Microsoft Corporation
MS-DOS Operating System Programmer's Reference Manual
Microsoft Corporation, Bellevue, Washington State, 1983

[Mitchell79] J. G. Mitchell, W. Maybury and R. Sweet
MESA Language Manual
XEROX Palo Alto Research Center Technical Report, 1979

[Moody80] J. K. M. Moody and M. Richards
A Coroutine Mechanism for BCPL
Software Practice and Experience 10 No. 10 pp. 765-771, 1980

[Needham82] R. M. Needham and A. J. Herbert
The Cambridge Distributed Computing System
Addison Wesley, London, 1982

[Ody84] N. J. Ody
Terminal Handling in a Distributed System
Ph.D. Thesis, University of Cambridge, 1984

[Rashid81] R. F. Rashid and G. G. Robertson
Accent: A Communication Oriented Network Operating System Kernel
Proceedings of the 8th Symposium on Operating System Principles
pp. 64-75, 1981

[Redell80] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray and S. C. Purcell
PILOT: An Operating System for a Personal Computer
Communications of the ACM 13 No. 2 pp. 81-92, 1980

[Richards69] M. Richards
BCPL: A Tool for Compiler Writing and Systems Programming
Proceedings of AFIPS Spring Joint Computer Conference 34 pp. 557-566, 1969

[Richards79a] M. Richards, A. R. Aylward, P. Bond, R. D. Evans and B. J. Knight
TRIPOS: A Portable Operating System for Mini-Computers
Software Practice and Experience 9 No. 7 pp. 513-526, 1979

[Richards79b] M. Richards
A Compact Function for Regular Expression Pattern Matching
Software Practice and Experience 9 No. 7 pp. 527-534, 1979

[Richardson80] M. F. Richardson
File Handler Version Three
Tripos Research Group Documentation, University of Cambridge, 1980

[Ritchie74] D. M. Ritchie
The UNIX Time-Sharing System
Communications of the ACM 17 No. 7 pp. 365-375, 1974

[Ritchie78] D. M.Ritchie
A Retrospective
Bell System Technical Journal 57 No. 6 pp. 1947-1969, 1978

[SinclairResearch84] Sinclair Research Ltd.
Sinclair QL User Guide and QDOS Documentation
Sinclair Research Ltd., Cambridge, 1984

[Stephenson73] C. J. Stephenson
On the Structure and Control of Commands
ACM SIGOPS 7 No. 4 pp. 22-26 and 127-136, 1973

[Stephenson83] C. J. Stephenson
New Methods for Dynamic Storage Allocation
Proceedings of the 9th Symposium on Operating System Principles
pp. 30-32, 1983

[Stoy72] J. E. Stoy and C. Strachey
OS6: An Experimental Operating System for a Small Computer
Part 1: General Principles and Structure
Part 2: Input/Output and Filing System
Computer Journal 15, Part 1 pp. 117-124, Part 2 pp. 195-203, 1972

[Stoye83] W. R. Stoye
WORCESTAR: A New Screen Editor for TRIPOS
Systems Research Group Note, University of Cambridge, 1983

[Thacker78] C. P. Thacker et al.
ALTO: A Personal Computer
In *Computer Structures: Readings and Examples*, Sieworek, Bell and Kewell, Eds. McGraw Hill, 1978

[VanWijngaarden76] A. van Wijngaarden, B. Mailloux, J. Peck, C. Koster, M. Sintzoff, C. Linsey, L. Meertens and R. Fisher
Revised Report on the Algorithmic Language ALGOL68
Springer Verlag, Berlin, 1976

[Ward79] S. A. Ward and C. J. Terman
An Approach to Personal Computing
Laboratory of Computer Science, Massachusetts Institute of Technology, 1979

[Ward80] S. A. Ward
TRIX: A Network Oriented Operating System
IEEE COMPCON '80 pp. 344-349, 1980

[Wilkes82] A. J. Wilkes
A Portable BCPL Library
Technical Report 30, University of Cambridge, 1982

[Wilkes75] M. V. Wilkes
Communication using a Digital Ring
Proceedings of the Pacific Area Communication Network Symposium
pp. 47-56, 1975

[Wilkes79] M. V. Wilkes and R. M. Needham
The Cambridge CAP Computer and its Operating System
Elsevier North Holland, New York, 1979

[Wilson81] I. D. Wilson
M68KASM: A Macro Assembler for the MC68000 Micro-Processor
Systems Research Group Documentation, University of Cambridge, 1981

