11/17/00

UNIX

William Stallings

This document is an extract from
Operating Systems: Internals and Design Principles, Fourth Edition
Prentice Hall, 2000, ISBN 0-13-031999-6
It is available at WilliamStallings.com/OS4e.html

2.6 TRADITIONAL UNIX SYSTEMS....4
History....4
Description....4
2.7 MODERN UNIX SYSTEMS....6
System V Release 4 (SVR4)....6
Solaris 2.x....6
4.4BSD....6
Linux....6
History....6
Modular Structure....7
3.4 UNIX SVR4 PROCESS MANAGEMENT....9
Process States....9
Process Description....9
Process Control....10
4.5 SOLARIS THREAD AND SMP MANAGEMENT....12
Multithreaded Architecture....12
Motivation....12
Process Structure....13
Thread Execution....13
Interrupts as Threads....14
4.6 LINUX PROCESS AND THREAD MANAGEMENT....16
Linux Processes....16
Linux Threads....16
6.7 UNIX CONCURRENCY MECHANISMS....17
Pipes....17
Messages....17
Shared Memory....17
Semaphores....17
Signals....18
6.8 SOLARIS THREAD SYNCHRONIZATION PRIMITIVES....19
Mutual Exclusion Lock....19
Semaphores....19
Readers/Writer Lock....19
Condition Variables....20
8.3 UNIX AND SOLARIS MEMORY MANAGEMENT....21
Paging System....21
Data Structures....21
Page Replacement....21
Kernel Memory Allocator....22
8.4 LINUX MEMORY MANAGEMENT....24
Linux Virtual Memory....24
Virtual Memory Addressing....24
Page Allocation....24
Page Replacement Algorithm....24
Kernel Memory Allocation....24
9.3 TRADITIONAL UNIX SCHEDULING....26
10.3 LINUX SCHEDULING....28
10.4 UNIX SVR4 SCHEDULING....29
11.8 UNIX SVR4 I/O....30
Buffer Cache....30
Character Queue....30
Unbuffered I/O....31
UNIX Devices....31

12.7 UNIX FILE MANAGEMENT 32
Inodes..32
File Allocation...32
13.6 SUN CLUSTER...34
Object and Communication Support...34
Process Management..34
Networking..34
Global File System..35
13.7 BEOWULF AND LINUX CLUSTERS 36
Beowulf Features 36
Beowulf Software 36

2.6 TRADITIONAL UNIX SYSTEMS

History

The history of UNIX is an oft-told tale and will not be repeated in great detail here. Instead, a brief summary is provided; highlights are depicted in Figure 2.14, which is based on a figure in [SALU94].[1]

UNIX was initially developed at Bell Labs and became operational on a PDP-7 in 1970. Some of the people involved at Bell Labs had also participated in the time-sharing work being done at MIT's Project MAC. That project led to the development of first CTSS and then Multics. Although it is common to say that UNIX is a scaled-down version of Multics, the developers of UNIX actually claimed to be more influenced by CTSS [RITC78b]. Nevertheless, UNIX incorporated many ideas from Multics.

Work on UNIX at Bell Labs, and later elsewhere, produced a series of versions of UNIX. The first notable milestone was porting the UNIX system from the PDP-7 to the PDP-11. This was the first hint that UNIX would be an operating system for all computers. The next important milestone was the rewriting of UNIX in the programming language C. This was an unheard-of strategy at the time. It was generally felt that something as complex as an operating system, which must deal with time-critical events, had to be written exclusively in assembly language. The C implementation demonstrated the advantages of using a high-level language for most if not all of the system code. Today, virtually all UNIX implementations are written in C.

These early versions of UNIX were quite popular within Bell Labs. In 1974, the UNIX system was described in a technical journal for the first time [RITC74]. This spurred great interest in the system. Licenses for UNIX were provided to commercial institutions as well as universities. The first widely available version outside Bell Labs was Version 6, in 1976. The follow-on Version 7, released in 1978, is the ancestor of most modern UNIX systems. The most important of the non-AT&T systems to be developed was done at the University of California at Berkeley, called UNIX BSD, running first on PDP and then VAX machines. AT&T continued to develop and refine the system. By 1982, Bell Labs had combined several AT&T variants of UNIX into a single system, marketed commercially as UNIX System III. A number of features were later added to the operating system to produce UNIX System V.

Description

Figure 2.15 provides a general description of the UNIX architecture. The underlying hardware is surrounded by the operating-system software. The operating system is often called the system kernel, or simply the kernel, to emphasize its isolation from the user and applications. This portion of UNIX is what we will be concerned with in our use of UNIX as an example in this book. However, UNIX comes equipped with a number of user services and interfaces that are considered part of the system. These can be grouped into the shell, other interface software, and the components of the C compiler (compiler, assembler, loader). The layer outside of this consists of user applications and the user interface to the C compiler.

A closer look at the kernel is provided in Figure 2.16. User programs can invoke operating-system services either directly or through library programs. The system call interface is the boundary with the user and allows higher-level software to gain access to specific kernel functions. At the other end, the operating system contains primitive routines that interact directly with the hardware. Between these two interfaces, the system is divided into two main parts, one concerned with process control and the other concerned with file management and I/O. The process control subsystem is responsible for memory management, the scheduling and dispatching of processes, and the synchronization and interprocess communication of processes. The file system exchanges data between memory and external devices either as a stream of

[1] A more complete family tree is presented in [MCKU96].

characters or in blocks. To achieve this, a variety of device drivers are used. For block-oriented transfers, a disk cache approach is used: a system buffer in main memory is interposed between the user address space and the external device.

The description in this subsection has dealt with what might be termed traditional UNIX systems; [VAHA96] uses this term to refer to System V Release 3 (SVR3), 4.3BSD, and earlier versions. The following general statements may be made about a traditional UNIX system. It is designed to run on a single processor and lacks the ability to protect its data structures from concurrent access by multiple processors. Its kernel is not very versatile, supporting a single type of file system, process scheduling policy, and executable file format. The traditional UNIX kernel is not designed to be extensible and has few facilities for code reuse. The result is that, as new features were added to the various UNIX versions, much new code had to be added, yielding a bloated and unmodular kernel.

