CS311
Operating Systems

Hugh Anderson
Maths and Computing Science, USP
anderson_h@usp.ac.fj
Preface

The FOLDOC dictionary of computing defines an operating system as:

“The low-level software which schedules tasks, allocates storage, handles the interface to peripheral hardware and presents a default interface to the user when no application program is running.”

“The OS may be split into a kernel which is always present and various system programs which use facilities provided by the kernel to perform higher-level housekeeping tasks, often acting as servers in a client-server relationship.”

“Some would include a graphical user interface and window system as part of the OS, others would not.”

“The facilities an operating system provides and its general design philosophy exert an extremely strong influence on programming style and on the technical cultures that grow up around the machines on which it runs.”

An underlying view of this course is that the functions of an operating system are those that were found to be necessary.

---

Contents

1 Introduction ........................................................................................................... 1
   1.1 Re-ordering jobs ......................................................................................... 2
   1.2 Spooling ...................................................................................................... 3
       1.2.1 Spooling with hardware .................................................................... 3
       1.2.2 Spooling with software .................................................................... 3
   1.3 Job control languages ................................................................................. 6
   1.4 Core dumps ................................................................................................. 7
   1.5 Libraries ..................................................................................................... 7
   1.6 Time sharing ............................................................................................... 8

2 System Functions: .................................................................................................. 11
   2.1 User interfaces ............................................................................................ 11
   2.2 Resource management ............................................................................... 13
   2.3 Synchronisation ......................................................................................... 13
   2.4 Protection: ................................................................................................. 14

3 OS architectures .................................................................................................... 15
   3.1 Monolithic ................................................................................................. 16
   3.2 Set of system calls ..................................................................................... 16
   3.3 Heirarchy of system calls ........................................................................... 17
   3.4 Distributed operating system ..................................................................... 18
## 4 Multiple processes

4.1 Simple process calls .................................. 19
4.2 Operations on processes .................................. 20
4.3 Data structures for processes ............................ 21
4.4 Scheduling methods .................................. 22
    4.4.1 Preemptive and non-preemptive scheduling ................. 22
4.5 Sample calls .................................. 23
    4.5.1 Terminate process - exit() .................. 23
    4.5.2 Execute process - exec() and fork() ............. 24
    4.5.3 Wait on event - pause() .......................... 25
    4.5.4 Signal event - sigsend() .................. 25

## 5 File systems

5.1 File types .................................. 27
    5.1.1 Stream of characters vs structured .................. 27
    5.1.2 Names .................................. 28
    5.1.3 Links and aliases .................................. 28
5.2 Redirection .................................. 28
    5.2.1 Stdin, stdout and stderr .......................... 29
    5.2.2 Pipes .................................. 29
5.3 File system architectures .................................. 30
    5.3.1 Multiple fs types .................................. 30
    5.3.2 Network file systems .................................. 31
    5.3.3 Distributed file systems .......................... 32
    5.3.4 Directory structures .................................. 34
5.4 Allocation of block resources .......................... 34
    5.4.1 Bit map .................................. 35
    5.4.2 Lists .................................. 37
    5.4.3 Buddy Lists .................................. 37
5.5 Disk scheduling & allocation .......................... 38
    5.5.1 Disk scheduling .................................. 38
## Contents

### 9 Memory management 83

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Virtual memory</td>
<td>83</td>
</tr>
<tr>
<td>9.2 Memory management unit</td>
<td>85</td>
</tr>
<tr>
<td>9.2.1 Dirty bit</td>
<td>87</td>
</tr>
<tr>
<td>9.2.2 Content addressable memory</td>
<td>87</td>
</tr>
<tr>
<td>9.2.3 Example: the Pentium TLB</td>
<td>88</td>
</tr>
<tr>
<td>9.2.4 Page replacement schemes</td>
<td>89</td>
</tr>
<tr>
<td>9.3 Segmentation</td>
<td>90</td>
</tr>
<tr>
<td>9.4 Fragmentation</td>
<td>90</td>
</tr>
<tr>
<td>9.5 Linking and loading</td>
<td>90</td>
</tr>
</tbody>
</table>

### 10 Distributed systems 93

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 Topics in distributed systems</td>
<td>94</td>
</tr>
<tr>
<td>10.1.1 Fault tolerance</td>
<td>94</td>
</tr>
<tr>
<td>10.1.2 Migration and load balancing</td>
<td>95</td>
</tr>
<tr>
<td>10.2 Case study: Mach kernel</td>
<td>96</td>
</tr>
<tr>
<td>10.3 Case study: Amoeba</td>
<td>97</td>
</tr>
<tr>
<td>10.3.1 Capabilities</td>
<td>97</td>
</tr>
<tr>
<td>10.3.2 Pool processors</td>
<td>98</td>
</tr>
<tr>
<td>10.3.3 Directory structure</td>
<td>98</td>
</tr>
<tr>
<td>10.3.4 Multiprocessor systems</td>
<td>99</td>
</tr>
<tr>
<td>10.3.5 Boot server</td>
<td>100</td>
</tr>
<tr>
<td>10.3.6 Parallel make</td>
<td>101</td>
</tr>
<tr>
<td>10.3.7 Amoeba commands</td>
<td>102</td>
</tr>
</tbody>
</table>

### 11 Security 105

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 Levels of rights</td>
<td>105</td>
</tr>
<tr>
<td>11.2 External and internal security</td>
<td>105</td>
</tr>
<tr>
<td>11.3 Shared key systems</td>
<td>106</td>
</tr>
<tr>
<td>11.3.1 Ciphertext</td>
<td>107</td>
</tr>
<tr>
<td>11.3.2 Product ciphers</td>
<td>108</td>
</tr>
<tr>
<td>11.3.3 Public keys</td>
<td>108</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The functions of an operating system can best be understood by examining the evolution of computer usage. The elements of modern operating systems are those that were found to be necessary during this evolution.

The first machines with real programs were developed in the 1940’s. In those days, the programmer would put his or her name down on a computer time chart (say for an hour), and then would have total control of the machine for that time.

The programmer operated the computer, put in a program, debugged and ran it. Often the program did not run to completion before the programmer’s time was up, and it would all have to be done again. Hence there was fierce competition for the ’resource’ (computer time).

The computer time itself was very valuable -

- An IBM 7094 cost $4M
- Over a 5 year lifetime of the machine this is about $90 an hour.

Programmers at that time were earning less than $3 an hour. If the programmer made mistakes operating the machine, the whole time slot could be wasted.

A new job appeared in the computer hierarchy:

- the professional computer operator.

The programmer was barred from the computer room, and had to submit jobs to the operator for processing. When things went wrong, the operator would print out a ’core dump’ (a listing of the contents of the computer memory) to help (?) the programmer find the errors and correct them. The operator could then immediately continue with the next job. This improved the computer usage, but annoyed programmers who remembered the ’good old days’ when you could correct the error while you were in the machine room, and then restart the job.
1.1 Re-ordering jobs

Computer operators could make more efficient use of the machines by re-ordering the jobs to be performed.

If there are 4 jobs (a,b,c &d) that take 8min, 4min, 1min and 16min, different ordering gives different results:

<table>
<thead>
<tr>
<th>Job</th>
<th>Finishes after</th>
<th>Job</th>
<th>Finishes after</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>8</td>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>12</td>
<td>b</td>
<td>5</td>
</tr>
<tr>
<td>c</td>
<td>13</td>
<td>a</td>
<td>13</td>
</tr>
<tr>
<td>d</td>
<td>29</td>
<td>d</td>
<td>29</td>
</tr>
</tbody>
</table>

The jobs all finish after 29 minutes, but the turnaround times for the four jobs are different. In the first ordering, the average turnaround time is $\frac{8+12+13+29}{4} = 15.5$ minutes. With the second ordering, the average turnaround time is $\frac{1+5+13+29}{4} = 12.0$ minutes.

This is called SJF (Shortest Job First) scheduling. It minimizes average turnaround time.

Note: - SJF can delay long jobs indefinitely and this may be very undesirable.

In modern operating systems, the scheduler determines which job to run next according to some measurable quality, or combination of qualities:

- Shortest job
- Highest priority
- Meet a deadline

Sometimes schedulers can go wrong. The TENEX multi-user operating system allowed individual users to use up all the process time merely by holding down a key on their terminal. The scheduler placed a high priority on keypresses in order to improve the key response time. When a keypress occurred, the corresponding processes were temporarily raised to an extremely high priority, preempting other processes.

\[1\text{Jobs such as “process your pay”, or “shutdown the nuclear reactor”, should not be delayed indefinitely.}\]
1.2. **SPOOLING**

The machines were still not being used at their maximum capacity, due to the time it took an operator to load the program and data, and the time it took to get the results off the printer. These were called the setup and teardown times.

**For example:** In Figure 1.1, a card reader reads about two cards a second. If a program was stored on 1000 cards, the computer would take about 8 minutes to read the cards in. The program might take 8 minutes to run, and 8 minutes to print, giving 33% CPU usage.

The programs might call a ‘card reader’ routine which actually read from the tape drive the card images as needed. Spooling was also used to buffer printing in the same way.

‘Card images’ are still an important part of many mainframes - even though the mainframes often don’t have card readers any more.

1.2.1 **Spooling with hardware**

One method used to fix this was to have another computer reading cards, and sending them to the main computer as needed - the computer would read ahead and buffer card images.

This was called a satellite processor. This technique relied on all the programmers using a common ‘card reading’ routine. This routine was modified to access the satellite processor instead of the card reader, and the programmers did not have to change their programs at all. The technique could also buffer the printed output, sending it to the satellite processor, which would print as and when it could.

1.2.2 **Spooling with software**

Another way of doing this was spooling\(^2\). The main computer used interrupts to perform the same function as the satellite processor.

\(^2\)Spooling is short for **Simultaneous Peripheral Operation On Line**.
For example: Every time a card is read, an interrupt occurs. The card image is written to tape. The next card read is started. The interrupt service routine returns to the program.

Input spooling in detail:

The process involves an Interrupt Service Routine (ISR) in the following way:

- When data is available from an input device, it causes a (hardware) interrupt at the CPU.
- The processor pauses its current activity, and saves away its current state.
- The processor then initiates the ISR to process the data:

  \[
  \text{If data available then} \\
  \text{store on disk}
  \]

- When a process wishes to read the input data, it accesses the disk image.

Since storing the input data away to disk may be very fast, this process may only reduce the CPU speed by a few percent:

We can see in this example a reduction in CPU speed of about 0.5%.
Output spooling in detail:

Spooling may also be done on the output side, with similar results. An interrupt service routine in the computer responds to end of line interrupts from the printer. It retrieves print data from the disk if any exists.

When the CPU wishes to print:

\[
\begin{align*}
\text{if ThePrinterIsFree then} & \quad \text{sendtoprinter( line )} \\
\text{else} & \quad \text{sendtodisk( line );}
\end{align*}
\]

When the printer finishes printing a line:

InterruptCPU

When the CPU receives a printer Interrupt:

\[
\begin{align*}
\text{if DiskBufferEmpty then} & \quad \text{DoNothing} \\
\text{else} & \quad \text{sendtoprinter( getlinefromdisk );} \\
& \quad \text{Continue ...}
\end{align*}
\]

Question: What are some examples of modern use of spooling?
1.3 Job control languages

Now programs were being 'batched' together and the card entry and program execution were not necessarily happening at the same time.

One problem that needed a solution involved faulty programs that continued to read cards. They could read right through the deck, and through the next program!

The JCL or Job Control Language fixed this. Special extra cards were added to the program decks - often with an identifying character in the first column ($).

The JCL cards marked the beginning and end of jobs, program areas and so on:

```
$START JOB
$DATA
$ENDJOB
```

JCLS became more complex including conditional tests:

```
$IF COMPILEDOK ...
```

They required, of course, a JCL interpreter on the computer - this program interpreted the JCL commands and then allocated computer resources to run the program.

For example: When I was at university, the B6700 JCL (WFL - Work Flow Language) was very complex - a complete language in itself - with variables and loops. Our student programs of course had a very low priority, and would take all day to run, but the WFL executed immediately the cards were read in, so the challenge was to write your programs in WFL! (And have them execute immediately!)

Question: What is the modern equivalent of a JCL?
1.4 Core dumps

When a program fails, the contents of the computer's memory were printed (or spooled, to be printed later).

Nowadays, in UNIX systems, a file is written to disk. It can then be viewed with a core dump debugger to find out why the program failed.

If the whole system crashes, a crash dump is saved - an image of the whole machine. The dump can be examined by engineers using a crash dump debugger to find out why the crash occurred.

These two facilities help to account for the relative stability of UNIX systems.

Note: Large operating systems such as VMS and UNIX often have hundreds of patches, but you normally only patch what you have to.

Question: What should you do with core files found in your directory?

1.5 Libraries

Large parts of many programs do the same things over and over again. Commonly used routines are put in libraries, which may be linked in at run time. There are three distinct advantages:

1. Saving on disk space
2. Routines can be updated
3. Routines are more reliable

For example: All ‘C’ routines (clearscreen etc) are kept in a common place (libc.a in a UNIX system). When you run a program, the OS uses the latest version of libc.a automatically.

Note - Most compilers can also statically link in the needed libraries if necessary. The executables are larger.

System changes and upgrades may become invisible - as long as the libraries are used. If you had a program that used library calls to send data to a printer, you could replace that call with one that spooled the printing without the program noticing.

Note - In general you should not bypass library or system calls. If you do, your programs date quickly.

\[3\] This file is often often called 'core', and overwrites any existing file with that name.
For example: Early versions of DOS provided a simple routine to output 1 character to the screen. It took about 2 mS for each character. To rewrite the whole screen took 4 seconds. S/W people bypassed the routine, writing directly to the hardware. The S/W stopped working when different hardware was used.

Question: Can you give an example of a program that has failed over time due to OS updates?

1.6 Time sharing

In the 1960s, advances in hardware and software allowed the machine to perform multiple programs (almost) at once, by time slicing between each running program. This allowed even more effective use of the CPU, as a machine could run one job while waiting for another one to complete an I/O operation.

One facility which began to be provided was the 'time-sharing’ service, where users at terminals could each have some percentage of the machine time (CTSS 1962). Each user saw a virtual machine that appeared to be a whole machine - just like they had in the 1940s!

In fact each user would just get a segment of the computer. The operating system had to ensure fair distribution of the computer resources, and that users couldn’t interfere with each other.

This development in mainframes was echoed in minis and microcomputers through the seventies and eighties. A timeline shows the development:
1.6. TIME SHARING

**Question:** What is an example of a modern time sharing system?
Chapter 2

System Functions:

An operating system may be thought of in two ways:

- **As a Resource Manager** - the primary resource is just the system hardware and CPU time.

- **As a virtual machine provider** - the provider shields the user from the horrors of the machine.

One viewpoint is a 'bottom-up' viewpoint, and the other 'top-down’. The main facilities developed have been:

1. CPU time management
2. I/O management
3. Memory management
4. Virtual machine interfaces
5. Protection schemes

**Question**: What is a virtual machine?

### 2.1 User interfaces

Early user interfaces were primitive things which barely disguised the underlying machines. Two things progressed this further however.
1. **The introduction of OS-360** - It brought a whole new train of thought to computing. This operating system included a huge collection of emulation software which allowed the computer to emulate nearly all the previous IBM machines, so that all the old IBM software was still useable on the newer machines. This promised an upwards path to all IBM users, whose machines could 'look like' an IBM 1401 or an IBM 7904 and so on. Each machine could contain many 'Virtual Systems'.

2. **MULTICS** - This development was a modest project by MIT, BELL and GE to provide all the computing for the BOSTON area: MULTICS - the parent of UNIX. The MULTICS user interface bore no relationship to the underlying machine. MULTICS was written in PL/1, and was portable across a range of machines. It provided each user with a virtual machine with:

   - Hierarchical directories:
   - A CLI interface:
   - Utilities (mail, editors etc)

MULTICS, and hence UNIX, separated the user interface from the operating system. The user interface was just a program run on the machine, which provided the significant features of the users view of the virtual machine. MS-DOS and UNIX provide similar CLI interfaces. The WIMP interface is another 'virtual machine' user interface. (MacIntosh, Amiga, Xerox Star, XWindow systems etc).

The important thing to note here is that the users view of the machine may not bear much relationship to the actual machine - there is an interface layer hiding the machine, and in fact hiding the system calls from the user.
Question: In DOS, what is the user interface?

2.2 Resource management

Another level of the operating system involves the management of resources. The resources are just those of the processor:

• Processor time
• Processor memory
• I/O devices (disks/terminals/printers/displays)

There are many methods used to allocate processor time, all attempting to improve usage. In some situations, average turnaround time may not be the best measure of system usage - for example the ‘shortest task’ algorithm given before may keep on delaying a process which MUST run before a certain time of day (deadline).

Machines are most efficient if the tasks spend as much time in memory as possible. The allocation of computer memory can also be done in many different ways (smallest user first and so on). Allocation of computer memory may involve re-allocation of computer time.

