A Survey of Multiprocessor Operating System Kernels (DRAFT)

Bodhisattwa Mukherjee (bodhi@cc.gatech.edu)
Karsten Schwan (schwan@cc.gatech.edu)
Prabha Gopinath (gopinath_prabha@ssdc.honeywell.com)

GIT–CC–92/05

5 November 1993

Abstract

Multiprocessors have been accepted as vehicles for improved computing speeds, cost/performance, and enhanced reliability or availability. However, the added performance requirements of user programs and functional capabilities of parallel hardware introduce new challenges to operating system design and implementation.

This paper reviews research and commercial developments in multiprocessor operating system kernels from the late 1970's to the early 1990's. The paper first discusses some common operating system structuring techniques and examines the advantages and disadvantages of using each technique. It then identifies some of the major design goals and key issues in multiprocessor operating systems. Issues and solution approaches are illustrated by review of a variety of research or commercial multiprocessor operating system kernels.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332–0280

Contents

1 Introduction

Parallel processing has become the premier approach for increasing the computational power of modern supercomputers, in part driven by large-scale scientific and engineering applications like weather prediction, materials and process modeling, and others, all of which require GigaFlop computing speeds and Terabytes of rapidly accessible primary and secondary storage. However, perhaps more important than the HPCC applications named above are commercial motivations for the development of parallel machines, which include improved machine cost/performance, scalability to different application requirements, and enhanced reliability or availability.

The purpose of this survey is to review some of the major concepts in operating systems for parallel machines, roughly reflecting the state of the art in the early 1990's. More importantly, we identify the main research issues to be addressed by any operating system for a multiprocessor machine, and we review the resulting solution approaches taken by a variety of commercial and research systems constructed during the last decade. Moreover, it should be apparent to the reader upon finishing this paper that many of these solution approaches are and have been applied to both parallel and distributed target hardware. This 'convergence' of technologies originally developed for parallel vs. distributed systems, by partially divergent technical communities and sometimes discussed with different terminologies is driven by recent technological developments: (1) multiprocessor engines, when scaled to hundreds of processors, can appear much like distributed sets of machines, and (2) distributed machines linked by high performance networks (especially local area networks or network devices derived from current ATM or even supercomputer routing technologies) are being increasingly used as parallel computing engines.

One self-imposed limitation of this survey is its focus on performance rather than reliability in parallel systems. Reliable systems are surveyed in several recent articles, including [237, 28, 149]. A second limitation of this survey is its treatment of operating system kernels rather than operating systems, thereby neglecting system functionalities like file systems, database support, network protocols, and others. This focus reflects an unfortunate lack of attention paid to such issues in many previous operating research projects and even in some commercial systems. Recent work is rapidly correcting such deficiencies. It includes industry efforts to offer concurrent I/O or file system support [114, 115] or even concurrent databases on parallel machines [195], work on communication protocols for high performance and parallel machines [110, 147], and research efforts addressing efficient file management on parallel machines [89, 31]. Such work is motivated by the intended use of parallel machines for commercial, large-scale data processing, by upcoming programs like NASA's EOS satellites which will generate Terabytes of data that have to be processed and re-processed for use in earth science applications [252], and it is motivated by the recent convergence of high performance computing and networking technologies resulting in large-scale, physically distributed, and heterogeneous parallel machines.

Brief survey of multiprocessor hardware. Several current textbooks in parallel computing provide good overviews of parallel machine architectures [111, 6, 233, 188]. For purposes of this paper, we briefly review some of the major types of parallel machines, eliding architectures for SIMD programs, functional programs, and systolic applications.

Depending on the coupling of processors and memory, multiprocessors may be broadly divided into two major categories:

- *Shared memory multiprocessors.* In a shared memory multiprocessor, all main memory is accessible to and shared by all processors, as shown in Figure 1. Shared memory

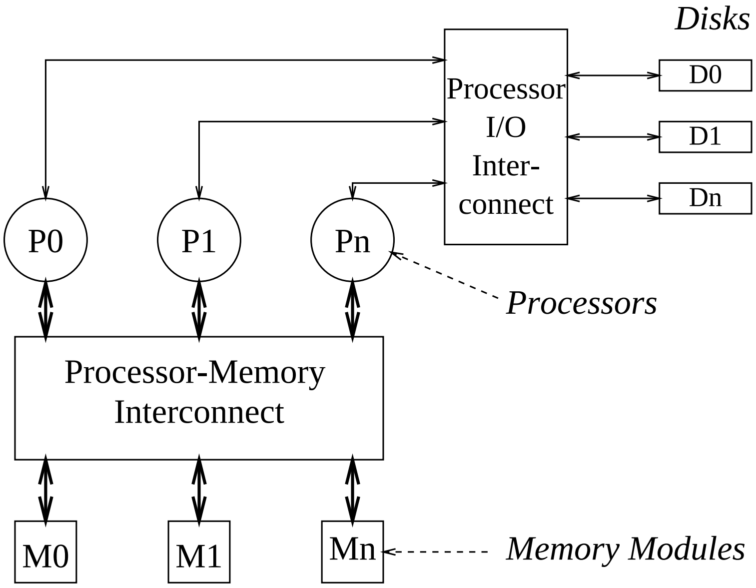

Figure 1: Multiprocessor Architectures

multiprocessors are classified on the basis of the cost of accessing shared memory:

1. *Uniform Memory Access (UMA) multiprocessors.* In an UMA architecture, the access time to shared memory is the same for all processors. A sample UMA architecture is the bus based architecture of the Sequent multiprocessor [224], where a common bus links several memory modules to computing modules consisting of a cache shared by two processor elements, and I/O devices are attached directly to the bus.
2. *Non-Uniform Memory Access (NUMA) multiprocessors.* In a NUMA architecture, all physical memory in the system is partitioned into modules, each of which is local to and associated with a specific processor. As a result, access time to local memory is less than that to nonlocal memory. Sample NUMA machines are the BBN Butterfly parallel processor [130] and the Kendall Square Research supercomputer [195]. The BBN machines use an interconnection network to connect all processors to memory units, whereas the KSR machines use cache-based algorithms and a hierarchical set of busses for connecting processors to memory units. In both machines, I/O devices are attached to individual processor modules.

- *NO Remote Memory Access (NORMA) multiprocessors.* In this class of architectures, each processor has its own local memory that is not shared by other processors in the system. Hypercubes like the NCube multiprocessors, past *Intel iPSC* machines and current *Intel iSC* mesh machines [114, 115], the Thinking Machines CM-5 [246, 140], and workstation clusters are examples of non-shared memory multiprocessors. Workstation clusters differ from hypercube or mesh machines in that the latter typically offer specialized hardware for low-latency inter-machine communication and also for implementation of selected global operations like global synchronization, addition, or broadcast.

UMA architectures are the most common parallel machines, in part because most such machines are simply used as high throughput multiprogrammed, multi-user timesharing machines, rather than as execution vehicles for single, large-scale parallel programs. Interestingly, although all memory is accessed via a single shared bus, even UMA machines often have NUMA characteristics because individual processors access shared memory via local caches. Cache misses and cache flushing can result in effectively non-uniform memory access times. Furthermore, bus contention may aggravate variability in memory access times, and scalability is limited in that the shared global bus imposes limits on the maximum number of processors and memory modules it can accommodate.

A NUMA architecture addresses the scalability problem by attaching local memory to each processor. Processors directly access local memory and communicate with each other and with remote memory modules through an interconnection switch. One type of switch is an interconnection network consisting of multiple levels of internal nodes, where systems are scaled by addition of internal switch nodes, as in the BBN Butterfly multiprocessors [130, 64]. A second type of switch consists of a hierarchical set of busses [121, 195], where access times to remote memory depend on either the number of internal switch nodes on the access path between the processor and the memory or on the number of traversed system busses. Because a NUMA architecture allows a large number of processors in a single machine, many experimental, large-scale multiprocessors are NUMA machines, an example being the *IBM RP3* which was designed to contain up to 512 processors [45, 44], and the *KSR* machine again offering up to 512 processors.

NORMA multiprocessors are the simplest to design and build, and have become the architecture of choice for current supercomputers like the Intel Paragon [114, 115], recent Cray machines, and others. In the simplest case, a collection of workstations on a local area network constitutes a NORMA multiprocessor. A typical NORMA multiprocessor consists of a number of processors interconnected on a high speed bus or network; the topology of interconnection varies. One major difference between NUMA and NORMA multiprocessors is that there is no hardware support for direct access to remote memory modules. As a result, NORMAs are more loosely coupled than NUMA machines. However, recent advances in supercomputer technologies are leading to tradeoffs in remote to local memory access times for NORMA machines (e.g., roughly 1:500 for local vs. remote memory access times) that can approximate those achieved for shared memory machines like the KSR (roughly 1:100). This suggests that future NUMA or NORMA parallel machines will require similar operating system and programming tool support in order to achieve high performance parallelism.

A main component of a multiprocessor is its interconnection network. It connects the processors, the memory modules and the other devices in a system. An interconnection network, which may be static or dynamic, facilitates communication among processors and memory modules. A few sample interconnection networks are: time shared or common buses, crossbar switches, hierarchical switches, and multistage networks. The design, structure and performance of various interconnection networks have been reviewed in other literature [264, 111, 6, 233, 188] and are beyond the scope of this survey.

The variety of different kinds of multiprocessor architectures coupled with diverse application requirements have resulted in many different designs, goals, features, and implementations of multiprocessor operating systems, in university research projects and in the commercial domain. This paper examines a few such projects and commercial endeavors. The remainder of this paper is organized as follows. Section 2 briefly reviews a few common structuring techniques used to build an operating system. Section 3 discusses some key design issues. Finally, Section 4 examines a few sample multiprocessor operating system kernels developed for research and commercial purposes.

2 Structuring an Operating System

A multiprocessor operating system is typically large and complex. Its maintainability, expandability, adaptability, and portability strongly depend on its internal structure. Different techniques for structuring operating systems [25] are described in this section, along with a discussion of some of the effects of such structuring on the ease with which an operating system can be adapted for use with multiprocessor computing engines. The various structuring techniques described in this section are not mutually exclusive; several such techniques may be used in the construction of a single system.

While the bulk of this paper focusses on operating system kernels, this section must occasionally comment on the organization of the entire operating system. In this context, we define an operating system *kernel* as the basic operating system functionality permitting use of the processors, the main memory, the interconnection network, and the other devices of the parallel machine. Higher level operating system functionalities like user interfaces, file systems, database support, and networking are not unimportant, but their discussion is outside the scope of this paper, in part because their performance will be strongly affected by the performance attributes and basic functionality of the underlying system kernel.

2.1 Monolithic Systems

Some operating systems such as Unix [196], OS/360 [166] and VMS [143] have been implemented with large, monolithic kernels insulated from user programs by simple hardware boundaries. No protection boundaries exist within the operating system kernel, and all communication among processes implementing higher level operating system functionality (e.g., file system daemons) happens through system-supplied shared memory or through explicit message-based communication constructs. It has been shown that the lack of a strong firewall within the large operating system kernel, combined with large kernel sizes and complexities, make such monolithic systems difficult to modify, debug and validate. The shallowness of the protection hierarchy makes the underlying hardware directly visible to a large amount of complicated operating system software. Monolithic systems are also extremely difficult to adapt for use in a distributed environment, and most such systems have no facilities that allow users to change some specific service provided by the operating system.

Recent experiences with the implementation of monolithic operating system kernels for parallel machines (e.g., the Sequent's or SGI's operating systems) have been confined to UMA machines. Attempts to build such systems for large-scale parallel machines like the RP3 have met with mixed success. As a result, more recent work in multiprocessor operating systems for machines like the RP3, the BBN Butterfly, and the KSR supercomputer have been based on Mach or on OSF Unix, both of which have smaller kernels and offer some facilities for kernel and operating system customization for different application domains and target machines (see Section 4.3 for a discussion of the Mach operating system's configuration support).

2.2 Capability Based Systems

In a capability-based system [70], each accessible entity exists in its own protection domain, but all entities reside within a single name space. A capability is both a name and a set of access rights for an entity, and is managed by an underlying hardware or software kernel. A process is not allowed to reference an entity for which the process' current domain does not have a capability. An entity can be shared by more than one domain, but a process in one domain can access and manipulate such an entity only by invoking an access method for which the process has sufficient rights (according to its capability). Invocation across

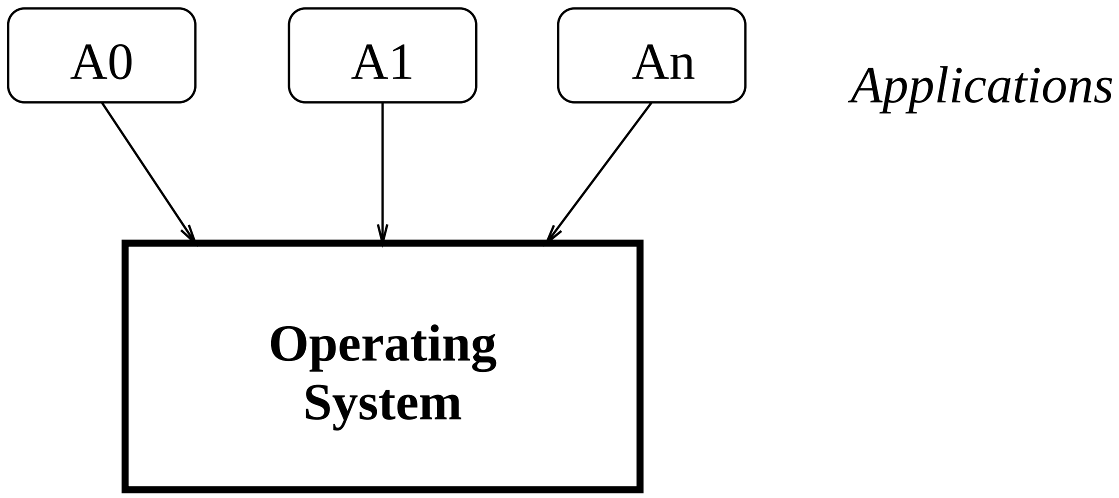

Figure 2: Monolithic Systems

protection domains happens via a *protected procedure call*, in which a process in one domain having an appropriate *execute* capability, transfers control to a second domain, and executes entirely within the context of the second domain. Parameter passing is by reference between the caller and the callee.

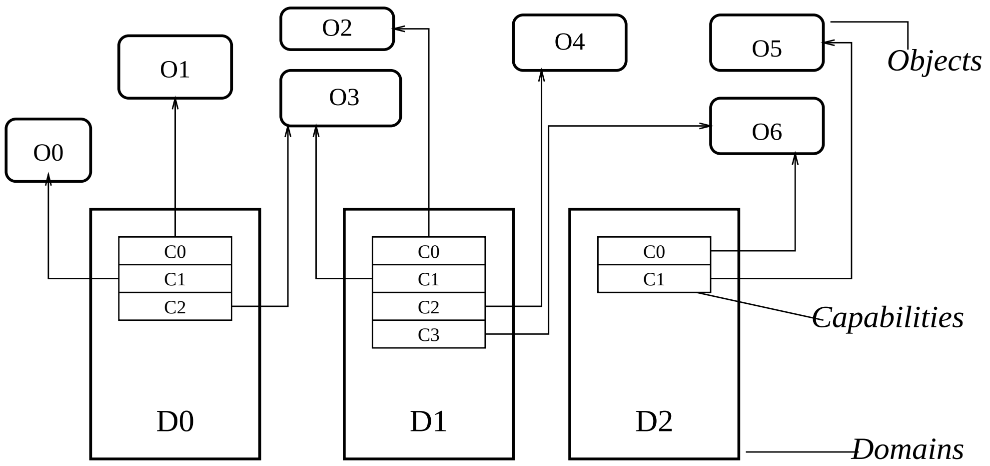

Figure 3: Capability Systems

The implementation of capability based addressing has been carried out using both software and hardware techniques. A few sample software based systems are Hydra [258] and Cal/Tss [235]. CAP [179] and the Intel/432 [62] are examples of hardware based systems. Despite early favorable predictions, system builders have been largely unsuccessful in implementing and programming capability based systems which perform as well as machines based on more traditional memory reference models [132, 59]. This may be due to the fact that most early research and commercial systems focussed on the use of capabilities for enforcement of protection boundaries and system security characteristics, typically by enforcing the *Principle of Least Privilege*. This principle states that each program and each user of a system should operate using the least set of privileges necessary to complete a job [206]. Unfortunately, the principle's implementations often implied the *Principle of Least Performance*, which means that anything that was protected was also expensive to access, so that users attempted to avoid using a system's protection mechanisms, and implementors of commercial operating systems avoided using protection mechanisms to the maximum extent possible. It appears, however, that the extremely large address spaces offered by modern 64 bit architectures are beginning to reverse this trend, in part because it is evident that program debugging and maintainability require some fire-walls within the 64 bit addressing range potentially accessible to a single parallel program.

2.3 Message Passing Systems

In a message passing system, a process always remains within its address space; communication among processes happens through message transfer via a communication channel. When a process in one address space requests a service from another address space, it creates a message describing its requirements, and sends it to the target address space. A process in the target address space receives the message, interprets it and services the request.

THE [72], Thoth [56], and Demos [16] are a few examples of the earliest message passing systems. The primary motivation behind the design of these systems was to decentralize the structure of an operating system running on a single computer. On the other hand, the motivation behind the latter message passing systems such as RIG [191], V [54], Accent [193], and various hypercube operating systems [205, 221] was to build an operating system on a structure of distributed computers.

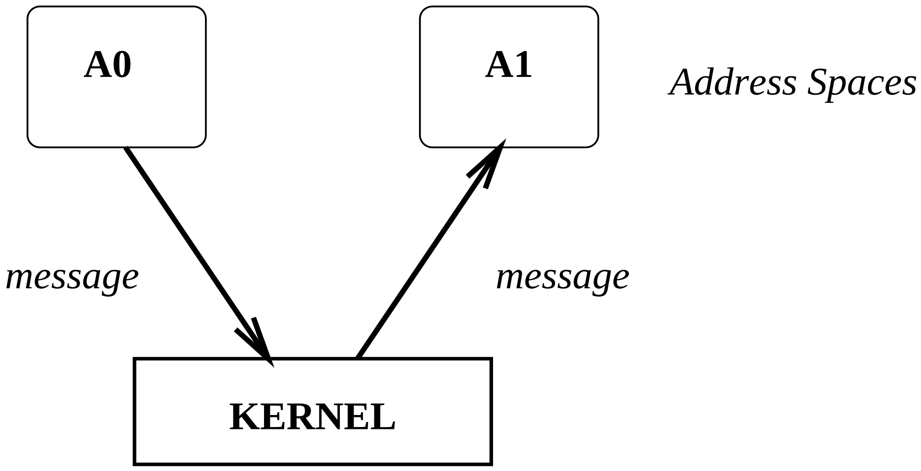

Figure 4: Message Passing Systems

In contrast to the fine-grained protection of capability systems, network based message passing systems rely on a coarse-grained protection mechanism. Communication facilities based on messages transparently permit both local and remote communication. Local communication takes place between two address spaces on the same machine, whereas remote communication takes place between address spaces on different machines connected via a communication network.

A message passing system enforces modularity and is suitable for distribution. However, programs have to be manually structured in a paradigm that is foreign to the control and data structuring mechanism of traditional "Algol-like" languages. Specifically, with message passing, data transfers have to be explicitly initiated whenever processes require communication. This gives programmers the opportunity to use application-specific knowledge to avoid unnecessary data transfers. However, message passing also requires that programmers use two entirely different mechanisms for access to memory: all local memory may be accessed using normal operations as done when accessing individual program variables, whereas all 'remote' memory must be accessed using message operations. Interestingly, recent research is beginning to address this dichotomy by providing a basic 'memory' abstraction for representation of both local and remote memory, and by addressing the potential performance penalties arising from providing this abstraction by 'weakening' the strong consistency requirements imposed on main memory [19, 145]. The resulting, weakened shared memory abstraction presented to programmers may be implemented efficiently because strong consistency and therefore, interprocessor communication is not required for all memory accesses. Other models of shared memory exploit programmer directives to reduce the cost of coherence maintenance [19], or they provide explicit primitives with which users can maintain application-specific notions of coherence of shared state [7] (see Section 3.2.2 for more information on this research).

One interesting lesson learned from current work on weakly consistent memories is that message passing and shared memory need not be different or mutually exclusive mechanisms. This may result in future operating systems that offer 'memory' abstractions or more strongly typed abstractions like 'shared queues' using object-like specification mechanisms. Such operating systems, therefore, might be structured as collections of cooperating objects, where object invocations may result in messages, in memory sharing, or in both, and where objects themselves may internally be structured as collections of cooperating objects or even as fragmented object [226, 57, 211, 159].

2.4 Language Based Mechanisms

Single language systems. CLU [148], Eden [134], Distributed Smalltalk [18], Emerald [125], and Linda [49] are a few examples of languages that integrate message based communication into the programming environment, either by defining all control structures in terms of messages, or by using messages as the basis for building "Algol-like" or entirely new control structures.

The advantages of a language based system are transparency and portability. References to local vs. remote objects are both handled transparently and automatically by the language's runtime system. In addition, the type system of the language can encourage optimizations that coalesce separate modules into one address space, while maintaining their logical separation. However, while such optimizations can alleviate certain performance problems with this approach, specific language primitives will inevitably impose performance penalties of which programmers must be aware in order to write efficient parallel programs. For example, the Ada rendezvous mechanism leads to well-known performance problems [43], the global addressing mechanisms and fixed language semantics of Linda can lead to inefficiencies concerning the update and access of remote information [211], and heap maintenance has been shown difficult for languages like Smalltalk [260, 186].

Another inherent problem with language based systems can be protection, where language-level typing mechanisms must be mapped to the protection mechanisms available in the underlying operating system [14, 254], which is not always easily done. In addition, any language based system will require all cooperating modules to be written in the same language, which precludes the use of mixed language environments. Furthermore, in order to guarantee the integrity of a system based on language-level decomposition, any executable code must be inspected by a trusted system entity to guarantee type-safety at runtime, essentially requiring access to its source code [25]. Finally, all language based systems will still require the availability of lower-level runtime system support for efficient program execution, as clearly apparent from current efforts to develop a threads-like common runtime system layer for both high performance Fortran and concurrent C++. It is precisely the structuring of such support that is the concern of this paper.

Remote procedure calls. Systems supporting Remote Procedure Calls (*RPC*) [30] occupy a middle ground between message based and single language systems. The use of RPC allows isolated components to be transparently integrated into a single logical system. In an RPC system, a procedure call interface hides the underlying communication mechanism that passes typed data objects among address spaces. Subsystems present themselves to one another in terms of interfaces implemented by servers. The absence of a single, uniform address space is compensated by automatic stub compilers and sophisticated runtime libraries [180, 110, 207] that transfer complex arguments in messages. RPC systems require that the data passed among cooperating modules be strongly typed; within a module, a programmer is free to mix languages, use weakly typed or untyped languages, violate typing if needed, and execute code for which source is not available [25].

