Lecture Notes for

Systems Design

12 Lectures to Part 1A

Easter Term 1996

(djg@cl.cam.ac.uk)

University of Cambridge
Computer Laboratory
New Museums Site
Pembroke Street
Cambridge
CB2 3QG

Contents

Preface

This course, System Design, is a general introduction to the hardware and software architectural design of modern computer systems, including time-sharing workstations and mainframes, personal computers and imbedded microcontrollers. It is divided into three main parts:

1. HARDWARE
2. RESIDENT SOFTWARE
3. SOFTWARE TOOLS

but the material in each section is actually quite highly interdependent. Many of the ideas are repeated in more than one section.

For those of you who have done the hardware course already, you should see how the first parts of this course build on the larger blocks introduced in hardware, such as multiplexors and tri-state busses. For those who will do hardware next year, do not worry, no hardware knowledge is assumed or required. System Design course material is useful background for the part 1B courses on operating systems, concurrency, processor architecture, assembly language programming and compilation. The System Design lectures will follow these printed notes closely. These notes are not supposed to be a substitute for reading some of the recommended books.

These notes are available online on the departmental 'Teaching' pages, along with any extra slides that I might use in the lectures. Anyone in the department may print a copy for their own use, but copyright belongs to DJ Greaves.

Please email comments on these notes and the course to djg@cl.cam.ac.uk or fill out a lectures appraisal form, available from M Levitt or on the Web.

Acronyms used in these notes

ALU - Arithmetic logic unit

API - application program interface

ASCII - American standard code for information interchange

ATM - Asynchronous Transfer Mode

BIOS - The Basic Input Output System

CLI - command line interpreter

COF - common object format

CPU - Central Processor Unit

CRC - Cyclic redundancy check

DRAM - dynamic random access memory

IAR - instruction address register (same as PC)

IDE - a 16 bit disk drive IO bus

IO - input and output

LAN - local area network

MAC - media access control

MUX - multiplexor of two or more inputs

OS - operating system

PC - program counter

PCMCIA - Personal Computer Memory Card Interface Association

PROM - user programmable read only memory

RAM - random access memory

RISC - Reduced instruction set computer

ROM - read only memory

RS latch - a latch with set and reset inputs

RS232 - a serial communications standard

SCSI - small computer system interface

SIMD - Single instruction, multiple data processor

MIMD - Multiple instruction, multiple data processor

TLB - translation lookaside buffer

UART - universal asynchronous receiver and transmitter

VM - Virtual memory

Please send me email if I use acronyms which are not on this list.

Recommended Reading

These first three books start with hardware and work upwards through system architecture in the manner of this System Design course.

M H Lewin *Logic Design and Computer Organisation.* Addison Wesley

V C Hamachar *Computer Organisation.* McGraw Hill

R D Downing and F W D Woodhams - *Computers, from logic to architecture.*

Four books about system software:

Maddix and Morgan - *Systems software: An introduction to language processors and operating systems.* A general book.

A Tanenbaum - *Structured Computer Organisation.* A book from the preparatory reading list covering many basic topics in computer structure and software approaches.

L L Beck - *Systems software, an introduction to systems programming.* 2nd Edition. A practical book with a less academic approach than usual.

A Tanenbaum - *Operating system design and implementation.* Another of Tanenbaum's very readable books, this one going from operating systems basics through to at least part 1B level.

Two more advanced books:

Pereson and Silberschatz - *Operating systems concepts.* Addison Wesley, 1983. Another part 1B level book, but very readable when you are ready for it.

M J Bach - *The design of the Unix operating System.* This advanced book presents simplified implementations of part of the Unix operating system in the C language.

Part I

Hardware Structure of a Computer

1 Components of a simple computer

Figure 1.1 shows the architecture of a general purpose Von Neumann computer. Program and data share the same memory, which is connected to a processor unit using busses. The processor contains registers, including the program counter register.

Input and output devices are also connected to the bus. In this simple computer, the only I/O device is a universal asynchronous receiver and transmitter (UART). This provides access to the most basic of text devices, the *Teletype* serial terminal.

The idea of a teletype for input and output, with its two simplex communication channels (input and output) underlies much of systems software, even today. A PC computer running a terminal emulator such as *Procomm* is identical in function to the ancient Teletype.

With a bus, one device is always the *bus master*. In this simple computer, the only bus master is the processor, meaning that the processor always drives the address bus and control wires. The data bus is bi-directional, being driven by the processor during a write cycle and by the addressed *slave* device during a read cycle. We will now look at all of these components in greater detail.

1.1 Memory, Registers and Storage Elements

A broadside register of N bits is made out of N D-types with a commoned clock input. It can hold 2^N different values.

A register file of M locations by N bits requires M broadside registers of N bits each. The diagram shows a register file with one read and one write port. Log2 M bits of write address are required to select which port is written on the clock edge. A similar width address bus selects which register drives the output bus.

A register file with two read ports is often used in conjunction with an Arithmetic Logic Unit (ALU) to compose the *Arithmetic Unit* or *Execution Unit* of a central processor unit (CPU) or microprocessor.

1.1.1 RAM

Note: familiarity with and understanding of the memory timing cycle diagrams is not expected of those who have not done the hardware course.

A Static Random Access Memory (SRAM) of m words of size n bits each has a bidirectional n bit data port and an a-bit (a=log2(m)) address input port. The write enable and output enable control inputs are normally active low. The chip select input (also active low) must be asserted (ie logic zero) before the device will recognise a read or write cycle. The address input may be changed during a read, but should not be changed during a write cycle.

Note that the RAM architectural view is a long skinny rectangle of latches (m by n) but its implementation is a square piece of silicon of dimension $\sqrt{mn}$) latches.

RAM is made from RS latches not D-types. This is possible because the RAM device does not have a clock input and is not edge-triggered on any of its inputs. This saves half the transistors compared with a master-slave (clocked) memory, such as the earlier register file.

Memory size is often given in kilobytes (kB) or megabytes (MB). $\begin{array}{rcl} 1\text{ kB} & = & 1024\text{ bytes} \\ 1\text{ MB} & = & 1024\text{ kB} \end{array}$

1.1.2 ROM

The Read Only Memory (ROM) holds m words of size n bits. The word addressed on the a-bit address input port selects one of the $m = 2^a$ internal stored words and delivers it to the output port provided chip select and output enable are asserted (ie low since they are active low signals).

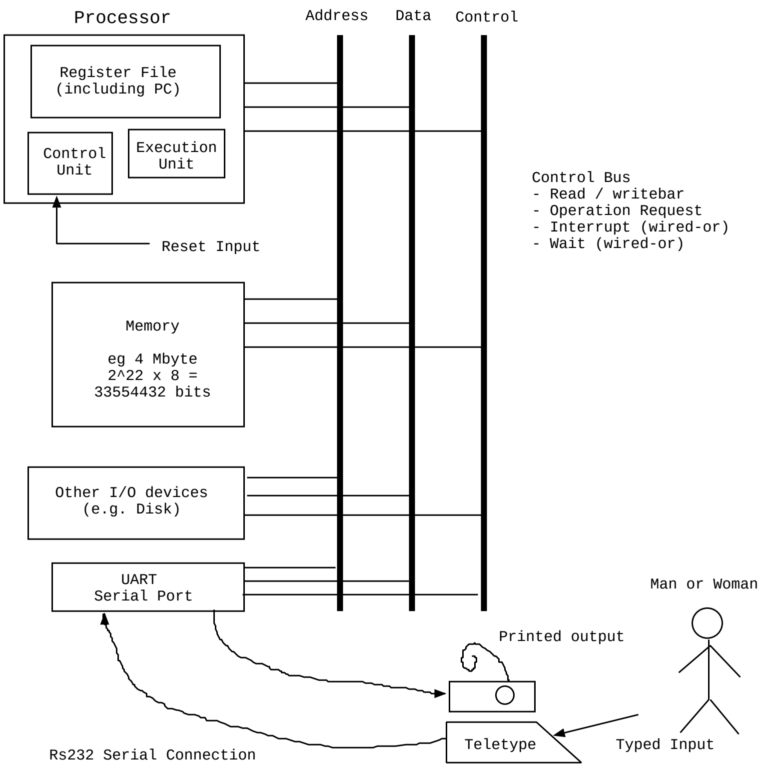

Figure 1.1: **Simple Von Neumann Computer Architecture**

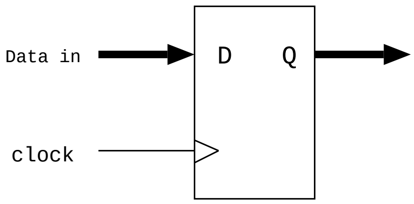

Figure 1.2: **Broadside Register**

The ROM is similar to the RAM, but has no write input.

ROMs are either programmed at manufacture using a customised mask or else electrically programmed in a programming machine which carefully injects static electricity into selected, otherwise insulated transistor gates - giving us a PROM. EPROMs are PROMs which can be erased - usually by exposure to ultraviolet light. They have a window in the top. EAPROMs (electrically erasable programmable read only memories) can be erased electrically and are used in PCMCIA memory cards.

1.1.3 Arithmetic Logic Unit

Figure 1.9 shows an N-bit arithmetic logic unit. This is a combinatorial device which yields an arithmetic or logical function of its inputs. With a four bit *function code* input, 16 logical functions can be done. These typically include

- A
- B
- A + B
- A + B + carry in

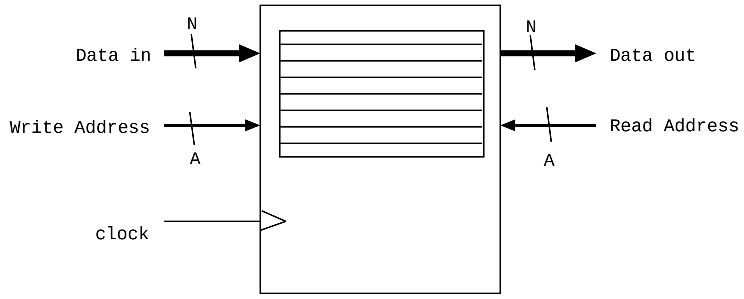

Figure 1.3: **Register File Symbol**

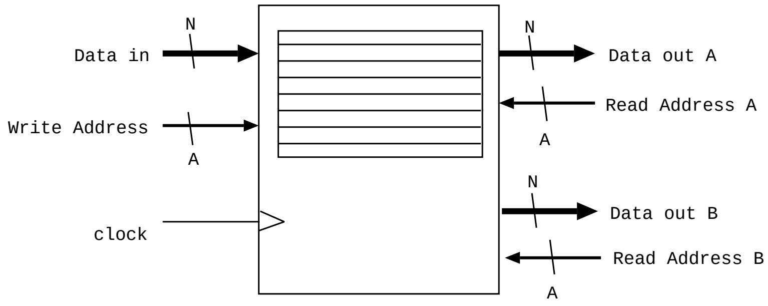

Figure 1.4: **Dual Read Ported Register File Symbol**

- A - B
- A bitwise-and B
- A bitwise-or B
- A
- A bitwise xor B
- etc..

1.2 Central Processor and Microprocessor

In the early days of computing, the CPU (central processor unit) was a large box containing the following

- Control Unit
- Execution Unit (Arithmetic Logic Unit plus register file)
- Main Memory

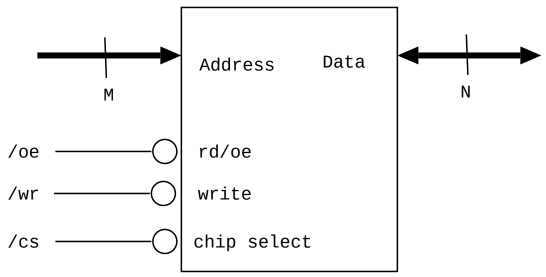

Figure 1.5: **RAM Memory Symbol**

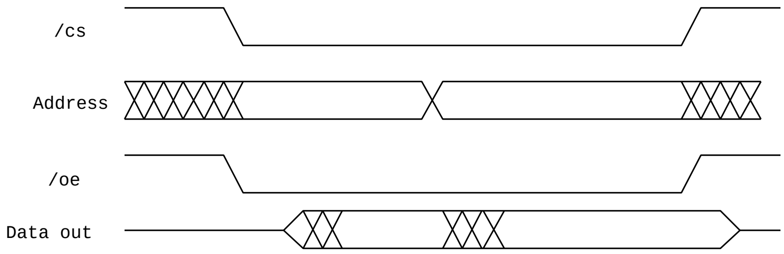

Figure 1.6: **RAM Memory Timing Diagram - Read Cycle**

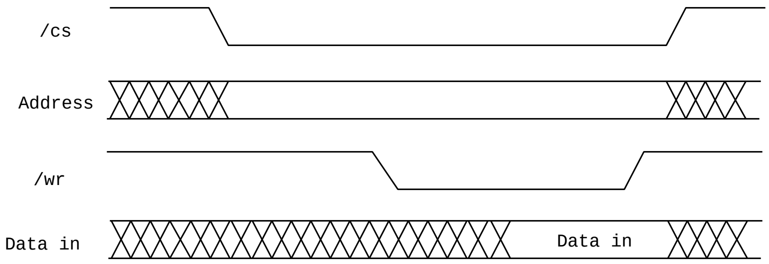

Figure 1.7: **RAM Memory Timing Diagram - Write Cycle**

The breakthrough that was the invention of the *microprocessor* was to put the first two of these three on a single chip. Today it is possible to also put about 64 kbytes of main memory on the same chip (or else put a cache on chip and use offchip RAM if more than 64 kbytes is needed).

The microprocessor has the following connections:

- Clock
- Reset
- Interrupt
- Data bus
- Address bus
- Operation request
- Read, not write.
- Wait - for generating wait states for slow devices.

The microprocessor makes bus cycles. A bus cycle happens as follows:

- It places the address on the address bus and sets read/writebar low if this is to be a write cycle and high if it is a read cycle.

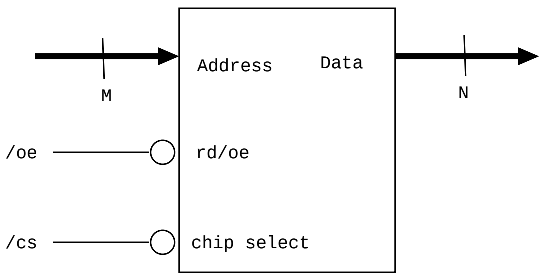

Figure 1.8: **ROM Symbol**

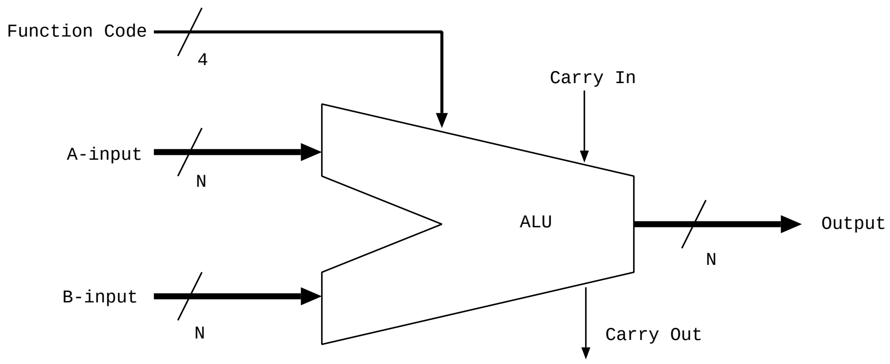

Figure 1.9: **Arithmetic Logic Unit Logic Symbol**

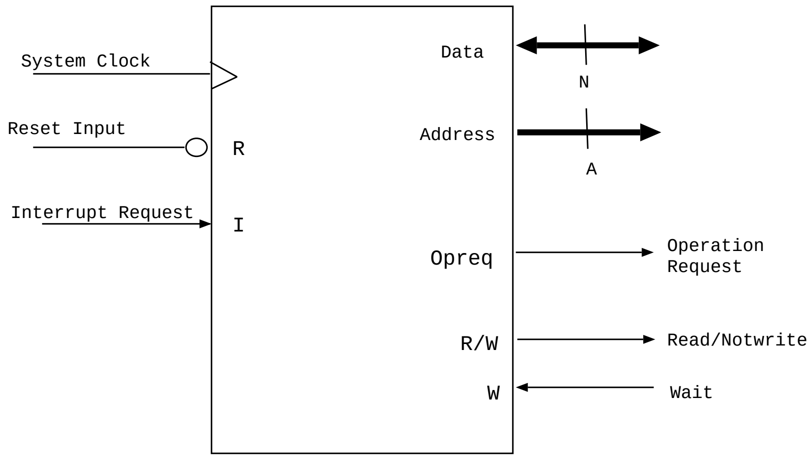

Figure 1.10: **Microprocessor Logic Symbol**

- If this is a write cycle, it drives the data to be written out to the data bus. If it is a read, it makes the data bus high-impedance ready to accept data.
- It asserts operation request (i.e. it puts it to logic 1).
- It waits half a clock cycle.
- If the wait input is asserted, it waits whole clock cycles until wait is not deasserted.
- If this is a read cycle, the processor copies whatever value has been driven onto the data bus to its internal destination
- It deasserts operation request.

Note: familiarity with and understanding of the detailed bus cycle behaviour is not expected of those who have not done the hardware course.

1.2.1 Memory Map

In a Von Neumann computer, the program and data share a single address space. The memory is a collection of binary digits (ones and zeros) or bits.

If there are a address lines, then the address space contains 2^a locations. In a byte-addressed machine, the bits in the memory are grouped into eight bit bytes for addressing. Each byte has its own address and each bit may be addressed via a byte address and an offset (from 0 to 7) within a byte.

For example, the address space (memory map) might contain be organised as follows

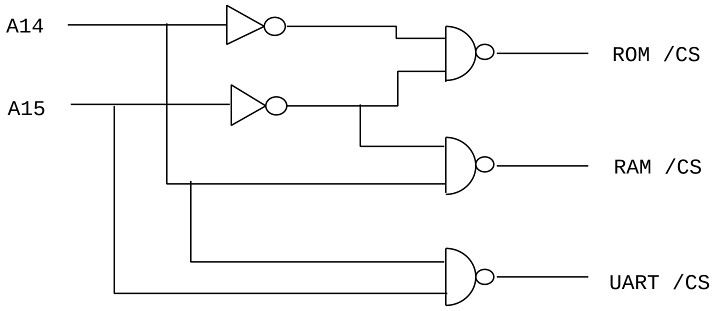

Figure 1.11: **Decoder logic to create the given memory map**

```
-------  -----    ----------------------
Start    End      Resource
-------  -----    ----------------------
0000     03FF     EPROM
0400     3FFF     Unused images of EPROM
4000     7FFF     RAM
8000     BFFF     Unused
C000     C001     Registers in the UART
C002     FFFF     Unused images of the UART
-------  -----    ----------------------
```

The processor makes access to the various resources in the machine by generating addresses on the address bus. Address decoding logic matches patterns on the high-order address lines to generate enable signals for each particular device. This logic sets the memory map (or address map) of the computer. The low-order address line are fed to each device to distinguish access to individual locations within the device.

Note: familiarity with and understanding of the decoder logic to create the given memory map is not expected of those who have not done the hardware course.

1.2.2 Processor Registers

The processor contains a number of registers available to the programmer. Registers are general purpose or special purpose. General purpose registers are used for fast access, temporary storage of variables. They have names such as

R0, R1, R2, R3, R4, R5, R6, R7 ...

or

AX, BX, CX, DX, BP, SI, SP, DI (Intel family)

or

A, B, C, D, E, H, L, IX, IY (Z80)

Special purpose registers typically include:

- a program counter (PC), which points to the current instruction
- a flags register, containing P I N Z V C
- a stack pointer register, which points to the last storage location in use in the stack.

In some architectures, any general register can be a program counter or stack pointer, but in most systems, special registers are used so that the hardware data paths may be optimised.

