INTRODUCTION TO OPERATING SYSTEMS

Les Robertson

CERN, Geneva, Switzerland

These notes were prepared as support material for two lectures entitled "Introduction to Operating Systems" given at the CERN School of Computing at Bad Herrenalb, Federal Republic of Germany, in August 1989. The first part covers some of the basic concepts and design issues of operating systems. The second part deals with distributed operating systems, including a few general aspects of architecture and design and giving brief descriptions of some commercially available distributed systems.

The notes are in part related to two excellent text books which are recommended for further study.

- *Operating Systems – Design and Implementation*, Andrew S. Tanenbaum, Prentice-Hall, 1987
- *Distributed Systems – Concepts and Design*, George F. Coulouris and Jean Dollimore, Addison-Wesley, 1988

WHY USE AN OPERATING SYSTEM?

An operating system performs two main functions: it provides the user with a *virtual machine* more convenient to use than the physical hardware; it manages the various resources of the computer configuration[1], sharing them out among programs and users.

The basic hardware architecture of most machines and peripheral devices is complicated and tedious to program, the details often being dictated by factors like hardware construction techniques or compatibility with long-obsolete predecessor systems, rather than with ease of use. In addition, all sorts of error conditions may arise which must be carefully recorded (in order to enable the maintenance engineer to fix the underlying problem) and recovered from (as in many cases the problem will only be intermittent). The operating system handles all of this detail, presenting the user with a high-level interface, the *virtual machine* which provides both an *extended set of instructions* (like *read_next_block*) and a standardised architecture (where hardware details which differ from one model of a computer to another, like the exact mechanism for initiating an I/O operation, may be hidden from the user).

Computers, even simple single-user systems, must be able to support more than one program. The operating system must organise the sharing of the processors, memory, disks, printers, network channels, and all the other resources of the configuration among the programs which are executing simultaneously on the machine. In the case of multi-user systems additional resource management tasks become apparent, such as protection and the implementation of resource sharing and accounting policies (such as disk space budgets, interactive response time goals, batch priority). Where the computer system is expensive, the resource management must also be concerned with maximising utilisation and throughput.

In addition to the operating system, there are many other system programs which are usually provided by the computer manufacturer, and which give additional functionality to the user. These include compilers, editors, mail handlers and command interpreters. These are not part of the operating sys-

---

[1] central processers, main memory, extended memory, disks, tape units, communications connections, printers, terminals, etc.

tem. They are programs which use the operating system and which differ from application programs only in that they are (generally) not written by the end-user.

SYSTEM CALLS

The interface between a user program and the operating system is defined as a set of software instructions known as *system calls*. The system call is usually implemented through a special instruction which causes control to be passed to a routine in the operating system which decodes the actual function. Sometimes, as in the case of the UNIX operating system, the system calls are defined as high-level language procedures in order to mask from the user program the details of the hardware mechanism, and thus improve the portability of the system.

PRINCIPAL FUNCTIONS

The main functions of the operating system may be grouped into the following areas.

- Process and processor management
- Input/output control (including networking)
- Memory management
- File system

We shall deal with each of these areas in more detail in the following sections.

PROCESS MANAGEMENT

A *process* is a *program in execution.* It includes the executable program itself, together with the current values of all the information which defines exactly what the program is doing (such as the data, stack contents, program counter, stack pointers and other registers). Whereas the *program* is *static* (it consists only of the code and some initial data values), the *process* is *dynamic*. The process may be *executing*, when all of the data describing its state will be loaded into the appropriate hardware, or it may be *suspended*, when the state data may have been copied into system tables. This data will have to be loaded into the appropriate hardware again prior to resuming execution of the process.

The process management function is responsible for creating processes, scheduling them into execution on real processors (dispatching), providing for inter-process synchronisation, maintaining process hierarchies, etc. Process management is the innermost mechanism of the operating system, providing the basic support for the parallel programming which is essential to the other parts of the system.

In the following discussion we shall consider that the operating system is implemented as a set of processes which execute on a number of physical processors, sharing a common memory. More than one process can therefore be executing simultaneously, using the same memory and other resources. We shall look at some of the synchronisation problems which arise and some of the solutions. Note that, even on a mono-processor system, synchronisation is a problem if the process dispatcher re-schedules the system (changes the executing process) without an explicit request from the running process. For example, a *pre-emptive* algorithm may be used, where a process will be interrupted if a higher priority process becomes executable (such as after an I/O interrupt).

Mutual Exclusion

Within the operating system there are many tables and lists which have to be updated by different processes. For example, suppose that requests are passed to a disk driver process through a list with a

fixed number of slots. Processes wishing to have an I/O request performed insert data describing the request in the next free slot, and the driver process removes entries from the head of the list. The list is implemented as a circular buffer, controlled by two pointers: *OUT* defines the next entry to be processed (head of the list), and *IN* defines the next free entry. In order to add an entry, a process reads *IN*, stores the data and increments *IN*. Imagine that two processes decide at more or less the same time to add an entry. The first reads *IN* and begins to store the request in the slot. Just at this moment there is an interruption for some reason and the dispatcher decides to reschedule the system. The first process is suspended and the second begins to execute. It now carries on with inserting its request in the list, reading *IN* and overwriting the data stored by the first process. The result is at best undefined.

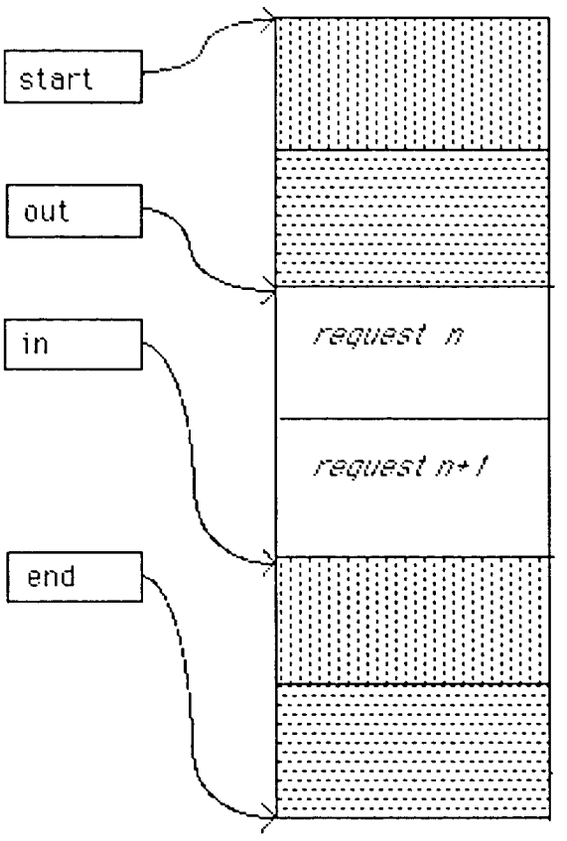


Figure 1: Driver request list implemented as a circular buffer

The sections of code in which the processes fill in an entry in the list must be executed in a *mutually exclusive* manner. They are called *critical sections*. For a given set of associated critical sections, only one process may be inside its critical section at any time.

There are various ways of implementing critical sections in operating systems, the following being two of the basic methods.

- *interrupt masking*
  This is sufficient only in the case of a mono-processor, but is often used for very short critical sections because it is very efficient (usually a single instruction).

- *test_and_set instruction*
  Most multi-processor systems have a form of test_and_set instruction in which a memory word is read into a register and, if its value is zero, a non-zero value is stored in the memory location. While the instruction is executing all other access to the memory word is blocked.

  The memory word may be called a *lock*, and the test_and_set instruction, if successful, *obtains the lock* for the process.

Critical sections can now be implemented using the *enter_region* and *leave_region* procedures in Figure 2. These, however, involve *busy waiting* and so they are not appropriate for implementing critical sections for processes executing on the same processor (it may be the higher-priority process which is looping in *enter_region*). In practice, *enter_region* would first disable interrupts, and *leave_region* would enable them again.

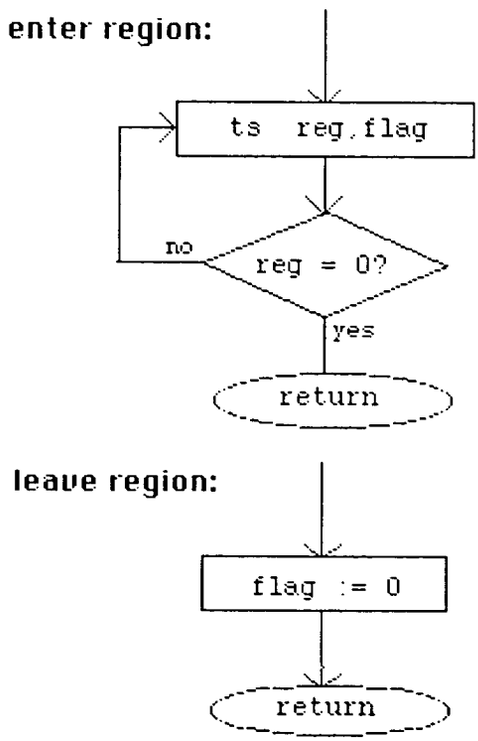


Figure 2: Procedures to implement Critical Sections

Serialisation

A more complicated situation than the simple list management one may make the identification of a critical section difficult. For example, a database system is used by processes which query (read) the data while other processes are modifying it. A modification (*transaction*) generally involves several database records and fields, and the database is *consistent* only before any of the modifications have been made, or after all have been made. A simple example: A process executes a transaction on a financial database which involves transferring a sum of money from field A to field B. At the same time a second process is querying the database in order to compute the sum of A and B. The sum computed by the second process may be short by the amount of the transfer, or may include it twice, depending on the order in which A and B are updated. Clearly we could solve this problem by mak-

ing all transactions (update or query) into mutually exclusive critical sections. The effect on performance, however, is obvious. The mutually exclusive critical sections are dependent on the set of data being updated, which is defined dynamically. The solution is to *serialise access to the data* through a mechanism called *two-phase locking*, in which locks to all the data items concerned are first obtained (phase 1) before any part of the transaction is performed. In the second phase, locks may only be released. Within an operating system, this technique is required in the areas of file system and disk space management, where a series of actions involves disk I/O and therefore obviates a simpler mutual-exclusion strategy.