IO handling is nearly always handled by the operating system, but care must be taken to ensure that reasonably efficient calls can be made. MS-DOS calls to the screen are often bypassed because they are character-at-a-time calls. This leads to compatibility problems with programs that do not use the MS-DOS calls.

Disks and tapes can store huge amounts of data, and the handling of them is done in the operating system. Most modern operating systems support some level of device independance, so that a program to write words to a printer could just as easily write to a screen, or a disk file.

Question: Which resource management area takes up the most room in your favourite OS?

2.3 Synchronisation

We now have a picture of multiple tasks running in a computer more or less at the same time. This introduces a new set of problems:

For example: let us imagine two processes which both want to use a printer and a disk drive at the same time. Process 1 asks for the printer first, process 2 then asks for the disk drive. Both processes are allocated exclusive use of the resource. Now process 1 asks for the disk drive, and finds it has already been allocated. What does this process do?

These resource allocation and process communication problems are normally the province of the operating system, and there are (now) well understood solutions to some of the problems.
Question: Can you identify a possible synchronization problem with the equipment you know about at USP?

### 2.4 Protection:

There are several areas of interest -

- Protection from acts of God
- Protection from hardware errors
- Protection from incorrect programs
- Protection from malicious programs
- Protection from malicious users
- Protection from oneself

Many operating systems provide backup facilities, and some hardware is designed with redundancy and backup in mind (an extreme example is the TANDEM computer range). Operating system software typically provides some level of protection from hardware errors. Protection from incorrectly running programs require certain architectural features in the processor.

Malicious users and programs require different techniques. Password protection is often used to lock out users, and memory protection hardware to stop user programs from 'looking' at other user programs (and hence spying).

Lastly there is of course protection from oneself! In MS-DOS when you accidentally type `del *.*` the files are not actually deleted, they are just marked as ready to be used as free space. PCTOOLS and similar programs can resurrect those files if they have not been used for other purposes. This is an example of the operating system protecting you from yourself!

Question: In what order would you rate these threats?
Chapter 3

OS architectures

The operating system itself is an arrangement of software, with some sort of access from the user programs. The software ranges from the tiny operating systems in single chip control systems (less than 2,000 lines of code) up to large mainframe operating system software (more than 1,000,000 lines of code). The operating system is often the largest software module that runs on a machine, and a lot of time has been spent studying operating systems and their architectures.

In general, an operating system is a set of event driven utilities. The events which drive it are:

- SYSTEM calls by user processes
- I/O (Interrupt) requests
- Fault recognition and response

Several distinct operating system architectures have developed:

- Monolithic
- Set of SYSTEM calls
- Layered
- Distributed

Question: Which sort of OS is DOS?
CHAPTER 3. OS ARCHITECTURES

3.1 Monolithic

The monolithic operating systems were developed in the early days of computing. Here the OS could be thought of as the only task running on the machine. It would call user programs (like subroutines) when it was not performing system tasks.

These operating systems were large collections of service routines, compiled all at once, and with very little internal structure. The monolithic operating system has died out over the years, and has been replaced with more modular architectures.

3.2 Set of system calls

These architectures take a reverse view of the computer, representing the operating system as being a collection of utilities or services provided by the computer to the user processes. Most modern operating systems are of this sort.

This is sometimes called the kernel or nucleus approach. This approach provides a minimal set of functions, and the rest of the operating system is constructed out of them. When the computer is executing any of these kernel functions, it is said to be in kernel or supervisor mode. When the computer is not executing the functions, it is said to be in user mode.
Kernel routines often perform tasks that 'hide' the real machine from the user process, and with some computer hardware, the kernel routines may be the only way that a process can access I/O devices or memory outside a certain fixed range. With careful design of hardware and software, processes may be completely protected from other rogue processes.

For example: The computer hardware may inhibit memory accesses outside a certain range unless the computer is in supervisor or kernel mode. Since the only way to enter kernel mode is by doing a system call, and since a user process cannot alter the operating system because it is outside of its memory range, no process may affect another process unless the operating system provides a system call to allow it.

Question: Would you expect memory management software to be inside or outside of the kernel?

3.3 Heirarchy of system calls

The previous two level structure can be extended to a multiple level (layered) one, with each layer providing a level of abstraction to some facility provided by the operating system.

For example: the I/O management in OS9 has a four level heirarchy as seen in Figure 3.2.
3.4 Distributed operating system

With layered operating systems, the interfaces between the layers are well defined and in general do not depend on anything except the parameters passed to the called process, and the parameters returned to the calling process. There are two main techniques used to pass parameters:

- Call by reference
- Call by value

In call by reference, the address of the parameter is passed to the service routine, and it will look at that address to find the parameter. If you wanted the service routine to run on a different computer, this parameter passing technique would not be suitable.

Message passing techniques are more suitable for distributed operating systems. In these systems, the parameters are passed as a message, and the process blocks until a reply is received. The message may be transferred around a computer network and processed by a different machine than the one on which the calling process is running.

The message passing model involves simultaneously running service processes which communicate via message passing, rather than the hierarchy of procedure calls in a conventional OS. Message passing is used in MINIX, QNX, the AMIGA operating systems and transputer equipment.

Message passing can be 'hidden' within many system calls by system software which copies the parameters to a message buffer and passes that buffer to the service routine before blocking until the completion of the call.

**Question:** Give an example of how blocking style code could be written.
Chapter 4

Multiple processes

4.1 Simple process calls

In our kernel model, we have a minimum set of system calls to handle processes and system facilities. In the following sections, more detailed information may be found by looking at the manual pages for the system, calls on a UNIX system.

One of the core parts of an operating system is the effective management of the processes within it. Process management may only be 2% of the operating system code, but it has to be the right 2%!

A process is an executing program - and may be halted and restarted as long as all the registers, data and stack areas are returned exactly as they were when the process was halted. This process information may be stored in a process table.

In Figure 4.1 we see the allowable states for a process. A running process is one that is currently getting processor time. When a process is blocked, it is unable to run until some event occurs. A ready process will run the next time the computer allocates time for it.

The allowable transitions are:

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
4.2 Operations on processes

The first essential operation required is of course one to start up a process! This call would allocate the resources needed for the process, and set up a Process descriptor before allowing the process to run. This service has of course a matching one to kill or remove a process.

Other process calls are:

- **Switch Process** - changes the current process to ready and then gives control to the next ready process. If no other process is ready, the current process is resumed. If no processes at all are ready, the scheduler must soak up the process time until a process is ready.

- **Suspend and resume Process** - change the state of the specified process to blocked (Suspend) and ready (resume). The process scheduler will decide what else to do with them. These calls just change the state variable in the process descriptor.

- **ChangePriority** - change the priority of one process to give it more or less access to the computers resources.

So how are processes set up in the first place? UNIX provides a system call (*fork*) which splits the current executing process into two. The *fork* call, allocates memory for the child process,
and then copies the parent’s ‘image’ into the memory before copying the parent process table entry. Each copy of the process gets a different returned value from the call, the parent getting the child’s process ID, and the child returning the value 0. Using these values, each process can determine what to do next:

```c
procid = fork();
if (procid != 0)
    doParentProcessing();
else
    doChildProcessing();
```

Other techniques may be used. For example, a specified process may be started up, inheriting its parent files, but with its own code and memory allocation determined by a module descriptor.

### 4.3 Data structures for processes

The data structures needed for process management are commonly called process descriptors or process information blocks. They normally contain information that represent the process to the operating system. The information may include:

- the state of the process,
- the identity of the process,
- resources in use by the process, and
- accounting information.

In MICROs (a small controller OS), the process descriptor is very simple:

```c
ProcessDescriptor = RECORD
    Status : (Ready,Running,Blocked);
    Module : address of process program area descriptor;
    DefaultIn,
    DefaultOut : Default Input and output path numbers;
    DefaultPath : Default path names;
    StackBase : address of process data area;
    StackSize : size of process data area
    StackPointer : current stackpointer
    UserID : Identity of the activating process;
    InfoWork,
    InfoFinal : pointer to statistics area;
END;
```
Various system services use the process descriptor.  

**For example:** when a process is killed or terminates normally, the activating process uses the information in the ProcessDescriptor to return memory and close open files. If this information was not easily available at this level, it may be quite hard to decide which memory blocks or which files must be returned to the system.

**Question:** Why would the process descriptor store addresses related to the base and limit of the stack/data area?

### 4.4 Scheduling methods

Process scheduling is often handled by the operating system. The first question is:

- What criterion do we use in scheduling?

There is no one best criterion for process time allocation, as we saw before - the nuclear power plant example. The sort of criteria we may use are:

- Fairness
- Efficiency
- Priority
- Response time
- Turnaround time

Often a mixture of several different measurements may be used, but sometimes unexpected results occur with complex systems as we saw before in Section 1.1.

The scheduler may be activated either by the process itself, or by a clock tick.

#### 4.4.1 Preemptive and non-preemptive scheduling

When the process voluntarily gives up the processor, it is called non-preemptive scheduling. When the scheduler switches processes automatically, it is called preemptive scheduling.

Preemptive scheduling has the advantage that the processes do not have to be specially written to call the scheduler - in fact the processes do not have to be running - and this type of scheduling is normally done in larger operating systems.
Smaller (controller based) operating systems often use non-pre-emptive scheduling, the individual processes relinquishing control of the processor from time to time. This has some advantages in that timing loops may be more accurately done, critical sections may be avoided more easily, and extra INTERRUPT hardware is not required.

MICROS uses a simple round robin pre-emptive scheduler. In this scheduler, each process gets one time slice if it is ready to run. If tasks wish to hog the processor for some reason, they must disable interrupts, use the processor and then re-enable interrupts.

**Note:** One buzzword that you sometimes hear is that UNIX (and MICROS) use lightweight execution threads and hence can do context switching quickly. Lightweight threads require no re-mapping of memory, and so context switching can be done just by swapping register sets for each process.

## 4.5 Sample calls

Table 4.1 summarizes some of the sorts of calls an operating system might have for manipulating processes. In our examples below, we have taken some of the calls from IRIX6.5, a commercial version of UNIX.

### 4.5.1 Terminate process - exit()

```c
#include <stdlib.h>
void exit(int status);
#include <unistd.h>
void _exit(int status);
```

The C library routine *exit*, invokes the system routine *_exit* upon completion of its own cleanup chores. Routine *_exit* terminates the calling process with the following consequences:

- All of the file descriptors, directory streams and message catalogue descriptors open in the calling process are closed.
- A SIGCHLD signal is sent to the calling process’s parent process.
• If the parent process of the calling process has not specified the SA_NOCLEWDWAIT flag, the calling process is transformed into a zombie process. A zombie process is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space.

• The parent process ID of all of the calling process’s existing child processes and zombie processes is set to 1.

• If the process belongs to a share group, it is removed from that group. Its stack segment is deallocated and removed from the share group’s virtual space.

### 4.5.2 Execute process - exec() and fork()

**exec()**

```c
#include <unistd.h>
int execv (const char *path, char *const *argv);
```

The system routine `exec` overlays a new process image on an old process. The new process image is constructed from an ordinary, executable file. This file is either an executable object file, or a file of data for an interpreter. There can be no return from a successful `exec` because the calling process image is overlaid by the new process image.

An interpreter file begins with a line of the form

```plaintext
#!/ pathname [arg]
```

where pathname is the path of the interpreter, and arg is an optional argument. When an interpreter file is exec’d, the system `execs` the specified interpreter. The pathname specified in the interpreter file is passed as `arg0` to the interpreter. If `arg` was specified in the interpreter file, it is passed as `arg1` to the interpreter. The remaining arguments to the interpreter are `arg0` through `argn` of the originally exec’d file.

**fork()**

```c
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void);
```

The system routine `fork` causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). The child process inherits the attributes of the parent process. Such attributes as:

• real user ID, real group ID,
4.5. SAMPLE CALLS

- effective user ID, effective group ID,
- nice value [see nice(2)] and so on.

Upon successful completion, `fork` returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of (pid_t)-1 is returned to the parent process, no child process is created, and errno is set to indicate the error.

### 4.5.3 Wait on event - pause()

```c
#include <unistd.h>
int pause(void);
```

The system call `pause` suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

If the signal causes termination of the calling process, `pause` does not return.

If the signal is caught by the calling process and control is returned from the signal-catching function [see `signal(2)`], the calling process resumes execution from the point of suspension, with a return value of -1 from `pause` and `errno` set to EINTR.

### 4.5.4 Signal event - sigsend()

```c
#include <sys/types.h>
#include <sys/signal.h>
#include <sys/procset.h>
int sigsend(idtype_t idtype, id_t id, int sig);
int sigsendset(procset_t *psp, int sig);
```

The system call `sigsend` sends a signal to the process or group of processes specified by `id` and `idtype`. The signal to be sent is specified by `sig` and is either zero or one of the values listed in `signal(5)`. If `sig` is zero (the null signal), error checking is performed but no signal is actually sent. This value can be used to check the validity of `id` and `idtype`.

In order to send the signal to the target process (pid), the sending process must have permission to do so.
Chapter 5

File systems

5.1 File types

A simple initial definition of a file\textsuperscript{1} might be:

*A named block of data.*

However, this definition proves inadequate when we look at the wider usage of the term in modern operating systems.

5.1.1 Stream of characters vs structured

A fundamental file type found on DOS, UNIX, and WinXX systems is the *stream of characters.* In these files, each 8-bit or 16-bit unit represents a single character.

On UNIX and NT systems, we build structured files (for example - files of records), *on-top-of* these stream of character files.

The *Macintosh* and *NeXt* computers use a more developed view of a file (which they call a *document*). A Macintosh file is a fork containing two elements:

1. The data belonging to the file
2. A *resource fork* containing information about the data

Typically the *resource fork* is used to keep file information that may change. If the file is an application, the *resource fork* may contain the text for menu items, or the colours and backgrounds of popups. The file fork contains the program itself. The resource fork may be separately edited using a *Macintosh* program called resedit.

\textsuperscript{1}IBM use the term *dataset* to refer to a file.
### 5.1.2 Names

The file names can be of different sizes, and use different characters.

**DOS file names:** DOS files use an 8-byte name with a 3-byte extension - Xxxxxxx.Yyy. There are limits on the characters allowed in file names.

**UNIX file names:** UNIX systems vary, but typically allow 255 character file names, using nearly any characters, and being case sensitive. The name “a.b.c.d.e()[\]” is a perfectly valid UNIX file name. So is “a name with spaces!”.

**NT file names:** NT systems support a range of file systems, including the older DOS based file systems. When using NTFS, file names are case insensitive, and may be up to 255 characters long, using nearly any characters.

### 5.1.3 Links and aliases

On some systems, a link or alias may be made to a file. A link is like a pointer to a file. Not all operating systems support the concept of a link.

The link is a useful filetype. If we wished to place a copy of a file called x.html in everyones home directories, we might use up a lot of disk space. By using a link, we can just have one copy of the file, and lots of links to it.

### 5.2 Redirection

Many operating systems treat all input and output as an instance of file I/O. (DOS is very much an exception with its special purpose graphics I/O). This allows you to redirect the output of programs to different devices or files.

- `dir > Testfile.txt` (directs output to a text file)
- `dir > LPT1` (directs output to a printer)
5.2. REDIRECTION

5.2.1 Stdin, stdout and stderr

UNIX processes have three default files known as stdin, stdout and stderr. UNIX commands normally route their output to the file stdout, route error messages to the file stderr, and accept input from the file stdin. Also by default, these files normally go to and from the controlling terminal (i.e. where you started the program from). Redirection changes the stdin, stdout and stderr files. stdin, stdout and stderr are just file number 0, 1 and 2.

5.2.2 Pipes

In UNIX it is easy to take the stdout of one program, and send it to the stdin of another command. This process is called piping the output of one command to another command.

```bash
ls | more
```

- `ls` - a command to list the contents of a directory.
- `more` - a command to display a file one screenful at a time.

In the above example, the output of the `ls` command is fed directly to the input of the `more` command. In Figure 5.1 we see the two processes `ls` and `more` running as separate tasks, with the stdout of the `ls` command piped to the stdin of the `more` command.

The `ls` and `more` commands run at the same time on a UNIX system. On a DOS/Win system by contrast, piping uses a temporary disk file. The `ls` command must complete before the `more` command runs.

On UNIX, the processes in a pipe must be synchronized so that the pipe does not overfill. This “stringing together” of commands is a characteristic of UNIX systems:

```bash
cat /etc/passwd | awk -F: '(print $5)' | sort | uniq | wc -l
```

The above UNIX command line consists of five commands piped together:
cat - concatenate (or list... or type...) a file, in this case the file /etc/passwd.
awk - run the awk program on the file, in this case to print out the fifth field, if the fields are separated by the colon character.
sort - sort the resulting file.
uniq - only display different lines.
wc - count the characters and in this case lines in a file.