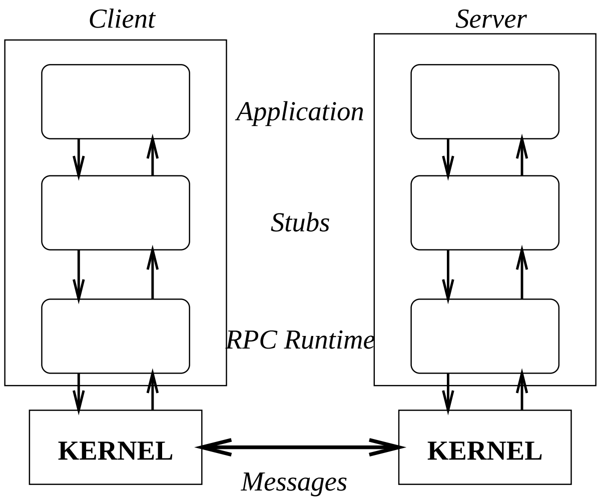

Figure 5: Remote Procedure Call

RPC is used for both local and remote communication between address space. An RPC between address spaces on different machines is often referred to as a *cross-machine RPC* whereas an RPC between address spaces on the same machine is referred to as a *cross-address space RPC*.

The principal components of an RPC system are *clients*, *servers*, and *messages*. A server is an address space which contains the code and data necessary to implement a set of procedures which are exported to other address space. A client is an address space which requests a service from a server by sending an appropriate message [180].

2.5 Object-Oriented and Object-Supporting Operating Systems

Several ongoing projects are using or exploring the *object-oriented* paradigm for building operating systems. Such systems may be broadly classified as object-oriented or object-supporting operating systems, depending on their internal structures and on the interfaces they provide to the user level [227, 198, 83].

Object-Oriented Operating Systems (*OOOS*). In an object-oriented operating system, an object encapsulates a system entity [156, 48]. An object-oriented language is primarily used to implement such an operating system [201]; the properties of the language such as data encapsulation, data abstraction, inheritance, polymorphism *etc.* are used to structure the system. An OOOS may or may not support objects at the user level. Examples of OOOSs are Choices [47] and Renaissance [202, 172].

Object-Supporting Operating Systems (*OSOS*). An object supporting operating system is not necessarily structured in an object-oriented fashion. However, it supports objects at the user level; the objects are typically language independent. Sample OSOSs are SOS [228], Cool [102], and CHAOS [213, 92, 91] for parallel machines, and Chorus [197] and Clouds [68] for distributed machines. OSOSs can further be classified into different groups depending on the kind of objects they support. In the *active-server* model of computation, objects are active entities containing threads of execution that service requests to the object [92, 91]. An OSOS supporting passive objects offers an *object-thread* model where a single thread of execution traverses all objects within an invocation chain [68, 128].

One use of object orientation in operating systems is to exploit type hierarchies to achieve operating system configuration (e.g., as done in Choices [47]) along with stronger notions of structuring than available in current systems. Another use of this technology is to use object based encapsulations of operating system services in order to represent operating system services internally in different ways, invisibly to services users. Examples of such uses are the internally parallel operating system servers offered in the Eden system [134] or in CHAOS [213, 92] and Presto [24, 23], the association of protection boundaries with certain objects as intended in Psyche [80], or the internally fragmented objects offered by Shapiro [227, 229, 98] for distributed systems, in 'Topologies' [211] for hypercube machines, and in 'Distributed Shared Abstractions' [57] for multiprocessor engines.

Unresolved issues with object-oriented operating systems include the efficient representation of object invocations, where it has become clear that 'not all invocations are equal', ranging from rapid unreliable invocations useful in real-time multiprocessor applications [213, 92] to reliable multicast invocations required for certain distributed programs [103]. In addition, as with remote procedure calls, it is unclear what levels of machine and language support are required for efficient implementation of object invocations (e.g., for parameter marshaling [103] or for crossing protection boundaries [258, 80]). However, object-oriented operating systems are likely to become increasingly important in part (1) because of the wide range of parallel architectures (or even sequential machines – ranging from digital pages to workstations) to be supported by future parallel operating systems and (2) because of the increasing importance of object-oriented languages like C++. It is likely that object-oriented operating systems will be constructed using the micro-kernel operating system structuring technique explained in Section 2.7.

2.6 Vertical and Horizontal Organizations

Some researchers have identified two broad classes of organization of operating system kernels referred to as horizontal and vertical organization [80]. These alternatives roughly correspond to the message-based and procedure-based organizations respectively [133].

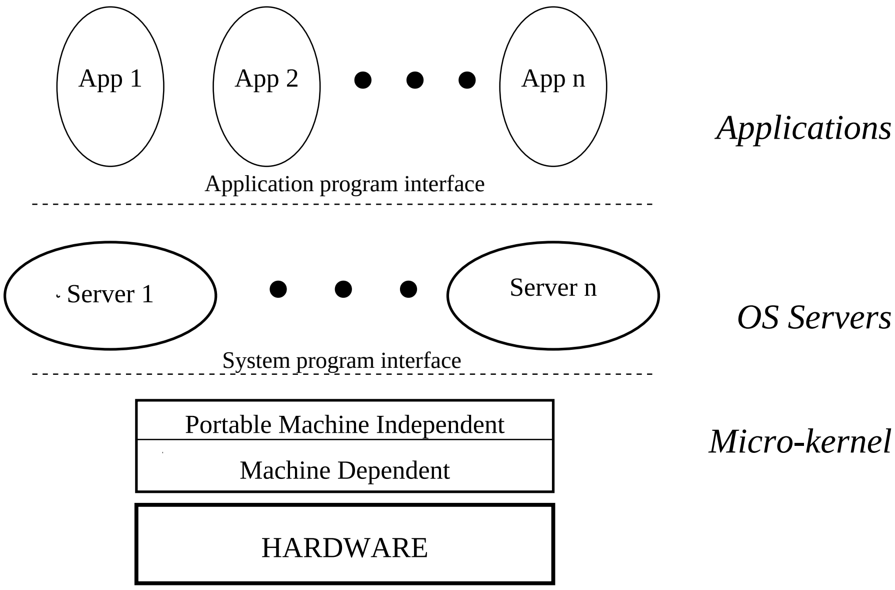

Figure 6: Micro-kernel based operating system

Vertical Organizations. In a vertical kernel, there is no fundamental distinction between a process in the user space and a process in the kernel. A user process enters the kernel via a trap when required, performs a kernel operation, and returns to user space. A kernel resource is represented by a data structure shared among processes. The vertical organization presents a uniform model for the user and the kernel level processes and closely mimics the hardware organization of a UMA multiprocessor. Most Unix [196] kernels are vertically organized.

Horizontal Organizations. In a horizontal kernel, each major kernel resource is represented by a separate kernel process (or thread), and a typical kernel operation requires communication among the set of kernel processes that represent the resources needed by the operation. The horizontal organization leads to a compartmentalization of the kernel in which all synchronization is subsumed by message passing. The horizontal organization closely mimics the hardware organization of a distributed memory multicomputer. Demos [16] and Minix [239] are examples of horizontal kernels.

2.7 Micro-kernel Based Operating Systems

Micro-kernel based operating systems are structured as a collection of system servers running on top of a minimal kernel (see Figure 6). The micro-kernel itself only implements the lowest-level (mostly hardware dependent) functions of an operating system. Such primitive functions include task and thread management, interprocess communication/synchronization, low-level memory management, and minimal device management (I/O). All higher level operating system services are implemented as user-level programs. Therefore, applications must use cross-address space RPC to interact with most operating system services. This implies that the performance of inter-process communication (IPC) mechanism plays a critical role in the performance of such operating systems [20].

The primary characteristic of micro-kernel based operating systems is modularity, thereby hoping to improve system extensibility, portability, reconfigurability, and improved support for distribution [251, 93, 92]. Improvements in distribution, extensibility, and reconfigurability [175] result from the separation of system components from each other, and from the use of message passing as the communication mechanism among them [251]. As a result, new services can be added (as new servers) or an existing server can be replaced by another without altering the existing components or the micro-kernel itself. Unlike large monolithic systems, such architectures also localize the hardware-dependent portions of the operating system inside the kernel, thereby potentially improving operating system portability. Furthermore, the use of common underlying services provides support for the coexistence and interoperability of multiple operating system environments on a single host as user-level programs [32]. Mach [32], Chorus [197], KeyKOS [42], QNX [106], and BirLiX [208] are a few examples of micro-kernel based operating systems.

2.8 Application-specific Operating Systems

Many application domains impose specific requirements on operating system functionality, performance, and structure. One blatant example of such requirements comes from the real-time domain, where application software and operating system support can be so intertwined that many systems may be better described as consisting of *operating software* – combined application software and operating system support functions – rather than as application software and an underlying operating system. This is because in contrast to other parallel or distributed application software, the control software of real-time systems cannot be termed *reliable* unless it exhibits two key attributes [209]: (1) computations must complete within well-defined timing constraints, and (2) programs must exhibit predictable behavior in the presence of uncertain operating environments [210, 27]. Operating software, then, must have direct access to the underlying resources typically controlled by the operating system, and for complex applications, it must deal with uncertainty in operating environments by even permitting programs or operating system components to *adapt* [215, 129] (i.e., change at runtime) in performance [209] and functionality during system execution [210, 213, 26, 96].

While many embedded real-time operating systems are offering functionality akin to multi-user systems, they do not impose any restrictions on resource use and reservation by application programs. For instance, in the CHAOS and CHAOS^{arc} operating systems [96, 213, 92], operating software implementing either application or operating system functionality consists of a number of autonomous *objects*, each providing a number of *operations* (entry points) that can be invoked by other objects. Such functionality appears no different from what is offered by other object-oriented operating systems. However, addressing the requirements of real-time programs, CHAOS^{arc} object invocations range from reliable invocations that maintain parameters and return information (or even communication 'streams' [213]) to invocations that implement unreliable 'control signals' or 'pulses' [213, 253]. Furthermore, invocation semantics can be varied by attachment of real-time attributes like delays and deadlines, where deadline semantics may vary from *guaranteed deadlines*, which are hard deadlines that must not be missed to *weak deadlines* [154], which specify that partial or incomplete results are acceptable when the deadline is missed. The resulting direct access to resources is a characteristic such real-time operating systems share only with certain single-user operating systems for parallel machines. On the other hand, system configurability is a property CHAOS^{arc} shares with many current high-performance operating systems, including the Synthesis kernel [163, 162], the Psyche system and its related research [219, 168], Presto [24, 23], and others [24, 23, 52].

Other examples of application-dependent operating system structures occur for database systems, where an operating system's I/O facilities and networking facilities may be determined or at least strongly affected by the primary application running on this system: a large-scale database performing transaction processing [231].

3 Design Issues

The basic functionality of a multiprocessor operating system must include most what is present in uniprocessor systems. However, complexities arise due to the additional functional capabilities in multiprocessor hardware and more importantly, due to the extreme requirements of performance imposed on the operating system. Namely, since one main reason for using parallel hardware is to improve the performance of large-scale application programs, an operating system or system constructs that perform poorly are simply not acceptable [11, 89]. Specific problems to be addressed concerning performance include protection in very large address spaces, deadlock prevention, exception handling for large-scale parallel programs, the efficient representation of asynchronous active entities like processes or threads, the provision of alternative communication schemes and synchronization mechanisms, and resource scheduling like process assignment to different processors and data placement in physically distributed memory [214], and finally, the parallelization of the operating system itself, again in order to provide scalable performance for varying application requirements [89, 109, 270].

A second major reason for using a multiprocessor system is to provide high reliability, and graceful degradation in the event of failure. Hence, several multiprocessor systems have been constructed and designed for improved fault tolerance. Such systems will not be discussed in this survey (see [28] for a survey of fault tolerant operating systems).

The following sections focus on design issues that concern operating system kernels or micro-kernels and therefore, the basic functionality that must be offered by multiprocessor operating systems, including processor management and scheduling, main memory management, and interprocess communication and synchronization.

3.1 Processor Management and Scheduling

The classic functions of an operating system include creation and management of active entities like processes. The effectiveness of parallel computing depends on the performance of the primitives offered by the system to express parallelism. If the cost of creating and managing parallelism is high, even a coarse-grained parallel program exhibits poor performance. Similarly, if the cost of creating and managing parallelism is low, even a fine-grained program can achieve excellent performance.

3.1.1 Heavyweight Processes to Lightweight Threads

One way to express parallelism is by using "Unix-like" processes sharing parts of their address spaces. Such a process consists of a single address space and a single thread of control. Kernels supporting such processes do not distinguish between a thread and its address space; they are sometimes referred to as *heavyweight* threads. The parallelism expressed using heavyweight threads is coarse-grained and is too inefficient for general purpose parallel programming for the following reasons:

- Since the kernel treats a thread and its address space as a single entity, threads and address space are created, scheduled, and destroyed together. As a result, the creation and deletion of heavyweight threads are expensive.
- Reallocating a processor to a different address space (context switch) is expensive. There is an initial scheduling cost to decide the address space to which the processor should be reallocated. Next, there is a cost for updating the virtual memory mapping registers and transferring the processor between address spaces. Finally, there is a long term cost associated with cache and TLB performance due to the address space change [169].

Hence, in many contemporary operating system kernels, address spaces and threads are decoupled, so that a single address space can have more than one execution threads. Such threads are referred to as *middleweight* threads or *kernel-level* threads when they are managed by the operating system kernel (POSIX Pthreads [185]). The advantages of middleweight threads are:

- The kernel can directly schedule an application's thread on the available physical processors.
- Kernel-level threads offer a general programming interface to the application.

Kernel-level threads also exhibit some problems that can make them impractical for use in fine-grained parallel programs [25]:

- The cost of generality of kernel-level threads is not acceptable to fine grained parallel applications. For example, saving and restoring floating point context in a context switch are expensive and may be unnecessary for a specific application program.
- A relatively costly protected kernel call is required to invoke any thread management operation, including thread synchronization. For example, many programs would rather have direct access to the hardware's synchronization operations.
- A single model represented by one style of kernel-level thread is unlikely to have an implementation that is efficient for all parallel programs.

To address the above problems with kernel-level threads, system researchers have turned to *user-level* threads, also known as *lightweight* threads. User-level threads are managed by runtime library routines linked into each application. A thread management operation does not require an expensive kernel call. Furthermore, lightweight threads enable an application program to use a thread management system, most appropriate to the problem domain. Mach Cthreads [60, 173, 212], the University of Washington threads [165, 9], SunOS LWP and threads [113, 127, 189], are a few popular lightweight thread implementations.

A lightweight thread generally executes in the context of a middleweight or a heavyweight thread. Specifically, the threads library schedules lightweight threads on top of middleweight or heavyweight threads, which in turn are scheduled by the kernel on the available physical processors. Such a *two-level scheduling* policy has some inherent problems:

- User level threads, typically, do not have any knowledge of kernel events (*e.g.,* processor preemption, I/O blocking and resuming, *etc.*). As a result, the application library cannot schedule a thread on a "just idle" processor.
- When the number of runnable kernel-level threads in a single address space is greater than the number of available processors, kernel-level threads must be multiplexed on the available processors. This implies that user-level threads built on top of kernel-level threads are actually scheduled by the kernel's thread scheduler, which has little or no knowledge of the application's scheduling requirements or current state [9].

Problems with multi-level scheduling arise from the lack of information flow between different scheduling levels. Anderson et al. in [9] attempt to solve these problems for two-level scheduling by:

1. Explicit vectoring of kernel events to the user level thread scheduler, using upcalls called *scheduler activations*[1], and by

[1]Scheduler activations are defined to be entities similar to kernel-level threads. One crucial distinction between scheduler activations and kernel level threads is that scheduler activations are not scheduled by the kernel. The kernel maintains the invariant that there are always exactly as many runnable scheduler activations as there are processors assigned to each address space.

2. notifying the kernel of user-level events affecting processor allocation.

Tucker and Gupta [249] propose a similar solution which dyanamically controls the number of processes used by applications. This scheme is discussed in details in Section 3.1.3. Similarly, Marsh et al. in [161] propose a set of kernel mechanisms (incorporated in the Psyche operating system) required to implement "first-class user-level" threads addressing the above problem. These mechanisms include shared kernel/user data structures (for asynchronous communication between the kernel and the user), software interrupts (for events that might require action on the part of a user-level scheduler), and a scheduler interface convention that facilitates interactions in user space between dissimilar kinds of threads (see section 4.5 for more on Psyche user-level threads).

A recent paper by Anderson et al. [10] also explores data structure alternatives when implementing user-level thread packages. Alternate implementations are evaluated in performance for thread run queues, idle processor queues, and for spinlock management. We are not aware of general solutions to the multi-level scheduling problem, other than the actual exchange or configuration of the operating system's threads scheduler by application programs, as often done in real-time systems [216].

3.1.2 Scheduler Structures

As with other operating system services for parallel machines, schedulers themselves must be structured to be scalable to different size target machines and to different application requirements. Mohan in his PhD thesis [170, 89] addresses this problem by designing a flexible run-queue structure, where scheduler run-queues can be configured such that any number of queues may be used by any number of processors. A similar approach to run-queue organization is taken in the Intel 432's iMAX operating system [63]. Recent work on real-time schedulers for parallel systems is also considering the effects of sharing alternative policy-level scheduling information on parallel scheduler performance [270]. Beyond this work, scheduler structuring remains largely unexplored, but should receive increased attention in operating systems for large-scale parallel machines like the Intel Paragon multiprocessor.

3.1.3 Scheduling Policies

A scheduling policy allocates available time and processors to a job or a process statically or dynamically [153]. Processor load balancing[2] is considered to be a part of a scheduling policy [232]. Basic theoretical results on static process scheduling on parallel machines show that the scheduling problem is NP-hard; static algorithms minimizing average response time include those described in [164] and [39]. Other scheduling algorithms appear in [267] and [141]. In this section, we focus on dynamic scheduling [164], and on scheduling for shared memory machines, where variations in distances between different processors on the parallel machine [39, 214] are not considered.

Static and Dynamic Scheduling: A static scheduler makes a one time decision per job regarding the number of processors to be allocated. Once decided, the job is guaranteed to have exactly that number of processors whenever it is active. A static scheduler offers low runtime scheduling overhead [78], but it also assumes a stable parallel application. This is a reasonable assumption for many large-scale scientific applications in which parallelism is derived by decomposition of regular data domains [204]. Recent work, however, is focussing more on dynamic scheduling for two reasons: (1) because most complex large-scale parallel applications exhibit irregular data domains or changes in domain decompositions over time,

[2] Processor load balancing concerns the dynamic distribution of processing loads among the processors of the parallel machine.

so that a static processor allocation rapidly becomes inefficient and (2) large-scale parallel machines are often used in multi-user mode, so that scheduling must take into account the requirements of multiple parallel applications sharing a single machine [158, 157, 225, 65].

J. Zahorjan and C. McCann compare the performance of static and dynamic schedulers for multi-user workloads. Their results include [267]:

- Independent of workload and overall system load, dynamic scheduling performs best when context switch overheads are small.
- The advantage of dynamic scheduling at low context switch costs increases with larger and more rapid changes in the parallelism exhibited by a workload.
- Dynamic scheduling performs increasingly well relative to the static counterpart as system load increases.
- In terms of average response time, dynamic scheduling dominates static scheduling for almost all overhead (context switch) values.

The dynamic policy occasionally exhibits a performance penalty when overhead values are very large. One reason for such performance degradation is a possible high rate of processor reallocation. Hence, some researchers have suggested to dampen the rate of processor allocation and release, thereby reducing the rate of "useless processors exchange" [267]. However, such a modification to the dynamic policy was found to be detrimental to performance [267].

As for uniprocessors, multiprocessor schedulers can be classified as preemptive or nonpreemptive schedulers. A scheduler can also be classified according to its scheduling granularity, which is determined by the executable unit being scheduled (For example, schedulers differ in that they may schedule individual or groups of processes). A few well accepted multiprocessor scheduling policies are reviewed next [141].

Single Shared Ready Queue: Research addressing UMA multiprocessors has typically assumed the use of a single ready queue shared by all processors. With this queue, policies like First Come First Served (FCFS) or Shortest Job First (SJF) are easily implemented, and have been evaluated in the literature. More interesting to us are schedulers and scheduling policies directly addressing the primary requirement of a parallel program: if performance improvements are to be attained by use of parallelism, then the program's processes must be scheduled to execute in parallel.

Coscheduling: The goal of coscheduling (or gang scheduling) is to achieve a high degree of simultaneous execution of processes belonging to a single job. This is particularly useful for a parallel application with cooperating processes that communicate frequently. A coscheduling policy schedules the runnable processes of a job to run simultaneously on different processors. Job preemption implies the simultaneous preemption of all of its processes. Effectively, the system context switches between jobs.

Ousterhout proposed and evaluated three different coscheduling algorithms in his PhD thesis [183]: matrix, continuous and undivided. In the matrix algorithm, processes of arriving jobs are arranged in a matrix with p columns and a certain number of rows, where p is the total number of processors in the system. The arrangement of jobs is such that all the processes in a job reside in a same row. The scheduling algorithm uses a round-robin mechanism to multiplex the system between different rows of the matrix, so that all the processes in a row are coscheduled.

A problem with the matrix algorithm is that a hole in the matrix may result in a processor being idle even though there are runnable processes. The continuous algorithm addresses this problem by arranging all processes in a linear sequence of activity slots[3]. The algorithm considers a window of p consecutive positions in the sequence at a particular moment. When a new job arrives, the window is checked to see if there are enough empty slots to satisfy its requirements. If not, the window is moved one or more positions to the right, until the leftmost activity slot in the window is empty but the slot just outside the window to the left is full. This process is repeated until a suitable window position is found to contain the entire job or the end of the linear sequence is reached. Scheduling consists of moving the window to the right at the beginning of each time slice until the leftmost process in the window is the leftmost process of a job that was not coscheduled in the previous time slice.

The most serious problem with the continuous algorithm is analogous to *external fragmentation* in a segmentation system. A new job may be split into fragments, which can result in unfair scheduling for a large, split jobs vs. small contiguous jobs. Ousterhout addresses this issue by designing an undivided algorithm, which is identical to the continuous algorithm except that all of the processes of each new job are required to be contiguous in the linear activity sequence. Leutenegger & Vernon [141] slightly modify the undivided algorithm to eliminate some of its performance problems: when a job arrives, its processes are appended to the end of a linked list of processes. Scheduling is done by moving a window of length equal to the number of processors over the linked list. Each process in the window receives one quantum of service on a processor. At the end of the quantum, the window is moved down the linked list until the first slot of the window is over the first process of a job that was not completely coscheduled in the previous quantum. When a process within the window is not runnable, the window is extended by one process and the non-runnable process is not scheduled. All processors that switch processes at the end of a quantum do so at the same time. A second algorithm modification improves expected performance for correlated workloads. This modification is in the movement of the window. At the end of each quanta, the window is only moved to the first process of the next job, even if the job was coscheduled in the previous time slice.

Round Robin (RR) Scheduling: Two versions of RR scheduling exist for multiprocessors. The first version is a straightforward extension of the uniprocessor round robin scheduling policy. On arrival of a job, its processes are appended to the end of the shared process queue. A round robin scheduling policy is then invoked on the process queue. The second version uses jobs rather than processes as the scheduling unit. The shared process queue is replaced by a shared job queue. Each entry of the job queue contains a queue holding its processes. Scheduling is done round robin on the jobs. The job in the front of the queue receives p quanta of size q, where p is the number of processors in the system and q is the quantum size. If a job has fewer processes than p, then the total quanta size, which is equal to pq, is divided equally among the processes. If the number of processes in a job exceeds p, then there are two choices. The first choice is same as the previous case, *i.e.,* divide the total quanta size pq equally among all processes. The other choice is to choose p processes from the job in a round robin fashion, each process executing for one quanta. The first alternative has more scheduling overhead than the second one.