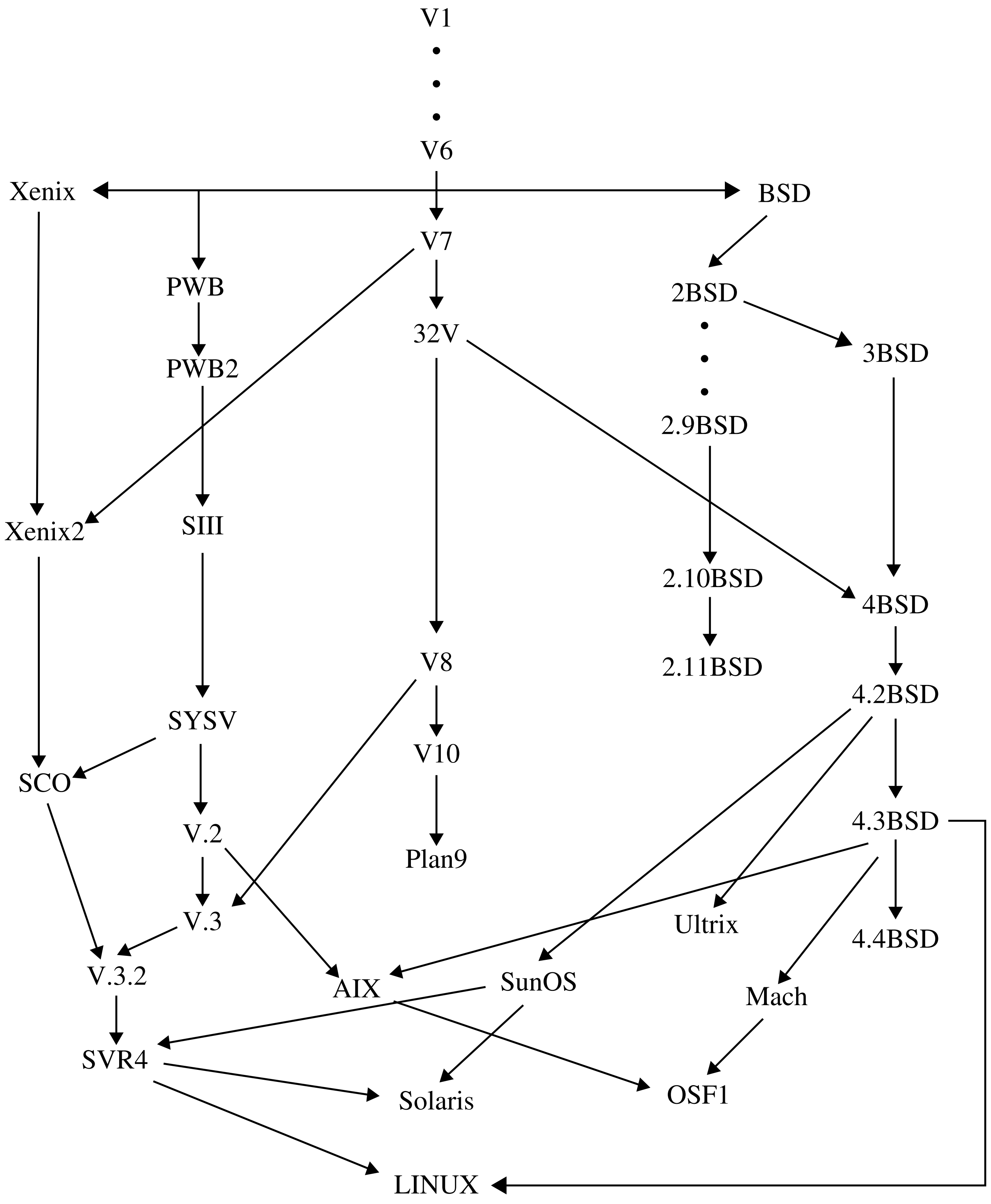


Figure 2.14 UNIX History

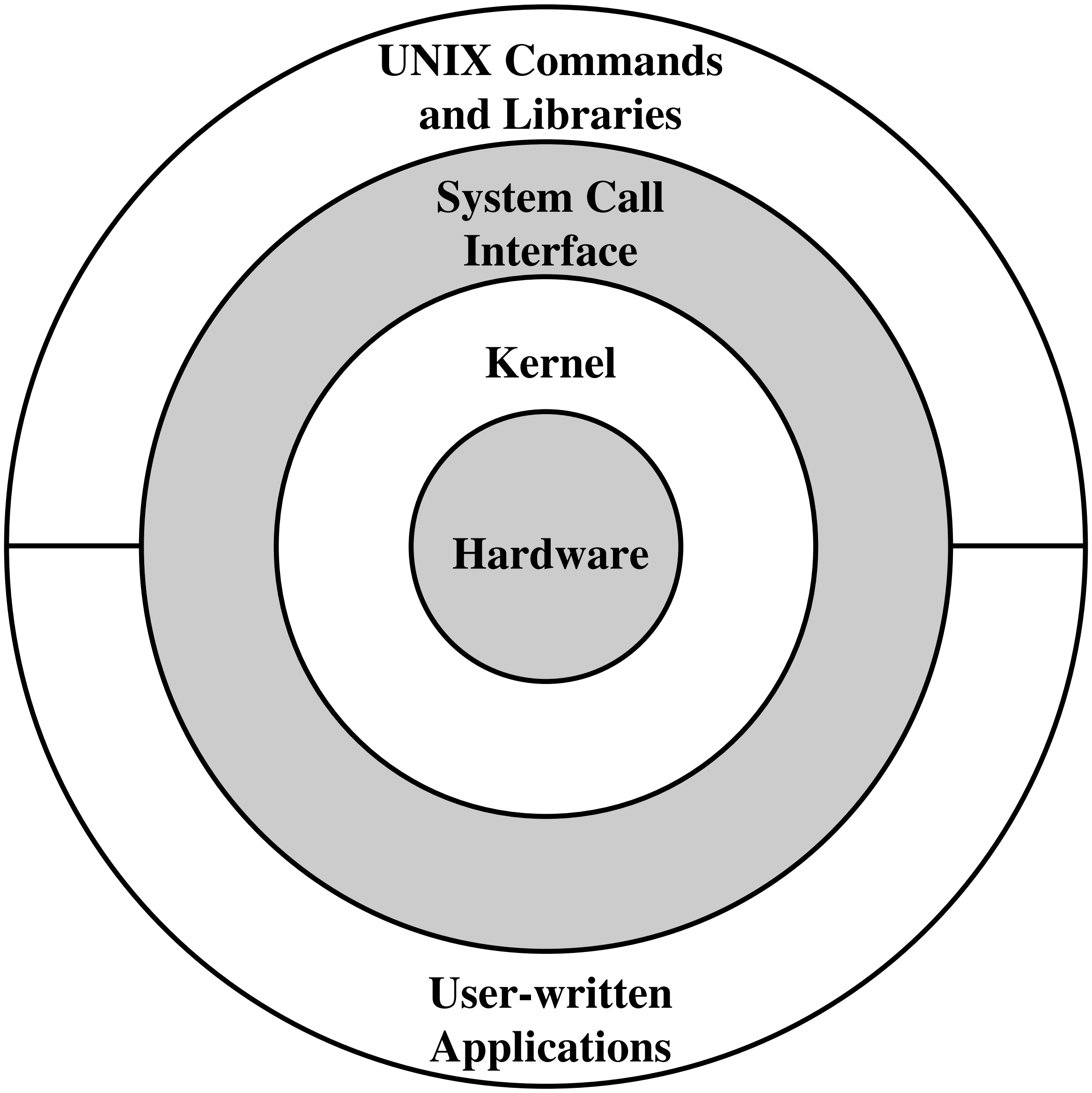


Figure 2.15 General UNIX Architecture

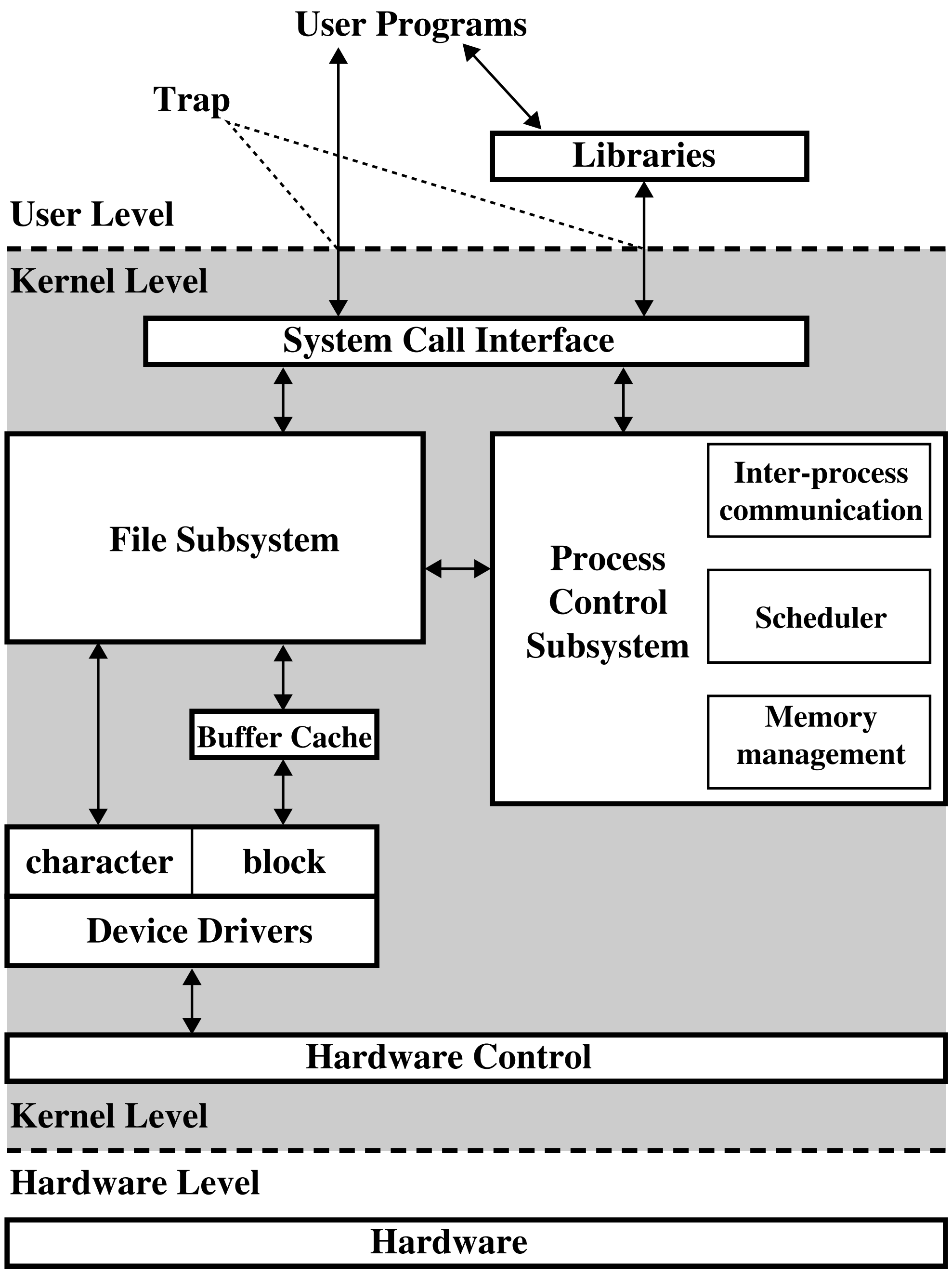


Figure 2.16 Traditional UNIX Kernel [BACH86]

2.7 MODERN UNIX SYSTEMS

As UNIX evolved, the number of different implementations proliferated, each providing some useful features. There was a need to produce a new implementation that unified many of the important innovations, added other modern OS-design features, and produced a more modular architecture. Typical of the modern UNIX kernel is the architecture depicted in Figure 2.17. There is a small core of facilities, written in a modular fashion, that provide functions and services needed by a number of OS processes. Each of the outer circles represents functions and an interface that may be implemented in a variety of ways.

We now turn to some examples of modern UNIX systems.

System V Release 4 (SVR4)

SVR4, developed jointly by AT&T and Sun Microsystems, combines features from SVR3, 4.3BSD, Microsoft Xenix System V, and SunOS. It was almost a total rewrite of the System V kernel and produced a clean, if complex, implementation. New features in the release include real-time processing support, process scheduling classes, dynamically allocated data structures, virtual memory management, virtual file system, and a preemptive kernel.

SVR4 draws on the efforts of both commercial and academic designers and was developed to provide a uniform platform for commercial UNIX deployment. It has succeeded in this objective and is perhaps the most important UNIX variant extant. It incorporates most of the important features ever developed on any UNIX system, and does so in an integrated, commercially viable fashion. SVR4 is running on machines ranging from 32-bit microprocessors up to supercomputers and is one of the most important operating systems ever developed. Many of the UNIX examples in this book are from SVR4.

Solaris 2.x

Solaris is Sun's SVR4-based UNIX release, with the latest version being 2.8. The version 2 Solaris implementations provide all of the features of SVR4 plus a number of more advanced features, such as a fully preemptable, multithreaded kernel, full support for SMP, and an object-oriented interface to file systems. Solaris is the most widely used and most successful commercial UNIX implementation. For some OS features, Solaris provides the UNIX examples in this book.

4.4BSD

The Berkeley Software Distribution BSD series of UNIX releases have played a key role in the development of OS design theory. 4.xBSD is widely used in academic installations and has served as the basis of a number of commercial UNIX products. It is probably safe to say that BSD is responsible for much of the popularity of UNIX and that most enhancements to UNIX first appeared in BSD versions.

4.4BSD is the final version of BSD to be released by Berkeley, with the design and implementation organization subsequently dissolved. It is a major upgrade to 4.3BSD and includes a new virtual memory system, changes in the kernel structure, and a long list of other feature enhancements.

Linux

History

Linux started out as a UNIX variant for the IBM PC architecture. The initial version was written by Linus Torvalds, a Finnish student of computer science. Torvalds posted an early version of Linux on the Internet in 1991. Since then, a number of people, collaborating over the Internet, have contributed to the development of Linux, all under the control of Torvalds. Because Linux is free and the source code is available, it became an early alternative to other UNIX workstations, such as those offered by Sun Microsystems, Digital Equipment Corp (now

Compaq), and Silicon Graphics. Today, Linux is a full-featured UNIX system that runs on all of these platforms and more.

Key to the success of Linux has been its character as a free package available under the auspices of the Free Software Foundation (FSF). FSF's goal is stable, platform-independent software that is free, high quality, and embraced by the user community. FSF's GNU project provides tools for software developers, and the GNU Public License (GPL) is the FSF seal of approval. Torvalds used GNU tools in developing his kernel, which he then released under the GPL. Thus, the Linux distributions that you see today are the product of FSF's GNU project, Torvald's individual effort, and many collaborators all over the world.

In addition to its use by many individual programmers, Linux has now made significant penetration into the corporate world [MANC00]. This is not primarily because of the free software, but because of the quality of the Linux kernel. Many talented programmers have contributed to the current version, resulting in a technically impressive product. Moreover, Linux is highly modular and easily configured. This makes it easy to squeeze optimal performance from a variety of hardware platforms. Plus, with the source code available, vendors can tweak applications and utilities to meet specific requirements. Throughout this book, we will provide details of Linux kernel internals.

Modular Structure

Most UNIX kernels are monolithic. Recall that a monolithic kernel is one that includes virtually all of the operating-system functionality in one large block of code that runs as a single process with a single address space. All the functional components of the kernel have access to all of its internal data structures and routines. If changes are made to any portion of a typical monolithic operating system, all the modules and routines must be relinked and reinstalled and the system rebooted before the changes can take effect. As a result, any modification, such as adding a new device driver or file system function, is difficult. This problem is especially acute for Linux, for which development is global and done by a loosely associated group of independent programmers.

To address this problem, Linux is organized as a collection of relatively independent blocks referred to as **loadable modules** [GOYE99]. The Linux loadable modules have two important characteristics:

- **Dynamic linking:** A kernel module can be loaded and linked into the kernel while the kernel is already in memory and executing. A module can also be unlinked and removed from memory at any time.
- **Stackable modules:** The modules are arranged in a hierarchy. Individual modules server as libraries when they are referenced by client modules higher up in the hierarchy and as clients when they reference modules further down.

Dynamic linking [FRAN97] eases the task of configuration and saves kernel memory. In Linux, a user program or user can explicitly load and unload kernel modules using the `insmod` and `rmmod` commands. The kernel itself monitors the need for particular functions and can load and unload modules as needed. With stackable modules, dependencies between modules can be defined. This has two benefits:

1. Code common to a set of similar modules (e.g., drivers for similar hardware) can be moved into a single module, reducing replication.
2. The kernel can make sure that needed modules are present, refraining from unloading a module on which other running modules depend, and loading any addition required modules when a new module is loaded.

Figure 2.18 is an example that illustrates the structures used by Linux to manage modules. The figure shows the list of kernel modules after only two modules have been loaded: FAT and

VFAT. Each module is defined by two tables, the module table and the symbol table. The module table includes the following elements:

- next: Pointer to the following module. All modules are organized into a linked list. The list begins with a pseudomodule (not shown in Figure 2.18).
- ref: List of modules that use this module.
- symtab: Pointer to this module's symbol table.
- name: Module name.
- size: Module size in memory pages.
- addr: Starting address of module.
- state: Current state of module.
- *cleanup(): Points to routine launched at module unloading.

The symbol table defines those symbols controlled by this module that are used elsewhere. It includes the following elements:

- size: Total table size.
- n_symbols: Number of symbols.
- n_refs: Number of references.
- symbols: Table of symbols.
- references: List of modules dependent on this module.

Figure 2.18 shows that the VFAT module was loaded after the FAT module and that the VFAT module is dependent on the FAT module.