[Wilson82a] I. D. Wilson
DEBUG68: A Symbolic Debugger for the MC68000 Micro-Processor
Systems Research Group Documentation, University of Cambridge, 1982

[Wilson82b] I. D. Wilson
Thoughts on a Local Winchester Disc for Processor Bank 68000s
Systems Research Group Note, University of Cambridge, 1982

[Wilson83] I. D. Wilson
The TRIPOS CLI Rethought
Systems Research Group Note, University of Cambridge, 1983

[Wilson84] I. D. Wilson
A Distributed MAIL System for TRIPOS
Systems Research Group Documentation, University of Cambridge, 1984

[Wilson85] I. D. Wilson
A New Storage Allocation Scheme for TRIPOS
Systems Research Group Note, University of Cambridge, 1985

[Wirth74] N. Wirth and K. Jensen
PASCAL: User Manual and Report
Springer Verlag, Berlin, 1974

[Wirth80] N. Wirth
Programming in MODULA-2
ETH Zürich Technical Report 36, 1980. Republished, Springer Verlag, Berlin, 1982

[Wirth81] N. Wirth
The Personal Computer LILITH
ETH Zürich Technical Report 40, 1981

[XEROX81a] XEROX Corporation
Internet Transport Protocols
XEROX System Integration Standard XSIS 028112, 1981

[XEROX81b] XEROX Corporation
Courier: The Remote Procedure Call Protocol
XEROX System Integration Standard XSIS 038112, 1981

[XEROX82] XEROX Corporation
XEROX 8000 Network Systems: 8010 Reference Guide
XEROX Corporation Documentation, 1982

[XEROX84] XEROX Corporation
XEROX Development Environment: Concepts and Principles
XEROX Corporation Documentation, Version 3.0, 1984

Appendix 1: Shell details

Command history mechanism

The Shell keeps a list of the most recently executed command lines, and provides a simple method for editing and re-executing previous lines. Any input line which begins with the character "!" is treated as one referring to a previous command line, with the line involved being specified by one of the following:

!<number>	Absolute command number
!-<number>	Relative command number
!!	Last command
!<string>	Last command beginning with <string>
!?<string>	Last command containing <string>

!<number> describes a command line absolutely and unambiguously by quoting its sequence number. For example:

```
!23
```

means "execute the command line whose sequence number is 23". !-<number> describes a command line relative to the current one by quoting the difference in the command sequence numbers. For example:

```
!-1
```

means "execute the last command line but 1", in other words, not the previous command command line, but the one before it. !! caters for the most usual case—that of executing the previous command line again. !<string> describes a command line textually, by defining the characters with which it starts. For example:

```
!co
```

means "execute the last command line which begins with the characters co". This would match the commands copy, compare and so on. !?<string> is similar to the previous case, in that it describes a command line textually, but in this instance the position of the characters in the line is undefined. For example:

```
!?bcpl
```

means "execute the last command line which contains the characters bcpl".

Line editing

After the text which selects a previous command line, edit commands can be used to make simple alterations. Edit commands are of the general form:

```
<delim> <before> <delim> <after> <delim>
```

where <delim> defines one of editing delimiter characters (one of /*+,?'"^~), <before> is the text to be replaced, and <after> is the replacement text. The command line which has been recalled is scanned from left to right, and the first

occurrence of the string `<before>` is replaced by the string `<after>`. Any number of these edit sequences may be applied to a command line, and only when they have all been applied successfully is the modified command line passed for execution. Given that one of the most common pieces of editing is to add something to the end of a previous command line, the *null string* is defined to match at the end of a command line. The following are examples of valid editing commands:

```
/wrong/correct/
^^ added at the end^
```

Shell commands

Shell commands are those which affect the local environment of the Shell only, and perform functions which would cause a loaded command to know too much about the Shell data structures.

The HISTORY command

The `history` command, synonyms `clist` and `cl`, causes information held in the command history to be printed out. Associated with each entry in the command history is a sequence number, which can be used to specify that line unambiguously.

The SWITCH command

The `switch` command, synonym `sw`, enables the Shell monitoring options to be interrogated and changed. Switches are simply boolean flags within the Shell, and there are three available to the user:

- `switch monitor`, when set, tells the Shell to print out status information about each loaded command which it terminates. This information includes the name of the command, the amount of time taken to load and execute the command, the amount of stack it used and the return and error codes if the command failed. When unset, no monitoring information is printed out.
- `switch reflect`, if set, tells the Shell to print out command lines passed to it from REX "obey" statements. This allows the monitoring of REX command sequences, without the necessity of modifying the original REX program. When unset, command lines are not reflected.
- `switch exec`, if set, tells the Shell to execute loaded commands. When unset, the Shell loads but does not execute commands, and this facility can be used in conjunction with `switch reflect` on order to debug REX command sequences.

