Technical Report

UCAM-CL-TR-166
ISSN 1476-2986

Number 166

Evolution of operating system structures

Jean Bacon

March 1989

ISSN 1476-2986

Evolution of Operating System Structures

Jean Bacon   March 89

Abstract

The development of structuring within operating systems is reviewed and related to the simultaneous evolution of concurrent programming languages. First, traditional, multi-user systems are considered and their evolution from monolithic closed systems to general domain structured systems is traced. Hardware support for protected sharing is emphasised for this type of system.

The technology directed trend towards single user workstations requires a different emphasis in system design. The reqirement for protection in such systems is less strong than in multi-user systems and, in a single language system, may to some extent be provided by software at compile time rather than hardware at run time. Distributed systems comprising single user workstations and dedicated server machines are considered and the special requirements for efficient implementation of servers are discussed.

The concepts of closed but structured and open system designs are helpful. It is argued that the open approach is most suited to the requirements of single user and distributed systems. Experiences of attempting to implement systems over a closed operating system base are presented.

Progress towards support for heterogeneity in distributed systems, so that interacting components written in a range of languages may interwork and may run on a variety of hardware, is presented.

The benefits of taking an object orientated view for system-level as well as language-level objects and for specification, generation and design of systems are discussed and work in this area is described.

An outline of formal approaches aimed at specification, verification and automatic generation of software is given.

Finally, design issues are summarised and conclusions drawn.

Contents

1. Introduction

In the late 60's and early 70's a great deal of research interest was directed towards design methods for structured operating systems. At this time, the main targets of the work were the large, timeshared mainframes since their operating systems were seen as prime examples illustrating the "software crisis". They were large, complex, incompletely specified, never fully debugged and difficult to maintain. Such systems are **closed** since all internal functions are hidden and their interfaces, the set of system calls, proscribe the functionality available to their users, see section 2.

The explicit statement of the concept of **process** and the formalisation of mechanisms for **inter process communication** provided tools for managing the concurrency aspects of operating systems design. Structuring tools, other than "program" and "procedure" were still to be developed. Early systems, often from research environments, based on processes with various primitives to support their inter-communication are described in section 3. Also at this stage, operating systems began to be written in high level languages sometimes with explicit support for concurrency. Two styles of system design were seen to have evolved, **message-based** (where service is requested from some other process by sending a message to it) and **procedural** (do-it-yourself by making procedure calls). Although functionally equivalent, the implementation efficiency may differ greatly. The requirement for a design method to aid the decomposition of systems into modules led to the **object model** of system design and associated structuring tools in languages (section 4).

Multi-user systems require hardware support for protection and sharing. Since the system must provide continuous service to all users it must be protected from maliciousness and error. Users must also be protected from each other. **Protection** may be provided by running each process in a separate virtual **address space.** The operating system may run in a separate address space or may be in every processes' address space. In this case protection may be achieved by a change to privileged state when a system call is made. **Sharing** is also desirable in order to avoid multiple copies of utilities such as editors, compilers and language support systems. This may be supported by providing each process with a segmented address space. In systems which support large numbers of segments per process, much of the operating system may also be provided as segments of each process and there is potential for a finer grain of protection than all-or-nothing. This line of development leads, via rings which are nested protection domains, to architectures and operating system structures based on fine grained protection domains. The incorporation of hardware support into system structure is reviewed in section 5. Although internally well structured, such systems are typically closed.

During the 1970's first minicomputers then microcomputers became a cost-effective way of providing computing power. For single user sytems the requirements for protection and sharing must be completely re-examined, although the requirement for a well structured operating system is still important for software management. Unfortunately, the research lessons had not become sufficiently widely known and accepted when the personal computer explosion took place and many operating systems for personal computers reverted to unstructured monoliths, tolerable because the total system size was small. Some single user workstations were carefully designed however, often with an **open structure**, in which the system modules are visible to the user and may be bypassed, or replaced. These are reviewed in section 6. Many minis and micros are still used as multi-user systems and a compromise must be reached on the degree of structuring and hardware support required. Unix^{TM} flourished in this environment. It was seen as advanced, compared with simpler, cruder operating systems marketed with personal computers and refreshingly simple compared with operating systems developed for mainframes and supporting hundreds of users. It's manufacturer independence was attractive for non-technical reasons. Above the Unix interface a good program development environment is provided. Below it is a structure appropriate to its design date in the early 1970's and aspects of this design are described in section 7.

The development of high bandwidth Local Area Network (LAN) media during the 1970's led to much research and development in distributed systems. Single user workstations, or single user machines acquired dynamically from a pool of processors, were augmented by services available across the network. Expensive printers and large, shared file stores [Svobodova84] could be provided in this way. The operating system in the user's machine is simpler since file management and much device management resides elsewhere. On the other hand, communications handling software must be provided. There is a requirement to support the development of such systems in distributed concurrent programming languages. Software structures that have been used as a basis for distributed systems are reviewed in section 8. A report of the problems arising from attempting to implement distributed systems above a closed system with an inappropriate interface, such as Unix, is also given.

It has now become commonplace to interconnect disparate LAN media and heterogeneous systems and therefore to have to incorporate the various associated protocol families into software systems. There is also a requirement for distributed systems to support the interaction between components written in a range of languages running on a variety of hardware. Such designs are the subject of research and standardisation and progress is reported in section 9.

The object orientated approach to programming and system design appears attractive as a possible way to manage the complexity of large systems. Some current systems take a unified, object orientated view in which large system objects, such as services, are specified and generated by similar techniques to those used for objects within a program. Progress in this area is reviewed in section 10.

Section 11 gives a brief overview of current research into formal approaches to designing and building systems. The long term goal is that a formal specification will be automatically translated into an implementation. The implementation language may, because of its mathematical basis, be unacceptably inefficient, for example, a pure applicative language based on stateless functions. Efficiency transformations may then automatically produce an implementation more akin to conventional imperative languages. Research is still at an early stage, particularly in the area of support for concurrency.

Section 12 summarises the lessons learned and the consensus established and highlights current research issues in systems design.

2. Monolithic Systems

A closed operating system provides an external interface, the set of system calls, to running programs [fig.1,12]. The characteristics of the real resources that it manages are hidden and it may be considered to be creating a virtual machine which is easier to use than the real machine. For example, the virtual machine works in terms of files with textnames, the real machine in terms of disc pages; the virtual machine in lines of text, the real machine in units of a single character. When IBM came to use the term

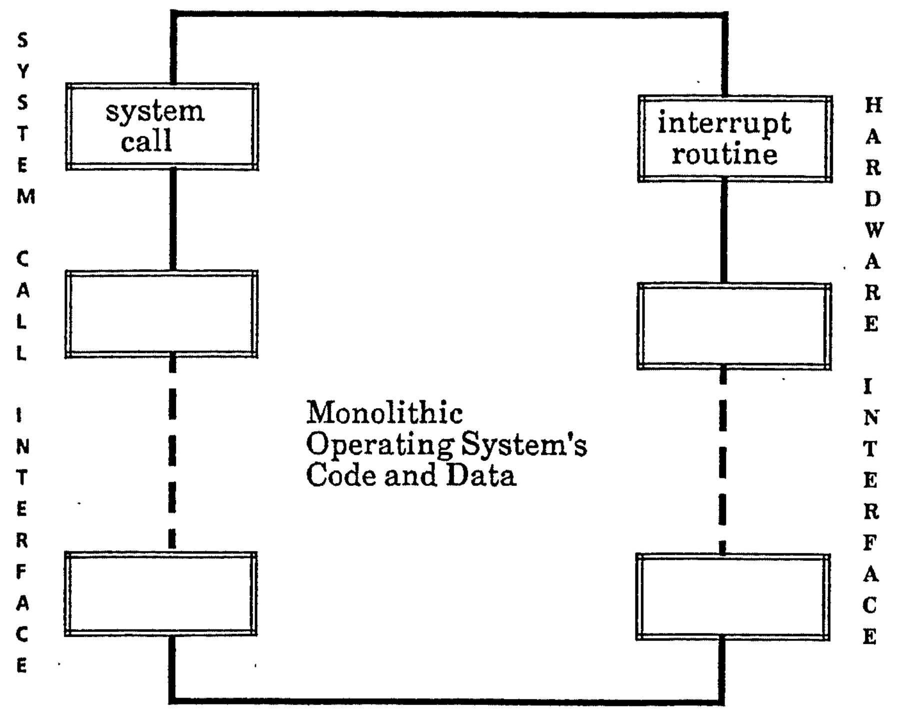

fig.1 a monolithic closed operating system

Virtual Machine in VM/370 they defined it to be a true superset of the real machine, ie. all the operation codes of the real machine are available in the virtual machine. The earlier and more general usage of the term is just of a higher level interface than the raw hardware.

Initial research into structured operating systems retained the system call interface and closed system approach. This was appropriate for multi-user systems in order to prevent users accessing resources in an uncontrolled way. The issue addressed was, therefore, how the monolithic system's code and data should be partitioned in accordance with its resource management functions. Also, ad-hoc methods had evolved to handle the inherent concurrency deriving from the simultaneous management of many devices and many users and a systematic approach was needed.

3. Processes and Inter-process Communication

The first major step towards a coherent approach to operating system design came from the explicit statement of the concept of **process**, for example in the Multics supervisor [Vyssotsky 65], and the formalisation of the means by which processes cooperate to perform a service or compete for resources.

In the THE system Dijkstra et al [Dijkstra68] designed a strict hierarchy of virtual resources managed by a layered system of processes. A brief outline of its structure is as follows, also see [fig.2]. The lowest level, level 0, creates virtual processors. Above this, processes exist and may communicate using a semaphore mechanism. The rest of the system may be written using these concurrency tools. All interrupts enter at this level, all but the clock interrupt are handled at higher levels. At level 1, one process provides a one-level virtual store. It synchronises with drum interrupts and with

level 4 (five processes)	five user processes
level 3 (one process for each physical device)	creates virtual devices synchronises with device interrupts synchronises with the memory manager and console process synchronises with requests from higher levels
level 2 (one process)	creates virtual consoles synchronises with console interrupts synchronises with the memory manager synchronises with requests from higher levels
level 1 (one process)	creates virtual memory synchronises with drum interrupts synchronises with requests from higher levels
level 0	creates virtual processors provides semaphores for ipc handles clock interrupt acknowledges other interrupts which are serviced at higher levels

fig.2 Structure of THE

requests for store from higher level processes. At level 2, one process provides virtual consoles for higher level processes and synchronises with their requests. It also synchronises with console interrupts and the memory manager process. At level 3, a separate process manages each physical device. Each process synchronises with it's device's interrupts, with the memory and console manager processes, and with higher level processes over I/O requests. At level 4, the highest, reside five user processes. In all cases where data are passed, producer - consumer style message buffering must be used, controlled by semaphores.

The concept that the lowest level of the operating system should create virtual processors, the **process model**, is widely used, with variations on the inter-process communication mechanism provided. A strict hierarchy as used in THE causes problems over choice of order: either the memory manager cannot output messages to the console or the console manager cannot use virtual memory, and has sometimes been used in modified form as suggested in [Haberman76]. A layered structure may be useful on a smaller scale, for example the protocol layers within a communications subsystem.