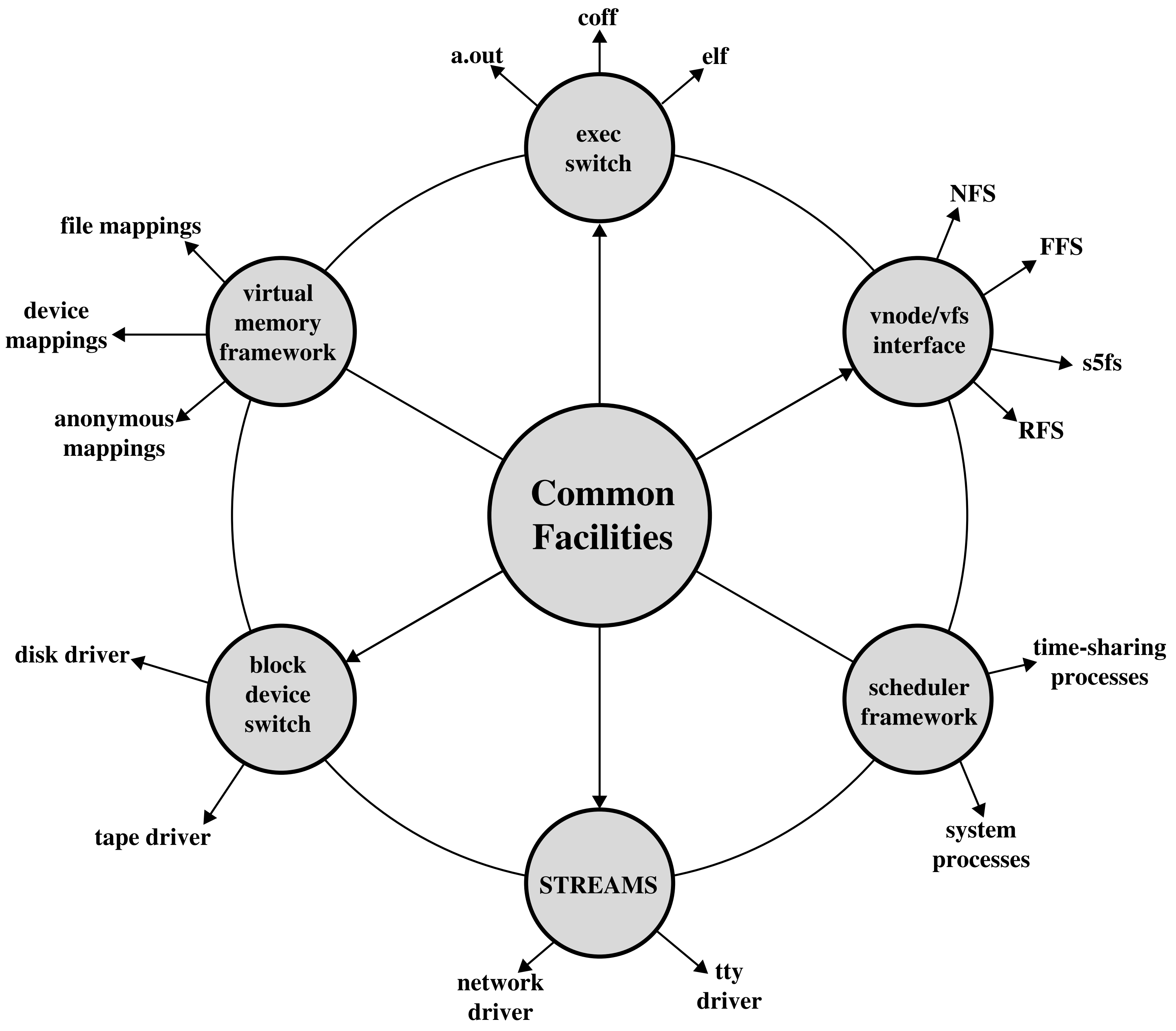


Figure 2.17 Modern UNIX Kernel [VAHA96]

3.4 UNIX SVR4 PROCESS MANAGEMENT

UNIX System V makes use of a simple but powerful process facility that is highly visible to the user. UNIX follows the model of Figure 3.14b, in which most of the operating system executes within the environment of a user process. Thus, two modes, user and kernel, are required. UNIX uses two categories of processes: system processes and user processes. System processes run in kernel mode and execute operating system code to perform administrative and housekeeping functions, such as allocation of memory and process swapping. User processes operate in user mode to execute user programs and utilities and in kernel mode to execute instructions belong to the kernel. A user process enters kernel mode by issuing a system call, when an exception (fault) is generated or when an interrupt occurs.

Process States

A total of nine process states are recognized by the UNIX operating system; these are listed in Table 3.9 and a state transition diagram is shown in Figure 3.16 (based on figure in [BACH86]). This figure is similar to Figure 3.7, with the two UNIX sleeping states corresponding to the two blocked states. The differences can be summarized quickly:

- UNIX employs two Running states to indicate whether the process is executing in user mode or kernel mode.
- A distinction is made between the two states: (Ready to Run, in Memory) and (Preempted). These are essentially the same state, as indicated by the dotted line joining them. The distinction is made to emphasize the way in which the preempted state is entered. When a process is running in kernel mode (as a result of a supervisor call, clock interrupt, or I/O interrupt), there will come a time when the kernel has completed its work and is ready to return control to the user program. At this point, the kernel may decide to preempt the current process in favor of one that is ready and of higher priority. In that case, the current process moves to the preempted state. However, for purposes of dispatching, those processes in the preempted state and those in the Ready to Run, in Memory state form one queue.

Preemption can only occur when a process is about to move from kernel mode to user mode. While a process is running in kernel mode, it may not be preempted. This makes UNIX unsuitable for real-time processing. A discussion of the requirements for real-time processing is provided in Chapter 10.

Two processes are unique in UNIX. Process 0 is a special process that is created when the system boots; in effect, it is predefined as a data structure loaded at boot time. It is the swapper process. In addition, process 0 spawns process 1, referred to as the init process; all other processes in the system have process 1 as an ancestor. When a new interactive user logs onto the system, it is process 1 that creates a user process for that user. Subsequently, the user process can create child processes in a branching tree, so that any particular application can consist of a number of related processes.

Process Description

A process in UNIX is a rather complex set of data structures that provide the operating system with all of the information necessary to manage and dispatch processes. Table 3.10 summarizes the elements of the process image, which are organized into three parts: user-level context, register context, and system-level context.

The **user-level context** contains the basic elements of a user's program and can be generated directly from a compiled object file. The user's program is separated into text and data areas; the text area is read-only and is intended to hold the program's instructions. While the process is executing, the processor uses the user stack area for procedure calls and returns and parameter passing. The shared memory area is a data area that is shared with other processes. There is only one physical copy of a shared memory area, but, by the use of virtual memory, it appears to each sharing process that the shared memory region is in its address space. When a process is not running, the processor status information is stored in the **register context** area.

The **system-level context** contains the remaining information that the operating system needs to manage the process. It consists of a static part, which is fixed in size and stays with a process throughout its lifetime, and a dynamic part, which varies in size through the life of the process. One element of the static part is the process table entry. This is actually part of the process table maintained by the operating system, with one entry per process. The process table entry contains process control information that is accessible to the kernel at all times; hence, in a virtual memory system, all process table entries are maintained in main memory. Table 3.11 lists the contents of a process table entry. The user area, or U area, contains additional process control information that is needed by the kernel when it is executing in the context of this process; it is also used when paging processes to and from memory. Table 3.12 shows the contents of this table.

The distinction between the process table entry and the U area reflects the fact that the UNIX kernel always executes in the context of some process. Much of the time, the kernel will be dealing with the concerns of that process. However, some of the time, such as when the kernel is performing a scheduling algorithm preparatory to dispatching another process, it will need access to information about other processes.

The third static portion of the system-level context is the per process region table, which is used by the memory management system. Finally, the kernel stack is the dynamic portion of the system-level context. This stack is used when the process is executing in kernel mode and contains the information that must be saved and restored as procedure calls and interrupts occur.

Process Control

Process creation in UNIX is made by means of the kernel system call, fork( ). When a process issues a fork request, the operating system performs the following functions [BACH86]:

1. It allocates a slot in the process table for the new process.
2. It assigns a unique process ID to the child process.
3. It makes a copy of the process image of the parent, with the exception of any shared memory.
4. It increments counters for any files owned by the parent, to reflect that an additional process now also owns those files.
5. It assigns the child process to a Ready to Run state.
6. It returns the ID number of the child to the parent process, and a 0 value to the child process.

All of this work is accomplished in kernel mode in the parent process. When the kernel has completed these functions it can do one of the following, as part of the dispatcher routine:

1. Stay in the parent process. Control returns to user mode at the point of the fork call of the parent.
2. Transfer control to the child process. The child process begins executing at the same point in the code as the parent, namely at the return from the fork call.
3. Transfer control to another process. Both parent and child are left in the Ready to Run state.

It is perhaps difficult to visualize this method of process creation because both parent and child are executing the same passage of code. The difference is this: when the return from the fork occurs, the return parameter is tested. If the value is zero, then this is the child process, and a branch can be executed to the appropriate user program to continue execution. If the value is nonzero, then this is the parent process, and the main line of execution can continue.

Table 3.9 UNIX Process States

User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run, in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Preempted	Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect.

Table 3.10 UNIX Process Image

User-Level Context	
Process Text	Executable machine instructions of the program
Process Data	Data accessible by the program of this process
User Stack	Contains the arguments, local variables, and pointers for functions executing in user mode
Shared Memory	Memory shared with other processes, used for interprocess communication
Register Context	
Program Counter	Address of next instruction to be executed; may be in kernel or user memory space of this process
Processor Status Register	Contains the hardware status at the time of preemption; contents and format are hardware dependent
Stack Pointer	Points to the top of the kernel or user stack, depending on the mode of operation at the time or preemption
General-Purpose Registers	Hardware dependent
System-Level Context	
Process Table Entry	Defines state of a process; this information is always accessible to the operating system
U (user) Area	Process control information that needs to be accessed only in the context of the process
Per Process Region Table	Defines the mapping from virtual to physical addresses; also contains a permission field that indicates the type of access allowed the process: read-only, read-write, or read-execute
Kernel Stack	Contains the stack frame of kernel procedures as the process executes in kernel mode

Table 3.11 UNIX Process Table Entry

Process Status	Current state of process.
Pointers	To U area and process memory area (text, data, stack).
Process Size	Enables the operating system to know how much space to allocate the process.
User Identifiers	The **real user ID** identifies the user who is responsible for the running process. The **effective user ID** may be used by a process to gain temporary privileges associated with a particular program; while that program is being executed as part of the process, the process operates with the effective user ID.
Process Identifiers	ID of this process; ID of parent process. These are set up when the process enters the Created state during the fork system call.
Event Descriptor	Valid when a process is in a sleeping state; when the event occurs, the process is transferred to a ready-to-run state.
Priority	Used for process scheduling.
Signal	Enumerates signals sent to a process but not yet handled.
Timers	Include process execution time, kernel resource utilization, and user-set timer used to send alarm signal to a process.
P_link	Pointer to the next link in the ready queue (valid if process is ready to execute).
Memory Status	Indicates whether process image is in main memory or swapped out. If it is in memory, this field also indicates whether it may be swapped out or is temporarily locked into main memory.

Table 3.12 UNIX U Area

Process Table Pointer	Indicates entry that corresponds to the U area.
User Identifiers	Real and effective user IDs. Used to determine user privileges.
Timers	Record time that the process (and its descendants) spent executing in user mode and in kernel mode.
Signal-Handler Array	For each type of signal defined in the system, indicates how the process will react to receipt of that signal (exit, ignore, execute specified user function).
Control Terminal	Indicates login terminal for this process, if one exists.
Error Field	Records errors encountered during a system call.
Return Value	Contains the result of system calls.
I/O Parameters	Describe the amount of data to transfer, the address of the source (or target) data array in user space, and file offsets for I/O.
File Parameters	Current directory and current root describe the file system environment of the process.
User File Descriptor Table	Records the files the process has open.
Limit Fields	Restrict the size of the process and the size of a file it can write.
Permission Modes Fields	Mask mode settings on files the process creates.