Synchronisation

We consider again the case of the I/O request list, this time as an example of a *producer-consumer* problem. The requestor produces entries in the list, and the disk driver consumes them. We see that, in addition to the need for *mutual exclusion* while manipulating the list, we also need a mechanism for *synchronisation* of the two processes.

1. The requestor may produce entries faster than the driver can handle them, and the list may become full. The requestor must *wait* for an entry to be removed.

2. The driver may complete processing of all the entries in the list. It must then *wait* for an entry to be placed in the list.

A mechanism is required to enable a process to go to sleep (be suspended) and later be notified (and eventually resumed) when a certain event has occurred. This mechanism can also be used to avoid the busy-waiting problem which we saw earlier with a simple locking mechanism.

One of the earliest synchronisation mechanisms is Dijkstra's *semaphore* (first implemented in the T.H.E. system). A semaphore is a variable which is acted on by two primitives which he called *P* and *V*. P (sometimes called DOWN) is a kind of lock instruction. It tests the semaphore and if its value is 1 it sets it to 0, indicating that the process has obtained the lock. If it is already 0 the process is queued on a list associated with the semaphore and suspended. The V (or UP) instruction sets the semaphore to 1 (opens the lock) and removes a waiting process from the list. Figure 3 shows how this works. The P and V primitives must be implemented as *atomic* events (indivisibly), achieved in the example by means of a *test_and_set* lock called "mutex".

We can now use a semaphore to control a critical section without busy-waiting. But semaphores can also be used to synchronise the producer and the consumer in the problem of managing the I/O request list, as shown in Figure 4. We need three semaphores:

M: lock to obtain mutually exclusive access to the list, initialised to 1 (lock free).
E: synchronise availability of empty slots in the list, initialised to 1 (empty slots available).
F: synchronise availability of requests in the list, initialised to 0 (no requests).

An extension to the concept of semaphores is the implementation of *counting semaphores*, where the V operation increments the semaphore by one, and the P operation decrements it if it is not already zero. These can be used to control the list in our example more elegantly as follows:

E: is now initialised to the number of slots in the list.
F: is initialised to zero, and represents the number of full slots in the list.

We still require the binary semaphore M to control the actual update of the list itself. Figure 5 shows how this is done.

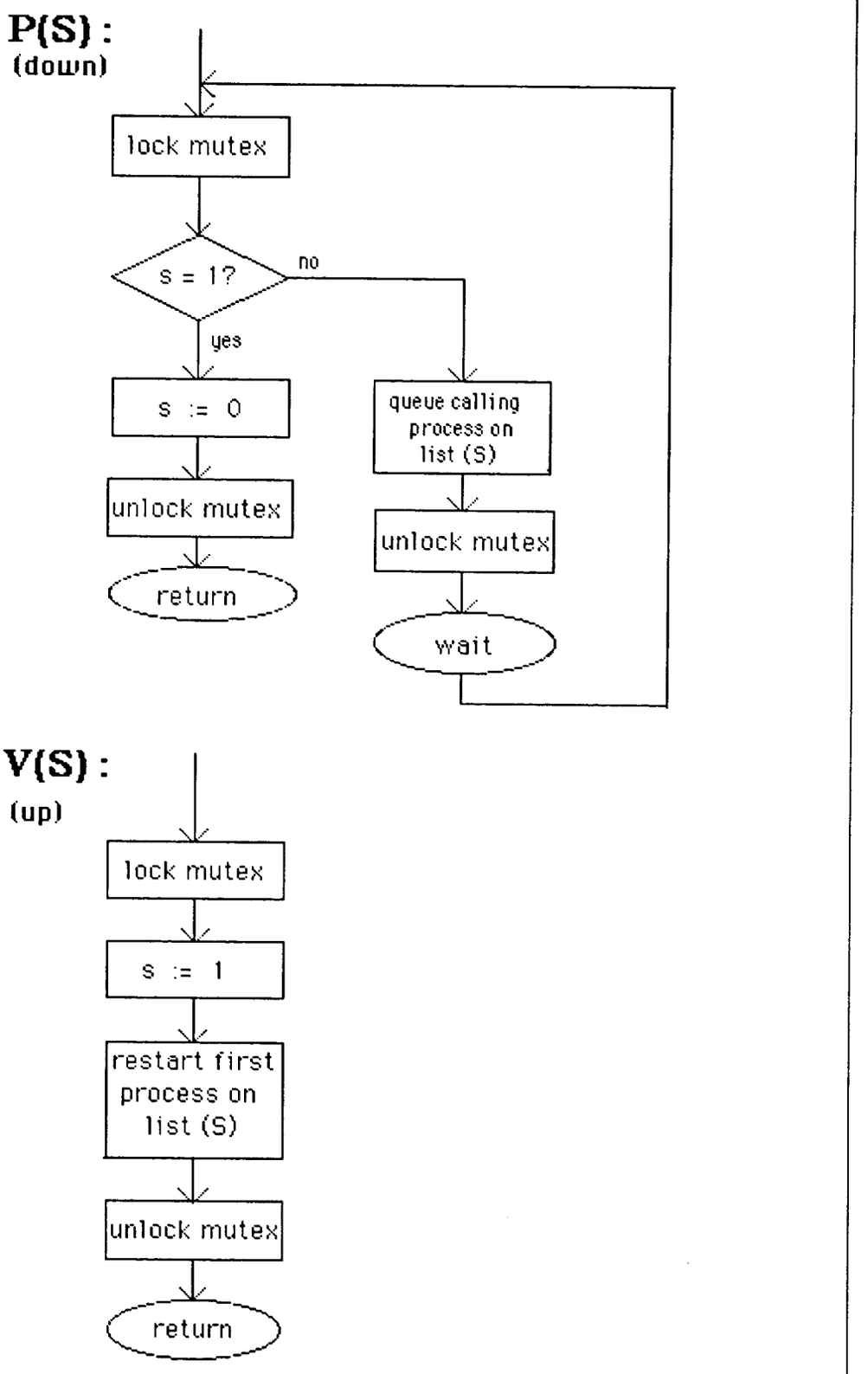


Figure 3: The P and V primitives of a binary semaphore

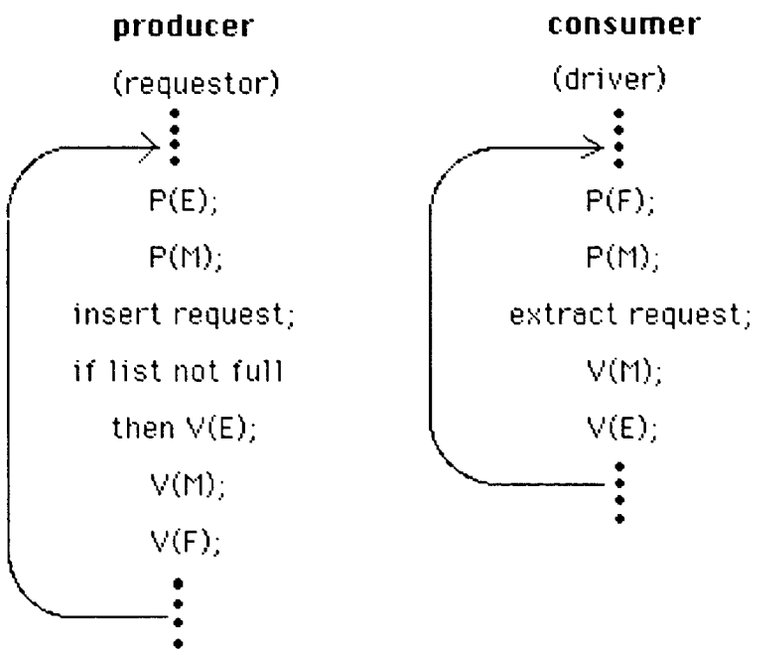


Figure 4: The Producer-Consumer problem using binary semaphores

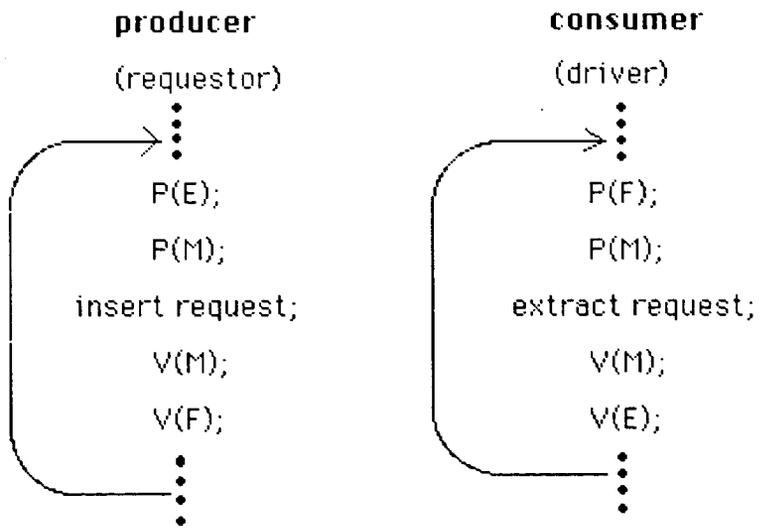


Figure 5: Producer-Consumer problem using counting semaphores

Using semaphores to handle inter-process communication is fairly error-prone. In our simple example we have used three semaphores, and it would not be difficult to make a mistake in the order of the P and V operations, which would go unnoticed during testing (where all possible asynchronous situations would not be available) but would be certain to cause mis-operation the first day that the system was introduced into production. In practice, an operating system will use standard functions to pass information through queues, which will relieve the system programmer from much of this burden.

With the growth in the use of parallel systems, applications programmers also have to organise synchronisation of processes. High-level techniques have been developed to help in this, such as *monitors* and the ADA *rendez-vous*. These, however, require the support of special language constructs and compilers and so are not of immediate application to designers of operating systems.

SYSTEM STRUCTURE

Monolithic Systems

In this (not uncommon) model, see Figure 6, the operating system is one large program which executes in *supervisor* or *kernel* mode, while the user processes execute in *problem* or *user* mode. Kernel mode allows privileged operations to take place. User processes are protected from each other and can communicate with the operating system only through system calls.

Monolithic systems do of course have some internal structure, but there is no mechanism within the operating system to prevent one part of it from accidentally overwriting another, or for *imposing* the designers' synchronisation rules. Such systems are very hard to test, and usually suffer from obscure bugs which are very difficult to reproduce.

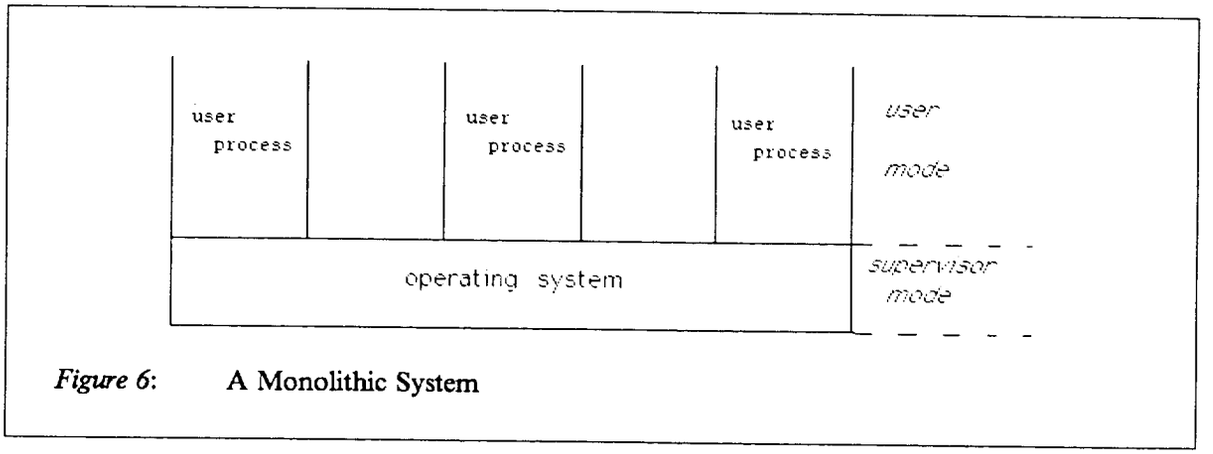


Figure 6: A Monolithic System

Layered Systems

One approach to the problem of designing well-structured systems has been to organise the operating system as a hierarchy of layers, each successive layer being built on the mechanisms provided by the underlying layers. For example, the T.H.E. system built by Dijkstra in 1968 was layered as shown in Figure 7. In that system, layer 0 dealt with process management, so none of the other layers had to worry about multiple processes. The T.H.E. system layers were only used as a design aid, because in the end the first four layers were linked together into a single program. The MULTICS system further generalised the concept of layers, being organised as a series of concentric *rings*, with privilege increasing towards the centre. Special hardware was used to provide protection between the rings. The VMS operating system is constructed as a series of layers with some degree of hardware protection (IPL levels).

Client-Server Model

A current trend is to move as much as possible of the operating system code out of the hostile asynchronous environment of the kernel and into user processes, leaving only process management, and the minimum of routines to interface to the privileged hardware instructions. A *client* process, a real user, calls a system function by sending a message to a *server* process (such as the file manager). The server responds by means of another message. From the point of view of the kernel, both client

and server are user processes.

In this *client-server* model, each part of the system handles only a limited set of functions, and so is much simpler to write and maintain. Executing as an application program, the consequences of malfunctioning are limited.

This model is readily adapted to distributed systems, as shown in Figure 9. The client is not concerned with the actual location of the server (apart of course from considerations of performance and reliability).

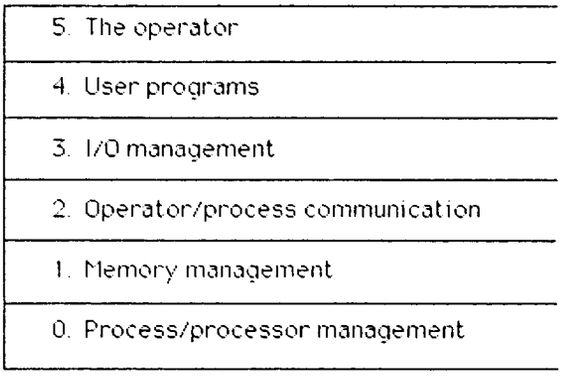


Figure 7: The T.H.E. system layers

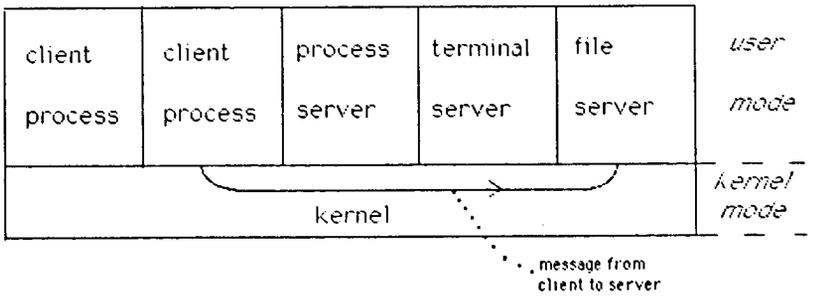


Figure 8: Client-server Model

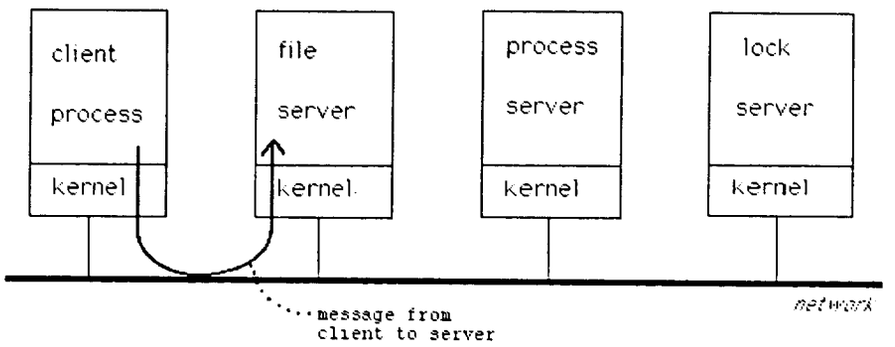


Figure 9: The client-server model extended to a distributed system

THE INPUT/OUTPUT SYSTEM

The aim of the I/O software is to hide from the user the complexities and peculiarities of devices and controllers, and present a standard program interface which is, as far as possible, device independent. For example, the *read_next_char* function should work on any peripheral capable of input (terminal, disk, magnetic tape, communications channel,...). The I/O software must deal with buffering the data and mapping characters into the blocks or packets or whatever physical format is used by the device itself.

The I/O software is usually organised into four separate layers:

- *interrupt handler*
  This is generally entered directly by the hardware when an interrupt occurs. It's task is to store hardware status and signal the driver process.

- *device driver*
  This is the device-dependent code, which can be very large and complicated if the software has to deal with complex topology and sharing issues (such as shared disk configurations with multiple data paths), or where, as in the case of magnetic tape systems, there are many potential error situations and recovery strategies. The current trend, however, is for much of this complexity to be handled by the controllers. Device drivers account for a very large part of the code of most operating systems.

- *device-independent software*
  Device-independent functions include:

  — a uniform interface to all devices
  This may be provided through an abstract device or I/O *stream* which can be read and written sequentially, and makes a disk look like a communications line or even a channel to another process (e.g. Unix pipe).

  — uniform device naming

  — protection and sharing strategies

- buffer management
- storage allocation on disks
- device allocation (for single-user devices like magnetic tapes
- error reporting

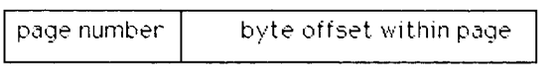
Much of this would normally be implemented as part of the file system.

- *user-level software*
  The user interface to devices is through library routines, often very specifically associated with the language being used. For reasons of efficiency, these library routines often implement some part of the device-independent functionality (such as character buffering for disks). The concept of a logical record is also usually implemented at the user level.

MEMORY MANAGEMENT

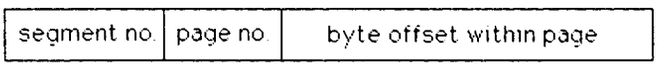
Although the amount of memory available on computers has increased dramatically over the past few years, the number of good ideas for using it seems to have ensured that demand still exceeds supply. A few years ago, many installations ran a time-sharing service for a number of users on a VAX computer with 2 MBytes of memory. Today many users of 6 MByte VAXstations would feel that they cannot reasonably use the new windowing software. Although the issues have changed somewhat, memory management remains an important function of the operating system.

Most computers today use virtual memory – with the notable exceptions of Cray systems at one end of the spectrum and PCs at the other. Virtual memory is usually implemented through a technique called *paging*. Each virtual address referred to by a program is made up of two parts, the page number and the offset within the page.



Before initiating execution of a process, the system stores the address of the start of the page table for the process in a special register. The hardware for address resolution uses the page number to look up the page table. A page table entry contains either the address of the *page frame* (the area of real memory in which the page is currently stored) or a flag indicating that the page is not in real memory (see Figure 10). In the former case the hardware generates the real address and continues execution of the instructi If the page is not in memory, a *page fault* is generated which signals the operating system to suspend the process while it obtains a page frame and reads in the page from external memory. Each process may have its own page table, but memory may be shared by arranging that the same page frame address appears in more than one page table.

A more powerful two-dimensional address space is usually provided by the hardware, using the technique known as *segmentation*. The virtual address now consists of a *segment number*, a page number and an offset within the page.



A segment table entry contains the address of a page table, its current length, and some protection flags (see Figure 11). This makes memory management much more convenient in two ways.

1. Segments can be allocated for different functions (like code, data, stack, heap), each of which can grow and shrink independently.

2. Segments can be shared between processes with adequate protection and without the need to copy page table entries each time a new process is allocated to the cpu.

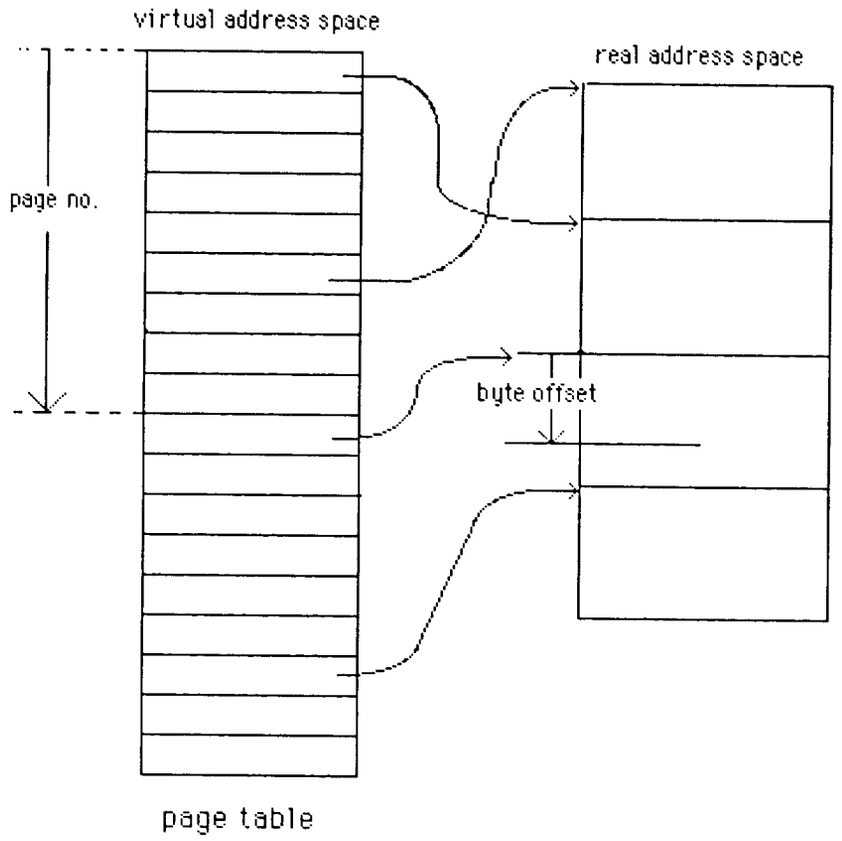


Figure 10: Page Table

Associated with each page are two flags maintained by the hardware: the *referenced bit* which is set whenever there is a read or write access to the page, and the *modified bit* which is set whenever the page is written to. The operating system uses these bits to implement a strategy for reclaiming real memory when required for new pages. The most common strategy is an approach to a *least recently used* algorithm. At regular intervals the system scans all page tables, copies the referenced bit and clears it. A page frame will be a good candidate for replacement if it has not been referenced during the period of the past few scans.

Working Sets

It has been observed that most processes exhibit a feature called *locality of reference*, which means that, during each phase of execution, they tend to reference only a subset of the pages. This subset, known as the *working set*, changes only slowly with time. The *size* of the working set changes even more slowly. The issue is how to establish the size of the working set and what to do if the sum of the working sets of executing processes exceeds the available real memory. Most systems will not permit a process to execute if it does not have its working set in memory. Processes are moved out of memory to secondary memory or disk (*swapped*) in order to ensure that there is enough real memory for the remaining processes to execute efficiently.

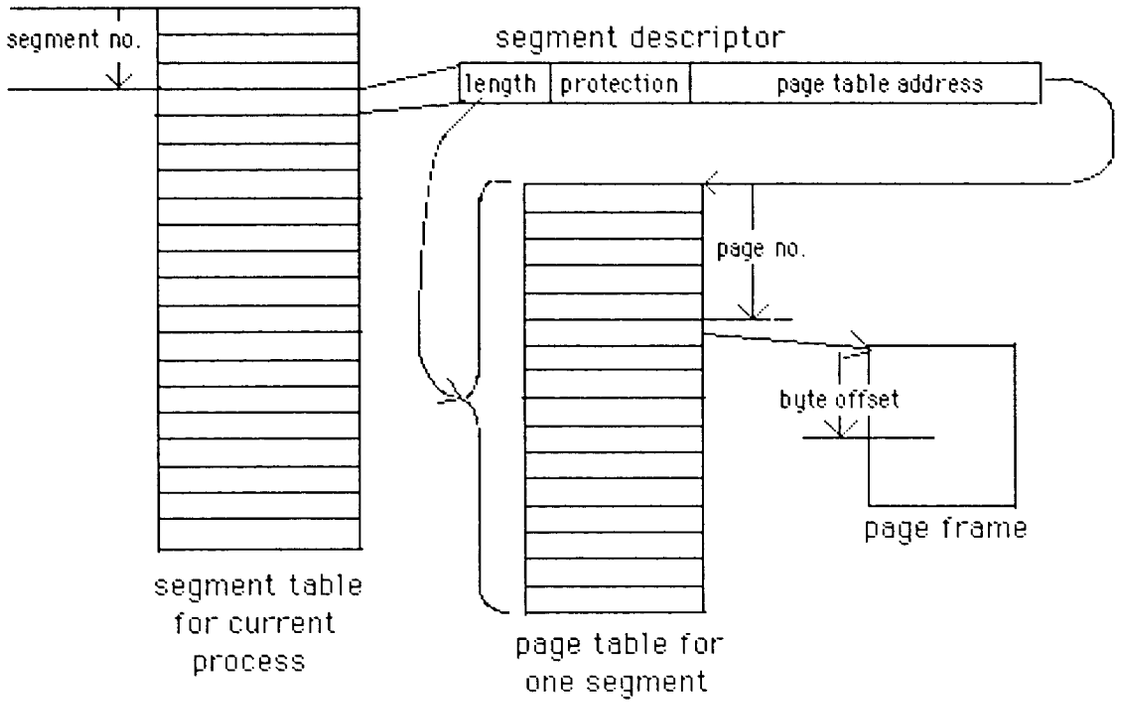


Figure 11: Segmented memory

Global versus Local Page Replacement

A *global* page replacement policy chooses the "best" candidate page frame in the whole system when it needs more space. A *local* policy prefers to take a page frame from the process which needs the new page. The former caters automatically for changing working set size, but allows individual processes to swamp memory. The local algorithm assumes that some other mechanism is used to estimate the correct working set size (such as observing the page fault frequency).

FILE SYSTEM

An important function of the operating system is to organise the magnetic disk storage into logical collections of data called *files*. The *file system* is responsible for mapping these logical files on to the real hardware, for providing a naming scheme and a protection system to enable the users to refer to the files and share them in a controlled manner, and for organising access to the data. It must also take care of efficiency of disk space utilisation and, most importantly, ensure the resilience of the file system to hardware errors. This section will briefly describe one implementation of a file system, that of the Unix operating system.

The user views the Unix file system as a tree of directories. Files may be named by specifying the *absolute path name* which starts with the "root" directory (called "/"), or the *relative path name* which assumes the current *working directory*. For example, in Figure 12, if the working directory is */usr/tom/progs*, the file called *main.f* may be referred to as */usr/tom/progs/primes/main.f* or as simply *primes/main.f*.

A Unix file is described by a record held in a special disk block called an *i-node*. This contains information about the file like its size, owner, creation date, time of last access, and list of the addresses of the disk blocks which comprise the file itself. The first ten data block addresses are stored in the i-node, but if the file is larger then a second i-node is used to contain the addresses of the next 256 data blocks, its address being stored in the *single indirection* pointer field of the first i-node. Double and triple indirection are supported for even larger files, as shown in Figure 13.

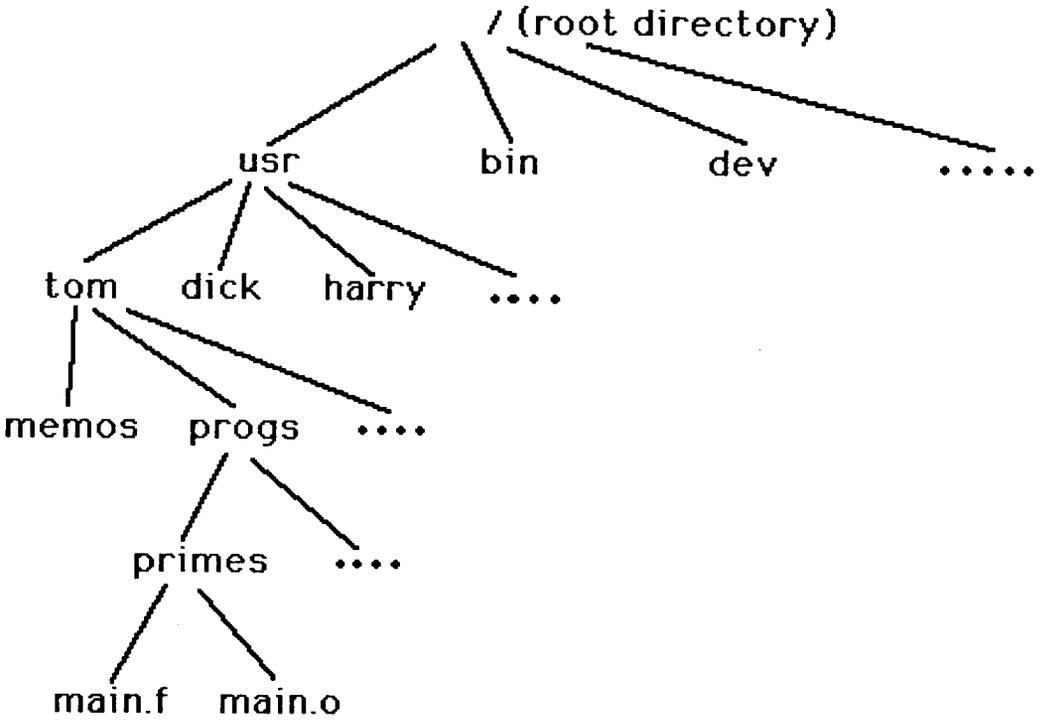


Figure 12: Users' View of the Unix File System

A *directory* is simply a file which contains a list of file names and associated i-node addresses. The "root" directory is not named in any other directory, but can be located because it has a well known i-node address.

In Unix the term *file-system* is used with a very specific meaning to refer to a self-contained set of i-nodes, directories and data blocks forming a complete section of the file tree. A file-system is stored in a pre-defined section of a disk. The format of a file-system is shown in Figure 14. The "boot block" enables the system itself to be loaded from the disk. The "super block" contains control information about the file-system, like its size, the number of i-nodes, the first part of the free store chain, a cache of free i-node addresses, etc. It is held in memory while the file system is mounted, and written back to disk at regular intervals. The i-node area is pre-formatted when a file system is created.

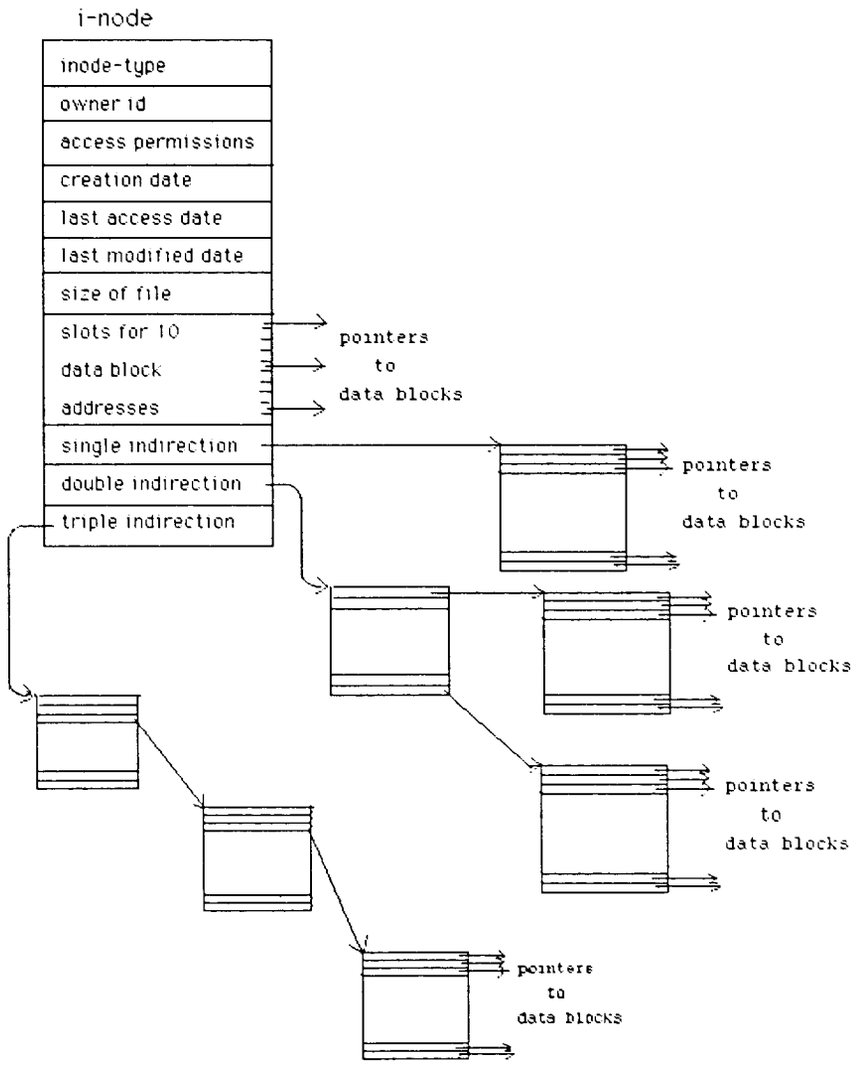


Figure 13: The i-node structure of the Unix file system



Figure 14: The internal layout of a Unix file-system

DISTRIBUTED OPERATING SYSTEMS

Scope

In the first part we discussed operating systems for *multi-processor* computers, where the processors *share memory*. In such tightly-coupled systems a process, be it a user process or a kernel process, is not aware of the particular processor on which it is currently executing, and indeed will run on different processors during successive periods of execution. The operating system caters for the concurrent execution of kernel and user processes. The multi-processor system, its memory, disks and other peripherals are seen as a single resource – the fact that there may be more than one processor is *transparent* to the user.

The key difference between an *integrated* multi-processor system and a *distributed* system is that in the latter case the memory is not shared. We are concerned here with distributed systems in which the computers are interconnected by a network of greater or lesser bandwidth which will allow any computer to communicate with any other.

There are of course architectures which involve multiple processors which do not share memory but use special techniques to interconnect processors in a non-general way (such as a network of *Transputers*). These pose specific problems, and offer special opportunities, to operating system designers, but we shall not consider them further here.

Motivation

The growing interest in distributed systems which we have seen over the past decade has largely been the result of two related factors.

- The availability of general-purpose micro-processors which are both cheap and powerful when compared with contemporary mini-computers and mainframes;

- The user-interface technology for single-user workstations[2], which had been developed during the 1970s principally at Xerox's Palo Alto Research Center, became widely available on standard products from Apollo, SUN and others.

Other factors which make distributed systems appear more attractive than central "time-sharing" services are:

- Guaranteed response – The end-user can pay for his own processor and then count on it being available whenever he needs it. The response time may be slower than on the central IBM at 3.00 a.m. on a Sunday morning, but it will be unaffected by the 600 users logged-on to the IBM at 15.00 the following Friday afternoon.

- Customisation – The individual can select a system to suit his application and budget – cpu, memory, local disks, graphical assist hardware, etc. – he no longer needs to make do with the choice of the computer centre management.

- Evolutionary Growth – A distributed service can expand smoothly and flexibly in response to the wide variety of needs of its users – cpu capacity, specialised text processing facilities, high-performance graphics, etc. – without posing the financial and logistic problems inherent in the replacement of heavily loaded mainframes.

---

[2] Software to support high-resolution bit-mapped screens, mouse, windows, pop-up menus, icons, etc.

Of course, in practice things are not so simple: a distributed *service* relies on general files servers and communications facilities which can become overloaded at peak times; the central batch and time-sharing service runs 24 hours a day, 52 weeks a year, confusing the cost comparison with a private cpu which is often idle between 6.00 p.m. and 8.00 a.m. Nevertheless, it is clear that the potential of workstations and the impressive rate of technological development in this area will make distributed systems much more important during the coming decade.

Requirements

The most important user requirement in a distributed operating system is ***transparency***. The user should not be concerned with the location of the different resources (the computers, disks, plotters, ..) making up the system. Ideally, the user or an application program should be able to use any of the computers on the network and see ***exactly the same environment***. We must contrast this with a *network operating system* which makes available common resources (printers, files, ..) to a collection of computers attached to the network, but where the users and applications are aware of exactly which machines they are executing on, and where they must make ***explicit*** requests to use the common facilities. VMS with DECnet is such a network operating system.

We may define *transparency* as the *concealment of separation from the user and application programmer, so that the system is perceived as a whole rather than as a collection of independent components.*

Important aspects of transparency are:

access transparency: which enables local and remote files and other objects to be accessed using the same operations;

location transparency: which enables objects to be used without knowledge of their location;

migration transparency: which enables objects (files, processes, ..) to be moved without affecting the operation of the user or application programs.

Another important requirement is that there should not be any loss of functionality over that obtained from a central integrated system. This implies a number of things, such as the way in which failures can be monitored and corrected, and the way in which the system can be managed (new users registered, new software installed, additional computers configured, network topology evolved) without unduly affecting or involving the end-user. The system must appear as a single environment on which general services (file backup and archiving, for example) can be made available as conveniently as on a central system.

The distributed operating system must be resilient to the failure of individual computers and sections of the network. It must also be possible for the system to evolve in size and topology smoothly, without affecting the applications.

In the practical world, the support of heterogeneous systems is an important issue, since some of the major advantages of using distributed systems will be lost if the system is tied to a specific manufacturer's processor.

Architectural Models

The most common model for building a distributed system is the *workstation* model, in which each user has a personal workstation on which he does most of his work. The workstations form a network together with some *server* computers which may provide specialised facilities such as more powerful processing or support of unusual or expensive peripherals (see Figure 15). The system supports a single global file system, so that all of the data can be accessed from any of the workstations.

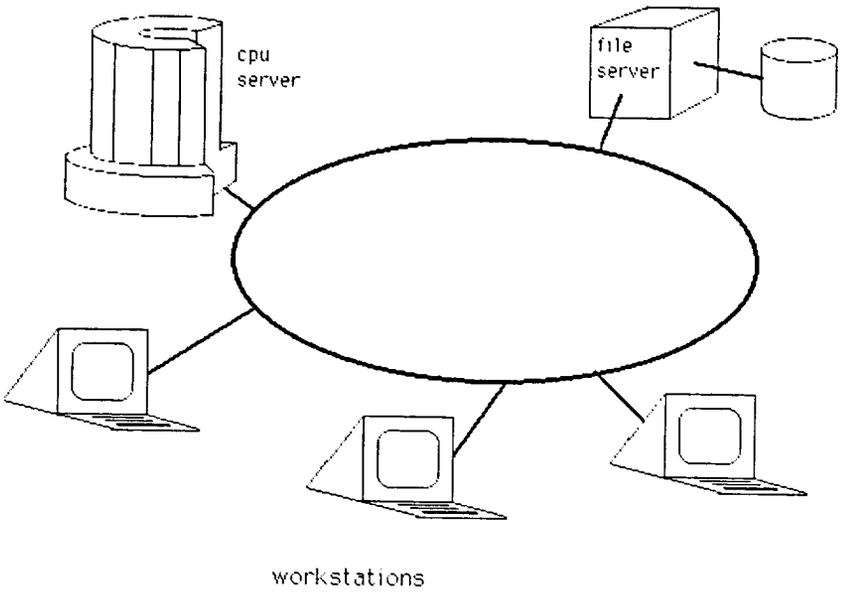


Figure 15: The workstation model

A more ambitious model, the *integrated* model, blurs the distinction between workstation and server, making all of the computers in the network appear as a single computing system (see Figure 16). The operating system decides where to execute a process and where to store a file. The desktop cpu will be used to support the interface to the operator sitting in front of it, but the processes he creates will execute anywhere in the system, and may even migrate from computer to computer according to system load and other factors.

Some distributed operating systems have been implemented by extending the functionality of a conventional operating system kernel. The kernel provides process management, the I/O system, memory management and a file system, but is extended to support these facilities across the network of computers. All of the communication between nodes is handled by the kernel itself (see Figure 17). This *closed architecture* results in a rather large and complex kernel which makes maintenance and development difficult. An alternative architecture adopts the *client-server model* which we looked at earlier. In this architecture the kernel is "stripped down", providing only minimal process support and hardware interfacing routines, together with a *message-passing* facility to enable processes to communicate with each other. Everything else is implemented in user-level server processes (see Figure 18). Inter-node communication and the management of the distributed environment is handled by a network server process which executes on each computer in the system. There are a number of advantages of this architecture. The kernel is relatively simple, and the complicated system functions are separated from each other in protected user-level processes, leading to easier development, testing and maintenance. The system is configurable – each node only runs the servers which it requires. For example, the disk block server may be present only on nodes which have disks connected to them. The system can be extended easily, simply by writing a new server. There could, for example, be more than one file system. In this sense we may call this an *open architecture*. With this type of architecture important issues are the efficiency of the message passing facility and context-switching between user-level processes.

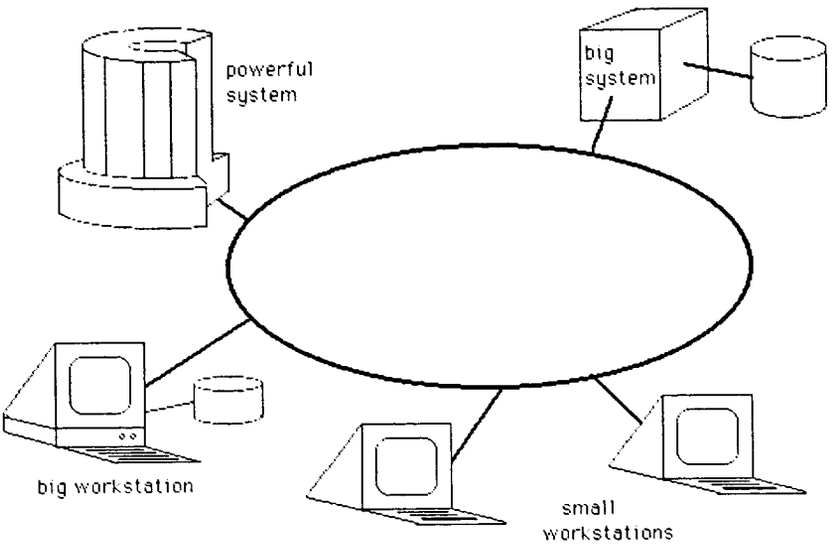


Figure 16: The integrated model

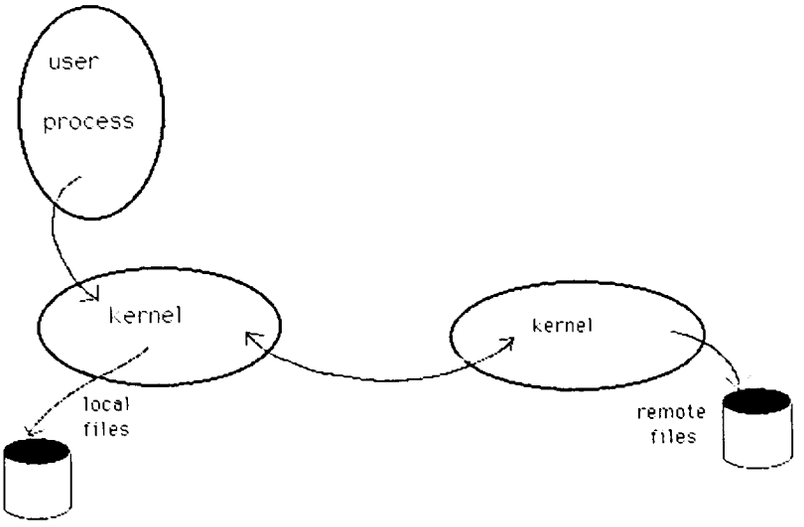


Figure 17: Closed architecture

Some Transparency Issues

- *File Naming*
  There are two approaches to the problem of providing a single naming scheme for all the files in a distributed system. The first is simply to extend the path name to include the location of the file: e.g. *//node/bin/prog*. In effect, the operating system maintains a "super root" directory consisting of the node names in the system, and uses it to start searches in the root directory of the node specified. This effectively *unites* the file systems of the different nodes. It has the advantage

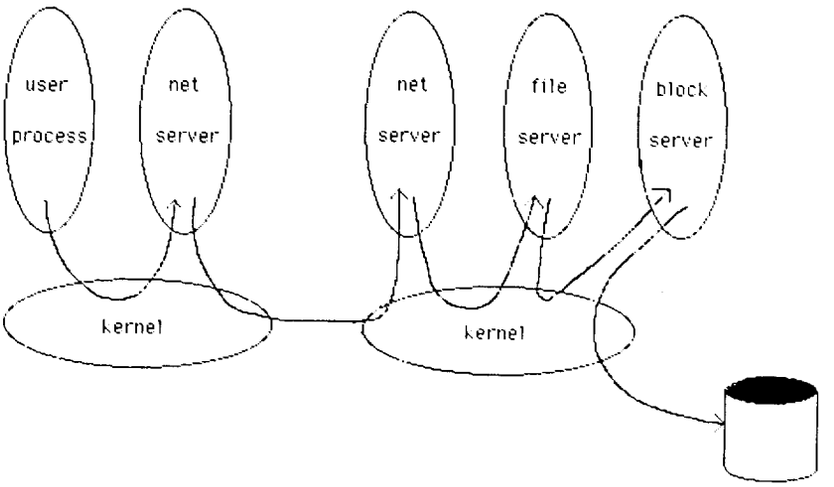


Figure 18: Open architecture

of being clear and simple, but has a major disadvantage in that the user must decide where to store files, and keep track of their location.

The second approach is to have a single global file system which looks exactly the same from all machines – there is only one *root* directory, one *bin* directory, and so on. The operating system decides where to place the files, and hides their actual location from the user. For performance or security reasons it may decide to replicate a file, but this is quite transparent to the user.

- *Protection*
  In a distributed operating system it is clearly important that a user does not have to identify himself explicitly with his user id and password to each of the computers which he uses. The authentication of the user should be performed once only, and his authority and rights supplied automatically for checking whenever he uses a resource on any of the computers in the network. In the case of an integrated system, we assume that the protected kernel code can safely maintain the information necessary to control access throughout a working session. However, even if we assume that the kernel of the operating system executing in each of the nodes of a distributed system can be trusted (and remember that the user has access to the hardware switches of his desktop workstation, and so can, if he is clever enough, load any operating system he likes), the network inter-connecting the nodes is unlikely to be secure, and even relatively simple and easily programmable devices like PCs can monitor validation protocols and emulate other workstations. There are no straightforward solutions to this problem.

EXAMPLES OF DISTRIBUTED OPERATING SYSTEMS

LOCUS

The LOCUS distributed operating system was developed at UCLA by Popek and others in the late 1970s. It looks like Unix in that its kernel suports all of the standard Unix system calls, and it can support almost all of the higher-level software and application programs which come with Unix. A copy of the kernel executes on each computer in the system, providing:

- system-wide file naming;

- a distributed file storage and access system;
- file replication;
- remote processes;
- process migration;
- heterogeneous processor support.

There is a high degree of transparency concerning files and processes. All of the computers in the system can act as both clients and servers. The user sees a single giant Unix system with the Unix file access, process creation (*fork*) and inter-process communication (*pipes, signals*) primitives implemented transparently across the network. When a new process is created, the system will select an appropriate node for it to run on. The system was initially developed for mini-computers (VAXes) using a model in which users connected through terminals to one of the computers and thereafter had transparent access to all of the resources of the pool of computers. It adapts well to workstations: the processes which require intimate interaction with the user, maintaining the details of the graphical user interface (windows, mouse, pop-up menus, etc.) naturally execute on the workstation on the user's desk, whereas all of the other processes he creates and files he uses may take advantage of the full network of workstations and mainframes.

One of the reasons why LOCUS is of particular interest as an example of a distributed system is that it has been used by IBM as the basis for the AIX/370 and AIX/PS2-TCF products. The LOCUS system has a closed architecture – it is implemented entirely within the kernel.

File Support

Three different locations are defined for file operations –

- *using site*
  The client computer which executes the process which issues the open, read, write requests.
- *storage site*
  The computer which stores the file on one of its disks.
- *current synchronisation site*
  Every file belongs to a *filegroup*, and for each filegroup there is a single computer which acts as the current synchronisation site, keeping track of the location of all files in its group, synchronising concurrent access to the files, and selecting storage sites for new and replicated files.

When a client program opens a file it supplies the path name. The LOCUS kernel at the using site searches the system directory structure and, if the file exists, obtains a file descriptor for it which consists of the filegroup and logical i-node number within the filegroup. The using site now requests the current synchronisation site for that filegroup to open the file, and supply the name of the storage site and the *file index*. The file index is equivalent to the i-node structure for a Unix file, containing control information like owner, permissions, etc. and the disk address and block numbers of the data which comprise the file. The current synchronisation site obtains the file index from the storage site and returns it to the using site. The open operation is described graphically in Figure 19.

Having completed the open, subsequent file access operations involve only the using site and the storage site. When the file is closed the current synchronisation site is informed by the storage site.

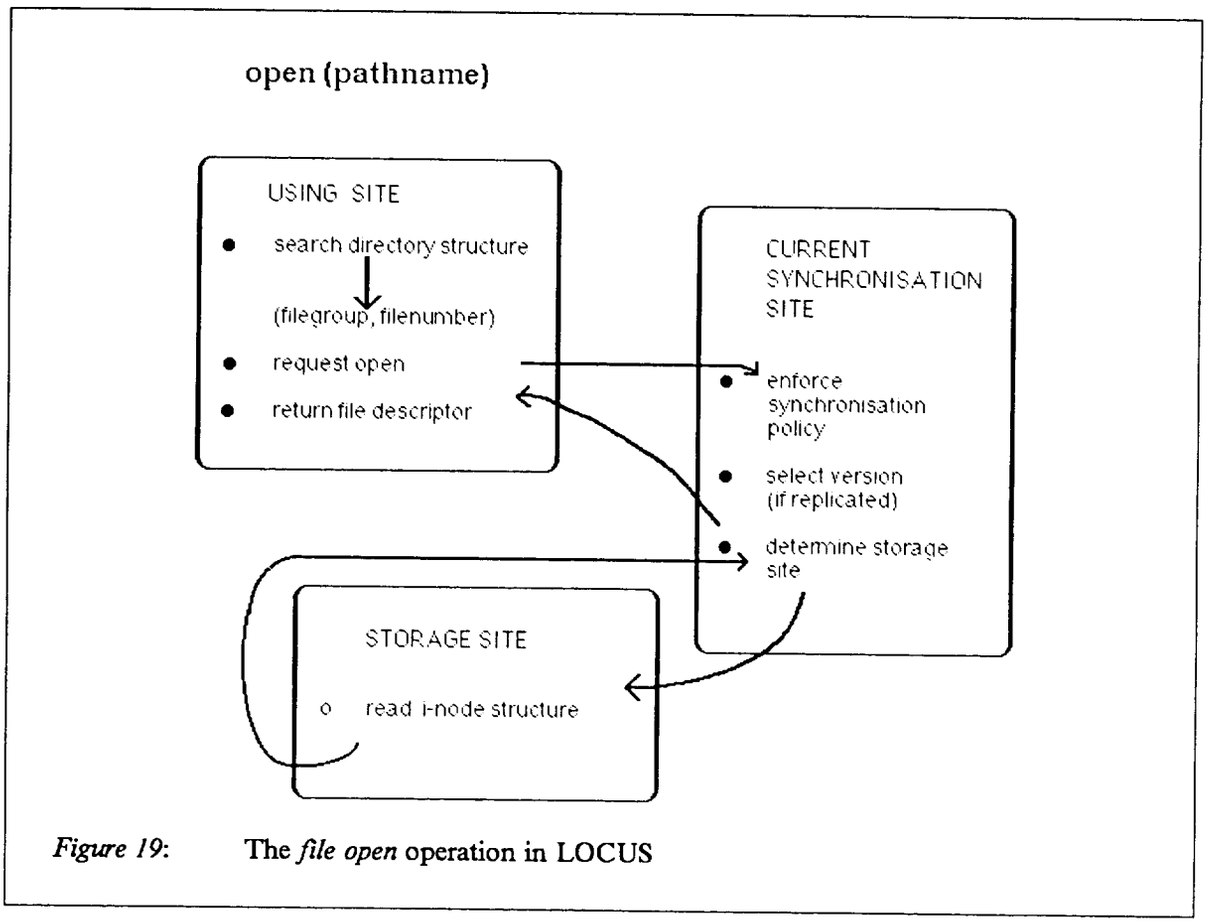


Figure 19: The *file open* operation in LOCUS

The directory search may involve opening a number of different directories as the path name is processed. As in Unix, directories are ordinary files, and so the same mechanism is used to open and read directories as described above. Caching of file indexes and buffering of data are used to improve the performance of the system.

The important point is that there is a *single distributed file system, and the storage location of a file is not implied by the path name.* The system provides full ***access*** and ***location transparency***. It supports replication of files (copies of a file may be held on different nodes, for security or performance reasons) but this is hidden from the user by the current synchronisation site. The file system looks exactly the same from all the computers in the system.

Distributed Process Support

The main functions which are concerned with the support of distributed processes are as follows.

- Process creation – in Unix this is the *fork* system call.
- Changing the executable program and data image of a process – the *exec* system call.
- Moving a process from one computer to another – *migration.*
- Inter-process communication – Unix *pipes* and *signals.*
- Process status tracking – for example, informing the parent process when a child process terminates.

- Execution site selection.

LOCUS uses process names which consist of the identity of the site on which the process originated, concatenated with a unique process number generated by that site. This provides a name which is guaranteed to be unique across the system, while not compromising transparency: since a process can migrate to another site, the name does not imply anything about where it is currently executing.

The selection of a site for execution of a process is influenced by several factors.

- When a process is created (*fork*) or its image changed (*exec*) a list of possible execution sites, which is contained in the process environment, is used to search for an acceptable location. The list may be modified by the user to influence (or force) site selection prior to issuing the *fork* or *exec* call.

- The load module must be compatible with the execution site. The *fork* system call creates a new process and copies the environment, data and executable image from the calling process. It can therefore only create a remote process on hardware which is compatible with that of the calling site. The *exec* system call replaces the executable image of a process and so the process may be moved to a site of a different hardware type if a suitable load module exists. LOCUS allows more than one type of load module to exist for a given program. This is handled by a technique in which the file with the apparent path name of the load module is in fact a *hidden directory* which points to a set of load modules prepared for different computer types.

- The user, or system, can issue a special command to cause a process to migrate to a specific remote location.

- Each process has associated with it a list of sites which it is *allowed* to use. This is set by the system when the user logs in, and cannot be changed. It restricts the choice of sites which the user may specify.

SUN Network File System (NFS)

The SUN Network File System (NFS) provides a distributed file service based on conventional Unix file systems. It is of interest because it is rather widely available, being supported on many Unix systems and also on several non-Unix systems such as IBM's VM and Digital's VMS.

The method used in Unix systems is to extend the *mount* system call to enable a directory on a remote computer to be mounted as part of the file name space on the local machine. In the example of Figure 20 the directory on the node *cray* with pathname */u/les* is mounted on node *sun* as a directory with pathname */user/les/cray*. The normal Unix input/output primitives may then be used from *sun* to open and access transparently files on *cray*.

In a Unix system, the *open* system call returns a unique *file descriptor* which may be used by the application in subsequent references to the file. Within the kernel this file descriptor is mapped on to a file system identifier and an *i-node* number within that file system. NFS extends this to include the computer identifier in the *internal* file identifier, or *v-node number*. When the file is accessed the system therefore knows if the file is local, and can be read and written using a local disk driver, or whether it must be accessed through a remote server. Because the *v-node* is not visible outside the kernel (the application cannot distinguish between a local and remote file descriptor), NFS provides a degree of *location transparency*. However, it is clear that the pathname actually specifies the physical location of the file. More important for practical purposes, the view of the distributed file system depends on the node being used by the observer. A user cannot move from one workstation to another without seeing the topology of the file system change. NFS supports only one level of remote indirection: it is not possible, for example, to mount the directory */user/les* from the node *sun* in Figure 20 on a third

computer and thereby give access to the nfs directory */user/les/cray*. Thus, setting the current directory to the user's home directory on a remote workstation provides a more limited view of the distributed file system than that seen by a process executing on the computer which stores the file system containing the home directory.

NFS is implemented as remote procedure calls between kernel processes. The server maintains no state information about the client's file usage between requests. This avoids the need for the server to cater for crashes of the client computers and network problems, but it implies that there is no control over concurrent use of files by different clients.

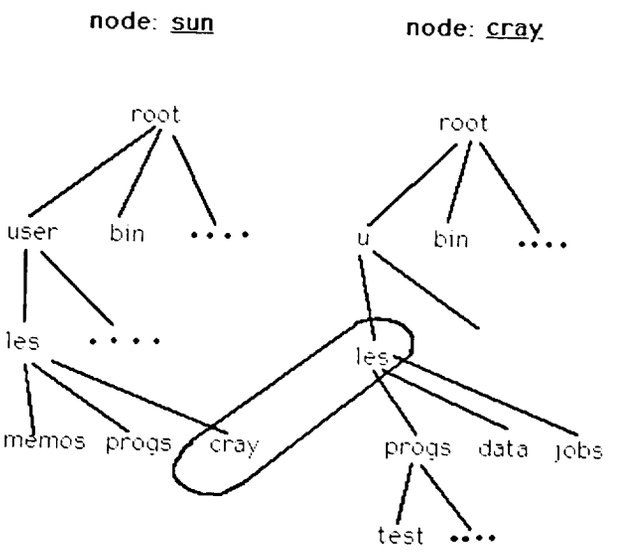


Figure 20: SUN Network File System

APOLLO DOMAIN

The Apollo Domain system runs on Apollo workstations connected together on a network, and provides a number of distributed functions including:

- system-wide file naming;
- transparent data access;
- system-wide locking;
- remote message and mailbox systems for inter-process communication.

Like LOCUS, each computer in the system runs a kernel which implements the file directory and access services. The application programs only interact with the local kernel. The interface between client and server nodes is handled entirely by the kernels. Open files are mapped into the virtual address space of client processes, and so the data *access* system is in fact based on a *distributed paging service*.

An *object storage system* is implemented as a general data storage mechanism.[3] Objects are assigned a system-wide unique identifier when they are created, consisting of the name of the creating computer concatenated with the time, but this does not imply the storage location: *location transparency* is maintained. An object is made up of a sequence of pages which are stored on a disk of the object's current *home* computer (not necessarily the creating computer).

The algorithm for locating an object is first to try the computer on which it was created, and then the computer which holds the directory it is named in. If this fails, a more general search will be carried out, and a *hints* file maintained to assist in future searches.

A directory system is provided to translate textual names into object identifiers. This implements a hierarchical structure which looks like a Unix file system, but with a pseudo top-level *super-root* directory containing the computer node names. Thus, a name like:

//sweetie/user/les/test

refers to a directory structure held on the node called *sweetie*. The file called *test* is not necessarily stored on *sweetie*, but in practice it usually is.

Domain provides full *access transparency*. It also provides *location transparency* in the sense that an ***object*** can be accessed without knowledge of its location. It does not, however, provide full transparency for file (object) names – or rather directories – in the sense that the user's view depends on his position in the network. Consider the example in Figure 21. A user executing on *sweetie* has his home directory set to *//sweetie/user/les*, and his search path includes */bin*, which refers to the directory of that name on *sweetie*. When the same user is executing on the node *fred*, his home directory is set as before, to *//sweetie/user/les*. However, if he uses an *absolute pathname* such as */bin*, this will refer to a file on the node on which he is executing (*fred*). This may be what is required, and indeed if *sweetie* and *fred* use incompatible hardware this will be just fine in the case of */bin*. In general, however, the user will need to be aware of the node on which he is executing, as his view of the file system is dependent on it. This illustrates the difference in *transparency* between Domain and LOCUS.

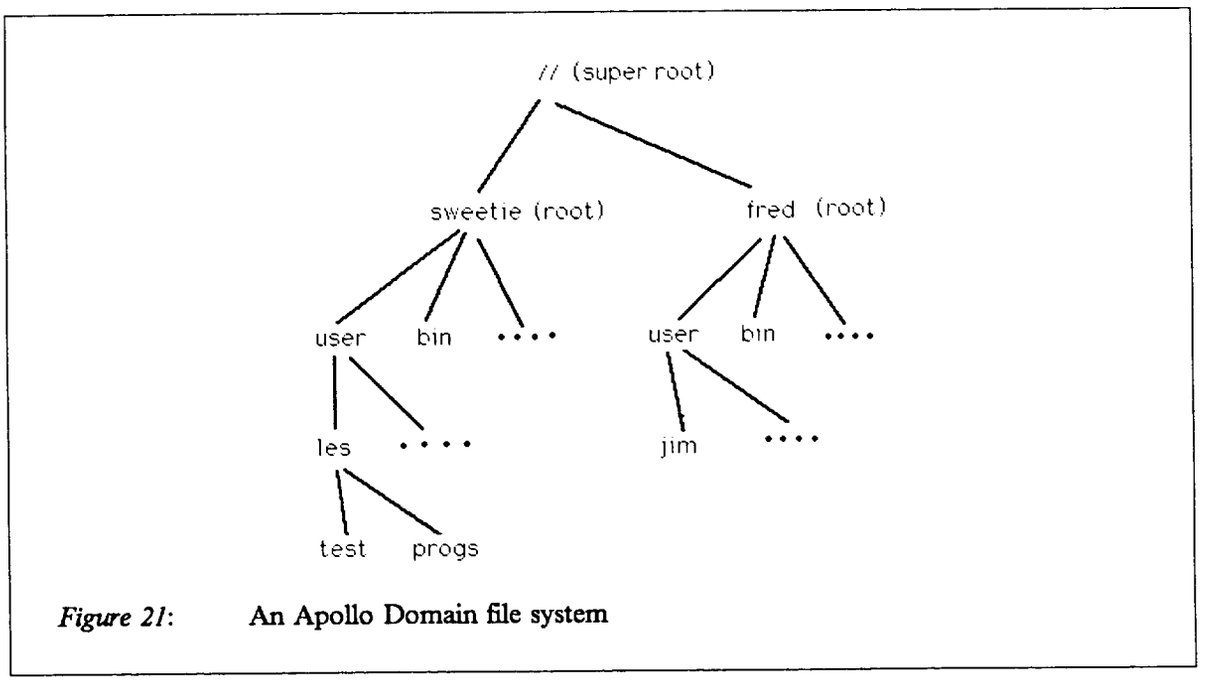


Figure 21: An Apollo Domain file system

[3] A *file* is an object with associated operators such as *read* and *write*.

MACH

The MACH distributed system kernel was developed at Carnegie-Mellon University, a successor to the earlier *Accent* kernel. MACH supports the client-server model, providing a light-weight kernel which executes on each computer in the system, and a set of inter-process communication facilities to enable system services like the file system, network support, process management and memory paging to be performed by servers running at the application program level. It has a number of features which are designed to provide efficient support of multi-processor systems and large virtual memories. The open architecture allows the support of, for example, multiple file systems and paging servers, a feature which is of considerable interest in a computer science research environment. MACH provides Unix emulation, in order to be able to benefit from the wealth of Unix software which is available. At present this is partly implemented within the kernel itself. Apart from its use at Carnegie-Mellon University, where it has been ported to a number of different computer architectures, MACH is used by several commercial companies, including *Next*, as the standard operating system.

Process Management

The idea of a *process* in MACH is split into two entities: the *task* and the *thread*. The *task* is an address space which owns resources and has *capabilities* to access other resources. The *thread* is the basic unit of cpu allocation. One or more threads run within a task, sharing the memory and other resources of the task. These light-weight threads enable efficient parallel processing to be achieved within a task on mutli-processor computers, since the costs of context switching, protection checking and thread (process) creation may be minimised. Figure 22 illustrates the task and thread structure of MACH, a Unix-style process being implemented as a task with a single thread.

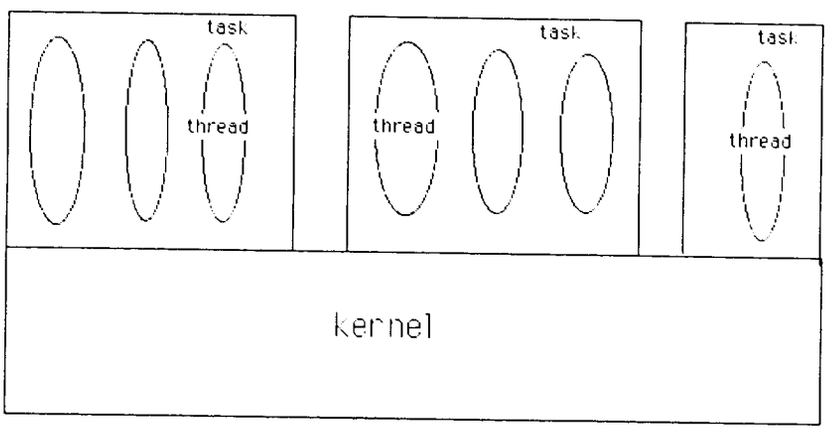


Figure 22: MACH *tasks* and *threads*

There are several mechanisms for communication between tasks and threads.

- *port*
  A *port* is a queue for messages between tasks and threads. A port is associated with an object (such as a task or a service) and use of a port is controlled through capabilities. In order that two tasks can communicate, they must each hold a capability for the same port.

- *message*
  A *message* is transmitted between tasks using *send* and *receive* operations on the same port. Messages may be of any size. A third port primitive is provided, *rpc_message*, which facilitates the implementation of remote procedure call systems using MACH.

- *fine-grain communication*
  This provides a very light-weight communication mechanism between threads within the same task, using the task's memory. There is minimal kernel involvement, and no protection. This may be used, for example, by a Fortran compiler to implement *micro-tasking* for a multi-processor shared-memory computer.

- *intra-application communication*
  This is intended for use by threads within the same task or in related tasks which share memory. It implements the standard port message primitives, but takes advantage of the shared memory to minimise message copying and kernel intervention.

- *inter-application communication*
  In this case the port message primitives are handled entirely by the kernel, giving full protection and supporting communication between tasks on different computers.

Memory Management

The address space of a task consists of a collection of *regions*. A region may be passed on to new tasks when they are created, and so separate tasks with a common ancestor may share memory.

In order to make the copying of virtual memory efficient a feature called *copy-on-write* is used. A copy-on-write page is shared between two tasks until one of them writes into it; only then is a new page allocated and placed in the address space of the writer in place of the shared page. This is used to avoid copying large messages between tasks, and to provide a new task with a copy of its parent's memory (not as a shared region, only in order to provide it with a copy of the parent's environment).

MACH supports paging servers implemented as tasks outside the kernel. When a memory region is allocated, an *external pager* is defined which will receive requests from the kernel to read and write pages to external storage.

Network Communication

Communication over networks is provided by *network server* tasks, one of which is present on each node of the system. The network server receives all messages on ports which are not present on the local system. It translates the local port number into a *network port identifier* and then sends on the message to the network server in the node which handles that network port. The remote network server maps the network port on to a local port number and forwards the message to the destination task. Only the network server is aware that a port is local or remote; other clients and servers see no difference between them. This is illustrated in Figure 23.

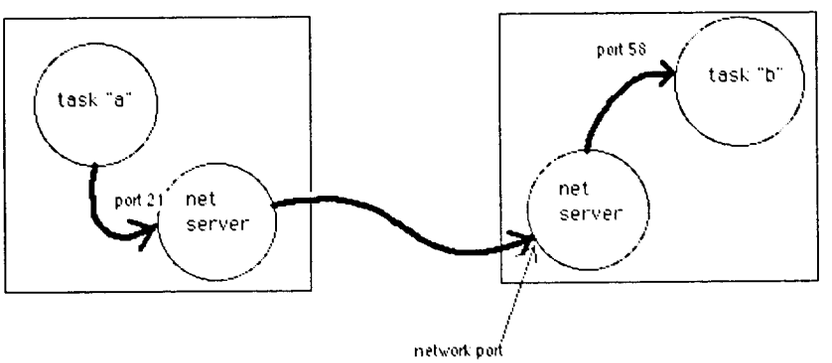


Figure 23: Passing messages across a network in MACH

Bibliography

Basics

- *Operating Systems – Design and Implementation,*
  Andrew S. Tanenbaum, Prentice-Hall, 1987

- *An Introduction to Operating Systems,*
  Harvey M. Deitel, Addison-Wesley, 1984

Unix

- *The Design of the Unix Operating System,*
  Maurice J. Bach, Prentice-Hall, 1986

Distributed Systems

- *Distributed Systems – Concepts and Design,*
  George F. Coulouris and Jean Dollimore, Addison-Wesley, 1988

- *Distributed Operating Systems,*
  Andrew S. Tanenbaum and Robbert van Renesse, Computing Surveys, Vol 17, No 4, December 1985

- *The LOCUS Distributed System Architecture,*
  Gerald J. Popek and Bruce J. Walker, The MIT Press, 1985

- *Mach: a new kernel foundation for Unix development,*
  R. Rashid et al, Proceedings of USENIX 1986 Summer Conference

- *Sun Network File System Reference Manual,*
  Sun Microsystems, Mountain View, California, 1987

- *The architecture of an integrated local network* (Apollo Domain),
  P.J.Leach, P.H.Levine, B.P.Douros, J.A.Hamilton, D.L.Nelson and B.L.Stumpf, IEEE Journal on Selected Areas in Communications, vol. SAC-1, no.5, 1983