Evaluation Report On Modern Object-Oriented Operating Systems

T. Douglas Steel.

Queen Mary & Westfield College,

University of London.

doug@cs.qmw.ac.uk

ABSTRACT

Traditional operating systems are beginning to come up against the costs of maintaining large systems. Object-oriented techniques can be used to help alleviate this problem. The object oriented model also maps neatly onto distributed systems issues - several entities communicating by message passing. Using such techniques is not without its own problems, and traditional such systems may have difficulty in areas such as resource control and security. Modern operating systems implement new process models which can be used to help in the design of distributed object-oriented systems will be studied.

1. Introduction

Unix was originally devised as a research operating system [Ritchie 74] and as such was given free with source code to universities and colleges. The feedback generated from these institutions was useful in developing the system to the sophistication required for commercial exploitation. Many of the features and utilities developed by the education & research community became "standard", and integral parts of Unix. Some of these features were added without regards to the internal elegance of structure of the unix kernel. This eventually led to several divergent and incompatible "flavours" of Unix being developed (e.g. BSD, System V, Version 7, SunOS, Xenix). Today various standards committees (e.g. OSF, UI, X/OPEN, POSIX) are each trying to tie together all these varying strands into their own*"Standard"* [Salus 90]. Rather than taking the commonalities of all the variations and producing a *minimal* standard these committees are aggregating all the features provided by all the versions into one humungeous, all encompassing, *maximal* standard [Swartz 90]. This "creeping featurism" has resulted in one almighty case of "kernel bloat". This has not been fatal (yet), but has led to some brave attempts to rationalize the functionality provided by the Unix kernel [Forsyth 90]. Others have taken a completely fresh start and are trying to produce operating systems to replace/supplant Unix, whilst still providing a reasonable level of compatibility.

A related trend in operating systems has been the arrival of "open" systems. An open system should allow a user to make choices of hardware and software independent of the underlying architecture (machine or operating system). In other words the consumers choice should *NOT* be constrained by the hardware manufacturer or the software author. When a site chooses to walk down the open systems road they should be able to freely intermix different hardware, system software, and applications. Most of the operating systems I will examine later have at least some concessions to open systems issues.

The above issues have a major impact on distributed systems - systems which consist of several co-operating, interconnected computers [Coulouris 88]. For example, it is desirable to be able to switch from one terminal/workstation to a better one provided by a different manufacturer without having to give undue consideration to compatibility issues (i.e. it is desirable to be able to "plug & go"). Open distributed systems provide this level of inter-changeability, allowing users to mix and match hardware, moving to improved systems incrementally as funds permit, rather than having to replace the whole system in one "upgrade". Particular attention will be paid to the support that modern operating systems provide for distributed systems - actually this is the primary goal in most of the systems studied here.

2. Philosophies, Models & Concepts

This section describes some of the important models and concepts behind modern operating systems. Many of the previous generation of operating systems (e.g. Unix) tried to provide a large number of functions in the kernel, this has led (as previously mentioned) to the size of these operating system kernels becoming considerable, thus making such operating systems difficult to fully understand and hence maintain or update. From open systems has come the desire to make operating systems flexible and easily expandable, whilst still having some degree of compatibility with the emerging standards. Distributed systems require a certain minimal functionality such as inter-process communication and networking, and this has also driven research into areas such as lightweight processes and object oriented systems. Before examining modern operating systems, it is desirable to define philosophies, models and concepts to be used as a reference.

2.1 Models For Distributed Systems

It is possible to classify distributed systems according to the conceptual model that they adopt [Coulouris 88, Tanenbaum 85]. It is difficult to define precise boundaries between the various models and this should be kept in mind when looking at the various models presented below. Here we will use the location of program execution as the distinguishing criteria.

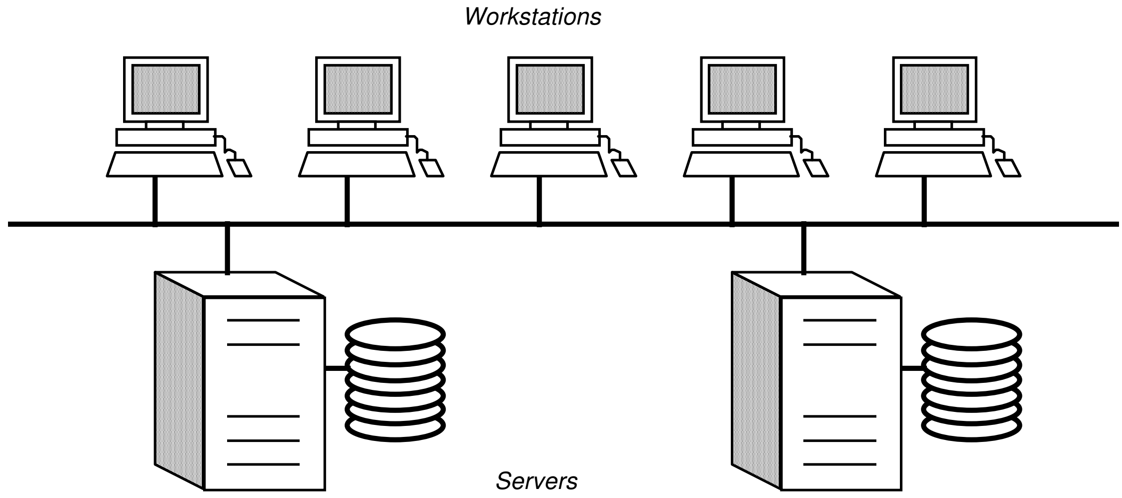

Figure 2.1 - The Workstation/Server Model

One of the simplest and most familiar models is the *workstation/server* model (fig 2.1), each workstation supplies the user with the ability to execute programs locally, while providing access to shared services (e.g. file servers, print servers). No attempt is made to try and provide any form of distributed program execution - the user has sole use of his/her workstation and cannot use more than the processor power available locally for the execution of programs. This model can therefore result in a very poor utilisation of resources (e.g. idle workstations) since only the local processor is available for executing user programs. The flexibility of the model is compromised by not being able to execute programs that require more resources (e.g. memory) than the local workstation can provide, and the only way to provide such resources is to upgrade the workstation. Sharing of data that is not located on one of the shared servers is likewise very difficult.

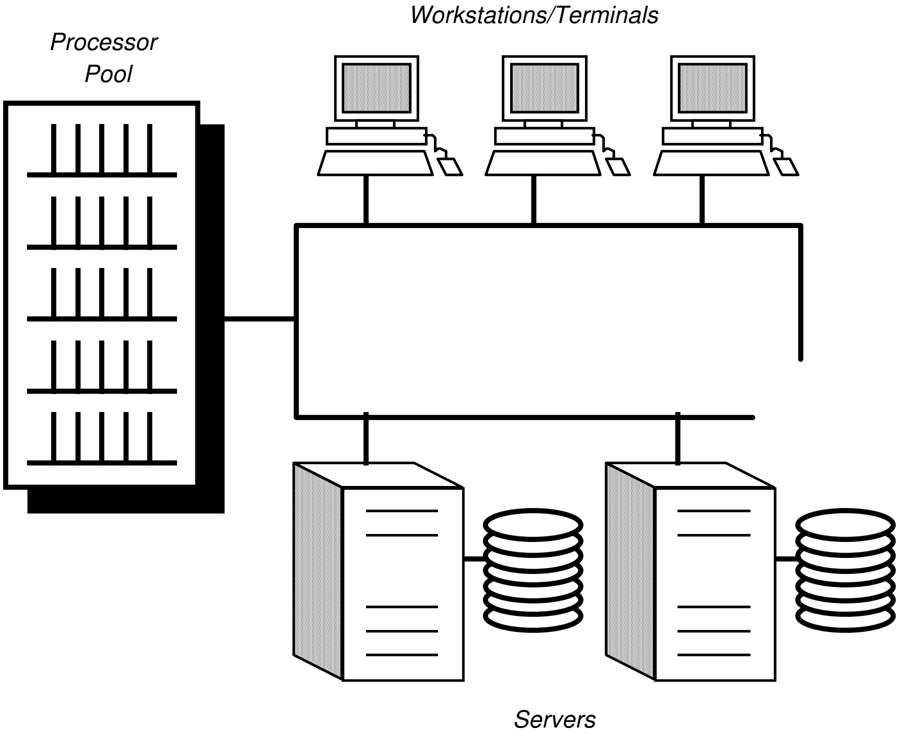

Figure 2.2 - The Processor Pool Model

On the *processor pool* model (fig 2.2), however, a user is able to make use of all the free processing power available in the processor pool if so desired. The users workstation in this case is in effect just an intelligent terminal, and does not run programs, except possibly the windowing system or user interface. This sort of system can be easily made to appear to the user as a standard timesharing system and it is possible to keep all the pool processors in a central location. It is possible, in this model, to make a more effective use of available resources (e.g. if more processor power is required, more pool processors can be allocated) than in the workstation/server model. As more (or better) processors are added to the pool, this benefits all the users of the system since this extra power is available to *all* users.

It is possible to combine the above two models to form a *hybrid* model, in which programs can be run on either the users workstation or on the pool. It is also possible (if process migration is implemented) to use idle workstations as additions to the processor pool.

Some implementations of the *integrated* model go a step further in making the entire system appear as one timesharing system. In this case it is not possible to know where a program will execute, since all processors (pool, workstations, servers) are available for program execution and program location will depend on system load, location of resources, etc. It can be difficult to distinguish this case from the hybrid model, the major difference is that of transparency. For instance, in the hybrid case it may be possible for the user to instruct the system to execute a program on his local workstation, whereas in the integrated model it is left completely up to the system to determine this.

Many of the implemented systems discussed later have been designed with only one of these models in mind, but are flexible enough to be configured for other models as well.

2.2 Services

The goal of operating system design is to provide a set of services to the users. Traditionally these services are provided by the kernel on each machine. This makes the changing of services difficult and every kernel needs to provide sufficient services for the users needs. It is desirable to confine the services that the kernel provides to the barest minimum necessary. Extra services should be provided outside of the kernel in a modular fashion to facilitate ease of change (see section 2.3). If it is possible to communicate with remote services it is not required to provide all services needed on all machines, thus reducing the memory overhead and allowing users to share services.

One service that is vital to current systems is that of the file service[Gifford 88], these may be :

- remote disks systems
- block service access systems
- file level access systems

Most systems store persistent data in a file (normally a stream of bytes), and these can be used as a unit of data sharing. Persistent data includes such objects as executable programs, source code, word processing documents and databases. A system should therefore provide some system of storing these objects, most systems examined below provide a *file server* which responds to requests to store or retrieve these objects. These file services may form part of an integrated I/O model, such as the original Unix view of everything as being a file.

Another vital service is that of the *name* or *directory service*. This provides a permanent store for users supplied names for system objects (see section 2.7).

2.3 Microkernels

As mentioned above, the removal of extraneous functionality from the kernel can result in a more modular system. Deciding what the minimal functionality required to provide a usable system is analogous to the arguments in the RISC/CISC debate. The question to ask is "How little can we get away with ?" At the same time it is desirable to come up with a basic set of abstractions for system resources, e.g. files, streams, objects. Reducing the functionality and set of abstractions used in an operating system should make the resulting system smaller, easier to understand and hence easier to maintain or modify.

Having a small kernel allows machine dependent parts of an operating system to be kept in a relatively small, well defined area. This aids operating system portability. By reducing kernel overheads it is possible to increase efficiency and performance. Additional functionality can be easily provided outside of the kernel increasing flexibility and extendability, i.e. such systems can truly be described as "open". It thus becomes possible to provide the functionality of a traditional operating system by plugging together appropriate modules. It also allows an operating system to be tailored far more accurately to the users needs. By providing mechanisms rather than policies for such features as protection, the kernel allows services to tradeoff performance against protection and vice-versa.

Several systems have been devised that have only the one type of kernel operation, that is *Inter-Process Communication*. All other services are provided by servers that may or may not reside within the kernel. Since IPC in these systems is the only interface with the kernel, it is possible to easily extend the kernel's services by simply adding extra servers. This does not require either the interface to the kernel to be changed or any additional abstractions. Such kernels also tend to provide services such as task switching and memory management. The resulting well defined kernel can be fine tuned to provide increased performance.

2.4 Inter-Process Communications

Traditional operating systems were designed before networking became commonplace and hence provide little support for IPC. A distributed system normally requires co-operation between processes running in different address spaces or processors. Modern operating systems need to be designed with this mode of computing in mind. Some form of communication is required to provide for this co-operation. *Remote-Procedure Calls* and/or *Pipes* [Gifford 86] can be used to provide transparent communication between co-operating processes. RPCs look, to the programmer, similar to the familiar local procedure calls and act in similar ways, the main difference being that the called procedure is executed in a separate address space form the callee, the flow of control, however, is very similar. Pipes, sometimes known as *Streams*, are normally accessed using special operations such as *send* and *receive*, or by using UNIX-like read/write calls. By using RPCs it is possible to ease the programmers transition from programming a normally centralised system to programming a distributed system. It is possible for an interface language and stub compiler to be defined for an RPC mechanism. This makes it possible to automatically generate stubs for argument marshalling, data conversion, and communication, thus further hiding such details from the programmer. RPCs suffer form being inherently synchronous, i.e. the calling task cannot continue until it has received a reply form the called procedure. When results are not required, or a request needs to be multi-cast to several different receiving tasks, RPC is not an ideal model. Other forms of IPC are possible e.g. pipes and streams, which provide ansynchronicity and allow the calling task to continue. Pipes can normally be viewed as a FIFO stream of bytes, the sender appends messages to this stream for the receiver to remove at a later time, and are thus inherently asynchronous.

The efficiency of IPC is paramount if a distributed system is to be usable. Both the latency and throughput of a system's IPC mechanism can greatly affect the overall systems performance. A high latency means that every time IPC is used the responsiveness of a system is degraded, hence systems programmers will try to avoid IPC when it is not strictly required, meaning that little advantage in parallelism or sharing can be gained. The throughput of an IPC mechanism is not as important as the latency since most messages (in current systems) are very small (~30bytes), however a low throughput means that programmers will only wish to perform an IPC with small amounts of data and avoid IPC for large amounts, rather than using the mechanism by default for all operations. RPCs are the normal communication model for low latency operations, but tend to have a high bandwidth overhead for large data objects. Pipes and streams conversely have a high throughput for large data objects, but a high latency for small data objects.

2.5 Light Weight Processes

The Unix heavyweight process, where every thread of execution has its own distinct address space, has proved to be a useful abstraction when these processes do not co-operate. Processes however have a very high overhead, both for creation/deletion and context switching. It is useful in a server to be able to process several requests in parallel, e.g. a file service may service several requests while waiting for the disc server to deliver the required data blocks. Using Unix style processes to provide such request processing is overkill, and does not allow sharing of data, e.g. caches, between processes, since each process has a distinct address space (some newer versions of Unix allow shared memory between processes). If a Unix file server tries to *fork* (the Unix way of creating a process) a new process to service a request, it incurs a high overhead in this process creation since the semantics of fork require the new process to have a completely new address space. Newer versions of Unix can avoid this, but this adds to the, already large, system call interface.

It is possible to split the Unix process abstraction into two separate components: address spaces, and *threads* of execution. An address space can have one or more threads associated with it, a thread is associated with one (and only one) address space. This combination of an address space with its associated threads is referred to as a *task*.. A Unix process can be viewed as a task with only one thread. Using this task abstraction, the above server would be able to create a new thread, within the current, task to service incoming requests. Since the creation of a thread does not require the creation of a new address space, thread creation/deletion is a reasonably inexpensive operation. All the threads within one task share memory and hence they can share relevant data such as file caches. Context switch time may or may not be any quicker than in the Unix process case, it depends whether the thread we are switching to is part of the same task. If a computer has shared memory multi-processors, threads may actually be run on different processors in parallel.

2.6 Memory Management

Traditional systems such as Unix normally provide some form of virtual memory. This allows each process to have separate, protected memory spaces. These protected address spaces make the sharing of data between co-operating tasks difficult. Mentioned above was the task-thread model of processes that allows threads to co-operate within an address space. Other models are possible such as processes or tasks being able to share portions of their address space in a controlled manner by using memory management hardware to map shared data into each process's address space. This is especially useful on shared memory multi-processors. By utilising the hardware memory management facilities, schemes such as Copy-on-write become possible, i.e. two processes/tasks share a read only version of the data, when one of them tries to write to it, the data is copied, and hence both processes see separate copies. Other possibilities include the creation of a network wide virtual memory space by trapping access to the non-resident portions of the address space and performing the operation over the net.

2.7 Naming and Unique Identifiers.

Any object in a computer system that has to be accessed by a programmer/user needs to have a name of some sort. These names tend to be two types: human-readable or system-specific. A human readable name is normally stored for later browsing or retrieval in some form of directory service. These directory servers map human-readable names onto the system-specific name which tends to be a globally unique identifier - *guid*. These guids need to be unique in the whole system in question e.g. an inter-network. One way of guaranteeing uniqueness is to combine, on the creating node, a locally (to a node) unique identifier with a unique node specifier (e.g. the internet number). Such guids may be used along with other information such as access rights and check fields to produce *capabilities*.

2.8 Internetworking

Not only are computers being connected together to form networks, networks are increasingly being connected to form *internetworks*. It is thus desirable to be able to use software intended for a network on an internet. Difficulties arise sue to the different modes of communication used on networks and internets. Local area networks are inherently fast and reliable, internets slow and unreliable. Internetworking protocols have to deal with many extra overheads, such as out of order packets. It is thus undesirable to use the same networking protocols over both nets and internets. Some systems use a fast local area protocol with a gateway providing protocol conversion for long-haul communication to another network on the internet.

2.9 Compatibility And Standards

A large amount of investment has been made to the current operating systems architectures such as Unix. It would be unwise (and probably commercial disaster) to ignore this and come up with a completely incompatible system. Complete binary code compatibility with existing systems is not always necessary or desirable, compromise solutions such as source code compatibility (e.g. POSIX compliance) are often all that is required. This form of compatibility can be provided by using a set of library routines which emulate system calls, etc. To provide complete binary compatibility a more complete form of emulation is required, included the trapping of system calls by the kernel providing the emulation. Some of the proposed standards for Unix at least give a well defined interface for such systems to emulate.

3. A Review Of Some Of The Major Players

This section will examine some of the significant operating systems that are now becoming available. Special attention is paid to the features that have been described in the last section, and also to any interesting or novel features of the systems being examined. Also examined will be some of the features that have filtered through into the latest versions of Unix. It is interesting to look at the motivation behind each of the systems, since although some of the implementors may have had widely different aims, some of the resulting systems are remarkably similar.

3.1 Mach

Mach was developed at Carnegie-Mellon University as part of the DARPAs Strategic Computing Initiative [Accetta 86] [Rashid 86] [Mason 87]. Mach was designed as a reimplementation of the Unix kernel taking note that Unix had become overpopulated with features. Their objective was to simplify (*kernelize*) the kernel of Unix whilst providing support for new systems models e.g. multi-processors, IPC and new virtual memory designs. Mach is aimed mainly at the workstation/server model, although it may be used to implement the other models as well.

Mach has been implemented with a small, extensible kernel using an object-based approach. The kernel deals only with objects. A small set of primitive functions and abstractions upon which more complex services and resources can be built. The kernel itself only supports IPC, virtual memory, low-level device drivers, multi-processor scheduling and redirection of Unix kernel traps. The Mach kernel supports the task/thread abstraction of processes and communication *ports* which are logical queues for *messages* (typed collections of objects). Kernel operations are all invoked by sending messages to a kernel port. Tasks and threads are both preemptable.

The virtual memory system is integrated with the message passing system. Their design provides support for large, sparsely populated address spaces and allows tasks to allocate/deallocate virtual memory regions, set the protection attributes on regions and specify the inheritance of regions. Memory can be shared between tasks using copy-on-write or read/write semantics. Virtual memory operations such as paging may be implemented by non-kernel tasks, defaulting to the kernel's default pager if no pager is defined. The virtual memory implementation has been dived into machine and machine *independent* parts to aid portability of the operating system. The machine dependent part has a simple page validate/invalidate/protect interface, with no knowledge of outside kernel data structures.

IPC in Mach is provided by ports and messages. A port is a protected kernel object through which messages are passed between tasks. They may have several senders, but only one receiver. Access to ports is controlled by capabilities. A message contains a fixed length header followed by a variable size collection of typed data objects. Message sends can be either synchronous or asynchronous. Tasks can delegate separate threads to handle asynchronous messages. The integration of IPC with virtual memory allows very large messages to be sent by mapping data between address spaces and marking that mapped space as copy-on-write. An interface language *Matchmaker* has been defined to allow the use of RPCs in various languages, e.g. C, Pascal, Lisp. Matchmaker is an RPC stub compiler which handles the details of message marshalling and communication for the programmer.

Mach IPC does not itself provide network support, i.e. it is used solely for communicating between processes on the one (possible multi-processor) machine. Network IPC is handled by user level *Network Servers* which acts as a local representative or proxy for a task on a remote machine. Messages destined for remote machines are passed to the local Network Server which determines the destination of the message and sends over the network. The Network Server implements abstract *Network Ports*, which are the network representation of a port. The network server maps network ports to their corresponding local ports.

Mach provides a transparent remote file server. The kernel traps remote file system operations and redirects the operations to the remote file system. The file system only caches executable files since these change infrequently. A special file type is used to allow remote file systems to be linked into the local directory structure. This allows an arbitrary number of remote filesystems to be used.

Since Mach was an attempt to "kernelize" Unix, it is appropriate that Unix compatibility is high. The kernel catches Unix system traps and translates these into a message send to the appropriate user level service which performs the required action. Using this mechanism Mach can directly execute BSD 4.3 binary programs. The performance of Mach is claimed to be on par with Unix, where specific Mach features are directly exploited the performance is significantly increased. Although Mach has provided significant extra functionality compared with Unix, most of this has not been fully exploited since most versions of Mach are used as direct Unix replacements. Mach does not seem to provide any significant new models of for distributed systems.

3.2 Chorus

Chorus was based on a research project at INRIA, and subsequently developed to its current commercial status by Chorus Sytemes [Rozier 87]. The main goal of the original project was to develop and implement a distributed operating system. Chorus is aimed primarily at the workstation/server model.

The Chorus kernel or *Nucleus* was not designed as the core of any specific operating system (e.g. Unix), but rather was designed to provide a generic set of functions for supporting a variety of differing systems. The Chorus Nucleus provides local services such as process scheduling and synchronization, memory allocation and structuring, and external event handling. Also provided are global services such as IPC which support RPC, multi-cast and asynchronous messaging. The Chorus Nucleus implements a basic set of abstractions: tasks (known as *Actors*) and threads, messages, ports and port groups, Unique Identifiers, address space regions, address segments, capabilities, and protection identifiers.

The virtual memory implementation of Chorus is based on the *segment* as the unit of representation of information [Abrossimov 89]. Segments are located on secondary storage (e.g. files) and are managed by system actors called *segment servers* or *mappers*. These servers define and maintain the segment's representation, access policies, protection, consistency, and capabilities. An actor's address space is divided into a number of regions which is mapped onto a portion of a segment. References to addresses within a region are treated as references to the mapped segment and are controlled by associated access rights. The nucleus maintains a per segment local cache of physical memory pages holding portions of the segment address space. Page faults are not handled directly by the nucleus, rather they are passed onto the segment's mapper. Mappers are charged with maintaining the consistency of cached segment portions shared between actors on different nodes. The mapper is able to implement techniques such as copy-on-write for shared segment pages. The implementation of the Chorus virtual memory system is divided into machine dependent and machine independent parts.

IPC in Chorus is implemented by exchanging messages via ports. The IPC mechanism can utilise the virtual memory management to allow large messages to be transferred between address spaces by deferred copying (copy-on-write) or moving page descriptors. Messages are addressed to ports, which provide decoupling between the implementation of a service and its interface.IPC may operate in either an asynchronous or synchronous (RPC, request/reply) mode. Ports can migrate between actors allowing successive actors to implement a service (e.g. if on actor crashes another may take its place without losing messages or having the clients be notified of the change). Ports can be grouped together into *port groups* which allow multi-casts to be sent to related actors. Messages can then be sent to :

• all ports in a group
• any one port in a group
• any one port in a group at a given site
• any one port in a group on the same site as a given Unique Identifier

Network support in Chorus is supplied by the *Network Manager*. This transparently hides the location of remote ports for actors which do not know about the underlying network or protocols. Actors which do know about these can use the network manager as a communications channel. The network manager is responsible for implementing network protocols for the system, which allows for wide area networks as well as the normal local area network case. Two kinds protocols are required :

- a Chorus specific protocol which deals with port localization, remote host failure, etc.
- a system independent protocol for data transmission between sites

A Unix sub-system is provided for Unix compatibility. The sub-system is an operating system interface that sits on top of the Chorus Nucleus and catches Unix system calls, redirecting them to a user-level server which provides the appropriate action. This scheme provides Chorus with the ability to run Unix System V binaries directly. Performance is claimed to be similar to that of Unix where no specific use is made of Chorus features. Additions to the Unix subsystem have been implemented to allow Unix programs access to the increased functionality of the underlying Chorus system. Various other sub-systems can be supported and support has been provided for real-time systems.

3.3 Amoeba

Amoeba is a research project based at Vrije Universiteit and CWI in the Netherlands [Tanenbaum 90]. Their chief goal was to build a system that gave users transparent access to multiple processors on a network. They have based their approach on the pool processor model. All computers in an Amoeba system run the same minimal kernel. This kernel was designed to be small, fast, reliable and to allow users processes to provide services traditionally provided by the kernel. Amoeba was not designed with compatibility to any current (or future) operating system in mind.

An object-based approach has been taken to the design of the Amoeba system. The kernel provides only message passing and tasks. All other facilities are provided by user-level tasks communicating using the kernel message passing primitives. The kernel supports the abstractions of ports, capabilities, task and threads, and message transactions. Tasks may be preempted, but currently the threads in a task are non preemptable and run until completion or they are blocked. Process migration is non-trivial and currently requires the state of a process to be swapped out to disk, before being swapped back onto a different processor.

Very limited memory management support is provided, for instance no support is provided for paging, although whole tasks may be swapped onto secondary storage. Memory management is limited to the allocation/deallocation of memory segments and to the protection between address spaces.

IPC in Amoeba is provided by an RPC message transaction system. Messages are sent to an object via a port. Capabilities provide for location transparency and protection. Messages contain a fixed length header optionally followed by a buffer of up to 30k bytes. The system is thus optimised for very small messages, which are claimed to be the most common case. All message transactions are synchronous and no form of asynchronous messaging is supported. A reliable multi-cast system has been developed. A class based interface language with multiple inheritance, *Amoeba Interface* language or *AIL*, has been developed to provided RPC stub generation and argument marshalling.

Amoeba RPC is itself a network protocol optimised for remote-procedure calls, and thus directly allows communication between remote entities. Messages are sent to ports which are independent of their current network location and may be migrated between tasks. Port location is handled by an initial broadcast message, the location of a port is then cached for later use. If a port is migrated then this procedure is repeated. Wide-area network protocols are provided in a SWAN gateway. These provide a transparent mechanism for RPCs over WANS.

The Bullet file service for Amoeba is an server for immutable files (i.e. files once created cannot be changed). The only operations provided are create, read, and delete, and these can only operate on whole files. Files are stored contiguously both on disk and the file system cache. The size of the file system cache and the client nodes buffer space constrain the size of files that can be accessed. This allows fast whole file read/writes without the overheads of block access. Other file systems have been provided for operations that Bullet cannot perform efficiently, e.g. appending to log files and random access to files, or to provide Unix file system compatibility.

Amoeba has not been provided with an interface that can catch Unix system calls directly, instead a set of servers with library routine interfaces allows Unix (Version 7) compatible programs to be re-compiled for Amoeba. This means that the Amoeba system call interface is not unnecessarily cluttered by the requirement of compatibility. This does mean that binary Unix programs cannot be run on Amoeba systems.

3.4 The V System

The V System was a research project developed at Stanford University [Cheriton 86]. The primary goal was to research issues in distributed systems. Their approach is based on the integrated model where idle workstations can be used as compute resources for the rest of the network. Compatibility to any other operating systems was not a central issue.

The V kernel was designed using a *software backplane* analogy - i.e. the system is built up by plugging together services using the kernel as the backplane. The kernel was designed to be small and only provides IPC, process management, memory management and device management. Several services are compiled as part of the kernel since they are required to execute in the processors privileged state. Abstractions supported are: task/thread, process migration, object identifiers, object groups, messages and message send/receive. Tasks are preemptable and can be easily migrated using kernel support [Theimer 85]. The bulk of the task's state is copied to the destination node *before* the task is migrated to a remote node. Since a program cannot execute during the actual migration phase this *pre-copying* reduces the amount of time that a task is frozen.

Tasks are scheduled using a two level scheme, the kernel locally implements a simple priority based scheduler, and a process manager handles load balancing between nodes. Tasks may be executed on any node in the system and it is possible to allow the system to choose lightly loaded nodes for task execution. It is possible for users to force all tasks from a node, this is useful if the user is intending to perform highly interactive tasks on a workstation that was being used as a pool processor.

Demand paged memory management has been included in the V kernel. An address space consists of a range of addresses, *regions*, bound to open files. The kernel provides binding, caching and consistency mechanisms. Consistency between regions on different nodes is provided by a simple ownership protocol and a lock manager. File caching is provided by the virtual memory system. A process-level directory system maps file open requests onto locally cached open files. This system provides support for large memory models on diskless nodes since it provides unified cache mechanism for pages and files. The paging system can also be used to help migrate tasks onto other nodes by flushing dirty pages onto disk and paging them onto the new node as required.

IPC in the V system is provided by an RPC mechanism. Messages are sent to a process id or group of process ids. The message passing protocol is optimised for small (32 bytes) messages with an optional data segment of up to 16kbytes. All such message sends are synchronous and no form of asynchronous messaging is provided. The *copyto* and *copyfrom* primitives may be used to transfer large quantities of data efficiently. A multi-cast mechanism is available from sending messages to process groups. Messages are always sent to specific processes and although a process may be located on any node (and can migrated between nodes) no form of process transparency.

The *V Message Transfer Protocol* has been developed to support efficient message passing on a local area network. It has been optimised for the request/response behaviour of an RPC system and for small messages. When a message send is invoked the kernel either delivers the message locally or passes it to the VMTP module which then sends it to the appropriate node. VMTP supports multi-cast, datagrams, forwarding, streaming, security and priority.

V systems may be interconnected with Unix systems using VMTP. The Unix IO model is supported with the provision UIO objects. This allows C programs which use the *stdio* library routines to be compiled and run under V. Performance of V is comparable to that of Unix unless V specific functions are used.

3.5 Plan 9 From Bell Labs

Plan 9 was developed at AT&T Bell Laboratories to exploit improvements in bit-mapped personal machines, network performance, and high-performance multi-processors [Pike 90] [Presotto 88,90].Their approach is based on the pool processor model, although the pool processors in this case are shared memory multi-processors. The kernel, which runs on all nodes in the system, was designed to be small flexible and extensible.

Every object in the Plan 9 system is treated as a file. This provides a unified abstraction for all system interaction. The kernel provides process management, memory management and file operations. Lightweight processes are not implemented. All objects appear as files in the local name-space hierarchy. Services are implemented as file services and manages a set of files which provide the interface to the service. Remote services may be attached at an appropriate place in this hierarchy.

Demand paged virtual memory management is supported. A processes memory space appears as a file under the */proc* hierarchy. Processes are controlled by writing to the processes control or *ctl* file.

IPC to a service is provided using the *read* and *write* operations to a file in the file system that the service manages, these operations may be either synchronous or asynchronous. IPC to a normal process is provided by writing to the processes *note* file in the *proc* file system, this is asynchronous.

Various network services may provide access to different networks. Again all network communication is performed by doing read/write operations to a file that the service manages.

Plan 9 is has a minimal compatibility with Unix programs. Most user programs that do not rely on the system call interface can be re-compiled to run on Plan 9. The performance is said to be higher than that of Unix.

3.6 Modern Unix

Versions of Unix have recently been developed which support some of the concepts of modern operating systems [Winterbottom 90] [Leffler 89] [Bach 86]. Most of theses concepts have been added onto the exiting set of abstractions with little or no thought given to keep the original simple Unix IO model (files). Unix has evolved and many of the current features are hangovers for historical or compatibility reasons. Most distributed Unix systems fall into the workstation/server model. Since it would be difficult to summarise each of the different variations of these features, this section will just try to give a flavour of how some of the systems implement them.

The Unix process model has been extended in several ways to provide threads. User-level thread packages allow programs to be written which make use of this abstraction by linking a set of run-time routines that provide thread semantics. Since this is provided at user level without any kernel support it is often limited and/or inefficient. Some systems may provide Kernel support, but this adds to the kernel overhead and may result in extra overheads for normal processes.

Shared memory has been added System V (and some BSD variants), again this has been added as an afterthought and the underlying kernel memory management is sometime not suitable. Copy-on-write semantics for process forking has been added to some Unixes to relieve some of the unrequired memory copying overhead implied by the semantics of fork followed by *exec*. BSD has instead added the *vfork* system call, which blocks the parent process until the child has done exec or has exited.

Different IPC mechanisms have been added to Unix. BSD has added the *socket* mechanism which allows processes to communicate synchronously or asynchronously. Sockets allow a degree of transparency once they have been created, but on creation the address and protocol type are required. SunOS has added SunRPC to its version of Unix, allowing a Unix process to access another using RPC semantics. This has been built on top of the socket mechanism.

Remote files systems have been added in various ways. One of the most common is Sun's Network File System *NFS* [Reid 90]. This gives access transparency to remote file systems which have been mounted into the local file system hierarchy. NFS provides a stateless server and is not limited to the Unix file system (e.g. PC-NFS).

Most of the additions to Unix do not degrade the normal performance of a system. However, because these facilities were added on top of a system which was not designed for distributed operation they tend to be inefficient compared to similar facilities on other systems.

3.7 Comparison of Features

It is possible to see many similarities between the operating systems discussed above. In particular similar approaches have often been made in solving common problems. Most of the systems have made some attempt at providing the features discussed in section 2. RPC seems to be the most common method used for interprocess communication.

Of particular note is the similarity between Mach and Chorus. Although these systems were designed for different purposes, many of the concepts and mechanisms used are strikingly similar, and they have both taken similar approaches to Unix emulation. Of the rest many similarities also occur between Amoeba and V. Again these seem at first to have completely different aims, but implement many of the same features.

One question that does arise from the differences in the studied operating systems is that of how little functionality can be placed in the kernel to produce a workable system. Amoeba, for instance, has a very minimal kernel and provides very limited support for so-called *standard* features such as virtual memory, and yet it is claimed to be quite usable.

4. Performance

It is very difficult to try to measure and compare the performance of such systems. There are many different factors to take into consideration that may at first sight appear to be insignificant. Such factors as network load, intended usage, and specific hardware performance can be very difficult to determine.

Most of the developers of the systems examined here provide some form of performance comparisons. These are extremely difficult to compare because of the differing tests and environments that were used. It is not always possible even to believe that the results published were entirely fair, whether or not they were meant to be fair. It is very easy to publish just those results that show favourable results for your system as compared with another. Since different systems are often designed with different intended uses, it seems very dubious to measure the performance of only those sets where your system performed well. Also seemingly similar system hardware (or even differently configured versions of the same hardware) may in fact produce strikingly different results[1]. With all these problems, it can still be seen that some performance figures are quoted with seeming authority and sometimes with a certain amount of one-upmanship. These differences in test environments and intended usage should always be kept in mind when looking at the performance figures listed below.

4.1 Inter-Process Communications

[1] The author has measured the performance of NFS for some operations and has produced results which vary by a factor of nearly two between a Sun3/60 and a Sun3/75, both of which use MC68020 running at 16.67 MHz.

System	Machine	Processor	ms/call	Mbits/s
Amoeba	Tadpole	MC68020	1.4	5.3
V	Sun 3/75	MC68020	2.5	4.4
Chorus[2]	Bull SPS 7/300	MC68020	1.4	
Unix (SunRPC)	Sun 3/50	MC68020	12.2	1.6

Table 4.1

It is interesting to examine the performance characteristics of the various systems IPC mechanisms. Low-latency and high bandwidth are the two most important things that determine an IPC mechanisms overall performance [Schroeder 90].

Latency in IPC is most often measured as the time taken for the round trip of a synchronous RPC. Bandwidth is normally quoted as the maximum throughput that an IPC system can achieve.

As can be seen from table 4.1, Amoeba has both the lowest latency and the highest maximum bandwidth. However this figure may be misleading for overall system performance depending on the mix of local/remote operations. The amount of caching of data in a local node may be more important than the RPC delay/bandwidth as we will see in the file system bench-marks below.

4.2 File System Performance

Again measurement of performance data is difficult and may be misleading [Roberts 90] [Mitchell 82]. The performance of various systems is affected by such things as :

- client/server cache sizes & hit ratios
- RPC performance for remote file operations
- I/O bandwidth
- disk seek times

[2]It is not known whether chorus measurement were for local or non-local RPC

In the papers that compare the Bullet file server with NFS, the testers restrict the performance NFS by turning off the client's cache, thus all requests have to go through to the server. This is not NFS's normal mode of operation, but this might seem to be reasonable, since it is the performance of the server that was being measured, not that of the client. Also the RPC times were not taken properly into consideration. It can be shown by subtracting the RPC delays from both the Bullet figures and those of NFS, that the Bullet server is not 2 to 3 times better than NFS as claimed, but rather 1/10 to 2/10 times better (and possibly worse because NFS seems to perform more RPCs per file system operations than Amoeba). The file server overhead in these cases is very small in comparison with the communication time involved at around 10-50%.

4.3 Comparison of Performance

As stated above it is very difficult to measure the performance of different systems, let alone extract comparable data from published reports. However the Amoeba system does seem to be a clear winner on RPC times. This is probably due to the integration of the RPC mechanism with the network protocol. It is impossible to say how such figures will affect real-world performance. File system performance is even more difficult to compare and there seems to be no clear winner here. The Bullet file server looks very quick in handling immutable files, but also seems very limited and inefficient on other cases. It was not possible to list the performance figures of all features for all of the operating systems studied, since these results have not been published.

5. Conclusions

The thread/task model is well suited to the efficient implementation of objects in a distributed system. The task can be thought of as the object's class while the various threads are the objects' instances. Protection by the use of capabilities is another feature which will help alleviate the problems of object security experienced in traditional systems.

The systems studied in this report show that it is possible to design high performance distributed systems. These systems (other than Unix) can have small kernels and implement most of their functionality by passing messages to independent server processes. They also show that Unix is not really in the fore-front of operating systems technology. Against the systems studied here, Unix is showing its 20+ years of age. It is a testament to the flexible, powerful design of Unix that it has in fact survived for so long against other challengers.

Systems such as Mach and Chorus show that most of the benefits of modern systems can be gained without sacrificing any compatibility with the previous generation of programs. The compatibility of these systems with standard versions of Unix means that these systems are immediately commercially viable, and are likely to make a large impact on the operating systems market in the near future.

The source code compatibility of Amoeba, V and Plan 9 show that it is not too difficult to come up with a radically different system model whilst still retain a fair degree of Unix compatibility. These systems look set to slowly filter into the education and research environment much the same way as Unix originally did.

To make the most of the features of all of these systems, however, it takes the writing of programs with the specific functionality of the system in mind. Some of the mechanisms used in the systems studied here have made their way into the Unix world with some degree of success, but it looks as if the more functions that get added to Unix the harder it will be for Unix to move forward and still remain compatible with previous versions. The future for modern operating systems is looking favourable, Unix will find it hard to keep up.

6. Future Work

Future work involves the studying of other systems which may also be suitable for distributed object-oriented work, systems such as SOS, Clouds, Choices and Commandos. Also intended are experiments with the Amoeba system and new features, such as threads and shared memory, provided by recent versions of Unix.

7. References

[Abrossimov 89] V. Abrossimov, M. Rozier, M. Gien "Virtual Memory Management In Chorus" *Proceedings of the Progress In Distributed Operating Systems and Distributed Systems Management Workshop* 1989

[Accetta 86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young "Mach: A New Kernel Foundation For UNIX Development" 1986

[Bach 86] M. Bach "The Design of the UNIX Operating System" *Prentice-Hall* 1986

[Cheriton 88] D.R. Cheriton "The V Distributed System" *Communications of the ACM* (March 1988) pp. 314-333

[Cheriton 89] D.R. Cheriton, and T.P. Mann "Decentralizing A Global Naming Service For Improved Performance And Fault Tolerance" *ACM Transactions on Computer Systems* May 1989 pp.147-183

[Coulouris 88] G.F. Coulouris, and J. Dollimore "Distributed Systems: Concepts and design" *Addison Wesley* 1988

[Forsyth 90] C.H. Forsyth "More Taste: Less Greed ? or Sending UNIX To The Fat Farm" *Proceedings of the UKUUG Summer Conference* 1990. pp.161-172

[Gifford 86] D.K. Gifford "Communication Models For Distributed Systems" *Notes for The Advanced Study Institute* August 1986

[Gifford 88] D.K. Gifford, R.M. Needham, and M.D. Schroeder "The Cedar File System" *Communications of the ACM* March 1988 pp. 288-298

[Leffler 89] S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman "The Design And Implementation of the 4.3BSD UNIX Operating System" *Addison Wesley* 1989

[Mason 87] W.A. Mason "Distributed Processing: The State Of The Art" *BYTE* November 1988 pp. 291-297

[Mitchell 82] J.G. Mitchell, and J. Dion "A Comparison of Two Network-Based File Servers" *Communications of the ACM* April 1982 pp. 233-245

[Pike 90] R. Pike "Plan 9 From Bell Labs" *Proceedings of the UKUUG Summer Conference* 1990 pp. 1-10

[Presotto 88] D.L. Presotto "Plan 9 From Bell Labs - The Network" *Proceedings of the EUUG Spring Conference* 1988 pp. 15-21

[Presotto 90] D.L. Presotto "Multiprocessor Streams for Plan 9" *Proceedings of the UKUUG Summer Conference* 1990 pp. 11-20

[Rashid 86] R.F. Rashid "From Rig to Accent to Mach: The Evolution of a Network Operating System" *Proceedings of ASM/IEEE Computer Society Fall Joint Computer Conference* 1986

[Reid 90] J. Reid "N(e)FS: The Protocol Is The Problem" *Proceedings of the UKUUG Summer Conference* 1990 pp. 77-84

[Ritchie 74] D.R. Ritchie, and K. Thompson "The UNIX Time-Sharing System" *Communications of the ACM* (July 1974) pp. 265-375

[Roberts 90] W. Roberts "The More I Find Out The Less I Know? NFS Fileserver Benchmarks (including one that works)" *Proceedings of the EUUG Autumn Conference* 1990

[Salus 90] P. H. Salus "Standards, Specifications and Open Systems" *Proceedings of the UKUUG Summer Conference* 1990 pp. 177-180

[Schroeder 90] M.D. Schroeder, and M. Burrows "Performance of Firefly RPC" *ACM Transactions on Computer Systems* February 1990 pp. 1-17

[Swartz 90] R. Swartz "The Case Against Unix Standards" *Proceedings of the UKUUG Summer Conference* 1990 pp. 173-176

[Tanenbaum 90] A.S. Tanenbaum "Beyond UNIX - A True Distributed System For The 1990s" *Proceedings of the UKUUG Summer Conference* 1990 pp. 251-260

[Theimer 85] M.M. Theimer, K.A. Lantz, and D.R. Cheriton "Preemptable Remote Execution Facilities In The V-System" *Proceedings of The 10th Symposium on Operating System Principles*, ACM SIGOPS 1985

[Winterbottom 90] P. Winterbottom "MESHIX: A UNIX Like Operating System For Distributed Machines" *Proceedings of the UKUUG Summer Conference* 1990 pp. 237-246