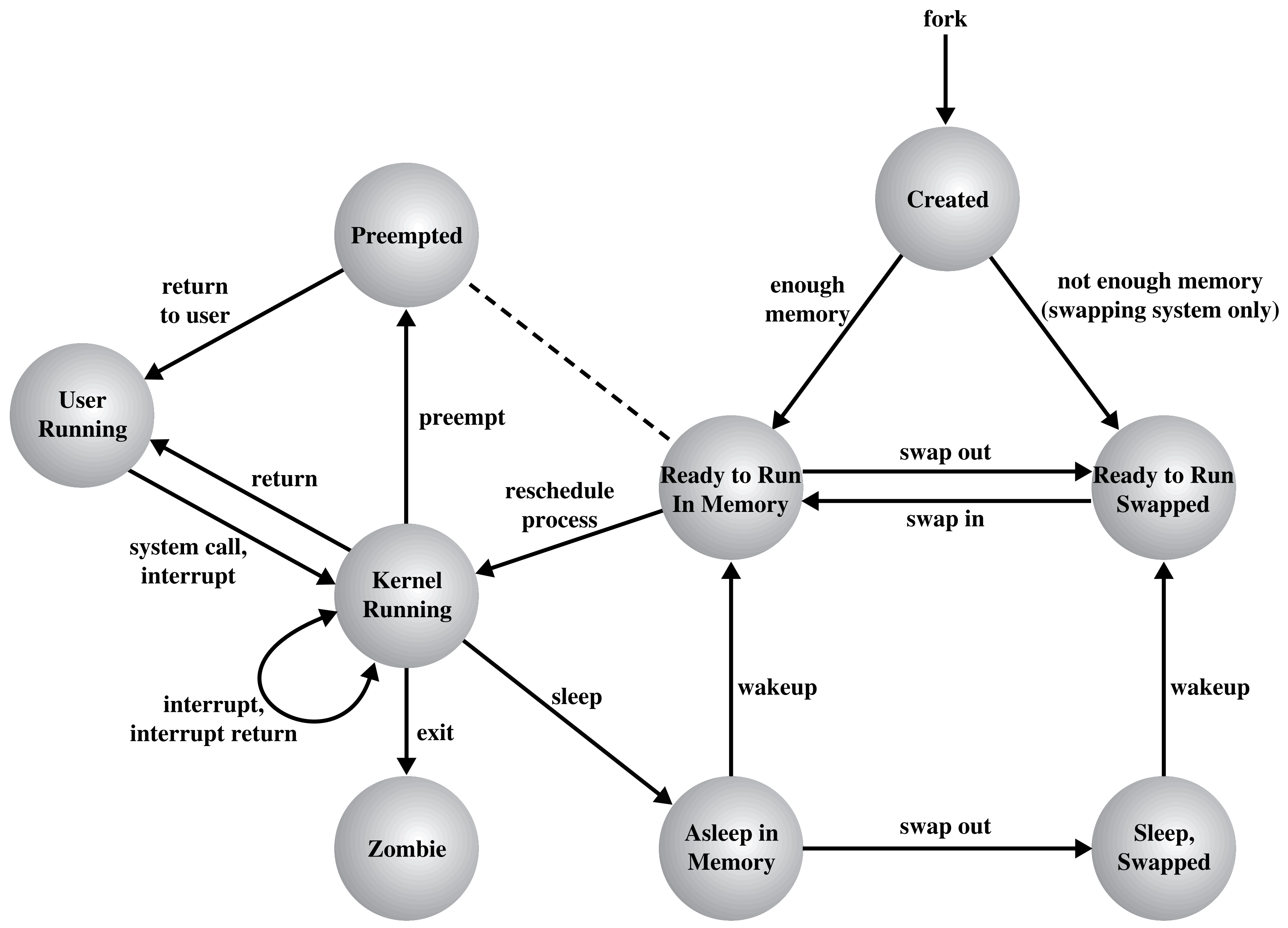


Figure 3.16 UNIX Process State Transition Diagram

4.5 SOLARIS THREAD AND SMP MANAGEMENT

Solaris implements an unusual multilevel thread support designed to provide considerable flexibility in exploiting processor resources.

Multithreaded Architecture

Solaris makes use of four separate thread-related concepts:

- **Process:** This is the normal UNIX process and includes the user's address space, stack, and process control block.
- **User-level threads:** Implemented through a threads library in the address space of a process, these are invisible to the operating system. User-level threads (ULTs)[2] are the interface for application parallelism.
- **Lightweight processes:** A lightweight process (LWP) can be viewed as a mapping between ULTs and kernel threads. Each LWP supports one or more ULTs and maps to one kernel thread. LWPs are scheduled by the kernel independently and may execute in parallel on multiprocessors.
- **Kernel threads:** These are the fundamental entities that can be scheduled and dispatched to run on one of the system processors.

Figure 4.15 illustrates the relationship among these four entities. Note that there is always exactly one kernel thread for each LWP. An LWP is visible within a process to the application. Thus, LWP data structures exist within their respective process address space. At the same time, each LWP is bound to a single dispatchable kernel thread, and the data structure for that kernel thread is maintained within the kernel's address space.

In our example, process 1 consists of a single ULT bound to a single LWP. Thus, there is a single thread of execution, corresponding to a traditional UNIX process. When concurrency is not required within a single process, an application uses this process structure. Process 2 corresponds to a pure ULT strategy. All of the ULTs are supported by a single kernel thread, and therefore only one ULT can execute at a time. This structure is useful for an application that can best be programmed in a way that expresses concurrency but for which it is not necessary to have parallel execution of multiple threads. Process 3 shows multiple threads multiplexed on a lesser number of LWPs. In general, Solaris allows applications to multiplex ULTs on a lesser or equal number of LWPs. This enables the application to specify the degree of parallelism at the kernel level that will support this process. Process 4 has its threads permanently bound to LWPs in a one-to-one mapping. This structure makes the kernel-level parallelism fully visible to the application. It is useful if threads will typically or frequently be suspended in a blocking fashion. Process 5 shows both a mapping of multiple ULTs onto multiple LWPs and the binding of a ULT to a LWP. In addition, one LWP is bound to a particular processor.

Not shown in the figure is the presence of kernel threads that are not associated with LWPs. The kernel creates, runs, and destroys these kernel threads to execute specific system functions. The use of kernel threads rather than kernel processes to implement system functions reduces the overhead of switching within the kernel (from a process switch to a thread switch).

Motivation

The combination of user-level and kernel-level threads gives the application programmer the opportunity to exploit concurrency in a way that is most efficient and most appropriate to a given application.

[2] Again, the acronym ULT is mine and is not found in the Solaris literature.

Some programs have logical parallelism that can be exploited to simplify and structure the code but do not need hardware parallelism. For example, an application that employs multiple windows, only one of which is active at a time, could with advantage be implemented as a set of ULTs on a single LWP. The advantage of restricting such applications to ULTs is efficiency. ULTs may be created, destroyed, blocked, activated, etc. without involving the kernel. If each ULT were known to the kernel, the kernel would have to allocate kernel data structures for each one and perform thread switching. As we have seen (Table 4.1), kernel-level thread switching is more expensive than user-level thread switching.

If an application involves threads that may block, such as when performing I/O, then having multiple LWPs to support an equal or greater number of ULTs is attractive. Neither the application nor the threads library need perform contortions to allow other threads within the same process to execute. Instead, if one thread in a process blocks, other threads within the process may run on the remaining LWPs.

Mapping ULTs one-to-one to LWPs is effective for some applications. For example, a parallel array computation could divide the rows of its arrays among different threads. If there is exactly one ULT per LWP, then no thread switching is required for the computation to proceed.

A mixture of threads that are permanently bound to LWPs and unbound threads (multiple threads sharing multiple LWPs) is appropriate for some applications. For example, a real-time application may want some threads to have systemwide priority and real-time scheduling, while other threads perform background functions and can share one or a small pool of LWPs.

Process Structure

Figure 4.16 compares, in general terms, the process structure of a traditional UNIX system with that of Solaris. On a typical UNIX implementation, the process structure includes the processor ID; the user IDs; a signal dispatch table, which the kernel uses to decide what to do when sending a signal to a process; file descriptors, which describe the state of files in use by this process; a memory map, which defines the address space for this process; and a processor state structure, which includes the kernel stack for this process. Solaris retains this basic structure but replaces the processor state block with a list of structures containing one data block for each LWP.

The LWP data structure includes the following elements:

- An LWP identifier
- The priority of this LWP and hence the kernel thread that supports it
- A signal mask that tells the kernel which signals will be accepted
- Saved values of user-level registers (when the LWP is not running)
- The kernel stack for this LWP, which includes system call arguments, results, and error codes for each call level
- Resource usage and profiling data
- Pointer to the corresponding kernel thread
- Pointer to the process structure

Thread Execution

Figure 4.17 shows a simplified view of both ULT and LWP execution states. The execution of user-level threads is managed by the threads library. Let us first consider unbound threads, that is, threads that share a number of LWPs. An unbound thread can be in one of four states: runnable, active, sleeping, or stopped. A ULT in the active state is currently assigned to a LWP and executes while the underlying kernel thread executes. A number of events may cause the ULT to leave the active state. Let us consider an active ULT called T1. The following events may occur:

- **Synchronization:** T1 invokes one of the concurrency primitives discussed in Chapter 5 to coordinate its activity with other threads and to enforce mutual exclusion. T1 is placed in the sleeping state. When the synchronization condition is met, T1 is moved to the runnable state.

- **Suspension:** Any thread (including T1) may cause T1 to be suspended and placed in the stopped state. T1 remains in that state until another thread issues a continue request, which moves it to the runnable state.
- **Preemption:** An active thread (T1 or some other thread) does something that causes another thread (T2) of higher priority to become runnable. If T1 is the lowest-priority active thread, it is preempted and moved to the runnable state, and T2 is assigned to the LWP made available.
- **Yielding:** If T1 executes the `thr_yield( )` library command, the threads scheduler in the library will look to see if there is another runnable thread (T2) of the same priority. If so, T1 is placed in the runnable state and T2 is assigned to the LWP made available. If not, T1 continues to run.

In all of the preceding cases, when T1 is moved out of the active state, the threads library selects another unbound thread in the runnable state and runs it on the newly available LWP.

Figure 4.17 also shows the state diagram for an LWP. We can view this state diagram as a detailed description of the ULT active state, because an unbound thread only has an LWP assigned to it when it is in the Active state. The LWP state diagram is reasonably self-explanatory. An active thread is only executing when its LWP is in the Running state. When an active thread executes a blocking system call, the LWP enters the Blocked state. However, the ULT remains bound to that LWP and, as far as the threads library is concerned, that ULT remains active.

With bound threads, the relationship between ULT and LWP is slightly different. For example, if a bound ULT moves to the Sleeping state awaiting a synchronization event, its LWP must also stop running. This is accomplished by having the LWP block on a kernel-level synchronization variable.

Interrupts as Threads

Most operating systems contain two fundamental forms of asynchronous activity: processes and interrupts. Processes (or threads) cooperate with each other and manage the use of shared data structures by means of a variety of primitives that enforce mutual exclusion (only one process at a time can execute certain code or access certain data) and that synchronize their execution. Interrupts are synchronized by preventing their occurrence for a period of time. Solaris unifies these two concepts into a single model, namely kernel threads and the mechanisms for scheduling and executing kernel threads. To do this, interrupts are converted to kernel threads.

The motivation for converting interrupts to threads is to reduce overhead. Interrupt handlers often manipulate data shared by the rest of the kernel. Therefore, while a kernel routine that accesses such data is executing, interrupts must be blocked, even though most interrupts will not affect that data. Typically, the way this is done is for the routine to set the interrupt priority level higher to block interrupts, and then lower the priority level after access is completed. These operations take time. The problem is magnified on a multiprocessor system. The kernel must protect more objects and may need to block interrupts on all processors.

The solution in Solaris can be summarized as follows:

1. Solaris employs a set of kernel threads to handle interrupts. As with any kernel thread, an interrupt thread has its own identifier, priority, context, and stack.
2. The kernel controls access to data structures and synchronizes among interrupt threads using mutual exclusion primitives, of the type discussed in Chapter 5. That is, the normal synchronization techniques for threads are used in handling interrupts.
3. Interrupt threads are assigned higher priorities than all other types of kernel threads.

When an interrupt occurs, it is delivered to a particular processor and the thread that was executing on that processor is pinned. A pinned thread cannot move to another processor and its context is preserved; it is simply suspended until the interrupt is processed. The processor then begins executing an interrupt thread. There is a pool of deactivated interrupt threads available, so that a new thread creation is not required. The interrupt thread then executes to handle the interrupt. If the handler routine needs access to a data structure that is currently locked in some fashion for use by another executing thread, the interrupt thread must wait for access to that data structure. An interrupt thread can only be preempted by another interrupt thread of higher priority.

Experience with Solaris interrupt threads indicates that this approach provides superior performance to the traditional interrupt-handling strategy [KLEI95].