The command prints out the number of accounts with unique names in the password file on a UNIX system.

Other useful UNIX commands in pipes are:

sed - a stream editor
perl - an HLL
grep - a pattern matching tool
echo - echoes arguments.

5.3 File system architectures

There are two common file system structures:

Disk based: as in A:\WINDOWS\system.ini. There are multiple file systems rooted on each physical disk.

Root file system based: as in UNIX and MAC systems. One file system is designated the file system root, and all other file systems are mounted into the root file system somewhere.

5.3.1 Multiple fs types

Some architectures (NT/UNIX and especially LINUX) support the use of multiple file system types. For example, LINUX systems can directly mount and use DOS FAT, WinNT NTFS, Macintosh and various formats of UNIX disk file systems, as well as Netware, SMB, Apple, NFS and other remote (network) file systems.

Sometimes this can produce strange results.
5.3. FILE SYSTEM ARCHITECTURES

5.3.2 Network file systems

The principal advantages of file server systems are in cost saving:

- 400MB of disk (about what is needed for the Microsoft Office products), may be replicated on each machine, or can be stored on a file server just once, with a reduction on the disk storage requirement for each client machine.

- In addition, updates to the software may need to only be done once, reducing administration costs.

There are multiple file serving standards, using a range of communication methods/protocols to transfer files to and from clients and servers. Most operating systems have clients and servers for a range of file server standards - there is no one best standard.

For example: In a medium sized homogeneous environment of PCs, a Novell Netware\textsuperscript{2} server on a remote machine may be used to serve files efficiently. In another, SMB\textsuperscript{3} may be used from an NT machine.

At USP, we have NFS, SMB and Macintosh file servers (which in general don’t interact with each other), and a client machine could have client software for each. The USP PCs generally have SMB clients installed, the Macintoshes have both SMB and Macintosh clients, and our UNIX machines have all clients installed.

**NFS - Network File System:** NFS is an older and stable file system based on UNIX file structures (It has similar permissions/attributes on the files). NFS is nearly always transported over IP - the Internet (and UNIX) base communication protocol, using a system called RPC - Remote Procedure Calls. NFS is the basis for the Internet file server proposed standard Web-NFS. Clients are available freely for all platforms, and any NFS client can connect to any NFS server given the correct access and permissions. NFS is an open standard.

**SMB - Server Message Blocks:** SMB is provided as standard on Win3.1, Win95 and WinNT systems, and can be installed on UNIX and Macintoshes. The systems do not necessarily interact all that well due to the range of developments to the standards over the last five years, but all systems provide a base (IBM LAN-Manager) functionality which can be used. (IBM and Microsoft originally developed the standard together). SMB structures map on to attributes and permissions appropriate to Microsoft operating systems. The details of SMB operation are not openly available, but various public domain implementations of both clients and servers exist.

\textsuperscript{2}Novell is the company, Netware is the product.

\textsuperscript{3}Server Message Blocks - another name for the IBM/Microsoft file server system.
Macintosh: Macintosh have public file and print server standards, rarely used outside the Macintosh world, but stable nevertheless, and provided standard on all Macintoshes (including the network hardware).

5.3.3 Distributed file systems

Distributed file systems have a different set of constraints from disk and simple network based file systems. We are particularly interested in:

- **Transparency** - different kinds:
  - access - how do you gain permissions?
  - location - where are the files stored?
  - failure - what happens when a server fails?
  - replication - how is replication done?

- **Security** - password/authentication

- **Migration** - how do you move servers?

- **Efficiency** - load balancing, caching and so on.

**AFS - Andrew File System**

This was developed at CMU (Carnegie-Mellon University) and is currently marketed by TRANSARC. It is the basis for the OSF DCE’s DFS (Open Software Foundation, Distributed Computing Environment, Distributed File Service).

AFS provides a world-wide single rooted file system:

```
/afs/<cell.name>
```

The cell name reflects a single management unit, responsible for users, volumes and so on within that cell. You may access any file by name on the (world-wide) file system if you have access permission:

```
/afs/usp.ac.fj/MACS/user/hugh/myfile.txt
```

The system is intended for efficient use on large networks, and its architecture is based on observations made about measurements taken on UNIX systems.
- Most accesses small (<10kb)
- Reads much more common than writes
- random access rare
- Most files only used by one user at a time.

Existing file systems were not particularly scalable (typical maximum is 1 filesystem to 200 machines). AFS design goals included an arbitrarily scalable system. The basic mechanism is local caching, with a server \textit{callback} mechanism to ensure cache coherency.

If the file is changed at the server, it has a list of \textit{callbacks} to make to the \textit{cached copy} machines, telling them the file has changed. Local cached copies can even be in \textit{diskless} workstations, as RAMDISK can be used.

- There is no need to keep local copies.
- Caching can be done at several levels
- AFS separates volumes into \textit{Read-Only}, \textit{Read-Write}, and \textit{Replicated} (or backup). \textit{Read-Only} volumes can be duplicated transparently.

The AFS design does not allow for \textit{Read-Write} volumes to be duplicated (with all its attendant problems), but does provide for a stable \textit{Read-Only} replication for backup purposes.

The backup is still part of the /afs hierarchy, and it is common to have a directory called \textit{OldFiles} in your home directory which contains all your backups.

\textbf{Security:} For security, AFS uses Kerberos (a public implementation of a secure authentication challenge-response protocol). This means that no passwords are passed in clear text. AFS gives you a key valid only to you for a short time, and uses mutual secure authentication.

\textbf{Access Control Lists:} provide per directory control instead of UNIX style per file.

AFS sits on top of RPC/UDP/IP with a special RPC method. It is optimised for WAN use (i.e. Large numbers of unacknowledged messages and retransmission only of missing data).

AFS is available on UNIX and NT, providing:

- Automatic load balancing, and
- Locational transparency - it is easy to move users or change servers without disruption.
5.3.4 Directory structures

The MICROS file descriptor record contains all the information about a file - the name, creation date, and information about the location of the file. A directory is an aggregation of records like this:

```pascal
FileDescriptor = Record
    FileName : name;
    OwnerID  : integer;
    attributes : (read, write, ...)
    FileSize  : integer;
    DateLastModified,
    DateCreated : date;
    LocationList : array [1..MaxExtents] of segmentdescriptor;
end;
```

This technique of putting all the information in the directory is a bit simple minded, and suffers because there is a maximum number of extents to the file - however it has worked OK for quite a few operating systems (including DOS).

UNIX by contrast has a very simple directory structure involving just the filename and an integer called an I-node. The I-node specifies a separate file descriptor. One advantage of this scheme is that different directory entries can 'point to' the same file.

UNIX-like file systems provide a level of device independance. An open call can initiate access to a terminal or a screen or a file, and subsequent reads or writes do not care what device is selected.

**For example:** The following procedure opens a file, writes a string and then closes the file:

```pascal
procedure test( filename:string);
begin
    f := open( filename,'w' );
    write( f,'A test string' );
    close( f )
end;
```

The call `test( '/dev/ttya' )` will write the string to a terminal. The call `test( 'ti' )` will write the string to a file in the current directory. The file will be called 'ti'.

5.4 Allocation of block resources

The methods used for allocating and deallocating space on a disk are also relevant when looking at memory allocation, since we normally allocate memory in pages of 1024 bytes, just as we

---

4MICROS is a microcomputer operating system used in cash registers and petrol pumps.
allocate disk in sectors of 512 or 1024 bytes. We call them both block resources, and this section could apply to either memory allocation and management, or disk sector allocation and management.

It is common to maintain a up-to-date information in a (memory or disk) management table. When the operating system is looking for free space to use, it may search this table using various search strategies:

**First fit:** Search the table, and use the first chunk large enough.

**Best fit:** Search the whole table, and use the smallest free chunk that is large enough.

**Quick fit:** Pre-divide the table into sections of different sizes (2k, 4k, 8k, and so on). Allocate from a section that is big enough. (Note that this method may be only 50-100% space efficient, but is normally very fast.

When the allocation and deallocation runs for a long time, the block resource may become fragmented, and this is generally something to avoid:

- a fragmented file on disk may be slow to access, or
- a fragmented section of memory may not allow loading of a large contiguous block.

There are various methods used to keep track of the block resource, and they reduce fragmentation to different extents. We will look at bitmaps, lists and the buddy system.

### 5.4.1 Bit map

The bit map system uses an array of bits. Each bit represents a single unit of allocation - for example a 1k page of memory, or a 1k sector on disk. If a bit has the value 0, then the corresponding unit is in use, if the bit has the value 1, then it is free.

To search for 8k of memory in a PMT (page management table), using the first fit algorithm, you would make a pass through the bitmap looking for 8 consecutive 1’s. Some processors have bit instructions which do exactly this.

When the memory is de-allocated, the corresponding bits are set back to 1, and the fragments join up without any further processing.
Figure 5.2: Bit map allocation.

Figure 5.3: List system.
5.4. ALLOCATION OF BLOCK RESOURCES

5.4.2 Lists

In these systems, a freelist is kept for each free section of the block resource. When the resource is first initialized, there is only one entry in the freelist.

When the operating system wishes to do allocation, it searches through the list, looking for a suitable entry (using either first fit, best fit or quick fit search strategies). List systems are reasonably quick when doing allocation and deallocation, but the defragmentation of the resource may cause problems.

The following table shows the number of comparisons needed for different lengths of free list.

<table>
<thead>
<tr>
<th>Length of List</th>
<th>Number of comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>101</td>
<td>5050</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1001</td>
<td>500500</td>
</tr>
</tbody>
</table>

A general rule:

\[ List \ Length \ n \Rightarrow \sum_{x=1}^{n-1} x \text{ comparisons} \]

List systems are fast to allocate and deallocate, but slow to defragment.

5.4.3 Buddy Lists

The buddy list system uses separate linked lists for each size of block allocated. In general, the sizes are arranged in powers of 2. Contiguous chunks always appear in the same list, and can be combined and then moved up to the next size list entry.

This system is:

- Fast to allocate
- Fast to deallocate
- Easy to defragment

\[^5\text{Note that if a processor does not have any fast bit instructions, searching the bit map may be a slow process.}\]
5.5 Disk scheduling & allocation

There are two main areas in which operating systems attempt to improve the behaviour of disk based file systems.

**Scheduling:** Disk arm scheduling can improve disk response time dramatically. If several processes are each wishing to use the disk, the operating system can alter the order of the seeks.

**Caching:** Disk controllers often buffer or cache disk sectors or whole tracks, only writing them out to disk when they must. This may introduce stability problems, but there are dramatic speed improvements.

In addition, disk management systems use various techniques for maintaining the list of free (or available) sectors. These are often done in a similar way to memory management schemes. The same first fit or best fit algorithms may be used to allocate disk space.

5.5.1 Disk scheduling

Accessing a sector on a disk involves various activities which take time, including

- waiting for the sector to arrive under the disk head (rotational latency)
- moving the head from track to track (seek time).
By reordering the track accesses, we can reduce the component associated with the track-to-track movement. Here are some of the algorithms used for reordering track accesses.

**FCFS First Come First Served**: Successive track requests are serviced in the order in which they occur.

**SSTF Shortest Seek Time First**: Service in order that minimizes arm movement by selecting the shortest seek next.

**Elevator**: Service all requests in the current direction, before changing direction - a little like an elevator.

If we had outstanding requests for accesses to tracks 1,700,2,600,44,35 (in that order), then the three algorithms above would seek as follows:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Seek order</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCFS</td>
<td>1 700 2 600 44 35</td>
<td>2558</td>
</tr>
<tr>
<td>SSTF</td>
<td>1 2 35 44 600 700</td>
<td>699</td>
</tr>
<tr>
<td>Elevator</td>
<td>1 2 35 44 600 700</td>
<td>699</td>
</tr>
</tbody>
</table>

**Question**: In what situations would SSTF and Elevator give different seek times?

### 5.5.2 Sector allocation

MICROS uses a disk bit map for keeping track of the free disk space. Disk space is allocated in blocks of sectors, determined by the disk type. The RAM disk has a block allocation size of 1 sector. When new files are opened, the operating system attempts to find a single contiguous area of the disk to put the file in. When files are extended in size, the operating system attempts to put the extents contiguous with the rest of the file.

When MICROS allocates sectors, the changed bit map is written back to disk. The bit map search and modify calls are exactly the same as those used for memory allocation in MICROS.

### 5.6 File system commands

Some common file system commands on UNIX are:

**cd (Change directory)**: The cd command moves you from your present directory to another directory. You must have execute (search) permission in the specified directory. If you do not specify a directory, cd moves you to your login directory.
pwd (Print working directory): The `pwd` command writes to standard output the full path-name of your current working directory. All directories are separated by a / (slash). The first / represents the root directory and the last directory named is your current working directory.

rm (Remove file): Removes (unlinks) files or directories.

mkdir (Make directory): Creates a new directory.

rmr (Remove directory): Removes an empty directory

mount (Mount remote FS): mounts a remote file system into the single rooted file system

umount (Unmount remote FS): unmounts a remote file system from the single rooted file system

5.7 Sample calls

5.7.1 Create file - open()

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char *path, int oflag, ... /* mode_t mode */);
```

The string path points to a path name naming a file. open opens a file descriptor for the named file and sets the file status flags according to the value of oflag. oflag values are constructed by OR-ing Flags

5.7.2 Close file - close()

```c
#include <unistd.h>
int close(int fildes);
```

The parameter fildes is a file descriptor obtained from a `creat, open, dup, fcntl, pipe, or iocntl` system call. The system call `close` closes the file descriptor indicated by fildes. All outstanding record locks owned by the process (on the file indicated by fildes) are removed.

If the link count of the file is zero, when all file descriptors associated with the file have been closed, the space occupied by the file is freed and the file is no longer accessible.
5.7. SAMPLE CALLS

5.7.3 Read and write files - read(), write()

#include <unistd.h>
ssize_t read(int fildes, void *buf, size_t nbyte);
ssize_t write(int fildes, const void *buf, size_t nbyte);

The system call read attempts to read nbyte bytes from the file associated with fildes into the buffer pointed to by buf. If nbyte is zero, read returns zero and has no other results. The parameter fildes is a file descriptor obtained from a creat, open, dup, fcntl, pipe, or ioctl system call.

The system call write attempts to write nbyte bytes from the buffer pointed to by buf to the file associated with fildes. If nbyte is zero and the file is a regular file, write returns zero and has no other results.

5.7.4 Control attributes - fcntl()

#include <unistd.h>
#include <fcntl.h>
int fcntl (int fildes, int cmd, ... /* arg */);

The fcntl system call provides for control over open descriptors. A large number of operations are possible using the fcntl call.
Chapter 6

Case study: UNIX

There are two main flavours of UNIX:

**BSD:** derived from the Berkeley source distribution, and

**SYSTEM V:** (pronounced System-Five!) - a more recent system, which attempts to patch up perceived errors in BSD UNIX.

To a UNIX user, the difference between the two flavours is commonly seen in different parameters to commands:

- `ps -aux` - a BSD process status command
- `ps -ef` - a similar System V command.

The name UNIX until recently could only be used by Bell Laboratories, and so we had Solaris, SunOS, AIX, SCO-UNIX, IRIX and so on. All of these operating systems are UNIX, and the base commands and system calls are extremely close in behaviour (Much more so than DOS version 6 and DOS version 7 for instance).

Originally, UNIX was free to Universities in source form, and many of the UNIX systems and libraries were developed by students. Commercial system developers such as IBM and Sun took the original sources and extended and modified them, normally without removing any of the original functionality. On commercial systems, you often find the original Berkeley copyright notices in configuration files.

Various organizations have tried to develop free versions of UNIX comparable to the extended commercial versions. GNU\(^1\) (Gnu is Not Unix) software is replacement software for most UNIX

---

\(^{1}\)The copyright on GNU software is known as a *copyleft*, and it is intended to prevent you from *inhibiting* further distribution of the software.
commands, and in general GNU utilities are considered to be of higher quality, and more stable\textsuperscript{2}, than either the originals or commercial versions. GCC is the GNU C compiler, and is today considered the highest quality C and C++ compiler available on any platform.

Various free kernels have been developed to complement the GNU utilities, and since 1998, they are considered more stable than any other UNIXes.

**FreeBSD**: A derivative from the original BSD, and

**LINUX**: (pronounced lin-ucks) - A ’new’ kernel - now with SMP support, and for multiple platforms.

The stability of the UNIX system as a whole is determined by the kernel. UNIX kernels range from 100,000 to 1,000,000 lines of code. When the kernel fails, a crash dump is written to disk, and this can be examined to discover the reason for the failure. This has led to a reduction in the number of severe errors in UNIX kernels.

### 6.1 Hardware support

Unix systems require memory management hardware and protected processor modes: kernel and user.

- The only way to get to kernel mode is to make an operating system call, or to have a hardware interrupt.

- Once you have done that, the code you are executing must be in the kernel.

- Critical tasks may only be done in kernel mode:
  
  - change memory maps
  - switch tasks
  - access IO devices
  - change the K/S mode bit

\textsuperscript{2}How do we know that they are more stable? - By running them with random input and then looking for core files. Typical UNIXes have 20\% or so of the utilities fail, but LINUX is better (about 5-10\%).
6.2 UNIX concepts

6.2.1 Multi-user and multi-tasking

By their nature, UNIX systems are both multiuser and multitasking. This means that there are fixed ways of dealing with problems related to multitasking and multiuser use.

**Multiple tasks:** UNIX systems have library calls to handle mutual exclusion and device access in sensible ways.

**Multiple users:** UNIX systems have standard login and password security routines.

6.2.2 Hostnames

UNIX systems are identified by a hostname. You can see this with the `uname` command:

```
manu> uname -a
OSF1 manu.usp.ac.fj V4.0 878 alpha
manu>
```

Most systems will also allow the special name `localhost` to be used when referring to itself.

6.2.3 Accounts

When you log in to a UNIX machine, you have already been set up with various configuration information:

- **User ID** - A unique (on this machine) number to refer to your account.
- **Group ID** - A group for you to belong to.
- **Home directory** - A directory for your files.
- **Login name** - The account name for you to log in with.
- **Password** - A password.
- **Name** - A name for you.
- **Initial program** - A first program for you to run.

All this information is stored in the file `/etc/passwd`, or on some more secure systems, in `/etc/passwd` and `/etc/shadow`.

There is a special account called the `root` account. The `root` account can access all files and change any permissions on the system.

---

3The `/etc/shadow` file can only be read by root, and contains an encrypted version of your password.
6.2.4 Case

Nearly every UNIX command is lower case, and in UNIX case matters!