In [249], the authors observe that the performance of an application worsens considerably when the number of processes in the system exceeds the total number of processors. They attribute this decreased performance to several problems:

- A process may be preempted while inside a spinlock-controlled critical section, while the other processes of the same application "busy wait" to enter the critical section. This

[3] In [183], Ousterhout defines an *activity slot* as a slot to which an activity (or a process) may be assigned. Scheduling consists of selecting one activity slot in each processor; the assigned activity is executed by the processor. The collection of activity slots is referred to as the *activity space*. The three algorithms, described here, assume that no job contains more than p (number of processors) processes.

problem is particularly acute for fine-grain parallel programs. Identical problems arise when programs' processes are engaged in producer/consumer relationships.

- Frequent context switches occur when the number of processes greatly exceeds the number of processors.
- When a processor is interleaved between multiple address space, cache misses can be a major source of performance degradation [245].

Careful application design and coscheduling may handle the problems associated with spinlock-controlled critical sections and those with producer-consumer processes, but they do not address performance degradation due to cache corruption or frequent context switches. A more direct solution is proposed by Zahorjan et al. [265], who describe a thread scheduler that avoids preempting processes inside critical sections. In contrast, Edler et al. [79] propose an approach combining coscheduling and preemption avoidance for critical sections. Multiple processes are combined to form a group. The scheduling policy of a group can be set so that either all processes in the group are scheduled and preempted simultaneously, or individual processes are scheduled and preempted normally, or processes in the group are never preempted. An individual process may choose to override its group scheduling policy. This policy is flexible, but it leave specific solutions to the critical section problem to user code.

The problems of cache corruption and context switch frequency are addressed by Lazowska and Squillante [135], who evaluate the performance of several multiprocessor scheduling policies based on the notion of processor affinity. A process's processor affinity is based on the contents of the processor's cache. The basic policy schedules a process on a processor on which it last executed hoping that a large percentage of its working set is still present in the processor's cache. Since the policy inherently discourages process migration, it may lead to severe load imbalance. The authors of [135] address this issue by proposing a variation on the basic policy which successfully reduces the cache corruption problem.

Similarly, affinity (for local memory) also plays a vital role in scheduling processes in a NUMA machine; the context of a process resides mostly near the processor on which the process executed last.

Vaswani and Zahorjan [250] also study the effect of cache affinity on kernel processor scheduling discipline for multiprogrammed, shared memory multiprocessors and conclude that the cache effects due to processor reallocation can be significant. However, their experiments demonstrate that affinity scheduling has negligible effect on performance for current multiprocessors. Finally, they conclude that even on future, faster machines, a scheduling policy based on dynamic reallocation of processors among jobs outperforms a more static, equipartition policy.

Tucker and Gupta [249] propose a different solution for reducing the frequency of context switches and for reducing cache corruption, which is explained next.

Dynamic Partitioning: The dynamic partitioning [187, 73] (also known as *Process control* with processor partitioning) policy proposed by Tucker and Gupta [249] has the goal of minimizing context switches, so that less time is spent rebuilding a processor's cache. Their approach is based on the hypothesis that an application performs best when the number of runnable processes is the same as the number of processors. As a result, each job is dynamically allocated an equal fraction of the total number of processors, but no job is allocated more processors than it has runnable processes. Each application program periodically polls a scheduling server to determine the number of processes it should ideally run. If the ideal number is less than the actual number, the process suspends some of its processes, if possible. If the ideal number is greater than the actual number, a process wakes up a previously suspended process. This policy has limited generality since it requires interactions between user processes and the operating system scheduler, and since it requires that user programs are written such that their processes can be suspended and woken up during program execution.

Hand-off Scheduling: A kernel level scheduler that accepts user hints is described in [36]. Two kinds of hints exist:

- Discouragement hints: A discouragement hint is used to discourage the scheduler to run the current thread. A discouragement hint may be either *mild*, *strong*, or *weak*. David Black in [36] discusses a scheduler that accepts discouragement hints.
- Hand-off hints: A hand-off hint is used to suggest the scheduler to run a specific thread. Using a hand-off hint, the current thread hands off the processor to another thread without intermediate scheduler interference. Such schedulers are better known as *hand-off* schedulers.

two experiments with scheduling hints are described in [36], and it is shown that scheduling hints can be used to improve program performance. Hand-off scheduling has been shown to perform better when program synchronization is exploited (e.g., the requester thread hands off the processor to the holder of the lock) and when interprocess communication takes place (e.g., the sender hands the processor off to the receiver).

In [101], Gupta et al. use a detailed simulation study to evaluate the performance of several scheduling strategies. These include regular priority scheduling, coscheduling or gang scheduling, process control with processor partitioning, hand-off scheduling, and affinity based scheduling [248]. In addition, tradeoffs between the use of busy waiting and blocking synchronization primitives are explored, in conjunction with their interactions with different scheduling strategies. A key focus of the study is the impact of different scheduling strategies on the caching behavior of an application. Results demonstrate that in situations where the number of processes exceeds the number of processors, regular priority-based scheduling in conjunction with busy-waiting synchronization primitives results in extremely poor processor utilization. In such cases, the use of blocking synchronization primitives improves performance significantly. Process control and gang scheduling strategies are shown to offer the highest performance, and their performance is relatively independent of the synchronization methods used. However, for applications that have sizable working sets that fit into the cache, process control performs better than gang scheduling. For the applications considered[4], the performance gains due to hand-off scheduling and processor affinity are shown to be small.

In [266] the authors study the effects of two environmental factors, multiprogramming and data-dependent execution times, on spinning overhead of parallel applications, and how the choice of scheduling discipline can be used to reduce the amount of spinning in each case. Specifically, they conclude that decisions about how to allocate processors to jobs and how to schedule the threads of a job on its processors must be made cooperatively. Furthermore, for the range of workloads and systems considered, the difference in mean performance between synchronization via spinning vs. blocking is found to be very little. They also compare the static scheduling policy to both fixed and variable self-scheduling policies when the number of independent tasks exceeds the number of processors available and conclude that the variable self-scheduling policy provides good overall performance and is the most robust with respect to overhead costs.

Ongoing research also addresses the development of schedulers for specific application domains or for specific target machines. One such area is real time systems. In a real time

[4]The applications considered are a particle-based simulator called MP3D, LU-decomposition on dense matrices, a parallel implementation of Goldberg and Tarjan's Maxflow algorithm [94], and a highly optimized block-based parallel algorithm for multiplying two matrices.

system, a scheduling policy must satisfy timing constraints such as deadlines, earliest start times, *etc.* of an incoming job. Before a job is assigned one or more physical processors, the scheduler checks whether the system can satisfy the job's timing constraints. This analysis is known as *schedulability analysis*. Schedulability analysis and scheduling for real time systems [53, 269, 37, 108, 150, 71, 268] are active areas of research and are not within the scope of this paper.

3.2 Memory Management

Memory management for UMA multiprocessors is conceptually similar to that for multiprogrammed uniprocessors. As mentioned earlier, in an UMA architecture, memory access times are equal for all processors. However, the underlying architecture typically supports some degree of parallelism in global memory access. As a result, even for UMA machines, operating system writers must exploit the available hardware parallelism when implementing efficient memory management. More interesting problems arise for NUMA and NORMA machines.

For early research on memory management in parallel machines, including the implementation of physically distributed, internally parallel memory managers, the reader is referred to [119, 120, 89], which present the innovative designs and structures of memory or object managers for the Cmmp and Cm* multiprocessor systems. Recent research has focussed primarily on page management, namely, on memory management in the context of virtual memory systems implemented for parallel machines.

3.2.1 Shared Virtual Memory

A typical virtual memory system manages a memory hierarchy consisting of a cache, uniformly accessible primary memory, and significantly slower secondary memory. Unlike the traditional approach of a three level store, the Accent operating system [84], supports *a single level store*, in which primary memory acts as a cache of secondary storage. Filesystem data and runtime allocated storage are both implemented as disk-based data objects. Copies of large messages are managed using *shadow paging* techniques. Other contemporary systems like the IBM System 38 and Apollo Aegis also use the single level store approach, but limit its application to the management of files.

A central feature of Accent is the integration of virtual memory and communication. Large amounts data can be transmitted between processes with high performance using memory mapping techniques. As a result, client and server processes can exchange potentially large data objects like files without concern for the traditional data copying cost of message passing. In effect, Accent carries into the domain of message passing systems the notion that I/O can be performed through virtual memory management.

The design of the Mach system's memory management is largely derived from the Accent system [194, 1, 263, 262]. Its single level store is attractive because it can simplify the construction of an application program by allowing programmers to map a file into the address space of a process, and because it can improve program performance by permitting file data to be read directly into physical memory pages rather than into intermediate buffers managed by the operating system. Furthermore, because physical memory is used to cache secondary storage, repeated references to the same data can often be made without corresponding disk transfers. The design and implementation of the Mach memory management is discussed in detail in Section 4.3 below.

Memory management services and implementations are generally dependent on the operating system[5] as well as the underlying machine architecture. Some of the current research

[5]For example, in real-time executives, memory management services are simple and primitive. General purpose operating systems such as Unix allow protected address spaces to co-exist on the machine hardware. Some distributed systems support distributed memory schemes.

has focussed on designing memory management functionalities and interfaces which are independent of the machine architecture and the operating system kernel. For example, the Mach operating system implements virtual memory management which is machine and operating system independent [194]. The machine-dependent portion of Mach's virtual memory subsystem is implemented as a separate module. All information important to the management of the virtual memory is maintained in machine-independent data structures and machine-dependent data structures contain only the mappings necessary to run the current mix of programs [194] (See Section 4.3 for more on the implementation of the Mach virtual memory management). Similarly, in [2], the authors present the design and the implementation of a scalable, kernel-independent, Generic Memory Management Interface (GMI) (for the Chorus [197] nucleus) which is suitable for various architectures (e.g. paged and/or segmented) and implementation schemes.

Some operating systems [84] allow applications to specify the protection level (inaccessible, read-only, read-write) of pages, and allow user programs to handle protection violations. In [12], the authors survey several user-level algorithms that make use of page-protection techniques, and analyze their common characteristics, in an attempt to identify the virtual-memory primitives the operating system should provide to user processes. The survey also benchmarks a number of systems to analyze a few operating systems' support for user-level page-protection techniques.

3.2.2 NUMA and NORMA Memory Management

Most early NUMA multiprocessor systems did not offer virtual memory support. However, recent NUMA or NORMA parallel machines like the Thinking Machines CM-5, the Kendall Square KSR, and the Intel Paragon, and all UMA operating systems routinely offer virtual memory.

A NUMA multiprocessor organization leads to memory management design choices that differ markedly from those that are common in systems designed for uniprocessors or UMA multiprocessors. Specifically, NUMA machines like the BBN Butterfly do not support cache or main memory consistency on different processors' memory modules. Such consistency is guaranteed only for local memory and caches (i.e., for non-shared memory), or it must be explicitly enforced for shared memory by user- or compiler-generated code performing explicit block or page moves [194, 262]. As a result, NUMA architectures implementing a shared memory programming model typically expose the existing memory access hierarchy to the application program, as done in BBN's Uniform System [247]. Motivations for exposing such information include:

1. Giving programmers the ability to minimize relatively expensive remote vs. less expensive local memory references (i.e., maximize program locality [120]), and
2. permitting programmers to avoid several forms of potential contention (switch or memory contention) caused by a large number of remote memory references [261].

Recent parallel NUMA architectures like the Kendall Square multiprocessor offer consistent global virtual memory. However, the performance reasons for exposing programmers to the underlying machine's NUMA properties persist, leading the system designers to include hardware instructions for page prefetches and poststores [112].

Research in memory management for parallel machines has focussed on designing techniques for NUMA multiprocessors that relieve programmers from the responsibility of explicit code and data placement. Realizing that the problem of memory management is similar to the problem of cache management and consistency for UMA multiprocessors, the Mach operating system's UMA implementation of memory management [194, 262] attempts to minimize the amount of data-copying and replication by using page *copy-on-write* and similar techniques for reduction of data movement. In essence, the operating system is exploiting the fact that the sharing of "read-only" data on a multiprocessor does not require the allocation of multiple private copies; different processes can remotely access a single copy located with the writing process. On NUMA machines, however, extra data movement in terms of page replication and migration may result in improved performance, due to decreased page access times for locally stored pages and due to the elimination of possible switch or memory contention for access to shared pages, as demonstrated by specific measurements on the BBN Butterfly multiprocessor reported in several sections below, including Section 4.5.

We briefly discuss several memory management algorithms for NUMA multiprocessors. Various multiprocessor operating systems such as Mach, Psyche, and PLATINUM use a variation or a mix of these algorithms.

The algorithms described below are categorized by whether they migrate and/or replicate data [234, 271]. One algorithm migrates data to the site where it is accessed in an attempt to exploit locality in data accesses and decreases the number of remote accesses. Two other algorithms replicate data so that multiple read accesses can happen at the same time using local accesses.

Migration algorithm: In the migration algorithm, the data is always migrated to the local memory of the processor which accesses it. If an application exhibits a high locality of reference, the cost of data migration is amortized over multiple accesses. Such an algorithm may cause thrashing of pages between local memory.

Read-replication algorithm: One disadvantage of the migration algorithm is that only the threads on one processor can access the data efficiently. An access from a second processor may cause another migration. Replication reduces the average cost of read operations, since it allows read to be simultaneously executed locally at multiple processors. However, a few write operations become more expensive, since a replica may have to be invalidated or updated to maintain consistency. If the ratio of reads over writes is large, the extra expense of the write operation may be offset. Replication can be naturally added to the migration algorithm for better performance.

Full-replication algorithm: Full replication allows data blocks to be replicated even while being written to. Keeping the data copies consistent is a major concern in this algorithm. A number of algorithms are available for this purpose. One of them is similar to the write-update algorithm for cache consistency. A few such algorithms are discussed in [234], and some specific NUMA memory management schemes are described for the individual parallel operating systems in Section 4.

The PLATINUM[6] operating system kernel, designed to be a platform for research on memory management systems for NUMA machines, implements and evaluates [61] a *coherent memory* abstraction on top of non-uniform access physical memory architectures. Since coherent memory is uniformly accessible from all processors in the system, it makes programming NUMA multiprocessors easier for users. PLATINUM's goal is to explore the possibility of achieving performance comparable to that of hand-tuned programs with a simple, easy-to-program shared-memory model. PLATINUM's implementation of coherent memory replicates and migrates data to the processors using it, thus creating the appearance that memory is uniformly and rapidly accessible. The protocol for controlling the data movement is derived by extending a directory-based cache coherency algorithm using selective invalidation [13].

[6] PLATINUM is an acronym for "Platform for Investigating Non-Uniform Memory"

Page Placement: Other than the Kendall Square Research Corporation's KSR machines [195] and the experimental Dash multiprocessors [90], NUMA multiprocessors do not have broadcast, invalidate, or snooping mechanisms that maintain consistency among multiple copies of a page when writes occur. Hence, programmers or operating systems restrict writable pages to a single copy. This gives rise to the page placement problem, which concerns the decision as to which local memory should contain the single page copy. Black et al. [35] refer to this problem as the *migration problem*. A similar problem is observed for read-only pages. Specifically, the *replication problem* is concerned with determining the set of local memories that should contain copies of a page. Here, the assumption is made that the set of local memories that contain copies of the page is monotonically non-decreasing.

In [40], the authors present the implementation of a page placement mechanism to automatically assign pages of virtual memory to appropriately located physical memory in the Mach operating system on the IBM ACE multiprocessor workstation. By managing locality in the operating system, the implementation hides the details of specific memory architectures, thus making programs more portable. The simple strategy for page replacement uses local memory as a cache over global, managing consistency with a directory-based ownership protocol similar to that used by Li [145] for distributed shared virtual memory. Their experience [40] indicates that even simple automatic strategies can produce nearly optimal page replacement. It also suggests that the dominant remaining source of avoidable performance degradation is *false sharing*[7] [41], which can be reduced by improving language processors or by tuning applications.

The DUnX kernel [124], developed as a framework for implementing dynamic page replacement policies, introduces a highly tunable parameterized dynamic page placement policy for NUMA multiprocessors addressing issues related to the tuning of that policy to suit different architectures and applications. The policy supports both migration and replication, uses a directory based invalidation scheme to ensure the coherence of replicated pages and uses a freeze/defrost[8] strategy to control page bouncing. Such a parameterized NUMA memory management policy can be tuned for architectural as well as application differences. In [124], the authors perform several experiments with the parameterized page replacement policy to confirm that dynamic placement policies are efficient, therefore a reasonably simple parameterized policy may form the basis for the development of machine-independent memory management subsystems for NUMA machines.

In [136], the authors present the implementation of the memory management system in the Psyche [220] multiprocessor operating system. The Psyche memory management system is structured into four layers of abstraction – NUMA, UMA, VUMA (virtual memory) and PUMA (Psyche memory). These abstractions are shown in Figure 7 and are discussed in Section 4.5.

Weak Memory: The Accent and Mach operating systems [84, 194, 262] for uniprocessor and UMA multiprocessors demonstrated the use of copy-on-write paging for message passing. Li at Yale showed that a modified Apollo Aegis kernel can support shared memory on a 10Mhz token ring [144], and similar results are demonstrated by the Clouds project at Georgia Tech [68]. Such work is motivated by the fact that parallel programming may be simplified when the underlying system provides a basic 'memory' abstraction for representation of both local and shared state. Unfortunately, performance penalties can result from the fact that the underlying system must implement its shared memory abstraction independent of how applications use it. For example, if the system keeps copies of shared data *coherent* or *consistent* by invalidation on writes, it will invalidate existing copies even when the processors having these copies will not access them in the future.

Current research is exploring how shared memory may be represented on parallel or dis-

[7] An object that is not writably shared, but that is on a writably shared page is falsely shared [40].

[8] Excessive page movement is controlled by freezing the page in place and forcing remote accesses [61]

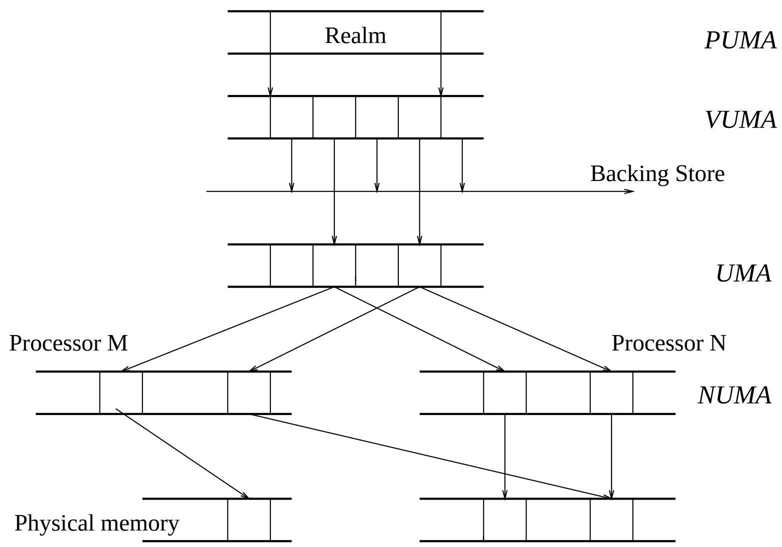

Figure 7: Psyche Memory Management Layers

tributed machines such that its performance can approximate that of message passing systems. For example, some memory models exploit the fact that synchronization is used to control access to shared state (e.g., properly labeled memory [90] or data race-free memory [4, 5]). This allows the underlying system to weaken memory consistency requirements. The resulting, weakened shared memory abstraction presented to programmers may be implemented efficiently because strong consistency and therefore, interprocessor communication is not required for all memory accesses. Other models of shared memory developed for distributed architectures exploit programmer directives to reduce the cost of coherence maintenance [50], or they provide explicit primitives with which users can maintain application-specific notions of coherence of shared state [7].

Mechanisms for memory or state sharing have significant effects on the performance of parallel and distributed applications. This is demonstrated by recent designs of and experimentation with alternative memory models for large scale multiprocessors that exhibit NUMA memory characteristics due to their use of caches to reduce communication latencies [90, 236]. In addition, in distributed memory machines like the Intel iPSC series, efficient state sharing is necessitated by significant differences in access times to local vs. remote information [211]. This is leading to redesigns of the Mach system paging mechanisms and page servers for its implementation on the Intel Paragon multiprocessor, and it is resulting in more general research on shared state or objects in distributed or shared memory parallel machine [211, 57]. Since differences in remote to local access times are even more pronounced in distributed machines like sets of workstations connected via high-speed networks [55], notions of distributed objects have been a topic of research for such systems for quite some time, as evidenced by work on Clouds [69] at Georgia Tech, on Chorus [197], and on fragmented objects [227] in France.

3.3 Synchronization

When multiple cooperating processes execute simultaneously, synchronization primitives are needed for concurrency control. When multiple processes share an address space, synchronization is required for shared memory consistency [99]. Two fundamental properties are enforced by synchronization: (1) mutual exclusion (to protect a critical section) and (2) event ordering. Classical synchronization primitives like *semaphores* [72], *monitors* [107, 131] *etc.* are widely discussed in the earlier literature and are therefore, not described here. Also not discussed are more complex synchronization mechanisms like path expressions and serializers [38], in part because such mechanisms are not in widespread use. Instead, this section briefly reviews some common and efficient synchronization constructs supported by recent multiprocessor operating systems.

3.3.1 Locks

A lock is a shared data structure used to enforce mutual exclusion. A critical section is normally protected by a lock. Although there are exceptions (*e.g.,* read-write locks), a lock is generally held by only one process at a time. A process holding a lock is called the lock owner. To enter a critical section, a process first atomically gains ownership of the associated lock (called *locking*). A contender process for the lock (or for the critical section protected by the lock) waits by either spinning or blocking until the lock is released by its current owner. When a process exits a critical section, it atomically releases lock ownership (called *unlocking*).

Multiprocessor operating systems typically support multiple types of locks, some of which are reviewed below.

Spin and Blocking Locks: Spin locks are the most primitive locks. When a lock is busy, a waiting process spins (*busy-waits*) until the lock is released. Most hardware supports spin locks by specific instructions in their instruction sets. Although spin-waiting consumes processor, bus, and memory cycles, early research in multiprocessor operating systems clearly demonstrates the performance advantages of simple locking strategies and lock implementations [258, 89, 120], showing that spin locks are useful in two situations – when the critical section is small (compared to the cost of blocking and resuming a process) or when no other work is available for the processor (since spin waiting results in minimum latency between lock release and reacquisition) [182].

When using a blocking lock, a waiting process (also called a contender process) blocks until awakened by the process releasing the lock. Such locks are also known as *mutex locks.*

Anderson et al. [10] compare the performance of a number of software spin-waiting algorithms. They also propose a few efficient spin-waiting algorithms such as *Ethernet style backoff* algorithm (introducing delay between successive spins analogous to Ethernet's backoff or Aloha), *software queueing* of spinning processors, and others. Results are demonstrated on a Sequent UMA machine, and it is not apparent how their results generalize to NUMA multiprocessors.

Synchronization for NUMA machines is addressed in [168] by Mellor-Crummey et al., who survey some spin lock algorithms and propose a new scalable algorithm (a list-based queuing lock, also known as *MCS* lock) that generates O(1) remote references per lock acquisition, independent of the number of processors attempting to acquire the lock.

The new generation of hardwares provide a few powerful atomic operations such as test-and-set and compare-and-swap which can simplify implementations of synchronization primitives and can even allow certain concurrent data structures to be implemented without blocking [104, 105]. Moreover, instructions such as fetch-and-add [97] allow certain common operations to be performed in parallel without critical sections [126].