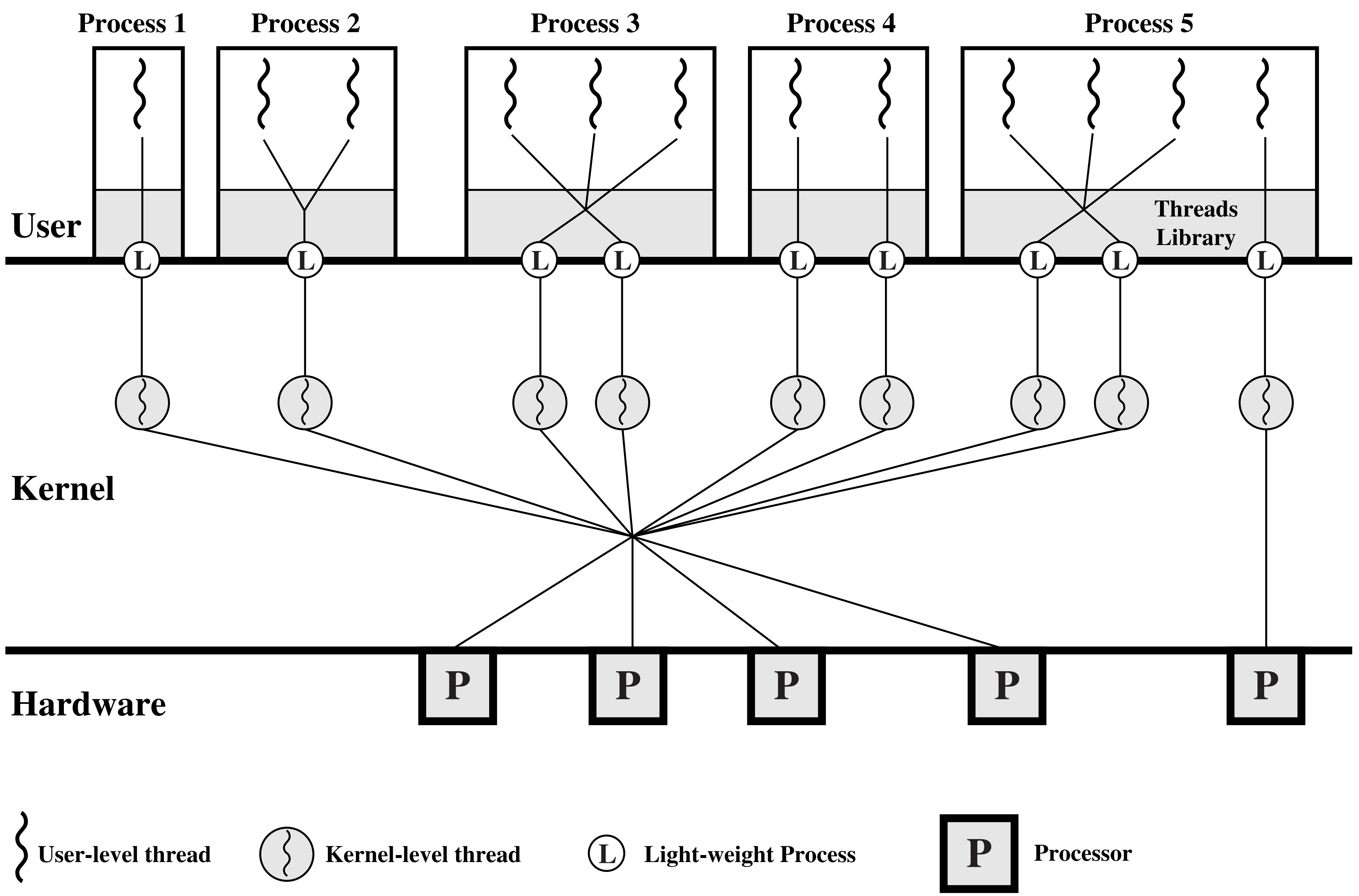


Figure 4.15 Solaris Multithreaded Architecture Example

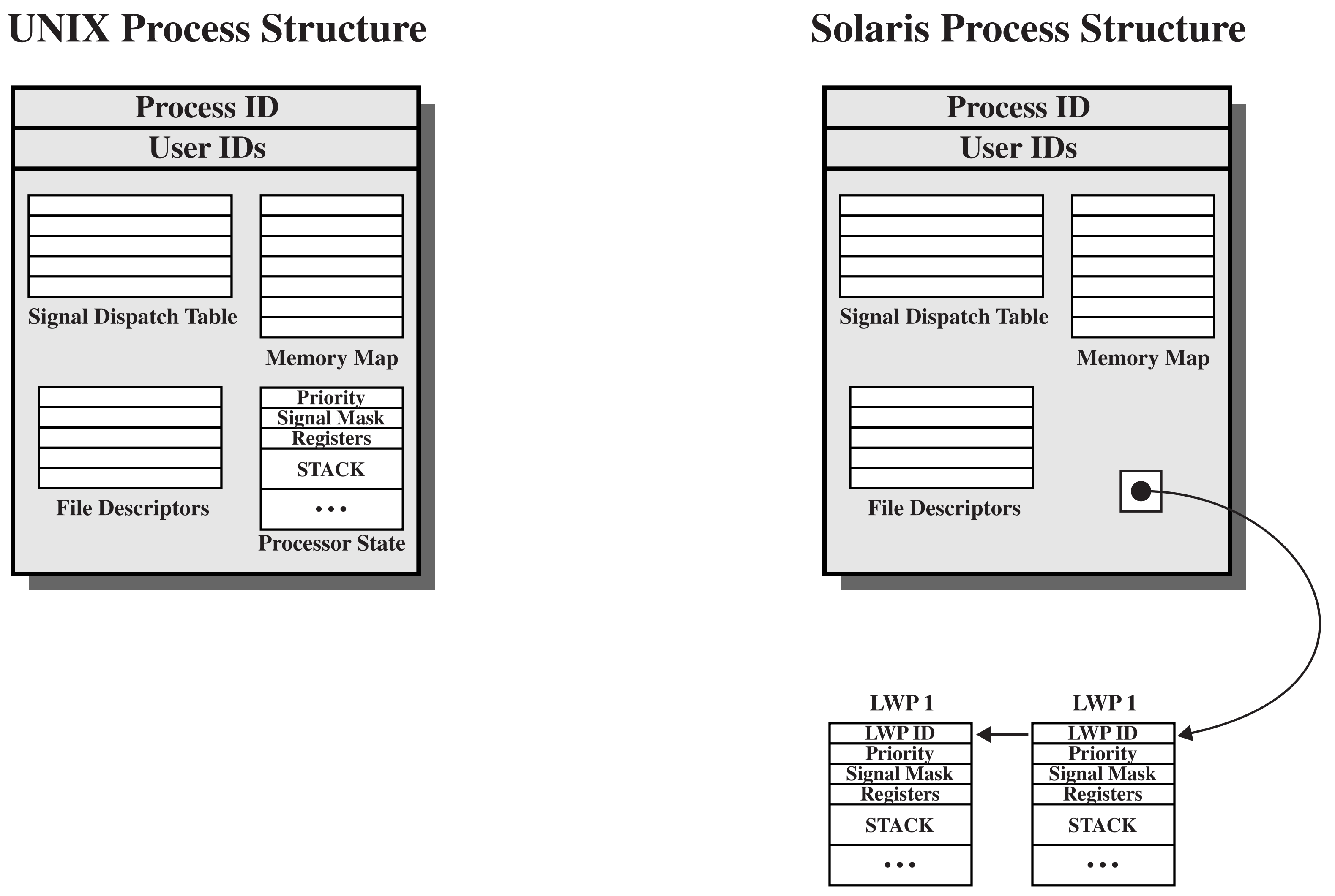


Figure 4.16 Process Structure in Traditional UNIX and Solaris [LEWI96]

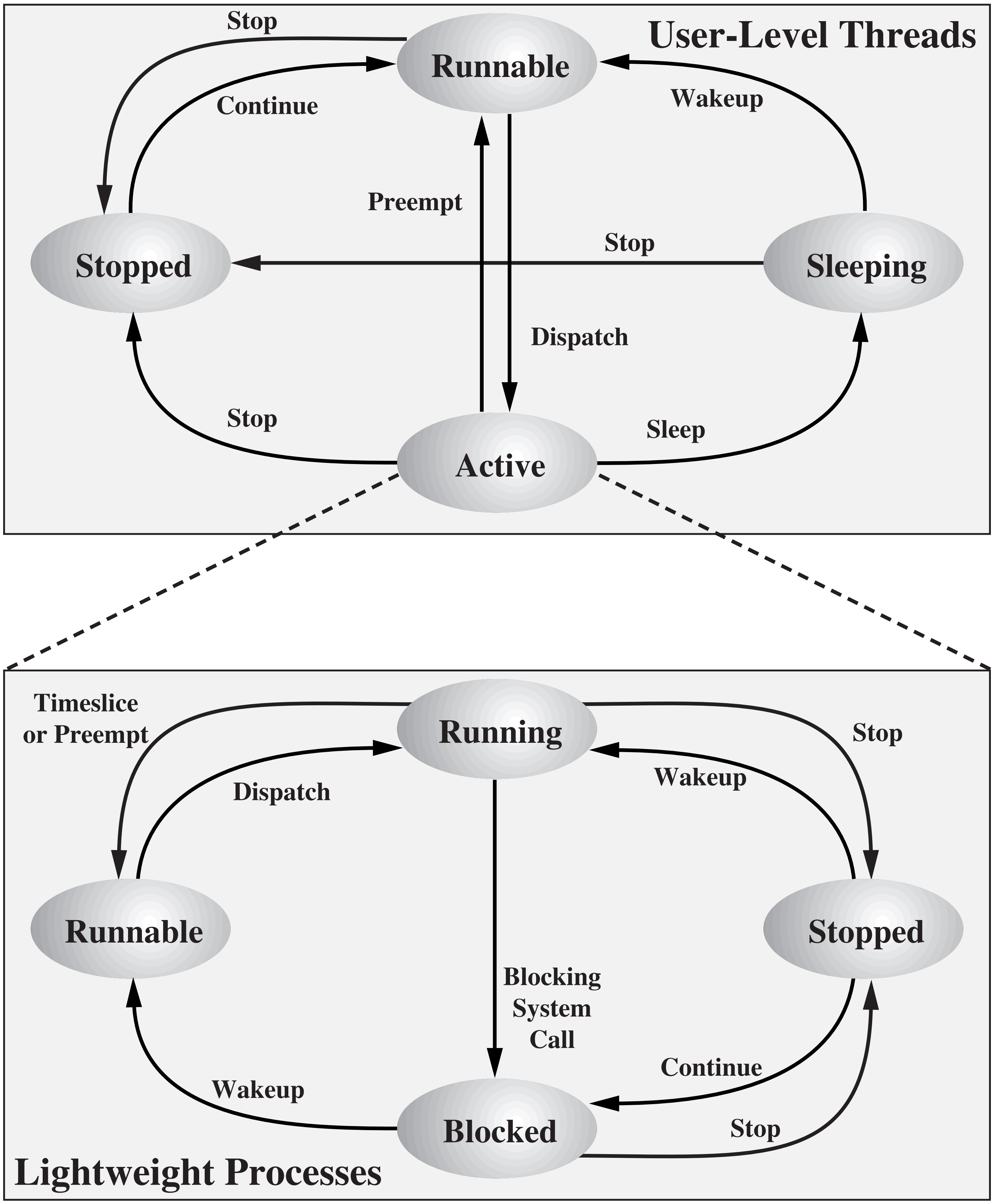


Figure 4.17 Solaris User-Level Thread and LWP States

4.6 LINUX PROCESS AND THREAD MANAGEMENT

Linux Processes

A process, or task, in Linux is represented by a `task_struct` data structure. Linux maintains a `task` table, which is a linear vector of pointers to every `task_struct` data structure currently defined. The `task_struct` data structure contains information in a number of categories:

- **State:** The execution state of the process (executing, ready, suspended, stopped, zombie). This is described subsequently.
- **Scheduling information:** Information needed by Linux to schedule processes. A process can be normal or real time and has a priority. Real-time processes are scheduled before normal processes, and within each category, relative priorities can be used. A counter keeps track of the amount of time a process is allowed to execute.
- **Identifiers:** Each process has a unique process identifier and also has user and group identifiers. A group identifier is used to assign resource access privileges to a group of user.
- **Interprocess communication:** Linux supports the IPC mechanisms found in UNIX SVR4, described in Chapter 6.
- **Links:** Each process includes a link to its parent process, links to its siblings (processes with the same parent), and links to all of its children.
- **Times and timers:** Includes process creation time and the amount of processor time so far consumed by the process. A process may also have associated one or more interval timers. A process defines an interval timer by means of a system call; as a result a signal is sent to the process when the timer expires. A timer may be single use or periodic.
- **File system:** Includes pointers to any files opened by this process.
- **Virtual memory:** Defines the virtual memory assigned to this process.
- **Processor-specific context:** The registers and stack information that constitute the context of this process.