```
opo 151% uname -a
IRIX opo 6.5 05190004 IP32
opo 152% uname -A
Illegal option -- A
Usage: uname [-snrvmpadR]
    uname [-S system name]
    uname [-V INSTVERSIONNUM]
opo 153% UNAME -a
UNAME: Command not found.
opo 154%
```

6.2.5 Virtual consoles

UNIX has about 30 years of history it carries around, and some of the systems appear a little archaic. Early systems used (initially) teletypes and (later) dumb terminals to access the UNIX system.

![Diagram of virtual consoles](image-url)

Each device had a serial cable connecting it to the UNIX host. When you pressed a key on the terminal it was sent to the UNIX machine. The UNIX box would return characters for display on the screen of the terminal.

Now we tend to network machines, and use entire computers (PCs) as terminals connecting to the UNIX host.
Standard UNIX I/O still outputs to the old teletypes (now virtual teletypes). There were hundreds of different types of terminals (VT102, VT220, VT320, H82, ADM42...), and the UNIX software can still write to any of these terminals (though they are now mostly found in museums). The technique used is to emulate the terminal with software on the PC. If there is a mismatch, things work erratically.

Some older terminals allowed multiplexing of sessions on the screen. Consequently, some of the emulation (virtual terminal) software can do this too. In summary, there are many ways for text-oriented virtual-terminal based programs to go wrong.

### 6.2.6 Networking

Networking is built in to the kernel of UNIX machines. The network protocol is known as IP the Internet Protocol, and is the same network protocol used on the Internet.

### 6.3 Shells

The text-based user interface for UNIX is just an ordinary program. There are a range of them (feel free to write your own!).

- **sh** - (ess-h) - the original shell
- **csh** - (sea-shell) - an advanced shell
- **tcsh** - (T-sea-shell) - an extended csh.
- **zsh** - (Zee-shell)
- **ksh** - (korn-shell)
• **bash** - (Bourne-again-shell - or bash)

The supervisor (root) sets up your initial shell, but you can easily try other shells by just typing their name:

```
opo 148% bash
bash$ # commands here...
bash$ exit
exit
opo 149%
```

### 6.3.1 Shell programming

UNIX shells can be used as programming languages in a similar fashion to batch files. In UNIX the term used is “*shell scripts*”. The scripts may be interpreted by any shell you decide to use, but it is common still to use either *sh* or *csh*, because they are both available on all UNIX systems.

A shell script can be made by creating a text file (say *myscript*), with the first line containing a specification of the program which is to interpret the rest of the script:

```
#!/bin/csh
# This line is a comment, but the previous line specified
# that the program /bin/csh was to be used to interpret the
# rest of this script:
date
ls
```

Note that a word beginning with `#` causes that word and all the following characters up to a newline to be ignored. You can then make the text file into an executable using the `chmod` command:

```
opo 41% chmod +x myscript
opo 42% myscript
Tue Apr 13 15:22:42 NZST 1999
bitmap.fig list.fig setup.fig myscript
opo 43%
```
6.3. SHELLS

6.3.2 Shell commands

The shells have internal commands, or otherwise commands are found and executed from disk. With \texttt{sh}, the internal commands are:

- \texttt{. file [ arg ... ]}: Read the complete file and execute the commands. The commands are executed in the current shell environment.

- \texttt{alias [ -tx ] [ name[=value ] ]}: Alias with no arguments prints the list of aliases, otherwise this command can create aliases for other commands.

- \texttt{break [ n ]}: Exit from the enclosing for, while, until or select loop, if any. If \texttt{n} is specified, break \texttt{n} levels.

- \texttt{command [ -p ] command_name [ argument ... ]}: The command utility causes the shell to treat the arguments as a simple command, suppressing the shell function lookup.

- \texttt{continue [ n ]}: Resume the next iteration of the enclosing for, while, until or select loop. If \texttt{n} is specified, resume at the \texttt{n}-th enclosing loop.

- \texttt{cd [ -LP ] arg}:
  - This command can be in either of two forms. In the first form it changes the current directory to \texttt{arg}. If \texttt{arg} is -, the directory is changed to the previous directory.

- \texttt{echo [ arg ... ]}:
  - Echoes its arguments - see echo(1) for usage and description.

- \texttt{eval [ arg ... ]}:
  - The arguments are read as input to the shell and the resulting command(s) executed.

- \texttt{exec [ arg ... ]}:
  - If \texttt{arg} is given, the command specified by the arguments is executed in place of this shell without creating a new process.

- \texttt{exit [ n ]}:
  - Causes the shell to exit with the exit status specified by \texttt{n}.

- \texttt{export [ name[=value ] ]}:
  - The given names are marked for automatic export to the environment of subsequently-executed commands.

- \texttt{fc -e [ old=new ] [ command ]}:
  - A range of commands from first to last is selected from the last HISTSIZE commands that were typed at the terminal.

- \texttt{fg [ job... ]}:
  - This command is only on systems that support job control. Each job specified is brought to the foreground. Otherwise, the current job is brought into the foreground.

- \texttt{getopts optstring name [ arg ... ]}:
  - Checks \texttt{arg} for legal options.

- \texttt{jobs [ -lnp ] [ job... ]}:
  - Lists information about each given job, or all active jobs if job is omitted.

- \texttt{kill [ -sig ] job}:
  - Sends either the TERM (terminate) signal or the specified signal to the specified jobs or processes.
let arg ...: Each arg is a separate arithmetic expression to be evaluated.

limit [ -h ] resource maximum-use: Limits the consumption of a resource by the current process.

newgrp [ -l ] [ arg ]: Equivalent to exec /usr/bin/newgrp arg.

print [ -Rnprsu[n ] ] [ arg ... ]: The shell output mechanism. With no flags the arguments are printed on standard output as described by echo(1).

pwd [ -LP ]: Displays the current working directory.

read [ -prsu[n ] ] [ name?prompt ] [ name ... ]: The shell input mechanism. One line is read and is broken up into fields.

readonly [ name[=value] ]: The given names are marked readonly and these names cannot be changed by subsequent assignment.

return [ n ]: Causes a shell function to return to the invoking script with the return status specified by n.

set [ -abCefhkmnpstuvx ] [ -o option ] [ -A name ] [ arg ... ]: Setting and unsetting variables.

times: Print the accumulated user and system times for the shell and for processes run from the shell.

trap [ arg ] [ sig ]: arg is a command to be read and executed when the shell receives signal(s) sig.

typeset [ +HLRZfilrtux[n] ] [ name[=value ] ]: Sets attributes and values for shell variables.

ulimit [ -[HS][a | cdfnstvm] ]: Prints or sets hard or soft resource limits.

umask [ -S ] [ mask ]: The user file-creation mask is set to mask.

unalias name ...: The variables given by the list of names are removed from the alias list.

unlimit [ -h ] resource: Removes the limitation on resource.

unset [ -fv ] name: The variables given by the list of names are unassigned.

wait [ job ]: Wait for the specified job and report its termination status.

whence [ -pv ] name: For each name, indicate how it is interpreted if used as a command name.

Any other commands are searched for using the PATH environment variable, to see if it is an external (or disk-based) command.
6.3. **SHELLS**

### 6.3.3 Shell programming language

In addition to the above commands, shells support a flexible programming language, with (environment) variables, and conventional programming constructs such as the following taken from the `sh` shell:

- **for identifier [ in word ... ] ; do list ; done:** Each time a `for` command is executed, `identifier` is set to the next word taken from the `in word` list.

- **select identifier [ in word ... ] ; do list ; done:** A `select` command prints on standard error (file descriptor 2), the set of words, each preceded by a number.

- **case word in [ [ ] pattern [ | pattern ] ... ] list ;; ] ... esac:** A `case` command executes the `list` associated with the first `pattern` that matches `word`.

- **if list ; then list [ elif list ; then list ] ... [ ; else list ] ; fi:** The `list` following `if` is executed and, if it returns a zero exit status, the `list` following the first `then` is executed. Otherwise the `list` following `elif` is executed and, if its value is zero, the `list` following the next `then` is executed. Failing that, the `else list` is executed. If no `else list` or `then list` is executed, the `if` command returns a zero exit status.

- **while list ; do list ; done:** A `while` command repeatedly executes the `while list` and, if the exit status of the last command in the `list` is zero, executes the `do list`; otherwise the loop terminates.

- (list): Execute `list` in a separate environment.

- { list; }: list is simply executed.

- [[expression]]: Evaluates expression and returns a zero exit status when expression is true.

- **function identifier { list;}:** Define a function which is referenced by `identifier`. The body of the function is the list of commands between `{ and `}.

- **time pipeline:** The pipeline is executed and the elapsed time as well as the user and system time are printed on standard error.

### 6.3.4 Shell example 1

The first example is a program used in a security system to process video images. This program has no loops - it is just a series of commands.

```bash
#!/bin/csh
set DATE='date|sed "s/ / /g"|sed "s/ /-/g"'
set DIRECT='echo $DATE|awk -F- '{print $6"/$2"/$3}''
set DATEFILE='echo $DATE|awk -F- '{print $4}''
```
The first three lines set the values of three shell variables, $DATE, $DIRECT and $DATEFILE. We can see the value of the first assignment by looking at the following progression:

<table>
<thead>
<tr>
<th>Command</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>Wed Apr 21 09:32:34 NZST 1999</td>
</tr>
<tr>
<td>date</td>
<td>sed &quot;s/ / /g&quot;</td>
</tr>
<tr>
<td>date</td>
<td>sed &quot;s/ /-/g&quot;</td>
</tr>
<tr>
<td>set DATE='date</td>
<td>sed &quot;s/ / /g&quot;</td>
</tr>
<tr>
<td>echo $DATE</td>
<td>1999/Apr/21</td>
</tr>
<tr>
<td>awk -F'-' '{print $6&quot;/&quot;$2&quot;/&quot;$3}'</td>
<td>09:35:05</td>
</tr>
<tr>
<td>echo $DATE</td>
<td>awk -F'-' '{print $4}'</td>
</tr>
<tr>
<td>echo $DATEFILE='echo $DATE</td>
<td>awk -F'-' '{print $4}'</td>
</tr>
</tbody>
</table>

After the three assignments, our three variables have the following values:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DATE</td>
<td>Wed-Apr-21-09:35:05-NZST-1999</td>
</tr>
<tr>
<td>$DIRECT</td>
<td>1999/Apr/21</td>
</tr>
<tr>
<td>$DATEFILE</td>
<td>09:35:05</td>
</tr>
</tbody>
</table>

The `totiff` command changes an RGB format security camera image into a TIFF format image:

4RGB is Red-Green-Blue - a common image format.
5TIFF - Tagged Image File Format - another common file format.
The `mogrify` commands annotate the image with the $DATE value, and then convert it to a more compact image format (JPG). The image at this time is kept in the `/tmp` directory, in the file `latestpic.jpg`.

The last few commands create a directory (if it does not already exist) with the name `/SECURITY/logs/1999/Apr/21` , and then copy `/tmp/latestpic.jpg` into that directory, renaming it to `09:35:05.jpg`:

```bash
opo 118% echo mkdir -p /SECURITY/logs/$DIRECT
mkdir -p /SECURITY/logs/1999/Apr/21
opo 119% echo cp /tmp/latestpic.jpg /SECURITY/logs/1999/Apr/21/$DATEFILE.jpg
cp /tmp/latestpic.jpg /SECURITY/logs/1999/Apr/21/09:35:05.jpg
```
opm 120%

Note that you can use the `echo` command to test the effect of variable name expansion in immediate mode, not just in your script.

### 6.3.5 Shell example 2

This example is the old BSD `which` command:

```bash
#!/bin/csh -f
#Tag 1792
#
# Copyright (c) 1980 Regents of the University of California.
# All rights reserved.
#
# Redistribution and use in source and binary forms are permitted
# provided that this notice is preserved and that due credit is given
# to the University of California at Berkeley. The name of the University
# may not be used to endorse or promote products derived from this
# software without specific prior written permission. This software
# is provided "as is" without express or implied warranty.
#
# which : tells you which program you get
#
# SGI changes:
#   Always uses current $path.
#   -a = "all" -- keep looking, even after a match is found.
#   -f = "fast" -- don't read ~/.cshrc.

set noglob
set _zurpath = ($path) # inherited via $PATH
foreach arg { $argv }
    switch ("$arg")
        case ":
            set _findall
            shift
            breaksw
        case ":
            set _fast
            shift
            breaksw
    endsw
end
unset noglob
if ( ! $?_fast & & -r ~/.cshrc ) source ~/.cshrc
set noglob

foreach arg { $argv }
    set alias = 'alias $arg'
    switch ( $#alias )
        case 0 :
            breaksw
        default :
            echo ${arg} aliased to \"$alias\"
            set arg = $alias[1]
            breaksw
    endsw
unset found
if ( "$arg:h" != "$arg:t" ) then
```
```bash
if ( -e "$arg" ) then
    echo $arg
    set found
else
    echo $arg not found
endif
continue
endif
foreach i ( $_curpath )
    if ( -x $i/$arg && ! -d $i/$arg ) then
        echo $i/$arg
        set found
        if ( ! $?_findall ) break
    endif
end
if ( ! $?found ) then
    echo $arg not in $_curpath
endif
end
if ( $?found ) then
    exit 0
else
    exit 1
endif
```

### 6.4 Wildcards or filename substitution

Shells allow you to use wildcards to specify files. This is quite different to DOS/WinNT wildcards which expect the application to interpret the special wildcard strings (e.g. `*.txt`). In UNIX shells, this behaviour is a property of the shell or command interpreter, and is available for use in all commands.

<table>
<thead>
<tr>
<th>Character</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Match any (zero or more) characters.</td>
</tr>
<tr>
<td>?</td>
<td>Match any single character.</td>
</tr>
<tr>
<td>[ ... ]</td>
<td>Match any single character in the enclosed list(s) or range(s). A list is a string of characters. A range is two characters separated by a minus-sign (-).</td>
</tr>
<tr>
<td>{ str, str, ... }</td>
<td>Expand to each string (or filename-matching pattern) in the comma-separated list.</td>
</tr>
</tbody>
</table>

### 6.5 Filters

Most UNIX commands accept input from stdin and put their output to stdout or stderr as shown in Section 5.2.1 on page 29. Commands which act in this way are commonly called filters, and we have filters which collate, sort, edit and so on. The following command could be used to edit all the WIN.INI files in subdirectories:
 ls */win.ini|sed -e "c/mail=mail/mail=uspmail/g"

### 6.6 File permissions

UNIX has a simple scheme for file and directory access. Each file or directory has the following nine permissions:

<table>
<thead>
<tr>
<th></th>
<th>Self</th>
<th>Group</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rwx</td>
<td>rwx</td>
<td>rwx</td>
</tr>
<tr>
<td>Read, Write and Execute</td>
<td>Read, Write and Execute</td>
<td>Read, Write and Execute</td>
<td></td>
</tr>
</tbody>
</table>

If we set the permissions of a file to `rwxr_x__x`, you would allow yourself all permissions, your group read and execute permissions, and everyone else only execute permissions.

The command to change permissions is `chmod`:

**chmod [-R] mode file ...** The command `chmod` changes the permissions of each given file according to mode, which can be either a symbolic representation of changes to make, or an octal number representing the bit pattern for the new permissions. Absolute changes to permissions are stated using octal numbers:

```sh
cchmod nnn file(s)
```

where `n` is a number from 0 to 7. Symbolic changes are stated using mnemonic characters:

```sh
cchmod a operator b file(s)
```

where `a` is one or more characters corresponding to user, group, or other, and operator is `+`, `-`, and `=` signifying assignment of permissions, and where `b` is one or more characters corresponding to type of permission.

### 6.7 Customization

There are system wide customization files, commonly found in the directory `/etc`. Each user may also customize his or her own environment.

The customization files for each user are stored in the user’s home directory, and are normally read by the shell when the user logs in. The filename depends on the default shell for the user, but is commonly either the hidden file `.login` or the hidden file `.cshrc`. On my account on `manu`, the `.cshrc` file looks like this:

```
# *****************************************************************

# In UNIX systems, if a filename starts with the character “.” (the full stop), it is termed a hidden file, and filename expansion will commonly ignore it. You must specify the file with the leading dot if you wish to access it.
```
As you can see, it does not do much.

6.8 File system structure

The UNIX file system is hierarchical, single rooted, allows long file names, and supports links. The top level directories typically are standard, though their is no actual requirement in the OS for the directories to be located or named as they are. The top level is something like this:

```
/    
|    
|  var bin dev lib etc usr tmp
|    
|    
|    
bin etc var
```

The contents of these top level directories are the base UNIX system, with a minimal set of utilities and configurations. The \texttt{/usr} directory contains extensions to the base system.
CHAPTER 6. CASE STUDY: UNIX

/bin: contains binaries (or executables) for essential UNIX commands, such as ls, mv, cp and so on. On machine manu, when you type ls you are executing the command /usr/bin/ls, but /bin is just a link to /usr/bin.

/dev: contains "device special files" which are specifications for attached I/O devices and pseudo I/O devices. The "device special file" /dev/console might represent the console device, and a command which outputs to the file /dev/console will appear on the console. Some other "device special files" are:

/dev/mouse: If you read this file you see the mouse data (movement and button presses).
/dev/null: If you putput to this file, the output gets discarded.
/dev/fd0, /dev/fd1, ...: Floppy disk devices. Normally if you wish to read or write to a disk in UNIX, you have to mount it into your single rooted file system somewhere. However, programs like format need to access the device directly (in a raw fashion). Note that this is the only way to access the disk directly - there is no way of bypassing "device special files" and getting directly to the I/O device.
/dev/hda1, /dev/hda2, ...: Hard disk devices accessed in raw fashion. It is common in UNIX systems for /dev/hda to refer to the whole disk, and /dev/hdal, /dev/hda2 to refer to partitions.
/dev/xd0, /dev/xd1, ...: These refer to external SCSI devices.
/dev/tty*, /dev/cu*, /dev/pty*: These devices refer to serial devices such as serial terminals, ports and modems.
/dev/lp1, /dev/lp2, ...: These devices refer to parallel devices such as centronics printers.

/etc: This directory, the et-cetera directory, contains configuration files - mostly in text, and also some programs for use by the system administrator. Most UNIX system management is done by editing a text file using a normal text editor (vi, nedit, emacs, pico). Even when you are using the fancy graphical administrative tools that many UNIXes use, they just end up editing these text files. Some important /etc files are:

/etc/passwd: contains the user passwords and login information.
/etc/aliases: other names for users and groups - particularly for redirecting e-mail.
/etc/mtab: The mounted file system table.
/etc/sendmail.cf: mail configuration.
/etc/TIMEZONE: The timezone which this machine is currently in. UNIX machines all use GMT as a base time, and keep track of local time by having a timezone configuration. The timestamp put on a file when you save it is always the GMT time, not your local time.
6.8. FILE SYSTEM STRUCTURE

/lib: This directory contains shared libraries. For example, the common C routines for doing mathematics, cout, cin and so on are stored in a file called /lib/libc.a or /lib/libc.so. It is commonly known as the C library. Any executable which needs to use routines in the C library will load it at runtime. It is common for UNIX programs to load five or more different libraries at run time. As a result, UNIX executables can be smaller, and can be updated by just changing the libraries.

It is also possible to bind in the libraries with the executable. For example, motif is a library (libXm.so), which you normally have to pay for. However if you bind the motif library into the executable, you can run the program without the library. This process is called static linking, and the executable is known as a statically linked binary. A binary of this sort is generally larger than a dynamically linked binary, and cannot be updated.

/home: This directory is sometimes used for user’s home directories, though this is one of the least standard directories. On manu, user home directories are found at /usr/users.

/tmp: This directory is used to store temporary files. For example, editors may keep backup files or intermediate files of some kind here. It is common for scripts to create temporary files in /tmp and then delete them later on. The directory /tmp should normally be placed on a partition with lots of free space. Nothing critical should happen if all the files in /tmp are deleted, and sometimes UNIX system managers delete files found in /tmp to free up space on a disk. All users can normally read and write to the directory /tmp:

drwxrwxrwt 10 root system 8192 Apr 23 11:42 tmp

Note that we call programs by other names sometimes - executable and binary are both used to refer to a program which will execute on a UNIX system.
CHAPTER 6. CASE STUDY: UNIX

/usr: Files and directories found in the root directory are considered essential to UNIX operation. A UNIX system would be unable to boot successfully if any of these base files were removed from the top levels of the file system hierarchy. The files and directories found in /usr are considered extra, and most UNIX systems have more extra files than they have base files. Though a minimal UNIX system might only have 100 or so files, it is common for systems to have thousands of files, even without user files. The /usr hierarchy often mirrors the / hierarchy.

/vmunix: Also in the root directory you will find the operating system kernel in a file called vmunix. Most UNIXes can boot any kernel (specified by name), so if an operating system upgrade fails, you can always revert to the old version by booting with the old kernel.

```
manu> ls -l /vmunix
-rwxr-xr-x 1 root system 8841552 Jun 2 1998 /vmunix
manu>
```

6.9 Environment variables

Each instance of a shell process has its own set of local variables. Some of these variables are available for use by shell scripts and programs. They are identified by name, and called environment variables. Environment variables may be added or deleted as needed, and there are normally few restrictions placed on their contents. They normally contain short text strings.

```
manu> printenv
DISPLAY=opo.usp.ac.fj:0.0
TERM=vt102
GWSOURCE=telnet
HOME=/usr/users/macs/anderson
SHELL=/usr/local/bin/tcsh
USER=anderson
LOGNAME=anderson
PATH=/usr/users/macs/anderson/bin:/usr/bin/X11:/usr/bin:/usr/sbin:/sbin:/usr/local/bin
HOSTTYPE=alpha
VENDOR=dec
OSTYPE=osf1
MACHTYPE=alpha
SHLVL=1
PWD=/usr/users/macs/anderson
GROUP=users
HOST=manu.usp.ac.fj
REMOTEHOST=opo.usp.ac.fj
MAIL=/usr/spool/mail/anderson
```

---

8The machine opo at USP has over 2000 separate executables, including over 20 text editors, 50 graphics and image editors, hundreds of graphic conversion programs and so on.
6.9. ENVIRONMENT VARIABLES

6.9.1 Retrieving - getenv()

When a shell starts up another shell, it *inherits* the parent environment including all the variables. When a shell starts up any process, that process can access the environment variables using the getenv system call.

```c
#include <stdlib.h>
char *getenv( const char *name);
```

The `getenv()` function searches the environment list for a string of the form `name=value`, and returns a pointer to a string containing the corresponding value for name. If such a string is present. If such a string is not present, a null pointer is returned.

The returned string should not be modified by the application, but may be overwritten by a subsequent call to the `getenv()` or `putenv()` functions.

6.9.2 Altering - putenv()

Process can alter the environment variables using the `putenv` system call.

```c
#include <stdlib.h>
char *putenv( const char *name);
```

The `putenv()` function sets the value of an environment variable by altering an existing variable or by creating a new one. The new environment variable remains in effect even after the program setting it is exited. The string parameter points to a string of the form "name=value", where name is the environment variable and value is the new value for it.

The `putenv()` function uses the `malloc()` function to enlarge the environment if needed.

Upon successful completion, a value of 0 (zero) is returned. If the malloc() function is unable to obtain sufficient space to expand the environment, the `putenv()` function returns a nonzero value.

6.9.3 Examples

The `PATH` environment variable normally indicates to the shell which directories to search when trying to find a program to execute. If you wished to change it you might do something like this:

```
manu> print-  
env PATH /usr/users/macs/anderson/bin:/usr/bin/X11:/usr/bin:.  
manu> setenv PATH /usr/users/macs/anderson/binnew:$PATH  
manu> print-  
env PATH /usr/users/macs/anderson/binnew:/usr/users/macs/anderson/bin:/usr/bin/X11:  
manu>
```
CHAPTER 6. CASE STUDY: UNIX

The **TERM** environment variable normally is set to the type of terminal or pseudo-terminal you are using.

```bash
manu> printenv TERM
vt102
manu> setenv TERM adm42
manu> printenv TERM
adm42
manu>
```

The **MANPATH** environment variable contains a list of extra directories to search when looking for manual pages.

The **SHELL** environment variable gives the pathname of current shell in use.

```bash
manu> printenv SHELL
/usr/local/bin/tcsh
manu>
```

The **DISPLAY** variable gives the current graphical X display. When you execute X applications, they use `getenv()` to find out this value, and then attempt to put the graphics onto that display. If you change **DISPLAY**, the graphics will be displayed somewhere else.

```bash
manu> printenv DISPLAY
opo.usp.ac.fj:0.0
manu>
```
6.10  OS processes

The UNIX operating system has a kernel for each of the operating system calls, but also has some software which must run all the time. These software modules are commonly called daemons, and each one has a well defined function.

If a daemon uses a configuration file, and you change the file, you must then signal the running daemon to re-read the file. This is commonly done by sending the daemon the HUP signal. In this way, you seldom have to re-boot a UNIX system.

6.10.1  Process initialization - init

The init program initializes the system by creating and controlling processes. The processes run by init at each run level are defined in the inittab file. The init program reads and processes the entries in the inittab file. The run level changes when a privileged user invokes init. The new init sends appropriate signals to the original init that tell it which run level to change to.

Running the init program is the last step in the boot process after the root file system is mounted. Here is a line in the inittab file which reruns a getty process on the console of the machine if it ever stops running.

```
s3:3:wait:/sbin/rc3 < /dev/console > /dev/console 2>&1
cons:1234:respawn:/usr/sbin/getty console console vt100
```

6.10.2  Socket multiplexer - inetd

The inetd daemon should be run at boot time by inetd in /sbin/init.d. It then listens for connections on certain Internet sockets. When a connection is found on one of its sockets, it decides what service the socket corresponds to, and invokes a program to service the request. After the program is finished, it continues to listen on the socket. Essentially, inetd allows running one daemon to invoke several others, reducing load on the system.

Upon execution, inetd reads its configuration information from a configuration file, which, by default, is /etc/inetd.conf. There must be an entry for each field of the configuration file, with entries for each field separated by a tab or a space. Comments are denoted by a # (number sign) at the beginning of a line. Here are some lines for /etc/inetd.conf on manu.

```
ftp stream tcp nowait root /usr/local/sbin/tcpd ftpd -l
telnet stream tcp nowait root /usr/local/sbin/tcpd telnetd
shell stream tcp nowait root /usr/local/sbin/tcpd rshd
login stream tcp nowait root /usr/local/sbin/tcpd rlogind
exec stream tcp nowait root /usr/local/sbin/tcpd rexecd
whois stream tcp nowait root /usr/local/etc/whois++d whois++d -l
```
6.10.3 Terminal management - getty

The getty command sets and manages terminals by setting up speed, terminal flags, and the line discipline. If command flags are provided, getty adapts the system to those specifications. getty prints the login prompt, waits for the user to enter a username, and invokes the login command.

The getty command uses the /etc/gettydefs file for terminal information.

- The line argument refers to the device name in /dev.
- The speed_label argument is a pointer into the /etc/gettydefs file where the definitions for speed and other associated flags are located.
- The terminal argument specifies the name of the terminal type.
- The line_discipline argument specifies the name of the line discipline.

6.10.4 Update file system - update

The update command executes a sync system call every 30 seconds. This ensures that the file system is up to date in the event of a system crash.

6.10.5 Logging events - syslogd

The syslogd daemon reads and logs messages to a set of files described in the /etc/syslog.conf configuration file.

Each message logged consists of one line. A message can contain a priority code, marked by a number in angle braces at the beginning of the line. The syslogd daemon configures when it starts up and when it receives a hangup signal. Here are some sample entries from the /etc/syslog.conf file on manu:

```
kern.debug /var/adm/syslog.dated/kern.log
user.debug /var/adm/syslog.dated/user.log
mail.debug /var/adm/syslog.dated/mail.log
daemon.debug /var/adm/syslog.dated/daemon.log
auth.debug /var/adm/syslog.dated/auth.log
syslog.debug /var/adm/syslog.dated/syslog.log
lpr.debug /var/adm/syslog.dated/lpr.log
```
6.10.6 Server for telnet - telnetd

The telnetd daemon is a server that supports the DARPA (Defense Advanced Research Projects Agency) standard telnet virtual terminal protocol. It is invoked by the Internet server for requests to connect to the telnet port as indicated by the /etc/services file. The -debug flag may be used, to start up telnetd manually. If started up this way, port may be specified to run telnetd on an alternate TCP port number.

6.10.7 RPC multiplexer - portmap

The portmap daemon is a server that converts Remote Procedure Call (RPC) program numbers into Defense Advanced Research Projects Agency (DARPA) protocol port numbers. It must be running in order to make RPC calls.

When an RPC server is started, it will tell portmap what port number it is listening to, and what RPC program numbers it is prepared to serve. When a client wishes to make an RPC call to a given program number, it will first contact portmap on the server machine to determine the port number where RPC packets should be sent.

6.10.8 Various servers:

nfsd  The nfsd daemon runs on a server machine to service NFS requests from client machines. The daemon spawns a number of server threads that process NFS requests from client machines. At least one server thread must be running for a machine to operate as a server.

named  The named daemon is the Internet domain name server. See RFC's 1033, 1034, and 1035 for more information on the Internet domain name system. Without any arguments, named will read the default boot file /etc/named.boot, read any initial data and listen for queries.

lpd  The lpd line printer daemon (spool area handler) normally is invoked at boot time. The daemon makes a single pass through the printcap file to determine the existing printers and to print all files that were not printed before the system shut down.

6.11 UNIX fun

You may wish to try the following commands:

    manu> cat "food in cans"
    cat: cannot open food in cans
manu> who is my match?
who: No match.

manu> make mistake
Make: Don’t know how to make mistake. Stop.

manu> make “sense of all of this”
Make: Don’t know how to make sense of all of this. Stop.

manu> got a light?
got: No match.

manu> !:say, what is saccharine?
Bad substitute.

and so on.
Chapter 7

Synchronization and Communication

In a multiprocess environment, there are a whole new class of problems.

Concurrency can be studied all on its own without reference to operating systems - although it is in them that most of these things raise their ugly heads.

Perhaps the first problem is: “How can processes communicate with each other?”

One technique involves allowing each program access to a shared memory area. However the access to that area must be very carefully controlled. The bank account problem demonstrates the sort of errors that can occur with shared memory areas. We call these sort of problems critical section problems.

In addition, some programming languages have started to add the structures needed for parallel processing. They often appear as additions to existing languages.

7.1 Deadlock

We have already seen situations which can cause deadlock without special care. In operating systems these are mostly related to concurrent resource allocation. There are four major strategies to deal with deadlock:

• Head in sand
• Detection
• Prevention
• Avoidance
All of these strategies except the first strategy are awkward to implement, but there are solutions to some of them.

**For example:** The bankers algorithm is often given as an example of deadlock avoidance. Enough resources to supply any one of the clients is kept at all times. If there are conflicting client requirements, the one that can complete is used. The bankers algorithm minimizes the amount of resources that must be kept free.

Most operating systems use the first strategy - in general they just hope not to get in a locked state. Applications themselves may attempt to avoid deadlock.

### 7.2 Parallelism

Parallelism may be pseudo-parallel as in multiple processes on one machine, or real parallel processing - often seen in graphics processors and channel I/O. It is useful to have some scheme to express parallelism in a programming language, and there are some languages expressly written for this (OCCAM etc). One simple addition to *Algol* was the `AND` construct, which expressed the concurrent execution of statements:

```plaintext
a[I]:=b+c and b[1]:=c+d;
```

Both statements could be executed at the same time.

Also in this line are the fork and join constructs (Conway [1963]) and parbegin, parend (Dijkstra [1965]).

Ada was designed with real time constraints in mind, and includes constructs to assist in writing concurrent task software. The tasks may communicate by using the `accept` and `select` statements. These statements provide mechanisms for both communication and synchronization.

### 7.3 Critical section problem

We can identify the following conditions to ensure that we avoid critical section problems:

1. No two processes simultaneously in their CS
2. No assumptions made about relative process speeds
3. No process outside its CS should block other processes
4. No process should wait for ever to enter its CS
The following schemes all handle critical section problems in inter-process communication to some degree or another:

- Disable interrupts
- Lock variables
- Strict alternation
- Semaphores
- Event counters
- Monitors
- Message passing

### 7.3.1 Monitors

Monitors have been proposed as a solution to the critical section problem. A monitor is a group of procedures, only one of which may execute at any one time. If other processes attempt to call procedures within the monitor section, they are blocked until the other process has exited it. The variables within the monitor are hidden from all outside procedures.