Semaphores are at too low a level for general use No assistance is offered to ensure that mutual exclusion and synchronisation are programmed correctly using them. One development is to make the primitives provided at the lowest level more powerful, as in **message passing systems** [Morris68, Brinch Hansen70], see below. An alternative is to hide them below higher level concurrency and structuring constructs provided by a programming language. Another important property of procedural systems is that processes must run in a shared address space in order to call shared procedures. A process gets work done by threading its way through the system, calling procedures, acquiring locks to share data and synchronising with other processes when necessary [fig.3].

In a retrospective paper on THE [Dijkstra71] Dijkstra proposed that the critical regions embedded in processes and delimited by semaphore operations, as used in THE, should be replaced by calls to a secretary process which encapsulates the operations on the shared object. Structuring ideas along these lines together with language level support were actively researched in the early 70's and led to language level **critical regions** [Hoare72, Brinch Hansen72] and **monitors** [Brinch Hansen73, Hoare74]. Monitors may be implemented by the language run-time system and the kernel invoked only when a change of process state is needed. The language provides a higher level syntax and the compiler ensures that a variable declared as shared is only accessed from within a critical region or a monitor procedure. A comprehensive review of concurrent programming languages is given in [Andrews83].

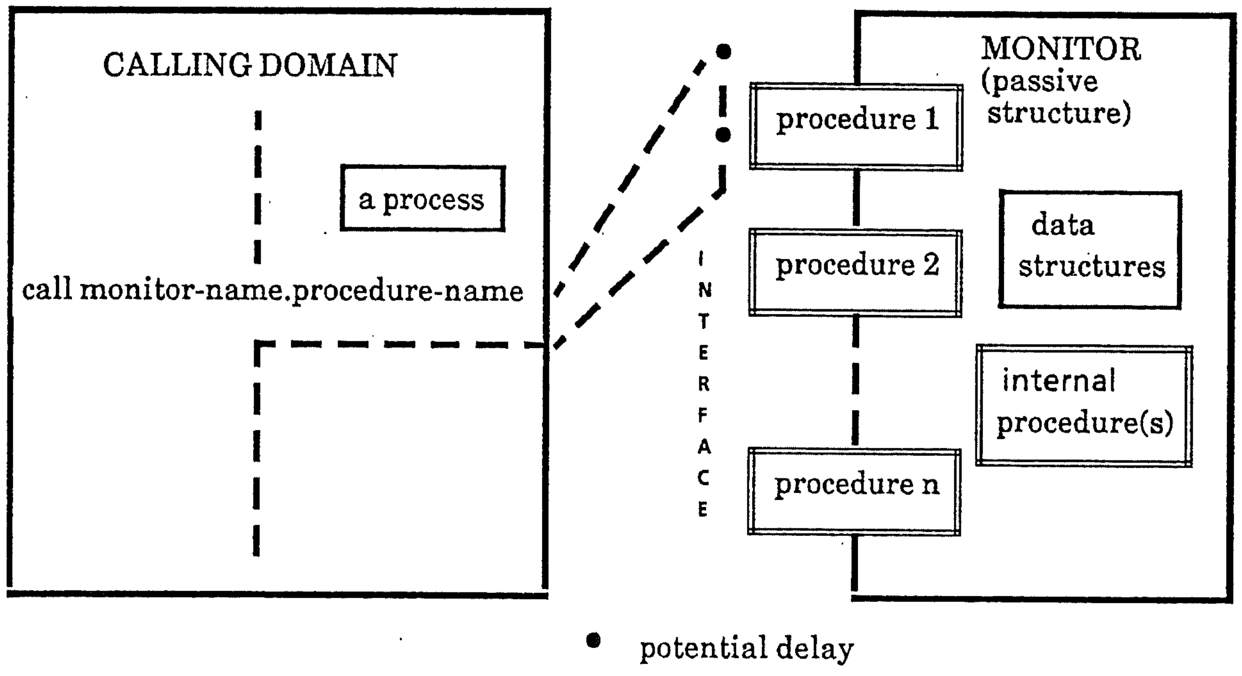

fig.3 Procedural System Structure

The justification for message passing systems is that processes usually synchronise to pass data and it is therefore reasonable for the system to support this activity at the lowest level. Message passing, or some similar mechanism, is essential when processes do not share memory, for example, when they run in separate address spaces [fig.4]. A problem is that the system kernel has to manage message buffers and deal with the possibility that they may become full and are at a level below virtual memory. Fixed length messages are easier to manage than variable length but provide a very low level facility. If message passing is to be made visible to the user it should be in terms of typed messages associated with typed ports as in the Conic system [Sloman84]. Synchronous message passing avoids the problem of buffer management and makes

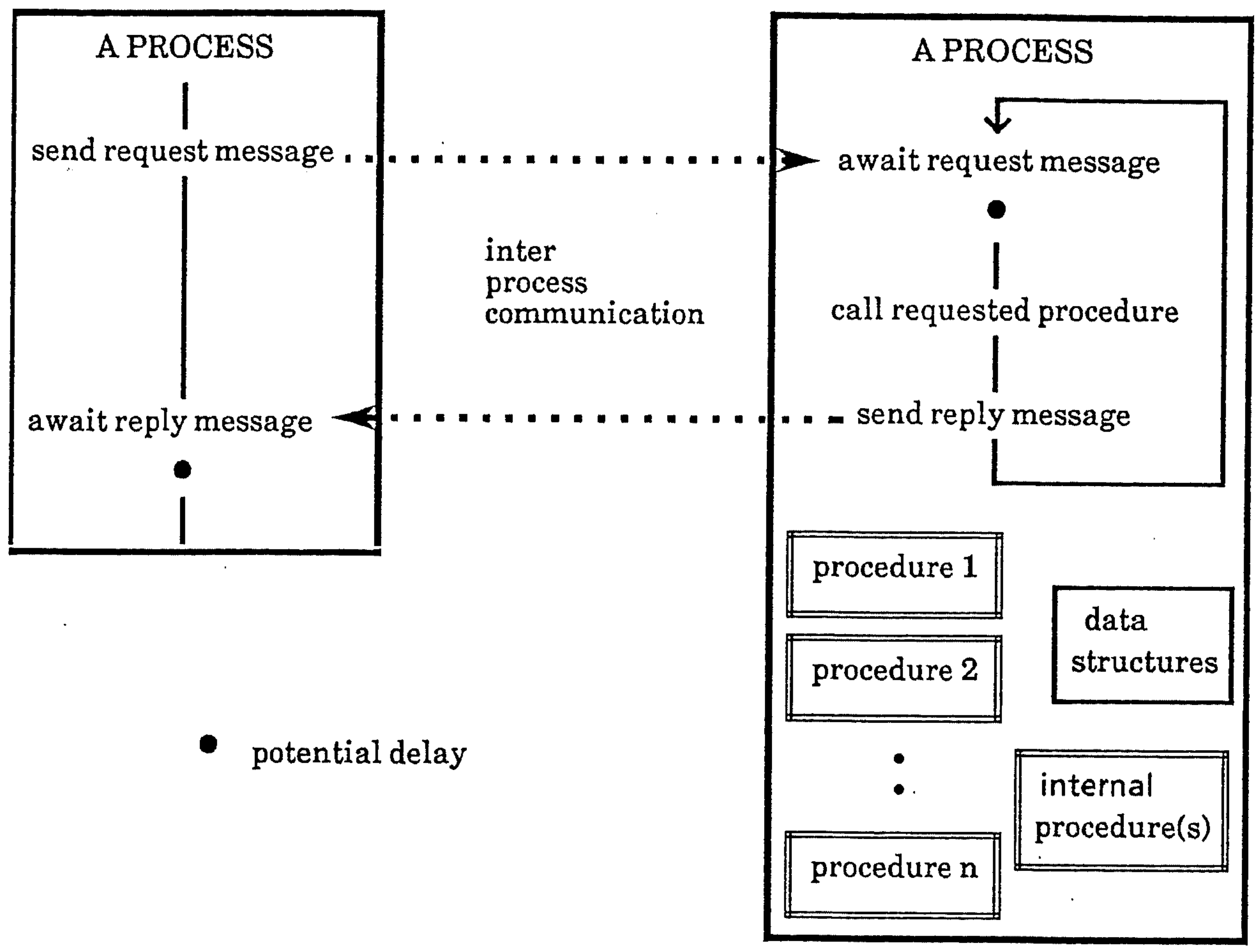

fig.4 Process, Message-passing Structure

system behaviour easier to comprehend and model mathematically. In practice, synchronous mechanisms make it essential to build extra buffering processes at a higher level in cases where it is not feasible for processes, such as system service processes, to wait for synchronisation.

The system model associated with **procedural**, concurrent languages is of processes running within a single address space. They may therefore call shared procedures to access shared data. **Monitors** are typically used to provide exclusive access to shared resources with condition variables for synchronisation over the state of the resource. The functional equivalence of the message passing and procedural models has been pointed out in [Lauer78]. It is often the case, however, that the procedural model may be implemented very efficiently. In a carefully designed system, monitor locks are rarely found to be claimed when tested, since the function of the monitor is to allocate a process to a queue associated with a specific condition. A process may test and set an unclaimed lock without kernel intervention and proceed into the monitor at the overhead of a simple procedure call. Only when the lock is already claimed need a trap into the kernel occur to block the process. In a message system, kernel action is required on all message primitive invocations.

4. Modular Decomposition

The above discussion has focussed on the use of **concurrency tools** in operating system design. Processes may be assigned according to the sources of asynchronous behaviour in a system, its users and its devices. This, in itself, is insufficient to produce the granularity of decomposition necessary for designing and implementing a large software system. A system design method and associated **structuring tools** are required. The **object model**, proposed in [Jones78], was clearly in line with developments in language systems towards support for data abstraction. In outline, each module in the system should represent either a single, distinct abstraction or an external resource and should provide a procedural interface to the operations defined for that abstraction. In a typical operating system, objects might be files, directories, streams, etc. An attractive feature of object orientated systems and languages is their ability to handle references to objects of arbitrary structure and complexity.

If the object paradigm is applied throughout all levels of system design, an operating system may be regarded as an object with the system calls as its interface, as may all the resource management modules within it. Each of these, in turn, operate on resource objects. Support is required for the object structure and operation invocation. It is useful to restate the basic terminology and definitions that are used in this paper since there are many variations on the basic object model:

An **object** is one of possibly many instances of its **class** or **template**. It can only assume values from a single, predefined (and possibly infinite) set of values and it can be manipulated only via a predefined set of operations [fig.5].