Figure 4.18 shows the execution states of a process. These are:

- **Running:** This state value corresponds to two states. A Running process is either executing or it is ready to execute.
- **Interruptable:** This is a blocked state, in which the process is waiting for an event, such as the end of an I/O operation, the availability of a resource, or a signal from another process.
- **Uninterruptable:** This is another blocked state. The difference between this and the Interruptable state is that in an uninterruptable state, a process is waiting directly on hardware conditions and therefore will not accept any signals.
- **Stopped:** The process has been halted, and can only resume by positive action from another process. For example, a process that is being debugged can be put into the Stopped state.
- **Zombie:** The process has been terminated but, for some reason, still must have its task structure in the process table.

Linux Threads

A new process is created in Linux by copying the attributes of the current process. A new process can be *cloned* so that it shares resources, such as files, signal handlers, and virtual memory. When the two processes share the same virtual memory, they function as threads within a single process. However, no separate type of data structure is defined for a thread. Thus, Linux makes no distinction between a thread and a process.

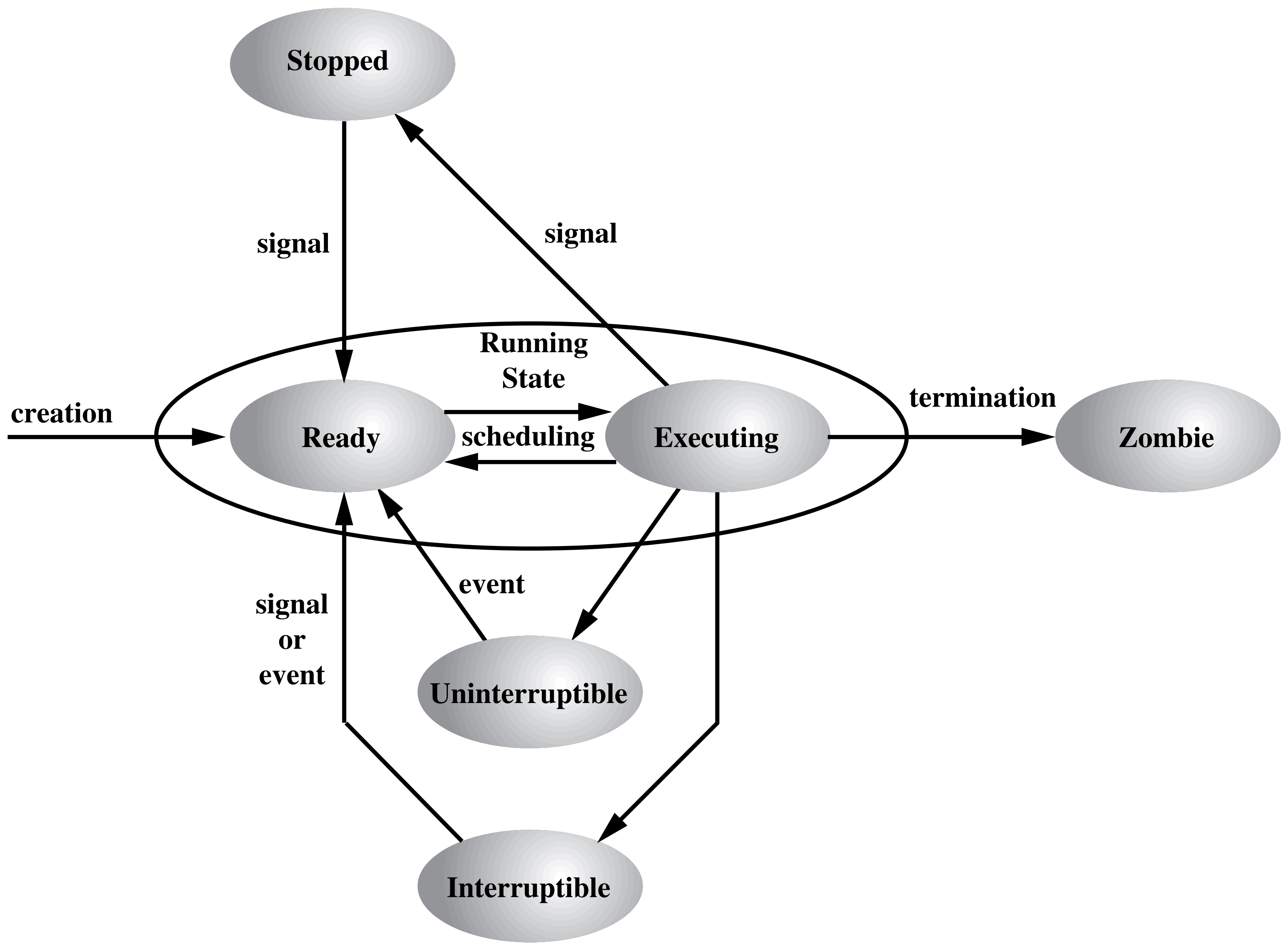


Figure 4.18 Linux Process/Thread Model

6.7 UNIX CONCURRENCY MECHANISMS

UNIX provides a variety of mechanisms for interprocessor communication and synchronization. Here, we look at the most important of these:

- Pipes
- Messages
- Shared memory
- Semaphores
- Signals

Pipes, messages, and shared memory provide a means of communicating data across processes, whereas semaphores and signals are used to trigger actions by other processes.

Pipes

One of the most significant contributions of UNIX to the development of operating systems is the pipe. Inspired by the concept of coroutines [RITC84], a pipe is a circular buffer allowing two processes to communicate on the producer-consumer model. Thus, it is a first-in-first-out queue, written by one process and read by another.

When a pipe is created, it is given a fixed size in bytes. When a process attempts to write into the pipe, the write request is immediately executed if there is sufficient room; otherwise the process is blocked. Similarly, a reading process is blocked if it attempts to read more bytes than are currently in the pipe; otherwise the read request is immediately executed. The operating system enforces mutual exclusion: that is, only one process can access a pipe at a time.

There are two types of pipes: named and unnamed. Only related processes can share unnamed pipes, while unrelated processes can share only named pipes.

Messages

A message is a block of text with an accompanying type. UNIX provides *msgsnd* and *msgrcv* system calls for processes to engage in message passing. Associated with each process is a message queue, which functions like a mailbox.

The message sender specifies the type of message with each message sent, and this can be used as a selection criterion by the receiver. The receiver can either retrieve messages in first-in-first-out order or by type. A process will suspend when trying to send a message to a full queue. A process will also suspend when trying to read from an empty queue. If a process attempts to read a message of a certain type and fails because no message of that type is present, the process is not suspended.

Shared Memory

The fastest form of interprocess communication provided in UNIX is shared memory. This is a common block of virtual memory shared by multiple processes. Processes read and write shared memory using the same machine instructions they use to read and write other portions of their virtual memory space. Permission is read-only or read-write for a process, determined on a per-process basis. Mutual exclusion constraints are not part of the shared-memory facility but must be provided by the processes using the shared memory.

Semaphores

The semaphore system calls in UNIX System V are a generalization of the *wait* and *signal* primitives defined in Chapter 5, in that several operations can be done simultaneously and the increment and decrement operations can be values greater than 1. The kernel does all of the requested operations atomically; no other process may access the semaphore until all operations are done.

A semaphore consists of the following elements:

- Current value of the semaphore
- Process ID of the last process to operate on the semaphore
- Number of processes waiting for the semaphore value to be greater than its current value
- Number of processes waiting for the semaphore value to be zero

Associated with the semaphore are queues of processes suspended on that semaphore.

Semaphores are actually created in sets, with a semaphore set consisting of one or more semaphores. There is a *semctl* system call that allows all of the semaphore values in the set to be set at the same time. In addition, there is a *semop* system call that takes as an argument a list of semaphore operations, each defined in one of the semaphores in a set. When this call is made, the kernel performs the indicated operations one at a time. For each operation, the actual function is specified by the value *sem_op*. The following are the possibilities.

- If *sem_op* is positive, the kernel increments the value of the semaphore and awakens all processes waiting for the value of the semaphore to increase.
- If *sem_op* is 0, the kernel checks the semaphore value. If 0, it continues with the other operations on the list; otherwise, it increments the number of processes waiting for this semaphore to be 0 and suspends the process on the event that the value of the semaphore equals 0.
- If *sem_op* is negative and its absolute value is less than or equal to the semaphore value, the kernel adds *sem-op* (a negative number) to the semaphore value. If the result is 0, the kernel awakens all processes waiting for the value of the semaphore to equal 0.
- If *sem_op* is negative and its absolute value is greater than the semaphore value, the kernel suspends the process on the event that the value of the semaphore increases

This generalization of the semaphore provides considerable flexibility in performing process synchronization and coordination.

Signals

A signal is a software mechanism that informs a process of the occurrence of asynchronous events. A signal is similar to a hardware interrupt but does not employ priorities. That is, all signals are treated equally; signals that occur at the same time are presented to a process one at a time, with no particular ordering.

Processes may send each other signals, or the kernel may send signals internally. A signal is delivered by updating a field in the process table for the process to which the signal is being sent. Because each signal is maintained as a single bit, signals of a given type cannot be queued. A signal is processed just after a process wakes up to run or whenever the process is preparing to return from a system call. A process may respond to a signal by performing some default action (e.g., termination), executing a signal handler function, or ignoring the signal.

Table 6.2 lists signals defined for UNIX SVR4.

Table 6.2 UNIX Signals

Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPT	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALARM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCLD	Death of a child
19	SIGPWR	Power failure

6.8 SOLARIS THREAD SYNCHRONIZATION PRIMITIVES

In addition to the concurrency mechanisms of UNIX SVR4, Solaris supports four thread synchronization primitives:

- Mutual exclusion (mutex) locks
- Semaphores
- Multiple readers, single writer (readers/writer) locks
- Condition variables

Solaris implements these primitives within the kernel for kernel threads; they are also provided in the threads library for user-level threads. Execution of a primitive creates a data structure that contains parameters specified by the creating thread (Figure 6.13). Once a synchronization object is created, there are essentially only two operations that can be performed: enter (acquire, lock) and release (unlock). There are no mechanisms in the kernel or the threads library to enforce mutual exclusion or to prevent deadlock. If a thread attempts to access a piece of data or code that is supposed to be protected but does not use the appropriate synchronization primitive, then such access occurs. If a thread locks an object and then fails to unlock it, no kernel action is taken.

All of the synchronization primitives require the existence of a hardware instruction that allows an object to be tested and set in one atomic operation, as discussed in Section 5.3.

Mutual Exclusion Lock

A mutex lock prevents more than one thread from proceeding when the lock is acquired. The thread that locks the mutex must be the one that unlocks it. A thread attempts to acquire a mutex lock by executing the `mutex_enter` primitive. If `mutex_enter` cannot set the lock (because it is already set by another thread), the blocking action depends on type-specific information stored in the mutex object. The default blocking policy is a spin lock: A blocked thread polls the status of the lock while executing in a spin wait loop. An interrupt-based blocking mechanism is optional. In this latter case, the mutex includes a *turnstile id* that identifies a queue of threads sleeping on this lock.

The primitives associated with a mutex lock are:

`mutex_enter()`	Acquires the lock, potentially blocking if it is already held
`mutex_exit()`	Releases the lock, potentially unblocking a waiter
`mutex_tryenter()`	Acquires the lock if it is not already held

The `mutex_tryenter()` primitive provides a nonblocking way of performing the mutual exclusion function. This enables the programmer to use a busy-wait approach for user-level threads, which avoids blocking the entire process because one thread is blocked.

Semaphores

Solaris provides classic counting semaphores, with the following primitives:

`sema_p()`	Decrements the semaphore, potentially blocking the thread
`sema_v()`	Increments the semaphore, potentially unblocking a waiting thread
`sema_tryp()`	Decrements the semaphore if blocking is not required

Again, the `sema_tryp()` primitive permits busy waiting.