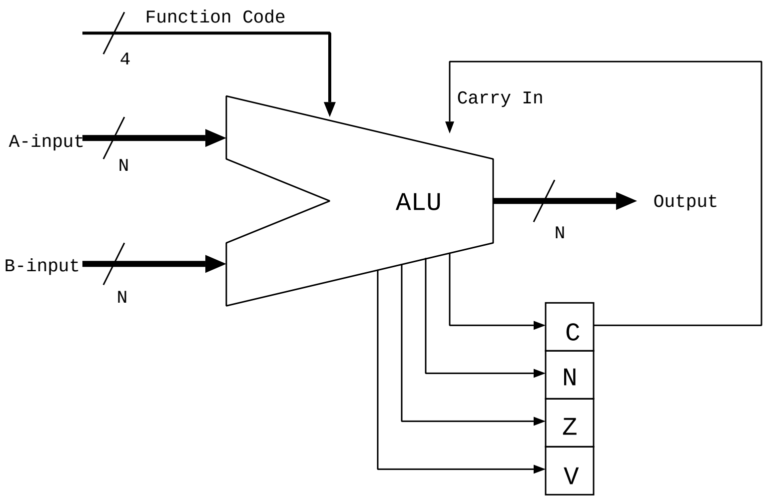

Figure 1.12: **Usual configuration of flags with an ALU**

The registers inside the processor generally have the same width as the data bus and this is known as the processor's *word size.* A modern general purpose computer typically has a 32 bit wide data bus and the same width address bus.

A *byte-addressed* computer gives each byte in address space its own address. I.e. incrementing the address by one moves you 8 bits along the memory array. Many computers in the past have been *word-addressed*.

When a data bus size larger than 8 bits is used, the computer may still be byte-addressed, but several bytes will be returned on each read cycle to external memory. The processor may require all of the bytes immediately, for instance if doing a word addition, or it may store them temporarily in a bus interface register or cache memory in the hope they will be useful shortly.

1.2.3 Flags Register

The processor flags register is a collection of single bit flags. Most processors have the following: Z N C V P I. The first four of these are used mainly with conditional branches (section 3.7.2), and the last two are used to support systems software.

Z - Zero Flag. This flag is updated after every update to a register. If the register was loaded with zero (all bits zero) the Z flag is set, else clear.

N - Negative Flag. This flag is updated after every update to a register. If the register was loaded with a value were the most significant bit was a one, N is set, else N is cleared.

C - Carry Flag. This flag is updated after every update to a register. If the update was a result generated in the ALU and the carry output from the ALU was a one, C is set, else C is cleared. The carry output from the ALU is the carry generated in the addition (or borrow in subtraction) from the most significant bit.

V - Overflow Flag. This flag is updated after every update to a register. If the update was a result generated in the ALU and the overflow output from the ALU was a one, V is set, else V is cleared. The overflow output from the ALU is the exclusive-or of the carry into and out of the most significant bit. It may be seen that overflow will be asserted if the data being operated on by the ALU is in two's complement form (section 3.3) and the result of the operation lies outside the representable value range. V is not useful for unsigned operations - use C instead.

I - Interrupt Enable Flag. This flag may be set and cleared by privileged instructions only. When set, the processor will respond to interrupts on its interrupt input(s), otherwise it will ignore them. This flag is set by systems software during critical regions of code where system data structures are transiently in an inconsistent state and so should not be viewed by other software, such as interrupt routines. It is not possible to experience an unexpected context switch while this flag is set (section 16.2). External hardware generally queues interrupts such that even if they occur while I is clear, they will persist until interrupts are again enabled (I set).

P - Privilege Flag When P is set, the full instruction set of the computer may be used and all of the memory map and register set of the processor may be accessed (section 15.1). When clear, a restricted subset of instructions may be executed which can only read or modify the general user registers and user memory space. An internal interrupt is generated, known as a privilege violation exception, if a privileged instruction is attempted when the flag is clear. This flag is set by reset, interrupt and system call instructions. Cleared by a privileged instruction of processor-specific type (such as RTI described in section 2.2).

2 Processor Operation

2.1 Fetch-Execute Cycle

The Control Unit performs the fetch-execute cycle.

1. Processor Reset. When the computer is switched on, a timer asserts the reset input to the processor for half a second or so. The reset input causes the program counter to be loaded with a fixed value, such as zero. A program must already be loaded at this address - typically a ROM is mapped here. When the reset input is deasserted, we proceed to normal operation, in step 2. Computers with a reset switch drive the reset input from the logical OR of the power on reset and the reset signal from the switch. enabling the computer to be reset without powering off and on again.

2. In this step we check for external interrupts. If (any of) the interrupt signal(s) to the processor is(are) active, save the program counter using a special purpose register (e.g. push it on the stack), set the privilege flag and load the program counter with a fixed value, or a value depending on the type of interrupt.

3. Fetch an instruction from memory at the address pointed to by the program counter. If the instruction spreads over several memory locations, fetch the rest of the instruction, incrementing the program counter as needed.

4. Obey the instruction (using the ALU if needed).

5. If the instruction was a jump, then the program counter will have been loaded with the next instruction's address during step 4, so go to step 2.

6. Else, increment the program counter to point to the next instruction and go to step 2.

Figure 2.1 shows the data paths of a Von Neumann style processor. It is clear from the structure that the instructions and data both arrive into the processor on the single data bus and that there is only one address bus - hence it a Von Neumann machine.

The data paths for saving the PC as a subroutine return address and for making indexed (calculated) data access are not shown. These paths require exchange of addresses between the control unit and the execution unit. Also there is no provision for calculating program counter relative addresses or interrupts.

2.2 Stacks and Subroutines

Most processors provide instructions for implementation of LIFO (last-in first-out) stacks. A register in the processor serves as a stack pointer, pointing to the item on the top of the stack. Two operations are needed for a stack - push and pop.

PUSH:
1. Advance pointer to free space
2. Store item at address pointed to by pointer.

POP:
1. Fetch item from address pointed to.
2. Retreat pointer to new top item.

A subroutine call instruction, often called JSR (jump to subroutine), takes an operand which is the address of the subroutine. It stores the program counter (which will be pointing to the instruction

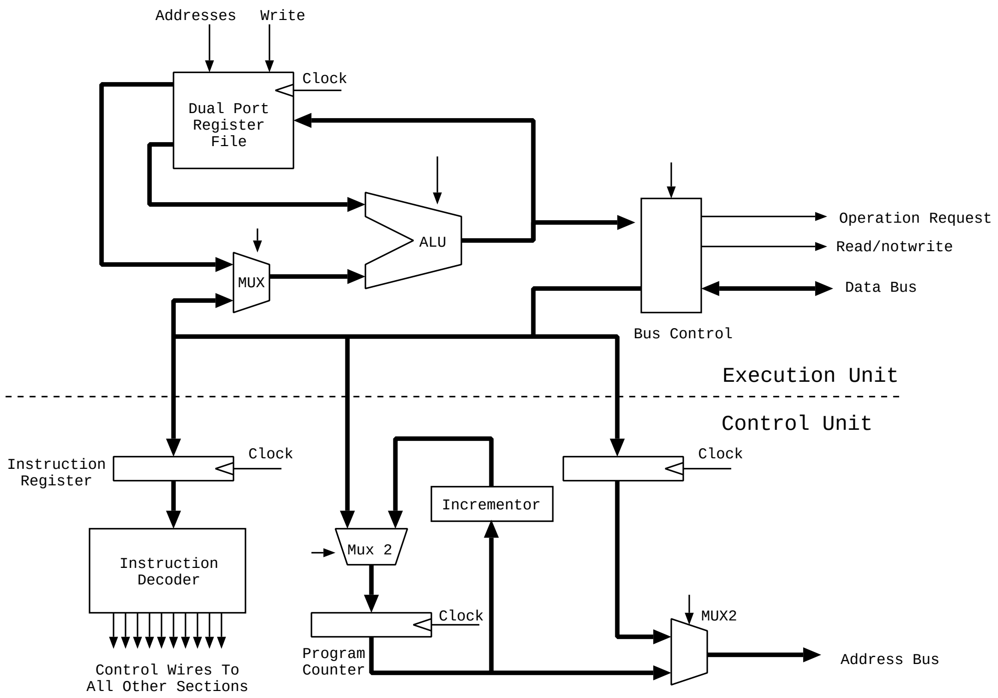

Figure 2.1: **Block diagram of a simple microprocessor**

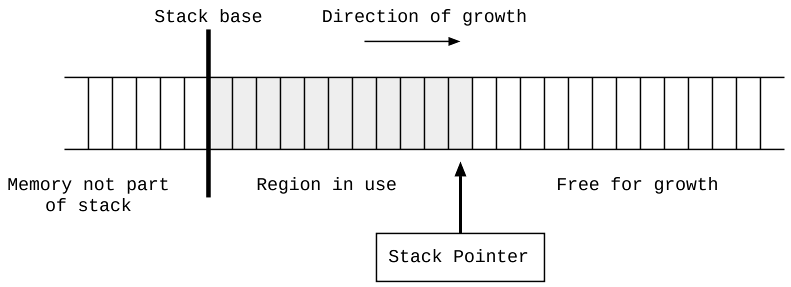

Figure 2.2: **example of a stack with a number of locations in use**

after the JSR) on the stack and then jumps to the operand address (i.e. loads it in the PC). A subroutine return instruction, often called RTS (return from subroutine), is found at the end of a subroutine. It takes no operands and simply pops the stack top value into the program counter, causing execution to resume at the point after the subroutine was called. Such subroutines can be recursive. A subroutine may keep other data, such as local variables, on the stack, provided it pops it off again before returning.

On a processor with a 32 bit program counter, the items stored on the subroutine return address stack will be 32 bit program counter values, so the advance and retreat operations are just add and subtracts of 4 on a byte-addressed machine.

A variant of the RTS instruction is RTI (return from interrupt). As well as doing an RTS, RTI pops an extra word off the stack and loads it in to the flags register. This is a convenient way to clear the privilege flag after a system call and to restore the other flags. System calls are described in section 15.2.

RTI is privileged.

Class	Source or cause
1	UART interrupt
1	Real-time clock interrupt
1	etc
2	Illegal instruction (bad opcode)
2	Illegal address of operand
2	Emulation of missing coprocessor
2	VM translation fault (page paged out)
2	Privilege violation
3	Trap (system call)

Table 2.1: **Examples of interrupts.**

2.3 Interrupts

Interrupts are unexpected, hardware driven deviations from the flow of control prescribed by the current program. It is normal for the interrupting software to save the exact state of the user registers at the time of the interrupt, so that the execution can continue at the point of the interrupt, at some later time. Hence the name, interrupt.

There are three main classes of interrupt:

1. An external device requests service by asserting an interrupt line which feeds into the processor. An example is the UART, which interrupts each time a new character has arrived in its internal registers or when ready for a new character to send.
2. While obeying the instruction in step 4 above, an error of some kind is detected. An example is a reference to non-existent memory or the loading of a bit pattern from memory in step 3 which is not a valid instruction.
3. The software interrupt, or service call, which is described in section 15.2 is also generally handled using similar processing steps, although it is not an unexpected interrupt as the above examples are.

The cause of the error in interrupt class 2 can often be fixed by the operatin system kernel (for instance correcting a page fault - section 5.2) and so when the kernel returns from the interrupt, the program counter should be wound back to restart the instruction. On the other hand, the class 1 interrupts are handled by the processor after the successful completion of an instruction, so on interrupt return, the next instruction should be executed.

The word *exception* is often used for an interrupt which occurs owing to an error.

3 Data Representation

Memory can hold any binary bit pattern, but is typically used to hold the following: signed and unsigned integers, floating-point numbers, ASCII text, machine instructions and complex structures such as a stack or composite records of pointers and simpler objects.

Let us look at examples of these and assume a 32 bit machine unles otherwise stated. When the word size is 32 bits, there are 2^{32} different combinations.

3.1 Hexadecimal Representation

From time to time, hexadecimal representation is a convenient way to express numbers. Hexadecimal is easier to read than binary, but easy to convert to and from binary. Numbers starting '0x' are in hexadecimal. A hexadecimal digit is one of 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F and represents a four bit binary field. For example, one thousand decimal can be converted to hexadecimal 0x3E8 by first converting it to binary and then grouping the coefficients in fours.

$$
\begin{aligned}
& 1000 \text{ decimal} \\
&= 512 + 256 + 128 + 64 + 32 + 8 \\
&= 1.512 + 1.256 + 1.128 + 1.64 + 1.32 + 0.16 + 1.8 + 0.4 + 0.2 + 0.1 \\
&= 1\ 1\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 0 \text{ binary} \\
&= 0011\_1110\_1000 \\
&= \text{0x3E8}
\end{aligned}
$$

The underscore may be placed in a number to help its readability.

3.2 Representing Unsigned Integers

Unsigned Integers may be held in four adjacent bytes in standard binary coding, ranging from zero to $2^{32} - 1$.

3.3 Representing Signed Integers

Two's complement form uses the most significant bit as a sign bit. If it is clear, a positive number is represented and the value is equivalent to the identical unsigned binary bit pattern. With a 32 bit word, positive values from zero to $2^{31} - 1$ can be stored.

Negative two's complement values are represented by the bit pattern which when added to the equivalent unsigned positive number in a normal binary adder results in zero. This pattern can be obtained by inverting all of the bits of the positive representation and then adding one. Values from -1 to -2^{31} are possible.

Signed integers are normally held in two's complement form since, from the above definition, the representation of the positive integers is unchanged and the arithmetic logic required for addition and subtraction is the same as for unsigned binary integers.

For example, minus 2 as an 8 bit two's complement number is binary 1111_1110 = 0xFE.

3.3.1 Some Examples of Signed and Unsigned Integer Arithmetic

Here are 8 Examples of Signed and Unsigned Operations in a 5 bit field:

1. Addition - Unsigned - No carry

```
00101    5
```

```
  + 00111   7
-------------
0  01100   12   No carry, so answer correct
-------------
```

2. Addition - Unsigned - With carry

```
    11110   30
  + 00111   7
-------------   Carry indicates answer is 32 too small
  1 00101   5    37 cannot be held in 5 bits
-------------
```

3. Subtraction - Unsigned - No Borrow

```
    11110   30
  + 11001   -7   Add two's complement to effect subtraction
-------------
  1 10111   23   Correct answer, carry of one indicates no borrowing
-------------   needed.
```

4. Subtraction - Unsigned - With Borrow

```
    00111   7
  + 00010  -30   Add two's complement to effect subtraction
-------------
0  01001  9    Carry zero implies a borrow of 32 was needed.
-------------
```

5. Addition - Signed - No overflow

```
    00101   5
  + 00111   7
-------------
0  01100   12   Carry into and out of msb the same so no overflow
-------------
   0
```

6. Addition - Signed - With overflow

```
    01010   10
  + 00111   7
-------------    17 cannot be held in 5 bit two's complement
0  10001   -15   Carry into and out of msb different.
-------------
   1
```

7. Subtraction - Signed - No overflow

```
    01010   10
  + 11001   -7
-------------
1  00011   3   Correct answer, carries both one, no overflow.
-------------
   1
```

8. Subtraction - Signed - With overflow

```
    10110   -10  (-10)
  + 10110   -10  (Take away another 10)
-------------
1  01100    12  Wrong answer.
-------------
   0
```

In overflow cases the sign of the result is always wrong (i.e. the N bit is inverted, so that's why we exclusive-or N with V in the conditional branches).

Decimal Value	S	Exponent	Mantissa	Hex representation
0	0	000 0000	0000 0000 0000 0000 0000 0000	0x0000_0000
1	0	000 0001	1000 0000 0000 0000 0000 0000	0x0180_0000
-1	1	000 0001	1000 0000 0000 0000 0000 0000	0x8180_0000
2	0	000 0010	1000 0000 0000 0000 0000 0000	0x0280_0000
2.5	0	000 0010	1010 0000 0000 0000 0000 0000	0x02A0_0000
0.25	0	111 1111	1000 0000 0000 0000 0000 0000	0x7F80_0000

Table 3.1: **Floating Point Representation Example (32 bit)**

3.4 Representing Floating Point Numbers

Floating point numbers are represented using three fields, a mantissa, a sign flag and an exponent packed into the 32 bit word. The exponent is generally to the base two. The exponent can be in two's complement to allow numbers smaller than +/- 1/2..

For example, a 32 bit word may use the most significant bit as the sign, the next seven bits as the exponent (-64 to +63) and the next 24 bits as the mantissa with the first mantissa bit having significance 0.5.

Some examples Note that bit 23 is set for all numbers except zero and numbers very close to zero. This is the *normal form* for them and generally offers greatest precision, although many numbers may be accurately represented not in normal form.

A 24 bit mantissa gives $24 \times log_{10}(2) = 7$ decimal digits of precision. The maximum value is approximately 2^{63} which is 9.2E18 and the smallest value is about 2^{-64} which is 5.4E-20.

Most processors (or floating point coprocessors) have a special set of instructions and special registers to process and hold floating point numbers.

Many processors also support extended precision floating point, which uses two words per number. The first byte of the additional word is not used, but the remaining three provide another 24 bits of mantissa, doubling the precision to about 14 decimal digits.

Always remember that floating point numbers are only an approximation to real numbers - they secretly have discrete values while deluding you into thinking them continuous.

3.5 Representing Text

Text may be represented using the American Standard Code for Information Interchange (ASCII) coding. Seven bits are used, giving 128 possible codes. Codes from 0 to 31 are non-printing control characters, such as warning bell, backspace and carriage return.

One byte in memory is normally used for one ASCII character, so four are stored in each 32 bit word. Processors where the word size is greater than one provide special instructions for access to individual bytes in memory to ease text handling. The frequent need to handle text on a computer has been a major influence in causing modern processors to use byte addressing.

3.6 Representing Data Structures

A pointer is a data value which is the address of some further data. These may be stored in memory. Consider the ML datatype

```
datatype rec = node of int * int * rec
         |     leaf of int;

val example = node(4, 5, node(6, 7, leaf(8)));
```

This could be stored as shown if figure 3.1. Note that the ML system has allocated constructor tag

```
   ASCII Character Set
-----------------------------
      2   3   4   5   6   7
-----------------------------
0  |  sp  0   @   P   `   p
1  |  !   1   A   Q   a   q
2  |  "   2   B   R   b   r       The ASCII code for a particular
3  |  #   3   C   S   c   s       character is 16 times the number at
4  |  $   4   D   T   d   t       the top plus the number down
5  |  %   5   E   U   e   u       the side.
6  |  &   6   F   V   f   v
7  |  '   7   G   W   g   w
8  |  (   8   H   X   h   x
9  |  )   9   I   Y   i   y
A  |  *   :   J   Z   j   z
B  |  +   ;   K   [   k   {
C  |  ,   <   L   \   l   |
D  |  -   =   M   ]   m   }
E  |  .   >   N   ^   n   ~
F  |  /   ?   O   _   o  del
-----------------------------
```

Table 3.2: **ASCI Character Set**

Address	Value	Comment
0xF30	2	Constructor tag for a leaf
0xF34	8	Integer 8
0xF3C	1	Constructor tag for a node
0xF40	6	Integer 6
0xF44	7	Integer 7
0xF48	0xF30	Address of inner node
0x1000	1	Constructor tag for a node
0x1004	4	Integer 4
0x1008	5	Integer 5
0x100C	0xF3C	Address of inner node

Figure 3.1: **Pointer structure stored in memory**

numbers to each instance of a rec so that it can tell whether it is a node (tag=1) or a leaf(tag=2). The inner nodes were allocated first, so are likely to be stored at a lower address, as shown.

3.7 Representing Machine Instructions

Processors vary in the number of bytes used for an instruction, but most modern RISC processors use 4 bytes for each instruction. An example RISC processor is the ARM device developed in Cambridge. Older processors, such as the Motorola 68000 (used in Macs) and the Intel family used in PCs have variable length instructions. Instructions vary from one to tens of bytes long.

Most processors contain at least the following basic instructions

```
ADD, SUBTRACT, LOAD, STORE, JUMP, JSR,  RTS, TRAP,
XOR, AND, OR, ROTATE, COMPARE, HALT, NOP
```

but the size and style of instruction sets varies wildly. Each instruction can be used with one or more *addressing modes*. The addressing modes provide a set of different ways of specifying the operands. The operands are the arguments to or result from the instruction.

The processor designer invents the instruction set, addressing mode set, number of registers etc.

```
Here is a selection of the 18 machine instructions for the
EDSAC computer [From Wilkes and Renwick, 1949].

  A n    Add the number in storage location n to the
         accumulator register.

  H n    Transfer the number in storage location n into the
         multiplier register.

  E n    If the number in the accumulator is greater than or
         equal to zero, execute next the order [instruction]
         which stands in storage location n; otherwise proceed
         serially.

  I n     Read the next row of holes on the tape and place the
          resulting five digits in the least significant places
          of storage location n.

  Z       Stop the machine and ring the warning bell.

These instructions are here represented in a symbolic form, using
a one character symbol for each instruction.  Each instruction
also has a binary representation, which is the machine instruction
code placed in memory for evaluation by the processor.
```

Figure 3.2: **Some of the instructions available on the EDSAC computer.**

He allocates an 'opcode' for each instruction and an encoding into memory for each instruction and its addressing mode.