Other work has evaluated the effects of other kernel components [8, 9, 10, 99] as well as applications on synchronization. For example, Zahorjan, Lazowska and Eager [265] first examines the extent to which multiprogramming and data-dependencies in an application complicate an user's decision to spin or block, then evaluate [266] how the overhead of spinning is affected by various scheduling policies.

Read-Write Locks: A read-write lock allows either multiple readers or a single writer to enter a critical section at the same time. The waiting processes may either spin or block depending on whether the lock is implemented as a *spinning read-write* lock or a *blocking read-write* lock.

Configurable Locks: In [176], Mukherjee and Schwan study the effects of application and hardware characteristics on multiprocessor locks, and propose a structure for configurable locks. Such locks allow applications to dynamically alter the waiting (spin, block or both) mechanism and the request handling mechanism (how the lock is scheduled). Their experiments with configurable locks demonstrate that combined locks (locks that both spin and block while waiting) improve application performance considerably compared to simple 'spin' or 'blocking' locks. Furthermore, hints from lock owners may be used to configure a lock for improving its waiting strategy ('advisory' or 'speculative' locks). In addition, an adaptation policy used to configure adaptive multiprocessor locks [177] is shown to improve application performance. such a lock detects changes in application characteristics and adapt itself to suit such changes.

In [126], the authors study seven strategies (including a few *competitive* strategies) for determining whether and how long to spin before blocking while waiting for a lock. The study concludes that for competitive strategies, performance is no worse than an optimal off-line strategy by some constant factor. Measurements indicate that the standard blocking strategy performs poorly compared to mixed strategies. Among the mixed strategies studied, adaptive algorithms perform better than non-adaptive ones.

A few object oriented-operating systems such as Choices [47] and Renaissance [202] take an object-oriented approach to lock configuration/customization. These systems define a few basic classes which provide simple and crude locks (implemented using hardware provided instructions). More sophisticated locks are implemented either on top of these existing classes, or by customizing the existing locks (see Section 4.8 and 4.9).

Barrier locks: A barrier lock implements a barrier in a parallel program. Once a process reaches a barrier, it is allowed to proceed if and only if all other cooperating processes reach the barrier. A waiting process may either spin or block depending on the implementation of the lock.

In [168], Mellor-Crummey et al. survey some barrier algorithms and propose a new scalable algorithm (a *tree-based* barrier) that spins on locally-accessible flag variables only, requires only $O(p)$ space for p processors, performs the theoretical minimum number of network transactions $(2p-2)$ on machines without broadcast, and performs $O(logp)$ network transactions on its critical path. Such a barrier lock implementation is reminiscent of lock implementations used in distributed memory machines, called *structured locks*.

Structured Locks: In distributed memory machines like hypercube or mesh multiprocessors, operating system constructs (e.g., I/O, exception handling, multicast communications, etc. [119, 139]) are physically distributed in order to offer efficient access to the global operating system functionalities required by application programs. Synchronization is no exception because it is a computation that must be performed globally for many physically distributed processes and processors. As a result, synchronization must be performed using explicit communication structures like rings or spanning trees, which touch upon all members of the group of processes being synchronized. In essence, a lock in a distributed memory machine is a fragmented and distributed abstraction shared among several independently executable processes, where the importance of this particular abstractions is demonstrated by explicit support in hardware in several parallel machines, including the Intel Paragon and the Thinking Machines CM line of machines. However, in contrast to the OS support for UMA and NUMA multiprocessors, synchronization abstractions for distributed memory machines can often be optimized substantially if they can be made *programmable* by application programmers or if synchronization can be combined with other communications being performed in application programs [211, 87].

3.3.2 Other Synchronization Constructs

Condition Variables. Condition variables make it possible for a thread to suspend its execution while awaiting an action by some other thread. A condition variable is associated with some shared variables protected by a mutex and a predicate (based on the shared variables). A process acquires the mutex and evaluates the predicate. If the predicate is not satisfied, the process waits on the condition variable. Such a wait atomically releases the mutex and suspends execution of the process. After a process changes the shared variables so that the predicate may be satisfied, it may signal a waiting thread. The signal allows blocked threads to resume action, to re-acquire the mutex, and to re-evaluate the predicate to determine whether to proceed or wait.

Events. Events are mainly used to control thread orderings. A process may wait on an event; it blocks until the event occurs. Upon event occurrence, a signal wakes up one or all waiting processes. Events come in different flavors. A state (*happened* or *not happened*) may or may not be associated with an event. A count may be associated with an event, which enables a process to wait for a particular occurrence of an event. More complicated event structures have been shown useful for several application domains and target machines, most prominently including the event handling facilities for active messages [151, 211] or the synchronization points designed by Gheith for real-time applications [92].

In [29], Birrell et al. present an informal description, implementations and formal specifications of various thread synchronization primitives (such as *Acquire* and *Release*, *Condition variables*, semaphores) supported by the Taos operating system.

3.4 Interprocess Communication

3.4.1 Basic Communication Primitives

Cooperating processes or threads in a multiprocessor environment often communicate and synchronize. Execution of one process can affect another by communication. Interprocess communication employs one of two schemes: shared variables or message passing

As mentioned earlier, when two processes communicate using shared memory, synchronization is required to guarantee memory consistency. The last section describes a few popular synchronization primitives. This section focuses on inter-process communication without using explicit shared variables.

In a shared memory multiprocessor, message passing primitives between disjoint address space may be implemented using global memory. Exchange of messages is a more abstract form of communication than accessing shared memory locations. Message passing subsumes communication, buffering, and synchronization.

Multiprocessor operating systems have experimented with a large variety of different communication abstractions, including *ports* [84, 262] *mailboxes* [119, 63], *links* [230, 127] *etc.* From an implementation point of view, such abstractions are kernel-handled message buffers. They may be either *unidirectional* or *bidirectional*. A process may send to or may receive messages from them. There may be *rights* (like send, receive, or ownership rights) associated with these entities. Different operating systems define different semantics on these abstractions.

The two basic communication primitives in all such abstractions are *send* and *receive*, which can again come in many different flavors. Sends and receives may be *blocking* (a process invoking a primitive blocks until the operation is complete), or they may be *non-blocking* (a process does not wait for the communication to be complete), or they may be *conditional* vs. *unconditional*. Communication between processes using these primitives may be *synchronous* or *asynchronous*, *etc.*

Many issues must be considered when designing an inter-process communication mechanism; they are reviewed in numerous surveys of distributed operating systems [241, 239, 240] and are not discussed in detail here. Issues include (1) whether the underlying hardware supports reliable or unreliable communication, (2) whether send and receives are blocking or non-blocking, (3) whether messages are typed or untyped and of variable or fixed length, (4) how message queues can be kept short, (5) how to handle queue overflows, (6) how to support message priority (it may be necessary that some messages are handled at higher priorities than others), (7) how to transmit names, (8) protection issues, and (9) how kernel and user programs must interact to result in efficient message transfers. Communication issues specific to hypercube or mesh machines are reviewed elsewhere [222, 221, 211, 67]. Examples of recent research in communication protocols for high performance or parallel machines are addressing the association of computational activities with messages [66, 151], the user-driven configuration of communication protocols for improved performance [110], and the parallelization of protocol processing [147].

Of interest to this survey is that parallel programs typically use both message mechanisms and shared memory (when available) for inter-process communication. This is demonstrated by implementations of message systems like PVM on the KSR supercomputer, and by implementations of message systems for the BBN Butterfly NUMA machine. Interestingly, comparisons of message passing with direct use of shared memory often result in inconclusive results, in part because such results strongly depend on the sizes, granularities, and frequencies of communications in parallel programs [138].

3.4.2 Remote Procedure Calls

Most recent shared-memory multiprocessor operating systems support *cross-address space remote procedure calls* [25] (*RPC*) as a means of inter-process communication. RPC is a higher level abstraction than message passing. It hides the message communication layer beneath a procedure call layer. RPC allows efficient and secure communications. Furthermore, cross-address space RPCs can be made to look identical to cross-machine RPCs, except that messages do not go out over the network and in most cases, only one operating system kernel is involved in RPC processing[9]. Otherwise, the same basic paradigm for control and data transfer is used. Messages are sent by way of the kernel between independent threads bound to different address space. The use and performance of cross-address space RPC is discussed extensively in [25].

In [21], Bershad et al. propose a new kernel-based communication facility called *Lightweight Remote Procedure Call* (*LRPC*) which is designed and optimized for communication between address space on the same machine. LRPC combines the control transfer and communication model of capability systems with the programming semantics and large-grained protection

[9] It is possible to have more than one kernel in a NUMA machine.

model of RPC. In [22], Bershad et al. propose another interprocess communication scheme, called *User-level Remote Procedure Call* (*URPC*). URPC decouples processor allocation from data transfer and thread management by combining fast cross-address space communication protocol using shared-memory with lightweight threads managed entirely at the user level. By decoupling, the operating system kernel can be entirely bypassed for cross-address space communications.

3.4.3 Object Invocations on Shared and Distributed Memory Machines

Recent research on object-oriented operating systems for parallel machines is concerned with a generalized form of remote procedure calls used for inter-object communications, also called *object invocations*. Specific results include the provision of mechanisms for implementation of alternative ways to invoke an object, as offered by the Spring operating system's *subcontract* mechanism [103] or the CHAOS system's *policy* abstraction [92]. The basic need for such alternatives is again derived from performance or reliability considerations, where applications would like to have the ability to vary the performance or even the semantics of object invocation (e.g., reliable vs. unreliable invocations) separately from the target objects being invoked or the precise parameters being passed. This is the ability provided by subcontracts in Spring, by attributes and policies in CHAOS, and by invocation attributes associated with individual accesses to fragmented objects built for hypercube machines in [211] and for multiprocessors in [57]. Sample variations in invocation semantics not concerned with the types of objects or operations being invoked include asynchronous vs. synchronous invocations, the ability to wait for acknowledgements of invocation receipt [211] as in distributed RPC implementations [180], the variation of when an invocation is considered complete (upon successful transmission of the invocation, upon receipt of the invocation, upon invocation completion, etc.), and the association of additional parameters governing how and when invocations are scheduled or processed, the latter being particularly important in real-time systems but also relevant to parallel applications where some invocations are more important than others.

4 Sample Multiprocessor Operating System Kernels

This section reviews several specific multiprocessor operating system kernels, including Hydra, StarOS, Mach (developed at Carnegie Mellon University), Elmwood and Psyche (developed at the University of Rochester), Presto (developed at the University of Washington), KTK (developed at the Georgia Institute of Technology), Choices (developed at the University of Illinois, Urbana-Champaign), Renaissance (developed at Purdue University), Mach/RP3 (developed at T. J. Watson Research Center, IBM), and a few successful commercial systems like Dynix (developed for Sequent Multiprocessors), Chrysalis (developed for BBN Butterfly machines), UMAX (developed for Encore Multimax multiprocessors).

4.1 HYDRA

HYDRA [258, 256, 257, 259, 142, 58] is one of the earliest successful multiprocessor kernels, developed at Carnegie-Mellon University and implemented on the *C.mmp* hardware. Extensive descriptions of the HYDRA system appear in [120] and [258].

The major goals of HYDRA are:

1. to develop a minimal kernel for multiprocessor systems from which an arbitrary set of operating system facilities and policies can be easily constructed,

2. to allow the arbitrary number of systems created from the kernel to co-exist simultaneously, and
3. to provide a uniform protection mechanism.

HYDRA rejects strict hierarchical layering in structuring the operating system, instead originating the concept of separation of mechanism from policy. Namely, the mechanisms in the kernel are intended to support the abstract notion of resources, or of policies controlling these resources.

The salient features of HYDRA are a generalized notion of *resources*, the definition of *execution domain*, and its *capability-based protection mechanism* controlling access to resources within a domain. Capabilities may be used to protect any user- or system-level entity. Capabilities are references to objects with sets of access rights; they are manipulated only by the kernel.

4.1.1 Execution Environment and Processes

Programs in HYDRA consists of three types of objects: *procedures*, *LNSs* and *processes*. A procedure object is an abstraction of "Algol-like" procedures consisting of code and data that accept and return parameters. Each procedure object contains a list of capabilities, which specify the rights for the objects it may reference during execution. These capabilities therefore, define the procedure's execution environment. A procedure object also contains some capability templates only partially specified at the time of procedure creation, which are filled in for each execution of the procedure and then contain capabilities of the parameter objects supplied by the caller. *Templates* are the formal parameter specifications for the actual parameters expected by the procedure; they are used for type checking and for checking access rights.

An LNS is a dynamic entity which defines the execution environment for an invocation. It consists of a single list of capabilities composed of a combination of caller-independent capabilities (listed in the procedure object) and caller-dependent capabilities (actual parameters). A unique LNS appears with each invocation and disappears after the procedure terminates. As a result, the execution domain changes each time a procedure is entered or exited. When an object is passed as a parameter, a new capability is created for the object in the new LNS which consists of a reference to the object and the right list specified by the template in the callee. Hence, a callee has more freedom to operate on a parameter object than a caller.

A process, in HYDRA, is defined as the unit of asynchronous processing. It is treated as "the smallest entity which can be independently scheduled" by an external agent. Internally, a process is a stack of LNSs representing the state of a single sequential task.

HYDRA implements interprocess communication and synchronization by providing elementary message buffering primitives, spin locks, and Dijkstra-style semaphore operations. In addition, several alternative implementations of synchronization constructs are offered, ranging from simple spin locks, to kernel-provided semaphores, to user-level 'policy' semaphores.

The HYDRA call mechanism permits a process to call a protected procedure, via the kernel. The kernel then checks the actual parameter capabilities supplied by the caller. If they meet the protection requirements, the kernel creates a new LNS defining the new environment. Control is then passed to the code of the procedure. Similarly, the procedure, after completion, returns through the kernel, whereupon the kernel deletes the callee's LNS and restores that of the caller. Thus, the kernel is entered twice for each protected procedure call.

4.1.2 Objects and Protection

As mentioned earlier, one of the major goals of HYDRA is to provide a uniform protection mechanism. HYDRA does not provide security levels (i.e., specific protection policies). Instead, it simply provides a flexible protection mechanism with which different types of security (policies) can be implemented. An *object* is the unit of protection. It consists of a unique name, a type, and a representation. The type of an object contains the unique name of a "distinguished object", which serves as the representative of the object's class. Since objects can be referenced by multiple capabilities, a reference count is maintained, and an object is deleted when that count becomes zero.

The representation of an object consists of a capability part and a data part. The data part of an object can be accessed with appropriate access rights, whereas the capability part is manipulated only by the kernel.

Each capability contains a rights list which details the operations that may be performed on an object referenced by the capability. A right list consists of *type independent rights* (kernel rights) and *type dependent rights* (auxiliary rights). Kernel rights define the operations which the kernel provides for controlled manipulation of objects and capabilities.

4.1.3 HYDRA and Parallel Computing

While the major contributions of HYDRA address protection issues, there are also many research results relevant to parallel systems, including:

- basic results establishing the concept of program locality, the effects of bus and memory contentions on parallel program performance, and the importance of asynchrony in parallel program design and implementation [258, 17, 181],
- demonstrations that high performance parallel programs require choices in the operating system mechanisms being provided (e.g., a variety of different synchronization mechanisms), since programs differ in granularities of parallelism, in frequencies of access to shared data, etc., and
- experiences with the use of specific operating system facilities by parallel programs, which then led to differences in design decisions for the HYDRA follow-up at CMU, the StarOS system reviewed next. One such insight was that parallel programmers seek performance and therefore, will prefer using simple, fast mechanisms over easy-to-use, slower mechanisms. This insight has affected most modern operating system designs for parallel machines in areas ranging from process to thread representations, communication systems designs, and the implementation of synchronization constructs [120, 258].

4.2 StarOS

StarOS [117, 116, 118, 119] is an experimental operating system for the *Cm** [121, 89, 77, 88, 238] multi-microprocessor computer developed at Carnegie-Mellon University. The design of StarOS is influenced by the protection mechanisms of Hydra, the underlying Cm* architecture, and its principal goals of achieving high performance and reliability for parallel machine users. Cm* is a Non Uniform Memory Access (NUMA) machine. It consists of a number of *computer modules*, each of which consists of a processor (a DEC LSI-11), a bus (an LSI-11 bus), local memory, a local switch (called the *Slocal*), and attached devices. The Slocal switch routes memory references either to a module's local memory or to the Map Bus, and it also handles references to its local memory originating from non-local processors. Non-local references are performed via a *Map Bus*, which connects up to fourteen computer modules into a *cluster*. The computer modules of a cluster share a switch, called the *Kmap*, which handles intra- and inter-cluster memory references. Clusters are connected via *intercluster buses* connecting multiple Kmap switches. The Kmaps and the Slocals together comprise a *hierarchical, distributed switch*. A Cm* configuration may have an arbitrary number of connected clusters. A cluster need not have a direct intercluster bus connection to another cluster in a configuration.

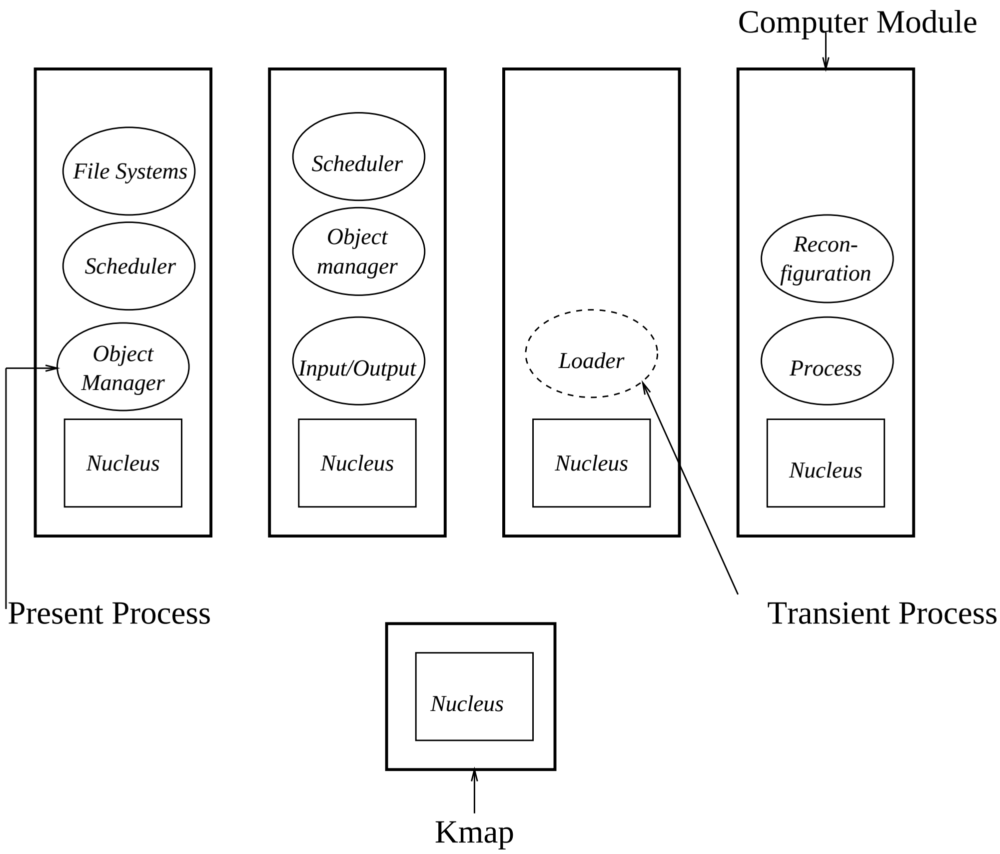

Figure 8: The StarOS Executive Task Force

Objects, capabilities and invocations. Like HYDRA, StarOS is an object-oriented system enforcing strong object typing, where the type of an object determines the set of functions defined on it. Furthermore, as with HYDRA, a process must possess a *capability* for an object in order to invoke it. A capability names a distinct and unique object, and it specifies object access *rights* (much slimmed down compared to HYDRA).

An object consists of two disjoint segments – the *data portion* (containing a sequence of data words) and the *capability list portion* (containing a sequence of capabilities). Similarly, each process has two name spaces – the *capability name space* and the *immediate address space*. The data portion of an object can be directly *mapped* to a window in the immediate address space of a process. Thus, the machine instructions provided by the underlying processor may directly be used to manipulate the data portion of an object. Such a mechanism allows an application to use small objects without additional cost compared to standard processors.

A process may directly invoke all objects for which it has capabilities.

StarOS allows users to create new objects and new object types dynamically. Existing or default object types are called *representation types*. Users may create new *abstract types* using these representation types.

One of the attributes of StarOS of interest to parallel systems is its definition of *modules*, *functions*, and *module invocations*, which are the blueprint for the implementation of similar functionality in the Intel 432's iMAX operating system [63] and in modern object-oriented operating systems like Eden [134], Choices [47], and CHAOS [213, 92]. A module defines an object by exporting a set of invocable functions. A function invocation by a process is performed asynchronously by passing invocation parameters to a process designated to execute the function. Invocation parameters are passed by passing a capability for a small object (called a *carrier*), which contains the parameters. Another important concept originated with StarOS is that function (or object) invokers will require substantial flexibility in how synchronization is performed in conjunction with object invocation. As a result, StarOS simply offers a low-level mechanism for implementing different synchronization schemes, which is a capability for a *mailbox* contained in the carrier. This mailbox will not only contain the return parameters of the invocation, but it can also be used for waiting on the invocation's completion. Furthermore, an invocation need not be routed to some fixed number of processes serving it (called *present* processes); it can also result in the creation of a new process (called a *transient* process) which executes the invoked function and then terminates. Present processes typically offer lower latency for function invocation since they can be pre-initialized for execution of specific functions. Such pre-initialized processes block on a common *invocation mailbox* waiting for an incoming carrier which represents a function request. Therefore, a second important innovation in parallel operating systems originating with StarOS is that since invocations and functions differ in functionality, so should the entities executing function code, as is the case with kernel- vs. user-level threads in current multiprocessor operating systems.

A third concept in parallel operating systems originating with both StarOS and the Roscoe operating system [230] is the structuring of operating systems and even operating system kernels as micro-kernels (called a *nucleus* in StarOS). Specifically, a small subset of the StarOS functions, called *instructions*, are defined to execute sequentially and synchronously with the invoking function. Collectively they are referred to as the nucleus. The nucleus is partly implemented in firmware (in the Kmap) and partly in software. A copy of the nucleus software runs on each computer module in its own address space, as its micro-kernel. All other operating system functionality resides in user-level, as present or transient processes, depending on its frequency of use.

4.2.1 Task Forces

A *task force* is the abstraction offered by StarOS for representation of parallel programs. A task force, programmed with the TASK language [117] (or with lower-level library support), is simply a collection of cooperating StarOS processes which collectively accomplish some joint task. Compared to a single process performing such a task, a task force can be programmed to take advantage of parallelism to achieve enhanced reliability (by triplication of selected services) and performance (by use of parallelism to implement services). In addition, a task force can be programmed to adapt to changes in user requirements or in the underlying hardware by growing or shrinking dynamically.

StarOS distinguishes an *executable task force* from a *static task force*. An executable task force is the set of processes with supporting code and data objects that performs the desired task. A static task force is a collection of modules, each exporting a set of functions, together with some input data. Any task force is first constructed in its static form, and then initialized to contain some number of processes comprising the executable task force.