In order to turn a switch off, the switch name is prefixed with "no", so for example, `switch nomonitor` disables the printing of monitor information. The default state of the switches is `nomonitor`, `noreflect` and `exec`.

The CHAR command

The **char** command enables the special characters used by the Shell, when decoding command lines and pipe expressions, to be investigated and altered. There are seven special characters available to the user:

- **char bra** is the character which introduces a pipe expression
- **char ket** is the character which terminates a pipe expression
- **char pipe** is the character which represents an anonymous pipe within a pipe expression.
- **char quote** is the character which delimits strings of uninterpreted characters
- **char sep** is the character which separates multiple synchronous commands
- **char and** is the character which separates multiple asynchronous commands
- **char escape** is the character which introduces a single uninterpreted character

The default values for all the Shell characters are **bra=(**, **ket=)**, **pipe=|**, **quote="**, **sep=;**, **and=&** and **escape=**.

Directory commands

Under the old CLI, the only command available for manipulating the current working directory was **set**. The current directory was always represented by a pointer to a lock (stored in the global *currentdir*), and the name of the current directory was lost as soon as the lock had been obtained. This scheme has various shortcomings, and the Shell directory commands were added in an attempt to fill this gap.

The SET command

The **set** command is like **run**, **globals** and so in, in that it is a command which is built-in in order to over-ride the loaded command of the same name. The function of Shell **set** command is identical to its loaded counterpart, except that it has access to the Shell "directory" data structure. This structure is presented to the user as a current working directory, and a stack of "pushed" directories. The **set** command alters the current working directory, allocating a new directory lock, and updating the name of the directory, which is saved internally within the Shell.

The UNSET command

The **unset** command allows the user to move to the parent of the current working directory. UNIX has a simple tree-shaped filing system, where each directory has one, and only one, parent. It is therefore possible to store within a directory an entry which points to the parent. This entry is called "..", and so can be selected in order to give the desired effect.

The TRIPOS filing system, on the other hand, is a directed graph structure, and a directory may have several parents. It is therefore impossible to implement this facility at a filing system level, and hence is done textually.

Filenames under TRIPOS are represented as a root name, followed by a series of directory names, followed by the final filename. It is possible to scan a filename from its end, searching backwards for either the root ":" character, or a separating "." character. The part of the filename before the character is the logical parent of the whole filename, so for example, the directory "`sys:idw.bcpl`" has the parent "`sys:idw`". Using the name of the current directory, which is saved at the time the **set** command is executed, it is possible to employ the above algorithm to calculate the name of the parent directory to be selected.

The PWD, PUSHD and POPD commands

The pwd command prints out the name of the current working directory, along with the name of any directories held on the directory stack. The **pushd** command allows the user to manipulate the stack of pushed directories. **pushd** without an argument swaps the current working directory with the top item on the directory stack, and can therefore be used to alternate between two different working directories. **pushd** followed by the name of a directory causes the current working directory to be pushed onto the directory stack, and the new one to be selected as the current working directory. The **popd** command is the opposite of **pushd** in that it "pops" the top item from the directory stack, and selects it as the current working directory, thus restoring a previous working environment. These commands are all equivalent to their UNIX counterparts.

Appendix 2: REX details

REX programs

A REX program consists of a series of statements, with each statement optionally terminated by a semicolon. These statements determine the flow of control of the program, the evaluation of expressions, and the interaction between the REX program and its Shell and I/O systems. REX input is free format, with line breaks allowed anywhere where their meaning would be unambiguous. Except in strings, the case of characters in a REX program is ignored.

Comments

Comments are introduced in REX by the comment character "$", with everything from the comment character to end of line being ignored. In order to avoid interpretation of random pieces of text as though they were REX programs, all files containing REX source must have a comment as the first non-blank item. For example:

```
$  This whole line is a comment
x  :=  3     $  This comment follows a statement
```

Numbers

Numbers are of data type *number*, and can be entered in decimal, binary, octal and hexadecimal. A decimal number is a string of decimal digits; a binary number is the prefix "#b" followed by a string of binary digits, with "#o" and "#x" being the prefixes for strings of octal and hexadecimal digits respectively. Items of type *number* are used in all arithmetic operations, and all arithmetic operations yield results which are of type *number*. Numbers are represented internally as BCPL integers, and only integer arithmetic is supported. The precision to which arithmetic operations are performed is defined by the word size of the underlying BCPL implementation.
For example:

```
12345
#b101010
#o17777
#x30ff
```

Strings

Strings are of data type *string*, and are entered as a series of characters delimited by either single or double quotes. If a quote of the same type as the delimiter is required in the string, then two quotes must be entered. Items of type *string* are used and yielded by all input/output functions, and most other data types can be coerced to and from data type *string*. Strings may have lengths from zero up to the maximum determined by the underlying BCPL system—usually 255.