3.7.1 Addressing Modes

In general, processors offer combinations of the following addressing modes:

Register a field specifies a register number

Absolute a field is the memory address of the operand

PC relative a field is an offset from the instruction

Immediate a field is the operand (read only)

Register indirect a field specifies a register which contains the address of the operand.

Indexed a field is an offset to be added to a specified register to generate the address of the operand.

Memory Indirect a field is an absolute address of the first of a number of bytes in memory which hold the actual operand address.

By 'field' I mean a set of bits in the instruction.

3.7.2 Example Instruction Set

Here we develop and present a subset of the instruction set of a simple, fictional 8 bit processor. This example processor is typical in style to the first generation 8 bit processors, such as the Z80, 6502 and 6800.

We will have four 8-bit general registers in the processor (needing two bits to address a register), a 65536 location address space (needing two bytes to specify an address), and variable length instructions. We will also need a 16 bit program counter and a flags register and a stack pointer register. This gives us the *user programming model* shown in figure 3.3.

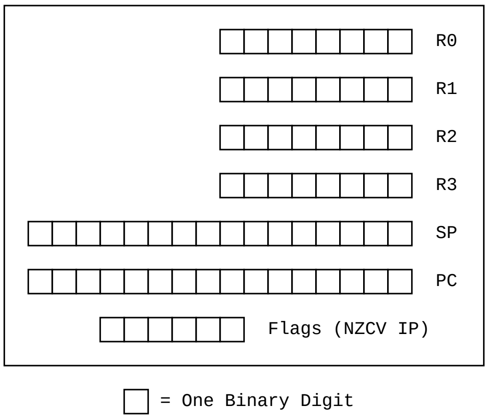

Figure 3.3: **A user programming model for an 8 bit processor with 16 bit address space.**

Instruction	Opcode	Modes
ADD	1	RA, R0-R, I
SUBTRACT	2	RA, R0-R, I
LOAD	3	RA, R0-R, I
STORE	4	RA, R0-R
JUMP	5	JC
RTS	6	none
JSR	7	A
etc		

Table 3.3: Part of instruction set for example processor

For instance, the instruction set might start as shown in table 3.3: where the opcode is stored in the top four bits of the first byte of each instruction. The addressing mode set might include those shown in table 3.4.

For the instructions which can take a variety of addressing modes, bits in the instruction must specify which mode is in use. A table is required to allocate these:

Mode	M1 M0
RA	0 0
R0-R	0 1
I	1 0

The processor must always be able to tell how long the instruction is going to be from the bytes read so far, and so how many bytes to read in total. This depends on the addressing mode. We can put the mode in the first byte between the opcode and the register number, giving the following format

```
B7  B6  B5  B4  B3  B2  B1   B0
<---opcode--->  M1  M0  <-reg->
```

The values of cc in the conditional branch are typically used in a manner similar to figure 3.4.

giving us this format

```
B7  B6  B5  B4  B3  B2  B1   B0
<---opcode--->  cc3 cc2 cc1 cc0.
```

Examples of use - I will do some examples in the lectures.

Mode	Meaning
A	One 16 bit address follows in the next two bytes
RA	One register and one memory location where the register is in the bottom two bits of the first byte and a 16 bit address is in the next two bytes following the instruction.
R0-R	The low two bits give a register for the source operand. Register zero is implied as the destination operand.
I	The operand is in the next byte.
JC	A condition (cc) in the bottom four bits of the first byte and a 16 bit address in the next two locations.

Table 3.4: Addressing modes for example procsessor

cc	Condition
0	Branch always
1	Branch never (NOP)
2	Branch if N (ie if N is a one)
3	Branch if ~N (ie if N is a zero)
4	Branch if V
5	Branch if ~V
6	Branch if Z
7	Branch if ~Z
8	Branch if C
9	Branch if ~C
A	Branch if ~Z & ~C
B	Branch if ~(~Z & ~C)
C	Branch if N != V
D	Branch if N == V
E	Branch if ~Z and N == V
F	Branch if ~(~Z and N == V)

Figure 3.4: **Table of branch conditions**

4 Caches

A cache memory is a small, fast store used to enhance the performance of a larger, slower store. The cache contains data recently retrieved from (or written to) the slower store with the hope that the information will be needed again.

Caches are widely used in systems

- Main processor cache of main memory
- VM unit cache of translations (VM mappings)
- Operating system cache of recently accessed disk pages
- Paging of physical memory under a VM system
- Distributed system cache of directory enquiries.

The hit ratio of a cache is the ratio of data retrievals that can be supplied by the cache to the total number of retrievals over some averaging interval.

Caches can be used to improve performance both for reading and writing. A write-through cache updates the main store when data is written, giving no performance gain for writing, but a copy-back cache keeps track of which entries are 'dirty' meaning that they have been updated but the write to main store has not yet occurred. The cache writes dirty entries out to main store when the main store would otherwise be idle.

4.1 A note on Cache Implementation

Implementation of caches and associative stores has been widely studied. Internally, a cache may be directly mapped, fully associative or set-associative. A directly mapped cache has only one place in the cache memory where an item from main store may be cached, meaning there is only one item to check to determine a cache hit and no doubt about where to cache new data. This has quite low performance in general, but is fast. A fully associative cache, on the other hand, has no restrictions on where in the cache an item is cached, meaning that every location needs to be examined before a hit can be determined. This is slow if done in software and complex when done in hardware, but generally has the highest hit ratio. A set-associative cache has a set of locations used for each item in main store, with a typical set size of four. Only the small number of entries in the set need to be checked to determine a hit. This clearly gives a compromise in performance and complexity and is widely used.

4.2 Example of Cache Speedup

Consider a cache with a hit ratio of about 90 percent and built from technology which operates 15 times faster than the main store. If the system speed without the cache is 1, then the system speed with the cache is

$$1/[(0.1 \times 1/1) + (0.9 \times 1/15)] = 6.25$$

In general, if the hit ratio of a cache is much less than about 0.9 the benefit of the cache starts to be small.

4.3 Instruction and Data Caches

Modern VLSI (very large scale intergration) is able to fit a microprocessor and a significant amount of cache memory onto a single silicon chip, about 1 cm on a side.

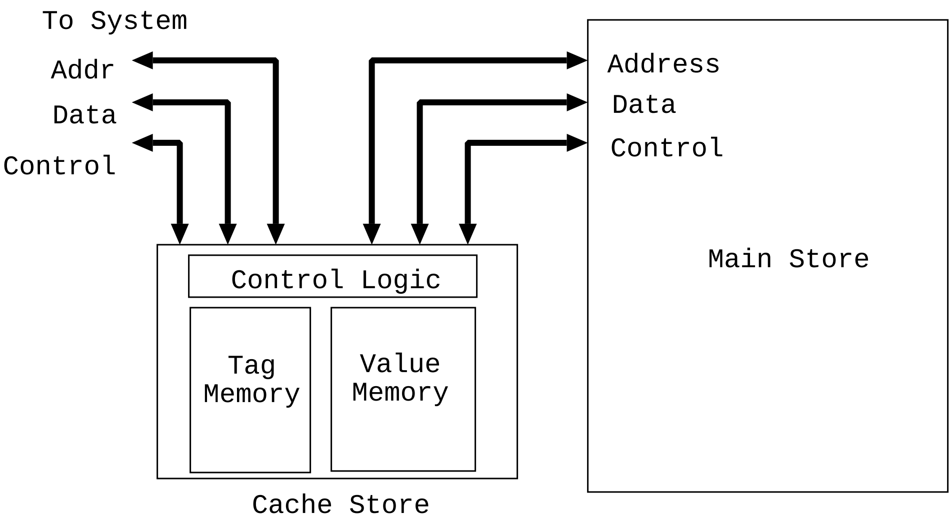

Figure 4.1: **Hardware Cache Generic Arrangement**

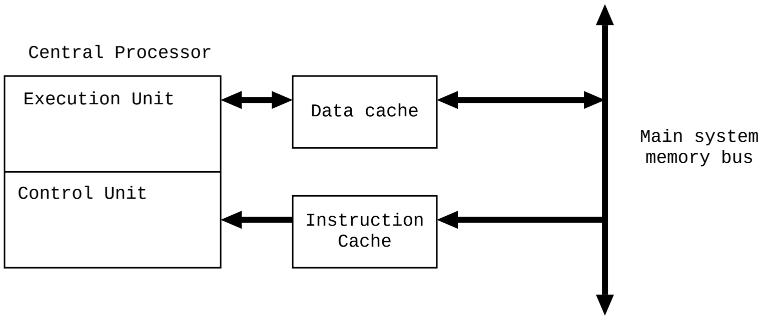

Figure 4.2: **Instruction and Data Caches**

It is common to divide the memory into two separate caches, one for instructions and one for data. The advantage of this is that they can be accessed in parallel, doubling performance. While data (operands) for the current instruction are being transferred between the processor and the data cache, in parallel the next instruction can be loaded from the instruction cache.

5 Virtual Memory Translation Hardware

Most modern general purpose computers have virtual memory translation hardware and this is often on the microprocessor chip, along with the CPU. The VM unit contains tables which can only be modified by the CPU when the privilege flag is set.

Figure 5.1 shows that a virtual memory system is created by feeding the address generated by the processor (the *virtual address*) through a memory translation unit to give the actual memory address to be used (the *physical address*). For any virtual address, the address translator produces either a physical address, which enables the instruction to proceed, or else a translation fault interrupt to the processor. Therefore, each time the processor tries to touch any memory which is not physically mapped, it gets such an interrupt, and systems software then either considers this a real error, or else reconfigures real memory, reprograms the address translator accordingly, and returns execution to the point of the instruction which failed, so that it may be run more successfully.

The instructions which modify the address translator mapping are privileged.

Virtual memory systems have several benefits:

- A program can use more memory than physically exists using *paging*.
- Separate processes will use separate programmings of the translation hardware, giving them isolation from each other, or controlled sharing.
- It is possible to have more memory than the processor can address.
- Address space becomes cheap, so that if you have to start a data structure at an arbitrary address (e.g. the base of a stack) then this can be in the middle of virtual address space, rather than in the middle of (expensive) real address space.

A VM system clearly has an overhead in translating every address generated by the processor. The delay through a translation unit will add a small percentage of time to each cycle. This penalty is normally outweighed by the above benefits of VM. (If the processor has caches which operate on virtual addresses, then only cache misses need to be translated, which is a help.)

5.1 VM unit implementation

Most translation units simply contain an associative memory which maps the input address to the output address. The low 12 or so address bits are not normally mapped, resulting in the concept of memory *pages*, inside which, the logical offset is equal to the physical offset. The associative memory is known as the *translation lookaside buffer* or TLB for short.

5.2 Paging

An operating system can supply more virtual memory than there is physical real memory if it *pages* memory onto a backing store, such as a disk. To do this, pages of memory which have not been used for a while are copied onto the disk and then become free for allocation as more fresh memory is required. The copying can be done while the computer is idle or when the computer runs out of empty memory and more is needed.

Fresh memory is needed when the program addresses memory in a page which has never been allocated. This is detected by the operating system through a page fault interrupt. Alternatively, a page fault occurs when the program access memory which has been used before and is now paged out to disk. The processor must again find 'fresh' memory, but this time must first read back the copied out data before returning to the faulting instruction for another go.

It is important not to page-out the paging-in software.

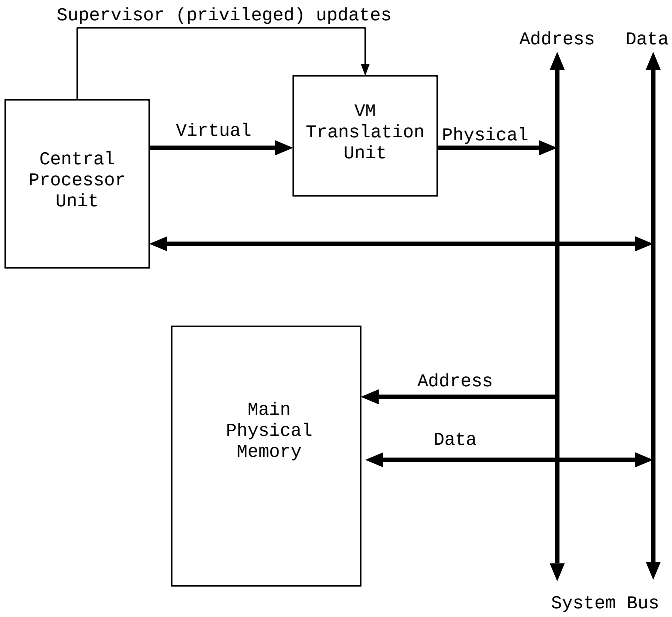

Figure 5.1: **Virtual Memory Basic Architecture**

6 Coprocessing

A coprocessor is a hardware unit which attaches closely to a main processor and contains its own registers and a specialised ALU. The coprocessor ALU is able to perform operations outside those supported directly by the ALU on the main processor. When a coprocessor is not fitted to a computer, the operations it would have performed must instead be performed using multiple general purpose instructions on the main processor, which is slower.

Coprocessors are often used for:

- Floating point arithmetic
- Graphical operations (area fill, curve draw etc)
- Encryption/decryption
- CRC calculation.

Instruction opcodes for the coprocessor are generally reserved within the instruction set of the main processor. When the coprocessor is absent, execution of one of these opcodes causes an exception (interrupt) to enable emulation software to run. When the coprocessor is present its instructions are essentially NOPs (no operations) to the main processor.

In some architectures, it is possible to use the main processor's addressing modes and operand address generation function for loads and stores to the registers in the coprocessor. For a load, the coprocessor just 'snoops' on the data bus. For a store, the main processor must make sure its databus is disabled (high impedance) so that the coprocessor can yield its register's contents onto the bus.

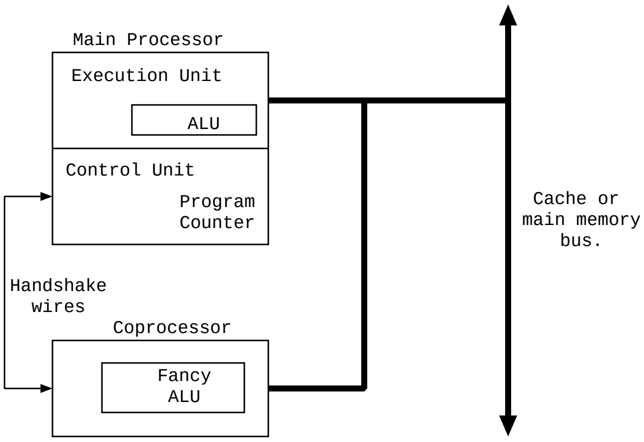

Figure 6.1: **Coprocessor Typical Configuration**

7 Input and Output

Input and output from the computer is via hardware devices connected to the bus known as peripherals. Typical peripherals are

- Universal asynchronous receiver transmitter (UART) (Better known as a serial port)
- Floppy disk controller
- IDE or SCSI bus controller for CD roms, hard disks etc.
- Keyboard
- Real-time clock
- Non-volatile memory
- Floppy disk controller
- Parallel port (for Centronix compatible printer)
- Network interface

7.1 Input and Output Cycles

The processor can interact with these peripherals through three main mechanisms:

1a. Programmed memory mapped IO

1b. IO space instructions

2. Interrupts

3. Direct memory access

We will look at these in turn. In most systems, all three mechanisms are used.

1a. Programmed memory mapped IO In memory mapped IO a small set of (uncached) locations in address space (outside main memory) is allocated to a peripheral with one location per register inside the peripheral. The peripherals are then accessed using the same instructions as used for access to main memory: ie load and store instructions. Unlike memory, the data found in the locations which are actually peripheral registers tend to change from one read cycle to the next. Storing the same value twice in one of the registers is also a potentially sensible operation, since the store might be of an ASCII character and each store operation may result in the character being printed on the printer.

1b. IO space instructions Older computers, especially those with limited address space, had special READ and WRITE instructions which performed the same functions as LOAD and STORE for memory mapped IO, generating identical bus cycles, except that an additional signal from the processor was asserted to indicate to the address space decoders that an IO space decode was required.

2. Interrupts The interrupt is a means for the peripheral to inform the processor that it is ready for attention. Without an interrupt, the processor would have to periodically poll the peripheral with a programmed IO instruction which is a wasteful effort.

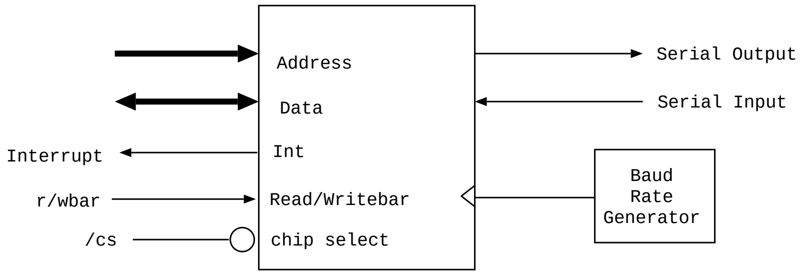

Figure 7.1: **UART Logic Symbol**

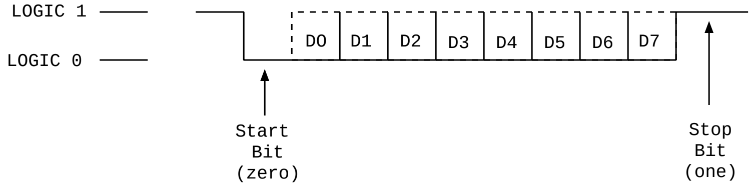

Figure 7.2: **RS232 transmission format**

3. **Direct memory access (DMA)** Direct memory access is where the processor temporarily gives up control of the bus (i.e. being bus master) and allows the peripheral to take control. The peripheral generates addresses and control signals to transfer data to or from the memory directly. DMA is normally faster for bulk data transfer than programmed IO since every bus cycle carries useful data. Programmed IO is slower because it requires each data word to be read from device/memory and then written to memory/device via a processor register with instruction fetch cycles in between.

7.2 Example: A UART for a serial port

A simple UART (universal asynchronous receiver and transmitter) may be addressed through two registers, a data register and a control/status register. A byte written to the data register is serialised by the UART and leaves the computer. Serial data which arrives may be read by the processor through the data register. The control/status register has status flags which change by themselves and control bits which retain the values written there by the processor.

A minimal interface model could be as follows (only one address line is needed if there are just two locations within the device):

The status bits have the following meanings:

`Rav - a one if a received byte is available.`

```
*---------**---------------------------------*
| A  R/~W ||  D7  D6  D5  D4  D3  D2  D1  D0 |
*---------**---------------------------------*
| 0  1    ||   <------ received byte ------> | Rx data
|         ||                                 |
| 0  0    ||   <------- byte to send ------> | Tx Data
|         ||                                 |
| 1  1    ||  Rav Tav Rin Tin Rie Tie Eer Eie| Status Read
|         ||                                 |
| 1  0    ||  X   X   X   X   Rie Tie X   Eie| Control Write
|         ||                                 |
*---------**---------------------------------*
```

Figure 7.3: Programming model for an example UART

```
Tav - a one if the transmit register is empty and so
      available for the next transmission

Rin - the interrupt line is asserted because the receive
      register is non-empty and Rie is set.

Tin - the interrupt line is asserted because the transmit
      register is empty and Tie is asserted.

Eer - an error of some sort has occurred since this register
      was last read.
```

The control bits are all interrupt enables. If set by the processor, then the interrupt output from the uart will be active when the interrupting event occurs.

A number of errors can occur, even with such a simple device. We have three overrun/underrun cases:

1. Processor reads data register when Rav zero
2. Data arrives when Rav is set because the processor has not read out the last byte yet.
3. Processor writes to data register when Tav zero because it has not waited for the last byte to be serialised.

Other errors can occur on the receive side if the stop bit is missing or if the UART detects the input data changing in what it thinks is the middle of a bit cell.

The error flag in this simple UART is not really much use because there is no cause indicator register. Real UARTs have multiple error flags and also allow the programmer to enable automatic insertion and checking of a parity bit between the last data bit and the stop bit.