Realizing that parallel programs are not simply collections of unrelated, multiple processes or threads, StarOS also supports two kinds of relationships between processes in an executable task force. First, the *dependence* relationship reflects the relationships between invoking and invoked processes. It can be used for process suspension and abnormal termination, and it defines forests of processes rooted at some number of reconfiguration processes (one of which is provided by default by the operating system). Second, the *bailout* relationship associates a mailbox with each process at the time of process creation; it is used to report process failures. Again, a simple bailout process is provided by default by the operating system. Modern, threads-based parallel operating systems only provide some of the StarOS functionality (i.e., the dependence relationship maintained for thread joining and forking), in part because the process containing its threads can play the role of the entity to which threads may 'bail out'.

Naturally, the StarOS operating system itself is also represented as a task force (see Figure 8), where user-level transient or present processes implement higher-level operating system functions, whereas each processor contains a micro-kernel (the nucleus) implementing the synchronous instructions defined by the system implementors.

4.2.2 Synchronization and Communication

In response to the underlying NUMA hardware, StarOS is a message based parallel operating system. A process never blocks as a side effect of a message send or receive. Instead, a process explicitly suspends its execution when it wishes to wait for the completion of actions taken by other processes. Such process interactions via messages are supported by StarOS *mailbox* objects. A mailbox is created to buffer data messages (single data words) or capability messages (single capabilities), but not both. The basic functions defined on a mailbox are *Send* and *Receive*, but these functions do not imply synchronization actions in conjunction with message communications. Such actions may be explicitly programmed using an *event* mechanism which permits processes to block on a mailbox when waiting for the occurrence of a specific event.

The mailboxes and the events are used to implement multiple send and receive semantics. Specifically, a receive may be invoked either in *conditional* or *registration* mode. If the mailbox is empty, a conditional receive returns with an error indication and without a message. In registration mode, the name of the invoker together with an event are placed into a queue (called a *registration queue*) associated with the mailbox. The event is defined beforehand by the process by specifying an event name, a capability for the mailbox, and a location within its address space where the message is to be stored. The process may then choose to block on that event. A send delivers a message directly to the registered receivers. The message is stored in the location associated with the event, and the event is signalled. If the registration queue is empty and the mailbox is not full, a send buffers a message in the mailbox. A send fails if the mailbox is full. Communication mechanisms similar to those of StarOS can be found in the iMAX operating system [63].

4.2.3 Scheduling

Responsibility for scheduling in StarOS is divided between *scheduler* processes and the *multiplexors*. A multiplexor is a low level mechanism that makes short term decisions about which process to execute next on its processor, whereas a scheduler implements some specific scheduling policy. Multiplexors use a priority-ordered set of mailboxes as run queues, where with each processor may be associated single or multiple run-queue mailboxes. These sets of run queues may overlap arbitrarily. A processor's multiplexor searches for the next process to run according to the priority order; the selected process is assigned the processor for some maximum time quantum. If the sum total of execution time of a process exceeds a scheduler-determined value, the multiplexor sends the process to its designated scheduler. There may be multiple schedulers in the system, but StarOS does not provide any specific support for interactions or possible conflicts among different schedulers, as done in distributed systems using bidding methods.

Few interesting higher-level schedulers were constructed for the StarOS system. Instead, the reader is referred to the re-engineered version of StarOS built by Ousterhout (called the Medusa system [184]), where several interesting, higher-level scheduling strategies were implemented and evaluated, typically referred to as 'co-scheduling' or 'gang scheduling' in the literature [182].

4.2.4 Reconfiguration

System initialization is performed by the *reconfiguration module*, which gathers data about the physical resources (*e.g.,* physical memory), creates the first operating system objects, initializes the Kmap, and initializes the nuclei. Other system initialization functions include determination of the number of replicated processes to be created for each function, the placement of code for a function in physical memory, the initial assignment of operating system processes to run queues *etc.*

The configuration of StarOS may change with time, as determined by the system's reconfiguration processes, which periodically examine the environment and adjust the StarOS configuration accordingly, to improve system performance or maintain some desired reliability level. For example, if a particular function is invoked frequently, more processes are created to execute that function. Moreover, the reconfiguration module dynamically configures the system to handle hardware and software faults. For example, if the physical environment changes (*e.g.,* addition or removal of clusters), StarOS can be expanded or reduced to accommodate such changes.

4.3 Mach

Mach [3, 191, 242, 243, 95, 192, 15] is a multiprocessor operating system kernel developed at Carnegie-Mellon University first for distributed systems, then for tightly-coupled UMA multiprocessors. Later extensions of Mach also address NUMA and NORMA machines [255]. Mach runs on a wide variety of uniprocessor and multiprocessor architectures, including the DEC VAX system, Sun 3 workstations, IBM PCs, the IBM RP3 multiprocessor, the Encore Multimax, and recently, the Intel Paragon. Mach is also supported as a product by a number of hardware vendors. Mach is the base technology for the OSF/1 operating system from the Open Software Foundation.

Mach[10] separates the Unix process abstraction into *tasks* and *threads* [190]. In addition, Mach provides the following:

- Machine independent virtual memory management [194].
- A capability based interprocess communication facility.
- Language support for RPC [76, 122, 123].
- Support for remote file accesses between autonomous systems.

[10] Mach is binary compatible with Berkley's Unix 4.3 bsd release

- Lightweight user-level threads known as Mach Cthreads [60, 173].
- Miscellaneous other support like debuggers for multithreaded applications [51], exception handling [33] *etc.*

Structurally, Mach is organized horizontally (developed using micro-kernel technology). The Mach kernel is a minimal, extensible kernel which provides a small set of primitive functions. It provides a base upon which complete system environments may be built. The actual system running on any particular machine is implemented by *servers* on the top of the kernel. The Mach kernel supports the following basic abstractions:

1. *Task:* A task, an execution environment for threads, is a collection of system resources including an address space.
2. *Thread:* A thread is defined as the basic unit of execution. Mach threads belong to the middle-weight process class (defined in Section 3.1.1).
3. *Port:* A port is a communication channel and is similar to an object reference in an object-oriented system. Any object other than messages may be represented by a port. An operation on an object is performed by sending a message to the corresponding port.
4. *Message:* A message is a typed collection of data objects used for communication between active threads.
5. *Memory Object:* A memory object is a repository of data which can be mapped into the address space of a task.

4.3.1 Memory Management

The Mach virtual memory management [84, 194, 1, 262, 263] system is designed to be architecture and operating system independent. Architecture independence is achieved by dividing the virtual memory implementation into machine independent and machine dependent portions. The machine independent portion has full knowledge of all the virtual memory related information, whereas the machine dependent portion manages the “pmaps”, which are hardware defined physical address maps (*e.g.,* VAX page tables). The machine dependent portion only contains the mappings essential to run the current programs. All mapping information for any page in the system can be reconstructed from the information in the machine independent portion at fault time. Due to this separation between machine dependent and independent portions, the page sizes in both portions may not be the same – the machine independent page size is a boot time parameter and must be a power of two of the machine dependent size.

The main data structures in the Mach virtual memory system are: a *resident page table*, which keeps track of information about machine independent pages, an *address map* which is a per-task doubly linked list of map entries, each of which maps a range of addresses to a region of a memory object, a *memory object* which is the unit of backing storage, and the *Pmaps* which maintain the machine dependent memory mapping information. Using these data structures, Mach supports large, sparse virtual address spaces and memory mapped files [1, 194, 244].

Mach implements a single level store by treating all primary memory as a cache for virtual memory objects. Mach allows tasks to allocate and deallocate regions of virtual memory, and to set the protection and inheritance of virtual memory regions. Inheritance may be specified as *shared*, *copy* or *none* on a per page basis. A page specified as *shared* is shared for reads and writes by both parent and child tasks, whereas a page specified as *copy* is effectively copied into the child's address map. However, for efficiency, a *copy-on-write* technique is used. A page specified as *none* is not passed to the child. The *copy-on-write* sharing between unrelated tasks is also employed for passing large messages between them. The virtual memory system exploits *lazy evaluation*, such as *Copy-on-write* and *Map-on-reference* whenever possible.

Another important feature of the Mach virtual memory system is its ability to handle page faults and to page out data requests at the user level [263]. A few basic paging services are provided inside the kernel. However, a *pager* may be specified and implemented outside the kernel at user level. This has become particularly important for real-time implementations of Mach and for implementations addressing the needs of specific parallel architectures.

The Mach kernel does not have specific support for distributed shared memory. However, DSM can be implemented using a server on top of the kernel [85].

4.3.2 Interprocess Communication

A port is a kernel-protected entity that is the basic transport abstraction in Mach. Messages are sent to and received from a port. A task gets access to a port by receiving a port capability with send or receive rights. Mach 3.0 also supports *send-once* rights for ports; these are useful for implementing RPC. It also provides *dead names* and *dead-name notifications*, which allow servers and clients to detect each others' terminations [75].

Messages are variable size collections of typed data. Mach supports both synchronous and asynchronous message transfers. The copy-on-write technique is employed for large message transfers. The ports and the messages together provide location independence, security, and data type tagging [75, 76, 122, 123].

Mach 3.0 supports *port sets* to let a few threads serve requests for multiple objects. A receive operation on a port set returns the next message sent to any of the member ports. A no-sender detection mechanism allows object servers to garbage collect the receive right and the represented object. In [75], the author discusses the design and the implementation of an IPC interface for Mach 3.0.

The Mach kernel implements messages only within a single machine. However, the transparency of Mach interprocess communication (IPC) allows a user-level server (network message servers) to extend the IPC across a network [100].

On top of the general message primitives, Mach implements various flavors of communication including server-client remote procedure calls, distributed object-oriented programming, and streams.

4.3.3 Scheduling

The Mach scheduler [36, 34, 35] consists of two parts, one responsible for processor allocation, the other responsible for scheduling threads on individual processors.

Processor Allocation. A user-level server performs processor allocation, using the mechanisms provided by the underlying Mach kernel. The processor allocation facility adds two new objects to the Mach kernel – the *processor* and the *processor set*. Much like the iMAX kernel's facilities [63], a processor object manipulates the physical processors, whereas a processor set is an independent entity to which processors and threads can be assigned. An application creates a processor set, and uses it as the basis of communication with the server. A server performs processor allocation by assigning processors to the processor sets provided by the clients. Thus, clients have the power to use and manage processors without having direct control over them. A server satisfies a client's requests in strict order in a greedy fashion [36, 34].

Thread Scheduling. Mach uses a priority based time-sharing scheduling technique within each processor set. Mach schedules individual threads without using any knowledge about the relationships among threads. The scheduler maintains a global run queue shared by the processors and a local run queue for each processor. Each run queue is a priority queue of runnable threads. Priority calculation for threads is discussed in detail in [36, 35]. Mach is "self scheduling" – a processor consults the run queue when it needs a thread to run. As mentioned in Section 3.1.3, the Mach scheduler accepts two kinds of user hints – discouragement hints and hand-off hints.

4.3.4 The Mach 3.0 Micro-kernel

The Mach 3.0 micro-kernel has evolved from Mach 2.5 by eliminating all the compatibility code for BSD Unix from the kernel. Major changes include the optimization of its IPC implementation (by optimizing ports and port rights) as well as the use of new algorithms, the use of continuations[11] [74, 75, 146] in scheduling, IPC, exception, and new page fault handling facilities [32].

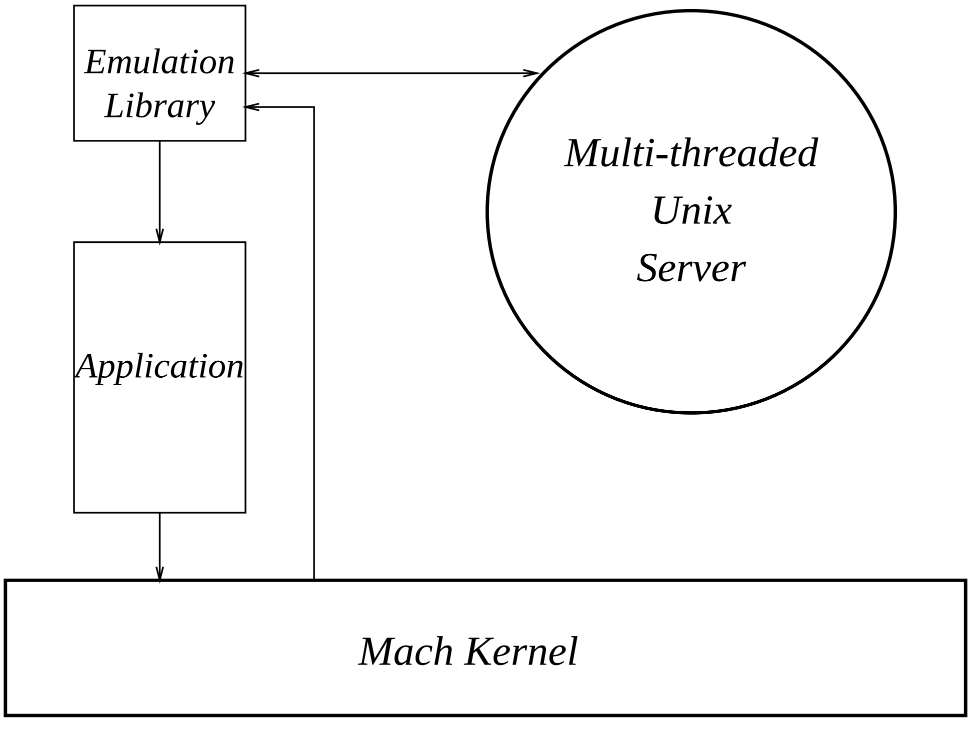

Figure 9: Unix Server on Mach Kernel

The basic facilities provided by the Mach kernel support the implementation of operating systems as Mach applications [32]. Figure 9 shows the organization of an application, the Unix Server and its relationship to the Mach kernel. The Unix server is implemented as a Mach task with multiple threads of control managed by the Mach Cthreads package. The

[11]A *continuation* is the address of a routine to call when a thread continues execution, plus a small data structure that contains local state needed by that routine

emulation library functions both as a translator for system service requests and as a cache for their results [32].

4.4 Elmwood

Elmwood [167, 137] is an object-oriented multiprocessor operating system designed and implemented at the University of Rochester. The major design goals of Elmwood are:

- to provide orthogonality of its mechanisms,
- to support significant protection domains,
- to support multiple users, and
- to structure the operating system in an object-oriented fashion with a small kernel.

In Elmwood, *objects* encapsulate abstractions within protection domains, *logical object names* (*LON*) provide access to protection domains, *processes* represent asynchrony, *semaphores* and *condition variables* provide synchronization, and *exceptions* report error conditions.

Elmwood consists of a series of layers implemented on the BBN Butterfly hardware. The first software layer is the program development layer. The next layer contains mechanisms to support concurrent programming. Subsequent layers constitute the Elmwood kernel, which implements process-like threads, physical memory allocation, objects, interprocess communication, and RPC.

4.4.1 Objects and LONS

An Elmwood object, which is a passive entity consisting of code and data, represents an instance of an abstract data type. An object is the basic unit of encapsulation, abstraction and protection. Each object exports a set of *entry procedures* to manipulate its local data. Elmwood supports the creation of long-lived processes that continue to perform object-related duties for the duration of the object's existence.

A LON is similar to a *capability* of an object. To invoke an entry procedure, a caller must have the appropriate LON. A LON refers to a unique object and a user-interpreted *context value*, which provides a mechanism to implement various security policies. LONs map many-to-one to objects. The different LONs for an object are the handles for different context values, enabling the object to distinguish clients. An object can create and invalidate a LON with a desired context value. Coarse-grain protection for access to objects is implemented by the kernel using LONs and fine-grain protection for access to operations within an object is implemented by the object using context values.

A LON may be passed as a parameter in an RPC. Once the creator of a LON passes it to another object, it loses control over the distribution of the LON. However, the object retains control over the interpretation of the context values. Only the object that creates a LON invalidates it. The kernel maintains the association between the LON used to invoke an object and the corresponding context value.

Since an object may contain active processes at deletion time, Elmwood supports two distinct object deletion mechanisms. The first mechanism deletes the object after all the activity within the object ceases whereas the second mechanism allows an object to be deleted immediately if and only if it is referenced using its canonical LON (The LON returned in object creation time).

4.4.2 Processes and Synchronization

The primitives for process management and synchronization in Elmwood are implemented as object operations implemented in the kernel. Processes do not have protected address spaces. A process executes the code associated with an object. Such orthogonality of processes and objects allow any number of processes executing simultaneously inside an object (much like is done in the Clouds operating system [68]). Any necessary synchronization must be provided by the object's implementation using synchronization primitives like semaphores and condition variables, both of which are implemented by Elmwood system objects. Elmwood supports multiprogramming, either time-slicing or multiprocessing transparently.

4.4.3 Interprocess Communication

Remote procedure call (RPC) is the only mechanism for communication between processes. Processes communicate using a common object as an intermediary. To avoid the need for compiler support, Elmwood requires an object programmer to provide a dispatcher for an object's entries, including initialization.

4.5 Psyche

Psyche [217, 218, 219, 220, 80] is a general-purpose operating system for large-scale shared-memory multiprocessors developed at the University of Rochester. Psyche is implemented on the BBN Butterfly Plus hardware.

The major design goals of the Psyche project are to support multi-model parallel computing [160] and to provide user-level flexibility in general. The intent is to allow applications or application components that use the machine in different ways to coexist and to interact productively.

Psyche provides a low-level kernel interface. It provides minimality of functions in the kernel rather than minimality of the kernel interface. It makes extensive use of shared data structures between the kernel and the user to minimize kernel calls and to make the hardware more readily available to a user-level code. The kernel exists primarily to implement protection and to perform operations that must occur in a privileged hardware state.

The Psyche kernel interface is based on the following four abstractions, all of which are shown in Figure 10:

1. *Realm*: A realm is the unit of data sharing. It encapsulates code and data. Interprocess communication is effected by invoking operations of realms accessible to more than one process. A computation in Psyche happens by invocation of the realm operations. An invocation may be *optimized* or *protected* depending on the degree of desired protection.

2. *Process*: A process in Psyche is a thread of control implemented at the user-level.

3. *Virtual Processor*: Virtual processors are the kernel level abstractions of physical processors. Processes run on virtual processors. One virtual processor schedules more than one use-level processes. A single process may run on different virtual processors at different times.

4. *Protection Domain*: Protection domains implement a mechanism to limit access to realms. A realm belongs to a distinguished protection domain in which protected calls to its operations execute. A process moves to that distinguished domain once it performs a protected invocation of the realm. Thus, protection domains provide the boundaries

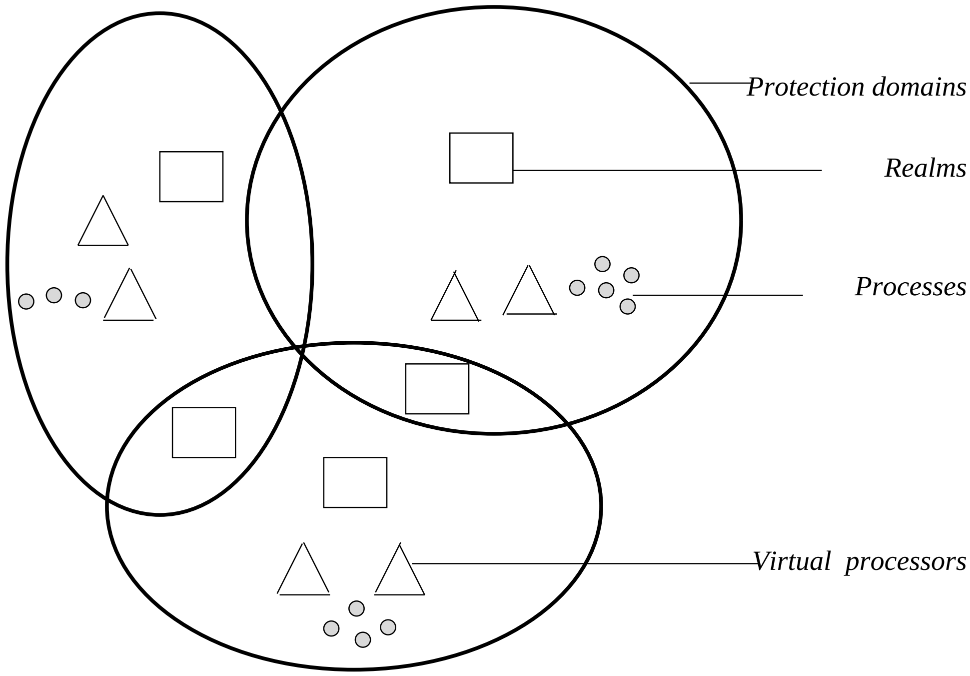

Figure 10: Basic Psyche Abstractions

between distinct models of parallelism. Implementation-wise, each protection domain has a separate page table which maps its realms.

To facilitate sharing of arbitrary realms at run time, Psyche arranges for every realm to have a unique system-wide virtual address space. Processes in different domains may be represented differently. The kernel keeps track of the call chains of processes moving between protection domains. However, it does not keep information regarding the representation/scheduling of processes inside a domain.

To execute a process within a protection domain, a user must ask the kernel to create a set of virtual processors which determines the maximum level of physical parallelism available to the domain's processes. The kernel then time-slices among the virtual processors currently located on a physical node.

The Psyche kernel is symmetric. Each cluster (node in case of Butterfly) contains a separate copy of most of the kernel code. The original Psyche implementation uses shared memory as the primary kernel-to-kernel communication mechanism. However, for better performance, the design was later modified so that the kernels communicate with each other via remote invocations [80] instead.

4.5.1 Synchronization

The Psyche kernel implements four types of synchronization – *disabled preemption* (for processor-local data structures), *locked-out interrupts* (to synchronize with device handlers), spin locks, and semaphores. Interesting experimental results with these primitives are reported in [168].

4.5.2 Memory Management

The Psyche virtual memory system (Figure 7) has the goal to integrate NUMA memory management with other kernel functions. The design [136] consists of four distinct abstraction layers. The lowest layer encapsulates physical page frames and page tables. The next layer provides an illusion of uniform memory access time through page replication and migration. The third layer provides a default pager for backing store and a mechanism for user level pagers, and the final layer implements the Psyche uniform address space and protection domains.

The Psyche virtual memory implementation has evolved over time to provide more user control. The data migration and replication layer is removed from the current implementation resulting in a simpler virtual memory system, where each virtual address space is represented by a hardware page table on a node that executes it. Code is replicated automatically on each executing processor. The creator of a realm owns the right to specify whether the replication should occur when the realm is created, when it is opened for access in a particular protection domain, or page by page on demand.

4.5.3 Scheduling

Psyche employs a two level scheduling scheme. The kernel scheduler schedules virtual processors on physical processors, and the user-level scheduler schedules processes on virtual processors. To support user-level scheduling the kernel provides a user with virtual processors, software interrupts, and magic pages [161].

The kernel scheduler schedules virtual processors on physical processors in a round-robin fashion. Users create the processes to run on virtual processors and have complete control over the scheduling of these processes.

Whenever a scheduling decision has to be made, the kernel communicates with virtual processors using software interrupts. Users can define handlers for each type of interrupt. When an interrupt occurs, a data structure shared between the kernel and the user is set to contain the state of the running process. A handler can use such information to perform context switch or to make long-term scheduling decisions.

The kernel maintains relevant information (*e.g.,* load on each physical processor, mapping between virtual and physical processors, states of virtual processors etc.) in magic pages, which it updates periodically. Such information facilitate decisions regarding user-level scheduling.

4.6 PRESTO

PRESTO [24, 23], developed at the University of Washington, is a set of tools for building parallel programs on shared-memory multiprocessors. PRESTO is not an operating system in the true sense. Instead, it is implemented on top of an existing operating system. However, PRESTO is included in this survey because it addresses issues concerning operating system configurability important to parallel systems research.