For example:

```
"This is a string of characters"
'Bob''s your Uncle'
```

Booleans

Booleans are of data type *bool*, and take one of the truth values TRUE or FALSE. Items of type *bool* are used in conditional constructs, and all the comparison and relational operators yield results which are of type *bool*.
For example:

```
true
false
```

Variables

Variable names are up to 64 characters in length, the first character of which is alphabetic, and the rest of which are alphabetic, numeric, or the characters "." or "_". Variables are capable of holding any value of any data type, and the data type of a variable may vary dynamically throughout the running of a program.
For example:

```
x
A.B.C.D
file4
This_Is_A_Long_Variable_Name
```

Nil

Nil is of data type *nil*, and has no associated value. Its main use is that it can safely be compared for equality or inequality with an object of any other data type. This means that it can be used as an "error value" when an operation fails (see `openin` and `openout`), or as a list terminator when general data structures are built out of array elements (see `arrays`).
For example:

```
nil
```

Operators

REX has arithmetic, relational, string and boolean operators. Each operator evaluates its arguments, coercing them to be of the required data type, operating on them, and returning the result of the operation, together with its associated data type.

Arithmetic operators

Arithmetic operators coerce their arguments to be of data type *number*, and return values also of data type *number*. All operations are performed using integer arithmetic, and the precision to which arithmetic operations are performed is determined by the word size of the underlying BCPL implementation.

The monadic arithmetic operators are:

`+`	Plus
`-`	Minus
`abs`	Absolute
`not`	Bitwise not

The dyadic arithmetic operators are:

`+`	Plus
`-`	Minus
`*`	Multiply
`/`	Divide
`rem`	Remainder
`<<`	Shift left
`>>`	Shift right
`&`	Bitwise and
`|`	Bitwise or

For example:

```
-5
abs (a + b * c)
a rem 10
a << (b / c)
(a & #xff) = b
```

Relational operators

Relational operators compare two items, and return the data type *bool*. There are three different types of comparison which are possible, depending on the data types of the associated operands, and the possibilities are tried in turn. Firstly, if either of the operands are of data type *nil*, then the comparison is performed on the data types—rather than the values—of the operands. In this case, only the equality operators, "=" and "~=", are valid. Secondly, if either of the items being compared are of data type *number*, then the other is coerced to data type *number*, and a numeric comparison is done. Finally, if neither of the two previous conditions apply, then both arguments are coerced to data type *string*, and a lexicographic comparison is done. With string comparisons, case equating is done as a matter of course.
The relational operators are:

`<`	Less than
`>`	Greater than
`<=`	Less than or equal
`>=`	Greater than or equal
`=`	Equal
`~=`	Not equal

For example:

```
a < b
```

```
count >= 3
vegetable ~= "cabbage"
```

String operators

String operators evaluate their operands, coercing them to be of data type *string*, and return values also of data type *string*. One of the operators has no special symbol, and is represented just as the juxtaposition of two operands. String concatenation will fail if the resulting string would be longer than the allowed maximum, the value of which is implementation dependent.
The string operators are:

`<juxtaposition>`	Join strings with separating space
`||`	Join strings without separating space

For example:

```
"the value of x is" x "and y is" y
directory || "." || filename
```

Boolean operators

Boolean operators coerce their arguments to be of data type *bool*, and return values also of data type *bool*. Note that the symbols used are the same as the bitwise arithmetic operators, and as with the relational operators, it is the data type of the operands which determines the nature of the operation performed.
The monadic boolean operator is:

`not`	Logical inverse

The dyadic boolean operators are:

`&`	Logical and
`|`	Logical or

For example:

```
not flag
a & (b | c)
```

System variables

To enable convenient access to the running environment, REX provides the following variables which can be used to investigate certain system parameters.