Readers/Writer Lock

The readers/writer lock allows multiple threads to have simultaneous read-only access to an object protected by the lock. It also allows a single thread to access the object for writing at one time, while excluding all readers. When the lock is acquired for writing it takes on the status of *write lock*: all threads attempting access for reading or writing must wait. If one or more readers have acquired the lock, its status is *read lock*. The primitives are:

`rw_enter()`	Attempts to acquire a lock as reader or writer
`rw_exit()`	Releases a lock as reader or writer
`rw_tryenter()`	Acquires the lock if blocking is not required
`rw_downgrade()`	A thread that has acquired a write lock converts it to a read lock. Any waiting writer remains waiting until this thread releases the lock. If there are no waiting writers, the primitive wakes up any pending readers.
`rw_tryupgrade()`	Attempts to convert a reader lock into a writer lock.

Condition Variables

A condition variable is used to wait until a particular condition is true. Condition variables must be used in conjunction with a mutex lock. This implements a monitor of the type illustrated in Figure 5.22. The primitives are:

`cv_wait()`	Blocks until the condition is signaled
`cv_signal()`	Wakes up one of the threads blocked in `cv_wait()`
`cv_broadcast()`	Wakes up all of the threads blocked in `cv_wait()`

`cv_wait()` releases the associated mutex before blocking and reacquires it before returning. Because reacquisition of the mutex may be blocked by other threads waiting for the mutex, the condition that caused the wait must be retested. Thus, typical usage is as follows:

```
    mutex_enter(&m)
 • •
    while (some_condition) {
     cv_wait(&cv, &m);
    }
 • •
    mutex_exit(&m);
```

This allows the condition to be a complex expression, because it is protected by the mutex.

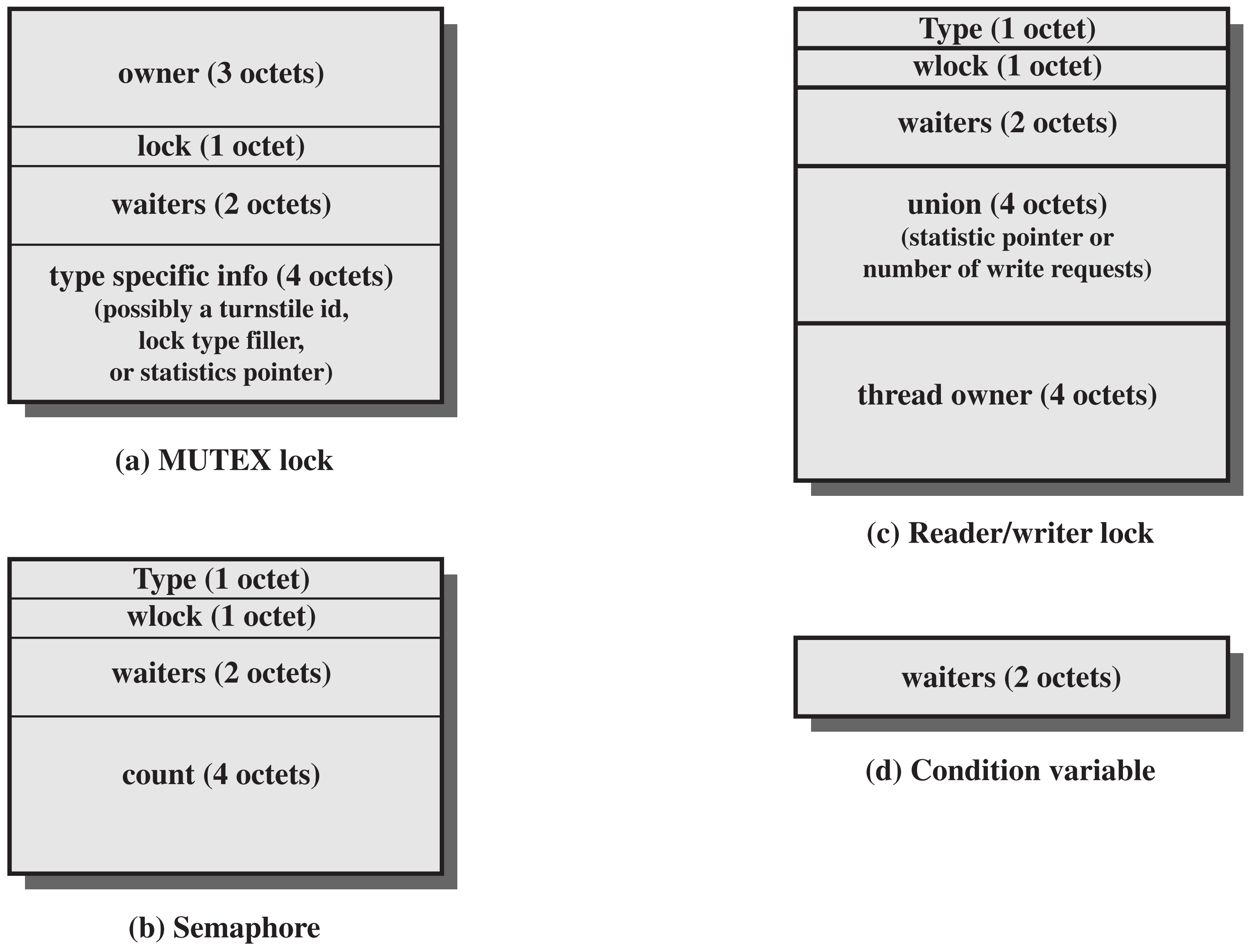


Figure 6.13 Solaris Synchronization Data Structures

8.3 UNIX AND SOLARIS MEMORY MANAGEMENT

Because UNIX is intended to be machine independent, its memory-management scheme will vary from one system to the next. Earlier versions of UNIX simply used variable partitioning with no virtual memory scheme. Current implementations, including SVR4 and Solaris 2.x, make use of paged virtual memory.

In SVR4 and Solaris, there are actually two separate memory-management schemes. The **paging system** provides a virtual memory capability that allocates page frames in main memory to processes and also allocates page frames to disk block buffers. Although this is an effective memory-management scheme for user processes and disk I/O, a paged virtual memory scheme is less suited to managing the memory allocation for the kernel. For this latter purpose, a **kernel memory allocator** is used. We examine these two mechanisms in turn.

Paging System

Data Structures

For paged virtual memory, UNIX makes use of a number of data structures that, with minor adjustment, are machine independent (Figure 8.22 and Table 8.5):

· **Page table:** Typically, there will be one page table per process, with one entry for each page in virtual memory for that process.
· **Disk block descriptor:** Associated with each page of a process is an entry in this table that describes the disk copy of the virtual page.
· **Page frame data table:** Describes each frame of real memory and is indexed by frame number.
· **Swap-use table:** There is one swap-use table for each swap device, with one entry for each page on the device.

Most of the fields defined in Table 8.5 are self-explanatory. A few warrant further comment. The Age field in the page table entry is an indication of how long it has been since a program referenced this frame. However, the number of bits and the frequency of update of this field are implementation dependent. Therefore, there is no universal UNIX use of this field for page replacement policy.

The Type of Storage field in the disk block descriptor is needed for the following reason: When an executable file is first used to create a new process, only a portion of the program and data for that file may be loaded into real memory. Later, as page faults occur, new portions of the program and data are loaded. It is only at the time of first loading that virtual memory pages are created and assigned to locations on one of the devices to be used for swapping. At that time, the operating system is told whether it needs to clear (set to 0) the locations in the page frame before the first loading of a block of the program or data.

Page Replacement

The page frame data table is used for page replacement. Several pointers are used to create lists within this table. All of the available frames are linked together in a list of free frames available for bringing in pages. When the number of available pages drops below a certain threshold, the kernel will steal a number of pages to compensate.

The page-replacement algorithm used in SVR4 is a refinement of the clock policy algorithm (Figure 8.16) known as the two-handed clock algorithm (Figure 8.23). The algorithm uses the reference bit in the page table entry for each page in memory that is eligible (not locked) to be swapped out. This bit is set to 0 when the page is first brought in and set to 1 when the page is referenced for a read or write. One hand in the clock algorithm, the fronthand, sweeps through the pages on the list of eligible pages and sets the reference bit to 0 on each page. Sometime later, the backhand sweeps through the same list and checks the reference bit. If the bit is set to 1, then that page has been referenced since the fronthand swept by; these frames are ignored. If the bit is still set to 0, then the page has not been referenced in the time interval between the visit by fronthand and backhand; these pages are placed on a list to be paged out.

Two parameters determine the operation of the algorithm:

- **Scanrate:** The rate at which the two hands scan through the page list, in pages per second
- **Handspread:** The gap between fronthand and backhand

These two parameters have default values set at boot time based on the amount of physical memory. The scanrate parameter can be altered to meet changing conditions. The parameter varies linearly between the values slowscan and fastscan (set at configuration time) as the amount of free memory varies between the values *lotsfree* and *minfree*. In other words, as the amount of free memory shrinks, the clock hands move more rapidly to free up more pages. The handspread parameter determines the gap between the fronthand and the backhand and therefore, together with scanrate, determines the window of opportunity to use a page before it is swapped out due to lack of use.

Kernel Memory Allocator

The kernel generates and destroys small tables and buffers frequently during the course of execution, each of which requires dynamic memory allocation. [VAHA96] lists the following examples:

- The pathname translation routing may allocate a buffer to copy a pathname from user space.
- The `allocb()` routine allocates STREAMS buffers of arbitrary size.
- Many UNIX implementations allocate zombie structures to retain exit status and resource usage information about deceased processes.
- In SVR4 and Solaris, the kernel allocates many objects (such as proc structures, vnodes, and file descriptor blocks) dynamically when needed.

Most of these blocks are significantly smaller than the typical machine page size, and therefore the paging mechanism would be inefficient for dynamic kernel memory allocation. For SVR4, a modification of the buddy system, described in Section 7.2, is used.

In buddy systems, the cost to allocate and free a block of memory is low compared to that of best-fit or first-fit policies [KNUT97]. However, in the case of kernel memory management, the allocation and free operations must be made as fast as possible. The drawback of the buddy system is the time required to fragment and coalesce blocks.

Barkley and Lee at AT&T proposed a variation known as a lazy buddy system [BARK89], and this is the technique adopted for SVR4. The authors observed that UNIX often exhibits steady-state behavior in kernel memory demand; that is, the amount of demand for blocks of a particular size varies slowly in time. Therefore, if a block of size 2^i is released and is immediately coalesced with its buddy into a block of size 2^{i+1}, the kernel may next request a block of size 2^i, which may necessitate splitting the larger block again. To avoid this unnecessary coalescing and splitting, the lazy buddy system defers coalescing until it seems likely that it is needed, and then coalesces as many blocks as possible.

The lazy buddy system uses the following parameters:

N_i = current number of blocks of size 2^i.

A_i = current number of blocks of size 2^i that are allocated (occupied).

G_i = current number of blocks of size 2^i that are globally free; these are blocks that are eligible for coalescing; if the buddy of such a block becomes globally free, then the two blocks will be coalesced into a globally free block of size 2^{i+1}. All free blocks (holes) in the standard buddy system could be considered globally free.

L_i = current number of blocks of size 2^i that are locally free; these are blocks that are not eligible for coalescing. Even if the buddy of such a block becomes free, the two blocks are not coalesced. Rather, the locally free blocks are retained in anticipation of future demand for a block of that size.

The following relationship holds:

$$N_i = A_i + G_i + L_i$$

In general, the lazy buddy system tries to maintain a pool of locally free blocks and only invokes coalescing if the number of locally free blocks exceeds a threshold. If there are too many locally free blocks, then there is a chance that there will be a lack of free blocks at the next level to satisfy demand. Most of the time, when a block is freed, coalescing does not occur, so there is minimal bookkeeping and operational costs. When a block is to be allocated, no distinction is made between locally and globally free blocks; again, this minimizes bookkeeping.

The criterion used for coalescing is that the number of locally free blocks of a given size should not exceed the number of allocated blocks of that size (i.e., we must have $L_i \leq A_i$). This is a reasonable guideline for restricting the growth of locally free blocks, and experiments in [BARK89] confirm that this scheme results in noticeable savings.

To implement the scheme, the authors define a delay variable as follows:

$$D_i = A_i - L_i = N_i - 2L_i - G_i$$

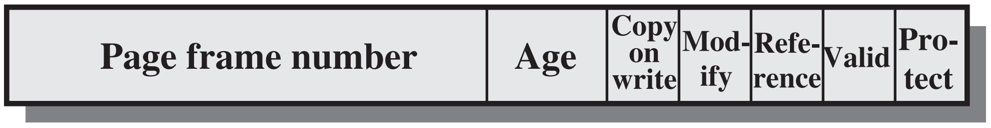
Figure 8.24 shows the algorithm.