```plaintext
monitor example;

var
  local: integer;

procedure dec( x:integer);
begin
  local := local-x
end;

procedure inc( x:integer);
begin
  local := local+x
end;

end monitor;
```

### 7.3.2 Interprocess communication

One way of looking at these problems is to describe them as examples of communication between processes which wish to co-operate. There are two basic methods of inter-process communication:
1. Shared memory

2. Messaging

The two methods are not mutually exclusive - we could if we want implement messaging using shared memory. Two fundamental primitives for messaging systems are:

\texttt{send(P, message)}: Send a message to process \( P \)

\texttt{receive(id, message)}: Receive any message (\( id \) is set to source process)

### 7.3.3 Disable interrupts

The simplest method of ensuring that only one process at a time may enter its critical section is to disable context switching during the critical section:

\begin{verbatim}
begin
  DisableInterrupts;
  ProcessCriticalSection;
  EnableInterrupts;
  ...
\end{verbatim}

The obvious disadvantage with this is that all the other processes are inhibited from running while this process does its critical section. Most of the other processes of course could be running throughout this time.

### 7.3.4 Turn variables

Another simple solution is to provide access alternately with a turn variable:

\begin{verbatim}
begin
  while turn<>myturn do
    (wait );
  ProcessCriticalSection;
  turn := otherturn;
  ...
\end{verbatim}

The problem with this is of course just that the processes may not wish to enter their CS’s alternately. Let’s try again, with two processes and two booleans to indicate if the processes are inside or outside of their CS (both initially false):
7.3. CRITICAL SECTION PROBLEM

begin
  while OtherProcessInside do
    {wait};
  MeInside := true;
  ProcessCriticalSection;
  MeInside := false;
  ...

Again this is not a foolproof solution. Mutual blocking would not occur, but both processes could enter their CS if they both arrived at the test at the same time. We can see that the solution to the critical section problem is not trivial!

The first published solution to this problem was proposed by a mathematician (T. Dekker) in the early 60s, and requires two interlocking variables, one to ensure that mutual blocking does not occur, and one to ensure that mutual exclusion does not occur. The solution is a little hard to understand and in 1981 Peterson proposed a simpler solution:

var
  turn : processID;
  Interested : array [processID] of boolean;

procedure EnterCS( proc:processID);
  var other: processID;
begin
  other := OtherID( proc);
  Interested[proc] := true;
  turn := proc;
  while (turn=proc) and Interested[other] do
    {wait}
end;

procedure LeaveCS( proc:processID);
begin
  Interested[proc] := false
end;

7.3.5 Semaphores

Dijkstra proposed another scheme in 1965 which allowed the construction of CS software. He introduced a variable type called a *semaphore* with only two operations that could be performed on them:

V(S) S is increased by 1 in an indivisible operation

P(S) If S>0 then decrement S (again an indivisible operation) otherwise wait until it can be done.
CHAPTER 7. SYNCHRONIZATION AND COMMUNICATION

The P and V operations are sometimes known as DOWN and UP in English. These simple operations are relatively easy to code, and provide all the mechanism needed to synchronize processes. The V call may activate other waiting processes, the P operation may sleep the calling process.

Ps and Vs may be constructed using interrupt masking, or using some of the TAS (Test and Set) instructions on many computers.

For example: the 6809 computer could use the LSR and LSL instructions to indivisibly test and set memory locations. The 68000 and the 80386 support indivisible test and set bit instructions (BTAS). All of these may be used directly to implement binary semaphores, or in conjunction with other variables to implement full semaphores.

The busy wait on the P can waste processor time, and the general technique is to maintain queues of processes waiting on each semaphore.

Semaphores are found in most operating systems, with operating system calls to initialize, lock, unlock and dispose of semaphore variables.

```c
#include <sys/mman.h>

msemaphore *msem_init( msemaphore *sem, int initial_value );
int msem_lock( msemaphore *sem, int condition );
int msem_unlock( msemaphore *sem, int condition );
int msem_remove( msemaphore *sem );
```

**msem_init(sem,initial_value):** allocates and initializes a new binary semaphore. If `initial_value` is MSEM_LOCKED, the new semaphore is initialized in the locked state. If `initial_value` is MSEM_UNLOCKED, the new semaphore is initialized in the unlocked state.

**msem_lock(sem,condition):** attempts to lock a binary semaphore. If the semaphore is not currently locked, it is locked and the function returns successfully. If the semaphore is currently locked, and the `condition` parameter is MSEM_IF_NOWAIT, then the function returns with an error. If the semaphore is currently locked, and the `condition` parameter is 0 (zero), then msem_lock() will not return until either the calling process is able to successfully lock the semaphore, or an error condition occurs.

**msem_unlock(sem,condition):** unlocks a binary semaphore.

**msem_remove(sem):** removes a binary semaphore.

We can use these semaphore variables to protect variables by declaring one semaphore for each protected variable, and using the calls in the following way:

```c
#include <sys/mman.h>

int variable1;
```
7.3. CRITICAL SECTION PROBLEM

```c
msemaphore lock1;
int variable2;
msemaphore lock2;

void *thread1(void * thread1args)
{
    ...
    msem_lock( &lock1, 0 );
    ... do critical stuff on variable 1 ...
    msem_unlock( &lock1, 0 );
    ...
}

... and similarly for other threads

main()
{
    msem_init( &lock1, MSEM_UNLOCKED );
    msem_init( &lock2, MSEM_UNLOCKED );
    ... set up threads ...
}
Chapter 8

Computer networking

The principal reasons for computer networks are resource sharing, saving money and reliability. The principal applications are:

- **remote programs** (rather than multiple copies)
- **remote databases** (as in the banking system, airline reservations)
- **value added communications** (service)

We also have widely varying speed requirements in computer networks, and both tight and loose couplings between machines, and hence there is no one solution to the computer networking problem.

Computer networks range from small systems interconnecting chips on a single PCB, up to world wide computer networks. There are many possible ways of connecting the machines together, (topologies), the main ways being the star, ring and bus systems. In general the larger networks are irregular and may contain sections with each of the above topologies. The smaller networks are generally regular.

![Diagram of Star, Ring, and Bus topologies](image)
The largest computer network in the world is the Internet\textsuperscript{1} which interconnects over 1,000,000 computers worldwide. The stability, management and topology of this network varies. In some areas, it is left up to University departments - in others there are strong commercial interests.

Computer networks rely on following standards. \textit{The nice thing about standards is that you have so many of them. Furthermore, if you do not like any of them, you can just wait for next year’s model.}

- The Electronic Industries Association: EIA
- The Institute of Electrical and Electronic Engineers: IEEE
- The American National Standards Institute: ANSI
- The International Organization for Standards: ISO
- The Consultative Committee on International Telephone and Telegraph: CCITT

### 8.1 Media

The transmission medium is just the medium by which the data is transferred. The type of medium can have a profound effect. In general there are two types:

- Those media that support point to point transmission
- Multiple access systems

One of the principal concerns is ‘\textit{how to make best use of a shared transmission medium}’. Even in a point to point transmission, each end of the transmission may attempt to use the medium and block out the other end’s use of the channel.

### 8.2 OSIRM

This model is dated, but provides a framework to hang your understanding of computer networking on. It is modelled on IBMs SNA (System Network Architecture) but with differences.

\textsuperscript{1}Note that you should always capitalize the word \textit{Internet} if you are referring to this ‘internet’. If you are referring to any ‘interconnected network’, you can use \textit{internet}. 
What are the advantages of this?

- We can change layers as needed, as long as the interface remains constant.
- The network software is easier to write, as it has been modularised.
- Client software need only know the interface.

Each layer is defined in terms of a **protocol** and an **interface**. The protocol specifies how communication is performed with the layer at the same level in the other computer. This does not for instance mean that the network layer directly communicates with the network layer in the other computer - each layer uses the services provided by the layer below it, and provides services to the layer above it. The definition of services between adjacent layers is called an **interface**. Remember -

- a **protocol** specifies the interaction between peer layers,
- an **interface** specifies the interactions between adjacent layers of the model.

### 8.2.1 The layers

**Physical Layer**

The physical layer is concerned with transmission of data bits over a communication line - that is the transfer of a `1` level to the other machine. In this layer we worry about connectors, cables, voltage levels etc.

**Datalink Layer**

The datalink layer is concerned with the (hopefully) error-free transmission of data between machines. This layer normally constructs frames, checks for errors and retransmits on error.
CHAPTER 8. COMPUTER NETWORKING

NETWORK LAYER

The network layer handles network routing and addressing. It is sometimes called the packet switching network function. At this level are performed line connection and termination requests. X.25 is the internationally accepted ISO network layer standard. IP is the Internet network layer standard.

TRANSPORT LAYER

The transport layer provides an interface between the ‘network communication’ layers (1..3) and the higher service layers. This layer ensures a network independent interface for the session layer. Since there can be varying network types, ISO provides for five levels of ‘Class of service’ ranging from a simple connection manager (class 0) to a full error and flow control manager (class 4). You would use class 0 when communicating over a PSDN, and class 4 when using a PSTN.

The Internet protocol suite has two common transport protocols:

- **TCP** - Transmission Control Protocol
- **UDP** - User Datagram Protocol

The transport layer isolates the upper layers from the technology, design and imperfections of the subnet.

APPLICATION LAYER

The application layer provides the user interface to the network wide services provided. For example file transfer, electronic mail. Normally this layer provides the operating system interface used by user applications.

8.3 Standards and APIs

Most network programming involves using an API which provides a *session layer* service. The API provides a set of system calls, and software which implements some *view* or *abstraction* of the network.

It is normal to use these programming *abstractions* when doing network programming. They allow you to model the behaviour of the system, and understand its behaviour. Before programming using one of these APIs, you need to understand the abstract model, *not* the underlying protocols.
8.3. STANDARDS AND APIS

8.3.1 Netbios

Netbios\(^2\) was developed by IBM. It is an interface to low level networking software, and through the API, you can send and receive messages to and from other Netbios machines. The Netbios calls on a PC are through Interrupt 5c, and require your software to initialize and maintain an NCB\(^3\) before calling the network software.

8.3.2 Sockets

The ‘socket’ is an abstraction for each endpoint of a communication link. The abstraction allows data communication paths to be treated in the same way as UNIX files. When a socket is initialized, the API system calls return a file descriptor number which may be used to read and write in the same way as those returned by file opening calls.

The processes to handle services such as these are started as needed using the UNIX fork command. The code looks like this:

```
repeat
  x := IncomingConnection;
  result := fork ();
  if result = childprocess then
    processchild(x)
  else
    close(x)
  until TheWorldComesToAnEnd!
```

The general method for using sockets is as follows:

- Before referencing the socket - create it:
  ```
  int socket(int sock_family, int sock_type, int protocol);
  ```

- Bind socket to a local address:
  ```
  int bind(int sock_descr, struct sockaddr *addr, int addrlen);
  ```

- Clients may issue a connect call:
  ```
  int connect(int sock_descr, struct sockaddr *peer_addr, int addrlen);
  ```

- Servers must listen:
  ```
  int listen(int sock_descr, int queue_length);
  ```
  and then accept incoming connections:
  ```
  int accept(int sock_descr, struct sockaddr *peer_addr, int *addrlen);
  ```

\(^2\) A networking ‘BIOS’. BIOS is the term used on IBM personal computers for the low level software that hides the underlying hardware in the system. It is often found in a PROM. The letters stand for Basic Input Output Subsystem.

\(^3\) Network Control Block.
Remote Procedure Calls

The RPC system was introduced by Sun, but unfortunately there are variations in its implementation. The intent is to provide a system in which networking calls look and act just like procedure calls. The program prepares parameters for the procedure call, and then makes a call to a stub procedure. The stub procedure uses the RPC runtime software to transfer the parameters, only returning when the call is complete.

RPC is responsible for ensuring that the parameter data is transferred and returned correctly.

8.4 Application layers

The application layer provides the user interface to the network wide services provided. Normally this layer provides the operating system interface used by user applications. Some sample standards:

SNMP - The Simple Network Management Protocol is a protocol for remote network management. It is commonly used to monitor the behaviour of remote systems.

SMB - Server Message Blocks are a Microsoft/IBM protocol for file servers, commonly known as Windows networking.

NCP - The Netware Core Protocols are Netware’s file system protocols.

DNS - The Domain Name Service supports the naming of entities on an IP network.
8.5 Distributed objects

Distributed objects have made it to the desktop (most notably in the form of Java applets - processor independant, reasonably efficient and safe). If we expect our objects to interoperate successfully in a distributed system, we have to agree on a set of standards. The two (incompatible) standards are:

1. OMG’s CORBA
2. Microsoft’s DCOM/OLE

CORBA object interfaces are specified using an IDL (Interface Definition Language). The IDL is language and implementation independant, and specifies object interfaces unambiguously. CORBA has been adopted by every operating system supplier except for Microsoft - who are promoting a different standard. However CORBA object extensions for Microsoft products are available from third parties.

8.6 Network operating systems

The generic term ‘Network Operating System’ (NOS) describes a uniform view of the services on a network. A NOS may involve all sorts of middleware running on different platforms, but tying them together with agreed protocols.

Example: - We may use a global time service - accessible from all systems on the local network. It is common to use NTP or a similar product to ensure clock synchronicity within an enterprise.

The NOS functions are often provided on-top-of or bundled with conventional OS’s such as NT or UNIX.

Note: The NOS view of the enterprise computing environment may become obsolete. Web services, and distributed object technologies are doing similar things, and have already spread to the desktop.
Chapter 9

Memory management

There are in general two types of memory management schemes:

- those that swap memory to and from disk (virtual memory systems), and
- those that do not.

The schemes that do not swap memory are the simplest, and often need only the simplest hardware.

For example: An early operating system (OS/360 MFT - Multiprogramming with a Fixed number of Tasks), used fixed partitions set up by the operator in the morning. Each process would be allocated one partition to run in, and would have access to that entire partition for the duration of its run. The IBM/360 computer had a 4 bit lock attached to each 2K of memory, and a 4 bit key in the PSW had to match the lock before access was gained. User processes could not alter the PSW, and this gave complete interprocess protection.

However we are mostly interested in the more sophisticated virtual memory (VM) systems.

9.1 Virtual memory

Virtual memory is a technique that allows the execution of programs that may not reside completely in memory, by using disk to act as memory. VM systems provide a virtual address space much larger than the physical address space.

For example: If we had 100 MB of physical memory and 300 MB of disk acting as memory, we could have a total of 400 MB active at one time. We could even run programs that needed 350 MB (larger than the physical memory, and larger than disk 'memory').

The part of the disk used in this way is called the 'swap' - it may be either a
• file used for the purpose, or
• a partition of disk.

It is common for VM systems to have a swap area two to ten times as large as the physical memory.

The memory is managed a page at a time, where a page is commonly 1024 (1k) bytes. Page information is maintained in a page management table by the operating system. The memory manager can then search this table to find memory using any of the following methods:

1. First Fit
2. Best Fit
3. Quick Fit

VM systems require specialised software and hardware. A (hardware) memory manager examines all references to memory, and translates them to an actual physical address. If the memory manager finds that the physical memory is not available, the instruction is held and the required memory page is fetched from disk (if it has been used before) and placed into some slot in physical memory. These systems require the ability of the processor to hold up an instruction midway through, and roll it back to restart later.

Note: The MOTOROLA 68000 had no facility to roll back an instruction, and so was incapable of supporting a VM system. The 68010 and 68020 however were able to do this, and the VM support chips and software are available for these processors.
When a VM system attempts to find a page, and does not find one, it is known as a page fault. Obviously it is in the operating system’s interest to minimize page faults, as each page fault delays the execution of the process. There are quite a few page replacement algorithms:

1. NRU - Not Recently Used
2. FIFO - First In First Out
3. LRU - Least Recently Used
4. NFU - Not Frequently Used

Most VM systems use one or the other of these algorithms, but it is always possible to find some situation in which the selected algorithm is not optimal.

When a processor spends all its time shuffling pages to and from disk, it is said to be thrashing. A good algorithm which is almost impossible to implement would be the “least amount of thrashing” algorithm.

Another solution to the page fault problem has been the development of the working set. Here a data structure is kept which maintains all the pages used by the program. These pages are preloaded before the program runs, and this normally inhibits further page faults. The working set is maintained by the operating system.