`rc`	Current Shell return code (*number*)
`mcname`	Current machine name (*string*)
`mctype`	Current machine type (*string*)
`systype`	Current operating system type (*string*)

Note that the above variables are special in that they cannot be assigned to, and so act much more like parameterless functions than variables.

The ARGS variable

On entry to a REX program, the variable **args** is of data type ***string***, and contains the string representation of the arguments passed to the program from the Shell. This will normally be split down into its individual elements (words) using the parse statement, or parsed with respect to a pattern using the **rdargs** function. Note that **args**, unlike the system variables **rc**, **mcname** etc., is a real variable, and hence can change its value and data type.

The LEN function

The len function has the following form:

```
len( <expression> )
```

The expression is evaluated, and coerced to be of data type ***string***. The result of the function is of data type ***number***, and is the number of characters in the string, in other words, its length.
For example:

```
len( "The flowers that bloom in the Spring" )
len( string1 || string2 )
```

The SUBSTR function

The substr function has the following form:

```
substr( <STR expression>, <LWB expression>, <UPB expression> )
```

The STR expression is evaluated, and coerced to be of data type ***string***. The LWB and UPB expressions are both evaluated, and coerced to be of data type ***number***. The result of the function is of data type ***string***, and is the substring of the STR argument, from position LWB to position UPB. Substrings of length 1, in other words, individual characters, can be obtained by using the **array subscript** notation, treating a string as an array of characters.
For example:

```
substr( "ABCDEF", 2, 4 )
substr( line, 10, len( line ) )
buffer[ 23 ]
```

Explicit coercions

Sometimes, the default coercion performed by REX is not what is required. To enable the programmer to override the default coercions, REX provides a series of functions which allow explicit coercions to be made. In the following list, the data type *char* (character) is not a true data type, but is simply a ***string*** of length 1.

num(<exp>)	Coerces <exp> to *number*
str(<exp>)	Coerces <exp> to *string*
ustr(<exp>)	Coerces <exp> to upper case *string*
lstr(<exp>)	Coerces <exp> to lower case *string*
chr(<exp>)	Coerces <exp> to *char*

```
asc( <exp> )                    Coerces <exp> from char to number
                                (its ASCII code)
```

For example:

```
num( a ) < num( a )
str( 1234 )
ustr( x || y || z )
chr( 13 )
asc( "a" )
```

Precedence of operators

The REX operators, in order of decreasing precedence, are as follows:

```
[]  ()                          Arrays & procedures
+ - abs not                     Monadic
* / rem
+ -
<< >> ||
> < = >= <= ~=
& |
<juxtaposition>
```

The precedence of operators can always be over-ridden by putting expressions within parentheses.

Synonyms

The following symbols are synonyms of one another:

```
not  \  ~
~=  \=
```

Arrays

REX allows the programmer to define arrays of up to 5 dimensions, with each array element having the property of a single variable. This means that the different elements in an array may be of totally different data types, and hence arrays can be thought of more as multi-dimensional records than as conventional matrices.

Array elements

Array elements are accessed by applying a subscript expression to something which evaluates to be of data type *array* (usually a variable). A subscript expression is a pair of square brackets, containing a list of subscripts separated by commas. Each subscript is coerced to be of data type *number* before being used as an index into the array.

For convenience, items of data type *string* can be treated exactly as if they were single dimension arrays of characters. Single characters may be extracted from the string in the expected manner, and one or more characters can be replaced by using array assignments.

For example:

```
matrix[ i*2, j+4 ]
(procedures[ 4 ])[ one, two, three ]
string[ 4 ]
string[ 10 ]  :=  "abcde"
```

The DIM statement

The dim statement has the following form:

```
dim <list of array definitions>
```

Each array definition comprises a variable name, followed by a pair of square brackets, containing a list of subscript bounds separated by commas. The subscript bounds are given either by a single expression representing the upper bound, or a pair of expressions separated by a colon, representing the lower and upper bounds. If the lower bound is omitted, a value of 1 is assumed.