PRESTO's goal is to provide a framework within which programmers can easily build a programming model that is most appropriate for a given application domain. PRESTO allows this degree of customization using an object-oriented programming paradigm. It encapsulates system entities like processor control, scheduling, concurrency, and synchronization inside default structures with fixed interfaces. Such interfaces insulate users from an object's internal state and implementation. As long as the interface remains the same, a change in an object's behavior or implementation is not noticed by others.

PRESTO supports five fundamental objects:

1. *Thread*: A thread can be created, destroyed, put to sleep and awakened.
2. *Spinlock*: A spinlock guards a critical section.
3. *Synchronization Object*: A synchronization object consists of a spinlock (spinlock implements the mutual exclusion), a queue (threads block in this queue), and a few objects required to implement its semantics. *PRESTO* does not enforce any semantics on a synchronization object.
4. *Scheduler*: The scheduler maintains a pool of threads.
5. *Processor*: It encapsulates a hardware processor. When idle, it extracts a thread from a pool maintained by the scheduler and executes it. The thread then becomes active.

4.6.1 Customization

The lowest levels of PRESTO's runtime kernel can be modified and/or extended by an application, if required. However, PRESTO does not support the idea of reconfiguration at the operating system kernel level. As stated by the authors in [24]:

> " ... it is infeasible for an operating system to permit easy redefinition of the concepts of a processor, scheduler, lock or even a thread. These are the most basic components of an operating system, and allowing users the freedom to change them could result in chaos. .... "

In PRESTO, customization is performed in three basic ways:

1. **Layered Extension:** Layered extension allows programmers to build new from existing primitives. The major problems with such layering are performance degradation and the difficulties in expressing a new abstraction in terms of existing ones. Layering is useful for building new abstractions, but it does not allow existing abstractions to be changed.
2. **Differential Extension:** Using the inherent property of the hierarchical type system of object-oriented languages, differential extension allows a programmer to build a new class from the existing ones by only specifying the changes.
3. **Lateral Extension:** Lateral extension allows programmers to change the behavior of objects dynamically. In PRESTO, for example, it is possible to replace the existing scheduler with a completely different one during program execution.

PRESTO was implemented on a Sequent shared memory multiprocessor. Some of the issues explored with the implementation include the utility of alternative internal representations of objects, so that an object "can maintain its own internal parallelism and control any concurrency imposed upon it by other objects" and the development of "synchro object" useful for efficient representation of complex events.

4.7 KTK

The *Kernel ToolKit* (KTK) [213, 92, 91, 178] under development at the Georgia Institute of Technology is a configurable object-based operating system kernel (designed using microkernel technology). The major design goal of the KTK project is to provide explicit support for on- and off-line program configuration. KTK is layered on a portable and configurable threads-based micro-kernel [178]. As a result, KTK can be run on diverse platforms, including at user level on SUN SPARCstations, on a Kendall Square Research supercomputer, SGI machines, and as a native operating system kernel on the GP1000 BBN Butterfly multiprocessor.

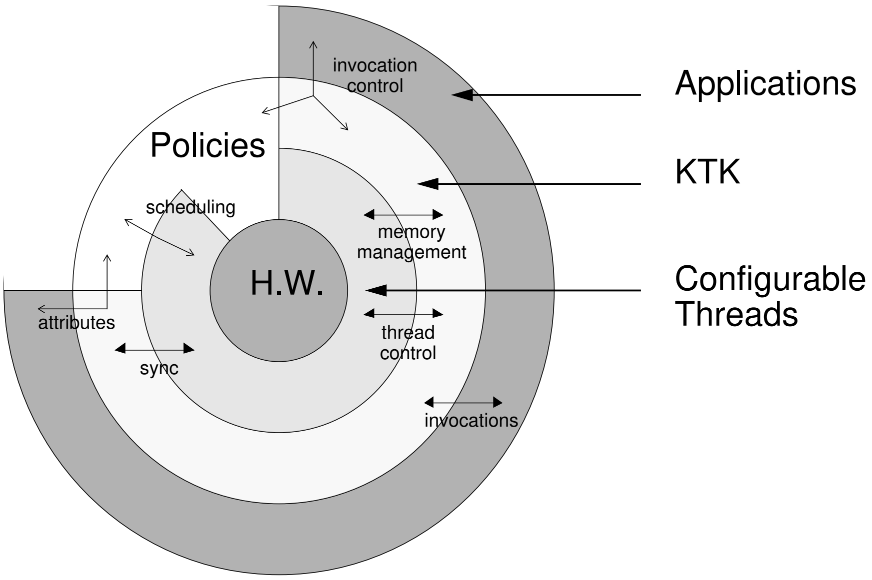

Figure 11: Structure of the Kernel Toolkit (KTK)

KTK structure. The Kernel Toolkit consists of the three major components shown in Figure 11: (1) a configurable micro-kernel as the threads layer, (2) the Kernel Toolkit's built-in object types and its support for attributes and policies, and (3) the various *policies* and *attributes* implemented for the application programs built with KTK.

The configurable micro-kernel is the partially machine dependent component [174] that implements the basic abstractions used by the remainder of KTK: *execution threads*, *virtual memory regions*, *synchronization primitives*, *monitoring support* for capture of parallel program and KTK state, and a limited number of *basic attributes* for the configuration of threads-level abstractions, such as synchronization primitives and low-level scheduling.

Objects. A KTK application program consists of a number of independent objects which interact by invoking each other's operations (methods). Each object maintains its own state, and that state is not directly accessible to other objects. Objects can range from *light-weight* procedure-like entities to multi-threaded servers with associated concurrency control and scheduling policies. *Complex* objects can be built by having objects as components of other objects, starting with four built-in object classes chosen due to their usefulness in a wide variety of parallel applications constructed with KTK: `'ADT'`, `'TADT'`, `'Monitor'` and `'Task'`[12] An `'ADT'` (abstract data type) defines an object that has no execution threads of its own and does not synchronize concurrent calls. Calling an ADT is performed in the address space of the caller. Calling a `'TADT'` (threaded abstract data type) creates a new execution thread for execution of the called operation, but also does not synchronize concurrent calls.

[12]The built-in object classes in KTK are quite similar to concurrent object constructs offered in recent designs and implementations of object-oriented concurrent languages like CC++.

A `'Monitor'` [107] is an object without execution threads that only allows a single call to be active at a time. It can also define *condition variables* on which calls can *wait*, thereby allowing other calls to proceed, until the condition variable is *signaled*. A `'Task'` (like Ada tasks) has a single execution thread. It defines a number of *entries* which can be called from other objects. All calls are performed in the context of the `'TASK'` and are taken one at a time. In addition to these primitive objects, KTK also provides support for distributed objects (DSA)[57] which permits programmers to define and create encapsulated, fragmented objects, and offers low-level mechanisms for implementing efficient, abstraction-specific communications among object fragments.

Typical KTK programs consist of complex objects constructed from the four built-in object classes (called 'layered' extension in PRESTO). KTK offers the other methods of extension identified by PRESTO and additional, dynamic configuration by permitting the definition of new policy classes and their linkage with the kernel, and by offering two distinct views of each object exist: (1) the *application view* and (2) the *system view*. The *application view* of objects is presented in terms of their classes characterizing their external interfaces (methods), where a *class* is an abstraction for a number of similar objects. The system view, on the other hand, is defined by each object's *policy* and *attributes*. Essentially, policies define a parameterized execution environment for objects in terms of attributes, *invocation semantics*, and *kernel interactions*:

- A policy interprets attributes defined at the time of creation for classes, objects, states and operations.
- A policy can define the invocation semantics to an object by intercepting invocation requests and by defining and interpreting invocation time attributes that can be specified as part of the invocation.
- A policy can also extend the KTK interface with special services.

Since policies are executed implicitly as a result of object creation and invocation, typical application programs see only the objects and classes defined in their code and offered by KTK.

Synchronization. In addition to primitive spin and blocking locks, KTK supports configurable lock objects which can be configured to suit application's requirements [176, 177]. Such locks contain a set of implementation-dependent attributes which can be dynamically altered to result in a continuous spectrum of lock configurations ranging from 'busy waiting' to 'blocking' (Some useful configurations are: combined locks, advisory locks, priority locks, handoff locks, and adaptive locks). Furthermore, configurable locks implement a "customized monitor" module that can be used to sense the current state of a lock [177].

Configuration. KTK's support for reconfiguration consists of three mechanisms – attributes, policies, and a monitoring mechanism (to sense current program state). The Kernel Toolkit offers explicit support for on- and off-line object configuration using the above mechanisms:

- KTK allows the specification of (configuration) attributes for object classes, object instances, state variables, operations and object invocations.
- Attributes are interpreted by system- or programmer-defined policies, which may be varied separately from the abstractions with which they are associated. For example, policies and attributes may be used to vary objects' internal implementations without changing their functionalities, or to vary the semantics and implementations of object

invocations without affecting the methods being invoked. A policy can be associated with any of the program components object instance', 'class', 'state variable', 'method', and even 'object invocation'. This association is performed such that the program component's implementation and specification are not affected.

- Dynamic configuration may be performed by policies at or below the object level of abstraction, therefore permitting programmers to make dynamic changes of selected attributes of lower-level runtime libraries and to exploit peculiarities of the underlying multiprocessor hardware. KTK also offers efficient mechanisms for the on-line capture of the program or operating system state required for dynamic configuration.

The mechanisms for configuration in KTK can also be used for kernel customization by specializing and/or modifying existing kernel abstractions. In addition, KTK is *extensible* in that new abstractions and functionality (ie., classes, policies, and attributes) are easily added while potentially maintaining a uniform kernel interface (e.g., when not adding any new kernel classes) [92, 147].

4.8 Choices

The operating system family called Choices (Class Hierarchical Open Interface for Custom Embedded Systems) [46, 47, 199, 200, 203, 155] is part of the Embedded Operating System (EOS) project at the University of Illinois at Urbana-Champaign. The Choices kernel is implemented on a 10 processor Encore Multimax multiprocessor using the C++ language.

Choices is an example of a customizable operating system that can be tailored for a particular hardware configuration or for a particular application. Choices is targeted towards large multiprocessors interconnected by shared-memory or high speed networks. It uses a class hierarchy and inheritance to represent the proper abstractions for deriving and building new instances of a Choices system.

Choices support multitasking, where each task may use multiple processors. The Choices' support for an application is divided into two parts – Germ and Kernel. A set of classes, called as *Germ*, encapsulates the major hardware dependencies, implements the mechanisms for managing physical resources, and provides a uniform hardware architecture to the rest of the classes in the hierarchy, whereas, a *Kernel* is a set of classes that implements the resource allocation policies and supports applications. A customized system is built using tailored kernel classes and derived germ classes to suit a particular hardware.

4.8.1 Tasks and Threads

Choices implement threads in the kernel [200]. It supports multiple threads within a single task. Although they are called lightweight threads in [47], according to the terminology introduced in this paper they are middleweight threads. A task is a collection of address spaces and active threads.

4.8.2 Interprocess Communication

Communication in Choices is achieved mainly via shared memory. The operating system class hierarchy provides a few common shared memory and message passing communication schemes that can be extended or customized. The existing communication schemes include a Path C++ class, monitors, semaphores, and simple varieties of guarded commands.

Communication between address spaces (tasks) is performed using shared *persistent* objects, which support stream-based communications, broadcasts, multicasts, and block I/O.

4.8.3 Memory Management

Germs provide two classes of objects for memory managements:

1. *Store*: A store encapsulates the physical memory. The operations allowed on a store are store instantiation, store destruction, page allocation, and page deallocation.
2. *Space*: A space encapsulates the paged virtual address range and its mapping onto physical memory and/or fault handler. The operations provided are space creation, space destruction, virtual page allocation and deallocation, mapping virtual pages onto physical pages of a store, and installation of fault handlers.

The *Universe* class represents the aggregate of the spaces in a virtual memory. A task can be constructed to use any space region. The threads within a task created in the kernel space are always resident and addressable.

4.8.4 Persistent Objects

Persistent objects are the instances of the classes that reside in memory for a longer period than the execution time of a particular task. Persistent objects are shared between tasks and implement many operating system components in a protected space outside the kernel. Germ and kernel objects are a few examples of persistent objects in Choices. Persistent objects may be active and may act as servers.

A task accesses a persistent object using an *object descriptor* and a method. The object descriptor implements a capability based addressing scheme. When accessed, persistent objects are mapped into an *Object Space*. Therefore, the contents of an object space may change over time according to requests from tasks associated with the object space. Multiple processors may map a persistent object in their virtual memory simultaneously.

4.8.5 Exception Handling

In Choices, there are two kinds of exceptions:

1. *Events*: Events are asynchronous mechanisms generated by hardware (handled by event mechanism in germs) or by software (kernel provided).
2. *Traps*: Traps, generated by an executing thread, are handled by kernel-provided or user-provided *trap handler* objects.

4.9 Renaissance

Renaissance [202], a successor of Choices [47, 199, 200, 203] operating system, is currently under development at Purdue University. It extends the ideas of Choices into a distributed object environment. The goal of Renaissance is to provide transparent access to remote objects that are distributed throughout a network of machines. Renaissance is an object-oriented operating system and is implemented using C++ language.

4.9.1 Process Management

A process represents a single control path through a program in execution. Three abstract classes manage the functions of process management and scheduling. The *Process* class represents a process and defines its context, the *Process Container* class represents a collection of Process classes (represents process queues), and the *Processor* class represents a physical processor (stores processor-specific information such as local ready queues). Instances of these classes manipulate the process model.

The context of a process is split into two objects. The Process object contains the architecture independent context and another object called *ProcessorContext* contains the architecture dependent context. Sending an appropriate message to a Processor object returns the next process in its ready queue and context switch between two processes is initiated by sending messages to corresponding Process objects.

4.9.2 Synchronization

Renaissance supports several mutual exclusion and synchronization classes. The *Lock* class defines a low-level mutual exclusion object. The lowest level concrete synchronization class is the *PrimitiveSpinLock* class which provides a simple and crude spin lock, implemented using hardware provided atomic test-and-set instructions. *SpinLock* is a subclass of PrimitiveSpinLock and implements a more sophisticated busy waiting lock. Other important synchronizations classes include the *Semaphore* Class (implements Dijkstra's counting semaphore [72]), the *GraciousSemaphore*[13] class, the *BusyWaitingReadWriteLock* and the *BlockingReadWriteLock* classes, the *TimedSemaphore* class (augments a counting semaphore with a timeout mechanism), and the *Event* class.

4.10 DYNIX

The *Dynix* operating system [224, 223] is an enhanced version of the Unix operating system developed to run on the Sequent multiprocessors. The Sequent multiprocessor is a bus based uniform access shared memory (UMA) machine.

4.10.1 Process Management and Scheduling

Like Unix, Dynix supports heavyweight processes. The processors in the system share a set of run queues. In addition, each processor has a private run queue. Dynix supports a preemptive priority based scheduling. The processes are grouped into 32 queues according to priorities. An idle processor first checks its private queue for a runnable process. If there is none, it selects a process to execute from the highest-priority non-empty shared queue. Within a queue, the processes are executed in a time shared, round robin fashion.

The priority of a process may change over its lifetime. Periodically (a system-defined time period), the priority of a process is recalculated. The priority of a CPU-intensive process is lowered gradually. A user may also alter the priority of a process to a certain extent.

In Dynix, a process can restrict itself to a particular processor[14]; the process is placed in the private queue of the processor.

A version of Dynix provides a parallel programming library which supports less expensive processes or threads of control. It supports multiple threads of control in one Dynix process.

[13] Gracious semaphore is originally introduced in [203]. Such a semaphore causes the current process to immediately relinquish the processor when a *V* message is sent and there are blocked processes.

[14] also known as *processor/process affinity*.

4.10.2 Synchronization

The Dynix kernel provides locks and semaphores for mutual exclusion and synchronization. The interrupt priority of a processor holding a lock is set to ensure that the processor is not interrupted. A semaphore is used to guard a large critical section.

4.10.3 Memory Management

The virtual memory system of Dynix is an enhanced version of Unix. Unlike Unix, which uses a *global model*, Dynix uses a *local model*. A process has a greater role in its own paging activity.

4.11 UMAX

UMAX [171] like Dynix is an extension of Unix, and runs on Encore Multimax multiprocessors. The Encore Multimax is a bus based, shared memory multiprocessor.

There are two versions of UMAX – UMAX 4.2 and UMAX V. UMAX 4.2 is compatible with Unix 4.2 bsd, whereas UMAX V is compatible with Unix System V.

4.11.1 Process Management and Scheduling

UMAX is a multi-user, multi-programmed, and multithreaded operating system. It is similar to Dynix, including its support of heavyweight "Unix-like" processes. A parallel program is constructed using multiple "Unix-like" processes sharing a portion of the process data space. However, for medium grained parallelism, UMAX also provides a multitasking library which implements the notion of *tasks*, which are similar to Mach threads. Each process may have multiple tasks sharing its address space.

4.11.2 Synchronization

The synchronization primitives provided are spin locks, semaphores (Dijkstra style), and Read/Write locks. Locks and semaphores are used hierarchically to prevent deadlock and indefinite postponement.

4.11.3 Memory Management

Both versions of UMAX support demand-paged virtual memory to provide up to 16 megabytes of virtual address space per processor.

4.12 Chrysalis

The *Chrysalis* operating system [64, 130, 81] provides a "Unix-like" environment on the Butterfly parallel processor. The Butterfly is a non-uniform memory access (NUMA) shared memory multiprocessor which uses an interconnection network as a processor/memory switch. Each machine node consists of a processor and its local memory. Collectively, the local memories of all processor nodes form the shared global memory of the machine. A processor can access any memory in the machine using the switch. A remote memory access is more expensive than a local memory access.

4.12.1 Process Management and Scheduling

The kernel provides the lowest level of the Chrysalis operating system functions, and it runs on each processor. It supports processes and per-processor schedulers which allow more than one process to run in a node concurrently with processes on other nodes.

4.12.2 Memory Management

The Butterfly hardware uses a segmented virtual memory management system. Memory protection (kernel/user mode read, write and execute attributes) is enforced on a per segment basis. Memory mapping is managed by the kernel and can be controlled directly by applications. The Chrysalis kernel provides another level of abstraction above the hardware's memory management by supporting objects which are associated with areas of physical memory or special system data structures. Such objects provide processor independent names for areas of memory or for system structures. An object can be mapped into the virtual address space of a process; object identifiers (called *object handles*) can be passed between processors via inter-process communication facilities.

4.12.3 Synchronization

The objects associated with system data structures provide inter-process communication and synchronization primitives. The two most common synchronization primitives are *events* and *dual queues*. Dual queues are interlocked data queues which are used as locks or to pass data between processes. A process may suspend on an event. When the event is posted, the process is scheduled to run. A few micro-coded atomic memory operations are also provided to build simple locks and other synchronization primitives.

A number of servers are implemented on top of the kernel to provide additional functionalities such as network capabilities, remote debugging, remote file system *etc.*

The application libraries built for Chrysalis export the operating system's interface to an application program. Such libraries includes the *Uniform System* which is used to develop parallel programs for the Butterfly parallel system, a buffer management package for communicating applications, a stream-oriented i/o interface, and a library of performance measurement tools for parallel applications.

4.13 RP3

The *RP3* [45, 44] is a NUMA shared memory multiprocessor developed at T. J. Watson Research Center, IBM. The RP3 hardware is designed to be scalable up to 512-way multiprocessing. A 64-way prototype machine was built and made operational. The RP3 architecture is designed to give an application direct control over the hardware as a way of enabling the application to achieve speedups on the parallel machine (*e.g.,* an application program may choose to be in charge of cache coherence).

The RP3 architecture consists of a number of processor memory elements (PME's) connected via an interconnection network. A PME consists of a RISC processor, a floating point unit, an I/O interface, a MMU, cache, a memory controller, a memory module, a network interface, a switch interface, and a performance measurement chip.

The RP3 operating system, like most other commercial systems, is an extension of Unix. Mach is the base implementation of Unix on the RP3[15]. Like Mach, Mach/RP3 also has a master processor, which is reserved for Unix system call service.

[15] A second operating system for the RP3 developed at NYU was never deployed on the actual machine [6].

4.13.1 Process Management and Scheduling

Mach/RP3 supports a version of co-scheduling or gang scheduling known as *family scheduling*. A *family* is a set of cooperating processes (possibly exchanging messages, sharing part or all of their address space, synchronizing often) working towards a single goal. A *thread family* is the largest schedulable unit on the RP3, and corresponds to the notion of a family. A port is associated with a family. A thread that has rights to a family port can request a processor to be allocated to or deallocated from the family. The threads in a family are allowed to run only on the allocated processors; non-family threads are barred from the allocated processors. The members of a family time-share its processors. A thread in a family can choose to be bound to a particular processor. A notable feature of RP3 family scheduling interface is its flexibility in allowing a thread to issue requests for other threads. For example, a thread can bind another thread, and a thread can request processor allocation for another family.

4.13.2 Memory Management

RP3 exports the non-uniform memory model to applications and therefore, does not deal with issues like data placement and cacheability. Mach/RP3 allows a task to specify *virtual memory attributes* for pages of its virtual address space (like inheritance and protection attributes in Mach). Such attributes include *location attributes* (a thread may specify ranges of virtual pages to be placed in the local memory), *replication attributes* (a thread can request to replicate ranges of virtual memory in different memory modules), and *cacheability attributes* (a thread can request ranges of virtual pages to be made cacheable).

4.14 Operating Systems for Distributed Memory Machines

There has not been much work on operating systems for distributed memory machines, in part because of designers' hardware backgrounds, in part due to a narrow focus on attaining high performance for application programs, where 'less operating system' was considered to result in 'better performance' for such machines. However, recent commercial ventures do not have these characteristics, because industry end users are demanding operating system compatibility with existing workstation or sequential supercomputer platforms. Instead, current distributed memory machines like the Intel Paragon and the Thinking Machines CM-5 offer full OSF Unix operating systems. Unfortunately, system implementors cannot make use of a plethora of past research results. A brief outline of operating system research on distributed memory machines follows.