**9.2 Memory management unit**

When the CPU puts out an address, it is examined by the MMU. Either

1. the MMU finds the address in its PMT (Page Management table) and translates it to a physical address, or
2. the MMU fails to find the address. In this case, the instruction is stopped half way through, and
   
   (a) if the physical memory is full, a page is swapped out to disk (leaving room to fetch the new page), and
   
   (b) if the new page has been accessed before, fetch it from disk and fill out the table, otherwise fill out the table, and then
   
   (c) the instruction is restarted.
 CHAPTER 9. MEMORY MANAGEMENT

Implications of this scheme:

1. If you don’t use a page it will eventually swap out to disk. Seldom used processes will then have a short start up delay as the pages are retrieved from disk.

2. To be efficient we must
   (a) minimise swapping, and
   (b) ensure frequently used pages are in memory, and
   (c) ensure infrequently used pages are swapped to disk.

Note: On UNIX, you can type ’ps -ef’ to see the state of processes. If memory used by a process is on disk it will have a status of IW.

Some additional points:

1. If you don’t refer to memory pages, they don’t get allocated, so we can run programs much larger than physical and swap.

2. The memory allocation scheme is independent for the program (noone has to write special code into their program to handle VM).

3. There are systems that can never run VM.

4. The best page allocation strategy is the one that minimises page faults - easy to say, hard to do.
9.2. MEMORY MANAGEMENT UNIT

9.2.1 Dirty bit

A simple optimisation to VM systems is for the MMU hardware to keep track of memory that it writes to. If a VM system retrieves a page from disk, and then does not write to that page in memory, then there is no need to save it back to disk if it has to be swapped.

The effect of this is to dramatically reduce the number of disk accesses needed in VM systems, since most VM pages are not written to.

The MMU has a single bit known as the dirty bit for each page that it manages. When you load a fresh page from disk, the dirty bit is cleared. When (or if) the CPU writes to a page, the dirty bit is set. When the OS wants to swap that page out to disk, it examines the dirty bit, and only writes if the dirty bit is set.

9.2.2 Content addressable memory

VM systems are one of the few applications of CAM - Content Addressable Memory.

Note: If our page size is 1024 bytes, the address bus is split, with the bottom 10 bits going directly to the memory and the upper address bits to the MMU. If the CPU had 32 address bits, 22 bits would be fed to the MMU to handle $2^{10}$ pages of each of $2^{10}$ bytes. If we had CAM, it would have to be 22 bits wide.

The size (length) of the CAM determines the size of physical memory. If our CAM had $2^{16}$ locations each of 22 bits, we could address $2^{16}$ pages of real memory (65 M) - in a virtual address space of $2^{22}$ pages.
9.2.3 Example: the Pentium TLB

The TLB or translation lookaside buffer is the hardware found in an Intel CPU which directly supports virtual memory.

To map memory, we use page descriptor tables. These tables tell us

- *where* in physical memory our pages are, and
- *what* their status is.

For the Pentium CPU, we have the following system:

1. The segment register identifies an entry in the descriptor table in memory. Each entry is 8 bytes, and there are 8192 entries.

2. The entry is used to generate the address, along with the low order bits, or to hold the instruction.

This of course would be very slow, so a cache of recently used descriptor table entries is maintained. On the Pentium the TLB has room for 32 8-byte entries.

1. The TLB is searched first, and if an entry is found it is used.
2. Otherwise the (external) descriptor table is used - updating the TLB.

The TLB uses CAM to achieve high speed. (We want to know if one of 8192 entries is found in a 32 Entry Table). When the TLB fails, the system reverts to the slow mechanism.

This system works fine in single-processor systems without a large number of contexts, but if we have a large number of contexts, each with lots of mapped VM pages, we have a large number of TLB failures.

In the Pentium, the TLB is inside the processor, and in a multi-processor system there are cache coherency problems.

### 9.2.4 Page replacement schemes

Some schemes:

**Working set:** The OS keeps track of all the pages and resources used by a process each time it runs. When you run a process these pages are preloaded. Most of the time they will be the only pages needed. (This is a system used by IBM mainframes - it also helps avoid deadlock problems).

**LRU - Least Recently Used:** This is a scheme to determine which pages to swap out to disk. The MMU keeps a time field associated with each page in memory. Each time the page is accessed this time field is updated. When the OS is looking for a page to swap to disk it uses the one with the oldest time field. The assumption with LRU is that if you haven’t accessed a page for a long time it will be less likely you will want to refer to it in the near future. This assumption may be incorrect in some cases.

**NRU - Not Recently Used:** This method involves setting a single bit whenever the page is accessed (like the dirty bit). Every 20 sec or so all the bits are cleared, and if a page bit did not need clearing another NRU bit is set. The NRU bit tells us if the page has been accessed in the last 20-40 secs. **Note:** this state is not necessarily the oldest page - it is just one of a number of pages that haven’t been used for at least 20 sec. NRU is a very common page placement algorithm.

**FIFO - First In, First Out:** This scheme can be done without hardware assistance. When a page is allocated the time is stored (in ordinary memory). When the OS is looking for a page to swap, it looks in that table, and removes the oldest page.

**NFU - Not Frequently Used:** A counter is kept for each page. Each time the page is accessed the counter is incremented. Every 5 sec the counter is halved. (This is a simple smoothing algorithm to implement in hardware - the binary number is shifted right by one). The page with the lowest count is swapped.


9.3 **Segmentation**

Segmentation can be thought of as a kind of two-dimensional addressing scheme. In paging, the machine has no idea what each page is used for. In segmented addressing, the compiler or assembler make some attempt to allocate segments for code, segments for data and so on. The memory hardware can often be quite simple, inhibiting any accesses outside each segment, and differentiating between segment use for code, segment use for data and so on.

Segmentation and paging often coexist - the MULTICS system had a segment table which was itself paged!

9.4 **Fragmentation**

In a linear memory processor, the operating system takes care not to fragment the memory too much. There are three main methods used for storing the memory allocation data:

1. **Bit maps** - storing the available memory space in an array of bits, each bit standing for some number of bytes.

2. **Lists** - search the free list for a suitable size segment, and when finished return the memory to the free list. This method is quick to search for memory, but adjacent blocks do not automatically recombine.

3. **Buddys** - uses a property of binary numbers to get the speed of table lookup of a list, and the recombining properties of the bit map.

We have met each of these in reference to the similar block allocation problem on disks.

9.5 **Linking and loading**

Compilers and assemblers generally produce relocatable object modules that are processed by a linker (linkage editor). The linker produces a relocatable binary which is tied to the final load addresses by the the loader. Sometimes a linking loader will do the second two steps at once.

Linking can be done at different times:

1. Before compilation
2. At compilation time
3. Between compilation and loading
4. At load time

5. At execution time

The object module has three sections:

1. A definition table (exported labels)

2. A use table (imported labels)

3. The code or data

The definition and use tables are just lists of the labels and their offsets in the code segment. The code segment contains ‘holes’ where there are unresolved references.

The linker reads all the object modules, and builds up a symbol table which it uses to resolve all the references and build the relocatable binary module. If you load in the object modules in a different order, you get a different relocatable binary, which acts in exactly the same way.
Chapter 10

Distributed systems

A distributed system is:

- A collection of computers, linked by a network, comprising an integrated system.

A distributed operating system is:

- The software for a D.S. which:
  - supports convenient program development
  - provides other software components with effective abstractions of the D.S. resources.

We are concerned with the general qualities of:

- **TRANSPARENCY**: concealing for users and processes the distributed nature of the system.
- **FAULT TOLERANCE**: how to cope with failures of all sorts.
- **DISTRIBUTED SERVICES**: Naming services, timing services, file systems, transaction processing (with atomicity) and so on.
- **SECURITY**: Logins and inter-process communication.
10.1 Topics in distributed systems

10.1.1 Fault tolerance

Some terminology first:

- **RAS** - Reliability, availability and serviceability (An IBM term)
- **MTTF** - Mean time to failure
- **MTTR** - Mean Time To Replace

The availability of a system is calculated with: \( 1 - \left( \frac{MTTR}{MTTF} \right) \), and we categorize availability into the following classes.

<table>
<thead>
<tr>
<th>Availability</th>
<th>Class</th>
<th>Downtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>90%</td>
<td>1</td>
<td>52,560 min/year</td>
</tr>
<tr>
<td>99%</td>
<td>2</td>
<td>5,256 min/year</td>
</tr>
<tr>
<td>99.9%</td>
<td>3</td>
<td>525 min/year</td>
</tr>
<tr>
<td>99.99%</td>
<td>4</td>
<td>52 min/year</td>
</tr>
<tr>
<td>99.999%</td>
<td>5</td>
<td>5 min/year</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

In the case of nuclear reaction monitoring, we might consider class 5 availability, but for an in-flight computer we might need class 9 (only 30 ms downtime).

It is common to use multiply-redundant systems (such as RAID) to ensure better availability. However, sometimes more is not better! We have to examine more closely the behaviour of these systems.

One simple centralised scheme is to use the simple voting system:
A voting system might have three inputs, and select the values which occur on a majority of the modules. There is no point in having two modules - you must have three or more. If the voter can detect faulty modules, the above system can withstand one failed module (It still needs two to get a consensus). In general, systems of this sort can cope with $t < \frac{N}{2}$ failed modules.

It is useful to view fault tolerance as being correctly operating processes protecting themselves against failed processes. A failed process may exhibit one or more of the following failure modes:

- **Always dead** - it executes no steps of its local algorithm.
- **Crashed** - process stops executing steps at some time
- **Byzantine** - process executes arbitrary steps of any kind

In a synchronous system, we may also have:

- **Timing** - process executes the correct steps at the wrong times

One area of interest is to look at algorithms in which each correctly operating process can decide something.

- No asynchronous deterministic algorithm for consensus exists that can tolerate even a single process failure.
- There are deterministic algorithms for some other classes of problems.
- Probabilistic algorithms for consensus can be constructed which can tolerate up to $\frac{1}{2}N$ simple failures, or $\frac{1}{3}N$ Byzantine failures.

### 10.1.2 Migration and load balancing

Some distributed operating systems allow the migration of processes from machine to machine. This may be done to reduce the load on a heavily loaded machine. This ability places constraints on the operating system, in that processes can have no processor dependency.

We may also be concerned with balancing the load on the processors. This normally involves monitoring processor load continuously.
10.2 Case study: Mach kernel

The mach microkernel is from CMU (Carnegie-Mellon University). The development started in 1985 and the kernel is considered stable. The original design goal was to:

- Retain compatibility with BSD UNIX
- Support distributed UNIX

Older versions of MACH placed the UNIX support inside the kernel. However, since mach Vs 3.0, all of this has been removed to a user-level compatibility library. (UNIX, MS-DOS, OS/2, VMS).

The kernel is found in OSF/1 (The Open Software foundation UNIX - DEC/ALPHA), NEXT workstations and is often retro-fitted to UNIX clusters. It is common to find mach where there are large numbers of UNIX workstations.

What’s not in the kernel:

- **a file system** - MACH file support is dependent on the user level compatibility library.
- **comms** - The kernel has messaging calls, but the datacomms is done by a user-level process. However, it is normal for TCP/IP to be used as a transport.

There are about 100 kernel calls in mach, for:

- **Tasks & Threads** - which may reside in different memory maps.
- **Tasks** - use heavyweight context switching.
- **Ports** - Anything you can communicate with is accessible through ports with associated buffering (queues).
- **Memory** - Both local virtual memory and support for Distributed Shared Memory (DSM).
- **Devices** - Just as in UNIX /NT calls to devices are made through the kernel.
- **Messaging** - simple calls to do messaging.

One unusual behaviour of mach messaging implemented at the kernel level is that messages may contain **pointers**! The kernel support for DSM allows pointers to still be valid on another machine using copy-on-write semantics.
10.3 Case study: Amoeba

Amoeba is a general purpose D.O.S. developed by Andrew Tanenbaum and others over the last 15 years. Most developmental work on it has ceased, but it is still maintained and used. It is primarily a research tool, but there are commercial uses too.

Amoeba comes with all its sources, and a UNIX compatibility library. Over 100 common UNIX utilities have been ported, so it is possible to use the machines in a very ‘UNIX-like’ way. Workstations may run X, and Amoeba uses its own high-speed communication protocol (FLIP).

When you run a program it runs on the lightest loaded processor in the processor pool. The network must be responsive to the demands of this sort of system, and so the FLIP protocol has been developed to ensure fast communication on a local LAN, for up to thousands of nodes.

10.3.1 Capabilities

All Amoeba objects are identified by a capability string which is encrypted using DES encryption. A capability is long enough so that you can’t just make them up.

If you have the string, you have whatever the capability allows you. If you want to give someone some access to a file, you can give them the capability string. They place this in their directory, and can see the file.

All AMOEBA objects are named/identified by capabilities with four fields:
To further prevent tampering, the capability is DES encrypted. The resultant bit stream may be used directly, or converted to and from an ASCII string with the \texttt{a2c} and \texttt{c2a} commands.

\textbf{10.3.2 Pool processors}

The capabilities for pool processors are stored in the Amoeba directory \texttt{/super/pool}, grouped by processor type.

\textbf{For example:} At USP, under \texttt{/super/pool/sparc}, are found the four machines \texttt{john}, \texttt{paul}, \texttt{george} and \texttt{ringo}. Limited copies of these capabilities are made available to each user in their \texttt{/profile/pool} directories.

A special server called the run server is responsible for allocating processes to suitable processors. The run server can run many pools at the same time. You request a run server to manage a directory containing processors with the makepool command. It places a capability called \texttt{.run} in that directory. You may query the status of the pool like this:

\begin{verbatim}
amoeba% std_status /profile/pool/.run
HOST STAT SINCE LREPL LPOLL NPOLL NBEST NCONS MIPS NRUN MBYTE
ringo UP 1d03h 1s 1s 19729 171 1630 11.700 0.000 10.816
gorge UP 1d03h 3s 3s 19732 170 1630 11.700 0.000 11.392
john UP 1d03h 2s 2s 20053 690 1630 14.000 0.36 8.704
paul UP 1d03h 2s 2s 19995 599 1630 14.000 1.024 9.190
\end{verbatim}

This tells us that \texttt{john} and \texttt{paul} are the most popular processors!

\textbf{10.3.3 Directory structure}

The Amoeba directory structure is not at all bound to the structure of the \texttt{files on disk}. A separate server (The SOAP server) maintains the \texttt{name/capability} relationships, independant of the items
found in the structure.

The SOAP server provides a directed graph structure - it often has 'loops' in it:

Each user may see a hierarchy something like that shown above. Note that there are links back to the user's root directory.

The system administrator sees other hierarchies:

- /super/unixroot - the unix files
- /super/admin - administrative files
- /super/cap - capabilities of system services (the boot server and so on)
- /super/hosts - the hosts

The users have (limited) links into this hierarchy - For example users have /profile/cap access.

### 10.3.4 Multiprocessor systems

In Amoeba systems, we may have a heterogeneous mix of SPARC, 680X0 and Intel-486 machines all acting as pool processors. How do we handle this?

There are several points:

1. The compilers generate code for any or all of the target architectures - so you may create binaries for each of the machines you are using.

2. If you have an executable `pload`, it is accessed through a directory called `pload` in your search path, containing binaries for any or all architectures you intend to use. You may find in your bin directory:
The run server is able to select appropriate pool processors depending on load, and availability of executables for the necessary architecture.

Amoeba in operation is similar to UNIX:

```
Telnetsvr {
    after Session_svr;
    machine SLASH "super/hosts/john";
    bootrate 29000;
}
```

### 10.3.5 Boot server

The boot server is the first process started in Amoeba. It is responsible for starting all other services. Its configuration file is directly loaded (i.e. - not through a bullet server), however there is a copy at:

```
/super/admin/bootinfo/Bootfile.vivian
```

Here is a segment of a typical file (from USP’s Amoeba system):
pollrate 30000;
program {
    SLASH "super/admin/bin/telnetsvr";
    argv {
        "telnetsvr", "-T", "john"
    };
    capv {
        STDIN, STDOUT, STDERR,
        ROOT = SLASH, WORK = SLASH,
        _SESSION = SLASH "dev/session"
    };
};

This segment ensures that the telnet daemon server is continually running on machine john.

The boot server continually checks and restarts processes if they fail. You can check the status of the processes the boot server monitors with:

```
std_status /super/cap/bootsvr/vivian
```

At USP, the boot server maintains the following servers:

- **Soap** - the directory server - runs on machine vivian.
- **Random** - a random number generator, in built in the kernel - it is found on machine vivian.
- **Login** - a login process is run on vivian to handle ‘console’ operations on the serial port.
- **Session** - a session server for handling remote logins. On vivian.
- **Run** - this server manages the system-wide process scheduler - it runs on paul.
- **TOD** - an inbuilt kernel server - we use the one on john.
- **IP** - the IP gateway runs on machine john.