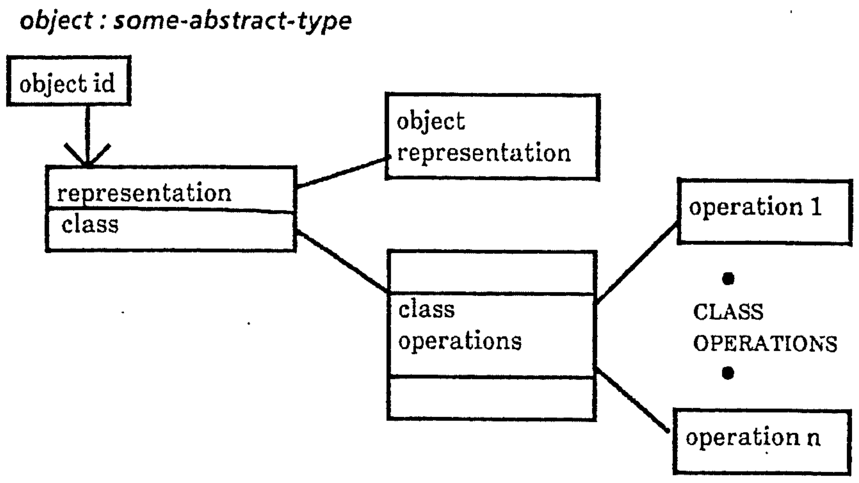

fig.5 An Object

The set of operations, their semantics and the set of possible values form the **abstract type** of the object [fig.6], while the implementation code for the operations (applicable to objects of that abstract type) is known as a class of the object. There may be several alternative implementations of the abstract type of the object, in which case there are correspondingly several alternative classes. Objects of a given abstract type may have different representations. An example is the abstract type *complex*. An object of this type may be represented as a modulus and argument pair or as a real part, imaginary part pair. The implementation code for *complex* operations differs for these alternative

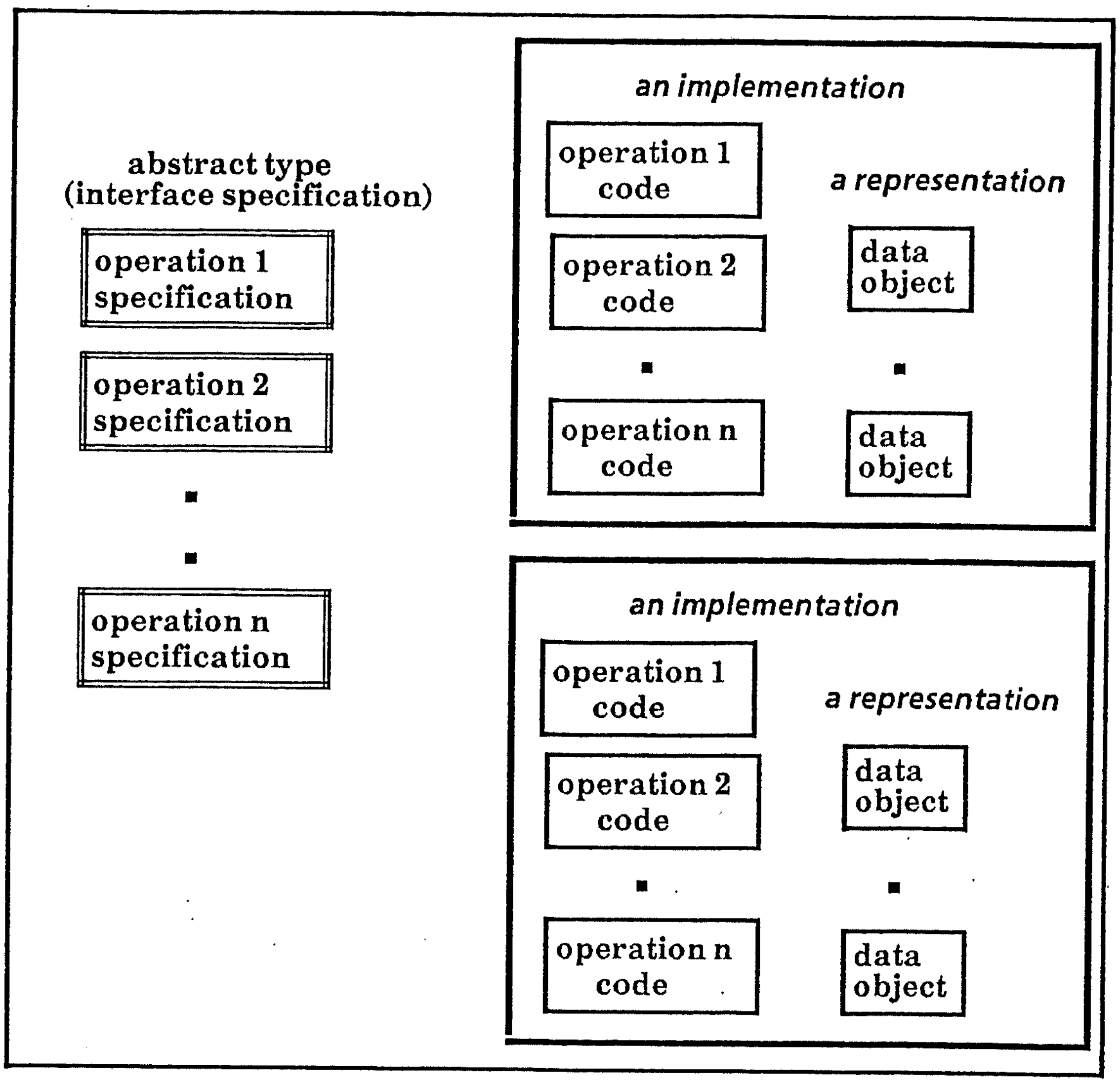

fig. 6 An Abstract Type

representations. A more system orientated example of the requirement for different representations and implementations is that a small object may be brought into main memory in its entirety whereas a large object of the same abstract type may need to be operated upon piecemeal. Another example is that a service may be implemented to run on different hardware.

An **active object** is one to which one or more processes are bound. An active object can therefore change its own state (value) independently of external operation invocation. A **passive object** has no internal processes. It has state but this may only be changed by external invocation of its operations.

If an object is passive its operations are invoked by procedure call by some external process. If an object is active, its operations may be invoked by message passing or by mechanisms such as remote procedure call, procedural rendezvous etc.

An **interface** is a defined subset (possibly the entire set) of the operations available on an object. An object may support more than one interface. An interface may be considered as a stateless object since it has a set of operations but an empty set of possible values.

This raises the issue of how close the object model for system design should be to the concepts of object orientated languages [Cardelli 85]. The use of data abstraction and information hiding is common to object orientated systems and languages. Other language concepts and mechanisms may be applicable to software engineering such as the inheritance of operations by one class from one or more other classes.

Languages with support for the separate compilation of modules such as Mesa, CLU and Modula 2 may be regarded as supporting an object orientated style and facilitate an object orientated approach within the system modules which they are used to implement. Higher level tools are required if large system components are to be configured and managed in a similar way.

The work on abstraction as an aid to modular decomposition also established the principle that the system designer should avoid building in policy decisions wherever possible. Instead, **mechanisms** should be provided which allow a variety of **policies** to be implemented. This principle was used in Hydra [Wulf75, Levin75].

5. Hardware Support for System Structure

5.1 Segmentation for Protection and Sharing

In multi-user systems it is necessary to protect the operating system from users and users from each other. It is desirable to support sharing, at least of system utilities. Segmentation hardware supports protection and sharing. A process runs in a separate, but segmented, virtual **address space**; a given segment may be in the address space of any number of processes.

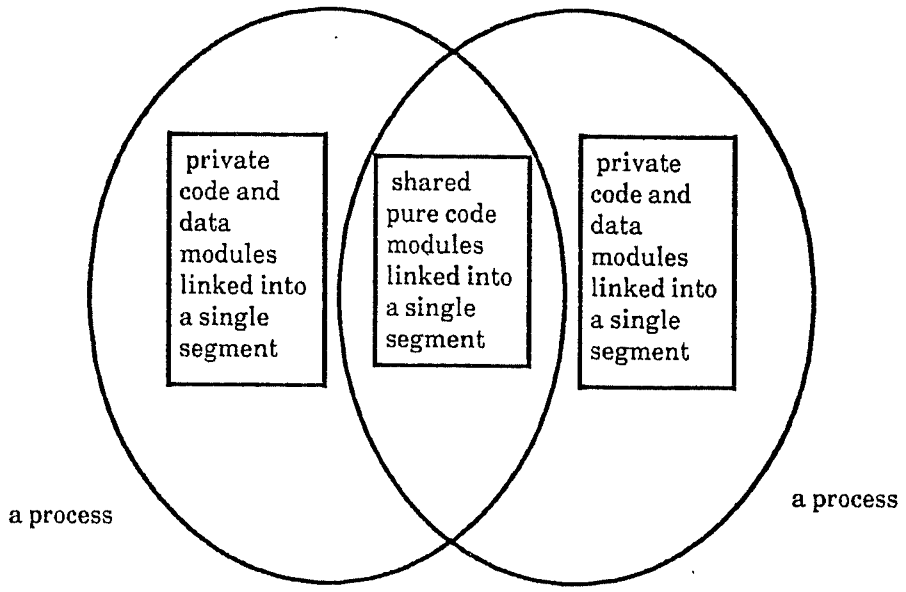
private code and data modules linked into a single segment
shared pure code modules linked into a single segment
private code and data modules linked into a single segment
a process
a process

fig.7 two segments per process

Early systems, for example, DEC's KA-10, would typically provide two segments per process, one shared, for compilers, editors etc., one private [fig.7] and the resident operating system would run in a separate address space. Operating system services were invoked by supervisor call. Subsequently, systems providing large numbers of segments per process were developed [fig.8]. The logical structure of a program may be retained at run time by using code, data and stack segments where appropriate. Logically distinct entities, the segments, are given separate protection. The idea of providing much of the operating system as segments shared by the user processes, invoked by an in-process procedure call seemed attractive. Examples of this style of system are Multics[Daley68], MU5[Morris68] and ICL 2900[Izatt80]. Various solutions to how the shared segments should be named by the sharing processes were employed [Fabry74]. The problem of introducing new shared segments dynamically was not completely solved.