The Caltech hypercube developments resulted in broad interest in the concept of *active messages*. The basic idea of active messages is to associate computing tasks with message transfers such that the system minimizes the delay between message arrival at some node and the initiation of computations enabled by that message. Dally at MIT is constructing hardware support for active message machines [66]. Livny and Manber explore a similar idea in their work on 'active channels' [152, 151], in which a token ring communication protocol is extended such that three classes of operations could be performed directly on the node interfaces: arithmetic, selection, and counting. Simulation studies of their hardware proposals show its usefulness in several applications, including dynamic load balancing, sorting, work distribution in DIB [82], *etc.*

As with Livny and Manber's work, research on an active message construct called 'topologies' on hypercubes [211] and 'distributed shared abstractions' on shared memory multiprocessors [57] is based on previous work in message-based operating systems for hypercubes, since it is assuming the presence of reliable message delivery and of processes on the in-

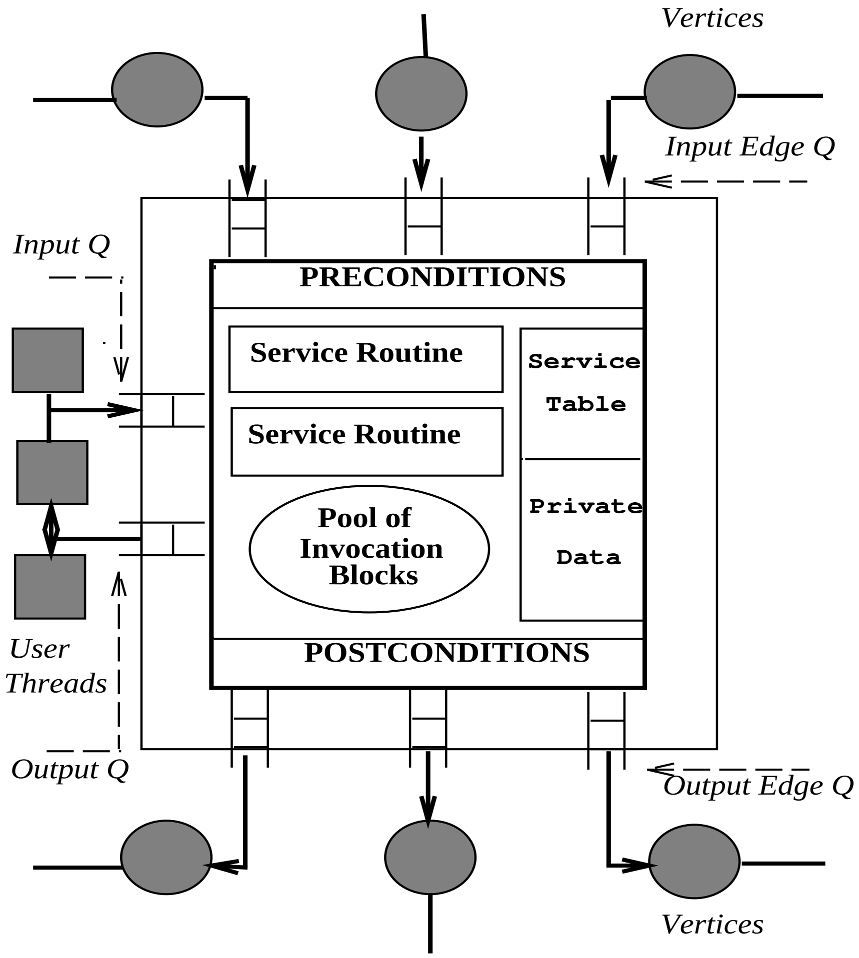

Figure 12: Object Fragments

dividual nodes of the hypercube. Based on such a lower-level message protocol, kernel- or user-level services are associated with the receipt or sending of individual messages, somewhat resembling the explicit user-level communication calls issued by user programs in the Crystalline operating system [86]. The event-driven execution model of 'topologies' is similar to the execution model supported by the reactive kernel [221] for the Symult series of multicomputers, which schedules user processes according to conditions that concern the receipt of messages for which processes are waiting.

A sample object fragment as defined in [211] and in [57] appears in Figure 12. The basic idea of active messages is reflected here, in terms of user threads and service routines closely associated with the incoming or outgoing message buffers of the communication topology linking object fragments. Simpler and higher performance implementations of services or threads attached to messages are being implemented in systems like those designed by Dally at MIT [66].

In other work, slight generalizations of the functionality of the Crystalline operating system are the MOOSE operating system [205] designed at CalTech and the early commercial operating systems developed at Intel and NCube for their hypercube machines. For example, MOOSE explicitly addresses process migration and load balancing, and the Intel operating systems offer simple notions of processes and process communication constructs, later amplified by support for concurrent I/O.

5 Conclusions and Future Work

Having reviewed this much material on multiprocessor operating systems, it is tempting to try to predict future developments in parallel operating systems. We will resist that temptation. Instead, we conclude by apologizing to the many system developers whose work is not mentioned in this survey and by stating our motivation for highlighting specific topics and systems.

Our main motivation for writing this survey is the realization that current multiprocessor developments can be assisted tremendously by judicious reviews of past research. This is particularly important in the case of parallel operating systems because the OS community's research focus on parallel systems in the late 70's shifted to distributed systems for much of the 80's and is only now again addressing parallel machines. Hoping to remind researchers of past results and insights, we have included relatively 'old' operating systems like HYDRA in this survey which can be thought of as one of the culminations of research on protection issues as well as a starting point for operating system developments for parallel machines. Similarly, the StarOS system along with Roscoe [230] and Medusa [184] provides early implementations of micro-kernels, of user-level operating system services, of internally parallel or reliable system services, and of alternative operating system constructs providing similar functionalities at differing costs. Another motivation for the inclusion of HYDRA and StarOS is our expectation that their insights on protection in operating systems may become more important with 64 bit address spaces on large-scale parallel machines.

A second motivation for writing this paper is the aforementioned 'convergence' of several technologies relevant to parallel computing. First, the convergence of high performance computing and networking technologies are now resulting in large-scale, physically distributed, and heterogeneous parallel machines. The associated technologies were originally developed for parallel vs. distributed systems, by partially divergent technical communities and sometimes discussed with different terminologies. Technological convergence is driven by recent hardware developments, where (1) multiprocessor engines, when scaled to hundreds of processors, can appear much like distributed sets of machines, and (2) distributed machines linked by high performance networks (especially local area networks or network devices derived from current ATM or even supercomputer routing technologies) are being increasingly used as parallel computing engines. This has become increasingly obvious with the evolution of the Mach operating system from one that addressed single or networks of workstations, to one also running on NUMA and now on NORMA machines. It has prompted us to include NUMA operating systems like Elmwood and Psyche in this survey, along with operating systems like the reactive kernel and constructs like topologies for NORMA machines, while also mentioning developments originally proposed for distributed systems like weak memory systems. We believe that many of the ideas in those designs and implementations can be fruitfully applied toward the development of future parallel machine operating systems.

Our third motivation for writing this survey is the current excitement in operating systems research in general, where new applications like interactive distributed systems and new software technologies like object-oriented software development and languages are prompting designers to seek new dimensions of system configurability not only for large-scale parallel machines addressed by this survey but also for sequential machines ranging from simple hand held communication and computation devices to supercomputers. One example of an industry effort to apply object-oriented technologies to operating system development is the

Spring operating system now being commercialized at SUN microsystems [103]. The sample research systems we included in this survey are Elmwood, Kernel ToolKit, Psyche, Choices, and Renaissance.

Two insights concerning operating systems for parallel machines underly much of this paper's presentation. First, modern operating systems are built using multiple and often related system structuring techniques, and they will inevitably offer different implementations of common functionality tailored to their target architectures and intended application domains. Second, no single set of operating system facilities will result in high performance for all possible parallel application programs. As a result, modern systems strive to place more and more system functionality into user-level code or even offer multiple and different user-level operating system platforms for parallel computing on a single underlying system kernel. Interesting future research questions include: What are the cutpoints between user-level vs. kernel functionality? What are the appropriate interfaces between both? What are the most suitable mechanisms for provision of such externally driven system configurability? A different statement posing the same questions is the admission that there is probably no single parallel programming model that is appropriate for all parallel application programs. As a result, future systems are more likely to offer multiple, diverse programming model rather than focus on a single powerful parallel programming paradigm.

References

[1] Jr. A. Tevanian. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments*. PhD thesis, School of Computer Science, Carnegie-Mellon University, December 1987. Technical Report CMU-CS-88-106.

[2] V. Abrossimov, M. Rozier, and M. Shapiro. Generic virtual memory management for operating system kernels. In *Proceedings of the 12th symposium on operating systems principles (SIGOPS Notices vol.23, no.5)*, pages 123–36, December 1989.

[3] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. In *Proceedings of the Summer 1986 Usenix Conference*, pages 93–112, July 1986.

[4] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[5] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. Technical Report 1051, Department of Computer Science, University of Wisconsin, Madison, September 1991.

[6] G. S. Almasi and A. Gottlieb. *Highly parallel computing*. Benjamin/Cummings, Redwood City, Calif., 1989.

[7] R. Ananthanarayanan, Mustaque Ahamad, and Richard J. LeBlanc. Application specific coherence control for high performance distributed shared memory. In *Proceedings of the 3rd USENIX Symposium on Experience with Distributed and Multiprocessor Systems (SEDMS III)*, pages 109–28, March 1992.

[8] T. E. Anderson. The perfomance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[9] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *Transactions on Computer Systems, ACM*, 10(1):53–79, February 1992.

[10] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[11] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (SIGPLAN Notices vol.26, no.4)*, pages 108–20, April 1991.

[12] A. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (SIGPLAN Notices, vol.26, no.4)*, pages 96–107, April 1991.

[13] J. Archibald. *The Cache Coherence Problem in Shared-Memory Multiprocessors*. PhD thesis, Department of Computer Science, University of Washington, February 1987.

[14] R. Atkinson, A. Demers, C. Hauser, C. Jacobi, P. Kessler, and M. Weiser. Experiences creating a portable cedar. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 322–8, Portland, OR, June 1989.

[15] R. Baron, D. Black, W. Bolosky, J. Chew, R. Draves, D. Golub, R. Rashid, A. Tevanian, and M. Young. *Mach Kernel Interface Manual*. School of Computer Science, Carnegie Mellon University, August 1990.

[16] Forest Baskett, John Howard, and John Montague. Task communication in demos. *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pages 23–31, Nov. 1977.

[17] Gerard M. Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*. PhD thesis, Computer Science Department, Carnegie-Mellon University, April 1978.

[18] J. Bennett. The design and implementation of distributed smalltalk. In *OOPSLA'87 Conference Proceedings*, pages 318–330, October 1987.

[19] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seattle, (SIGPLAN Notices, vol.25, no.3)*, pages 168–76. ACM, March 1990.

[20] B. Bershad. The increasing irrelevance of ipc performance for microkernel-based operating systems. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 205–211, April 1992.

[21] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990. Also appeared in Proceedings of the 12th ACM Symposium on Operating Systems Principles, Dec. 1989.

[22] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.

[23] B. Bershad, E. Lazowska, and H. Levy. Presto: A system for object-oriented parallel programming. *Software: Practice and Experience*, 18(8):713–732, August 1988.

[24] B. Bershad, E. Lazowska, H. Levy, and D. Wagner. An open environment for building parallel programming systems. In *Proceedings of the Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, pages 1–9, July 1988.

[25] Brian N. Bershad. High performance cross-address space communication. Technical Report 90-06-02, Dept. of Computer Science and Eng., University of Washington, June 1990. Ph.D. dissertation.

[26] T. Bihari and K. Schwan. A comparison of four adaptation algorithms for increasing the reliability of real-time software. In *Proceedings of the Ninth Real-Time Systems Symposium, Huntsville, AL*, pages 232–241. IEEE, Dec. 1988.

[27] T. Bihari and K. Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991.

[28] Kenneth P. Birman and et.al. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, pages 502–508, June 1985.

[29] A. Birrell, J. Guttag, J. Horning, and R. Levin. Synchronization primitives for a multiprocessor: A formal specification. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 94–102. ACM, December 1987.

[30] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[31] P. Biswas, K. Ramakrishnan, D. Towsley, and C. Krishna. Performance analysis of distributed file systems with non-volatile caches. In *Proceedings of the 2nd International Symposium on High Performance Distributed Computing*, pages 252–262, July 1993.

[32] D. Black, D. Golub, D. Julin, R. Rashid, R. Draves, R. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel operating system architectures and mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, April 1992.

[33] D. Black, D. Golub, R. Rashid, A. Tevanian, and M. Young. The mach exception handling facility. Technical Report CMU-CS-88-129, School of Computer Science, Carnegie Mellon University, April 1988.

[34] David L. Black. The mach cpu_server: An implementation of processor allocation. School of Computer Science, Carnegie-Mellon University, August 1989.

[35] David. L. Black. *Scheduling and Resource Management Techniques for Multiprocessors*. PhD thesis, School of Computer Science, Carnegie Mellon University, July 1990. Techreport CMU-CS-90-152.

[36] David. L. Black. Scheduling support for concurrency and parallelism in the mach operating system. *IEEE Computer*, 23(5):35–43, May 1990.

[37] Ben A. Blake and Karsten Schwan. Experimental evaluation of a real-time scheduler for a multiprocessor system. *IEEE Transactions on Software Engineering*, 17(1):34–44, January 1991.

[38] Toby Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, MIT/LCS/TR-303, March 1983.

[39] S.H. Bokhari. Dual processor scheduling with dynamic reassignment. *IEEE Transactions on Software Engineering*, SE-5(4):341–349, July 1979.

[40] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for numa memory management. In *Proceedings of the twelfth ACM symposium on Operating Systems Principles*, pages 19–31, December 1989.

[41] W. Bolosky and M. Scott. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 57–71, September 1993.

[42] A. Bomberger, N. Hardy, A. Frantz, C. Landau, W. Frantz, J. Shapiro, and A. Hardy. The keykos nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, April 1992.

[43] Ronald F. Brender and Isaac R. Nassi. What is ada? *IEEE Computer Magazine*, 14(6):17–25, June 1981.

[44] R. Bryant, H. Y. Chang, and B. Rosenburg. Experience developing the rp3 operating system. In *Proceedings of the 2nd Usenix Symposium on Experience with Distributed and Multiprocessor Systems (SEDMSII)*, pages 1–18, March 1991.

[45] R. Bryant, H. Y. Chang, and B. Rosenburg. Operating system support for parallel programming on rp3. *IBM Journal of R & D*, November 1991.

[46] R. Campbell, N. Islam, and P. Madany. Choices, frameworks and refinement. *Computing Systems*, 5(3):217–57, Summer 1992.

[47] R. Campbell, G. Johnston, and V. Russo. Choices (class hierarchical open interface for custom embedded systems). *Operating Systems Review*, 21(3):9–17, July 1987.

[48] R. Campbell, V. Russo, and G. Johnston. The design of a multiprocessor operating system. In *Proceedings of the USENIX C++ Conference*, pages 109–25, November 1987.

[49] Nicholas Carriero and David Gelernter. The s/netś linda kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, may 1986.

[50] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *Proceedings of the thirteenth ACM symposium on Operating Systems Principles*, pages 152–164, October 1991.

[51] D. Casewell and D. Black. Implementing a mach debugger for multithreaded applications. In *Proceedings of the Winter 1990 Usenix Technical Conference and Exhibition*, pages 25–39, January 1990.

[52] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems and Principles*, pages 147–158, December 1989.

[53] Sheng-Chang Cheng, John A. Stankovic, and Krithi Ramamritham. Scheduling algorithms for hard real-time systems - a brief survey. In *Tutorial Hard Real-Time Systems*, pages 150–173. IEEE, 1988.

[54] D. Cheriton. The v kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[55] D. R. Cheriton. Problem-oriented shared memory: A decentralized approach to distributed system design. *Proceedings of the sixth. International Conference on Distributed Computing Systems, Cambridge,MA.*, pages 190–197, May 1986.

[56] D.R. Cheriton, M.A. Malcolm, L.S. Melen, and G.R. Sager. Thoth, a portable real-time operating system. *Comm. of the Assoc. Comput. Mach.*, 22(2):105–115, Feb. 1979.

[57] Christian Clemencon, Bodhisattwa Mukherjee, and Karsten Schwan. Distributed shared abstractions (dsa) on large-scale multiprocessors. In *Proc. of the Fourth USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 227–246. USENIX, September 1993. To be published in IEEE Transactions of Software Engineering.

[58] E. Cohen and D. Jefferson. Protection in the hydra operating system. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles*, pages 141–160. ACM, 1975.

[59] R. Colwell, E. Gehringer, and E. Jensen. Performance effects of architectural complexity in the intel 432. *ACM Transactions on Computer Systems*, 6(3):296–339, August 1988.

[60] E. Cooper and R. Draves. C threads. Technical Report CMU-CS-88-154, Dept. of Computer Science, Carnegie Mellon University, June 1988.

[61] A. Cox and R. Fowler. The implementation of a coherent memory abstraction on a numa multiprocessor: experiences with platinum. In *Proceedings of the twelfth ACM symposium on Operating Systems Principles*, pages 32–44, December 1989.

[62] G. Cox, M. Corwin, K. Lai, and F. Pollack. Interprocess communication and processor dispatching on the intel 432. *ACM Transactions on Computer Systems*, 1(1):45–66, February 1983.

[63] George Cox, William M. Corwin, Konrad K. Lai, and Fred J. Pollack. A unified model and implementation for interprocess communication in a multiprocessor environment. In *Proceedings of the 8th Symposium on Operating System Principles, Asilomar*, pages 44–53. Assoc. Comput. Mach., Dec. 1981.

[64] W. Crowther, J. Goodhue, R. Gurwitz, R. Rettberg, and R. Thomas. The butterfly(tm) parallel processor. Technical report, BBN Laboratories Incorporated.

[65] S. Curran and M. Stumm. A comparison of basic cpu scheduling algorithms for multiprocessor unix. *Computing Systems*, 3(4):551–79, October 1990.

[66] Dally, Chao, Chein, Hassoun, Horwat, Kaplan, Song, Totty, and Wills. Architecture of a message driven processor. *Proceedings of the 14th Annual International Symposium On Computer Architecture*, 15(2):189–196, Jun 1987.

[67] W. Dally and C. Seitz. The torus routing chip. *J. Distributed Computing*, 1(4), 1986.

[68] Partha Dasgupta, Richard J. LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. The CLOUDS distributed operating system. *IEEE Computer*, 24(11):34–44, Nov. 1991.

[69] Partha Dasgupta, Richard J. LeBlanc Jr., and William F. Appelbe. The clouds distributed operating system: Functional description, implementation details and related work. In *Proceedings of the 9th International Conference on Distributed Computing Systems, San Jose, CA.*, pages 2–9. IEEE, June 1988.

[70] J. Dennis and E. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3), March 1966.

[71] M. L. Dertouzos and A. K. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, Dec. 1989.

[72] E. Dijkstra. The structure of the “the”-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[73] L. Dowdy. On the partitioning of multiprocessor systems. Technical Report 88-06, Department of Computer Science, Vanderbilt University, July 1988.

[74] P. D. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 122–136, October 1991.

[75] R. Draves. The revised ipc interface. In *Proceedings of the Usenix Mach Conference*, pages 101–122, October 1990.

[76] R. Draves, M. Jones, and M. Thompson. Mig – the mach interface generator. Department of Computer Science, Carnegie-Mellon University, July 1989.

[77] Eds. I. Durham, S.F. Fuller, and A.K. Jones. The cm* review report. Technical report, Comp. Science Dept., Carnegie-Mellon Univ., 1977.

[78] D. Eager, J. Zahorjan, and E. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–23, March 1989.

[79] J. Edler, J. Lipkis, and E. Schonberg. Process management for highly parallel unix systems. In *Proceedings of the USENIX Workshop on UNIX and Supercomputers*, pages 1–17, September 1988.

[80] Jr. E.M. Chaves, P.C. Das, T.L. LeBlanc, B.D. Marsh, and M.L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency: Practice and Experience*, 5(3):171–192, May 1993.

[81] W. Milliken et al. *Chrysalis Programmer's Manual, version 2.2.* BBN Laboratories, June 1985.

[82] R. Finkel and U. Manber. Dib - a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–255, April 1987.

[83] R. Finlayson. Object-oriented operating systems. *TCOS Newsletter*, 5(1):17–21, 1991.

[84] R. Fitzgerald and R. Rashid. The integration of virtual memory management and interprocess communication in accent. *ACM Transactions on Computer Systems*, 4(2), May 1986.

[85] A. Forin, J. Barrera, M. Young, and R. Rashid. Design, implementation and performance evaluation of a distributed shared memory server for mach. In *Proceedings of the Winter Usenix Technical Conference*, January 1989.

[86] G.C. Fox and A. Kolawa. Implementation of the high performance cystalline operating system on intel ipsc hypercube. Technical report, Caltech Concurrent Computational Program and Physics Dept., Caltech, Pasadena CA, Jan. 1986.

[87] Geoffrey C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems On Concurrent Processors*. Prentice-Hall, 1988.

[88] S.H. Fuller, J.K. Ousterhout, L. Raskin, P.I. Rubinfeld, P.J. Sindhu, and R.J. Swan. Multiprocessors: An overview and working example. *Proceedings of the IEEE*, 66(2):216–228, Feb. 1978.

[89] Edward F. Gehringer, Daniel P. Siewiorek, and Zary Segall. *Parallel Processing: The Cm* Experience*. Digital Press, Digital Equipment Corporation, 1987.

[90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[91] Ahmed Gheith, Bodhisattwa Mukherjee, Dilma Silva, and Karsten Schwan. Ktk: Kernel support for configurable objects and invocations. In *Second International Workshop on Configurable Distributed Systems*. IEEE, ACM, March 1994.

[92] Ahmed Gheith and Karsten Schwan. Chaos-arc – kernel support for multi-weight objects, invocations, and atomicity in real-time applications. *ACM Transactions on Computer Systems*, 11(1):33–72, April 1993.

[93] M. Gien. Micro-kernel design. *UNIX REVIEW*, 8(11):58–63, November 1990.

[94] A. Goldberg and R. Tarjan. A new approach to the maximum flow problem. In *Proceedings of the 18th ACM Symposium on Theory of Computing*, pages 136–146, 1986.

[95] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proceedings of the Summer Usenix Technical Conference*, pages 87–96, June 1990.

[96] Prabha Gopinath and Karsten Schwan. Chaos: Why one cannot have only an operating system for real-time applications. *Operating Systems Review*, 23(3):106–125, July 1989.

[97] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The nyu ultracomputer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1990.

[98] Y. Gourhant and M. Shapiro. Fog/c++: a fragmented-object generator. In *Proceedings of the USENIX C++ Conference*, pages 63–74, April 1990.

[99] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–70, June 1990.

[100] MACH Networking Group. Network server design. School of Computer Science, Carnegie Mellon University, August 1989.

[101] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating systems scheduling policies and synchronization methods of the performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–32, May 1991.

[102] S. Habert, L. Mosseri, and V. Abrossimov. Cool: Kernel support for object-oriented environments. In *ECOOP/OOPSLA'90 Conference (SIGPLAN Notices vol.25, no.10*, pages 269–277. ACM, October 1990.

[103] G. Hamilton, M. Powell, and J. Mitchell. Subcontract: A flexible base for distributed programming. Technical report, Sun Microsystems Laboratories Inc., SMLI TR-93-13, April 1993.

[104] M. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 276–90, August 1988.

[105] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM Sigplan Symposium on Principles and Practice of Parallel Programming (SIGPLAN Notices vol.25, no.3)*, pages 197–206, March 1990.

[106] D. Hildebrand. An architectural overview of qnx. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126, April 1992.

[107] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[108] W. A. Horn. Some simple scheduling algorithms. *Naval Res. Logist. Quart.*, 21:177–185, 1974.

[109] R. Hou and Y. Patt. Trading disk capacity for performance. In *Proceedings of the 2nd International Symposium on High Performance Distributed Computing*, pages 263–270, July 1993.

[110] N. Hutchinson, L. Peterson, M. Abbott, and S. O'Malley. Rpc in the x-kernel: Evaluating new design techniques. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 91–101, December 1989.

[111] K. Hwang and F. Briggs. *Computer Architecture and Parallel Processing*. Computer Science Series. McGraw-Hill, New York, 1984.

[112] H. Burkhardt III, S. Frank, B. Knobe, and J. Rothnie. Overview of the ksr1 computer system. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.

[113] Sun Microsystem Inc. *Sun OS 4.0 Reference Manual*, November 1987. Section 3L.

[114] Intel Corporation, Oregon. *Intel iPSC/2 and iPSC/860 User's Guide*, 1989.

[115] Intel Corporation, Beaverton, Oregon. *Touchstone Delta System User's Guide*, 1991.

[116] A. Jones, R. Chansler Jr., I. Durham, P. Feller, and K. Schwans. Software management of cm* – a distributed multiprocessor. In *Proceedings of the National Computer Conference*, pages 657–663, 1977.

[117] A. Jones and K. Schwans. Task forces: Distributed software for solving problems of substantial size. In *Proceedings of the Fourth International Conference on Software Engineering*, 1979.

[118] A.K. Jones, R.J. Chansler, I. Durham, P. Feiler, D. Scelza, K. Schwan, and S. Vegdahl. Programming issues raised by a multiprocessor. *Proceedings of the IEEE*, 66(2):229–237, Feb. 1978.

[119] A.K. Jones, R.J. Chansler, I. Durham, J. Mohan, K. Schwan, and S. Vegdahl. Staros, a multiprocessor operating system. In *Proceedings of the 7th Symposium on Operating System Principles, Asilomar, CA*, pages 117–127. Assoc. Comput. Mach., Dec.10-12 1979.

[120] Anita K. Jones and Peter Schwarz. Experience using multiprocessor systems: A status report. *Surveys of the Assoc. Comput. Mach.*, 12(2):121–166, June 1980.

[121] Eds. A.K. Jones and Ed Gehringer. The cm* multiprocessor project: A research review. Technical report, School of Computer Science, Carnegie-Mellon University, CMU-CS-80-131, July 1980.

[122] M. Jones and R. Rashid. Mach and matchmaker: Kernel and language support for object-oriented distributed systems. Technical Report CMU-CS-88-129, School of Computer Science, Carnegie Mellon University, September 1986.

[123] M. Jones, R. Rashid, and M. Thompson. Matchmaker: An interface specification language. In *Proceedings of the ACM Conference on Principles of Programming Languages*, January 1985.

[124] Richard Larowe Jr., Carla Ellis, and Laurence Kaplan. The robustness of numa memory management. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 137–151, October 1991.

[125] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[126] A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceeding of the thirteenth ACM symposium on operating systems principles*, pages 41–55, October 1991.

[127] J. Kepecs. Lightweight processes for unix implementation and application. In *Proceedings of the Proc. 1985 USENIX Summer Conference*, pages 299–308, 1985.

[128] Y. Khalidi and M. Nelson. An implementation of unix on an object oriented operating system. In *Proceedings of the 1993 Winter Usenix Conference*, San Diego, January 1993.

[129] Jeff Kramer and Jeff MaGee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436, April 1985.

[130] BBN Laboratories. *Butterfly(TM) Parallel Processor Overview*. BBN Computer Company, Cambridge, MA, 1st edition, June 1985.

[131] B. Lampson and D. Redell. Experiences with processes and monitors in mesa. *Communications of the ACM*, 23(2):105–117, 1980.

[132] B. Lampson and H. Sturgis. Reflections on an operating system design. *Communications of the ACM*, 19:25–65, 1976.

[133] H. Lauer and R. Needham. On the duality of operating system structures. *Operating Systems Review*, 13(2):3–19, February 1979.

[134] E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler, and S. Vestal. The architecture of the eden system. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 148–159, December 1981.

[135] E. Lazowska and M. Squillante. Using processor-cache affinity in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–43, February 1993.

[136] T. Leblanc, B. Marsh, and M. Scott. Memory management for large-scale numa multiprocessors. Technical Report TR 311, Department of Computer Science, University of Rochester, March 1989.

[137] T. Leblanc, J. Mellor-Crummey, N. Gafter, L. Crowl, and P. Dibble. The elmwood multiprocessor operating system. *Software - Practice and Experience*, 19(11):1029–1056, November 1989.

[138] T. J. Leblanc. Shared memory versus message-passing in a tightly-coupled multiprocessor: A case study. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 463–466, August 1986.

[139] Thomas J. LeBlanc and S.A. Friedberg. Hierarchical process composition in distributed operating systems. In *Proceedings of the 5th International Conference on Distributed Computing Systems, Denver, Colorado*, pages 26–34, May 1985.

[140] C. E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the 1992 ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[141] S. Leutenegger and M. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 226–36, May 1990.

[142] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the 5th Symposium on Operating System Principles, Austin, Texas*, Nov. 1975.

[143] H.M. Levy and R.H. Eckhouse. *Computer Programming and Architecture*. Digital Press, 1989.

[144] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Dept. of Computer Science, Yale University, 1986.

[145] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[146] J. Liedtke. Fast thread management and communication without continuations. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 213–21, April 1992.

[147] Bert Lindgren, Bobby Krupczak, Mostafa Ammar, and Karsten Schwan. An architecture and toolkit for parallel and configurable protocols. In *Proceedings of the International Conference on Network Protocols (ICNP-93)*, September 1993.

[148] B. Liskov. Abstraction mechanisms in clu. *Communications of the ACM*, 20(8):564–576, March 1977.

[149] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. on Prog. Lang. and Systems.*, 5(3):381–404, July 1983.

[150] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[151] M. Livny and U. Manber. Distributed computation via active messages. *IEEE Transactions on Computers*, C-34(12):1185–1190, Dec 1985.

[152] M. Livny and U. Manber. Active channels and their applications to parallel computing. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 367–369, August 1987.

[153] S. Lo and V. Gilgor. A comparative analysis of multiprocessor scheduling algorithms. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, September 1987.

[154] C. Douglas Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie Mellon University, 1986.

[155] P. Madany, R. Campbell, V. Russo, D. Leyens, and S. Cook. A class hierarchy for building stream-oriented file systems. In *Proceedings of the 1989 European Conference on Object-Oriented Programming (ECOOP '89)*, pages 311–28, July 1989.

[156] P. Madany, D. Leyens, V. Russo, and R. Campbell. A c++ class hierarchy for building unix-like file systems. In *Proceedings of the USENIX C++ Conference*, pages 65–79, October 1988.

[157] S. Majumdar, D. Eager, and R. Bunt. Scheduling in multiprogrammed parallel systems. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 104–13, May 1988.

[158] S. Majumdar, D. Eager, and R. Bunt. Characterisation of programs for scheduling in multiprogrammed parallel systems. *Performance Evaluation*, 13(2):109–30, October 1991.

[159] M. Makpangou, Y. Gourhant, and M. Shapiro. Boar: a library of fragmented object types for distributed abstractions. In *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, pages 164–8, October 1991.

[160] B. Marsh, C. Brown, T. Leblanc, M. Scott, T. Becker, C. Quiroz, P. Das, and J. Karlsson. The rochester checkers player: multimodel parallel programming for animate vision. *IEEE Computer*, 25(2):12–19, February 1992.

[161] B. Marsh, M. Scott, T. Leblanc, and E. Markatos. First-class user-level threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 110–21, October 1991.

[162] H. Massalin and C. Pu. Reimplementing the synthesis kernel on the sony news workstation. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 177–186, April 1992.

[163] Henry Massalin and Calton Pu. Threads and input/output in the synthesis kernel. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 191–201. SIGOPS, Assoc. Comput. Mach., Dec. 1989.

[164] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor scheduling pollicy for multiprogrammed, shared memory multiprocessors. Technical Report 90-03-02, Department of Computer Science and Engineering, University of Washington, March 1991.

[165] P. McJones and G. Swart. Evolving the unix system interface to support multithreaded programs. In *Proceedings of the USENIX Winter Conference*, pages 393–404, 1989.

[166] G. Mealy, B. Witt, and W. Clark. The functional structure of os/360. *IBM Systems Journal*, 5(1), January 1966.

[167] J. Mellor-Crummey, T. Leblanc, L. Crowl, N. Gafter, and P. Dibble. Elmwood - an object-oriented multiprocessor operating system. Technical Report BPR 20, Department of Computer Science, University of Rochester, September 1987.

[168] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

[169] J. Mogul and A. Borg. The effects of context switches on cache performance. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (SIGPLAN Notices vol.26, no.4)*, pages 75–84, April 1991.

[170] Joseph Mohan. *Performance of Parallel Programs: Model and Analyses*. PhD thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., July 1984.

[171] R. Moore, I. Naasi, and D. Siewiorek J. O'Neil. The encore multimax (tm): A multiprocessor computing environment. Technical Report ETR 86-004, Encore Computer Corporation, 1986.

[172] P. Muckelbauer and V. Russo. Distributed object interoperability via a network type system. In *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, pages 169–72, October 1991.

[173] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. In *Proceedings of the Sun User Group Technical Conference*, pages 101–112, June 1991.

[174] Bodhisattwa Mukherjee, Greg Eisenhauer, and Kaushik Ghosh. A machine independent interface for lightweight threads. Technical Report GIT-CC-93-53, College of Computing, Georgia Institute of Technology, August 1993. To be published in Operating System Review.

[175] Bodhisattwa Mukherjee and Karsten Schwan. Experimentation with a reconfigurable micro-kernel. In *Proc. of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 45–60, September 1993.

[176] Bodhisattwa Mukherjee and Karsten Schwan. Experiments with a configurable lock for multiprocessors. In *Proc. of the twenty secondth International Conference on Parallel Processing*, volume 2, pages 205–208, August 1993.

[177] Bodhisattwa Mukherjee and Karsten Schwan. Improving performance by use of adaptive objects: Experimentation with a configurable multiprocessor thread package. In *Proc. of the second International Symposium on High Performance Distributed Computing*, pages 59–66, July 1993.

[178] Bodhisattwa Mukherjee and Karsten Schwan. Survey of real-time operating systems. Technical Report GIT-CC-93/18, College of Computing, Georgia Institute of Technology, March 1993.

[179] R. Needham. The cambridge cap computer and its protection system. In *Proceedings of the 6th ACM Symposium on Operating Systems Principles*, pages 1–10, Purdue University, November 1977. Assoc. Comput. Mach., SIGOPS.

[180] B.J. Nelson. *Remote Procedure Call*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, May 1981.

[181] P. Oleinick. *The Implementation and Evaluation of Parallel Algorithms on a Multiprocessor*. PhD thesis, Carnegie-Mellon University, 1978.

[182] J. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of Distributed Computing Systems Conference*, pages 22–30, October 1982.

[183] J.K. Ousterhout. *Partitioning and Cooperation in a Distributed Operating System*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, April 1980.

[184] John K. Ousterhout, Donald A. Scelza, and Pradeep Sindhu. Medusa: An experiment in distributed operating system structure. *Comm. of the Assoc. Comput. Mach.*, 23(2):92–104, Feb. 1980.

[185] IEEE POSIX P1003.4a. *Threads Extension for Portable Operating Systems.*

[186] J. Pallas and D. Ungar. Multiprocessor smalltalk: a case study of a multiprocessor-based programming environment. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation (SIGPLAN Notices, vol.23, no.7)*, pages 268–77, Atlanta, GA, June 1988.

[187] K. Park and L. Dowdy. Dynamic partitioning of multiprocessor systems. *International Journal of Parallel Programming*, 18(2):91–120, April 1989.

[188] D. Patterson and J. Hennessy. *Computer architecture : a quantitative approach.* Morgan Kaufman Publishers, San Mateo, Calif., 1990.

[189] M. Powell, S. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. Sunos multi-thread architecture. In *Proceedings of the USENIX winter conference*, pages 1–14, 1991.

[190] R. Rashid. Threads of a new system. *Unix Review*, 4(8):37–49.

[191] R. Rashid. From rig to accent to mach: The evolution of a network operating system. In *Proceedings of the ACM/IEEE Computer Society Fall Joint Computer Conference*, pages 1128–37, November 1986.

[192] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, and R. Sanzi. Mach: A foundation for open systems. In *Proceedings of the 2nd Workshop on Workstation Operating Systems,IEEE*, pages 109–13, September 1989.

[193] R. Rashid and G. Robertson. Accent: A communnication oriented network operating system kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 64–75, December 1981.

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

[195] Kendall Square Research. Technical summary, 1992.

[196] D. Ritchie and K. Thompson. The unix time-sharing system. *Communications of the Assoc. Comput. Mach.*, 17(7), 1974.

[197] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the chorus operating system. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–69, April 1992.

[198] V. Russo. Object-oriented operating system design. *TCOS Newsletter*, 5(1):34–38, 1991.

[199] V. Russo and R. Campbell. Virtual memory and backing storage management in multiprocessor operating systems using object-oriented design techniques. In *OOPSLA'89 Conference Proceedings (SIGPLAN Notices vol.24, no.10)*, pages 267–78, October 1989.

[200] V. Russo, G. Johnston, and R. Campbell. Process management and exception handling in multiprocessor operating systems using object-oriented design techniques. In *Proceedings of the 3rd Annual Conference on Object-Orientated Programming Systems, Languages, and Applications (OOPSLA 88, SIGPLAN Notices, vol.23, no.11)*, pages 248–58, September 1988.

[201] V. Russo, P. Madany, and R. Campbell. C++ and operating systems performance: a case study. In *Proceedings of the USENIX C++ Conference*, pages 103–14, April 1990.

[202] V. Russo and P. Muckelbauer. Process scheduling and synchronization in the renaissance object-oriented multiprocessor operating system. In *Proceedings of the 2nd Usenix Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS II)*, pages 117–32, March 1991.

[203] V. F. Russo. *An Object-Oriented Operating System*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1991.

[204] P. Sadayappan and F. Ercal. Nearest-neighbor mapping of finite element graphs onto processors meshes. *IEEE Transactions On Computers*, C-36(12):1408–1420, Dec 1987.

[205] J. Salmon and S. Callahan. Moose: A multitasking os for hypercubes. In *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications, Pasadena, CA*, pages 391–396. ACM, JPL, Jan. 1988.

[206] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[207] M. Schroeder and M. Burrows. Performance of firefly rpc. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[208] P. Schuller, H. Hartig, W. Kuhnhauser, and H. Streich. Performance of the birlix operating system. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 147–160, April 1992.

[209] Karsten Schwan, Thomas E. Bihari, and Ben Blake. Adaptive, reliable software for distributed and parallel, real-time systems. In *Proceedings of the Sixth Symposium on Reliability in Distributed Software, Williamsburg, Virginia*, pages 32–44. IEEE, March 1987.

[210] Karsten Schwan, Tom Bihari, Bruce W. Weide, and Gregor Taulbee. High-performance operating system primitives for robotics and real-time control systems. *ACM Transactions on Computer Systems*, 5(3):189–231, Aug. 1987.

[211] Karsten Schwan and Win Bo. Topologies – distributed objects on multicomputers. *ACM Transactions on Computer Systems*, 8(2):111–157, May 1990.

[212] Karsten Schwan, Harold Forbes, Ahmed Gheith, Bodhisattwa Mukherjee, and Yiannis Samiotakis. A cthread library for multiprocessors. Technical Report GIT-ICS-91/02, College of Computing, Georgia Institute of Technology, 1991.

[213] Karsten Schwan, Prabha Gopinath, and Win Bo. Chaos – kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, C-36(8):904–916, July 1987.

[214] Karsten Schwan and Anita K. Jones. Specifying resource allocation for the cm* multiprocessor. *IEEE Software*, 3(3):60–70, May 1984.

[215] Karsten Schwan and Rajiv Ramnath. Adaptable operating software for manufacturing systems and robots: A computer science research agenda. In *Proceedings of the 5th Real-Time Systems Symposium, Austin, Texas*, pages 255–262. IEEE, Dec. 1984.

[216] Karsten Schwan, Hongyi Zhou, and Ahmed Gheith. Multiprocessor real-time threads. *Operating Systems Review*, 25(4):35–46, Oct. 1991. Also appears in the Jan. 1992 issue of Operating Systems Review.

[217] M. Scott, T. Leblanc, and B. Marsh. Design rationale for psyche, a general purpose multiprocessor operating system. In *Proceedings of the 1988 International Conference on Parallel Processing (V II - Software)*, pages 255–262, August 1988.

[218] M. Scott, T. Leblanc, and B. Marsh. Evolution of an operating system for large scale shared-memory multiprocessors. Technical Report TR 309, Department of Computer Science, University of Rochester, March 1989.

[219] M. Scott, T. Leblanc, and B. Marsh. Multi-model parallel programming in psyche. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 70–78, March 1990.

[220] M. Scott, T. Leblanc, B. Marsh, T. Becker, C. Dubnicki, E. Markatos, and N. Smithline. Implementation issues for the psyche multiprocessor operating system. *Computing Systems*, 3(1):101–137, Winter 1990.

[221] C. Seitz. Reactive kernel. In *Proceedings of the 3rd Conf. On Hypercube Concurrent Computers and Applications, Pasadena, CA*, pages 1520–1528. ACM, Jan. 1988.

[222] Charles Seitz and William Athas. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, 21(8):9–24, August 1988.

[223] Sequent Computer Systems, Inc. *Dynix Programmer's Manual*, 1986.

[224] Sequent Computer Systems, Inc. *Symmetry Technical Summary, Rev 1.4*, 1987.

[225] K. Sevcik. Characterizations of parallelism in applications and their use in scheduling. In *Proceedings of the 1989 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 171–80, May 1989.

[226] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the Sixth International Conference on Distributed Computing Systems, Boston, Mass.*, pages 198–204. IEEE, May 1986.

[227] M. Shapiro. Object-support operating systems. *TCOS Newsletter*, 5(1):39–42, 1991.

[228] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. Sos: An object-oriented operating system – assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[229] M. Shapiro and M. Makpangou. Distributed abstractions, lightweight references. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 263–7, April 1992.

[230] M.H. Solomon and R.A. Finkel. The roscoe distributed operating system. In *Proceedings of the 7th Symposium on Operating System Principles, Asilomar, CA*, pages 108–114. Assoc. Comput. Mach., Dec.10-12 1979.

[231] A. Spector, D. Daniels, D. Duchamp, J. Eppinger, and R. Pausch. Distributed transactions for reliable systems. In *Proceedings of the Eleventh ACM Symposium on Operation Systems Principles*, pages 127–146. ACM SIGOPS, Nov. 1987.

[232] M. Squillante and R. Nelson. Analysis of task migration in shared-memory multiprocessor scheduling. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 143–55, May 1991.

[233] Harold S. Stone. *High-performance computer architecture*. Addison-Wesley Pub. Co., Reading, Mass., 1987.

[234] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 23(5):55–64, May 1990.

[235] H. Sturgis. A postmortem for a time sharing system. Technical Report CSL-1974-001, Xerox, 1974.

[236] SUN. *The SPARC Architecture Manual*. Sun Microsystems Inc., No. 800-199-12, Version 8, January 1991.

[237] Liba Svobodova. Resilient distributed computing. *IEEE Transactions on Software Engineering*, pages 257–267, May 1984.

[238] R. J. Swan, S. H. Fuller, and D. P. Siewiorek. Cm*: A modular, multi-microprocessor. In *Proceedings of the National Computer Conference*, pages 637–644. Assoc. Comput. Mach., 1977.

[239] A. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[240] A. Tanenbaum. *Modern operating systems*. Prentice-Hall, Englewood Cliffs, N.J., 2nd edition, 1992.

[241] A. Tanenbaum and R. Van Renesse. Distributed operating systems. *Computing Surveys*, 17(4):419–470, December 1985.

[242] A. Tevanian and R. Rashid. Mach: A basis for future unix development. Technical Report CMU-CS-87-139, School of Computer Science, Carnegie-Mellon University, June 1987.

[243] A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M. Young. Mach threads and the unix kernel: The battle for control. In *Proceedings of the Summer 1987 USENIX Conference*, pages 185–97, June 1987.

[244] A. Tevanian, R. Rashid, M. Young, D. Golub, M. Thompson, W. Bolosky, and R. Sanzi. A unix interface for shared memory and memory mapped files under mach. In *Proceedings of the Summer 1987 USENIX Conference*, pages 53–67, June 1987.

[245] D. Thiebaut and H. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, November 1987.

[246] Thinking Machines Corporation, Cambridge, Massachusetts. *The Connection Machine CM-5 Technical Summary*, October 1991.

[247] R. Thomas and W. Crowther. The uniform system: An approach to runtime support for large scale shared memory parallel processors. In *Proceedings of the 1988 International Conference on Parallel Processing, V. II – Software*, pages 245–254, August 1988.

[248] J. Torrellas, A. Tucker, and A. Gupta. Benefits of cache-affinity scheduling in shared-memory multiprocessors. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 272–4, May 1993.

[249] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159–166, December 1989.

[250] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the thirteenth ACM Symposium on Operating Systems Principles*, pages 26–40, October 1991.

[251] J. Walpole, J. Inouye, and R. Konuru. Modularity and interfaces in micro-kernel design and implementation: A case study of chorus on the hp pa-risc. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 71–82, April 1992.

[252] M. Wehner, J. Ambrosiano, J. Brown, W. Dannevik, P. Eltgroth, and A. Mirin. Toward a high performance distributed memory climate model. In *Proceedings of the 2nd International Symposium on High Performance Distributed Computing*, pages 102–113, July 1993.

[253] B.W. Weide, M.E. Brown, J.A.S. Alegria, and G.R. Meyer. A graphical interconnection language and its application to concurrent and real- time programming. In *Proceedings of the 20th Annual Allerton Conf. on Comm, Control, and Comp., Univ. of Ill.*, pages 567–576. IEEE, Oct. 1982.

[254] M. Weiser, A. Demers, and C. Hauser. The portable common run-time approach to interoperability. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 114–122, December 1989.

[255] J. Wendorf. *Operating System/Application Concurrency in Tightly Coupled Multiprocessor Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, December 1987. Technical Report CMU-CS-88-117.

[256] W. Wulf. Reliable hardware/software architecture. *IEEE Transactions on Software Engineering*, SE-1(2):233–240, June 1975.

[257] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

[258] W. Wulf, R. Levin, and S. Harbison. *Hydra/C.mmp: An Experimental Computer System*. McGraw-Hill Advanced Computer Science Series, 1981.

[259] W. Wulf, R. Levin, and C. Pierson. Overview of the hydra operating system. In *Proceedings of the 5th Symposium on Operating System Principles, Austin, Texas*, pages 122–131. ACM, Nov. 1975.

[260] G. Yaoqing and Y. Kwong. A survey of implementations of concurrent, parallel and distributed smalltalk. *SIGPLAN Notices*, 28(9):29–35, September 1993.

[261] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

[262] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocesor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SIGOPS Notices vol.21, no.5)*, pages 63–76, November 1987.

[263] M. W. Young. *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.* PhD thesis, School of Computer Science, Carnegie Mellon University, November 1989. Technical Report CMU-CS-89-202.

[264] Tse yun Feng. A survey of interconnection networks. *IEEE Computer*, pages 12–27, December 1981.

[265] J. Zahorjan and E. Lazowska amd D. Eager. Spinning versus blocking in parallel systems with uncertainty. In *Proceedings of the International Symposium on Performance of Distributed and Parallel Systems*, December 1988.

[266] J. zahorjan, E. Lazowska, and D. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–98, April 1991.

[267] J. Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–225, May 1990.

[268] Wei Zhao, Krithi Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, C-36(8):949–960, August 1987.

[269] Hongyi Zhou and Karsten Schwan. Dynamic scheduling for hard real-time systems: Toward real-time threads. In *Proceedings of the Joint IEEE Workshop on Real-Time Operating Systems and Software and IFAC Workshop on Real-Time Programming, Atlanta, GA*. IEEE, May 1991.

[270] Hongyi Zhou, Karsten Schwan, and Ian Akyildiz. Performance effects of information sharing in a distributed multiprocessor real-time scheduler. Technical report, College of Computing, Georgia Tech, GIT-CC-91/40, Sept. 1991. Abbreviated version in Proceedings of the 1992 IEEE Real-Time Systems Symposium, Phoenix.

[271] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):540–54, september 1992.