### 10.3.6 Parallel make

The utility `make` is a tool for compiling large systems.

**For example:** If you had 100 C files and 30 header files, you can either recompile everything every time you make a change to any one file, or you can use `make` to define the dependencies
between the files, and only recompile those files that have to be recompiled. This is done by checking the times of the source and object files along with the dependencies.

In a distributed environment, the conventional make system has some drawbacks - in particular, the time value may be an inappropriate measure for identifying if files have changed. Different machines may have different times.

The parallel make utility uses a different check method to determine what to compile, and also attempts to do actions in parallel if possible. In the Amoeba system, amake uses the capability of the files to see if they have changed (if you edit a file, its capability string changes). We keep pairs of capability strings for the source, and target files and don’t recompile if they haven’t changed.

10.3.7 Amoeba commands

Amoeba has a series of UNIX-like commands, however it also has a large number of Amoeba-specific commands, and extended UNIX commands - aman and aps given below are Amoeba versions of man and ps. Here is a selection:

**ppload** - return the load on each processor in your pool directory:

```
amoeba% ppload
paul load 0.27, 14.0 Mips, 8.8 Mb free, up 1 day 4 hrs
john load 0.04, 14.0 Mips, 8.7 Mb free, up 1 day 4 hrs
ringo load 0.11, 11.7 Mips, 10.8 Mb free, up 1 day 4 hrs
```

**aman** - Amoeba manual pages:

```
amoeba% aman ls
ls(U) User Program  ls(U)
Name
ls - list the contents of a directory
Synopsis
   ls [-1ACFRacdfgilrstu] name ...
```

**aps** - shows the process status on all machines:

```
amoeba% aps -mv
POSIX  S
HOST   PS  PID  T  OWNER  COMMAND
john   3   R  (getowner: null port)  session -a /bin/sh
paul  7 156  R  session               server -sh
paul  4 191  R  session               server aps -mv
```

**amake** - the parallel make command
stun - kill a process

gax - manage a parallel program

std_* - the std_* commands are standard commands that can be applied to all managed objects:

```
amoeba% std_status /super/cap/bootsvr/vivian
configuration ok
NAME      FLAGS      INFO
Random_cap 1         "random number server" Sat May 30 22:04:12 1998
Tod_cap   1          "TOD server"                      Sat May 30 22:07:14 1998
```
Chapter 11

Security

The security of a computer system is dependant on all sorts of factors beyond the operating system. There is just the simple security of locks on the doors to computer rooms. Large scale events can be unavoidable (Terrorist attack/Tsunamai/Earthquake/fire/power failure) and an assessment of the relative likelihood of each of these events combined with how much should be spent to protect the system is needed.

11.1 Levels of rights

Many operating systems provide levels of rights. For example: a user may have the rights to look in a certain range of directories and read the contents. Alternatively it may be done by looking at the resource and saying ‘this resource may be used by these classes of users’. Many operating systems apply a combination of these two strategies.

0S/9 protects files with eleven booleans attached to each file. They are used for the read, write and execute permissions for each of the user, the group and everyone else. The other two booleans are for shareable and directory.

11.2 External and internal security

Security can be both an external and an internal problem. Operating system software should in general check all calls for validity to ensure that incorrectly running programs do not damage the system.

A well known technique to protect the computer is to ensure that the user of the computer has the right to use it! Several schemes suggest themselves - password protection is obviously one of the most common - there are also interactive schemes in which the computer asks the user for
a number according to some pre-agreed scheme. This second method would be different every
time the user logged in, and so cannot be cracked by snooping.

**For example:** the user and the computer could agree that the permuting function might be 'divide
the number given into the time and date (reversed) and return the remainder'. If the date was
16:40 12/05/1990 and the given number was 5, it would be divided into 099150210461, and the
remainder of 1 would be returned as the password.

The common password protection which we use is dangerously easy to overcome. In 1979
students at a university in the US conducted a test of the security of the student UNIX system.

One of their experiments was to compare the (encrypted) list of passwords on the UNIX machine
with an encrypted list of words which they generated from a dictionary. They were able to
discover 86% of the passwords, including the main supervisory ones. This same study outlined
the main access points to break into computer systems:

- lowly paid employees
- undocumented system calls
- system calls called with incorrect parameters

Sometimes other features of the operating system can completely destroy the security:

**For Example:** The TENEX (DEC 10) operating system had a completely (?) secure password
protected scheme, and also had a handy utility which could display paging faults for you. Pro-
grammers found that if you wrote a program which attempted to use a resource, and passed a
password with just its first letter before a page boundary and the second letter after the page
boundary, the program would either

- fail because the first letter was wrong, or
- display a page break and then fail because the second letter was wrong.

All the letters A-Z were tried and then the password was moved back by one byte and the second
letter searched. This technique could discover the ten letter password in about 130 tries (normally
it would be 26 to the power of 10). The fix just involved a tiny change to the password scanning
routine.

### 11.3 Shared key systems

Shared key systems are generally considered inadequate, due to the difficulty in distributing keys.
11.3. SHARED KEY SYSTEMS

11.3.1 Ciphertext

The ciphertext systems encode the input stream using a substitution rule:

<table>
<thead>
<tr>
<th>Code</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Q</td>
</tr>
<tr>
<td>B</td>
<td>V</td>
</tr>
<tr>
<td>C</td>
<td>X</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The S-box (Substitution-Box) encodes 2 bit numbers to other 2 bit numbers and can be represented by the permutation (3,4,2,1). This is an S-box:

Ciphertext systems are easily breakable particularly if you know the likely frequency of each of the codes. In the English language, the most common letters are:

ETAOINShrdlu

(From most to least common)
11.3.2 Product ciphers

We have seen two types of cipher: If you use both types at once, you have a product cipher which is generally harder to decode, especially if the P box has differing numbers of input and output lines (1 to many, 1 to 1, many to 1).

DES - Data Encryption Standard

First proposed by IBM using 128 bit keys, its security was reduced by the US NSA (National Security Agency) to a 56 bit key (presumably so they could decode it in a reasonable length of time. At 1ms/guess. It would take $10^{80}$ years to solve 128 bit key encryption. The DES Standard gives a business level of safety, and is a product cipher.

The (shared) 56 bit key is used to generate 16 subkeys, which each control a sequenced P-box or S-box stage. DES works on 64 bit messages.

Note: If you intercept the key, you can decode. There are about $10^{17}$ keys.

11.3.3 Public keys

A machine can publish K1 as a public key, as long as it is not possible to create K2 from K1.

We can use this to provide authentication as well. If one machine wants to authentically transmit information, it encodes using both its private key and the recipient’s public key:
The second machine uses the others public key and its own private key to decode.

**RSA (Rivest, Shamir, Adelman)**

This public key system relies on the properties of extremely large prime number to generate keys.

**To create public key** $K_p$:

1. Select two large primes $P$ and $Q$.
2. Assign $x = (P - 1)(Q - 1)$.
3. Choose $E$ relative prime to $x$. (This must satisfy a condition for $K_s$ given later)
4. Assign $N = P \times Q$.
5. $K_p$ is $N$ concatenated with $E$.

**To create private (secret) key** $K_s$:

1. Choose $D$ such that $\text{mod}(D \times E, x) = 1$.
2. $K_s$ is $N$ concatenated with $D$.

**We encode plain text** $P$ by:

1. Pretend $P$ is a number.
2. Calculate $c = \text{mod}(P^E, N)$.

**To decode** $C$ **back to** $P$:

1. Calculate $P = \text{mod}(C^D, N)$.

**We can calculate this with**:

```plaintext
c := 1; \{ attempting to calculate mod(P^Q, n) \}
x := 0;
while x<>Q do
    begin
        x := x+1;
        c := mod(c*P, N)
    end;
```

At the end of this procedure, $C$ contains $\text{mod}(P^Q, N)$.
Chapter 12

Case study: NT

NT is a Microsoft 32-bit proprietary multi-tasking, single-user operating system. NT was written in C and C++, from the ground-up. It does not carry a thirty year history as UNIX does - however it does carry around some DOS history. NT version 3.51 has a C-2 security classification from the US government as long as it is not connected to a network - this gives some level of confidence in its use.

The operating system is available in two configurations, which have the limitations shown in the table below.

<table>
<thead>
<tr>
<th></th>
<th>NT workstation</th>
<th>NT server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client connections</td>
<td>10</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Symmetric multiprocessing</td>
<td>2 cpus</td>
<td>4 cpus</td>
</tr>
<tr>
<td>RAS (Dial in modem)</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>Disk fault tolerance support</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Macintosh services</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

NT is available on several platforms, including Intel, and also the DEC Alpha processor. All processor dependent code is found in a DLL known as HAL - the hardware abstraction layer.

12.1 Architecture

NT has a conventional layered architecture, with a microkernel, HAL, an executive, various programming APIs, and multi-threaded subsystems.

In 1997, for efficiency reasons, Microsoft moved the graphics subsystem from a user mode subsystem into the kernel. The graphics subsystem is a large section of code, and the resultant increase in size of the kernel mode software has reduced the reliability of NT.

---

1NT originally stood for New Technology.
2Most UNIXes also have this level of security classification even when they are connected to a network.
12.1.1 Layering

The core component of NT is a microkernel similar to Mach. One or more operating system subsystems can be layered over the microkernel - at the same time if desired. This means that NT can run multiple operating systems simultaneously. There is an OS/2 subsystem, a POSIX subsystem, and the WIN32 subsystem.

The OS/2 subsystem can be used to run OS/2 on-top-of NT.

The POSIX subsystem contains many of the standard UNIX system calls.

Application programmers normally use the WIN32 subsystem when developing programs, particularly because WIN32 NT applications often run directly on Windows 95 (which also has the WIN32 API - albeit missing some functions). A Windows 95 WIN32 application will nearly always run on NT.

12.1.2 WIN32

The WIN32 file API uses handles for almost all system objects. Handles are similar to UNIX file descriptors, and can be passed around to routines, or inherited from parent processes and so on.

The WIN32 subsystem has a CreateProcess() call for creating new processes. This system call has 10 arguments - there is no equivalent to the UNIX fork() or execv() calls. It is not possible to overlay one process with another as in UNIX.

The WIN32 API allows runtime linking of libraries. On an NT system the libraries all have the DLL extension. These libraries are much more restricted in their behaviour than UNIX libraries - not supporting versioning, or any way to override a DLL call, or even any way to directly access data.

For character based applications, WIN32 supports a simple console window which is similar to the UNIX terminal I/O systems, although the single user nature of NT fixes the console window on the local screen display - there is no facility for redirection. For graphically based applications, the WIN32 graphical API provides a set of calls to manipulate windows on the local screen display.

12.1.3 Memory

The operating system uses a linear 32-bit address space, with a demand-paged virtual memory system. Previous Microsoft operating systems relied on a memory model consisting of segments of 64k of memory, but NT allows applications to directly address up to 2GB of memory.

The NT paging algorithm optimizes swapping by keeping track of pages associated with a particular process, and attempting to swap process pages together.

\[\text{DLL stands for dynamic linked libraries.}\]
12.2 Accounts

NT uses the familiar concept of users - with passwords, account names, home directories and rights. Each account can belong to various groups, which may give a range of access rights.

12.2.1 Groups

An NT group is an alias for a set of users, along with permissions and user rights. If you change the permissions for a group, all members of that group inherit the changes automatically. This facility is useful for administration - you can organize your users into groups (sales, marketing...), and then you only need to administer the one group account.

The default group accounts on an NT system are:

**guests**: This group gives a strictly limited access to the machine.

**Users**: This group is for normal use of the computer, with member accounts able to run applications and manage files.

**Power users**: This group can perform extra administrative tasks.

**Administrators**: This group can perform any administrative task - it is similar to the root account on UNIX systems.

**Replicator**: This is a special purpose account used when replicating files and directories across a network of NT systems.

**Backup operators**: This group can perform system backups.

12.2.2 Security

Microsoft use the jargon security policy to refer to schemes for managing access to resources. There are three security policies:

1. **Account policy** - managing password aging and length, and password lockouts.

2. **User rights policy** - managing rights associated with user and group accounts.

3. **Audit policy** - managing types of event to record in the audit logs.
12.3 Networking

NT comes standard with support for a range of protocols. These protocols are implemented as loadable drivers:

SMB: The Server Message Block protocol is principally used for file serving.

IP: IP is the Internet standard protocol set - arguably the most common networking protocols in use today, and there is support for TCP and UDP over IP. NT can also recognize multicast IP.

NetBIOS: A low level abstraction for small networks, developed for DOS systems and still supported. NetBIOS applications offer their services using a NetBIOS name which consists of 15 characters plus a 'type' character.

NetBEUI: A small protocol for use on small networks - upto 254 machines.

Named pipes: A connection oriented message system.

12.3.1 UNC

NT systems use named pipes extensively, and the names follow the UNC (uniform naming convention) format:

- \SERVER-NAME\SHARE-NAME\...

The SERVER-NAME is the name of the machine providing the other end of the pipe, the SHARE-NAME identifies the particular service provided, and this may be followed by directory and file names. The UNC system is suitable for use on small networks, but not on extended networks due to its restricted naming system for hosts.

12.3.2 Domains and workgroups

A domain and a workgroup are different in that a distributable authentication database is associated with a domain, for secure login access over a network. Different access rights can be granted to users if they successfully authenticate against a domain logon server.

Workgroup authentication is on a machine-by-machine basis.

SMB networking provides a mechanism by which clients can access a list of machines that are available within the network. This list is called the browse list, and is found on one machine termed a browse master. Periodically, elections are held to determine which machine gets to be the browse master.
12.4  File system

Windows NT supports three file systems, HPFS, NTFS and FAT.

**FAT:** The window system found on Windows 95 is similar to the older DOS files system, except that it allows 255 character long file names.

**HPFS:** This window system is found in OS/2.

**NTFS:** The NT native file system is similar to the Berkeley UNIX file system, with long names.

Access to these file systems is through the WIN32 API, which does not provide support for case sensitivity in directory names and files, and inhibits files from starting with the strings `aux`, `com1`, `com2` and `nul`. Certain characters are not available for use in file names.

NT text files follow the older DOS convention of using both a carriage return and a newline at the end of a line of text. UNIX uses a newline character only. When transferring files between UNIX and DOS systems, some applications will automatically translate for you, but some do not.

12.5  Summary

NT is a modern single user multi-tasking operating system. It is currently undergoing continuous development, and we can expect to see significant improvement in its behaviour over the next few years.

NT is significantly better than any previous Microsoft operating system in virtually any area, providing a true 32 bit API, protected process operation, VM and a microkernel. The operating system has been ported to at least four platforms.

Though NT server has found a market in providing servers for small to medium businesses, it is limited by its single-user orientation, lack of support for distributed graphics, and (so far) a hard limit of 8 processors per SMP machine.

---

4 WinCenter is a third party product which provides extensions to NT which support multi-user use.

5 WinFrame is a third party product which provides extensions to NT which support distributed graphics.
Chapter 13

Window systems

13.1 X

The X window system is a sophisticated and well developed system which allows for multimedia software and hardware components to be distributed around a network. At its simplest level, it allows a program and its display to be on different computers.

When the X window system was developed, there were clearly defined design goals:

1. Mechanism - not policy. This meant that the system supplies a way of distributing graphics, but it does not dictate exactly how you use the system.

2. If you can get 90% of what you want with 10% of the work - do it!

The architectural view of X is a little peculiar. The designers view the display as central to the system, and the software running on the display is called the X-server:

---

1The system is called X, or the X window system. It is not called X-windows!
From the diagram, we can easily identify three essential components of X:

1. **The X server** - providing the high resolution graphical display(s), keyboard and mouse.

2. **The X protocol** - providing standard communication methods between the distributed software components.

3. **X clients** - programs with graphical display output, running on (perhaps) many machines.

However, there are two other components:

- **The Window manager(s)** - providing decorations around each window.

- **The Display manager(s)** - providing access to the system.

### 13.1.1 Display Manager

The display manager controls access to displays. The above diagram shows a simple display manager allowing you to select one of a number of hosts. When you select a host, you are presented with a login prompt for that particular host.

Sample logins:

```
Welcome to oso
IBIX 6.3
Login:  high
Password:  
```

There are of course other display managers.

---

2 Mechanism, not policy!
13.1.2  Window managers

There are many different window managers, ranging from the 'bleak' twm:

![Image of twm window manager]

The fvwm window manager:

![Image of fvwm window manager]

You don’t actually have to have a window manager (but then you can’t resize or move the windows with the mouse...).

![Image of fvwm window manager]

There are even Win95 lookalikes:

---

³Fvwm is a lookalike for the Motif window manager.
13.2 Thin client systems

A relatively recent development involves moving the X-server (or equivalent) from the machine with the display to a larger machine, and then using a smaller computer to actually display the data. Here is a thin client written in Java, running as an applet in a web page: