Computer Science
University of Manchester

Trends in Operating Systems Towards Dynamic User-level Policy Provision

K.R. Mayes

Technical Report UMCS–93–9–1

Trends in Operating Systems Towards Dynamic User-level Policy Provision*

K.R. Mayes

Department of Computer Science
University of Manchester
Oxford Road, Manchester, UK.
ken@cs.man.ac.uk

Technical reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from ftp.cs.man.ac.uk in the directory /pub/TR. The files are stored as PostScript, in compressed form, with the report number as filename. Alternatively, reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, UK.

Abstract

It is possible to distinguish between policy and mechanism in operating system design. There is a trend to move policy out of the operating system kernel and into the user-level. This trend is described with respect to example operating system types. A system is proposed which takes this policy/mechanism split to the extreme of having the operating system kernel reduced to a hardware object which provides a low-level but abstract view of the actual hardware. Such a system would be flexible enough to allow investigation of dynamic, user-level, provision of policy.

1 Introduction

The conventional operating system maps user programs to the resources of the machine largely by means of the 'process' abstraction – the program in execution. It ensures that user processes are protected from each other whilst allowing sharing between processes in a controlled way. Resources can be shared in a time-sliced fashion on general-purpose time-share machines, or in a distributed fashion on parallel machines. The operating system provides an example of code reuse – it does not have to be written by each user. The operating system provides an interface for access to its routines. This interface may be inappropriate. Operating system interfaces tend to be 'general purpose' in order to support the requirements of all categories of users. This means that the interface primitives may not support efficient implementations for all applications. In order to provide this generality, interfaces tend to be low-level and thus need to be specialised to support particular programming languages. Despite the generality, conventional operating systems have *policy* associated with their interface primitives. For example, a new thread of control is created in Unix via the fork() system call, which creates a new address space as well, whether or not this is required. In the Chorus microkernel (Chorus, 1990), which does support a multi-threaded process model, a thread becoming blocked[1] may cause a heavyweight process switch, whereas the application may require that a thread within the same process is scheduled. This means that the semantics of the operating system primitives may be inappropriate for particular applications. Similarly, the bundling of policy into the operating system means that some users have insufficient control over the hardware.

Language runtime system (RTS) implementations are necessary because operating system abstractions are generally too low level for direct use by a language. The requirements on the operating system abstractions vary with the richness of the language (Weiser et al., 1989), so that it is difficult for a single set of abstractions to support all languages. In general, an RTS specialises the operating system abstract machine for use by a language. Most existing parallel language implementations multiplex their own process structure onto a smaller set of operating system-provided processes (Pierson, 1989). For example, Multilisp multiprocessor implementations have one operating

[1] A thread may become blocked by making a call to some kernel interface primitive, such as a message receive.

system-provided process for each physical processor. Each operating system-provided process runs a user-mode scheduler to provide Multilisp tasks (Halstead, 1989). There are many problems associated with building a language on a conventional operating system platform: Mismatches between operating system process semantics and language resource sharing requirements; lack of flexibility due to expense of process creation; overheads of operating system process context switching; blocking of all language runtime system tasks if one such task blocks in the operating system (Pierson, 1989; Halstead, 1989).

Such language-specific runtime environments inhibit language interaction. Weiser et al. (1989) listed the principal prerequisites for language interoperation as: a shared address space; symbolic name binding; shared I/O; shared data representation. Their solution was to implement a language-independent and operating system-independent runtime system, which was designed to run on top of existing operating systems but below the language runtime systems. This common layer implemented common policies and so supported interoperability. Other examples of this kind of approach are languages supporting parallel semantics such as Strand and PCN (Foster and Taylor, 1989; Foster and Tuecke, 1993), and communication paradigm abstract machines such as Linda (Carriero and Gelernter, 1989). A related approach is to reduce the complexity of the operating system interface itself: On distributed store machines, virtual shared memory (VSM) and its variants[2] (for example, Li and Hudak, 1986; Tam et al., 1990) give application programmers the illusion of programming a shared store multiprocessor and thus avoiding explicit message passing.

The operating system may not provide the user with sufficient control of the hardware. A general-purpose operating system such as Unix has facilities such as file system and process management designed to serve time-sharing users. Data management and persistent object stores need sophisticated use of caching for performance (Sechrest and Park, 1991). The caching activities and physical memory management of the operating system may be too inflexible to support these. There is a mismatch between the file organisations provided by an operating system and those needed by a database system (Unwalla and Kerridge, 1992). For example, building INGRES on top of the existing UNIX file system caused problems (Stonebraker et al., 1983)[3]. Most portable database systems implement their own storage access mechanism synchronously with the system calls using low level operating system interface routines (Unwalla and Kerridge, 1992). Process scheduling and inter-process communication can also cause problems for databases due to the overheads associated with task switching and message passing. There is also the problem caused by the operating system descheduling a process holding a short-term lock which will be required by other processes. Interestingly, Stonebraker (1981) suggested that an ultimate solution might be to provide a 'special scheduling class' for database management systems. Unwalla and Kerridge (1992) concluded that an operating system

[2] Variously termed Shared Virtual Memory or Distributed Shared Memory.

[3] Unix randomly allocates disk blocks to a file – since INGRES substantially uses sequential access, random allocation is undesirable; Unix uses indirect blocks which must be read on access to a file, presenting an overhead; Unix block size may be inappropriate for a database.

'does not help with the efficient operation of a database machine'[4] Operating system-level interference with Fortran computations may reduce the speed of the computation by reducing control of the Fortran program over the scheduling policies and hardware resources of the machine. For example, Bryant et al (1991) described their experience with Fortran users of the IBM RP3 system. These users regarded the operating system "as an adversary bent on denying them direct access to the hardware". A major problem with regard to performance of applications is that of data locality: an executing thread of control should be close to the data it will access. Operating system activity, for example by load balancing, can destroy application- or compiler-specified locality.

The provision of more control for the user means separating *mechanism* from *policy*. There are aspects of a computation which require specific access to the resources of the hardware: creating a thread of control; creating an address space; associating physical store with a virtual address; writing to disk; writing to a network. The routines which access and control hardware resources represent mechanism. Policy is that which determines in what circumstances, and for what reasons, these hardware access routines should be called. This mechanism/policy division can be demonstrated by means of two simple examples. First, the mechanism component of a process switch consists of writing to the register set, especially to the program counter (pc) and stack pointer (sp) registers for a thread of control switch, and to some hardware address translation context register (cr) for a virtual address space switch. The policy component will determine whether pc and sp alone are switched, or whether cr is switched as well. Second, the mechanism component of writing data to a disk consists (ultimately) of executing a disk device driver routine, which is passed a data buffer. The policy component would determine whether this driver write routine is called immediately, or whether the actual disk-write must be explicitly requested via some special system call.

The desirability of separating policy from mechanism has been recognised for some time, and was a motivating force in the design of the Hydra operating system (Wulf et al (1974)). The trend in operating system work has been to move policy out of the kernel and into the realm of the user. That is, to put policy at *user-level*. In general, kernel-level routines run in a privileged or protected mode. User-level routines run in non-privileged mode. The term *user-level policy* simply refers to non-privilege mode routines which implement decisions about when to call mechanism routines. The implication is that these policy routines can be provided by the *users* of the computer system themselves, rather than by the systems programmers who created the operating system kernel. There are two ways in which user-level policy can be implemented: by providing a user-level *server process* or by providing a *library* of routines. In general, system management has been implemented at user-level by server processes, whereas run-time systems for languages have been implemented by libraries. Existing microkernels support user-provision of store, file and network management by user-level servers. However, it has been argued that none of these existing systems provide sufficient flexibility for library and language implementors (Scott et al., 1990; Lazowska, 1992). New work on Psyche (Marsh et al.,

[4]Their system was based on transputers and used the micro-coded scheduler built into the transputer. This provides high priority non-preemptive mode and low priority time-slicing mode.

1991) and on scheduler activations (Anderson et al. 1992) take thread scheduling out of the kernel by enabling the kernel to make 'upcalls' to the user level.

The argument here is that the operating system interface should allow each user to define precisely the runtime system to be used to run a particular program. *By using the library approach, the bulk of the conventional operating system functionality can become effectively part of the runtime system.* This may be used to present a general-purpose abstract machine so that, for example, the user can easily port applications from one hardware platform to another. Providing low level access to hardware mechanisms would provide for fast implementations of such general-purpose abstract machines. Equally, a runtime system could support some special-purpose abstract machine, optimised for a particular application on particular hardware. The point is that the operating system should provide mechanism routines only, allowing all policy into the runtime systems. An approach like this is found in the Psyche operating system for multiprocessor systems (Scott et al., 1989). This is a minimal kernel specifically designed to support multiple programming models. Psyche provides a single address space, protection, and, as mentioned above, support for user-level scheduling. The single address space abstraction provided by Psyche provides for sharing. Psyche designers were forced to emulate a shared address space on the 24-bit virtual addresses that were available to them. However, the advent of 64-bit virtual addressing will enable systems to use a true single virtual address space. The issues related to how operating systems can use a very large address space are currently being investigated (Carter et al., 1992; Koldinger et al., 1992; Lazowska, 1992; Okamoto et al., 1992).

However, there is a drawback: whereas the resource management code within the operating system does not have to be rewritten for each application, a severely minimal kernel supporting only basic hardware-dependent operations with minimal inbuilt policy would require that each new run-time system would have to reimplement the functionality that resides in a conventional operating system kernel. Given the advantages of an object-based approach to operating system structure (Bacon, 1989; Hofmann et al., 1991), it may be appropriate to investigate various aspects of object-oriented techniques to provide for code reuse from libraries. Systems such as Presto (Faust and Levy, 1990), and particularly Choices (Johnson and Russo, 1991), achieve code reuse by inheritance.

The essence of this paper is to suggest that all policy could be implemented in user-level object-based libraries giving support, and code-reuse, for a variety of RTSs. The design, and possible interface, of a hardware-dependent kernel which allows *dynamic* switching of user-defined policy modules is discussed. The problems of designing and implementing such a dynamic system are extremely interesting in themselves, and provide sufficient reason for the work proposed in the final sections. However, further motivation can be found in the variety of applications which would benefit from having user-definable and user-switchable resource management. The most obvious applications relate to process management, specifically to scheduling. The run-time systems for parallel prologs can have schedulers for both and- and or- parallelism. Thus the overall runtime-level scheduling system must be flexible enough for applications where the kind of parallelism varies along the course of execution (Castro Dutra, 1991). With parallel Lisp, scheduling strategy is naturally two-phased, creating parallel tasks until the machine is fully utilised, then running sequentially with each task (Halstead, 1989). As was noted with a thread-based implementation of Id, "exposing scheduling to the compiler allows it to synthesise particular scheduling policies in specific portions of the program" (Culler et al., 1991). Parallel database performance is also influenced by scheduling policy. Franaszek and Robinson (1985) noted that scheduling policies which take into account the state of transactions[5] could increase the level of concurrency. This paper presents the motivations behind the design of a flexible system with a minimal kernel which is responsible for hardware-dependent activity only. All other operating system functionality is to be placed into the user-level and is to be dynamically switchable.

The bulk of the paper is concerned with a review of operating system trends towards minimal and customisable systems. In order to achieve a perspective on trends, operating systems are reviewed in the context of the facilities they provide and the kind of hardware they control. Section 2 describes segment, page and capability-based systems in order to introduce the issues of protection and process models. The early operating systems were the small *supervisors* which provided simple multi-programming environments and drove devices. Subsequently, operating systems increased in size to become giant monolithic containers for complex process and backing store abstractions. In such conventional operating systems there is an equivalence between process and address space, with mechanisms provided for sharing segments between processes. Protection may be segment or page-based. Protection domain-based machines provided a more dynamic, fine-grained approach to naming, protection and sharing. Hydra, an early capability/protection domain machine, recognised a division between policy and mechanism. Section 3 deals with the general approaches to operating systems for parallel machines. Section 4 describes parallel monolithic and capability systems. Section 5 deals with the microkernel approach which developed the policy/mechanism separation and sought to reduce operating system complexity whilst providing monolithic operating system functionality in user-space. More recently, operating system kernels have become more concerned with tailoring the kernel interface to specific requirements, rather than providing a general purpose interface. Such systems can be termed language paradigm-specific and are dealt with in Section 6. Other systems allow users to determine the interface: Customisable kernels, dating back to MU5 (Morris and Ibbett, 1979), can provide for static customisation at build-time (Section 7). In Section 8, the so-called 'nanokernels' (Lazowska, 1992) are described. Here, the further development of the policy/mechanism split is intended to support language run-time libraries and to allow language interworking (these systems are termed here 'language paradigm general') and, at least potentially, to support dynamic changes in resource management policy. Finally, in Section 9, the investigations which have occurred into how operating systems will change in response to the new 64-bit address processors are described. Section 10 discusses the trends described in the previous sections, and presents a list of aims for a new kernel. The next section, Section 11, gives an outline of a system which is currently being constructed. The system will allow investigations into operating systems design and implementation, and will act as

[5]For example, whether the transactions are executing or waiting on a lock.

a base for determining if applications can benefit from run-time switching of operating system policy. Section 12 provides a conclusion to the paper.

2 Early supervisors to monolithic kernels and capability machines

The code for handling early hardware was extremely small. On the KDF9, a copy of the operating system[6], on tape, was spliced onto the front of the program paper tape. It was 1K 48 bit words long (Warboys, personal communication). In early time-sharing systems the programs that performed various administrative or switching functions were known collectively as the *supervisor* (Wilkes, 1968). Such supervisors were resident in core, and were small; the supervisor code for the Dartmouth Time-Sharing Computer System was 9.5K in size, some running on the master and some on the slave machine. (Kurtz and Lochner, 1965). These supervisors were essentially device driver routines and a simple scheduler, plus some command interpretation. Supervisor code ran in privileged mode and was accessed via trap and interrupt. Such early timesharing systems had only one user progam resident in store; scheduling was by swapping between core and disk on timer interrupt. Interestingly, it was recognised early that a scheduling algorithm designed to suit the needs of one community of users would not necessarily be suitable for another community (Wilkes, 1968).

The complexity and size of the supervisor code increased with multi-programming. Hardware with base-limit registers for addressing with relocation gave the facility for segmentation; segmentation facilitated sharing and required protection mechanisms. A program came to be represented as a set of segments; the segments were regarded as composing the address space of the process which was the program in execution. Segmentation plus paging and associative store (for translation lookaside buffers) allowed the address space to be virtual, and be paged-in from disk. Each of the processes running on such segment-based machines has a single virtual address space associated with it, represented by a table of segment descriptors. The hardware provided a register into which the base address of the current process segment table could be written by the supervisor. This address was used by the hardware for address translation.

From this basis it is possible to trace three approaches:

1. *descriptor-based systems*
   These systems rely on segmentation hardware. A process has an associated address space. The address space of a process is defined by the segment descriptors resident in the segment table of the process, accessed via a segment table base register. In general, descriptors are not freely passed around between processes. Segments may be shared between processes by those processes explicitly requesting the kernel to map a named segment into their address spaces. This allows kernel/supervisor

[6]'Operating system' in the sense of 'routines which a user-program can use to control the hardware'

code and data to reside in all process address spaces rather than to be isolated in a separate address space. There arise protection problems:

(a) May a particular process share a particular segment? Permissions may be associated with a named segment; where this is a file, for example in Multics (Organick, 1972), those permissions will be the file permissions.

(b) Given that a segment is part of the address space of a process, how may it be accessed? Descriptor-based systems, such as Multics and VME (Warboys, 1980), implement a hierarchy of protection rings. Procedure calls across ring boundaries are validated by the kernel. In the Multics implementation of ring boundary-crossing, the segment descriptor takes on a role similar to that of a capability.[7]

(c) A particular case of controlling access to a mapped segment is the issue of how to protect the kernel code and data, where this is in segments shared by all processes. There are two alternatives:

    i. Place the kernel in pages which may only be accessed when the supervisor bit is set in the process status register. Kernel routines may then only be accessed after a trap instruction has been executed to set the supervisor bit.

    ii. Place the kernel routines in pages accessed via segments located in the high privilege protection rings.

    In Multics, the supervisor routines were placed in the innermost, most privileged ring[8]. The GE645 hardware on which Multics ran did have two modes (master and slave). However only a very few of the Multics supervisory procedures were coded as master-mode procedures (those which required the privileged instructions only accessible to master mode execution). The Multics supervisor used the general ring protection mechanism, rather than some other special mechanism, such as hardware supervisor mode, to protect itself.

Protection and sharing are at the level of the segment. Sharing of segments by more than one process may be facilitated by an indirection mechanism – at each invocation of a shared code segment, a process-specific linkage segment is produced. All linkage segments for a given procedure have the same layout (as determined by the compiler (Saltzer, 1978)) but the bindings to other segments may differ from process to process. The linkage segment serves as an indirection from the shared code virtual addresses to the private, process-local segment to be substituted for

[7]In Multics, for each ring in which a process executes there is a *separate* table of segment descriptors for that process – each of these table instances is identical except for the fault-inducing bit-patterns in the descriptors. For example, when executing in one ring, access to a particular descriptor by the process will cause a fault (because the process is crossing a protection boundary), whereas the same descriptor, when accessed in a different ring, will not cause a fault.

[8]It was intended to have the supervisor in two rings, but this was found to be too expensive in terms of the cost of frequent ring-crossing.

the virtual address (possibly via the process segment table entry). Linkages of this basic type are procedure linkage table of VME (Buckle, 1978) and linkage section of Multics (Daley and Dennis, 1968).

2. *capability-based systems* These systems basically rely on segmentation-based hardware, though there may be a special set of registers to hold the capabilities. Capabilities are essentially segment descriptors which can be freely passed between processes. The address space of a process is not static. It is defined by the C-list (a table of capabilities currently possessed by a process, equivalent to the segment table of descriptor architectures). The C-list provides the context for virtual address resolution – a virtual address is relative to the C-list base. However, the C-list used by a process changes dynamically. Capabilities are able to migrate (as parameters of a call for example) to an invocation of, say, a shared procedure. Here they are incorporated into an appropriate position in the C-list structure of the invocation. Thus the thread of control moves into the invocation of the procedure and executes in the context of a C-list composed of segments provided by both procedure and caller.

   The set of capabilities currently defining the address space of the process form a protection domain. Processes move through a series of protection domains as the execution continues. Protection and sharing is at the level of the protection domain; this can be fine-grained, and leads to object-based systems where each protected object resides in its own domain. Here, a capability list is associated with an object, in contrast to the association of the segment descriptor table with a process.

   In a typical capability system, an abstract object is defined by its Base Capability Segment (BCS), which contains private capabilities for the defined operations, internal representation and Auxiliary Capability Segment (for object-private capabilities and capability parameters of a call to the object). A caller on the abstract object has an 'enter' capability for the BCS (Corsini et al, 1984). Using this enter capability switches the caller into the context provided by the BCS – the object representation may only be accessed by procedures referenced by virtual addresses within that context[9]. Thus with capability segments, the contents of the data segment are tied to a distinguished set of operations – these are accessed via capabilities within the BCS representing the object. It should however be noted that a similar mechanism is possible in descriptor-based systems. The VME operating system supports the creation of instances of objects referenced by *currencies* (Warboys, 1980). A VME currency is a descriptor which references – possibly indirectly – a Procedure Linkage Table (PLT) containing descriptors for the segments comprising the object instance, including data segments as well as code for allowed operations. Thus the descriptor-based PLT is equivalent to the capability-based BCS in representing an object. However, in capability systems, for example Hydra

[9]Changing context in a capability system is usually achieved by a CALL or ENTER instruction accompanied by an 'enter' capability as originally described by Dennis and Van Horn (1966). This switches the current context to be the C-list of the called object.

(Wulf et al, 1974), it is possible to create new types of object and to incorporate them into the system. Descriptor-based systems place kernel in the most privileged protection ring, whereas capability-based systems have a single privilege level. Thus capability-based systems facilitate the addition of user-level policy to the operating system. Indeed it was one of the main aims of the Hydra project to support this *policy/mechanism split* (Wulf et al., 1981).

The protection of operating system code in capability systems uses the same mechanisms as user code – protected procedures. There is no system call in capability systems. Software implementations such as Hydra do have a 'kernel' which resides in a specific address space separate from user addresses[10]. However, this form of protection was used for Hydra because its kernel actually implemented the capability machine. Hydra was a small kernel (260K in size) upon which an operating system could be built using the capability architecture provided by the kernel. Other systems, which had support for capabilities in hardware, used protection domains to protect all the operating system, for example the iMAX operating system of the Intel iAPX 432[11] (Kahn et al., 1981) and the CAP capability machine operating system (Levy, 1984).

3. *page-based systems* These systems are based on the modern paging hardware represented by the MMU. Unix is taken as an example. Traditionally, the address space of a Unix process consisted of three segments: code, data and stack. More recently a Unix process, exemplified by SunOS (Gingell et al., 1987) has an address space which consists of a vector of pages. Access control is at the page level. However, the implementation of virtual memory in the SunOS kernel has the concept of a mapped object. This kernel-implemented mapping is actually called a segment: "a segment describes a contiguous mapping of virtual addresses onto some underlying entity" (Moran, 1988)[12]. The segment has become a means of *logically* grouping pages together. Its major role is to provide a mechanism for accessing the mapped object (usually a file) when a page fault occurs on an address within the segment. The segments representing mapped objects interact with the MMU paging hardware via the intermediary of page table entries and page faults.

   There are two privilege levels in Unix – supervisor and user. Protection in Unix relies on two mechanisms:

   (a) In the usual implementation, the kernel is mapped into the address space of all processes[13]. For protection, the Unix kernel resides on pages with supervisor

[10]The C.mmp multiprocessor consisted of 16 bit address PDP-11s Two bits in the PDP-11 Process Status Register were pre/appended to the 16 bit address to give an 18 bit UNIBUS address. The PSR bit values 11 represent kernel and 00 represent user address spaces.

[11]The iAPX432 system provided protection at the level of language-level objects, and as a result, despite the hardware implementation, the performance suffered and the system failed (Bacon, 1989).

[12]The mapped address range must be page-aligned.

[13]It is also possible to give the kernel its own addressing context.

access permission. Access in user-mode to kernel virtual addresses will cause an access fault.

(b) Protection between user processes is provided, for example in the Sparc MMU (Cypress, 1990), by assigning a single MMU context to a process. An MMU context is an entry in the context table used by the MMU to find the address of the page tables for the current process. All virtual address translation occurs via this context entry. Translation lookaside buffer entries also contain a context number tag. Thus, virtual addresses can only be resolved within the context of the current process. Pages can be shared between processes by requesting the kernel to explicitly map the same physical entity into the address spaces.

These protection mechanisms are simple compared to the protection rings and domains of descriptor and capability architectures. However, there are effects of this approach:

(a) Placing the kernel behind a trap interface makes for a monolithic operating system residing behind a closed system call trap interface.

(b) the non-kernel part of the address space of the process in effect becomes a single large protection domain.

(c) when segments are shared between processes, there is no control over the operation which can be applied to them, unlike, say, capability architectures.

The growth of Unix kernel executable from Version 7 (size about 170K) to SVR4 (size about 2M) can be viewed as being a result of this monolithic nature – all new functionality has to be added to the supervisor space, with an associated increase in the width of the system call interface[14].

A monolithic system arises, not from having a system call interface (the Chorus microkernel, discussed below, has one), but from close-coupling of policy and mechanism behind that interface. Ring-based systems can be just as monolithic as two-level systems. Multics placed all its supervisor in ring-0, so references to kernel routines would cause traps[15]. In VME, in the case where a user is accessing an object at a different protection level, the currency used is a *system call descriptor* which is passed to the user of an object instance. Whether the currency is a normal descriptor or a system call descriptor is determined by *loader*. If a system call descriptor is involved, then the indirection between the currency and the referenced PLT is via the virtual machine-local System Call Table and access rights checking mechanism.

[14]The number of system calls in Version 7 Unix was about 50; there are around 120 system calls in SVR4 (Stevens, 1992).

[15]Access to an address of a segment in another ring may cause a fault. The 'Fault Interceptor' calls a ring-0 module, the 'Gatekeeper' to check rights. If the call is allowed, the Gatekeeper calls a master-mode descriptor base register switching routine to point to the segment in the target ring.

It is interesting that the original two-ring supervisor intended for Multics was abandoned on the grounds of efficiency (Organick, 1972). The advantages of some kind of vertical modularity within the kernel did not apparently outweigh the overheads of boundary-crossing[16]. The simpler user/kernel division found on page-based systems like Unix has been said to represent a 'degenerate' case of the multiple protection rings of descriptor-based systems (Bacon, 1989). Such monolithic systems, having foregone the flexibility afforded by having a single privilege level, can reduce costs by having only one protection boundary to cross[17].

3 Operating Systems for Parallel Machines

There are various classifications of parallel machine architectures (e.g. Rashid, 1987; Gurd, 1988). However, only two broad classes will be considered here. Machines with multiple nodes all sharing the same physical store are termed *multiprocessors*. These correspond to UMA and NUMA architectures of Rashid (1987). Machines with multiple nodes each able to access only their own node-local store[18], that is, with *distributed store* are termed *multicomputers*. These correspond to NORMA architectures of Rashid (1987). There are two basic implementations:

1. single multi-threaded kernel with locks on shared data structures.

2. multiple kernels with communication.

Variations of these exist:

1. On a conventional shared-store multiprocessor machine, the single kernel with locks is generally used.

2. On a distributed-store multicomputer machine with a VSM implementation in hardware, the multiple kernel approach can be used, but all kernels can share instances of some locked data structures.

3. On a distributed-store multicomputer machine with a VSM implementation in hardware, the single kernel approach can be used, the VSM mechanism sharing the locked data, but also having certain 'held' data structures replicated on each node inaccessible to the VSM mechanism[19].

[16]The round-trip ring crossing during a procedure call on Multics was of the order of 2 to 3 milliseconds (Organick, 1972). This contrasts to the 20 to 30 millisecond cross-domain call on Hydra (Wulf et al., 1981) and, at the other extreme, to a hardware-implemented domain switch on the Intel iAPX 432 of 65 microseconds (Kahn et al, 1981).

[17]The cost of system calls is similar to the domain call in the iAPX 432, varying from a minimum of 70 microseconds on Sprite on a Sun 3/60 (Douglis et al., 1991), to 7 microseconds for a null system call on Plan 9 on a MIPS (Pike et al., 1991).

[18]Remote store access involves activity over some internal network.

[19]This is the other way of viewing the previous approach.

4. On a distributed-store multicomputer machine, the multiple kernel approach with communications can be used, with the communication between kernels hidden behind a remote procedure call interface.

5. On a distributed-store multicomputer machine, the multiple kernel approach with communications can be used, with the communication between kernels explicit as kernel interprocess communication (IPC).

4 Monolithic and Classical Capability

4.1 Multiprocessor

There have been a number of Unix ports to multiprocessor machines. In a multiprocessor there is a single instance of the Unix kernel. Multiple threads of control can execute in the kernel. The conventional Unix kernel is essentially single-threaded; there is, in effect, a single lock on the whole kernel, in the sense that only one *active* thread of control will be executing in kernel at one time. However, though a process running in conventional Unix kernel can never be *pre-empted*[20], a process can *sleep* in the kernel, and locks are necessary, even in conventional Unix, on data structures which might be accessed by more than one (sleeping) process. Parallelisation of the Unix kernel consists of locking critical sections of the standard Unix kernel. For example, in an SVR4/MP version, the dispatcher queue is shared by all processors; a processor runs code which acquires a lock and takes the highest priority process off the queue (Campbell et al., 1991). In the same SVR4/MP system, a lock hierarchy was necessary to parallelise the virtual memory system. A lock was first obtained for the address space structure, which also locked the segment structures. Various lower-level structures beneath this are potentially shared and must be locked. Beneath this layer are data structures associated with pages, such as the free page list, and the MMU page tables, which must also be locked. A lock hierarchy is used to prevent deadlock by ordering lock acquisitions. Data shared with interrupt-handling routines are protected conventionally by setting the interrupt priority level to mask interrupts[21]. On multiprocessor systems interrupts can occur on any processor, so interrupt level masking cannot be used for synchronisation. Either all interrupt-handling work can be passed to a dedicated kernel process which acquires locks in the usual way, or locks are associated with interrupt levels, so that interrupt handling can synchronise directly with process activity (Paciorek et al., 1991).

The capability-based Hydra kernel ran on a multiprocessor machine with one copy of the kernel so that many processors could be executing kernel code simultaneously. The designers decided to have fine-grain locks, by locking data items rather than code, and have a large number of locks. There were 'literally thousands of lock cells' in Hydra (Wulf et al., 1981). Other synchronisation mechanisms, semaphores and ports, were also

[20] A preemptive timer interrupt will set a flag to show that the time-slice is finished, and a process switch will occur on return to user level.

[21] Interrupt handling has no process context and so locks cannot be used.

provided. The iMax operating system of the iAPX 432 multiprocessor similarly relies on explicit synchronisation within the system (Kahn et al., 1981).

Descriptor-based VME has operated on the multiprocessor 2900 hardware and on the multicomputer Series 39 hardware. VME thus shows an interesting evolution. On the 2900, VME was a multi-threaded kernel where all kernel state is shared and protected by locks. In contrast, on the Series 39 multicomputer, where a node is a processor with local store, there is a separate kernel on each node. Essentially the evolution has been to separate the local and global concerns. There are node-local kernel components and global kernel components. Processes on a node map kernel segments which are specific to that node and contain node-specific state. Similarly processes map kernel-global segments for sharing global information. Node-local resources, such as processor scheduling, and store allocation and mapping, are managed locally and require local kernel state. This local state is placed in pages that are not moved by an underlying VSM mechanism. Global system data is replicated across nodes and updated by the VSM mechanism with a weak coherency protocol. Such VSM-mapped, shared data structures are protected by locks. Co-ordination between multiple nodes is via the controllers of the Series 39 machine optical fibre ring, so that kernel co-ordination is not normally required (Holt, 1992). The aim is to minimise interaction between nodal kernels by maximising local activity. Where global activity does occur, the message-passing involved is hidden by the hardware VSM mechanism.

There is an interesting comparison with the KSR machine (Rothnie, 1992) which is physically a multicomputer, but supports VSM in hardware. Whereas the multicomputer VME emphasises multiple nodal kernel instances with some shared data, KSR emphasises a single kernel instance with some private nodal data. There is a *single* instance of the OSF/1 kernel[22]. Data structures are locked and shared via the VSM mechanism. However, each node contains node-local code and data, such as that associated with page-fault handling.

4.2 Multicomputer

As discussed in the previous section, multicomputers with VSM in hardware can use the VSM to hide the message passing between nodes. Whether the kernel on each node is regarded as being a separate instance seems to be a matter of emphasis; certainly the store of each node will contain kernel code and data, though that data may be currently invalidated. Systems not possessing VSM in hardware must express the message passing between kernels in other ways. Unix on distributed store systems are generally re-implementations providing the Unix interface. A Unix-compatible kernel runs on each node, and communication between kernels is generally in the form of client-server model, via some remote procedure call (RPC) mechanism. The approach is based on the process/single address space association, where one process provides a service for other client processes. Access to the service can be implemented over a network. RPC is appropriate

[22]OSF/1 is a multithreaded Mach microkernel-based operating system. Because KSR is a highly parallel machine, the KSR operating system has more, finer-grain, locks than the original OSF/1.

for distributed Unix implementation because both RPC and standard Unix system calls are single-threaded and synchronous. As standard, Unix kernel services are obtained via the trap interface using system calls. If the service requires remote activity, access to the remote kernel occurs via RPC. Multi-threading of server processes avoids bottlenecks at the remote kernel. Examples of monolithic kernels are: Locus (Popek and Walker, 1985); DUNIX (Litman, 1988); Sprite (Ousterhout et al., 1988)[23]. The microkernels described in the next section also provide multicomputer-based Unix. RPC and server processes will be discussed further in the context of these microkernels.

5 Classical Microkernels

Mach, Chorus and Amoeba microkernels form the bases of distributed operating systems, and are concerned with giving a single system image. These microkernels are intended to support policy provided by user-level libraries. The Hydra system, mentioned above, was an early system which was concerned with presenting a policy/mechanism split and so will be discussed first.

The Hydra kernel provided mechanisms for protection and minimal policy (Wulf et al., 1981). It supported the creation and manipulation of new object types, and instances of types, so that the operating system could be extended by user-level objects that implement policy. The kernel provided low-level dispatching mechanisms only. The kernel virtual memory implementation provided for the naming of the objects via capabilities. Hydra kernel facilitated user-level policy by protecting all objects, user or operating system, in the same way. In contrast, the microkernels are derived, not from descriptor/capability ancestors, but from the paging systems where the process and address space are closely coupled (within the paging MMU). So these systems rely, not on the equality of objects each in a protection domain, but on the equality of processes each in an address space. The basis of user-level policy implementation is not an object (in the Hydra sense), but is rather a server process. As will be discussed, there are equivalences however; for example, a remote procedure call between clients and servers is very similar to a cross-protection domain call between objects. In these systems, services are provided by server processes; thus IPC tends to replace the system call as a means of accessing the operating system services. Obviously system performance is improved if IPC is fast. So, for example, handoff scheduling (Black, 1990) has been implemented in Mach to enhance message exchange.

Mach, Chorus and Amoeba are described here as being typical of microkernel systems. All three systems view an operating system as a set of servers running on top of a small microkernel.

1. **background and system image**
   Mach was first ported to uniprocessors and multiprocessors (Rashid et al., 1988), and because of this history, is generally regarded as a multiprocessor, rather than a multicomputer, operating system. In contrast, Chorus is a microkernel designed

[23] Sprite has a multi-threaded kernel.

to manage distributed systems, and is intended to give users of multicomputers the single system image that is more readily available with multiprocessors (Rozier et al., 1988; Herrmann et al., 1991). Similarly, Amoeba was designed to provide a transparent distributed system, that is, like Chorus, to provide a single system view so that a collection of independent computers appear to users as a single timesharing system. Unlike Chorus and Mach, Amoeba is based on the processor pool model, in which there is no concept of the 'home machine' of a user. Mach and Chorus both support UNIX-compatible interfaces; Mach is associated with BSD, and Chorus with SystemV (Golub et al., 1990; Guillemont et al., 1991).

2. **Provision of operating system features by servers**
   All three microkernels provide operating system facilities via user-level server processes. Mach and Chorus support UNIX. It is interesting to look at the history of support for UNIX in both Mach and Chorus because it shows the evolution of a 'pure' microkernel. Mach originally (version 2.5) provided UNIX by having a 'unix compatibility layer' of BSD code within the Mach kernel. This was removed for Release 3.0, leaving a pure Mach microkernel (Golub et al., 1990). Similarly the Chorus kernel (version 2), whilst not actually containing any UNIX software, did contain code which was specifically intended to support UNIX emulation (Armand et al., 1986; Rozier et al., 1988). Chorus version 3 changed Chorus from being a UNIX-compatible distributed operating system to being a distributed microkernel (Guillemont et al., 1991). Both Mach and Chorus now provide full UNIX emulation as user-level servers.

   Unlike Mach and Chorus, Amoeba was not intended to provide UNIX binary compatibility (Tanenbaum et al., 1990). Rather it was aimed at experimenting with new operating system facilities for distributed computing (Douglis et al., 1991). However, Amoeba does have a UNIX emulation library and a session server which handles state information when necessary (Mullender et al., 1990)[24].

   Mach and Chorus are intended to support existing operating system interfaces which can reside together as collections of user-level servers ('subsystems' in Chorus terminology). Each subsystem consists of a number of system servers; in the case of the UNIX subsystem of Chorus, these include a process manager, file manager, terminal manager, socket manager and pipe manager (Chorus, 1990). These servers cooperate to provide UNIX semantics. Similarly, the Amoeba operating system is provided by user-level servers running on a microkernel. Though Amoeba provides default servers, it is intended that these are user-replaceable (Tanenbaum et al., 1990).

   The provision of a standard operating system interface obviously makes the system more acceptable to users because of the reuse of utility software this allows. The designers of the Hydra system felt that, in retrospect, they should not have chosen

[24]However, there are differences to UNIX; the default file and directory servers, for example, have non-UNIX semantics (Amoeba, 1992).

to implement a general-purpose, time-shared system (Wulf et al., 1981). The time to build the additional software for such a system, and their lack of provision of a well thought out user interface command language, impaired the acceptance of Hydra by the user community.

3. **Policy and mechanism**
   The provision, via servers, of operating system features in these systems means that policy can migrate out of the kernel. Partitioning by server process has two aspects:

   (a) decomposition of operating system functionality by server allows a modular design.

   (b) the equivalence of server and address space means that modules are protected.

   However, some policy remains in the kernels of these systems.

   (a) **Process model**
   All three systems have a two-level process model, with the heavyweight process defining the address space in which lightweight (kernel) threads run. This equivalence of process with address space is one of the conventional properties of these microkernels. Scheduling policies are fixed by the kernel, though Mach allows 'user-level hints' to modify the scheduler's behaviour (Black, 1990). These are of two kinds: discouragement (the current thread should not run) and handoff (a specific thread should run)[25]. Mach 3.0 uses *continuations* to improve performance of the scheduler[26]. A continuation is a routine which the rescheduled thread should enter, plus a data structure representing the saved thread context state. This local state would normally be saved in a process control block and thread kernel stack, and so less store is needed by the kernel (Draves et al., 1991). Chorus implements preemptive scheduling on thread absolute priority[27] (Chorus, 1992). Thread scheduling within a process in Amoeba is cooperative (Tanenbaum et al., 1990), with round-robin time-sharing between processes (Douglis et al., 1991).

   (b) **Virtual memory**
   The valid regions of an address space of processes of the three systems are those which have been mapped to objects. Mach and Chorus allow user-level servers

[25]The kernel's use of handoff scheduling provides a good example of the unfortunate effect of putting such a policy in the kernel. Mach's message passing subsystem uses handoff scheduling inside the kernel, immediately suspending a sender and scheduling a blocked receiver. This avoids the run queue entirely and "aids performance" (Black, 1990). Kernel handoff scheduling however caused problems for Duchamp (1991) implementing a transaction manager on Mach 2, causing the creation of 25 to 35 threads when 1 thread only was needed.

[26]Mach 3.0 also uses continuations for IPC, exception handling and page fault handling (Black et al., 1992).

[27]Each thread has a *relative* priority within its actor. However scheduling is based on the *absolute* priority which is the sum of actor priority and thread relative priority.

to create and manage such memory objects. These servers provide data access routines. Communication between such memory servers and the kernel is via IPC. Memory objects implement their own policies, and can refer to network services as well as local backing store. This provides a means of implementing distributed shared virtual memory in these systems. Multiple servers can run simultaneously and provide various coherency schemes (Abrossimov et al., 1989; Herrmann et al., 1991). Amoeba has a simpler memory model, with no paging; since a process must be entirely in memory to run, Amoeba RPC can immediately access blocks of data in contiguous physical memory. Segments in Amoeba are managed by the kernel, though a segment identifier (actually a capability) can be passed between processes and the segment mapped and unmapped by passing this capability to the kernel (Tanenbaum et al., 1990).

(c) **IPC**

Threads communicate via messages sent to ports identified by global identifiers. Servers are identified by their port. Amoeba kernel provides only RPC for point-to-point communication on the grounds that send and receive primitives lead to 'spaghetti' (Tanenbaum, 1992). There are two components to IPC: a local, strictly IPC, component, and a remote, networking, component. The three microkernels being considered show a gradation of kernel involvement with these components.

Mach perhaps shows its multiprocessor origins (where there is no requirement for the network component) by originally having only the local IPC implemented in the kernel. Networking runs as a user-level 'netmsg' server on each node of a distributed Mach installation. This server implements the network protocols (Draves, 1990). Similarly, Chorus local IPC is implemented in the kernel, the network protocols reside in network servers. However, unlike Mach, these servers work in close cooperation with the kernel and share message buffers with it. In both Mach and Chorus there is a coupling between the IPC and memory management mechanisms to optimise local transfer of data by remapping physical pages and the use of copy-on-write semantics (Draves, 1990; Guillemont et al., 1991). All Amoeba IPC and networking resides in the kernel. There are two communication layers, the RPC layer and the FLIP (Fast Local Internet Protocol) layer. The FLIP layer supports process migration, and so is central to the transparency achieved by Amoeba. The existence of this gradation indicates that fast communications requires kernel implementation. This view is supported by the subsequent incorporation of the network component of IPC into the Mach kernel. This provided the performance necessary for IPC on a multicomputer, at the cost of flexibility (Barrera, 1991).

(d) **Naming**

A gradation can be seen amongst these systems in the use of capabilities for naming and protection. These capabilities are different to the capabilities of classical capability systems. Mach capabilities only refer to ports and are managed by the kernel. Chorus Ports are named by 64-bit unique identifiers.

Other entities, for example processes (called actors) and segments, are named via capabilities. A Chorus capability is a 64-bit unique identifier plus a 64-bit key. The interpretation of the key is determined by the kernel or by the server managing the named entity (Chorus, 1990). In contrast, *all* objects in Amoeba are named by capabilities. All Amoeba capabilities contain the identifier of the port of the server which manages the object to which the capability refers. Interpretation of the capability is entirely the responsibility of the server.

(e) **Processor allocation**
In multiprocessor, multicomputer and networked systems, processor allocation is an important performance issue. In such systems processors form a resource which can be allocated to computations in two basic ways: temporal allocation and spatial allocation. Temporal allocation is the classical scheduling of processes running on a particular processor. Spatial allocation is the distribution of computations over all the available processors. This section deals with the support for spatial allocation. Processor allocation policy in Mach resides in a privileged server task and in the application which is able to request processors from the server. The kernel provides the mechanism only. This placing of policy in the server provides more flexibility, and allows implementation of various policies, for example gang scheduling (Black, 1990). In Amoeba, processor allocation is determined by a user-level 'run server' which has access to CPU loading (and other) information (Tanenbaum, 1992). Chorus allows servers to set the creation sites of new processes, and so could support the provision of user-level processor allocation.

(f) **Access to operating system facilities**
There are two possible approaches to the provision of an interface to the operating system largely implemented as user-level servers. These extremes are demonstrated by Chorus and by Amoeba.

i. conventional trap interface plus IPC
Chorus operating system subsystem services, though generally provided by user-level servers, are accessed via the trap interface of a protected subsystem. It is of interest to examine the facilities provided by Chorus for building an operating system subsystem. Essentially there are three components:

A. a protected 'process manager' code and data which reside in kernel address space. The subsystem interface provides a means of trapping into this code. This provides the traditional system call interface to the operating system subsystem[28].

B. most of the functionality of the subsystem is provided by user-level servers. The Chorus IPC mechanisms are used to communicate between the protected 'system call' interface and the user-level servers.

[28]Chorus allows a kernel actor to attach handlers to both trap and exception numbers.

C. both the protected process manager and the user-level servers use the facilities of the Chorus 'nucleus' by calling the nucleus interface functions (presumably via a trap in the case of access from user-level).

ii. system call redirection plus IPC
Mach has a system call redirection mechanism so that system calls made by an application program are redirected to an emulation library. The emulation library is mapped into the application address space. The emulation library can then do RPC with the server in order to obtain the service associated with the system call. In the case of 4.3BSD Unix emulation, the server is a multithreaded Mach task.

iii. client RPC to server
Amoeba servers are accessed via libraries of stubs. Amoeba clients call the stub procedures and these send the appropriate RPC to the appropriate server. This arrangement allows users to substitute their own file servers for the Amoeba default file servers by generating their own library of server stubs.

5.1 Fine granularity Mach servers

The 4.3BSD Unix running on Mach is implemented as a single monolithic server. New work on Mach has been to decompose this single server approach into finer-grain, more general servers which can be composed together to give various functionalities. The aim of this work is to have inter-changeable components, code reuse through object-oriented techniques, and portability (Black et al., 1992). These fine-grain servers are presumably based on Mach tasks, accessed, as in the single server case, via an emulation library.

6 Language paradigm specific

Whereas the kernels in the preceding section were mainly concerned with conventional-level operating system support, some kernels are concerned, to a greater or lesser extent, with supporting particular language paradigms. Two possible approaches are to:

1. Provide a language paradigm-specific abstract machine upon which both applications and the operating system can run. The operating system then is essentially divided into that part which helps to provide the abstract machine, and that which runs on the abstract machine.

2. Provide a conventional process-based abstract machine, implemented by the operating system, where the interface primitives are specialised to support particular language run-time systems.

The Hydra kernel (Section 3), which implemented a capability machine upon which the rest of the operating system could run, can be considered to be a precursor for the first approach. However, the example of this approach which will be dealt with in this section is the Flagship system. The Flagship system is a parallel packet-based graph reduction multicomputer machine[29] (Banach et al., 1988). The European Declarative System (EDS) multicomputer system (Skelton et al., 1992) provides an example of the second approach. These systems have basically similar hardware[30], and both are intended to run declarative paradigm languages in a parallel environment. However, the approach to system software is very different in the two systems.

6.1 Operating system implementation

The lowest level of the EDS system software, the Primitive Machine Interface, presents an abstraction of the hardware to the kernel (Ward and Townsend, 1990). The upper interface of the kernel provides the Process Control Language (PCL) primitives which are basically similar to those of Mach and Chorus. Indeed, EDS kernel uses Chorus as an implementation 'starting point' (Istavrinos and Borrmann, 1990). The EDS microkernel interface has basically two components. The major part of the interface (PCL) provides mechanisms for supporting non-numeric applications in parallel: relational database and declarative-paradigm languages[31]. The requirements of the execution models for the languages are provided by the PCL interface. In particular, store coherency and virtually-shared store is required for the parallel implementation of the languages. Interestingly, the PCL primitives included (in the final version of the interface) an 'unbundled' set of store primitives which allowed each paradigm to define its own coherency protocols (Istavrinos, 1989). Earlier versions of PCL had sought to provide primitives in which coherency schemes were 'bundled'. This provides a specific example of the trend to 'unbundle' kernel interfaces to remove policy.

The second component of the EDS kernel interface is a set of primitives which support a UNIX operating system interface (Wong and Paci, 1992). Thus the EDS kernel is intended to support directly both a general-purpose operating system and specific language RTSs.

In contrast to EDS, Flagship kernel had a novel design. At a low level in the system, a Basic Execution Mechanism provided a graph reduction (GR) abstract machine. Particular actions were associated with different packet types[32] (Watson, 1987). As in EDS, there is a hardware-dependent layer. The low-level, hardware-dependent resource allocation routines[33] were implemented *imperatively* as a 'hardware ADT' on each node. The operations of this hardware object provided access to the resources of the *packet-based*

[29] In such a graph reduction machine, a computational expression is represented as a graph whose nodes are packets (linear chunks of store). A packet can contain code, base values, or pointers to other packets. The pointers form the arcs of the graph. A packet has housekeeping information and a type. The type of the packet determines how the machine behaves with respect to the packet.

[30] Flagship has nodes consisting of a single 68000 CPU; EDS has node consisting of two Sparc CPUs. Both machines have an internal Delta network.

[31] The languages are able to interwork (Wong and Paci, 1992).

[32] New computational models could be implemented by adding new packet types and defining the behaviour of the system when reducing a graph involving packets of those types.

[33] These routines deal with node-local activity and would not benefit from parallel execution.

graph reduction abstract machine. This does more than simply provide an abstraction of the hardware; it provides resource allocation appropriate for a graph reduction machine. Scheduling, for example, was based on the root packets of active computational subgraphs. Store allocation was based on pages and packets. Much of the functionality associated with conventional operating system kernels resided in this hardware ADT. The hardware-*independent* part of the operating system kernel was written in Hope+ (Perry, 1987), with annotations to handle state, and so ran on the graph reduction abstract machine. This part of the operating system (called the Flagship 'kernel') could potentially benefit from implicit parallelism in the Hope+ program. From these graph reduction computations it was able to call the imperatively-implemented hardware ADT operations.

Flagship operating system did not provide the abstract machine for the languages, but rather ran on the same GR abstract machine (Mayes and Keane, 1993). The implicit parallelism obtained from the GR machine was able to be utilised by the operating system kernel itself[34].

6.2 Process model

EDS is relatively conventional: a heavyweight process (task) defines an address space within which lightweight threads of control are active. A software implementation of VSM means that a task can be distributed, and have threads running on more than one node. An extra level is introduced into the process model, between a task and thread, to provide a context which represents the collection of threads of a task on one node – the 'team'. EDS, with its conventional equality of process and address space, is a server-based system. The Flagship operating system took a different approach. The units of functionality were not server processes, but rather passive objects; instances of FADTs. The Flagship *process* was the unit of resource allocation (Leunig, 1987); not of execution or of virtual addressing. The schedulable entities in Flagship were active subgraphs, rather than explicit threads of control.

6.3 Inter-node activity

The EDS system is optimised for fast message passing. Of the two Sparc processors on each node, only one receives interrupts from the network – processing of network activity is restricted to this CPU. IPC (including RPC) is via ports, but with a 'connect-first' semantics for performance. The existence, in the kernel interface, of primitives for the establishment of connections provides an example of the specialisation of the PCL interface for database applications.

As noted above, a virtual address space can be distributed over several nodes, using software-implemented VSM. Thus, regions of the virtual address space can be shared

[34]There is a nomenclature problem here. Hydra 'kernel' provided a capability abstract machine for capability-style operating system to run on top of it. Flagship 'Hardware FADT' code and Basic Execution Mechanism code provided a GR abstract machine upon which the Flagship 'kernel' ran.

irrespective of nodal distribution. Special hardware on each node allows copying of 128 byte 'sectors' rather than entire 4K pages (Ward and Townsend, 1990). Thus, remote store access is supported at three levels in the EDS machine (Istavrinos and Borrmann, 1990):

1. Page copying and remote update is provided in the processor architecture.
2. Virtual memory management is provided by the kernel, including management of the address space of a *distributed* task.
3. Store coherency policy is controlled by handlers which are provided by each language-specific execution model.

The EDS system supports both RPC for function shipping and VSM for data shipping.

In contrast, message passing on Flagship was implicit in the graph reduction execution mechanism. Flagship did not have VSM; rather, it had a single global address space. Addresses in Flagship were [node, address] pairs. The interpretation of the address was done by the Basic Execution Mechanism of the system. If the address referred to the store of a remote node, then the computation on that subgraph is suspended, and a request for a copy is sent to the remote node (Watson, 1990). During message latencies, the processor does other work. This type of access to global addresses represents data shipping. Flagship also implemented function shipping for computational graphs involving access to state[35]. Here, the graph would be exported to the node containing the state, and execution occurred on that node.

6.4 Protection

Protection in the EDS system is based on the conventional association between heavy-weight process and virtual address space. Thus protection is afforded by the MMU addressing context.

Flagship has a global address space and so had protection based on protection domains. Accesses between protection domains was to be via *capability packets* (Holdsworth et al. 1989). The execution of code associated with a subgraph occurs in the context of an environment. This environment defines the protection domain of the root packet of the computation; references to packets in other protection domains would be via capability packets[36].

6.5 Modularity of design

Both systems can be classified as being object-based in design. The EDS kernel is designed in terms of C++ classes. The Flagship kernel was designed in terms of ADTs and implemented using Hope+ modules.

[35] The term 'state' here refers to *updatable* state. On Flagship such state was held in special packets and special rules were associated with computations involving stateholder packets.

[36] This was not implemented

The units of functionality within the EDS system are servers, as in Mach and Chorus.

The Flagship kernel, which ran on top of the graph reduction abstract machine, was structured as a collection of subsystems[37] each of which consisted of several manager ADTs[38]. Each subsystem was constructed to maximise the amount of local, as opposed to global, activity and so reduce bottlenecks and increase parallelism (Keane and Mayes, 1992). The amount of parallel activity within the operating system was however reduced because of serialisability constraints imposed on access to the state of the manager ADT instances. Although Flagship has many specialised features, it also shares many features with systems such as Psyche. It provides a good basis for the design of a flexible kernel, particularly the representation of the hardware-dependent layer as an ADT instance. The state of this ADT instance was provided by system data structures accessible by *primitive procedures* which formed the operations of the hardware ADT. Conversely, the hardware ADT instance could communicate with the kernel via a *notifications* interface whose component operations were provided by the manager ADTs of the kernel subsystems.

7 Customisable operating systems

These operating systems are grouped together because they all allow some degree of customisation of the kernel. This customisation may be for a particular hardware port, or may be change the behaviour of the kernel to specialise it for a particular purpose. These systems exhibit the 'operating system family' approach. Three systems will be discussed: MU5 (Morris and Ibbett, 1979), x-kernel (Hutchinson and Peterson, 1988) and Choices (Russo and Campbell, 1989).

The MU5 operating system, intended to run on a range of systems from single processor to a network of several processors, is reminiscent of the Mach/Chorus approach. It had a small kernel[39] and several 'virtual machines' (processes) running operating system tasks.

The x-kernel is an operating system kernel for networks of machines (Peterson et al., 1990). It supports a library of protocols so that it can access different network resources (such as RPC and file access) with different protocol combinations. The x-kernel has a fixed process manager and store manager, whose designs support the needs of communications. The primary feature of the x-kernel is to allow the construction of kernels with different communication protocols using an object-oriented infrastructure (Hutchinson and Peterson, 1988). The x-kernel views everything as a protocol, including user processes and devices.

Choices is an operating system for multiprocessor architectures, based on the idea of using object-oriented programming and inheritance to build a kernel from layers that are collections of objects (Russo et al., 1988). Johnson and Russo (1991) reported that

[37]For example, process, store, environment managers.

[38]For example globalStoreManager, processStoreManager and localProcessStoreManager, instances of which constituted the storeManager subsystem.

[39]18K, half of which was paged, and half resident.

Choices consisted of process mangement, VM management, file systems and networking. Choices was being ported to a distributed store machine (Russo and Campbell, 1989).

There are two important issues when considering customisation:

1. Whether the way in which the customisable components of the operating system are implemented facilitates **reuse of code**.
   In MU5, the modules which make up the operating system are source files. Similarly, in the x-kernel a protocol object is represented as a C source file. In contrast, in Choices, the implementation language is C++. Thus Choices achieves a policy and mechanism split (orthogonally to the user/kernel division) by defining behaviour using *abstract* classes, and implementation using *concrete* classes, a technique which depends on inheritance (Russo et al., 1988). A particular system component can be specialised for a particular requirement by deriving a subclass from a class representing an abstraction of that component. In addition to inheritance, Choices also reuses code by structuring subsystems as collections of connected objects (Johnson and Russo 1991). .

2. Whether the **configuration** of the operating system occurs **statically** at build-time, or can occur **dynamically** at run-time.

   In MU5 configuration was static; the operating system source was a matrix of files, in which one row contained all versions of one module of the operating system. A particular operating system could be built by choosing an element from each row. Module interaction was via interface procedures. This structure allowed not only building different hardware-compatibilities into kernels, but also provided a variety of, for example, scheduling policy.

   The intention of the designers of the x-kernel is to decompose conventional protocols so as to define a collection of 'library protocols' from which programmers can build new protocols that provide exactly the communication support they need (Hutchinson and Peterson, 1988). The version of x-kernel described in 1990 statically loaded all protocols to be configured into a given kernel. Howver, it was intended to enhance the system for dynamic loading of protocols (Peterson et al., 1990).

   Choices achieves code reuse and customisation using both inheritance and by building new object collections from existing classes. However, inheritance fixes the behaviour of an object at creation time (Micallef, 1988) and is essentially static. In contrast, the polymorphism provided by recombining collections of objects gives late binding of procedure calls, and is a dynamic mechanism. However, it is not clear that Choices provides dynamic loading and reconfiguration of the kernel.

8 Language paradigm general

The systems in this section have been called 'nanokernels' by Lazowska (1992). As this term suggests, a major feature of such systems is that they are intended to be even more minimal than microkernels, with even more policy residing at user level. The best example of this type of system is Psyche. Other work on aspects of this approach is found in the scheduler activations implementation of Anderson et al. (1992). The main aim of Psyche was to allow processes from dissimilar programming models to share data abstractions and to synchronise correctly, hence the focus on a single global address space and user-level provision of process management. Whereas Mach, Chorus and Amoeba allow for user-level provision of memory management, network management and file management, these projects are concerned with putting thread scheduling into user space[40]. The advantage of user-level threads are performance and flexibility. However, many existing user-level thread packages map onto kernel-based threads of control. These suffer from too close coupling between user and kernel threads. When a user thread blocks in the kernel, or is rescheduled by the kernel, then either the entire set of threads blocks, or there are synchronisation problems on critical sections (Marsh et al., 1991; Anderson et al., 1992). The implementation of user-level threads in Psyche and scheduler activations seek to de-couple user and kernel threads. The two implementations differ from each other, but both are based on the provision, by kernel threads, of virtual processors on which to run user threads. However, if a user thread running on a kernel thread blocks, then that kernel thread, or a kernel thread from a pool of kernel threads, can be assigned to run a different user thread. Both systems are based on a two-way interaction between user and kernel levels. The basic role of the kernel is to deliver events to the appropriate user-level thread package routine rather than deal with the events itself. Coarse grain resource management, such as multiprogramming remains in the kernel. The kernel preemption associated with multiprogramming presents a problem where user-level threads may be entering critical sections. The scheduler activations work takes a multiprocessor UMA solution to this by migrating the thread and completing preempted critical section execution on a different processor. However, in Psyche, which runs on a multiprocessor NUMA architecture, the kernel issues a warning to the thread package that it is about to be preempted. In line with Psyche's aim of providing multimodel interworking (Scott et al., 1990), Psyche allows for the coexistence and interaction of different thread packages.

The designer of Psyche intended for the provision of policy by the writers of libraries and language runtime systems. However, there appears to be no facility, as a part of the Psyche system, for code reuse to support these.

The mechanisms used by Psyche are extremely interesting. In Psyche the conventional link between process and address space does not exist. Psyche presents a single address space abstraction. Logically there is a single global virtual to physical address mapping in the system. This logical memory map has regions of the single virtual address space mapped to specific modules or *realms* which provide code and data. There exist many

[40] Recent work has implemented scheduler activations in the Mach microkernel (Barton-Davis et al., 1993).

instances of this logical mapping. Each instance is a particular hardware MMU map and represents a *protection domain*. Each realm of code and data originates in its own protection domain, and threads originating in that realm run in that protection domain. Access by such a thread to an operation of an unmapped realm will cause a page fault. Depending on access rights associated with the thread and with the target realm, the kernel may either return an error, or map the realm into the current protection domain and continue the thread, or start a new thread in the target protection domain to run the target operation (Scott et al., 1989; Marsh et al., 1991).

This means that in Psyche the creation of shared memory segments occurs implicitly on access to a shareable region. This is unlike the situation in more conventional models like that of Mach and Chorus, where processes explicitly map shared segments of their address spaces. In Mach and Chorus, page faults on such shared segments involve IPC from the kernel to the server managing the previously mapped segment. This implicit mapping of a segment of address space also differs from classical capability approaches.

The limitations of Psyche are related to its NUMA design, in contrast to Mach, Chorus and Amoeba which will operate on loosely-coupled machines transparently. For example, IPC in Psyche is via invocation of operations of realms shared between more than one protection domain (Scott et al., 1990). Whilst this is acceptable for a NUMA architecture, Psyche needs to be extended to support multicomputers. Networks could be incorporated by either using a shared virtual memory approach or by having special network interface realms to support cross-machine operations (Scott et al., 1989).

9 Single address space

Several systems have been constructed with a single, global address space. Flagship and Psyche have already been mentioned. A capability-based machine, the IBM/38 supported a flat, single-level 64-bit virtual address space[41]. Other operating system work is starting to focus on the large address space model associated with the advent of 64-bit addressing. In such systems the standard process equals address space association can break down (Carter et al., 1992). Process virtual address spaces will be regions in the single address space, and protection mechanisms based on the protection domain model will be required (Lazowska, 1992). However, virtual address translations will be the same in all protection domains, facilitating sharing.

The three systems discussed here are the Opal system (Koldinger et al., 1992), the Rice system (Carter et al., 1992) and the Toshiba system (Okamoto et al., 1992).

9.1 Single-level store

The Opal system (Koldinger et al., 1992) is an operating system design which supports a single global virtual address space that maps all primary and secondary store in a

[41]The hardware supported only a 48-bit physical address but when an object is created, microcode extends the address by adding an additional 16-bit field (Levy, 1981).

local area network. Okamoto et al. (1992) from the Toshiba Corporation have described a micro-kernel design which takes advantage of the single large address space available with 64-bit CPUs. There is a one-level store – all files are mapped into the address space and accessed by virtual address.

Having the backing store as part of the virtual address space reduces the number of abstractions supported by the kernel. There is a thread of control and a protection domain consisting of address space regions. The Toshiba system has just two abstractions: the thread and the memory section, though the 'thread' is a combination of thread of control and protection domain. The memory section is similar to the Mach memory object. However, here the memory section is a segment in the *single* virtual address space, and is thus addressed by a single, unique virtual address.

9.2 Protection

Protection is provided for conventional multiple-address space systems by the address translation hardware and associated tables. Single address space proposals seek similar support for protection.

In Opal, the hardware basis of protection is at the level of the MMU TLB. A protection lookaside buffer has been proposed (Koldinger et al., 1991) which allows a separation of protection information from issues of address translation. In the Toshiba micro-kernel, protection between virtual address space sections is via access control lists associated with the memory sections and with threads accessing them. These access control lists are to be loaded into MMU page table entries and into TLB entries for access-time checking.

The Rice proposal takes a different approach, using explicit traps rather than hardware detection of access faults. Here cross-protection domain calls are detected at bind-time by a name server. The name server plants a stub into the code. If the call is to cross a protection domain, then the stub causes a trap for access checking (Carter et al., 1992).

9.3 Distributed implementation

The single global address space provides the ability to map all primary and secondary store entities into the same address space on all nodes of a distributed machine. The conventional RPC client-server model tends to be used to support distribution of computation.

The Rice approach is to provide RPC for function shipping (remote invocation) and VSM for data shipping (local invocation). At runtime, the mechanism actually used for invocation is determined by the type of stub planted by the name server into the code of the executing thread.

The Toshiba kernel has a 'gate' section mapped into a protection domain which represents an indirection for access checking between RPC client and server code.

Management of physical storage backing a virtual segment in Opal is implemented by user-level memory servers as in Mach and Chorus. Access to the servers is via Amoeba and Chorus-like capabilities which are interpreted by the server and not by the kernel (Mach capabilities require kernel verification). These capabilities are associated with the server ports (Lazowska, 1992). Similarly, with the Toshiba kernel, in a distributed environment the virtual address space can be backed by external pagers, as in Mach and Chorus, to provide network transparency.

10 Discussion of trends

Perhaps the major trend which can be distinguished is the tendency towards 'unbundling' the operating system. There are two, interrelated trends.

1. The unbundling of the design of monolithic kernels into uncoupled, interacting modules. These can be moved into user space, allowing a lower level kernel interface.

2. The unbundling of policy imposed by the classical operating system interface, in order to provide lower-level primitives. These primitives allow the language or classical operating system interface implementor to construct exactly the required high level interface.

Based on the above trends, it has been possible to move policy out of the kernel. In existing systems this has been achieved with memory management, network management, file management, and processor management (i.e. scheduling and processor partitioning). Kernels generally possess a distinct hardware-dependent layer upon which the rest of the kernel is built. In general, it is this low level code which implements the 'mechanisms'. Building user-level policy modules more or less directly on this layer requires some facility for upcalls into user-level. Specific solutions to moving policy out of the kernel require different implementations on different architectures.

A kernel can provide the conventional monolithic system call interface, a low level interface with facilities for plugging in user-level policy, or an interface supporting a specialised abstract machine for support of specific languages.

Protection and modularity are related. In monolithic systems, everything needing protection was placed in the kernel. Server-based systems give modularity and protection at the level of the conventional process running in an address space. Protection domain systems give modularity and protection at the level of the object or ADT instance[42].

The conventional idea of a process in an address space can give way to the more general idea of a thread of computation moving through protection domains.

Operating systems can assist in hiding physical distribution. The two main abstractions provided to hide message passing, in systems which have the conventional process-address space equivalence, are RPC and VSM. These represent function shipping and

[42] Amoeba supports objects and associates object capabilities with its RPC mechanism. The interpretation of the capability is carried out by the RPC server process and not, as in Hydra, by the kernel. Similarly, the Amoeba object identifier is valid only in the context of the server and is not universally valid as in capability/protection domain architectures.

data shipping[43] respectively. RPC also represents a mechanism for a thread of computation crossing a protection barrier between a client and a server address space. In protection domain-based systems, other mechanisms are used to hide message passing and provide data and function shipping. A single address space facilitates sharing, but the hardware protection provided by MMU context switches is no longer available. It is possible that a single address space can provide a single abstraction for hiding message passing in a distributed machine. Provision of a common interface for user-level policy modules for process management facilitate language interaction at runtime. The sharing provided by a common address space assists such multi-model working.

10.1 Aims for a new kernel

This section denotes a list of interrelated features which would seem, on the basis of the trends identified, to be desirable at the operating system level.

1. To allow policy at user-level.
2. To allow different high level abstract machines to be supported[44].
3. To allow dynamic change of policy between applications[45].
4. To allow interworking between paradigms.
5. To allow abstract machines to explicitly and easily control the distribution of their computations over the distributed hardware[46].
6. To allow subsequent porting to 64 bit addressing machines by supporting both process address space and single global address space paradigms.

There is an extra item which was mentioned in the Introduction: To allow reuse of user-level operating system code. There are two forms of reuse: libraries of resource management entities to which applications can link; resource management *sources* which can be reused to derive new manager entities tailored for particular purposes.

11 A flexible kernel as the basis for experimentation

Items from the above list can be regarded as providing a system for *investigating* many operating system level issues. User-level provision of resource management policy facilitates experimentation.

Whilst in a general-purpose system there would be three major components:

[43] This assumes the *conventional* implementation of VSM, which involves making a local copy of the referenced remote data.

[44] Preferably without reconfiguring the kernel.

[45] And preferably *within* applications.

[46] As determined by policy defined in user-level libraries.

1. A development environment in which specific resource management classes can be easily defined and derived from pre-existing classes. Object-oriented techniques allow both modularity and code-reuse. Such an environment should provide, for example, a browser for pre-existing libraries, and a development route for instantiating new libraries.

2. A minimal hardware-dependent kernel which provides an abstraction of the hardware features and whose design emphasises *dynamic* behaviour and flexibility. This would be represented as a hardware abstract class.

3. A linker/loader mechanism which can act as a bridge between the hardware dependent kernel and user-level resource manager objects.

it is proposed initially to emphasise work on the design of the hardware-dependent layer. The target for implementation is the EDS machine, a sparc-based multicomputer.

11.1 A hardware abstract class

The kernel should form an abstraction of the hardware, as in the 'hardware_ADT' of the Flagship system. Low-level, but general, operations provide controlled access to specific low-level hardware resources on specific machines[47]. The location of the hardware-dependent interface will be determined by:

1. policy/mechanism split – if policy decisions are involved then the code should reside at user-level.

2. supervisor-mode machine instructions – code requiring access to supervisor-mode instructions resides, behind a trap instruction, in the hardware object.

No definition of the implementation of the hardware class is included in the hardware class definition so, provided the interface operations are sufficiently general, porting of the system to various hardware platforms is facilitated. Furthermore, the hardware class provides a definition of the interface operations, or *downcalls*, which can be used by the hardware-independent, user-level manager code. The *trap interface* should be internal to a hardware object, implicit within the hardware operations. That is, on a given CPU architecture, some hardware object operations will require a trap instruction, some will not. The provisional hardware class iterface will be described in Section 11.5.

11.2 Resource management abstract classes

A set of resource manager abstract classes will be defined.

[47] Such as registers and devices.

1. Interface operations define the interaction between resource managers. Such de-coupling between managers allows functional replacement of managers[48].

2. Support must be provided for the re-use of user-level code. Since it is the intention for the kernel to provide only an abstraction of the hardware facilities, most of the code will reside at user-level. Without some kind of explicit support for such coding, the usefulness of the system will be severely limited. Manager code reuse is facilitated by object-based techniques in the provision of user-definable policies.

3. These interface operations also define the interface for *upcalls* from the hardware class.

11.3 Dynamic behaviour and flexibility

The initial purpose of the system is to act as a basis for experimentation with resource management policies, where they should be sited with respect to the conventional user/kernel division, and how they interact. Thus the design must provide a flexible platform for dynamic switching of policy and siting of policy.

11.3.1 User-level policy

1. Associated with the user-level resource manager object libraries must be some loader mechanism. In order to allow dynamic loading of objects at runtime, the loader must have an interface with the hardware object.

2. The loader can also be defined by its interface operations and implemented as an object, whose state is composed of the object library.

3. It is possible to place the loader at the user level, and to associate a loader instance with a particular computation.

4. This would give flexibility in loading policy, and potential for restricting the access to certain libraries.

11.3.2 Reconfiguration of hardware object

There should be some mechanism available so that the hardware object behaviour can be reconfigured interactively. It is interesting to consider a mechanism based on a 'computation0'; the computation loaded by the bootstrap, and which provides the base environment of the system. It may be possible to associate a switch to this computation0 with a keyboard-generated interrupt. The process manager, say, of computation0 could then provide a command interpreter for reconfiguring the kernel. The computation0 switch

[48] It is not clear that complete de-coupling is possible. There may be some compatibility problems with respect to levels of sophistication of resource management. So, for example, a complex Process Manager may have some dependencies on having access to a Virtual Memory Manager of equivalent sophistication. This is a topic for investigation.

must be placed behind some security mechanism, such as a password kept in hardware object state.

11.3.3 Moving policy from user to supervisor-level

The flexibility and dynamic behaviour so far discussed is with respect to switching user-level policy and reconfiguring the hardware object. A further useful feature is enable the user-level policy code and data to run in supervisor mode. *This would allow investigation of performance implications of calls across the trap/return_from_trap interface.* A resource manager running in supervisor mode would need to be designated as 'trusted', in that it has access to all supervisor privileges and can bypass all protection. This means generalising the 'upcall' mechanism so that all calls to resource manager operations look the same, whether implemented as an upcall or an in-supervisor procedure call.

Thus the proposed system has three basic components: a set of user-level management objects, a supervisor-level hardware object, and zero or more members of the set of user-level management objects running as privileged objects in supervisor mode.

11.4 Support for multiple computations

The system should be able to support multiple computations, where each computation will be associated with its own set of resource manager objects. A single computation *could* consist of several threads of control, possibly running in several virtual addressing contexts. A computation would thus determine its own process structure and scheduling policy. The hardware object simply loads registers as instructed by downcalls from the Process Manager object of the computation. However, where more than one computation is running, the hardware object will be responsible for allocating CPU time to each. Placing support for time-slicing of computations in the hardware object means adding code and state associated with this policy. This does *not* imply the re-introduction of general operating system policy into the hardware object: At any time the resource management policies currently controlling the behaviour of the machine will be those of the management objects of the current computation. This *does* imply that conflicting policies can be implemented by two run-time systems co-existing on the machine. Each run-time system will be associated with a separate computation, and so with the customised[49] resource management objects of that computation. This co-existence of computations is discussed in the following two subsections.

11.4.1 Support for protection of computations

1. Protection between computations is afforded by the hardware via virtual address translation mechanisms. To protect one computation from another, they are given separate sets of virtual to physical address translation contexts[50].

[49]Customised for the run-time system.

[50]It is possible to implement cross-computation sharing by allowing separate computations to access the same contexts.

2. Protection is lost if the hardware object allows user-level control of changing virtual address translation context. Thus there must be some checking done, in the hardware object, on reception of a request by a computation to change translation contexts. This means that the hardware object must contain some state relating to rights to a context. This should be minimal and as free from policy as possible.

3. The hardware object will be mapped into all virtual address translation contexts, but the mapping will be supervisor-mode access only. Physical pages containing kernel data and code must not be remapped into some user-mode accessible region of the address space.

11.4.2 Support for time-slicing of computations

1. In order to implement time-slicing, the hardware object needs to switch between computations, on timer interrupts. That is to switch between sets of resource manager objects.

2. This presents problems of interrupting user-level critical sections. The system design must support user-level critical sections. This may mean some policy in the hardware object.

3. It is envisaged that it may be the case that a user may wish to have sole-access to a system, and may wish an application to run as the sole computation on the machine. Thus there should be some mechanism available *at run-time* to switch off time-slicing. The 'computation0' environment mentioned above could provide a command for switching off preemptive time-slicing.

11.5 Hardware class operations

The precise definition of the hardware class operations necessary for optimal performance and protection must be the topic of research. However, a provisional list of operations can be described. The interface can be divided into four components:

1. Those operations concerned with upcalls to the user-level resource managers, and with setting up state, in the hardware object, associated with a computation.

   (a) To create a computation, that is essentially to set up pointers to the user-level resource managers. These objects will be available via the loader object specified for that computation.

   (b) In order to avoid problems with calling and returning across the trap interface, it is proposed to implement upcalls as a signal-like mechanism. There will thus be operations associated with setting pointers to handler routines and signal queues.

   (c) To change resource manager objects, used for on-the-fly change of policy.

2. Those operations concerned with writing to, and reading from, the hardware registers, including the hardware address translation context.
   (a) To allocate or deallocate a hardware address translation context.
   (b) To write or read the current hardware address translation context, to allow user-level schedulers to switch between address spaces.
   (c) To write or read the general register context for a thread of control switch. Although this may be done from user-level on some hardware, there may be associated problems which require supervisor-mode operation, such as flushing register windows in Sparc architectures.
3. Those operations concerned with manipulation of virtual address spaces and physical store.
   (a) To validate or invalidate address ranges, associated with the option of eager physical store allocation. That is, without waiting for a page fault to occur before allocating store.
   (b) To allow a computation to allocate or free physical store. Such operations must be associated with resource accounting.
   (c) To associate physical store addresses with particular virtual address ranges, in particular hardware address translation contexts.
   (d) To set and to read hardware address translation data strutures. For example, level three page table entries of the Sparc MMU.
4. Those operations concerned with manipulating devices.
   (a) To add or remove device driver objects to the system.
   (b) To access particular device driver code. For example, send and receive message operations would be provided, not by the hardware object, but by a network manager object residing at user-level. The hardware class will simply provide an operation for writing to and reading from the network. This will, in turn, call the appropriate network device driver routine.

12 Conclusions and future work

This paper has described trends in operating systems. Two trends are of particular interest when considering support for applications on parallel machines: moving policy out of the operating system kernel, and providing efficient support for specific language abstract machines. It is proposed that by providing a low-level interface to hardware mechanisms, and providing sufficient protection, much of the operating system policy can be sited in user-level run-time libraries. Moreover, it may be possible for a computation executing on such a system to dynamically change this policy at run-time. A system, which is to use object-oriented techniques of code re-use and modularity, has been outlined in this paper. The proposed system is currently being implemented on the EDS multicomputer system.

13 Acknowledgements

The author would like to thank Warwick Heath, John Keane, Andy Nisbet, Stuart Quick, and Brian Warboys for contributions to, discussions about, and criticisms of, this work. Stuart Quick has begun the implementation of the proposed system and has made a significant contribution to the initial definition of the hardware class. Thanks also to the referee for helpful comments on the paper as originally submitted.

14 References

Amoeba (1992) Amoeba Reference Manual. User Guide. Vrije Universiteit.

Abrossimov, V., M. Rozier and M. Shapiro (1989) Generic virtual memory management for operating system kernels. Proc. 12th ACM Symp. on Operating System Principles, 123-136.

Anderson, T.E., B.N. Bershad, E.D. Lazowska and H.M. Levy (1992) Scheduler activations: Effective kernel support for the user-level management of parallelism. ACM Transactions on Computer Systems 10(1), 53-79.

Armand, F., M. Gien, M. Guillemont and P. Leonard (1986) Towards a distributed UNIX system – The CHORUS approach. EUUG Conference Procedings (Autumn), 413-431.

Bacon, J. (1989) Evolution of operating system structures. Univ. of Cambridge Computer Lab. Technical Report 166.

Banach, R., J. Sargeant, I. Watson, P. Watson and V. Woods (1988) The Flagship Project. IEE/BCS UK IT 88, Conference Publication, University College Swansea (July), 242-245.

Barrera III, J.S. (1991) A fast Mach network IPC implementation. Proc. Usenix Mach Symp. (November), 1-11.

Barton-Davis, P., D. McNamee, R. Vaswani and E.D. Lazowska (1993) Adding scheduler activations to Mach 3.0. Univ. of Washington Technical Report 92-08-03.

Black, D. (1990) Scheduling support for concurrency and parallelism in the Mach operating system. IEEE Computer 23(5), 35-43.

Black, D., D.B. Golub, D.P. Julin, R.F. Rashid, R.P. Draves, R.W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan and D. Bohman (1992) Microkernel operating system architecture and Mach. Proc. Usenix Workshop on Microkernels and Other Kernel Architectures (April), 11-30.

Bryant, R., H. Chang and B. Rosenburg (1991) Experience developing the RP3 operating system. Usenix Association Distributed and Multiprocessor Systems (Summer), 1-18.

Buckle, J.K. (1978) The ICL 2900 series. Macmillan Press.

Campbell, M., R. Barton, J. Browning, D. Cervenka, B. Curry, T. Davis, T. Edmonds, R. Holt, J. Slice, T. Smith and R. Wescott (1991) The parallelization of UNIX System V Release 4.0. Proc. Conference of Usenix Association (Winter), 307-323.

Carriero, N. and Gelernter (1989) Linda in context. Comms ACM 32(4). 444-458.

Carter, J.B., D.B. Johnson, A.L. Cox and W. Zwaenepoel (1992) Distributed operating systems based on a protected global virtual address space. Rice COMP TR92-186.

Castro Dutra, I. de (1991) A flexible scheduler for the Andorra-I system. LNCS 569, 70-82.

Chorus (1990) CHORUS kernel v3.2 – Specification and interface. Chorus Systemes CS/TR-89-60.

Corsini, P., G. Frosini and L. Lopriore (1984) The implementation of abstract objects in a capability based addressing architecture. Computer J. 27, 127-134.

Culler, D.E., A. Sah, K.E. Schauser, T. von Eicken and J. Wawrzynek (1991) Fine-grained parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. 4th Int. Conf. on Architectural Support for Programmming Languages and Operating SYstems (April), 164-175.

Cypress (1990) Sparc Risc user's guide. Cypress Semiconductor.

Daley, R.C. and J.B. Dennis (1968) Virtual memory, processes, and sharing in MULTICS. Comms ACM 11, 306-312.

Dennis, J.B. and E.C. Van Horn (1966) Programming semantics for multiprogrammed computations. Comms. ACM 9, 143-155.

Draves, R. (1990) A revised IPC interface. Proc. Usenix Mach Symp., 101-121.

Draves, R., R. Dean, B. Bershad and R. Rashid (1991) Continuations: Unifying thread management and communications in operating systems. Proc. 13th ACM Symp. on operating systems principles (October), 122-136.

Duchamp, D. (1991) Experience with threads and RPC in Mach. Usenix Association, Distributed and Multiprocessor Systems (Summer), 87-104.

Faust, J.E. and H.M. Levy (1990) The performance of an object-oriented threads package. ECOOP/OOPSLA Proceedings (October), 278-288.

Foster, I., C. Kesselman, and S. Taylor (1990) Concurrency: Simple concepts and powerful tools. Computer J. 33(6), 501-507.

Foster, I and S. Tuecke (1993) Parallel Programming with PCN. Argonne National Laboratory ANL-91/32.

Foster, I and S. Taylor (1989) Strand: New concepts in parallel programming. Prentice Hall.

Franaszek, P. and J.T. Robinson (1985) Limitations of concurrency in transaction processing. ACM Transactions on Database Systems 10(1), 1-28.

Gingell, R.A., J.P. Moran and W.A. Shannon (1987) Virtual memory architecture in SunOS. Usenix Association Conference Proceedings (Summer), 81-94.

Golub, D., R. Dean, A. Forin and R. Rashid (1990) Unix as an application program. Usenix Association Conference Proceedings (Summer), 87-95.

Guillemont, M., J. Lipkis, D. Orr and M. Rozzier (1991) A second-generation micro-kernel based UNIX; lessons in performance and compatibility. Usenix Association Conference Procedings (Winter), 13-21.

Gurd, J.R. (1988) A taxonomy of parallel computer architectures. The Design and Application of Parallel Digital Processors, IEE International Specialist Seminar, Conference Publication No. 298, Lisbon, Portugal, April, 57-61.

Halstead, R.H. (1989) New ideas in parallel List: Language design, implementation and programming tools. LNCS 441, 2-57.

Herrmann, B., M.I. Ortega, L. Philippe (1991) UNIX on a multicomputer: The benefits of the CHORUS architecture. Chorus Systems Technical Report CS/TR-91-46.

Holdsworth, S., J.A. Keane and K.R. Mayes (1989) Aspects of protection on the Flagship machine: binding, context and environment. ICL Technical Journal 6, 757-777.

Holt, N. (1992) Virtual shared memory in commercial applications. Univ. of Manchester Centre for Novel Computing Virtual Shared Memory Symposium.

Hofmann, F., P. Schlenk and T. Eirich (1991) Encapsulation and interaction in future operating systems. LNCS 563, 241-246.

Hutchinson, N.C. and L.L. Peterson (1988) Design of the x-kernel. Proc. of the SIGCOMM 1988 Symp. (August), 65-75.

Istavrinos, P. (1989) Specification of the Process Control Language (PCL). ESPRIT EP 2025 Deliverable, EDS.DD.1S.0007.

Istavrinos, P., and L. Borrmann (1990) A process and memory model for a parallel distributed-memory machine. LNCS 457, 479-488.

Johnson, R.E. and V.F. Russo (1991) Reusing object-oriented designs. Univ. of Illinois Technical Report UIUCDCS 91-1696.

Kahn, K.C., W.M. Corwin, T.D. Dennis, H. D'Hooge, D.E. Hubka, L.A. Hutchins, J.T. Montague, F.J. Pollack and M.R. Gifkins (1981) iMax: A multiprocessor operating system for an object-based computer. Proc. 8th Symp. on Operating System Principles, 127-136.

Keane, J.A. and K.R. Mayes (1992) Resource management on a packet-based parallel graph reduction machine. LNCS 634, 417-422.

Koldinger, E.J., H.M. Levy, J.S. Chase and S.J. Eggers (1991) The protection lookaside buffer: efficient protection for single address-space computers.

Koldinger, E.J., J.S. Chase and S.J. Eggers (1992) Architectural support for single address space operating systems. Dept of Comp. Sci. and Eng. Univ. of Washington Technical Report 92-03-10.

Kurtz, T.E. and K.M. Lochner (1965) Supervisory systems for the Dartmouth time-sharing system. Computers and Automation 14(10), 25.

Lazowska, E.D. (1992) System Support for high performance multiprocessing. Usenix Association Proc. Symp. on Experiences with Distributed and Multiprocessor Systems (March), 1-11.

Leunig, S.R. (1987) Abstract Data Types in the Flagship System Software. Flagship Document FLAG/DD/303, ICL.

Levy, H.M. (1984) Capability-based computer systems. Digital Press.

Li, K. and P. Hudak (1986) Memory coherence in shared virtual memory systems. Proc. 5th ACM Symp. Principles of Distributed Computing, 229-239.

Marsh, B.D., M.L. Scott, T.J. LeBlanc and E.P. Markatos (1991) First-class user-level threads. ACM operating system Review 25(5) (Proc 13th ACM Symp on operating system Principles (Oct, 1991)), 110-121.

Mayes, K.R. and J.A. Keane (1993) Levels of atomic action in the Flagship parallel system. Concurrency: Practice and Experience 5(3), 193-212.

Micallef, J. (1988) Encapsulation, reusability and extensibility in object-oriented programming languages. Journal of Object Oriented Programming 1(1), 12-35.

Moran, J.P. (1988) SunOS virtual memory implementation. Conf. Proc. of European Unix User Group (April), 285-300.

Morris, D. and Ibbett, R.N. (1979) The MU5 Computer System. MacMillan Press.

Mullender, S.J., G. van Rossum, A.S. Tanenbaum, R. van Renesse and H. van Staveren (1990) Amoeba – A distributed operating system for the 1990s. IEEE Computer 23, 44-53.

Okamoto, T., H. Segawa, S.H. Shin, H. Nozue, K. Maeda, M. Saito (1992) A micro kernel architecture for next generation processors. Proc. of Usenix Workshop on Micro-kernels and Other Kernel Architectures (April), 83-94.

Organick, E.I. (1972) The Multics system: An examination of its structure. The MIT Press.

Paciorek, N., S. Lo Verso, A. Langerman (1991) Debugging multiprocessor operating system kernels. Usenix Association Proc. Summer Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMSII), 185-201.

Perry, N. (1987) Hope+. Internal Document. IC/FPR/LANG/2.5.1/7, Department of Computing, Imperial College, London.

Peterson, L., N. Hutchinson, S. O'Malley and H. Rao (1990) The x-kernel: A platform for accessing internet resources. IEEE Computer 23(5), 23-33.

Pierson, D.L. (1989) Integrating parallel Lisp with modern UNIX-based operating systems. LNCS 441, 312-315.

Pike, R., D. Presotto, K. Thompson and H. Trickey (1991) Designing Plan 9. Dr. Dobb's J. 172, 49-60.

Rashid, R. (1987) Designs for parallel architectures. Unix Review (April), 36-43.

Rashid, R., A. Tevanian Jr., M. Young, D. Golub, R. Baron, D. Black, W.J. Bolosky and J. Chew (1988) Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. IEEE Transactions on Computers 37(8), 896-908.

Rothnie, J. (1992) Kendall Square Research introduction to the KSR1. Proc. of the 1992 DAGS/PC Symp. (June), 200-210.

Rozier, M., V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard and W. Neuhauser (1988) CHORUS distributed operating systems. Chorus Systemes Technical Report CS/TR-88-7.6.

Russo, V.F., and R.H. Campbell (1989) Virtual memory and backing store management in multiprocessor operating systems using object-oriented design techniques. OOPSLA '89 Proceedings (October), 267-278.

Russo, V., G. Johnston and R. Campbell (1988) Process management and exception handling in multiprocessor operating systems using object-oriented design techniques. OOPSLA '88 Proceeedings (September), 248-258.

Saltzer, J.H. (1978) Naming and binding of objects. LNCS 60, 99-208.

Scott, M.L., T.J. LeBlanc, and B.D. Marsh (1989) Design Rationale for Psyche, a general-purpose multiprocessor operating system. University of Rochester Computer Science and Engineering Research Review 1988-1989, 5-13.

Scott, M.L., T.J. LeBlanc, and B.D. Marsh (1990) Multi-model parallel programming in Psyche. Proc 2nd ACM SIGPLAN Symp on Principles and Practice of Parallel Programming (March, 1990), 70-78.

Sechrest S. and Y. Park (1991) User-level physical memory management for Mach. Usenix Association Mach Symp., Nov, 1991 189-199.

Skelton, C.J., C. Hammer, M. Lopez, M.J. Reeve, P. Townsend and K.F. Wong (1992) EDS: A parallel computer system for advanced information processing. LNCS 605, 3-20.

Stevens, W.R. (1992) Advanced programming in the Unix environment. Addison-Wesley.

Stonebraker, M. (1981) Operating system support for database management. Comms ACM 24(7), 412-418.

Stonebraker, M., J. Woodfill, J. Ranstrom, M. Murphy, M. Meyer and E. Allman (1983) Performance enhancements to a relational database system. ACM Transactions on Database Systems 8(2), 167-185.

Tam, M., J.M. Smith and D.J. Farber (1990) A taxonomy-based comparison of several distributed shared memory systems. ACM Operating System Review 24(3), 40-67.

Tanenbaum, A.S., R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, J. Janson and G. van Rossum (1990) Experiences with the AMOEBA distributed operating system. Comms ACM 33(12), 46-63.

Tanenbaum, A.S. (1992) Modern operating systems. Prentice Hall.

Unwalla, M. and J. Kerridge (1992) Control of a large massively parallel database machine using SQL catalogue extensions, and a DSDL in preference to an operating system. LNCS 618, 138-155.

Warboys, B.C. (1980) VME/B a model for the realisation of a total system concept. ICL Technical Journal, 132-146.

Ward, M. and P. Townsend (1990) EDS hardware architecture. LNCS 457, 816-827.

Watson, P. (1987) The Flagship Basic Execution Mechanism. Flagship Document FS/MU/PW/018-87, Department of Computer Science, University of Manchester.

Watson, P. (1990) The FLAGSHIP parallel machine. In Multiprocessor Parallel Architectures, eds T.J. Fountain and M.J. Shute, 57-82.

Weiser, M., A. Demers, and C. Hauser (1989) The Portable Common Runtime Approach to Interoperability. Proc. 12th ACM Symp. on Operating System Principles, 114-122.

Wilkes, M.V. (1968) Time-sharing computer systems. MacDonald.

Wong, K-F and M. Paci (1992) Performance evaluation of an OLTP application on the EDS database server using a behavioural simulation model. In Parallel Processing and Data Management Ed P. Valduriez, Chapman and Hall, 317-350.

Wulf, W., E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson and F. Pollack (1974) HYDRA: The kernel of a multiprocessor operating system. Comms ACM 17(6), 337-345.

Wulf, W.A., R. Levin and S.P. Harbison (1981) HYDRA/C.mmp, an experimental computer system. McGraw-Hill.