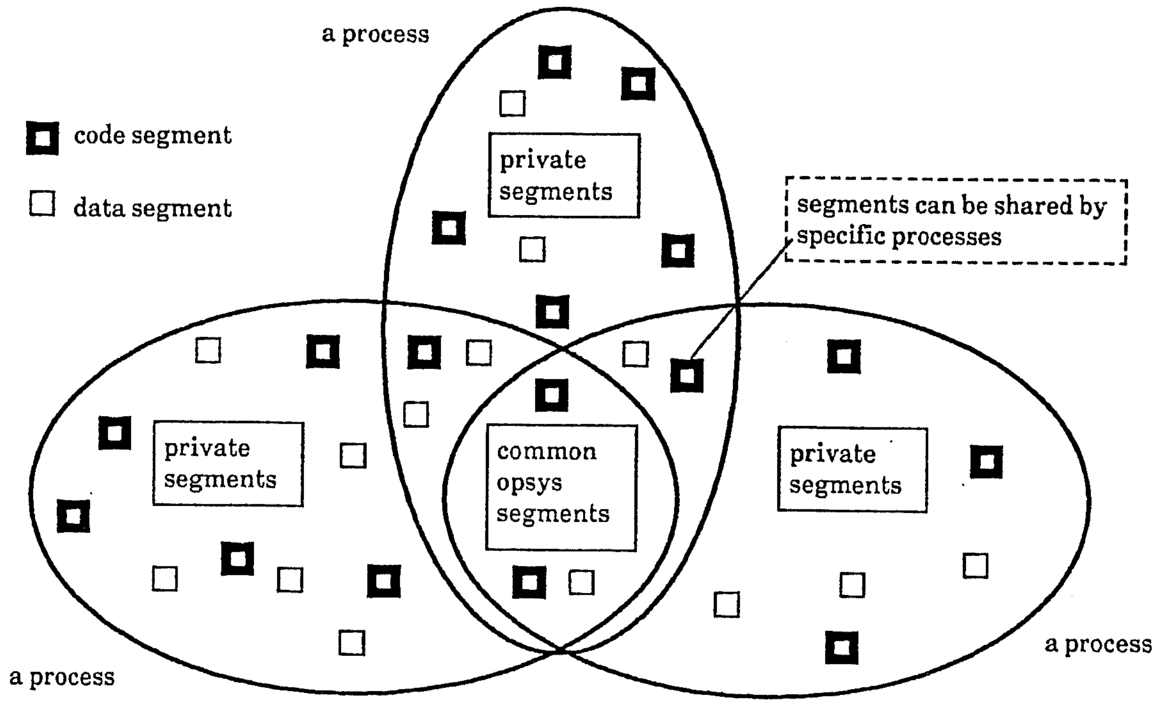

fig.8 General Segmentation

If much of the operating system runs in-process, the need for strict access protection is paramount. Sensitive system data and the code that accesses it are now in the address space of all processes. It is necessary for calls to and returns from system procedures to be validated and for entry points to be strictly enforced. Protection schemes based on rings (nested protection domains) were superimposed on the structure [fig.9] to support this. Operating system segments are allocated to inner, privileged rings and have free access to segments of that process in outer rings [Multics, ICL2900]. Control can be exercised over inward calls. The degenerate case of this is the two state machine where the operating system runs in privileged state and all other code in unprivileged state. A system call to an operating system procedure causes a state change.

It was realised that the concept of "the process" having access rights to segments is inadequate. Rather, it is the processes' invocation of a procedure segment that should have access rights to a given data segment. Domain architectures therefore support the notion of protected subsystems both within and external to the operating system. The model, towards which the structure based on rings of segments is moving, is of

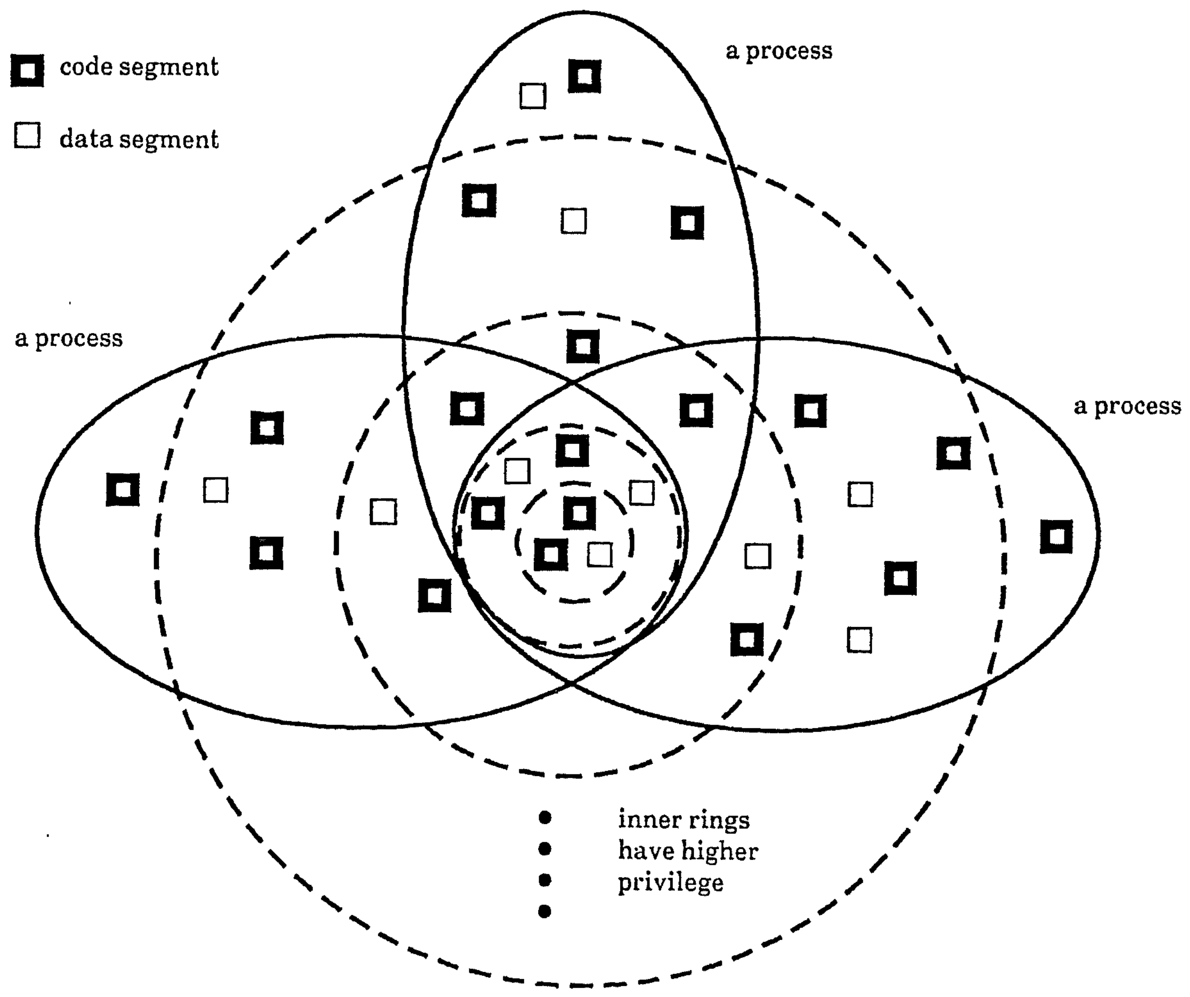

fig.9 general segmentation with nested protection domains

free-standing protection domains comprising packages of data segments and the code segments through which they are accessed. This is comparable with the object orientated approach to system design described in section 3. Such domains might be provided within the context of the overall address space of each process [fig 10], as in CAP1, or the protection domains might be seen as inhabiting a system wide address space [fig.11], shared by all processes, as in CAP3 [Wilkes79]. System structures of this type, enforced by segmentation hardware at run time are, in theory, akin to those supported by software at compile time in modular languages. In practice, the granularity of domains supported by hardware tends to be much coarser since protection architectures incur high performance penalties due to domain switching overhead. The Intel iAPX432 aimed to provide object orientated protection and addressing at the granularity of language level objects and Ada was chosen as the system implementation language. The performance penalty was too great for the majority of its potential users and the system failed.

Systems in which a very high level of protection is required are unlikely to be so concerned with sharing. Also, the move towards single user systems, which are often based on inexpensive hardware with simple or non-existant memory management,

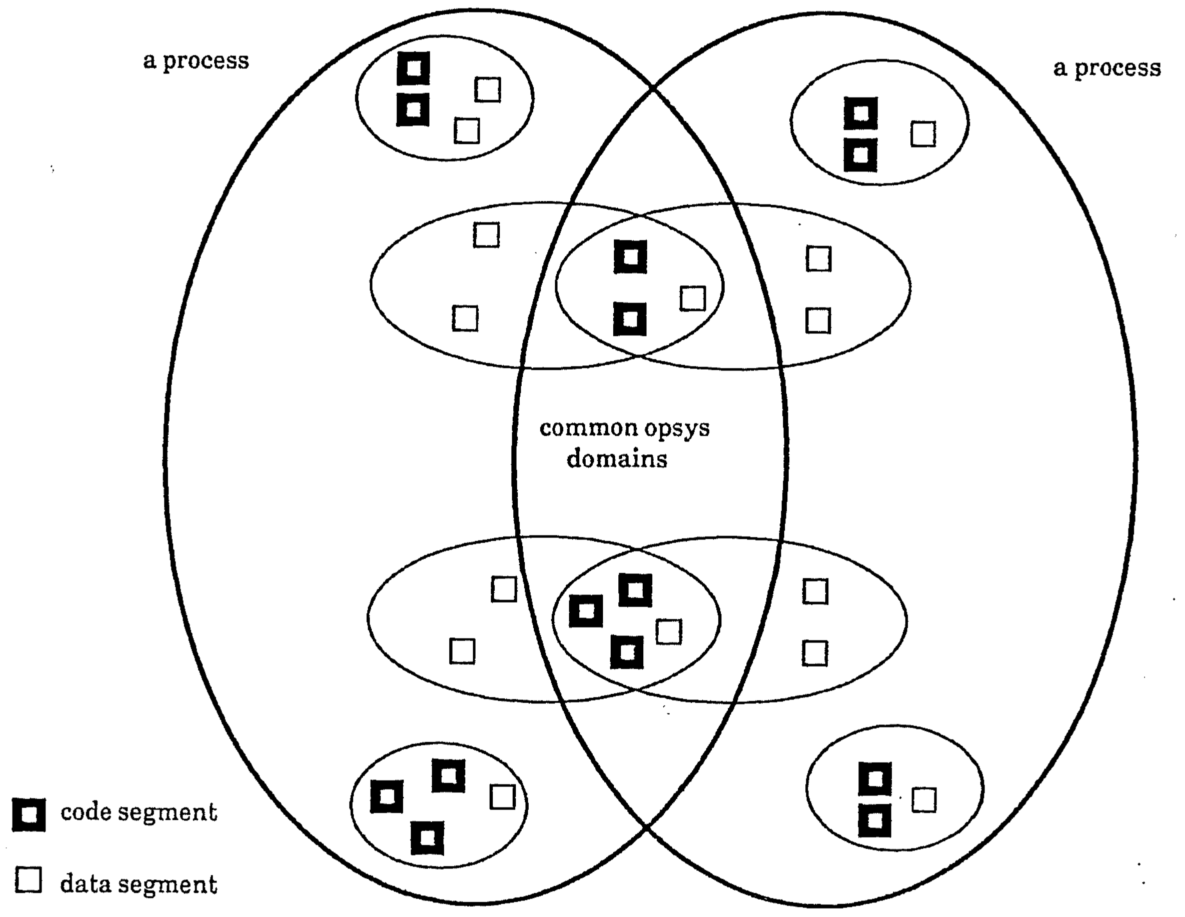

fig.10 multiple protection domains per process

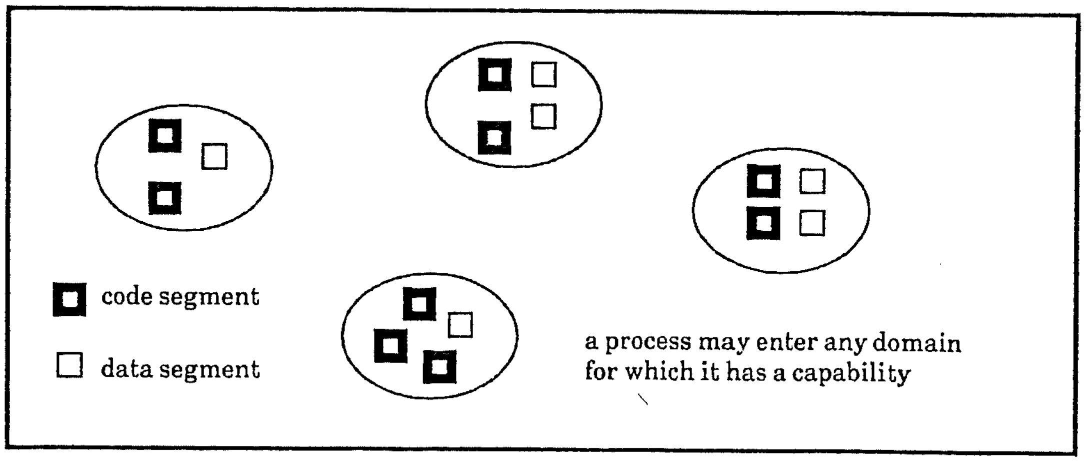
code segment
data segment
a process may enter any domain for which it has a capability

fig.11 general domain structured system

made protection concerns less pressing and changed the emphasis of system design at least temporarily.

5.2 Mapped or Separated File Input and Output

In most systems the file store is considered separate from the backing store used to implement virtual memory and the programmer has explicit file input and output operations. In a segmented system the concepts of segment and file may be integrated and a file may become a segment when opened. The blocks of the file may become pages of the associated segment, paged into main memory directly from the file store on demand. This approach was used in the Multics system [Daley 68] where an intermediate level of storage, a high speed drum, was also employed.

Integrated file I/O gives the potential for simplification within the operating system. Also, the same copy of a segment is easily shared. In a separated system a file is typically copied to the backing store on open and a cataloguing system is required if that copy is to be shared. In a mapped system there is uncertainty as to when information is actually written out, and therefore secure, and a specific instruction is usually provided to achieve this.

It is possible for file I/O to be mapped in a system with paging rather than segmentation hardware, as in the early versions of Pilot [Redell 80]. A file must be mapped explicitly into the available linear address space.

The programming language model of file I/O tends to assume a separated system and the elegance and economy of an integrated system has not in general been appreciated. There is potentially even more integration if typed program objects may be made persistent and mapped into the program's address space on activation, see section 10.

6. Single User Systems

Personal computers and workstations serve a single user. Workstations typically have powerful processors, a graphical interface based on a bit map display, possibly a local hard disc and a network connection. Many workstation operating systems were designed as part of distributed systems, which are discussed separately below. For multi-user systems, protection requirements are paramount. If the operating system of a single user system is corrupted the machine may be reloaded and restarted.

In multi-user systems, the high level interface of the closed system was enforced to avoid interference resulting from uncontrolled use of system resources. In a single user system, there is no reason why a local device should not be programmed at a low level by the user. A high level interface may be offered as a convenient service but need not be enforced. This philosophy leads to the design of **open operating systems**, [fig.12,13] [Lampson79]. A commonly used structure is based on the **process model.** A minimal kernel provides processes, inter-process communication and memory management. Other operating system services are provided as modules above the kernel and can be used, bypassed or replaced as the user wishes. It is unfortunate that ISO also chose to use the term "open system" for a system of unspecified structure which is open to communication.

It may be useful to impose a partial ordering on operating system services if only to avoid circular dependencies. Even within open systems designs, for example, a file directory (text naming) service is at a higher level than a file storage service. A large number of layers and mandatory calling through every layer causes high overheads

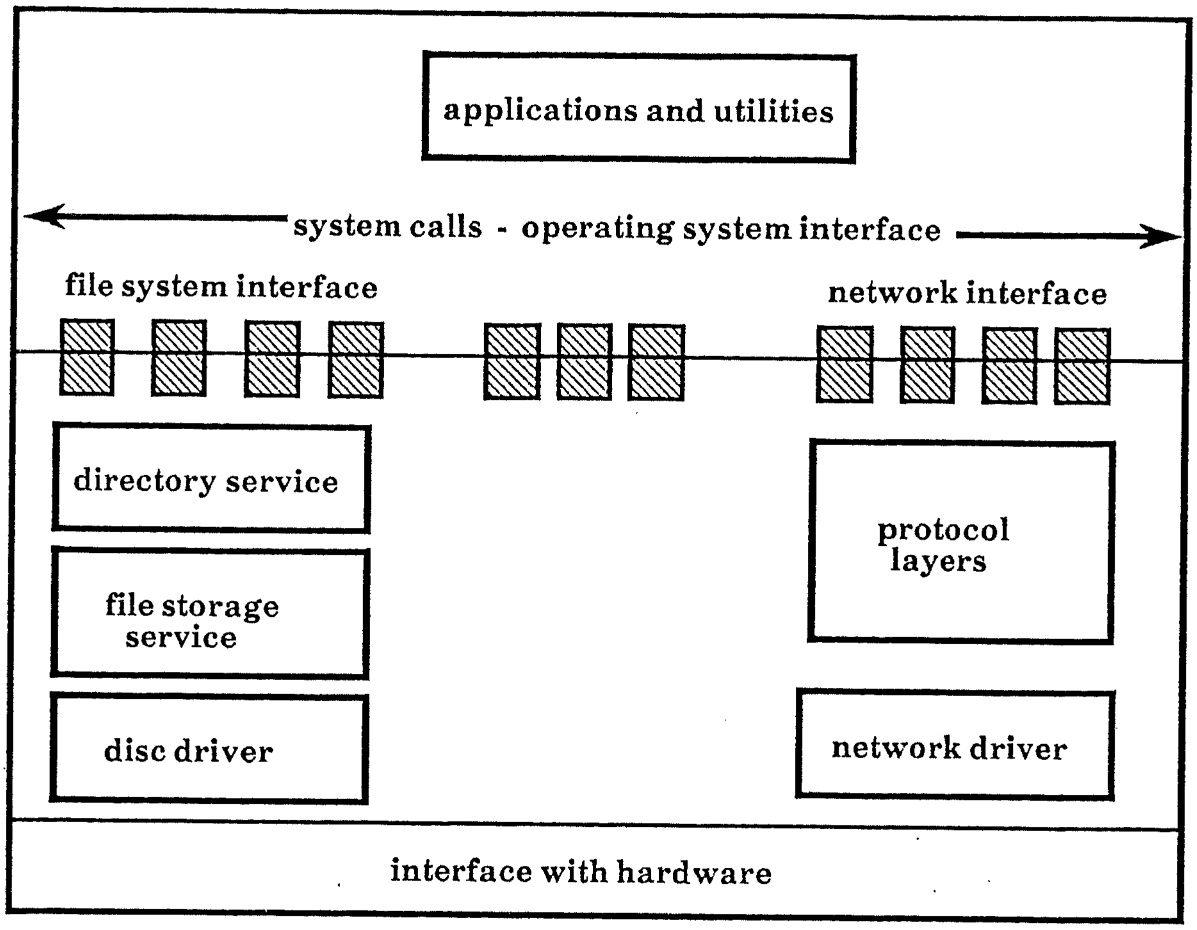

fig. 12 A Closed System Structure: selected modules

and should be avoided. Even in THE in which a strict hierarchical ordering was imposed, see section 3, a process at level N could call inwards for service, bypassing levels as appropriate.

Open operating systems are often designed as single language systems with a common address space for all system and user processes. They depend on language restrictions being enforced by the compiler, or by programming conventions, for protection against error. For such a system, memory management hardware is not essential and if a large virtual address space is required, a simple demand paging system is sufficient as protection and sharing are not important.

An example of an operating system developed specifically for personal computers is Tripos [Richards79]. The system is open and structured as a set of processes, each with a unique static priority. The multifunction nature of some system processes is captured in an internal coroutine structure. Asynchronous message passing is used for inter-process and device-process communication. All processes share a single address space, however, and messages, which may vary in length, need not be copied but can be read in place by the recipient. This elegance of implementation is a consequence of a complete disregard of protection issues, only justified in a single user system.

Tripos was written in BCPL, a typeless language, and was designed to be portable. If a strongly typed language is used, preferably with separate compilation facilities for modules or objects as described in section 3, a degree of protection is provided. Writing

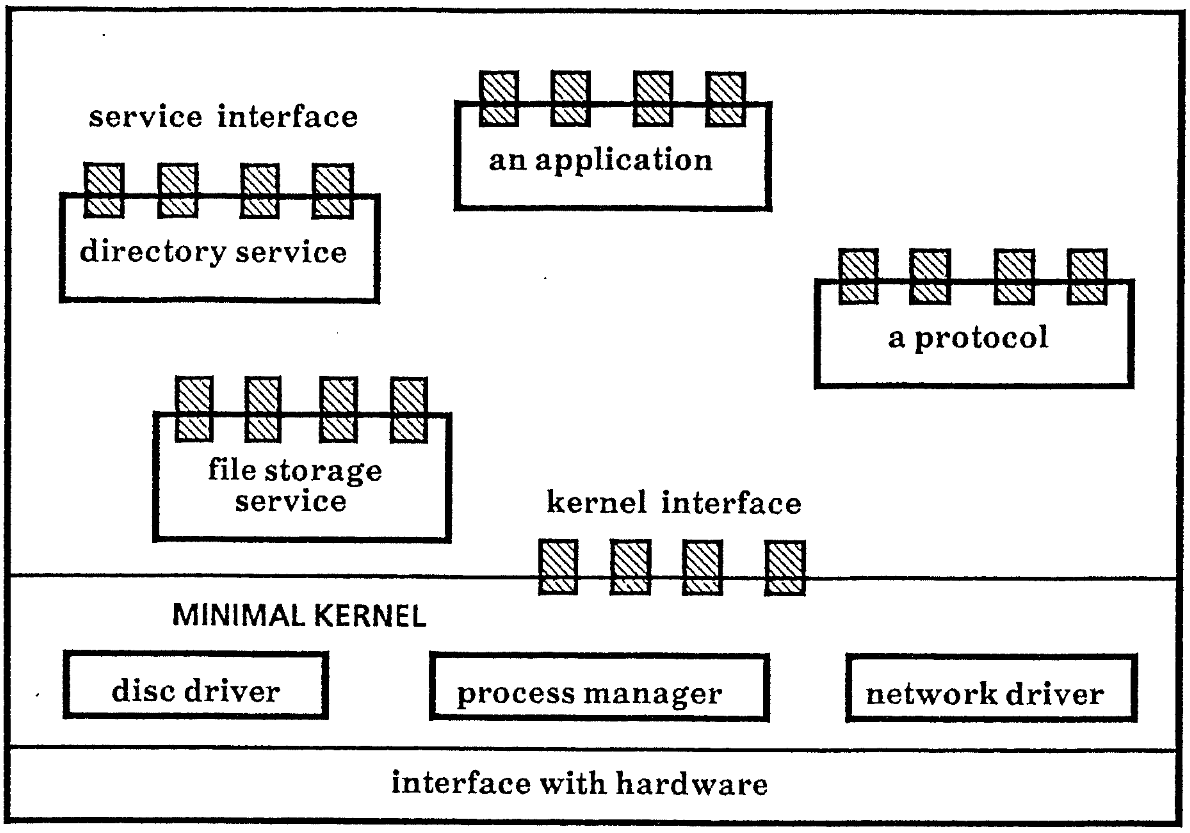

fig. 13 An Open System Structure: selected modules

in a high level language enforces structuring in that modules are accessed only through their interfaces. Compile time type checking ensures that interface specifications are adhered to, although protection enforced by software at compile time is not proof against all run-time errors [Swinehart86].