Executing `dim` has the effect of allocating a new storage element and defining the dimensionality and bounds of the array. Because of this, `dim` can be used to define elements of more complicated data structures, and since arrays may themselves contain references to other arrays, trees and lists are easy to construct. After the array is defined, the variable is assigned a pointer to the array, and given the data type *array*.
For example:

```
dim values[ 100 ]
dim a[ 10, 10 ], b[ 10, 20, 30 ]
dim matrix1[ -5:+5 ], matrix2[ 0:10, 0:10 ]
```

The TABLE statement

The `table` statement has the following form:

```
table <list of expressions>
```

Tables are single dimension arrays whose elements are initialised. They have data type *array*, dimensionality 1, lower bound 1, with the upper bound being set by the number of items in the expression list. Each of the items in the list is evaluated, and the resulting value and data type is assigned to the appropriate array element. As with arrays, individual elements of tables can be of any data type, and each time a table is defined, a new storage element is allocated. This means that tables can easily be used to allocate records when building a data structure.
For example:

```
veg  :=  table "Courgette", "Parsnip", "Artichoke"
res  :=  table nil, val1, val2, val3
```

General control statements

REX provides a series of general control statements, which are involved with assignment, conditional execution, looping and other flow control issues. The number of these constructs has been kept to a minimum, without reducing the facilities available to the programmer.

The assignment statement

An assignment statement has the form:

```
<variable item>  :=  <expression>
```

The expression is evaluated, and the resulting value and its associated data type are assigned to the variable. The variable item must be something which is capable of being updated, and hence must be a variable or an array element. If the variable item is already defined, then there is no necessity for its new value and data type to be compatible with the old.
For example:

```
answer   :=  42
message  :=  "HELP!"
a[ 1 ]   :=  true
```

The IF statement

The `if` statement has the following form:

```
if  <expression>  then  <statements>  fi
if  <expression>  then  <statements>  else  <statements>  fi
```

The expression is evaluated, and coerced to be of data type *bool*. If the value is **true**, then the statements of the **then** clause are executed. If the value is **false**, then, if an **else** clause has been given, its statements are executed instead. Multiple *if then else* statements can be linked by use of **elif**, with only one **fi** being required at the end of the set of statements.
For example:

```
if  x = 3  then  x := x + 1  fi

if  action = "greeting"
    then  reply  :=  "hello"
    else  reply  :=  "goodbye"
fi

if     person = "happy"
       then  state  :=  "cheerful"
elif   person = "sad"
       then  state  :=  "miserable"
       else  state  :=  "mediocre"
fi
```

The DO statement

The do statement is the single REX looping construct, providing all the facilities of for, while, until and repeat loops in other languages. Associated with each loop are six parameters, each of which can be omitted by the user if that facility is not actually required. The parameters, if present, must be introduced by one of the following keywords:

```
for <variable>
from <expression>
to <expression>
by <expression>
while <expression>
until <expression>
```

During execution, the loop is represented by a loop count, which starts at the value given by the from parameter, and continues until it exceeds the to parameter. Each time round the loop, the loop count is adjusted by the value of the by parameter, which can be positive or negative. If the for parameter is present, then the value of the loop count is assigned to this variable each time the loop is executed. As well as the loop count, there are two boolean conditions which are tested each time round the loop. The while parameter is evaluated, and looping only continues if the value is true. Similarly, the until parameter is evaluated, and looping continues if the value is false.

If any of the above parameters are omitted, then suitable default values are taken. The default values are:

```
from 1
to <maxint>
by 1
while true
until false
```

where <maxint> is the largest positive integer capable of being represented by the underlying BCPL system. The result of the defaults is that, if no parameters are given (the simple "do ... od" construct), the effect is to execute the loop <maxint> times, or in practice, for ever.
For example:

```
do  handlecommands()  od

for i to 10 do  array[ i ]  :=  -i  od

until x > 100 do  x  :=  function( x )  od
```

The BREAK statement

The break statement has the following form:

```
break
```

Executing break has the effect of leaving the current do loop prematurely, overriding the terminating loop conditions.

The LOOP statement

The `loop` statement has the following form:

```
loop
```

Executing `loop` has the effect of skipping the rest of the statements in the **do** loop, and going to the point where the looping conditions are re-calculated.

The SKIP statement

The `skip` statement has the form:

```
skip
```

Executing `skip` has no effect whatsoever. **skip** is a dummy operation, and is included because **proc**, **if** and **do** clauses, each require at least one executable statement.

The EXIT statement

The `exit` statement has the following form:

```
exit( <expression list> )
```

The expression list is a (possibly empty) list of expressions, separated by commas. If the expression list is not empty, then the first expression is evaluated, and coerced to be of data type *number*. The rest of the expressions are evaluated, coerced to be of data type *string*, and concatenated together with separating space characters. The effect of the **exit** statement is to leave the current REX program, no matter how deep the loop nesting or procedure recursion. The *number*, if present, is taken as a return code, and passed back to the operating system. The *string*, if present, is printed out as an error message to the main Shell output channel.
For example:

```
exit()
exit( 10 )
exit( 20, "Compilation of" file "failed" )
```

Procedures

A procedure is a group of statements which are executed in a slightly different environment to the main program. All procedures can be recursive and may or may not return a result, thus allowing the same body of code to be used as either a routine or a function.