Note that an enhanced version of the UART could insert a FIFO memory in series with its transmit and receive streams without modifying the programming interface model. This would reduce the chance of overrun or underrun and can reduce the number of interrupts to the processor.

7.2.1 UART Programming Procedure

Typically the system will keep input and output circular buffers for data received but not yet processed and data generated but not yet transmitted. Overall this calls for three sections of code to be written:

wrch a routine to queue a byte for output

rdch a routine to read an input byte

uisr the interrupt service routine.

7.3 Keyboard and Mouse Interfaces

Most keyboards and mice these days have a serial link to the system unit. A circuit very like a UART is required for their interface.

Keyboards normally generate *scan codes* rather than ASCII directly. A code is generated each time each key is pressed or released. In this way, different alphabets and layouts can be supported merely by changing the system software and keytops. Codes are generated when a key is released, since for shift and control keys, the computer needs to know when the key was released.

Mice generate a code each time a button is pressed or released and each time they cross a line in their quantisation grid. The cross code is one of four, indicating one of the four possible directions of crossing.

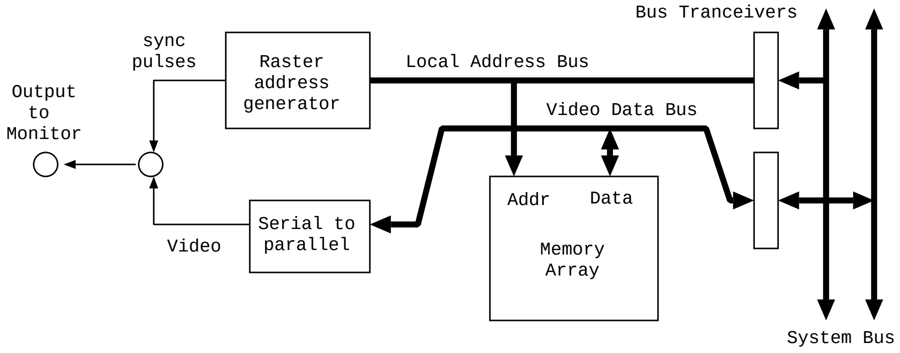

Figure 7.4: **Memory Mapped Display Controller Block Diagram**

7.4 Memory-mapped display

A memory-mapped display looks like a normal region of RAM to the processor. A pixel-mapped monochrome display where each pixel is either on or off needs one bit per pixel. If it is 1024 pixels by 650 pixels then the memory region needed is

$$1024 \times 650/8 \text{ bytes} = 83200 \text{ bytes}.$$

Displays which have colour or greyscales require more bits per pixel. Normally the lowest byte in memory corresponds to the top left corner of the screen and the ordering is left-to-right, top-to-bottom. The bits making up each each byte are either used msb first or lsb first to make up eight horizontal pixels.

Systems software is provided which takes a stream of ASCII bytes and renders them using a font table (or character generator table). The software typically also provides a cursor and interprets the non-printing control characters to move the cursor.

7.5 Intelligent Devices

Sometimes an I/O device will help in the data processing. Examples are graphical display devices which accept high-level commands for drawing circles or shading polygons and network interface devices which do scatter/gather of blocks of memory chained together by the main processor or perform retransmission in the case of an error. These intelligent devices process in parallel with the main host processor, thereby freeing it up.

Some devices today contain processors yet retain their older dumb interface to the host processor. This gives backwards compatibility. Examples are disk drives which perform internal caching and read-ahead of data blocks, without telling the host processor, and even a large class of PC keyboards which send a convoluted sequence of numeric-shift down and up strokes each side of a press of the cursor key press in order to emulate early PC keyboards which did not have dedicated cursor keys.

8 Disks, Controllers and Filing Systems

Disk drives record data serially on a rotating magnetic medium in concentric tracks. On a hard disk drive there are multiple platters, whereas on a floppy there is just the single plastic disk (one platter). Each platter has two surfaces each with its own read/write head. The set of tracks across the platters which can be accessed without moving the head assembly are known as a *cylinder*.

The sector is the basic unit of reading or writing and typically contains between 128 and 2048 bytes.

A note on sectoring: Disks are either hard or soft sectored. This refers to the method of knowing which sector is which around a track. A hard-sectored disk uses a physical index mark on the hub which causes sector pulses to be sent to the controller as the disk rotates, with a special double pulse, once per rotation, to identify sector 0. With a soft-sectored disk, the sectors consist of two parts, a header containing the address, put there when the disk is formatted, and then the sector data region which is updated each time the sector is written. Under hard-sectoring, the controller counts pulses from the double pulse to the required sector, whereas with soft-sectoring, the controller reads each sector header as it passes.

8.1 Disk Drive Controller

A disk controller sits on the processor bus and generates control signals for the disk drive. Hard disks and floppy disks are similar in most respects, as the typical parameters in figure 8.1 shows.

The functions of a disk controller are:

- Generate pulses for moving the head in or out
- Generate head select value
- Scan sector headers (or count index pulses)
- Convert data to/from serial format for the head.
- Generate/check a CRC (cyclic redundancy check) for each sector.

Disk drive timing is slow then fast: there may be milliseconds of delay between the desire to access a sector and the head being in the correct place, but once there, the data is read or written

Parameter	Hard Drive	Floppy Drive	
Number of platters	4	1	
Number of heads	8	2	
Rotation speed	3600	300	rpm
Bit rate (MFM)	10000	250	kbps
Bits per track	166	50	kbits
Bytes per track	20	6	kbyte
Sector size	1024	512	bytes
Sectors per track	18	9	
Cylinders	500	80	
Capacity	72000	737	kbytes
Seek one track			ms
Seek full width			ms

Figure 8.1: **Disk Drive Example Parameters**

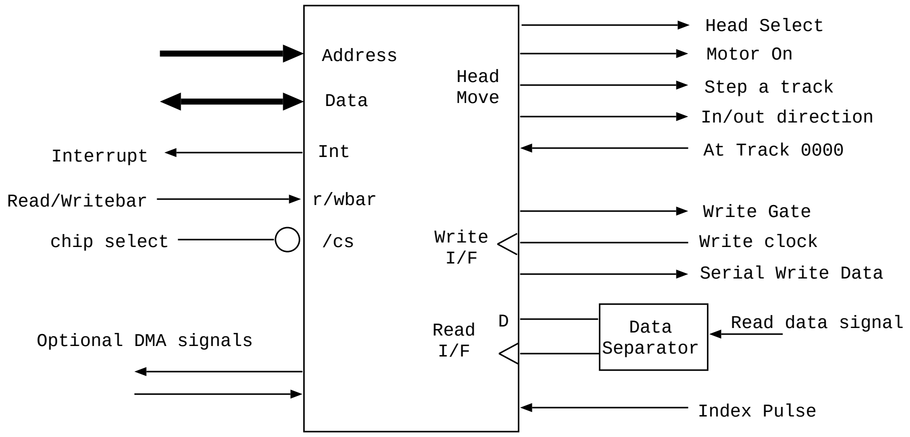

Figure 8.2: **A Disk Controller**

from/to the disk surface at full rotational speed. The following sequence of processor to controller interactions is typically used to overcome this:

1. Processor writes a command to the controller using programmed IO. The command is either a read or a write and contains the head, track and sector number.
2. The controller generates pulses to move the head assembly to the correct cylinder, waits for the correct sector to come under the head, then issues a DMA request.
3. The processor gives up the bus and the memory system sinks or sources the required sector's worth of data, then relinquishes the bus.
4. The controller generates an interrupt so that the processor may read a status register in the controller telling the outcome of the operation (e.g. CRC error) and then issue the next controller command.

A note on imbedded disk caches: Many modern disk controllers have a built-in track memory buffer RAM. The processor may access this RAM at its own speed, removing the real-time requirement to transfer data from main processor store to/from the disk. Disk drives for PCs typically have multiple such buffers, arranged to provide caching, in order to increase performance when using the primitive systems software on a basic PC, but it is always more expensive to invest in specialised RAM subsystems than using part of the main system memory. The Linux OS can use all 'unused' system memory as a disk cache.

8.2 Data arrangement on disk

In a typical operating system, a disk will contain files and directories and each file will consist of an ordered finite sequence of bytes. The filing system has conventions which map these objects onto the array of sectors provided by the disk. The disk must also explicitly, or implicitly, represent a free sector list.

A simple *serial* file system keeps files in consecutive sectors, but this leads to free space fragmentation over a period of time. The fragmentation problem occurs when the medium (disk) is quite old and has had many files created and then deleted. The effect is that sometimes a new file cannot be writtent to the disk beacuse, although there is sufficient free space, it is fragmented into small chunks, with no chunk large enough. *Compaction* is the name of the process of copying things around so that the free space is coalesced.

A *sequential file* system links arbitrary sectors into a chain and so overcomes the 'hard' fragmentation problem of not being able to write a new file, but there is a 'soft' fragmentation problem which can result in low performance as files weave over the medium in ever more twisty chains.

A *random access* file system can be built on a sequential file system. For random access to an offset within a file it is necessary to have stored, or on-the-fly generate, an index which maps offsets within the file to a sector number and byte within the sector.

8.3 Example: MSDOS File Organisation

In MSDOS, a disk is first logically divided into four sections with the following typical sizes.

1. The boot block. 1 sector.

2a. The file allocation table (FAT) 3 sectors.

2b. A copy of the FAT 3 sectors.

3. The root directory 5 sectors.

4. The dynamically allocated remainder

The boot block is on cylinder 0, head 0, sector 1 for all disk types and encodes a description of the disk parameters including sides, tracks, sectors per track and bytes per sector. This establishes a logical to physical mapping function, such that given any integer in the range 1 to the number of sectors on the disk, the track, side and sector number can be calculated. The rest of the filing system can then simply deal with logical sector numbers. The mapping is structured so that a small change in logical sector number results in a small displacement on the disk surface, so that serial allocation of sector numbers as a file is written does not cause the head to scurry all over the disk. The boot block also gives the FAT size and the sector number of the first sector in the root directory.

The file allocation table (FAT) is an array indexed by a logical sector number where each entry is also a logic sector number. There are two copies of it in order to reduce the possibility of loss. They are (should be) always the same. The FAT is used to stitch sectors together into a chain (linked list), since given a sector number, the FAT can give the successor number. Two special values are used: one to represent that a sector is not in use and one for the last sector of a chain.

The directory consists of a number of 32 byte fixed-length records. Each may hold a file or subdirectory name. The bytes are used as follows:

```
0-7    File name
8-10   Filename extension
11     Attribute flags
        1=Read only, 2=Hidden, 4=System file, 8=Volume label
        16=subdirectory, 32=Archived flag.
12-21  Reserved
22-23  Time of modification
24-25  Date of modification
26-27  Starting logical sector number
28-31  File size in bytes
```

Byte 0 can also hold a non-printing ASCII code to represent the directory slot status:

```
00=never used slot
E5=erased slot (was in use once).
```

Here is an example directory listing.

```
   HTCORE    CV     8:57:52  18-july-1994  6005  (cluster=52, sector=115)
   LINKCTL   CV     8:57:56  18-july-1994  6373  (cluster=58, sector=127)
   TAB       C      14:52:48  18-july-1994  34078  (cluster=265, sector=541)
   GAVINS    CV     11:38:56  18-july-1994  29630  (cluster=312, sector=635)
   VPLUS     C      20:20:28  23-oct-1994  7473  (cluster=351, sector=713)
   VERIOUT   C      20:20:8  23-oct-1994  4643  (cluster=348, sector=707)
   SYSDES    N      17:18:48  15-april-1995  2581  (cluster=371, sector=753)
Displayed total 358987 bytes.  354304 bytes free.
```

And here is the hexadecimal and ASCII display of part of the sector containing it:

```
48 54 43 4F 52 45 20 20 43 56 20 00 00 00 00 00   HTCORE  CV .....
00 00 00 00 00 00 3A 47 F2 1C 34 00 75 17 00 00   ......:G..4.u...
4C 49 4E 4B 43 54 4C 20 43 56 20 00 00 00 00 00   LINKCTL CV .....
00 00 00 00 00 00 3C 47 F2 1C 3A 00 E5 18 00 00   ......<G..:.....
54 41 42 20 20 20 20 20 43 20 20 20 00 00 00 00   TAB     C   ....
00 00 00 00 00 00 98 76 F2 1C 09 01 1E 85 00 00   .......v........
47 41 56 49 4E 53 20 20 43 56 20 20 00 00 00 00   GAVINS  CV  ....
00 00 00 00 00 00 DC 5C F2 1C 38 01 BE 73 00 00   .......\..8..s..
56 50 4C 55 53 20 20 20 43 20 20 00 00 00 00 00   VPLUS   C  .....
00 00 00 00 00 00 8E A2 57 1D 5F 01 31 1D 00 00   ........W._.1...
56 45 52 49 4F 55 54 20 43 20 20 00 00 00 00 00   VERIOUT C  .....
00 00 00 00 00 00 84 A2 57 1D 5C 01 23 12 00 00   ........W.\.#...
53 59 53 44 45 53 20 20 4E 20 20 20 00 00 00 00   SYSDES  N   ....
00 00 00 00 00 00 58 8A 8F 1E 73 01 15 0A 00 00   ......X...s.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

8.3.1 Sequential Read and Write Algorithms

Most operating systems offer sequential, byte-orientated read and write access to files. We need two sets of three subroutines available for calling by the application program

1. Open the file - file name is an argument,
2. Transfer a byte of data,
3. Close the file,

one set for reading, another set for writing.

The data transfer phase transfers bytes to or from a RAM buffer of size one sector maintained by the filing system software. For writing, when all of the bytes from one sector have been supplied, the filing system writes out the RAM buffer to the device and chains to a new sector. For reading, the filing system first reads a sector from the device into the buffer, then suplies the bytes one at a time to the application. When all of the bytes from one sector have been supplied, the filing system chains to the next sector.

To write a file the following algorithm can be used (if the file existed before, then it should first be deleted):

```
Open:
    1. Load the FAT into memory
    2. Find a free sector to be the current sector
Data Write: (This is done as many times as needed)
    3. Store bytes in a sector-sized memory buffer until full
    4. Write the buffer to the disk at the current sector address
    5. Find a nearby free sector as the current sector
    6. Make an entry in the memory copy of the FAT to link the old
       current sector to the new current sector.
Close:
    7: Write partially filled buffer to current sector
    8: Find an empty directory slot and make an entry.
    9: Write out the memory copy of the FAT (twice).
```

To sequentially read a file

```
Open:
    1: Scan the directory for the file name
       If not found then raise 'no such file' error.
    2: Set the current sector to be the starting sector specified
       in the directory entry.
Data Read: (This is repeated until we reach the end of file)
    3: Read the current sector into a memory buffer
    4: Supply the bytes to the application
```

```
    5: Chain to next sector.
Close:
    6: No disk activity - just mark buffer as free for use again.
```

9 System Busses

The performance of a computer is critically dependent on the bus cycle time, but the speed that a bus can reliably be operated at is inversely proportional to the number of devices connected and its physical extent.

Two or more busses are used in all but the simplest imbedded microcontrollers rather than having a single bus which interconnects all system devices with a maximum of compromise.

9.1 The Processor Bus

The fastest bus connects the processor to its cache. The cycle rate of this bus is also the processor clock cycle rate (e.g. 66 MHz) for modern processor designs. When a microprocessor and its cache are on the same chip, this bus is entirely on the chip too, but the processor must slow down to the external bus speed each time the cache misses.

9.2 The Memory Bus

The next speed bus is the main memory bus, which is optimised mainly for memory access cycles, and so runs at the speed of the memory system. Typical DRAM systems provide 32 bits at a time and require 120 nanoseconds to respond to a random request and 120 nanoseconds to recover afterwards. The cache controller for the main processor, after a miss, will often also fetch the surrounding 12 bytes, since many DRAM chips can supply these with a delay of 40 nanoseconds per 32 bit word.

On a high-performance computer, the width of this bus may be between 128 and 1024 bits. This enables use of the same DRAM chips as used widely in cheap computers, but gives greater memory bandwidth.

9.3 Peripheral and IO Busses

Only a fraction (e.g. 1 to 15 percent) of processor cycles are to IO devices, and IO devices tend to

- be slow
- quite big
- require modular instantiation (e.g. they're on plug in cards),
- have a design lifetime longer than the processor's
- come from various manufacturers.

The memory bus would suffer in performance if it had to directly meet these requirements. This leads us to adopt a slower, separate, well-defined, stable bus specification for IO devices.

The bus bridge connects the memory bus to the IO bus. When the processor accesses IO space, the bus bridge copies the address to the IO bus and the data in the appropriate direction (read or write) and also generates wait states for the processor to match the bus speeds.

Direct memory access (DMA) (see also section 7.1) is supported over the bus bridge. An IO device can temporarily deny the processor access to its memory when a block of data is streaming in from or out to a device.

9.3.1 IO Bus Examples

History is littered with bus specifications. Example IO busses in common use today are:

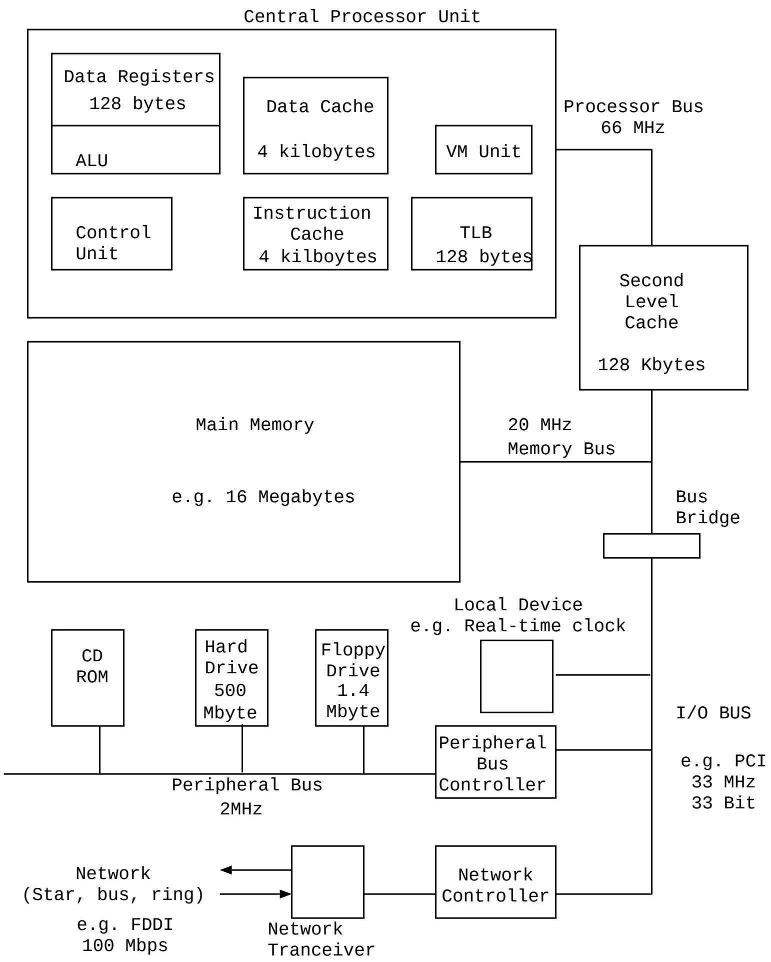

Figure 9.1: **A system with multiple busses**

IDE

SCSI

VME

Turbochannel

S Bus

ISA

PCI

EISA

IBM Channels

Fibrechannel

SCI

IEEE-488

Multibus

Nubus

Many computers today have both a local IO bus and a separate peripheral bus. The local IO bus tends to be directly addressed by the processor during programmed IO cycles and can also support DMA. It is confined to a printed circuit backplane or motherboard. The separate peripheral bus (e.g. SCSI) is connected by a special, semi-intelligent controller device and physically consists of ribbon cables.

9.3.2 Peripheral Bus Example - SCSI

The small computer systems interface (SCSI) bus was defined by IEEE committee X3.131 in 1986 to connect disk drives, CD ROMs, tape streamers and other similar peripherals within a computer. The SCSI bus is connected to the host bus using a SCSI controller which is typically a single chip.

The wires making up SCSI are

```
DB0-7   - 8 bit data path
DBP     - parity bit on the data
ATN     - attention
BSY     - busy
ACK     - acknowledge
RST     - reset
MSG     - message
SEL     - select
C/D     - command or data
REQ     - request
I/O     - in or out
```

These wires run in parallel to all connected devices. Each device on a SCSI bus is manually allocated a separate number in the range 0 through 7.

The SCSI bus, IEEE-488 bus, Fibrechannel and several others operate using approximately the following ordered phases:

- **Bus Free Phase.** The bus is idle.
- **Arbitration Phase.** This is an election process between contending *initiators* or potential *masters* to gain control of the bus. In simple systems there is only one candidate who can stay in control all the time.
- **Selection Phase.** The master selects another device to be the *target* for some transactions. The master puts the device number of the target on the bus and the target responds by asserting a signal, such as the SCSI BSY.
- **Command Phase.** The master sends some command bytes to the target device.
- **Data Transfer Phase.** The master sends data or reads data from the target.
- **Result Phase.** The master reads a status report from the target to check whether any errors occurred.
- **Bus Free Phase Again.** The master gives up the bus.

For reads there is often a pause between the command and the data transfer, whereas for writes the pause may be between the data transfer phase and the result phase. The bus and device can remain selected during the pause, or can be re-arbitrated to service another device. The SCSI REQ signal can be asserted by a target device when it wishes to be serviced - the signal typically is processed by the SCSI controller to become a system interrupt or DMA request.

New variants of SCSI have 16 bits of data instead of 8 and faster data transfer rates.

9.4 The Local Area Network

The local area network (LAN) may be one of several popular varieties. The most popular contemporary network is Ethernet, with IBM's Token Ring and Apple's Appletalk also in wide use.

Dest Address	Source Address	Data Payload

Figure 9.2: **A Datagram**

All of these have the ability to accept, send and receive *datagrams*. A datagram has a *payload* of between zero and a few thousand bytes, prefixed by source and destination addresses. It is carried inside a network *frame*.

Every network interface card ever built (in recent times) has a unique MAC (media access control) 48 bit address burned into a PROM. These are given out by a central authority. To communicate with another machine on the network it is necessary to know its address somehow. This address and the payload information are passed to the network interface hardware.

The interface puts the address at the start of the network frame (the destination address), followed by the machine's own address (the source address), followed by the information to be carried. To form the whole frame, network hardware also adds initial preambles and flags before these fields and a CRC on the end for error detection. Equipment, such as bridges, routers and other computers, connected to the network, examines the destination address of passing frames and filters out those they are interested in. The filtering consists of matching the destination address against the local MAC address in the PROM. The interface typically DMAs the received frame into pre-allocated memory buffers and then raises an interrupt. Frames with CRC errors are dropped.

The network interface device driver (section 17.1) will maintain counts of transmitted, received and received-with-bad CRC frames, for management purposes.

9.5 Supercomputers

A supercomputer is any computer where the programming model contains more than one processor unit. This should not be confused with a super-scalar processor, where the programming model is unchanged, even though there may be multiple execution units operating in parallel whenever they can.

There are two main kinds of supercomputer:

- SIMD. Single instruction, multiple data.
- MIMD. Multiple instruction, multiple data.

In the first class we have array processors. These fetch one instruction at a time in the same way as a normal SISD (single instruction, single data) uniprocessors, but each instruction operates on an array of registers or set of consecutive memory locations. Physically, one instruction fetch and decode unit with one program counter is required and multiple ALUs and sets of data registers are required. The number of data physically operated on at one time depends on the the number of physical ALUs (etc.) installed in the machine, and this may vary from model to model. In order to make programs portable, on a model with a low number of processors, the actual instruction will have to repeated to handle the data width expected. This repetition can be done by the instruction decoder or in higher levels of the software preparation tools. SIMD machines are typically used for finite element modelling for such applications as weather forecasting and stress analysis in spacecraft aerofoils.

In the MIMD class of supercomputers we have closely coupled and loosely coupled varieties. Processors in loosely coupled MIMD machines operate more autonomously and with less communication than in closely coupled machines. A set of workstations on a local area network or a set of processors all sharing the same main memory but with their own caches (such as a multiprocessor PC or thor) are loosely coupled. Typically each processor runs a separate program, but they share a common operating system and file system.

In a closely coupled MIMD machine, many tens or hundreds of processor units operate, each with their own program counter, cache and local memory. The processors are interconnected very fast links in a three, four or more dimensional interconnection pattern, so that each has many logical neighbours. Using these links, processors can read or write directly to and from the local memory of their neighbours. At any given processor, the local memories of the neighbouring processors is typically logically mapped into the address space of that processor and in order to access the neighbouring memory over the links, all that software needs do is generate a read or write instruction to the correct address: all of the communication aspects are handled in hardware. The new Hitachi computer in our department is an example of a closely coupled MIMD machine (I think).

Part II

Resident Software

10 System Software, Resident Software and others

Figure 10.1 is a little taxonomy of software. We start by dividing software into two major classes, system software and applications software.

An applications software program, or more briefly, an *application*, performs a specific user function. Examples are a spreadsheet, a wordprocessor, a database or a game. Systems software is all other software.

A subset of system software is resident software. Resident software is used to load and execute an application program. It includes the resident operating system kernel, device drivers, filing system and protocol software, authentication software, windowing software, emulators for floating point units and any other expected resources, and interpreters. This software is the subject of part II of these notes.

Another subset of system software is program preparation tools. Program preparation tools include compilers, macroprocessors and the link editor. These are used to create a program and store it in an executable file (or files). This software is the subject of part III of these notes.

A third class is systems applications. These are programs which appear to the system as normal applications, but to the users are vital parts of the environment, without which the system would be unusable. Examples are a directory display program, programs for login, logout, password changing and backup of the filing system.

<table>
<tr><td colspan="3">All Software</td></tr>
<tr><td colspan="2">Systems Software</td><td>Applications Software</td></tr>
<tr><td>Resident software</td><td>Preparation software</td><td></td></tr>
</table>

Figure 10.1: **Taxonomy on software**

11 Loading the first program - *booting*

When the processor is reset the program counter is forced to a fixed starting location. This memory must already hold a valid program or else the computer crashes.

A *crash* is when the program counter does not point to a valid program, but points to data, uninitialised store or unused memory map.

The initial program is either stored in non-volatile memory, such as ROM (read only memory) or must have been loaded into (volatile) store by another method. Older computers had their initial program loaded in hardware from front panel switches or paper tape, but today we always use either

- mask programmed ROM
- eraseable ROM using floating gate technology.

(Volatile memory is memory which forgets its contents when power is removed, such as static and dynamic random access memory (SRAM and DRAM) used in today's computers for general purpose memory.)

The initial program is typically a loader which loads in the operating system *kernel*. The kernel may be defined as the part of the operating system which is loaded in at start of day and remains there until the computer is switched off or *rebooted*.

The boot ROM may perform basic peripheral and memory tests before loading the kernel. The ROM may search for other device-specific ROMs on plug in cards and execute routines from them. Under normal operation, these ROMs are not accessed (but see note about BIOS in section 12.1). If the any of the tests fail, or the ROM fails to find and load a kernel, the ROM may enter an interactive command line environment for debugging. The debugging commands normally available allow the human to suggest the device number and filename of an alternative kernel, display and change memory locations, jump to an address, run the ROM-based diagnostic routines, disassemble memory into assembly language and set *breakpoints*. A breakpoint is a patch to a loaded program which causes the interactive debugger to be reentered when program flow reaches that point (a TRAP instruction is often used).

11.0.1 Example Diagnostics

Here is a *backtrace* and register dump from a machine which has failed. The failure is a trap 'offset mismatch', whatever that may be

```
 0: trap: offset mismatch: vector: 108 offset: 108
 0: CPU STATUS:
 0:   D0 0x000308a5   D1 0x00000000   D2 0x00000000   D3 0x00000000
 0:   D4 0x00000000   D5 0x00000000   D6 0x00000000   D7 0x00000000
 0:   A0 0x00044a72   A1 0x00000000   A2 0x00044f6a   A3 0x00000000
 0:   A4 0x00000000   A5 0x00000000   A6 0x0003fb44   A7 0x00000000
 0:   PC 0x0001538e   SR 0x3700
 0:  VBR 0x00000000
 0: CAAR 0x0004b98c CACR 0x00003111
 0:  ISP 0x00021300  MSP 0x0003fb3c
 0:  DFC 0x00000000  SFC 0x00000000
 0: -- Start backtrace --
 0: pc 0x0001538e
 0: ra 0x0000e7a2 args 0x000493e0 0x00000000 0x00000000
 0: -- End backtrace --
147-Bug>
```

The last line is the interactive prompt for debugging.

Next we display the memory at the value of the program counter when it failed, in hexadecimal (with ASCII for free) and then in disassembled assembly language:

```
147-Bug>md 15380,153c0
00015380 60FE  4E5E  4E75  4E56   0000  202E  0008  51C8      '~N^NuNV.. ...QH
00015390 FFFE  4240  5380  64F6   4E5E  4E75  6475  6D70      .~B@S.dvN^Nudump
000153A0 5F74  6362  3A20  6E6F   2074  6872  6561  6420      _tcb: no thread
000153B0 7275  6E6E  696E  670A   0070  635F  7468  7265      running..pc_thre
000153C0 6164                                                 ad
147-Bug>
147-Bug>
147-Bug>
147-Bug>md 1538e;di
0001538E 51C8FFFE          DBF.W       D0,$1538E
00015392 4240              CLR.W       D0
00015394 5380              SUBQ.L      #$1,D0
00015396 64F6              BCC.B       $1538E
00015398 4E5E              UNLK        A6
0001539A 4E75              RTS
0001539C 6475              DC.W        $6475
0001539E 6D70              BLT.B       $15410
147-Bug>
```

In general, we have the following sequence of events when a PC or workstation is switched on:

1. ROM does a system self test (find memory size)
2. ROM finds a kernel, loads it and jumps to it.
3. Kernel sets up its data structures and starts the filing system
4. Kernel reads system configuration files and obeys them
5. Kernel starts up login process to accept users
6. Login program allows the user to log in.
7. Login program starts a shell or window manager
8. User identifies an application program
9. Kernel loads the application and starts it
10. Application runs - talks to the user and other devices
11. Application ends, returning control to shell or window manager
12. User finally logs out - kernel frees up resources that were in use.

12 Kernel and Shell

The permanently resident software for a general purpose computer is often called the operating system *kernel*. It is loaded at boot time and remains in place while other program come and go. Other names for it are the executive, the supervisor and, in a simple computer, the monitor program.

The kernel must provide the ability to:

- Load transient programs and allocate memory and execution time to them,
- Reclaim resources used by the transient programs once they have exited,
- Protect one loaded program from another so that a fault in one does not affect others or the kernel.
- Provide resources to the loaded program, such as access to the filing system, printers, networking, time of day, interprocess communication, more memory, etc.

In UNIX the *shell* is just like any other loaded program, except it happens to be the one loaded by the login program when a user logs in. All programs have the ability to load and execute other programs, so there is nothing special about the shell.

One shell is loaded into the system for each user who logs on and for each window that a user starts, etc. (In actual fact, only one copy of the code needs to be loaded and this can be shared amongst all users; separate copies of the data structures and variables used by the shell are needed for each instance.)

12.1 A note on a Basic Input Output System

In general terms, a Basic Input Output System or BIOS is found on PC computers and others where the operating system kernel is not compiled (linked) specifically for that machine. In a UNIX machine, the kernel is updated by the technician who installs a new peripheral or otherwise changes the hardware, in order to deal with the changes. On a computer with a BIOS, the kernel is identical over many different computers from different manufacturers. The differences from one computer to another are adsorbed into the BIOS.

12.2 Command Line Interpreter or Shell

The shell or (CLI) provides access for humans to the computer. It is often based around a command line interface, but increasingly icon and mouse-based interfaces are used.

The most important function of the shell is to allow a user to easily select a program to run, and to ask the kernel to load it and run it. Other common functions of shells are:

- Collect together groups of characters from the input and pass them as arguments to the loaded program. Quotes can be used to help delimit the arguments

  ```
  $ calc  -print "3 * 5 * sin(4/5)"
  ```

- Support an environment of named shell variables, whose value can be inspected by running programs. A macro expansion facility enables the values of shell variables to be expanded where needed, using, for example, the dollars sign.

```
$ setenv CAT "on the mattress"
$ echo The cat sat $CAT
The cat sat on the mattress
```

- Enable a list of shell commands to be read from a file and executed as though they came from the keyboard. Such a file is called a shellscript or a .com file. In conjunction with the shell variables, and the argument substitution mechanism, this gives a simple interpretative language useful for system control.
- Detach, reattach and stop (interrupt) running programs.
- Redirect input and output streams which would normally go to the screen and keyboard, so that they connect with other programs or to files.

```
$
$
$  cat > testfile
This is
my best ever test file.
$  cat testfile
This is
my best ever test file.
$
$  cat testfile testfile > test2
$  wc testfile test2
     2         7        32 testfile
     4        14        64 test2
     6        21        96 total
$  cat testfile testfile | wc
     4        14        64
$  wc < testfile
     2         7        32
$
```

This has used two programs: *cat* concatenates files on its command line and standard input to its standard output: *wc* is a word counter which gives the number of lines, words and characters in the files given to it.

The search path for loadable programs may be changed

```
$ setenv PATH .:/bin:/usr/bin:/usr/ucb/bin
```

Here we have a colon separated list of four directories which will be searched for every command entered. Certain commands, such as setenv are built in to the shell. (Dot is the current directory, another variable kept on a per-shell basis).

13 Window Systems

System software to make a bit-mapped display and a mouse into a windowing system is often used with a multiprocessing computer.

Typically, one program running on the computer acts as a window manager and is responsible for defining the boundaries and layering of the windows. This program receives and interprets mouse commands which move and resize or iconify the windows. The windows themselves are essentially areas of memory where an application program can write.

Not all application programs wish to be involved with windows and mice: instead they would perfer to do text IO. This is where a virtual terminal program (such as xterm or dosbox) comes in.

13.1 Virtual Terminal Windows

Windows are often driven by a program that makes the window into a virtual terminal, exactly like the old Teletype. Each window has an associated input and output stream, connected to the process which owns it. Output from a process is displayed in the window. Input from the keyboard is directed to the process whose window currently has *focus*. The currently foucssed window is generally selected using the mouse pointer.

13.2 General Windows

In general, a a window is white space which the program that owns the window can draw any graphic in. To give a system an overall sense of consistency and to reduce the work of the application program writer, most systems provide a set of *widgets*, such as an interactive dialogue box, scroll bar or slider and button controls.

13.3 Directory view (ICON) program

Most windowing operating systems have a directory viewer program which renders the contents of a file system directory using icons (e.g. *finder* on the Mac, *filemanager* on MS Windows 3). The program deduces information about each file, using pattern matching of fields contained in the directory entry or by reading the start of the file, and selects an appropriate icon to use for it.

Basic filing system operations, such as rename, move, copy and erase are integrated with the program. The viewer program can typically launch application programs. A double click on the application's ICON being the normal method. This invokes the system loader. Sometimes it is possible to click on a data file generated by an application (such as a word processing document). The system will then find the actual application and load that with an argument set to the item clicked on.

14 System Resources

Resources are provided to an application program through the *application program interface* (API).

14.1 What does a program expect to see when loaded in ?

A loaded program expects to see:

In memory
1. It sees itself (ie it can jump to parts of itself, etc)
2. It sees its static data structures
3. One register (the stack pointer) loaded to point to a region of free memory which it will use as a stack.

In the processor
4. A program counter which is set to the program's start address
5. Other registers, free for use
6. Execution of its instructions at some rate
7. Floating point coprocessor, graphics accelerators, etc.

And also services
8. Standard input and output and error byte streams
9. Access to disk filing resources
10. Access to a heap allocation memory manager
11. I/O access to other devices, processors, printers, time and date

And some more advanced features, beyond the basic hardware abstraction
12. Current directory and other variables, such as host and user name
13. Authentication and directory services
14. The ability to fire up other programs and talk to them.

15. More than one processor - a threads system
16. Multiple address spaces - OS managed shared objects
17. Exclusion and locking services.

14.2 Storage Allocation - Dynamic and Static

A loaded program has static and dynamic requirements for RAM store.

14.2.1 Static Storage

Static variables are declared explicitly by the programmer at program writing time. Their quantity and size does not change during program execution.

14.2.2 Dynamic Storage

Dynamic storage is allocated (and deallocated) as program execution continues and the amount required by any particular run of a program may not be predictable: i.e. it may depend on the input data. The operating system (kernel) must provide memory on demand to hold the dynamically allocated items and arbitrate resource requests between all loaded programs.

Dynamic storage generally consists of the stack and the heap.

THE STACK

The stack is a region of memory used for local variables, subroutine return addresses and any other memory which is bound to be deallocated in the reverse order to allocation. Stack usage is generally proportional to the amount of recursion in the program.

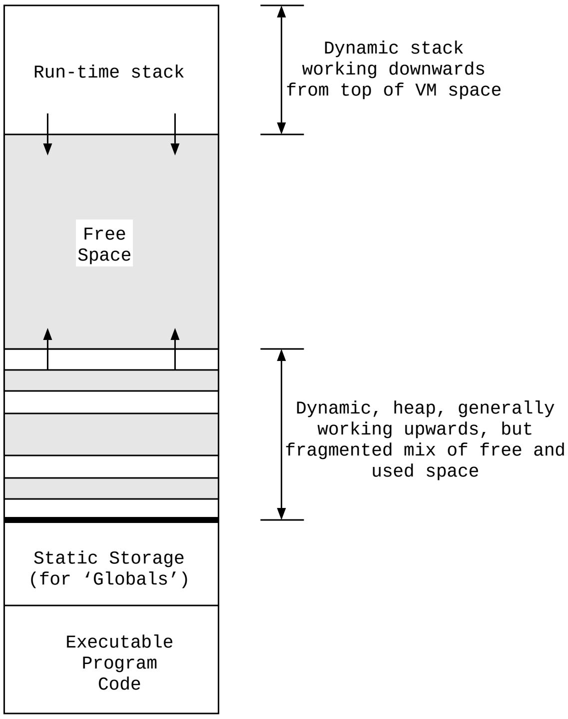

Figure 14.1: **Static and Dynamic Storage for a Process**

The Heap

The heap contains data structures dynamically allocated which cannot be held on the (or a) stack. Deallocation order is not related to allocation order, but is random. Heap store may become fragmented as areas are deallocated which are in between areas still in use.

The heap is used for structures of the C *struct* type allocated by malloc, which is like the Pascal/Modula *RECORD* and *OBJECT* constructors. In an interpreted language, such as ML or BASIC, defined values and functions are held in the heap.

(*Note:* do not confuse this heap with the structured heap used by heapsort.)

14.2.3 Examples of Dynamic Storage in ML

The values of the variable x can be kept on a stack but the list defined in paradise must be held on the heap.

```
- funn sumn x = if x>1 then x+sumn(x-1) else 1;
val sumn = fn : int -> int


- sumn 55;
val it = 1540 : int


- fun consn x = if x>1 then x::cons(x-1) else [1];
val consn = fn : int -> int list;

- consn 33;
val it = [33,32,31,30,29,28 ....] : int list;

- val paradise = consn 33;
val paradise = [33,32,31,30,29,28 ....] : int list;
```

15 Protection and Sharing

Access to a computer is physically controlled by keeping it in a locked room and requiring passwords for remote access. Protection is also required within the computer. This latter class is the subject of this section.

- When multiple users are logged on, a faulty or malicious program must not be able to seriously interfere with other users' activities.
- Individual programs desire to see a fixed set of resources each time they are run, including memory, processing and IO capability. The resources available should not significantly depend on what other programs are doing or using.
- The operating system kernel must protect itself from faulty or malicious programs.

A typical fault which could cause system upset is an out-of-bounds array access. A program which allocates a small buffer and then tries to load a large amount of data will access and corrupt something else instead.

The solution to this is to divide the system into separate *protection domains*. Typically each running process has its own protection domain and there is one further (or several) for the kernel. Transfer of control between protection domains is performed with special instructions - most importantly the trap.

15.1 Instruction Classes

As described in section 1.2.3, processors have a privileged flag (some have several). When the flag is clear, the instruction set and memory map of the processor are restricted.

```
Flag value     Name of mode      Instructions that may be executed
----------     ------------      --------------------------------------
  0             User mode         Normal instructions, including TRAP
  1             Kernel mode       All instructions.
```

To prevent application code from interfering with resources of other loaded programs, application code runs in *user mode* with the privilege flag clear. When the flag is set, the full instruction set of the computer becomes available, including instructions which access hardware resources and memory reserved for the kernel. The flag cannot be set by application code without using the trap instruction.

An attempt to execute a forbidden (privileged) instruction when in user mode causes a privilege violation interrupt, which is handled with appropriate code in the kernel.

15.1.1 Normal Instructions

Normal instructions are restricted to moving data between the processor registers and process memory and performing operations on the data, such as add, divide, shift etc. Branch and jump instructions are allowed. Normal instructions can be called *harmless* instructions.

15.1.2 Privileged Instructions

Privileged instructions are those which manipulate hardware I/O devices or have access to kernel memory and special kernel mode registers which do not form part of the user programming model. These include interrupt control registers and the VM translator configuration.

15.1.3 Emulated Instructions

To the programmer (or compiler), there is nothing special about an emulated instruction. It is included in the object code along with other normal instructions, but the instruction may not be part of the hardware instruction set of the computer.

Emulated instructions are not directly executed by the processor: instead they cause an interrupt. The kernel uses a sequence of normal instructions which perform the function of the emulated instruction, then returns control to the next instruction in the original sequence. Examples of these are

- use of a floating point register on a machine without a floating point register.
- use of graphics drawing instructions on a machine without a graphics accelerator chip
- use of an old instruction which was available on former models of a computer, but which is now rarely used (maybe always was) and so the price to performance ratio of the computer has been increased by deleting the hardware from the CPU and adding emulation software.

15.2 System calls to the OS Kernel - The TRAP

Trap and *Service Call* and *System Call* and *Software Interrupt* are equivalent names for the usual way in which a loaded program transfers control to the operating system in order to obtain service.

The operating system kernel is the one privileged program in the system. It has complete control of the system. The application code cannot and must not simply jump into kernel system code since:

1. The operating system cannot offer security if any application could jump in to it at any point.
2. The privileged mode flag (see below) will not be set and so the operating system will experience a privilege violation exception the first time it tries to execute a privileged instruction.
3. The areas of memory accessible by the kernel and a user application program are different.

The loaded program is not linked with the operating system, so direct reference by name to kernel routines is not possible. (Linking is studied in the final section of this course, but basically we are saying that a loaded program does not and should not know where in the OS memory, code for particular functions is located, so cannot go directly to it, the way it can other parts of itself).

The TRAP instruction transfers control to the kernel with the privilege flag set, but at a controlled (set of) entry point(s). The requested application function is normally identified using one of the registers according to an operating system specific service call protocol. For example,

> *'If register 1 contains the value 4 then the system call is to read data from the current input stream up to the number of bytes held in register 2 and store them in the memory pointed to by register 3.'*

Similar definitions will exist for about 100 or so values of register one, to have access to each of the kernel services. These definitions form the major part of the application program interface (API).

The operating system will make authentication checks that each request for service is valid. For example, if the specified destination address for the read information is outside the allocated memory of the requesting process, then an error is flagged. For such as severe error, the calling program would normally be stopped and control returned to the shell.

16 Multiprocessing

16.1 Notes on Process, Processor and Thread

A CPU or *processor* for short is the hardware unit which forms the heart of the computer. A physical processor only has one program counter and one set of user registers.

A process is an abstraction of a basic computer, consisting of one memory map and (normally) one thread of execution, corresponding to one set of CPU registers. In a time sharing computer, there are many processes, each with its own memory allocation. The VM system helps isolate the regions of memory. Under certain operating systems, each process can have more than one thread of execution.

A thread of execution is an abstraction of a processor. In a computer with one processor, this processor must partition its time between the threads, using *context swaps*. In a multi-processor computer, several threads may be executing in parallel on the different processors.

Each thread must normally have its own stack, and these must be managed by the kernel.

16.2 Context Swap

A *context swap* is the process of changing from one thread to another. This is needed whenever there are more threads than processors, which is the normal case in a time-sharing system. Each processor spends a period of time processing one thread and then moves on to the next. This is called *time slicing*.

To effect a context swap, the program counter and other user registers are saved in an area of OS memory specific to the currently running thread and values for the new thread are loaded from the new thread's save area. Flow of control then resumes at the point in the old thread's execution where it was left off.

In addition, a new layout of memory is typically required for the new thread, so the VM unit is reprogrammed by the kernel to suit the new thread before resumption.

In an operating system which supports more than one thread per process, we can have major and minor context swaps. A minor context swap is when the processor changes between handling threads in the same process and a major swap is when the old and new threads are in different processes. The memory map of the computer is unchanged under a minor swap, so the cost in kernel overheads is lower (i.e. no reconfiguration of the memory translator unit is required).

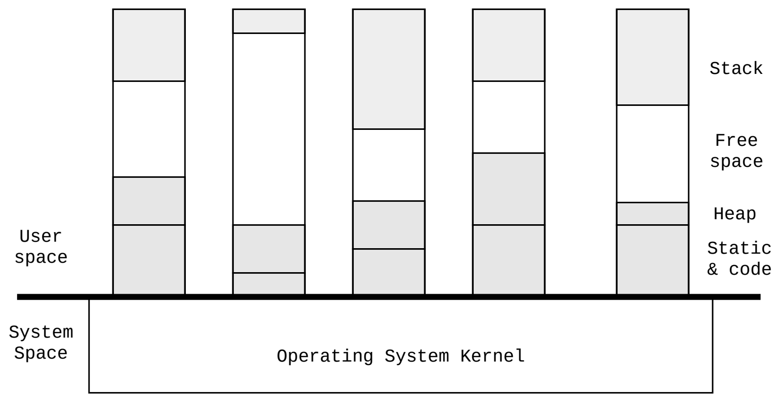

Figure 16.1: **Abstract Picture of Multiple VM spaces and Kernel**

16.3 Three states of a user process as managed by the Scheduler

The scheduler is a kernel component which decides which of the loaded user processes or threads to run next.

As mentioned in the last section, the contents of the processor registers for threads which are not running are stored in a kernel structure for that thread. This is the thread control block. There are other flags and values in the thread control block which help the scheduler decide which process to run next. These indicate whether there is any work for the thread to do, what priority should be assigned to that work, and whether the thread must wait for some other thread to finish something or release a locked resource.

There are three states a thread can be in:

1. **Running** A single active thread is in the running state (several on a multi-processor machine). No process is in the running state when the processor is executing context swap and scheduler software in the kernel, but this should only take a few percent of system time.
2. **Blocked** A blocked thread cannot be run because it has no work to do. A blocked thread is generally waiting for one or more of the following:
   - Waiting for a disk block to arrive
   - Waiting for a keyboard or network input
   - Waiting for input messages from other threads.
   - Waiting for a timer clock to expire
   - Waiting for a resource to be freed by another thread.
3. **Ready to run** All other threads are in the ready to run state.

The scheduler must make a decision about which thread to run. It uses rules such as

1. The thread is not blocked,
2. The thread has not been run for a while
3. Other general *ad hoc* rules, such as priority of one thread over another.

The scheduler is generally run after every system call or interrupt on the route to returning to user mode. This is because registers must be reloaded at this time in any case and because system calls and interrupts generally tend to move a thread from the blocked state to the ready to run state, or vice versa.

A *CPU-intensive program* is a program which processes for long periods without making I/O requests. It will tend to get *pre-empted* when it makes a page fault (i.e. requires a new VM organisation) or when a device being used by other programs makes an interrupt, or finally, by a system timer interrupt, used explicitly for time slicing.

16.4 Memory Maps

The physical memory map is partially occupied by random access memory. Other physical items, such as I/O devices and the bootstrap ROM are also mapped in. The physical memory contains the kernel, the kernel data space and many user address spaces. There need be no real structure to the contents of the physical memory - it all depends on how the VM unit is programmed.

The memory map of a single process contains a run-time stack, free space, a heap, the static data structures of the process and program code. The kernel memory map is disjoint from the user space addresses, so that when in kernel mode, the kernel can make access to a user process space at the same addresses the user address space will use. The I/O devices and other items do not appear in the user address space (they are not mapped by the VM unit) so user programs cannot access them directly.

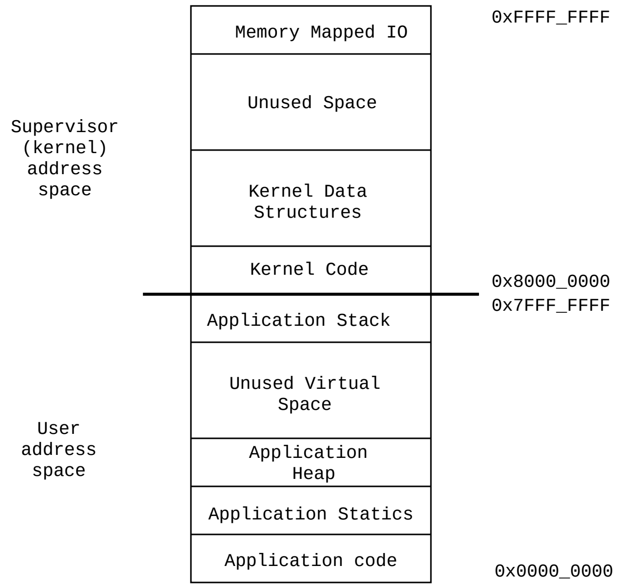

Figure 16.2: **User and Kernel Space Memory Map**

17 Input and Output

In a time sharing computer it is not sensible for loaded application programs to make direct access to hardware peripheral devices. Two main reasons for this are:

1. Access must be shared in a controlled way to prevent two or more programs trying to make one physical device do two things at once.
2. Individual devices vary in their details and a general purpose application program should be able to run on many computers without modification.
3. Many devices require considerable software to make them behave usefully (e.g. software to turn a bit-mapped display into an ASCII character output terminal), and it would be wasteful (and largely impractical[1]) to have a copy of this software as part of loadable program.

The loaded program uses system calls to request input and output services. As shown in figure 17.1, IO system calls may either be directed by the kernel directly to a 'raw' *device driver* or to a *virtual device* created by a layer of protocol or 'cooking' software.

Providing abstractions of devices helps software portability, since when the actual devices vary from one machine to another, the abstract devices can be the same. Application code can then be the same, reducing the number of versions which a vendor has to support. The differences between one computer and another are adsorbed in the kernel.

17.1 The Device Driver

The device driver is the section of software associated with a physical hardware input or output device which is specific to that device. The purpose of the device driver is to soak up device specific details and present the device to the rest of the system as one of the types of virtual device that the system expects.

For example, on a PC, different types of video display adapter cards can be used. A simple one will use software in the device driver to render the characters and flash the cursor. This is clearly a fair amount of software which uses memory and execution resources of the main 'host' processor. A more advanced 'intelligent' display adaptor might have its own processor within. The device driver then has very little work to do, simply passing the kernel requests on to the imbedded processor. However, in either case, the interface between the device driver and the kernel is the same.

The device driver is generally linked into the kernel (or BIOS), but some OS allow loadable device drivers. It must run in privileged mode since it has direct access to the hardware registers.

17.1.1 The interface between the device driver and the kernel

Systems try to divide IO devices into a few classes intended to encompass all known peripherals and all future inventions.

Kernel software access devices using an operating system specific set of access points, which are the same for all devices in that class. For example we might have

- start(dev)
- stop(dev)
- read(dev, buffer, length, blockno)
- write(dev, buffer, length, blockno)

[1] I say largely because the early Apple Macintosh and X-windows systems suffered form requiring large amounts of library software to be loaded with each application program.

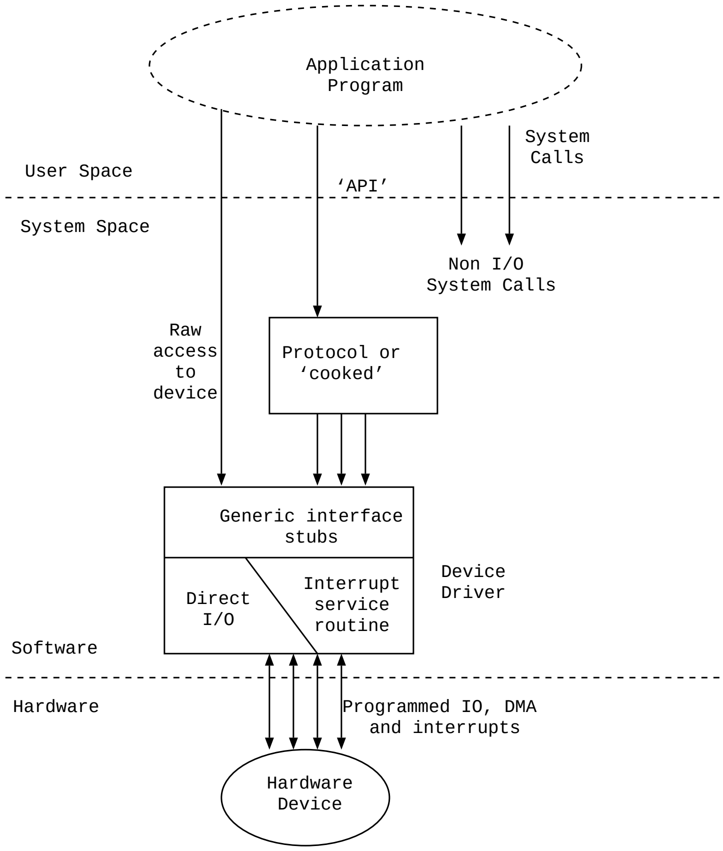

Figure 17.1: **Device access diagram**

- ioctl(dev, operation, argument)
- interrupt_service_routine(dev)

17.1.2 Block and Character devices

Unix divides its IO devices into Block and Character devices.

Consider a disk device - it has a number of platters, each with two surfaces and a number of tracks. Each track is formatted into sectors. A sector contains, typically, 512 bytes, and is the smallest unit of data that an interface to a disk drive will handle. A disk is therefore a block I/O device. The device driver will perform the logical to physical mapping discussed in section 8.3.

An ASCII terminal, however, generates individual keystrokes and individual character output. Many programs, such as a full-screen editor, wish to have access to individual keystrokes, and so the terminal is naturally a character I/O device.

17.2 Examples

Here we examine a few example IO devices and consider what functionality should be placed in the device driver, what else should be provided by operating system software and what should be left to an application. (In Part III of the course we see that, within the loaded application, functionality can either be generated by the application author or provided by linked library routines.)

17.2.1 Example: Terminal Input and Output

The device driver for a terminal is the archetypal character IO device.

The *cooked mode* for a terminal allows line-by-line data entry through which typing errors may be corrected. The kernel buffers up received input characters until a newline (carriage return) is received, before handing them on to the user program. The kernel allows the person using the terminal to type backspace to correct input. For output, the newline character is turned into the carriage return linefeed sequence. A *raw mode* is also available, for programs such as full-screen editors which wish to interpret keystrokes in more exotic ways.

17.2.2 Printer Output

A device driver for a printer normally turns whatever the printer hardware does into a character output device.

Other operating system functions which can be built on top of the device driver include

- Spooling (vital for multiuser printers)
- An ASCII to postscript convertor
- A charging or access control mechanism

17.2.3 Disk Access and Filing again

As mentioned, the device driver can adsorb physical details to give a linear array of disk blocks.

Most operating systems offer a complete filing system for the disk devices, implementing file names, directories, free space management and selective access denial for security.

17.2.4 Typical functions for file open, read, write and delete

As mentioned, the smallest unit readable or writable on the disk is a sector, but the systems software gives the programmer other options, such as the byte by byte interface shown here:

To create a file and write some bytes to it, we can use

```
FILE *s = fopen("/usr/users/djg/testfile", "w");
fprintf(s, "Hello World !\n");
fclose(s);
```

The 'file print formatted' library function (fprintf) puts each character of its argument to the output stream, byte by byte.

To read this file we can use

```
FILE *s = fopen("/usr/users/djg/testfile", "r");
while (feof(s)==0) { char c = getc(s); putchar(c); }
fclose(s);
```

where the while statement in line 2 copies each character from the file to the standard output through the character variable 'c' until the 'file end of file' library routine returns a non-zero value.

To delete the file we can use the remove library function (which turns straight into a system call in the run time system (as described in section 26.1))

```
int j = remove("/usr/users/djg/testfile");
if (j != 0) printf("Failed to remove file\n");
```

Here we print an error if the call to remove failed for any reason.

Note: Although these examples use the C language, dont worry if you are not yet familiar with C. The important aspect is that the operating system provides routines such as `remove` which can be called from your high level language to invoke operating system functions.

17.2.5 Network input and output

The functions of a device driver for a network are generally to soak up network-specific details and give the ability to send and receive a variable length block of data (the datagram). (This is essentially as described in section 9.4.) For an Asynchronous Transfer Mode (ATM) network, which sends data in 48 byte cells, this would involve segmentation and reassembly of the datagram into these cells.

The operating system protocol software built on top of the device driver can include a transport protocol (section 18.3) or a network filing system to give access to remote disk systems on other machines or file servers, as though they were part of the local file system.

18 Interprocess Communication

For many applications, such as remote login or database access, data needs to be sent between process spaces. These spaces may be on a common computer or on different computers over the network. It is helpful if the same operations can be used for both these situations and perhaps also for data communication between a process and the kernel or between a process and IO devices.

18.1 Communication Paradigms

18.1.1 File Communication

A simple form of communication is for one process to write a file which another has a look at. Communication can be through the contents of the file, the name of the file, or even the existence of the file.

18.1.2 Signals

In Unix, it is possible to send and receive *signals*. To receive signals, a process may register a *signal handler* by making a system call which identifies one of its own subroutines as a handler routine. To send a signal, a process makes a signal system call which causes the receiving process to effectively take an interrupt and run the signal handler routine that is has registered.

Signals are not normally available over the network.

18.1.3 Bytestream Communication

A major type of communication channel is the reliable bytestream. Data is written into the stream one byte at a time and is delivered (possibly after a varying amount of delay) at the receiver in the same order.

Invoking operating system functions for every byte moved is hopelessly inefficient for most applications but the abstraction is useful, especially for access to files, printers and (virtual) terminals. In practice, characters are bunched up in user space and handed in chunks to and from the kernel with the read and write system calls.

An example of bytestream communication is when we *pipe* the standard output of one process into the input of another - section 12.2.

18.1.4 Datagram Communication

LANs are considered *unreliable*, unlike disks and terminals, which are considered reliable. A LAN must offer some specific integrity to be useful. The level of service provided is *block integrity* (term invented by djg), which means that:

- A received block exactly matches one of the blocks sent at the source.
- Not all sent blocks are necessarily received (some sometimes get lost).
- You may receive the same block more than once.
- You may receive blocks out of order.
- You will not receive anyone else's data blocks.

18.1.5 Communication through Remote Procedure Call (RPC)

A remote procedure call mechanism gives a process the illusion that it can directly call subroutines on other computers. What actually happens is that a local *stub routine* on the source computer is provided and the process simply directly calls this. The stub routine copies its arguments into a datagram and sends this on the network to the destination computer, where a receiving process calls the real routine with the provided arguments. The result from the real routine (if needed) is then sent back to the source computer in another datagram. When the stub receives this, it returns to its caller.

The stub routine is typically generated by a macro-processor (section 20).

18.1.6 Communication through Shared Memory Objects

A novel form of communication is for many processes to have access to the same heap, or for a given process to access sections of the heaps of its colleague processes. This form of communication can be implemented by keeping a copy of the shared heap space on all cooperating computers and invoking help from the VM translator unit. Areas of shared heap are kept in an area of VM where write access is denied. Any process which tries to update the shared heap will generate a VM interrupt. The kernel software will then make the modification to the heap data and send messages on the network to update the heaps on the other computers.

18.2 A note on Reliability

Errors may be detected or undetected. Clearly we can do nothing about undetected errors. Machines increase the probability of detecting errors using parity bits, checksums and CRCs.

When we say something is treated as *reliable*, we mean that the chance of a (detected) error is sufficiently low that it is acceptable if the whole system must be rebooted every time such an error occurs. For unreliable devices, the software must cope with the error conditions and make another attempt, in the hope that the error does not recur.

18.3 A transport protocol for a bytestream

On top of a local-area network, a kernel typically offers a reliable block stream protocol, which gives the application program the impression that no blocks are lost or repeated in the network. It can do this by putting a sequence number at the start of each transmitted block, and checking the order of sequence numbers at the receiver. (These sequence numbers are stripped off again before the data is passed up to the application.) When there is a sequence error, duplicates are just ignored, but the gaps require special re-transmit messages to be sent. These can get lost and repeated as well, so care is needed in protocol design.

A system can also offer byte-by-byte access to the block stream, to give a bytestream.

Part III

Software Tools

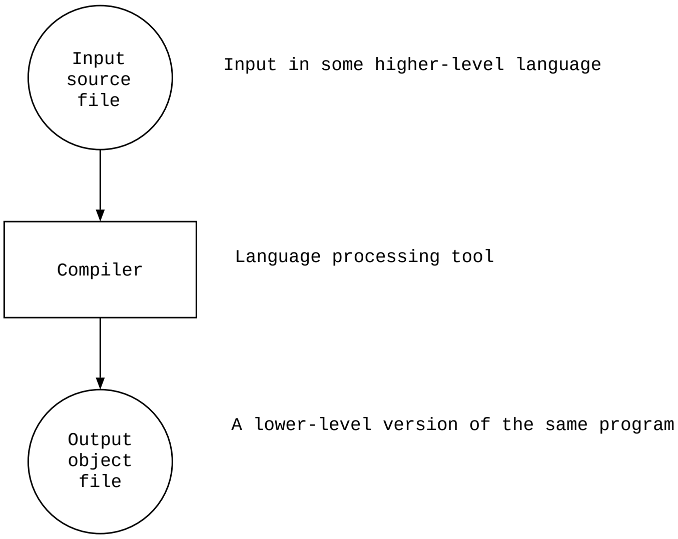

Figure 18.1: **Software Processing**

A concise view of software flow is:

1. A program is prepared using system software tools.
2. It is shipped to the customer or user, perhaps on a disk.
3. The disk is put in the target machine. The program expects various resources to be available in the target machine. These are provided by the resident system software.

How many different types of target machine will one object disk run on ? (We must consider both the interpreted and compiled cases.)

The Big Picture of software flow is at the end of this part of the notes, but perhaps worth looking at from time to time before we reach it. We will explore all aspects of it. Central aspects are the function of the link editor and the system call interface to the kernel.

In the methods of program preparation which we consider, a source file is first created with an editor (independent of the language tools) and then this file is fed through stages of language processing to end up with something that may be executed.

Sometimes several stages of compilation are needed:

```
----------    -----------------     -----------------
Compiler      Source Language       Object Language
----------    -----------------     -----------------
  C++           C++                   C
  cpp           C                     vanilla C
  c compiler    vanilla C             assembly language
  assembler     assembly language     machine code
----------    -----------------     -----------------
```

The higher level language is easier for a humans to write programs in than the lower level languages. High level languages have

- portability over many machines
- conciseness, leading to fewer keystrokes and fewer mistakes
- type checking, leading to fewer mistakes.

The lower level language is more machine specific, harder to understand and contains less checking.

A compiler which takes assembly language as input is called an assembler.

In a strongly-typed, high level language, the compiler will not accept inconsistent use of data. For example, adding a binary number to string is not generally valid:

```
let x = 4 + "Hello";
```

Whereas in low level languages, including assembler, where there is no type checking, the concept of what the bit patterns represent is lost and so no such checks can be made.

The loader will load any set of bits into memory (it assumes that they form a binary executable program) and jump to them, but they could be nonsence. The nonsence 'program' will generally cause one of

- privilege violation,
- memory access violation, or
- illegal instruction exception

after a short while, so the OS will stop it.

19 Program Preparation Methods

We consider four operations on a file containing software. The file may be:

1. Macro-processed.
2. Compiled
3. Interpreted.
4. Executed by hardware

Another important step is the use of a *link editor* to combine separately compiled files into a larger file.

By *macro-processing* we mean feeding the file through a program which modifies, expands or deletes sections of the file accorinding to a set of rules.

By *compilation* we mean conversion of the user program into the executable machine instruction code of the target processor. Each section of the user's program is only processed by the compiler once, regardless of how many times it will execute in loops when running. *The term compiling is also used when a section of code is transformed from one representation into a lower form, even if the lower form is not machine instruction code. This lower form may be interpreted.*

By *interpretation* we mean using another program, called an *interpreter*, to scan a representation of the user's program. Each loop of a program's execution requires the body of the loop in the representation to be rescanned. For interpretation, the data must be present at the time of interpretation and the answer is then produced. The interpreter itself is a program and it may either be interpreted (by a lower-level interpreter) or compiled (the more normal case).

By *execution* we mean loading the file into memory and jumping to it. The processor of a computer can only execute machine code instructions. Any particular program must either be compiled all of the way down into machine code or else compiled into a language which is suitable for interpretation (if the source language is not suitable for efficient interpretation).

20 Preprocessing and Macro Expansion

A program preparation step in common use is macro programming. The C language is defined to be macroprocessed before compilation, using the C preprocessor.

In a macro processor, the input consists of a mixture of

- Object language extended with macro invocations
- Macro definitions
- Conditional text inclusion rules

and the output is pure object language with no macro language left. A macroprocessor can perhaps be classed as a compiler, since it produces a lower level form at its output, but the unique feature of macro processors is that the input is very like the output.

Macroprocessing and textual inclusion form an important part of the big picture of software flow, since inclusion of the same set of macro definitions into many modules, which are finally linked together, introduces common ground between the modules and can avoid inconsistencies.

Sometimes the output is fed back into the input until *closure* occurs, meaning that no further changes take place.

An example of a part of a header file which is included in several source files:

```
#define   RED   1
#define   BLUE  2
#define   GREEN 3
#define   BROWN 4
```

The hash-define macro directive instructs the macro processor to change all instances of the first argument (a colour) with the second argument (a number). Suppose these directives were in a file called 'colours.h' and every source file which is to be used in a program included the line

```
#include "colours.h"
```

near the start, and all of the source files were passed through the macroprocessor before compilation, then all instances of the colours within the files (from the point of the include onwards) would be replaced by their numbers.

This would be useful if the language that is being compiled understands numbers (most do) but would consider a colour a syntax error (most do).

The hash include macro directive causes textual inclusion of the referenced file. That is, it is as though the referenced file were copied out in the source file at the point of the include.

20.1 Schema Consistency

Macro-generated code can be used in order to give semantic consistency to a program. Sometimes a high-level language is not high enough for its native mechanisms to assure a programmer that he is not going to make certain types of bug.

For instance, in database programming, it might be quite easy to make the database inconsistent. If the database contains financial information, then it is generally appropriate that money added to one account is subtracted from another. If the code to do these two operations is in separate statements of the native high-level language (e.g. COBOL), then one or other can be forgotten. However, if the programmer uses a macro to move sums of money, then provided the macro is correct, money can never be created or destroyed. The point is, the macro only needs be written once.

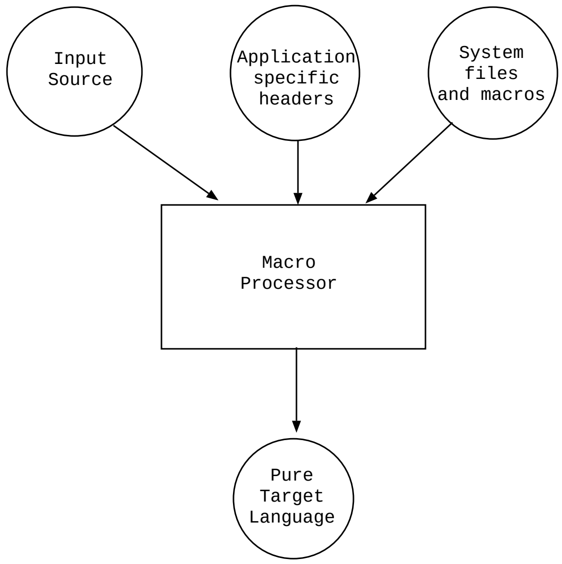

Figure 20.1: **A Macro-Processor at work**

21 Program Compilation

A compiler generally operates in several phases, each one generating a data structure which is the input to the next phase. Typical phases are

1. Macro Preprocessing (optional)
2. Comment stripping and Lexical Analysis
3. Syntax Analysis
4. Pseudo Code Generation
5. Optimisation
6. Target code generation
7. Optimisation
8. Perform internal linking of jump statements and other references
9. Write target code to output file.

Here is a simple ML program:

```
- fun sumn x = if x >= 10 then x+sumn(x-1) else 50;
val sumn = fn : int -> int;
```

The white space characters, which are spaces, tabs and newlines are removed by the lexical analyser, after they have served their job of delimiting input tokens

Here are the lexical tokens for the simple ML program.

```
Input text     Token      Auxiliary Value
  fun            builtin   fun
  sumn           userid    id0001
  x              userid    id0002
  =              operator  equal
  if             builtin   if
  x              userid    id0002
  >=             operator  greater_or_equal
  1              number    1
  then           builtin   then
  x              userid    id0002
  +              operator  plus
  sumn           userid    id0001
  (              leftparen
  x              userid    id0002
  -              operator  minus
  1              number    1
  )              rightparen
  else           builtin   else
  50             number    50
  ;              semicolon
```

The lexical analyser recognises the logical extent of the input token and classifies it according to whether it is a number, a user identifier, or a group of characters making up a composite symbol such as '>='. Otherwise the input token is a valid single character token or an error. User defined input symbols are just given numbers (or pointer address) so that the rest of the compiler does not need to make string comparisons.

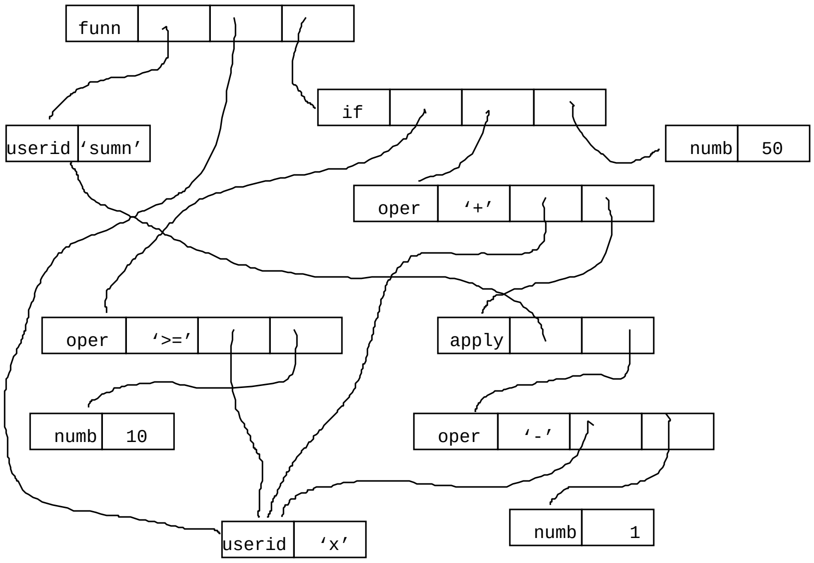

Figure 21.1: **A fragment of syntax tree**

The syntax analyser groups together the lexical tokens into valid constructs of the language or else flags a syntax error. This process is called *parsing*. The syntax analyser ends up building a parse tree of the input source. The operator application precedence of the input determines which tree nodes are parents of which others.

Figure 21.1 is part of a suitable tree for the example program: The syntax analyser, which built the tree, has not included tokens which are merely *syntactic sugaring*, such as parenthesis and the '=' in the syntax for function definition.

Note: In fact the syntax tree shown is not sufficient for the ML language since alternative signatures to a function are not possible.

Any parts of the tree which can be reduced to constant expressions are so reduced.

The tree can be interpreted directly or turned into compiled object code using a code generator for the target language and machine.

When high-level language programs are to be compiled for a new target computer, a new code generation part of the compiler is required, but that is (with luck) all.

Section 25.4 gives an example of the input and output from a compiler.

22 Assembly Language Again

Assembly language is the lowest form of symbolic programming. The next lowest form of programming would be to consult binary opcodes in the manual (or from memory if one has done sufficient of this in the past) and type their values into the machine in hexadecimal or other number base.

In section 3.7.2 we have already seen an example assembly language for a simple 8 bit processor. There is a different assembly language for each family of processors.

Examples are

- 68000 68010 68020 68030 68040
- Mips R2000, R3000, R4000
- IBM system 360, System 370, ... the current 3090
- 8088 8086 80186 80286 iapx186 286 386 (486) ..

Within one family of processors, an assembly language for one processor will mostly compile for another but there is no portability between families.

Lower rated machines within a family may not have all of the instructions of the later machines, but this can often be overcome by emulation. This is as described earlier in section 15.1.3.

Assembly language needs to be used

- When speed of execution is very important
- When a machine-specific function which is not supported in a (non-machine-specific) high level language needs to be executed.

Assembly language is converted to machine code by a compiler program called the assembler.

The source file contains:

- Common machine assembly language instructions
- Privileged machine assembly instructions (e.g. for IO)
- Operating system calls (TRAPS)
- Commands to the assembler (directives)
- Definitions for uninitialised data storage regions
- Definitions for initilialised data storage regions
- Comments

22.1 The Structure of Assembly Language

Assembly language is always line based, with each line being converted by the assembler into one machine instruction. The format of a line of assembly language has the following four fields:

```
LABEL          OP             OPERANDS         COMMENTS


Optional       Instruction    Source or        Comments are
label as a     symbolic       destinations     ignored by
target for     mnemonic       for data to      the assembler and
jumps          (what to do)   be used.         are sometimes left
                                               out by lazy programmers.
```

22.2 Example: Assembly on an Motorola 6800

This example is in Motorola 6800 Assembly Language. It is a simple program which writes 'Hello' fifteen times to the output device by making traps into a machine code monitor (loaded elsewhere in memory).

Here is the source file:

```
#include "hdr104.s63"  ; Textual include of system call and ASCII defines.

     TITLE "A Demonstration of Assembly Programming"

; All of these two lines is a comment, since they
; start with a semicolon.

; EQU is the equate directive, for defining constants
LoadAddress EQU $400

; The presence of the origin statement (ORG) causes
; an absolute output file.
      ORG  LoadAddress

      LDAB =15     ; Load the value 15
; This system call writes a null terminated ASCII string
Loop  TRAP T.WrStr
      DEFB 'Hello World!',CR,LF,0
      DECB
      BNE Loop

      TRAP 0       ; Trap zero is exit to os

; Notes:
;    A dollars sign introduces a hex number.
;    The equals sign introduces immediate addressing.

      END
```

The assembler mnemonics used in the example are

Mnemonic	Class	Purpose
TITLE	Directive	Gives some text to put on the listing
EQU	Directive	Assign a value to an assembler variable
ORG	Directive	Set the origin for program loading
LDAB	Instruction	Load a value into register B
TRAP	Instruction	Trap
DEFB	Directive	Define initialised byte values
DECB	Instruction	Subtract one from register B
BNE	Instruction	Branch if not equal to zero (Z flag clear)
END	Directive	End

This source file is fed into the assembler and the assembler produces the listing and object files shown on the next page.

Certain of the symbols used in the program have been defined using equates placed in the library header file "mon140.s63", which has been textually included in the assembly. An example of an equate is the definition for LoadAddress.

LISTING FILE

```
224                    title "A Demonstration of Assembly Programming"
225
226              ; all of this line is a comment, since it
227              ; started with a semicolon.
228
```

```
229                ; equ is the equate directive, for defining constants
230 =00000400          loadaddress equ $400
231
232                ; the presence of the origin statement (org) causes
233                ; an absolute output file.
234 =00000400                org  loadaddress
235
236 000400 C60F              ldab =15     ; load the value 15
237                ; this system call writes a null terminated ASCII string
238 000402 3F04        loop  trap t.wrstr
239 000404 48656C6C          defb 'Hello World!',cr,lf,0
        6F20576F726C64210D0A00
240 000413 5A                decb
241 000414 26EC              bne loop
242
243 000416 3F00              trap 0       ; trap zero is exit to os
244
245                ; notes:
246                ;    a dollars sign introduces a hex number.
247                ;    the equals sign introduces immediate addressing.
248
249                      end

ERRORS = 0,   SRC CARDS READ = 249
```

Object file

```
-T00000018
:02040000C60F
:020402003F04
:0F04040048656C6C6F20576F726C64210D0A00
:010413005A
:0204140026EC
:020416003F00
```

The format of a line of object code here is

- a colon
- two digits giving the number of object bytes
- four digits giving the load address
- the object bytes themselves.

The included header file does not generate any code, it just introduces labels. Note that the result of assembling the included file(s) has not been listed in the listing file.

The assembler has ignored upper and lower case outside quotes.

23 Interpreted Software

In the interpreted case, the language processor, the program and the data all have to be present in memory at once. A form of editor is also normally present, so that interactive operation is achieved.

Reasons and places for using interpreters:

- Interpreted languages are often interactive allowing source development to be integrated with testing and running.
- Portability.
- Secure execution environment.
- When programs are only likely to be run once (e.g. database lookup), or other cases where the cost of compilation is too great.
- Highly compact source code invoking a set of common or standard operations.
- Top level system control.
- 4th generation languages or other systems where compiler technology is not advanced.

Interpretation is the most demanding of memory since everything is loaded at once - code, data, interpreter, editor, tools, library.

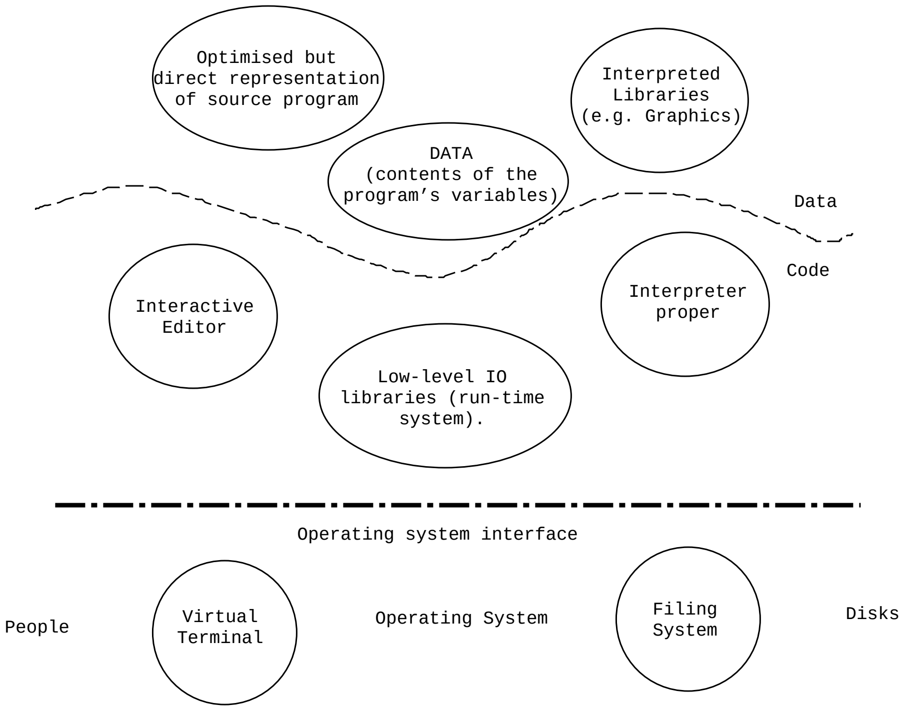

Figure 23.1: **An interpreter and its data structures**

24 Loading and Execution by Hardware

Only *executable* files may be loaded and executed directly by the processor hardware. Executable files are clearly machine specific, whereas the high level source form of the programs, before compilation, can generally be compiled for many different computers and processors.

An executable file generally consists of a collection of software modules, known as Common Object Format files (COF files) or binaries, which have been linked together using a link editor, but it may be just a small object file generated by a single run of the assembler.

The loader is the part of the operating system which copies an executable file into memory, arranging the various regions of program and data within the address space. An important function of the loader is *relocation*.

24.0.1 Absolute and Relocatable Addressing

Every item in memory that is loaded from an executable file has a memory address given by

$$\text{Address in memory} = A + B$$

where

A = Offset of item from start of file
B = Base load point of the file.

All references to an item in memory must perform this sum to gain access, otherwise they will miss and get something else instead. There are several approaches:

1. **Relocatable Code.** Most processor instruction sets include a relative addressing mode (section 3.7.1). In relative addressing, the operand in the instruction specifies the offset of the target data from the current instruction. This will be constant regardless of the base load point, so the program may be validly loaded at any address and is 'relocatable'. The addition is done by the processor in hardware for each access.

2. **Loader Relocated Code.** An absolute addressing mode specifies the absolute address of the target. The interpretation of an absolute address is independent of the location where the instruction is executed. Code using absolute addressing for access to data structures loaded with the program is not relocatable and so must be accompanied with a list of places which need to be modified to correct for the load point. This relocation information usually consists of a list of offsets from the file start to which the base load address needs to be added. The system loader relocates the code before starting execution. (A similar approach is usually used by link editors when stacking COF files on top of each other.)

3. **Fixed Load Address Code** A program which is only ever to be loaded at one address in memory does not need relocation, but can be compiled/assembled directly for that address. In UNIX all loaded programs start at address 0. Since the system has virtual memory, many programs may feel they are loaded at address zero at any one time.

25 Modular Compilation

25.1 Benefits of a Modular Approach to Programming

When we split a large program into several smaller modules, we experience several potential benefits:

1. A modular approach gives a simple way to
   - use code from different programmers or supliers in one program
   - use different programming languages for different parts of the program.
2. Well defined interfaces between modules help reduce the quantity of code which needs to be considered by a human at one time.
3. Re-use of previous modules - leads to cost savings.
4. Compilation time is reduced, since only the modified module needs to be recompiled.

The cost of a compilation, linking or assembly can be expressed as a polynomial function of the input source size s.

```
                                 2
 Time cost (s)  = A  +   B s   +  C s  + (higher terms)
```

A is the time to invoke the compiler and other fixed overheads.

B is a coefficient of proportionality for a linear pass through the source code.

C is a coefficient resulting from identifier look-up. Every identifier must be matched against a table of previously encountered identifiers.

The term of s^2 dominates for large programs. If the program is split into two smaller modules, the compilation time for both of them summed is lower than the one larger compilation.

In practice, we might only compile one module after a change and then use the linker to produce a new executable, giving even greater savings of time.

25.2 Common object format (COF)

The Common Object Format files (COF file) for a module contains

D Debugging information

I Initialised static data area

G Global symbols exported directory

E External references needed directory

R Relocation information

O Object Code Proper.

In figure 25.1, three binaries are combined to form an executable. The entry point of the executable will be the start of module 1, since it was placed first. Module 2 will have been relocated by the length of module 1 and module 3 by the sum of the first two. All of the references in E1 must have been found in either G2 or G3, and similarly for the other two modules.

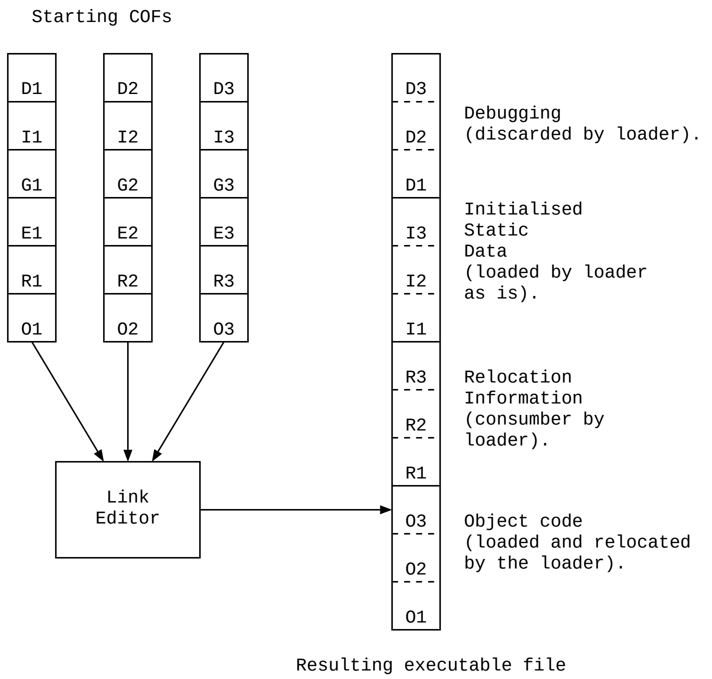

Figure 25.1: **Picture of 3 COFs and a link editor**

25.3 The Link Editor

A collection of Common Object Format files (COF files) are linked together with a link editor (also known as a *linker*) to produce an executable file. All of the COF files have the same format (hence the name) regardless of what high level language they were written in to start with.

Each COF file forms part of the desired final executable and it is the job of the link editor to stitch them together. There are two main functions of the link editor:

1. The object codes are *relocated* so that they form a consecutive set (see page on relocatable code).
2. External references in each file are resolved using a symbol table which is created from the concatenated set of exported global symbols from all of the files.

The output file is one of the following:

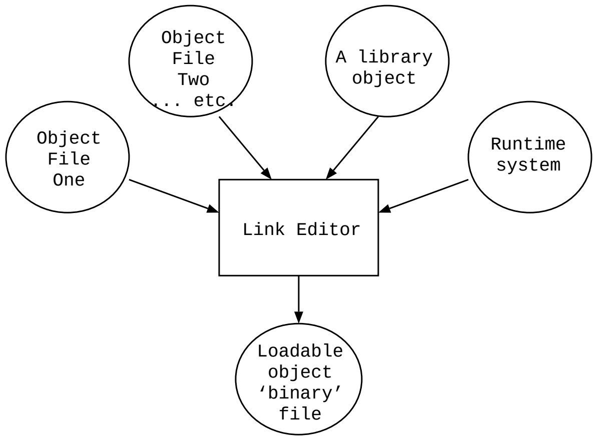

Figure 25.2: **Linking**

- An absolute loadable image (ie suitable for a boot ROM etc)
- A relocatable image which is relocated at load time
- An *archive*, which is a new file in COF form, suitable for linking with further COF modules in the future.

The format of the output file is selected with command line flag options to the link editor.

Another variation on the output produced by a linker is a stripped executable which has no D sections. We will not be able to use the symbolic features of the system debugger, owing to the missing D sections, but the executable file will be smaller and more secure against reverse engineering by malicious competitors.

A COF file is relocatable, meaning it can be loaded anywhere in memory. Binary executable files are not always relocatable, as described next.

25.4 An example of compilation into a COF

Here is a simple source file in the language *C*

```
/* test.c */
void writes(char *p)
{
  while (*p) putchar(*p++);
}

int main(int argc, char **argv)
{
  if (argc == 2) writes("Second arg is %s\n");
  return argc;
}
/* end of test.c */
```

And here is the output from a C compiler in assembly language. This compiler has generated code for the Motorola 68000 microprocessor. Downing and Woodhams describe the 68000 in fair detail. (*Note:* you are not expected to have specific knowledge of a processor architecture for this course.)

```
; cbg closure c compiler 'test.c'
divzerotrap extern
divzero jmp divzerotrap

;
;void writes(char *p)
;{
           data '_writes'
_writes    global
;  while (*p) putchar(*p++);
           exg a6,a0
           move.l a0,(a6)
           move.l a1,-4(a6)
Y10        move.l -8(A6),A0
           tst.B (A0)
           beq Y11
_putchar   extern
           move.L -8(A6),A0
           addq.L =1,-8(A6)
           move.B (A0),D0
           and.l =$FF,D0
           move.L D0,-20(A6)
           lea.l -12(a6),a0
           lea.l Y12(pc),a1
           jmp _putchar
```

```
Y12
          bra Y10
Y11
;}
;int main(int argc, char **argv)
          move.l -4(a6),a0
          move.l (a6),a6
          jmp (a0)
;{
          data '_main'
_main     global
;  if (argc == 2) writes("Second arg is %s\n");
          exg a6,a0
          move.l a0,(a6)
          move.l a1,-4(a6)
          cmpi.L =2,-8(A6)
          bne Y13
          move.L =Y14,-24(A6)
          lea.l -16(a6),a0
          lea.l Y15(pc),a1
          jmp _writes
Y15
;  return argc;
Y13       move.L -8(A6),D0
          move.l -4(a6),a0
          move.l (a6),a6
          jmp (a0)
;}
;
Y14       data 'Second arg is %s',10,0
```

The compiler has listed the source file as comments in the object file. A semicolon is used in this assembly language to denote that the whole line is a comment.

The programmer's identifiers have had an underscore prefixed on to them, so that they cannot clash with those in any of the system COFs.

Here is what happens if we assemble it:

```
** CBG 68k assembler job: test.s
1                                          ; cbg closure c compiler 'test.c'
2               DIVZEROTRAP                Extern
3 000000 4EF90000   DIVZERO                    Jmp divzerotrap
         0000
4
5                                          ;
6                                          ;void writes(char *p)
7                                          ;{
8
9
10 000006 5F777269                              Data '_writes'
         746573
11               _WRITES                    Global
12                                          ;  while (*p) putchar(*p++);
13 00000E CD48                                  Exg a6,a0
14 000010 2C88                                  Move.l a0,(a6)
15 000012 2D49FFFC                              Move.l a1,-4(a6)
16               Y10
17 000016 206EFFF8                              Move.l -8(A6),A0
18 00001A 4A10                                  Tst.B (A0)
19 00001C 67000026                              Beq Y11
20               _PUTCHAR                   Extern
22 000020 206EFFF8                              Move.L -8(A6),A0
```

```
23 000024 52AEFFF8                          Addq.L =1,-8(A6)
24 000028 1010                              Move.B (A0),D0
25 00002A C0BC0000                          And.l =$FF,D0
          00FF
26 000030 2D40FFEC                          Move.L D0,-20(A6)
27 000034 41EEFFF4                          Lea.l -12(a6),a0
28 000038 43FA0008                          Lea.l Y12(pc),a1
29 00003C 4EF90000                          Jmp _putchar
          0000
30                 Y12
31 000042 60D2                              Bra Y10
32                 Y11
33                                      ;}
35                                      ;int main(int argc, char **argv)
36 000044 206EFFFC                          Move.l -4(a6),a0
37 000048 2C56                              Move.l (a6),a6
38 00004A 4ED0                              Jmp (a0)
39                                      ;{
40
41
42 00004C 5F6D6169                          Data '_main'
          6E
43                 _MAIN                Global
44                                      ;  if (argc == 2) writes("Second arg is %s\n");
45 000052 CD48                              Exg a6,a0
46 000054 2C88                              Move.l a0,(a6)
47 000056 2D49FFFC                          Move.l a1,-4(a6)
48 00005A 0CAE0000                          Cmpi.L =2,-8(A6)
          0002FFF8
49 000062 66000018                          Bne Y13
50 000066 2D7C0000                          Move.L =Y14,-24(A6)
          0088FFE8
51 00006E 41EEFFF0                          Lea.l -16(a6),a0
52 000072 43FA0008                          Lea.l Y15(pc),a1
53 000076 4EF90000                          Jmp _writes
          000E
54                 Y15
55                                      ;  return argc;
56                 Y13
57 00007C 202EFFF8                          Move.L -8(A6),D0
58 000080 206EFFFC                          Move.l -4(a6),a0
59 000084 2C56                              Move.l (a6),a6
60 000086 4ED0                              Jmp (a0)
61                                      ;}
62                                      ;
63                                      ;
64 000088 5365636F   Y14                    Data 'Second arg is %s',10,0
          6E642061726720697320257300A00


ERRORS = 0,   SRC CARDS READ = 65
```

And here is the COF file (listed in hex)

```
$$ CBG 68k assembler job: test.s
-Global _WRITES r0000000E
-Global _MAIN r00000052
-T000000AC
4EF900000000 5F77726974657300 CD48 2C88 2D49FFFC
206EFFF8 4A10 67000026 206EFFF8 52AEFFF8 1010
C0BC000000FF 2D40FFEC 41EEFFF4 43FA0008 4EF900000000
60D2 206EFFFC 2C56 4ED0 5F6D61696E00 CD48 2C88
2D49FFFC 0CAE00000002FFF8 66000018 2D7C00000088FFE8
```

```
41EEFFF0 43FA0008 4EF90000000E 202EFFF8 206EFFFC
2C56 4ED0 5365636F6E6420617267206973202573 0A00
-Relocation
>Y14 00000068
>_WRITES 00000078
-Externals
>_PUTCHAR 0000003E
>DIVZEROTRAP 00000002
-End
```

26 The run time system - a fragment

The run time system is COF which is linked with every program generated by a given compiler as one of the modules forming its final executable form.

The run time system is the vital link between a compiler and the operating system for which it is producing code. The run time system contains entry and exit code to join the application to the operating system and system call *stubs* for every system call in use.

The run time system also provides code which the compiler assumes is present to which it references in its generated object code. Examples are

- system startoff code
- long division routines
- other maths libraries
- divide by zero error message
- other runtime errors and exception handlers
- heap storage allocator

26.1 System Call Stubs

A compiler for a high level language will not generate operating system specific instructions, particularly, it will not generate system calls. The run time system must therefore be written in assembler.

To access a system call, the user's programs must call library stub routines within the run time system. These contain the operating system specific trap codes and this leaves the compiler free of dependencies on one particular operating system in the code it generates.

Here is an example minimal runtime system to support the 'Hello World' program in the previous example.

```
; Here is the {\em start off} code which forms the entry
; point of all programs that get linked.
_main      extern               ;
           lea   X100(pc),a1    ; Put RLA in a1
           jmp _main            ; Jump to user's routine
x100       trap #0              ; System call to exit user space


_putchar   global               ; A routine globally available
           move.b -9(a6),d0     ; Transfer arg from stack to a register
           trap #25             ; This is the putchar system call
           jmp (a1)             ; Return link address in a1
```

Two parts of the runtime system are shown, one is the entry and exit code and the other is the stub routine for the putchar output routine. In this example, register a1 always contains the return address for a routine and operating system functions are selected using an argument to the trap instruction, rather than loading a number into a register (as described earlier).

A full runtime system may export a hundred or so global symbols such as 'putchar' and use a similar number of system calls. The mapping between the provided routines and the system calls may be one-to-many or many-to-one depending on the operating system.

(*Note:* putchar is a preprocessed macro in most C systems, but here we cut out several layers of glue I/O routines and turn it directly in to a system call with index 25.)

26.2 When to use a particular route to execution?

1. Compilation.

- Compilation is the 'default' method
- Fastest and most economical at execution time
- Security of code - source files need not be distributed
- Portability of high level source
- Type checking and formal methods filter bugs out but lead to no run time overhead.

2. Assembly and Machine Code Execution.

- All programming routes end up at direct execution
- Hand crafted assembler is often faster than compiler generated code
- Compilers cannot generate instructions which are highly machine specific since such functions are not represented in a portable source language.
- Assembler language is machine specific and not portable.

3. Interpreted Execution.

- Interpreted languages are often interactive allowing source development to be integrated with testing and running.
- Portability again.
- When programs are only likely to be run once (e.g. database lookup), or other cases where the cost of compilation is too great.
- 4th generation languages or other systems where compiler technology is not advanced.

Interpretation is the most demanding of memory since everything is loaded at once - code, data, interpreter, editor, tools, library.

4. Macro Preprocessing.

- To add new features to an existing language
- Ensure integrity of generated code using schemas
- For conciseness and therefore elimination of bugs.

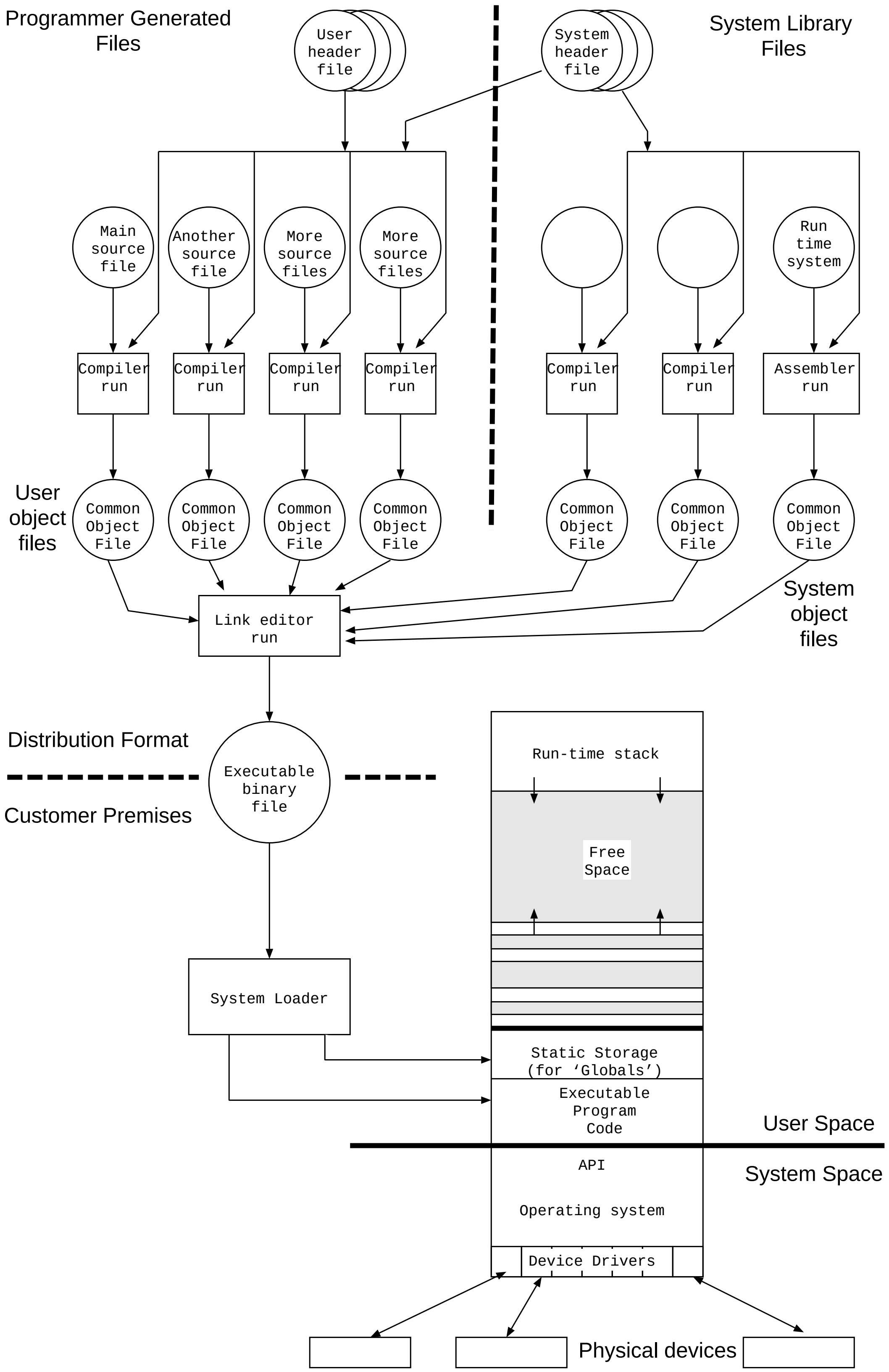

Figure 26.1: **Big picture of Software Flow**

System Design
Questions to help with revision.

General Questions

1. Systems software provides layers of abstraction which help us with concealment of detail and portability of programs. Describe a set of layers leading from the highest level language to the lowest level instruction execution.

2. How does the software in your CD player do and how did it get there ?

First third of the course - Computer Architecture

3. In a processor, explain the function of general purpose and special purpose registers. What are the advantages and disadvantages of having many registers.

4. What type of supercomputer could you easily make using off-the-shelf, everyday microprocessors ?

5. Describe six or seven separate physical areas of memory found in a typical computer.

6. What happens when the reset switch of a computer is released? Describe typical steps from reset to loading of a user application program.

Second third of the course - Resident Software

7. At the level of the loaded binary program, what similarities are there between a program written for an early single-user computer and a program written to run on a modern time-sharing computer?

8. There are two different device drivers which are used with the dumb and intelligent versions of the same peripheral (e.g. a PC display adaptor/controller). How will they differ and how similar will they be ? What are the advantages and possibilities of having a single device driver which can cope with both models of peripheral.

9. How may one user program be protected from another and the kernel be protected from all of them?

10. Define a process scheduler for a time-sharing system. How does it cope with *CPU intensive* programs ?

11. What is a kernel and what is a shell? What do icon and mouse based user interface have in common with command line based shells?

Third third of the course - Program Preparation

12. What is the purpose of the link editor? Describe the contents of typical common object format files and executable binary files.

13. Invent some typical lines of assembly language and describe what they do. Illustrate the contents of the common object format file resulting from your assembly. Is your code relocatable?

14. When should assembly language be used? Why should it otherwise be avoided?

15. What are the advantages of dividing a program into many modules which are later combined?

16. Compare compilation and interpretation as programming methods? What other execution methods are there?

17. How can one program be made available on a many computers of 1. different types and 2. different models ?

18. What communication resources might a program expect when chatting to another program a) over a network b) in the same machine? Should they be different?