Examples of workstation operating systems with an open structure, developed as part of distributed operating systems, are Pilot [Redell80], developed for the Xerox range of workstations with Mesa as preferred language, Accent [Rashid81] and Mach[Jones86], developed at Carnegie-Mellon University, Apollo [Leach83], V [Cheriton84], Mayflower[Hamilton84] with Concurrent CLU as preferred language, and Amoeba [Tanenbaum85].

Many workstation operating systems are Unix based and it is of interest to examine the structure of Unix (section 7), but, more important, the structures that Unix is constrained to support when systems are implemented above it (section 8). A detailed criticism is given in [Blair85].

7. Unix Structure Outline

Unix has become very widely used and its design features are well known [Quarterman85] and will not be described in great detail here. It provides a good program development environment, compared with many general purpose operating systems, and has a wide range of software tools and flexible facilities for their composition. In this section its features are outlined, and their consequences pointed out, and in section 8 the constraints imposed on those attempting to implement system software above Unix are emphasised.

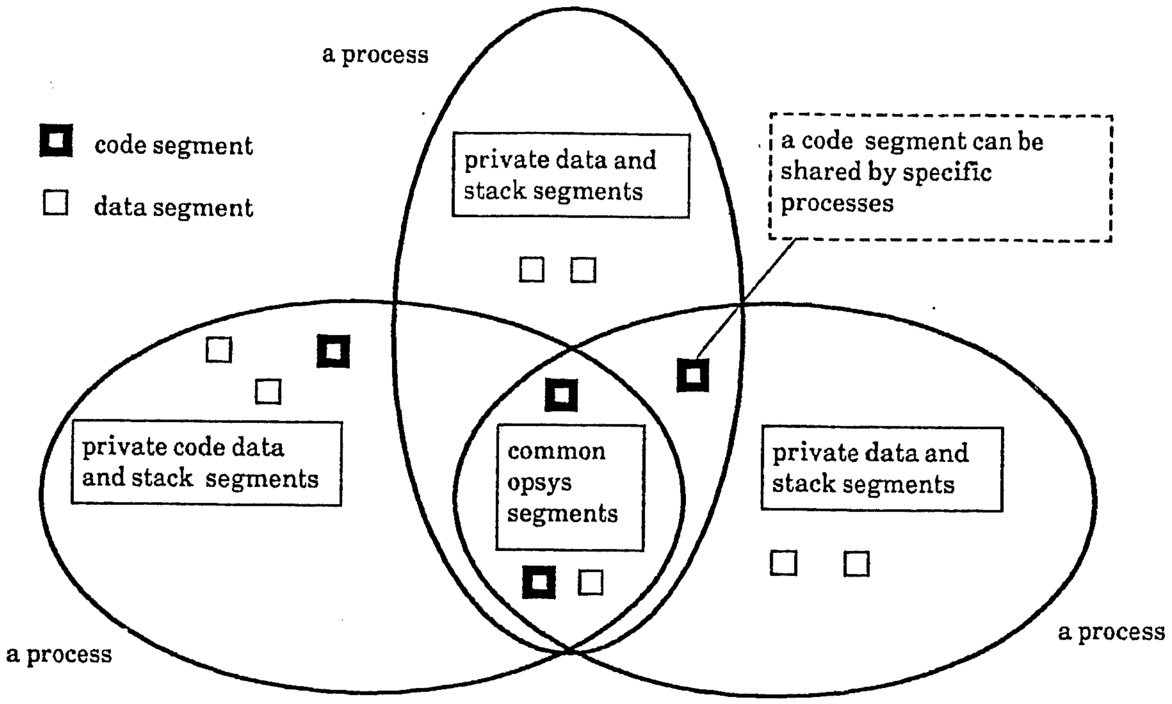
a process
▣ code segment
□ data segment
private data and stack segments
a code segment can be shared by specific processes
private code data and stack segments
common opsys segments
private data and stack segments
a process
a process

fig. 14 Unix structure

Unix was implemented in C, with a small amount of assembler, initially for the PDP 11 range. Sources were freely available and at this time the system was small enough to be comprehended in its entirety. The design philosophy was that algorithms should be simple and intelligible rather than being directed towards high performance. Its design was a reaction against the size and complexity of the large timesharing systems being developed at the time. Unfortunately, recent releases, which address some of the problems described below, are approaching the size of the systems its original designers deplored.

Compatibility of file, device and inter-process I/O, achieved through a unified naming scheme, is attractive to users because of the flexibility it provides. Its consequence is a very limited inter-process communication facility which consists of a synchronisation facility added to a one way, one-to-one byte stream [fig.15].

Unix is a closed system. Its "kernel" includes process, memory, device and file management. The interface to the file management service is at the directory service level. System calls are blocking (synchronous) since when user process n makes a system call, it becomes system process n and executes the kernel [fig.14]. Should an interrupt occur when such a system process is running, the interrupt service routine is executed (in the context of the interrupted process) and control always returns to the interrupted process. Only when this process executes a wait primitive may some other process, made runnable by the interrupt, be considered for scheduling, ie. non-preemptive scheduling is used for processes executing the kernel.

Unix processes are extremely heavyweight, having a great deal of associated state. Each process runs in a separate address space, part of which is occupied by the Unix

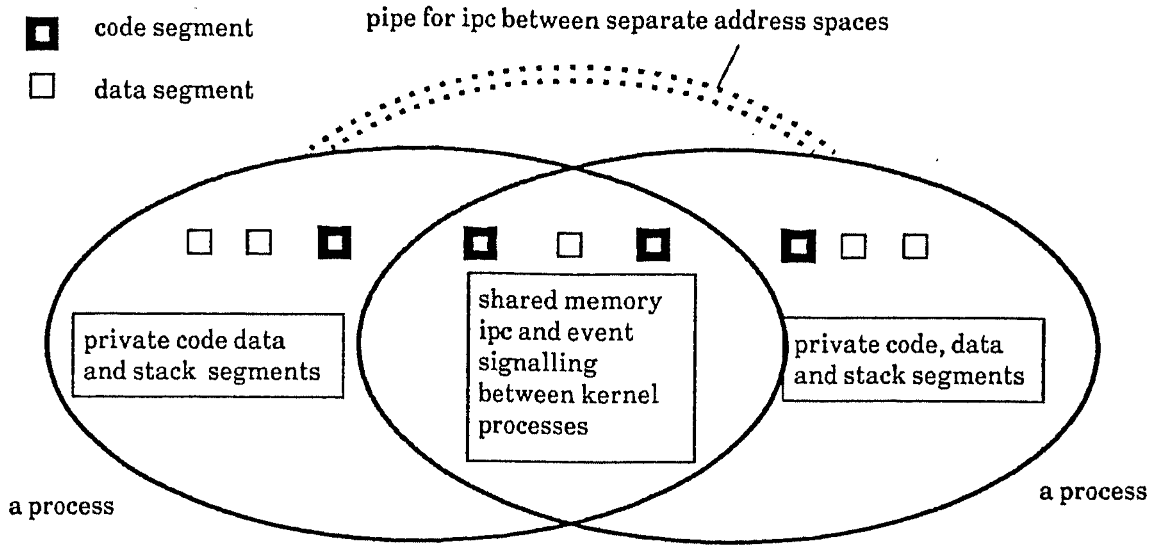
code segment
data segment
pipe for ipc between separate address spaces
private code data and stack segments
shared memory ipc and event signalling between kernel processes
private code, data and stack segments
a process
a process

fig. 15 Unix IPC

kernel, and, although code may be shared, data may not. Dynamic process creation is provided by the **fork** primitive but the overhead is high. A new address space is first created containing a replica of the parent. If a new software system is required by the child it must then execute the **exec** primitive which causes the newly created replica to be overlayed. Any subsequent interaction between such related processes is expensive, because of context switching, and is limited to an exit, with status, by the child for which the parent waits or use of one or more pipes. Pipes (like open files) are passed from parent to child on creation and pipes may therefore only be used between processes with a common ancestor.

Device buffers are implemented as a cache and there is no guarantee of when or in what order they are written to disc. In spite of this resemblance to a mapped system, I/O is in fact provided in a conventional manner with a separated file store.

Scheduling and swapping algorithms, designed to be simple and intelligible in accordance with the general Unix philosophy, can neglect a process for seconds under some circumstances. This makes the system unusable for real time work.

The "set user id" facility supports protected subsystems and programmable access controls. Each named object has nine conventional access control bits, for owner,etc., associated with it but also an additional set uid bit. If this bit is set by the owner, any other user who executes the file as a program inherits the access rights of the owner, by effectively assuming the identity of the owner, for the duration of the program run. Both the inherited "effective id" and the real user id are available through system calls thus allowing access control to be programmed on an individual basis. The earlier Titan file system [Fraser 69] had also allowed access rights to files to be associated with which program was being run rather than which user was running it.

Unix is not ideal as a single-user system because of its closed design and heavyweight process structure. However, for the majority of users, the quality of the program development environment it provides usually outweighs such structuring considerations. Nor is it ideal for a large, multi-user system because of its neglect of performance. Because of a status approaching that of a de facto standard, several projects have provided the Unix system call interface but have reimplemented the kernel, for example Locus [Popek81], Mach [Jones 86] and Topaz for the DEC Firefly. This may improve performance but the designer of systems which must be implemented above Unix still has to live with heavyweight processes and byte orientated inter-process communication.

In summary, although Unix is adequate as a program development environment, it has severe disadvantages as an implementation base for high performance software.

8. Distributed Systems

The basic model of a system comprising a set of **services** which may be invoked by **clients** is often used and may be viewed as an implementation of a more general system architecture based on objects. A general discussion of how the software structures and associated inter-process communication facilities introduced in section 2 above may be extended for a distributed environment is given in [Bacon81]. An overview of a number of distributed operating systems is given in [Tanenbaum 85].

In the absence of a modular structure, the functions of an operating system cannot be distributed, and the only option is to interwork complete systems. This approach has been used for traditional computer networks of mainframes and for distributed systems based on Unix since it has a closed, unstructured kernel. A degree of integration, for example, providing a single global file store, has sometimes been provided [Popek81]. This model of operating systems also appears to be implicitly assumed in the ISO "open systems" standardisation.

Workstation based distributed systems typically provide shared public services, such as printing, file, mail and registration services. Design criteria for the workstation operating systems are as described in section 6 above and requirements for implementing servers are discussed below. Most projects have adopted an **open operating system structure.** A kernel provides functions required at every node such as process, memory, local device and network driving, and other operating system functions are included as required. Several projects, for example V, Amoeba, Accent, Pilot and Mayflower have emphasised their minimal kernel.

Some researchers into distributed processing in a workstation model environment have attempted to use spare capacity or unused workstations through "worm" mechanisms [Schoch82,Dannenberg85]. The disadvantage is that each workstation has an owner who is very likely to resent this intrusion. The overhead in the application is also considerable.

Distributed systems based on the pool of processors model [Needham82] provide the user with a terminal, or as costs decrease more likely a graphics terminal or workstation. If only a terminal is provided the user typically acquires a computer from the pool. The approach allows a share of a machine or several machines to be acquired by the same mechanism and therefore potentially supports distributed concurrent application programming. A workstation per user allows functions such as editing, running mail programs or control of debugging to be located in the workstation and the pool to be used to provide processing engines.

An advantage of the processor bank approach is that any software system may potentially run as a subsystem on processor bank machines. The subsystems may be independently managed and have their own authentication, access controls, naming schemes etc. It is necessary for the underlying system to ensure protection between the autonomous subsystems and to facilitate sharing of common services such as file storage and printing. In the Cambridge Distributed Computing System (CDCS), for example, each subsystem must register its current users with an authorisation server, the active name table manager (ANT). ANT issues a session key which acts as a capability for use of any common service.

8.1 Multi-threaded Servers

Servers tend to be heavily used and care must be taken to prevent them becoming system bottlenecks. Distributed systems projects quickly discover the need to provide concurrency in servers [Bacon81,Clark85]. A classic example is the dedicated file server that takes a request from the network, processes it, finds the disc must be accessed, and requests service from the disc handler. At this point it must be possible to start work on a new request.

If a message-based model is used, an interface process takes requests and assigns them to worker processes [Gentleman81]. The problem of the workers' access to shared data must be solved. In a procedural system, server code is conceptually executed by all clients. Since the procedure calls are in fact from remote nodes, local processes are assigned to make the call on behalf of the remote clients. As described in section 3, efficient implementation requires lightweight processes or **threads** executing the service code within a shared address space, accessing shared data under interlock. Language level support for communication between distributed components has been provided by several projects. Conic supports message passing, of typed messages via typed ports [Sloman84], but a remote procedure call facility is becoming increasingly widely accepted [Birrell84,Hamilton84].

Projects that have attempted to implement over Unix have reported problems in implementing servers [Black85, Satyanarayanan85]. If a Unix process is forked for each client request, data cannot be shared in memory and schemes involving pipes or files have to be devised. A great deal of context switching overhead is involved. If a single Unix process is used for a server the implementor must provide coroutines within that Unix process and must solve the problem of, or live with, blocking system calls.

9. Heterogeneity

Even if a system is initially based on homogeneous hardware and software, it is inevitable that, as technology evolves, integration of new computers and connection media will be required. A local system will typically be based on interconnected LANs on which a number of systems reside and there will be a (human) management hierarchy with a single jurisdictive authority. In the more general case, independent subsystems and autonomous management domains must be accommodated.

In the local case in particular, it is important that performance should not be sacrificed in the more frequent case of intra system working to provide infrequently required inter system working. A simple example of a requirement for inter system working is when a host system with good software tools is used to develope software for a target environment designed for high performance. In this case, cross compilers and linkers ensure that data representation is managed between host and target systems. Transfer from the host to target domain may be effected by down line loading from the host domain, by file transfer between file services in the host and target domains or by use of a common service by both host and target domains.

A file transfer program that knows about differences in file naming and file storage conventions provides minimal support for interworking between systems. If a common file storage service is used, at least for those files that are used in more than one system, transfer between systems is avoided but problems of incompatible data representations and naming conventions have still, in general, to be solved.

The provision of common services such as file storage, printing, mail, remote computation etc. is an elegant and extensible way to support heterogeneity. The CDCS contains the elements of this approach, as described in section 8. If the processor bank approach is used, it is not only possible to incorporate new and special purpose hardware but also, separate software systems may coexist above the basic infrastructure. Heterogeneity is accommodated provided the separate systems adopt certain conventions when they come to use common system services. The "universal" file storage service [Birrell 80] is such that any number of client operating systems with different file naming schemes and access control policies may make use of it. A universal file directory service is a possible extension [Seaborne 87].

If common services are to be invoked, infrastructure is required to support service naming (each service may name the objects it manages independently) and authorisation for service use, as well as communication with the service. The underlying system may be written in a single implementation language with an integrated communication mechanism, such as remote procedure call, as a basis for building application protocols. A system kernel will provide primitives suitable for constructing efficient services, see section 8.1 above. As noted above, many distributed system kernels, although described as minimal, contain a substantial amount of protocol handling, device driving etc. Although written in high level languages, some machine dependent parts may require assembler and if a variety of hardware is used for the nodes and connection media, the kernel must be implemented for each type. Each time a new device or protocol is required, the kernel must be rebuilt to include the new facility. A major design aim is to minimise the kernel to achieve portability and extensibility without sacrificing efficiency.

More generally, it is desirable for programs written in a number of different languages to interwork. Mayflower (Concurrent CLU) RPC, for example, was extended to interwork both with programs written in Mesa and employing Xerox Courier as external data representation and with programs using Sun's XDR. A number of transport protocols may also be selected [Bacon 87]. A similar approach, allowing selection from multiple standards, has been used at the University of Washington in the Heterogeneous Computer Systems project [Black 86].

The general aim is to retain a language level, type safe communication facility within a heterogeneous system. This avoids the overhead and limitations associated with the definition of a transfer syntax for a limited number of system types which are explicitly tagged for run time type checking as in the ISO presentation layer standard, ASN.1 (Abstract Syntax Notation 1) [ISO 85]. The Mercury project at MIT is addressing this area [Liskov 87]. A general discussion of approaches to heterogeneity in distributed systems is given in [Notkin 86].

As distributed systems evolve and are interconnected, possibly over wide areas, the requirement for heterogeneous hardware and software components to interwork becomes increasingly important. The ISO "Open Systems Interconnection" reference model and associated set of standards address the requirement for **open communication**. Development of a framework and standards for **open distributed processing** has more recently been initiated [ECMA86, ANSA87].

In outline, the lower ISO layers are concerned with provision of data transportation services and protocols and the upper layers, residing in the end systems, with augmenting the transport service to support distributed applications. The original connection orientated emphasis has been extended to include more lightweight connectionless services and protocols which have become feasible with modern connection media and desirable for many applications. Several proposals have been made for protocols based on the remote operation concept, which are simpler than the ISO session and presentation standards. Remote procedure call is one such protocol.

If heterogeneous components are to interwork, infrastructure must be provided. A client server model again provides a useful framework. An open distributed processing system may be described in terms of a kernel which supports services and their invocation. The purpose of the system is to facilitate the remote use of application services and a basic set of infrastructure services are necessary for distributed processing to take place. Management services are also envisaged to allow managers, within their autonomous local systems, to monitor and tune performance and exercise control, as are general services to aid the user.

In outline, infrastructure services are required:

- to allow specification and **registration** of service interfaces by service providers
- for **authorisation** and **accounting** for service use
- for **authentication** of principals invoking services
- to allow potential clients to locate and invoke services
- to assist **configuration** of services

10. Object-Orientated Systems

The object model was introduced in section 4 as a design method for decomposing systems into modules. Languages with an object orientated style have been found appropriate for managing the complexity of the large programs required to implement each system module. The notion of automatically programming an entire system, whether centralised or distributed, by regarding its components as objects is attractive [Black85, Black86, ECMA86, Jones86].

The system components described as services in section 9 may be regarded as active objects. The communications service, for example, may be regarded as one such object. In a large distributed system a conceptually centralised service may be implemented as a set of distributed servers. A server is an active object which is an instance of some class of the abstract type of the associated service. Alternative implementations of a service for heterogeneous hardware are modelled by multiple classes of the abstract service. A service may be configured by inheriting standard interfaces for monitoring, control, accounting etc. in addition to the client interface [ANSA 87]. Any object may adopt the role of client by invoking an operation on some other object. Current projects are Clouds[LeBlanc85], Distributed Smalltalk [Decouchant86], Somiw [Shapiro86] and Comandos [Horn 87].

Although languages which provide varying degrees of support for objects are now widely available, the system services may not be designed explicitly to support objects. The storage service, for example, is likely to support only the file abstraction and the directory service allows textnames and access controls to be associated only with files. When a program writes its typed data to a file, all typing information is lost. Many software tools work in terms of typed objects, for example, the program state gathered by a debugger at a breakpoint or the nodes of a parse tree constructed by a compiler. If such tools require the protection afforded by persistence of type information they must themselves use a method to flatten their typed objects before preserving them in a file. At a higher, system level, directories, bank accounts, text files etc may be regarded as typed objects, as may any service. A current design issue is therefore to what extent the system infrastructure should support objects. Object stores have been built but performance has been unacceptable, for example Swallow [Svobodova84]. Research is still in progress in this area at both the language [Atkinson84,87] and system [Crawley86] levels.

11. Formal Methods

A long-term goal of the application of formal methods to systems engineering is the translation of an unambiguous specification into a correct implementation. Such a facility would allow rapid prototyping and consequent refinement of the specification and would support system maintenance and evolution, in that all the implications of a given modification could automatically be transmitted throughout the system. If the automatically derived implementation is too inefficient for practical use, efficiency transformations may be automatically applied.

At present, specification languages provide a precise notation to express the functionality of system components but few automatic techniques exist, even for consistency checking within a single module. The importance, as an aid to human communication, of an unambiguous and concise notation for system design, should not however be underestimated.

Current design methods range from systematic methods, often developed as a company style and subsequently publicised [Jackson84], through semi-formal notations, developed for some specific design exercise [Birrell86], to mathematically based specification languages [Jones84,Hayes87]. State based specification is appropriate for capturing the functionality of modules, services, object classes etc. It is insufficient in itself to capture dynamic system behaviour. Protocol specification is also required [Billington87]and, more generally, the behaviour of concurrent systems which is the focus of [Hoare85,Milner80].

When the long term goals are realised, human insight will still be necessary for designing systems, but formal tools should make many time consuming and tedious activities unnecessary.

12. Summary and Conclusions

Some simple basic principles that have become established for structuring both centralised and distributed systems are:

- the **object model** to aid modular decomposition
- an **open** modular structure with partial hierarchical ordering but without mandatory layering
- the provision of **mechanisms** via which a range of **policies** may be implemented
- a **minimal kernel** to aid portability and extensibility
- use of whatever **concurrency tools** are appropriate or available for internal implementation of each object.

More specific guidelines are beyond the scope of this paper. A more detailed discussion is given in [Lampson83].

Research in progress is addressing:

- an object orientated approach to programming in the (very) large, including specification, configuration and generation of system components
- to what extent the system infrastructure should support objects and how this might be achieved with acceptable performance
- support for heterogeneity within and between systems
- tools for specification and implementation based on formal methods

Continued developments in technology make it necessary to review research directions critically and frequently. The human effort involved in systems development is so great that every attempt should be made to avoid misdirecting it.

Acknowledgements

Although the material presented here represents my personal view I should like to acknowledge helpful discussions with Roger Needham and colleagues in the Systems Group at the Computer Laboratory Cambridge.

References

[Andrews83]Andrews G and Schneider F B "Concepts and Notations for Concurrent Programming" ACM Computing Surveys, 15(1), 3-43, Mar 83

[ANSA 87] ANSA Reference Manual version 00.03, Advanced Networked Systems Architecture, 24 Hills Road, Cambridge UK, CB2 1JP

[Atkinson84] Atkinson M P, "PS Algol Reference Manual" Edinburgh University Report PPR-4-83

[Atkinson87] Atkinson M P and Buneman O P, "Types and Persistence in Database Programming Languages", ACM Computing Surveys 19(2), 105-190, June 87

[Bacon81] Bacon J M, "An Approach to Distributed Software Systems" ACM SIGOPS OSR 15(4), 62-74, Oct 81, and: Distributed Computing: Concepts and Implementations,Editors McEntire et al, IEEE press 1984

[Bacon87] Bacon J M, "Distributed Computing with RPC: the Cambridge Approach." proc IFIP TC10/ WG10.3 conference on"Distributed Processing", Amsterdam Oct 87, North Holland 1988

[Billington87] Billington J, "PROTEAN: A Specification and Verification Aid for Communication Protocols" to be published in IEEE Trans SE, special issue on Computer Communications

[Black85] Black A, "Supporting Distributed Applications: Experience with Eden" ACM SOSP10, OSR 19(5), 181-193, Dec 85

[Black86] Black A et al, "Object Structure in the Emerald System" ACM SIGPLAN Notices, 21(11), 78-86, Nov 86

[Black 87] Black A et al, "Interconnecting Heterogeneous Computer Systems" University of Washington, Dept. of Computer Science, FR-35

[Blair85] Blair G S et al "A Critique of Unix" Software Practice and Experience 15(12), 1125-1139, Dec 85

[Birrell 80] Birrell A D and Needham R M "A Universal File Server" IEEE Trans SE, SE-6 (5), 450-453, Sept 80

[Birrell84] Birrell A D and Nelson B J, "Implementing Remote Procedure Call", ACM Transactions on Computer Systems 2(1), 39-59, Feb 84

[Birrell86] Birrell A D et al, "A Global Authentication Service Without Global Trust" Proc IEEE Conference on Security, 223-230, California, 1986

[Brinch Hansen70] Brinch Hansen P, "The Nucleus of a Multiprogramming Operating System"Comm ACM, 14(4), 238-250, April 70

[Brinch Hansen72] Brinch Hansen P, "Structured Multiprogramming" Comm ACM, 15(7), 574-578, July 72

[Brinch Hansen73] Brinch Hansen P "Operating Systems Principles" Prentice Hall 1973

[Cardelli85] Cardelli L and Wegner P, "On Understanding Types, Data Abstraction and Polymorphism" ACM Computing Surveys, 17(4), 471-522, Dec 85

[Cheriton84] Cheriton D R, "The V Kernel: A Software Base for Distributed Systems" IEEE Software, 1(2), April 84

[Clark85] Clark D, "The Structuring of Systems Using Upcalls" ACM SOSP10, OSR 19[5), 171-180, Dec 85

[Crawley86] Crawley SC, "An Object Based File System for Large Scale Applications" Software Engineering Environments, Somerville (ed), Peter Peregrinus Ltd 1986

[Daley68] Daley R C and Dennis J B "Virtual Memory, Processes and Sharing in Multics" Comm ACM, 11(5), 306-312, May 68

[Dannenberg85] Dannenberg R B and Hibbard P G, "A Butler Process for Resource Sharing on Spice Machines", ACM Trans Office Information Systems, 3(3), 234-252, July 85

[Decouchant86] Decouchant D, "Design of a Distributed Object Manager for the Smalltalk 80 System" ACM SIGPLAN Notices 21(11), 444-452, Nov 86

[Dijkstra68] Dijkstra E W et al "The Structure of THE Operating System" Comm ACM, 11(5), 341-346, May 68

[Dijkstra71] Dijkstra E W "Hierarchical Ordering of Sequential Processes" Acta Informatica, 1(2), 115-138, Feb 71

[ECMA86] ECMA TC32-TG2 Distributed Application Service Environment (DASE)

[Fabry74] Fabry R, "Capability Based Addressing" Comm ACM, 17(7), 403-412, July 74

[Fraser69] Fraser A "Integrity of a Mass Storage Filing System" Computer Journal 12(1), 1969

[Gentleman81]Gentleman W M, "Message Passing Between Sequential Processes, The Reply Primitive and the Administrator Concept", Software, Practice and Experience, 11(5), 435-466, May 81

[Goldberg80] Goldberg A and Robson D "Smalltalk-80: The Language and its Implementation" Addison-Wesley 1983

[Haberman76] Haberman A N et al "Modularisation and Hierarchy in a Family of Operating Systems" Comm ACM, 19(5), 266-272, May 76

[Hamilton70] Hamilton K G, "A Remote Procedure Call System" PhD thesis, Cambridge 1984,TR 70"

[Hayes87] Hayes I (editor), "Specification Case Studies", Prentice Hall 1987

[Hoare72] Hoare C A R, "Towards a Theory of Parallel Programming" in Hoare and Perrot (eds), Academic Press, 61-71, 1972

[Hoare74] Hoare C A R "Monitors: An Operating System Structuring Concept" Comm ACM, 17(10), 549-557, Oct 74

[Hoare85] Hoare C A R "Communicating Sequential Processes", Prentice Hall 1985

[Horn87] Horn C, "Conformance, Genericity, Inheritance and Enhancement" proc ECOOP, Paris, June 87

[ISO85] Specification of Basic Encoding Rules for Abstract Syntax Notation One, ASN.1 ISO/DIS 8825 June 85

[Izatt80] Izatt W T "Domain Architecture and the ICL 2900 Series" Software Practice and Experience, 10(4), 329-332, Apr 80

[Jackson83] Jackson M A, "System Development", Prentice Hall 1983

[Jones78] Jones A K, "The Object Model - A Conceptual Tool for Structuring Software" in Operating Systems - An Advanced Course, ed. Bayer R et al Springer Verlag, LNCS 60, 1978

[Jones84] Jones C B, "Software Development, A Rigorous Approach" Prentice Hall 1984

[Jones86] Jones M b and Rashid R F, "Mach and Matchmaker, Kernel and Language Support for Object-Orientated Distributed Systems", ACM SIGPLAN Notices, 21(11), 67-77, Nov 86

[Lampson83] Lampson B, "Hints for Computer System Design" ACM SOSP9, OSR 17(5), 33-48, Oct 834

[Lampson79] Lampson B W and Sproull R F "An Open Operating System for a Single User Machine" ACM SOSP7, 98-105, 1979

[Lauer78] Lauer H C and Needham R M, "On the Duality of Operating System Structures" ACM OSR 13(2), 3-19, April 79, also in: Proc 2nd Int Symp on Op Sys IRIA Oct 78

[Leach83] Leach P J et al "The Architecture of an Integrated Local Network" IEEE Journal on Selected Areas in Communication, SAC-1(5), 842-857, Nov 83

[LeBlanc85] LeBlanc R et al, "The Clouds Project" Georgia Institute of Technology TR85-0, Jan 85

[Levin75] Levin R et al "Policy Mechanism Separation in Hydra" Proc ACM SOSP5, 132-140, Nov 75

[Liskov87] Liskov B et al, "Communication in the Mercury System" to be published

[Milner80] Milner R, "A Calculus for Communicating Systems" Springer Verlag, LNCS 92, 1980

[Morris68] Morris D and Detlefsen G D, "A Virtual Processor for Real Time Operation" Comm ACM, 11(5), 17-28, May 68

[Needham82] Needham R M and Herbert A J "The Cambridge Distributed Computing System"Addison Wesley 1982

[Notkin 86] Notkin D et al, "Report on ACM SIGOPS Workshop on Accommodating Heterogeneity" ACM Operating Systems Review, 20(2), 9-24, April 86

[Popek81] Popek G et al "LOCUS: A Network Transparent, High Reliability, Distributed System" Proc ACM SOSP8, 169-177, Dec 81

[Quarterman85] Quarterman J S et al "4.2BSD and 4.3BSD as Examples of the UNIX System" ACM Computing Surveys, 17(4) 379-418, Dec 85

[Rashid81] Rashid R and Robertson G "Accent: A Communication Orientated Network Operating System Kernel" Proc ACM SOSP8, 64-75, Dec 81

[Redell80] Redell D D et al "Pilot: An Operating System for a Personal Computer" Comm ACM, 23(2), 81-92, Feb. 80

[Richards79]Richards M et al, "Tripos - A Portable, Real -time Operating System" Software, Practice and Experience, 9, 513-526, 1979

[Satyanarayanan85] Satyanarayanan M et al, "The ITC Distributed File System: Principles and Design" ACM SOSP10, OSR 19(5), 35-50, Dec 85

[Seaborne 87] Seaborne A F, "Filing in a Heterogeneous Network" University of Cambridge PhD thesis, 1988

[Shapiro86] Shapiro M, "Structure and Encapsulation in Distributed Systems: The Proxy Principle" Proc 6th IEEE International DCS Conference, Boston MA, May 86

[Schoch82] Schoch J F and Hupp J A "The "Worm" Programs - Early Experience with a Distributed Computation" Comm ACM. 25(3), 172-180, Mar 82

[Sloman84] Sloman M et al "Building Flexible Distributed Systems in Conic" in Duce D A (ed) Distributed Computing Systems Programme, Peter Peregrinus, Sept 84

[Svobodova84] Svobodova L "File Servers for Network Based Distributed Systems" ACM Computing Surveys, 16(4), 353-398, Dec 84

[Tanenbaum85] Tanenbaum A S and van-Renesse R, "Distributed Operating Systems" ACM Computing Surveys 17(4), 419-470, Dec 85

[Vyssotsky65] Vyssotsky V A, Corbato F J and Graham R M "Structure of the Multics Supervisor"AFIPS FJCC Vol 27, Part 1, 203-212, 1965

[Wilkes79] Wilkes M V and Needham R M, "The Cambridge CAP Computer and its Operating System" Elsevier/North Holland, Operating and Programming System Series, 1979

[Wulf75] Wulf W A et al "An Overview of the HYDRA Operating System Development" Proc ACM SOSP5, 122-131, Nov 75