Within the procedure, variables defined outside the procedure may be referenced, but when a variable assignment is executed, a copy of the variable local to the procedure is made. The variables in the formal parameter list for the procedure are also local to the procedure, the calling mechanism therefore being "call by value". Note that there is no concept of a global variable, and the only way a procedure can cause side effects is to update array elements.

Procedure calling

Procedures are called by applying a parameter expression to something which evaluates to be of data type *proc* (usually a variable). A parameter expression is a pair of round brackets, containing a list of parameters separated by commas. If no parameters are being passed, then the list is empty. Each of the parameters is evaluated, and their values and associated data types are assigned to the variables which form the procedure's formal parameter list. The number of parameters passed must correspond exactly to the number expected by the procedure. After the parameters have been set up, the statements of the procedure body are executed. If control reaches the end of the statements, then the procedure returns without passing back a result.

Since a procedure may or may not return a result, a procedure call is valid both as a statement and as an item in an expression.
For example:

```
x  :=  function( 1, 2, 3 ) + 4
subroutine( "a b c d", arg2, x[ 4, 5 ] )
(procedures[ 5 ])()
```

The PROC statement

The proc statement has the following form:

```
proc <variable> ( <variable list> ):  <statements>  corp
```

Executing proc has the effect of defining a procedure ready for use later in the program. The variable list is a (possibly empty) list of variables, separated by commas, which define the procedure's formal parameters, and the statements form the body of the procedure. After the procedure is defined, the variable is assigned a reference to the procedure, and given the data type *proc*.
For example:

```
proc update( a, b, c ):  a[ 1 ] := b + c  corp

proc apply( subroutine, x ):
     for i to x  do  subroutine( x )  od
corp
```

The RETURN statement

The return statement has the following form:

```
return
```

Executing `return` has the effect of causing return from the current procedure, without passing back a result.

The RESULT statement

The result statement has the following form:

```
result <expression>
```

Executing `result` has the effect of causing return from the current procedure, with the expression being evaluated, and the resulting value and its associated

data type being passed back as the result of the procedure. Note that, through the use of the `table` statement, more than one result can effectively be returned from a procedure.
For example:

```
result 3
result "The answer is" 42
result table 100, 200, 300
```

The DUMPSTACK statement

The `dumpstack` statement has the form:

```
dumpstack( <expression list> )
```

The expression list is a (possibly empty) list of expressions separated by commas. The items in the expression list are evaluated, coerced to be of data type *string*, and concatenated together with separating space characters. The effect of the `dumpstack` statement is to dump the state of all the variables on the program stack to the main Shell output channel. The position of the stack high water mark, stack frame and stack base are all printed out, along with the names and values of the variables on the stack. The *string* calculated from the expression list is printed out as a message at the head of the variable dump. Note that the `dumpstack` statement is primarily provided for debugging use only.
For example:

```
dumpstack()
dumpstack( "Entering procedure CALCULATE" )
```

Interface to the Shell

Because of the way that the REX system was conceived, there is a simple and clean interface between programs written in REX and the Shell environment. It is possible to perform I/O using the main Shell I/O channels, and command lines can be passed to the Shell for execution.

The SAY statement

The say statement has the following form:

```
say <expression>
```

The expression is evaluated and coerced to be of data type *string*, and then written out to the main Shell output channel, with a terminating newline character.
For example:

```
say "The value of x is" x "and x squared is" x*x
say "*** Command" command "failed with return code" rc
say a b c d e
```

The QUEUE statement

The queue statement has the following form:

```
queue <expression>
```

The expression is evaluated and coerced to be of data type *string*, and then enqueued onto the main Shell input channel, ready to be read by the Shell or a program running under it.
For example:

```
queue "diablo"
queue "margin" margin "; pause; go"
```

The OBEY statement

The obey statement has the following form:

```
obey <expression>
```

The expression is evaluated and coerced to be of data type *string*, and then passed to the Shell as a command line to be obeyed, exactly as if the command line had been typed at the console. The command line is obeyed immediately, and hence any program which is loaded may read input lines which have previously been enqueued.
For example:

```
obey "print" file "opt vdu"
obey "bcpl bcpl." || file  "to obj." || file
```

The PARSE statement

The parse statement has the following form:

```
parse <expression> : <variable item list>
```

The expression is evaluated, and coerced to be of data type *string*, and is then parsed into the variable item list. The variable item list is a list of either variables or array elements separated by commas. If only one variable is present, then `parse` is exactly equivalent to an assignment statement, with the expression being assigned to the variable. If more than one variable is present, then the expression is split into "words", each of data type *string*. One word is assigned to each variable, except the last variable in the list which has whatever is left of the expression assigned to it.

Words are defined as being strings of non-blank characters, separated by spaces. Words may contain spaces, so long as they are contained within quotes (see the definition of "strings" for the syntax used), the outermost level of quotes being stripped off by the parsing operation. `parse` is also called implicitly by the `read` and `prompt` statements, allowing input lines to be parsed as they are read.
For example, after:

```
parse "The 'parse statement' at work": a, b, c
```

The variables would have the following values:

```
a       "The"
b       "parse statement"
c       "at work"
```

For example:

```
parse line: a, b, c
parse args: file, etc
parse a[ 1 ] || a[ 2 ]: val1, val2, val3
```

The RDARGS function

The rdargs function has the following form:

```
rdargs( <expression>, <pattern>, <variable item list> )
```

The expression and pattern are both evaluated, and coerced to be of data type *string.* The BCPL procedure *rdargs* is then applied, parsing the expression with respect to the pattern given, assigning each of the results to the corresponding variable or array element. The number of items in the variable item list must correspond exactly with the number of results yielded by the parsing operation. Each variable will have data type *string* if the relevant argument is given—for "/A" and "/K" items, the value corresponds to the parsed result, and for "/S" items, the value corresponds to the name of the switch. In both cases, if the argument or switch is omitted, the variable will have the value *nil.* The resulting value is of type *bool*, and indicates whether the parsing operations was successful. For example:

```
ok  :=  rdargs( args, "from/a,to/k,quiet/s", a.f, a.t, a.q )
ok  :=  rdargs( string, pattern, v1, v2, v3, v4 )
```

The PROMPT statement

The prompt statement has the following form:

```
prompt <expression> : <variable item list>
```

The expression is evaluated, and coerced to be of data type *string.* This string is then printed out to the main Shell output channel as a prompt, without a terminating newline character. A line is then read from the main Shell input channel, and parsed into the variable item list, exactly as in the **parse** statement. For example:

```
prompt "File name: ": filename
prompt "A B and C: ": a, b, c
```

Input/output

Simple I/O to the main Shell channels is available using the **prompt** and **say** statements, but REX also allows similar I/O to other user defined channels. There are facilities to open and close channels, read from and write to channels, and test for channel "end of file". All I/O is on a record by record basis, with records

being separated by the newline character.

The OPENIN and OPENOUT functions

The openin and openout functions have the following form:

```
openin( <expression> )
openout( <expression> )
```

The expression is evaluated, and coerced to be of data type *string*. The resulting string is taken as the name of an operating system file, and opened for input if the function is openin, or output if the function is openout. On success, the resulting value is a pointer to the open channel, with its associated data type of *input channel* or *output channel*. On failure, the resulting value is *nil*. Note that the form which the file name takes is operating system dependent.
For example:

```
inchannel   :=  openin( "a.b.c" )
outchannel  :=  openout( dir || "." || file )
```

The READ statement

The read statement has the following form:

```
read <expression> : <variable item list>
```

The expression is evaluated, and coerced to be of data type *input channel*. A line is then read from the input channel, and parsed into the variable item list, exactly as in the parse and prompt statements.
For example:

```
read inchannel: line
read inchannel: a, b, c, etc
```

The WRITE statement

The write statement has the following form:

```
write <expression> : <expression>
```

The first expression is evaluated, and coerced to be of data type *output channel*. The second expression is evaluated, coerced to be of data type *string*, and then written out to the specified output channel, with a terminating newline character.
For example:

```
write outchannel: "My name is" myname
write outchannel: a b c d
```

The CLOSE statement

The close statement has the following form:

```
close( <expression> )
```

The expression is evaluated, and coerced to be of data type either *input channel* or *output channel*. The specified channel is then closed, after which no further I/O on that channel is possible.

For example:

```
close( inchannel )
close( outchannel )
```

The EOF function

The **eof** function has the following form:

```
eof( <expression> )
```

The expression is evaluated, and coerced to be of data type *input channel.* The result is of data type *bool,* and is **true** if end of file has been reached on that channel, or **false** otherwise.

For example:

```
endoffile  :=  eof( inchannel )

if  eof( inchannel )  then  break  fi
```