Table 8.5 UNIX SVR4 Memory Management Parameters (page 1 of 2)

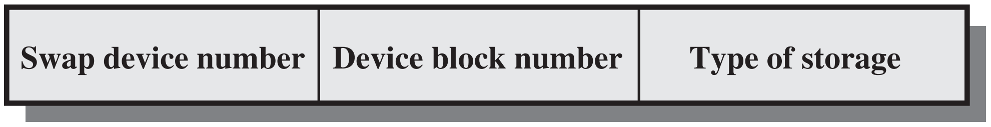
Page Table Entry	
Page frame number	Refers to frame in real memory.
Age	Indicates how long the page has been in memory without being referenced. The length and contents of this field are processor dependent.
Copy on write	Set when more than one process shares a page. If one of the processes writes into the page, a separate copy of the page must first be made for all other processes that share the page. This feature allows the copy operation to be deferred until necessary and avoided in cases where it turns out not to be necessary.
Modify	Indicates page has been modified.
Reference	Indicates page has been referenced. This bit is set to zero when the page is first loaded and may be periodically reset by the page replacement algorithm.
Valid	Indicates page is in main memory.
Protect	Indicates whether write operation is allowed.
Disk Block Descriptor	
Swap device number	Logical device number of the secondary device that holds the corresponding page. This allows more than one device to be used for swapping.
Device block number	Block location of page on swap device.
Type of storage	Storage may be swap unit or executable file. In the latter case, there is an indication as to whether the virtual memory to be allocated should be cleared first.

Table 8.5 UNIX SVR4 Memory Management Parameters (page 2 of 2)

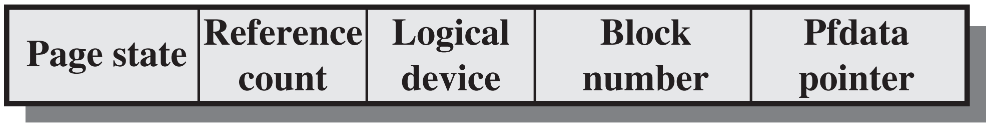
Page Frame Data Table Entry	
Page State	Indicates whether this frame is available or has an associated page. In the latter case, the status of the page is specified: on swap device, in executable file, or DMA in progress.
Reference count	Number of processes that reference the page.
Logical device	Logical device that contains a copy of the page.
Block number	Block location of the page copy on the logical device.
Pfdata pointer	Pointer to other pfdata table entries on a list of free pages and on a hash queue of pages.
Swap-use Table Entry	
Reference count	Number of page table entries that point to a page on the swap device.
Page/storage unit number	Page identifier on storage unit.



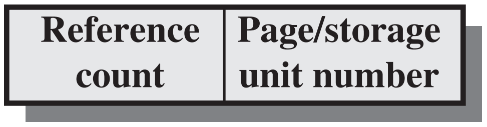
(a) Page table entry



(b) Disk block descriptor



(c) Page frame data table entry



(d) Swap-use table entry

Figure 8.22 UNIX SVR4 Memory Management Formats

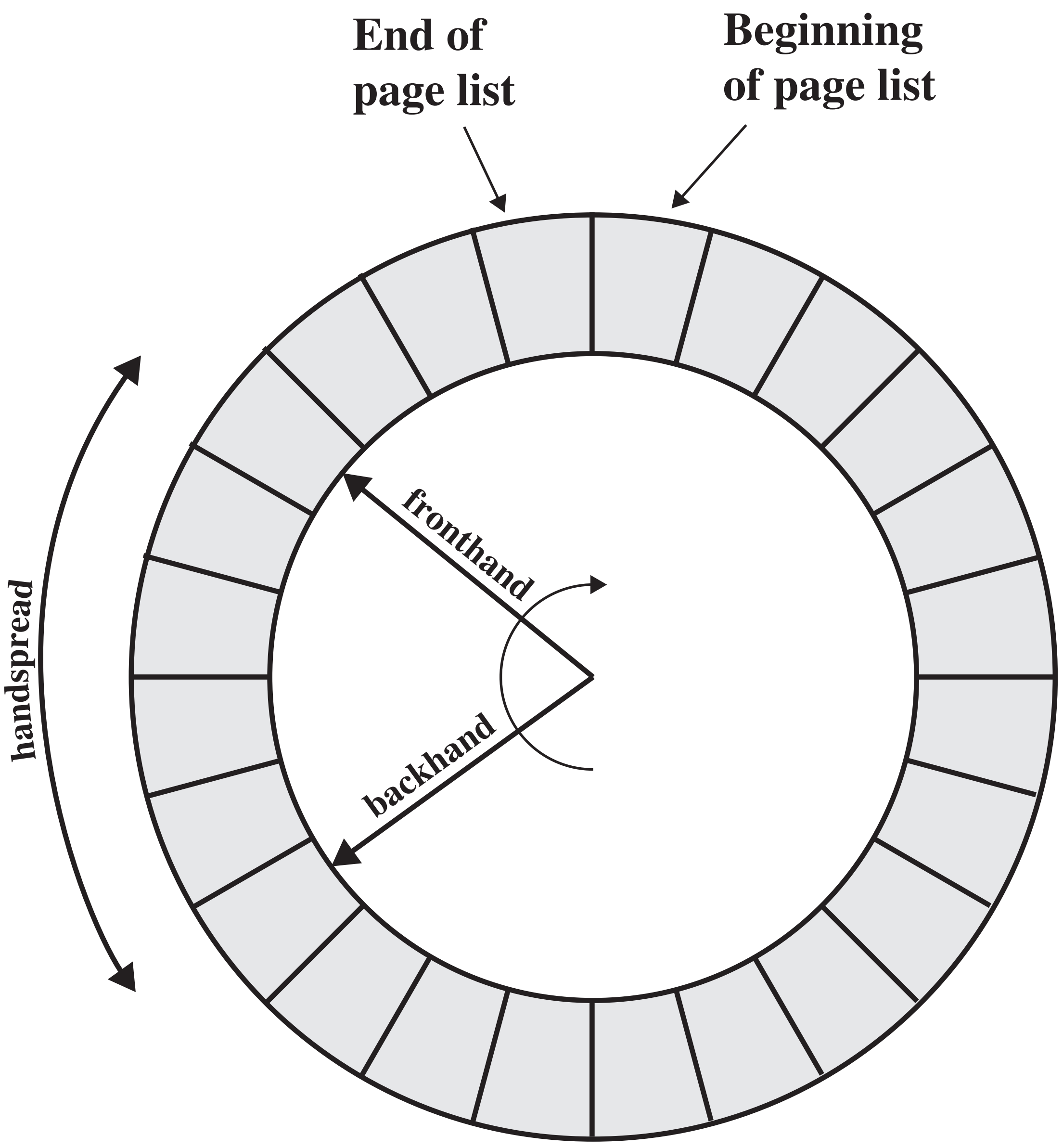


Figure 8.23 Two-Handed Clock Page-Replacement Algorithm

```
Initial value of $D_i$ is 0
After an operation, the value of $D_i$ is updated as follows

(I) if the next operation is a block allocate request:
        if there is any free block, select one to allocate
            if the selected block is locally free
                then $D_i := D_i + 2$
                else $D_i := D_i + 1$
        otherwise
            first get two blocks by splitting a larger one into two (recursive operation)
            allocate one and mark the other locally free
            $D_i$ remains unchanged (but D may change for other block sizes because of the
                                    recursive call)

(II) if the next operation is a block free request
        Case $D_i \geq 2$
            mark it locally free and free it locally
            $D_i := D_i - 2$
        Case $D_i = 1$
            mark it globally free and free it globally; coalesce if possible
            $D_i := 0$
        Case $D_i = 0$
            mark it globally free and free it globally; coalesce if possible
            select one locally free block of size 2i and free it globally; coalesce if possible
            $D_i := 0$
```

Figure 8.24 Lazy Buddy System Algorithm

8.4 LINUX MEMORY MANAGEMENT

Linux shares many of the characteristics of the memory management schemes of other UNIX implementations but has its own unique features. Overall, the Linux memory-management scheme is quite complex [DUBE98]. Here, we give a brief overview.

Linux Virtual Memory

Virtual Memory Addressing

Linux makes use of a three-level page table structure, consisting of the following types of tables (each individual table is the size of one page):

- **Page directory:** An active process has a single page directory that is the size of one page. Each entry in the page directory points to one page of the page middle directory. The page directory must be in main memory for an active process.
- **Page middle directory:** The page middle directory may span multiple pages. Each entry in the page middle directory points to one page in the page table.
- **Page table:** The page table may also span multiple pages. Each page table entry refers to one virtual page of the process.

To use this three-level page table structure, a virtual address in Linux is viewed as consisting of four fields. The leftmost (most significant) field is used as an index into the page directory. The next field serves as an index into the page middle directory. The third field serves as an index into the page table. The fourth field gives the offset within the selected page of memory.

The Linux page table structure is platform independent and was designed to accommodate the 64-bit Alpha processor, which provides hardware support for three levels of paging. With 64-bit addresses, the use of only two levels of pages on the Alpha would result in very large page tables and directories. The 32-bit Pentium/x86 architecture has a two-level hardware paging mechanism. The Linux software accommodates the two-level scheme by defining the size of the page middle directory as one.

Page Allocation

To enhance the efficiency of reading in and writing out pages to and from main memory, Linux defines a mechanism for dealing with contiguous blocks of pages mapped into contiguous blocks of page frames. For this purpose, the buddy system is used. The kernel maintains a list of contiguous page frame groups of fixed size; a group may consists of 1, 2, 4, 8, 16, or 32 page frames. As pages are allocated and deallocated in main memory, the available groups are split and merged using the buddy algorithm.

Page Replacement Algorithm

The Linux page replacement algorithm is based on the clock algorithm described in Section 8.2 (see Figure 8.16). In the simple clock algorithm, a use bit and a modify bit are associated with each page in main memory. In the Linux scheme, the use bit is replaced with an 8-bit age variable. Each time that a page is accessed, the age variable is incremented. In the background, Linux periodically sweeps through the global page pool and decrements the age variable for each page as it rotates through all the pages in main memory. A page with an age of 0 is an "old" page that has not been referenced in some time and is the best candidate for replacement. The larger the value of age, the more frequently a page has been used in recent times and the less eligible it is for replacement. Thus, the Linux algorithm is a form of least frequently used policy.

Kernel Memory Allocation

The foundation of kernel memory allocation for Linux is the page allocation mechanism used for user virtual memory management. As in the virtual memory scheme, a buddy algorithm is used so that memory for the kernel can be allocated and deallocated in units of one or more pages. Because the minimum amount of memory that can be allocated in this fashion is one page, the page allocator alone would be inefficient because the kernel requires small short-term memory chunks in odd sizes. To accommodate these small chunks, Linux uses a scheme known as *slab allocation* [BONW94] within an allocated page. On a Pentium/x86 machine, the page size is 4 kbytes, and chunks within a page may be allocated of sizes 32, 64, 128, 252, 508, 2040, and 4080 bytes.

The slab allocator is relatively complex and is not examined in detail here; a good description can be found in [VAHA96]. In essence, Linux maintains a set of linked lists, one for each size of chunk. Chunks may be split and aggregated in a manner similar to the buddy algorithm, and moved between lists accordingly.

9.3 TRADITIONAL UNIX SCHEDULING

In this section we examine traditional UNIX scheduling, which is used in both SVR3 and 4.3 BSD UNIX. These systems are primarily targeted at the time-sharing interactive environment. The scheduling algorithm is designed to provide good response time for interactive users while ensuring that low-priority background jobs do not starve. Although this algorithm has been replaced in modern UNIX systems, it is worthwhile to examine the approach because it is representative of practical time-sharing scheduling algorithms. The scheduling scheme for SVR4 includes an accommodation for real-time requirements, and so its discussion is deferred to Chapter 10.

The traditional UNIX scheduler employs multilevel feedback using round robin within each of the priority queues. The system makes use of 1-second preemption. That is, if a running process does not block or complete within 1 second, it is preempted. Priority is based on process type and execution history. The following formulas apply:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i-1)}{2} + nice_j$$

where

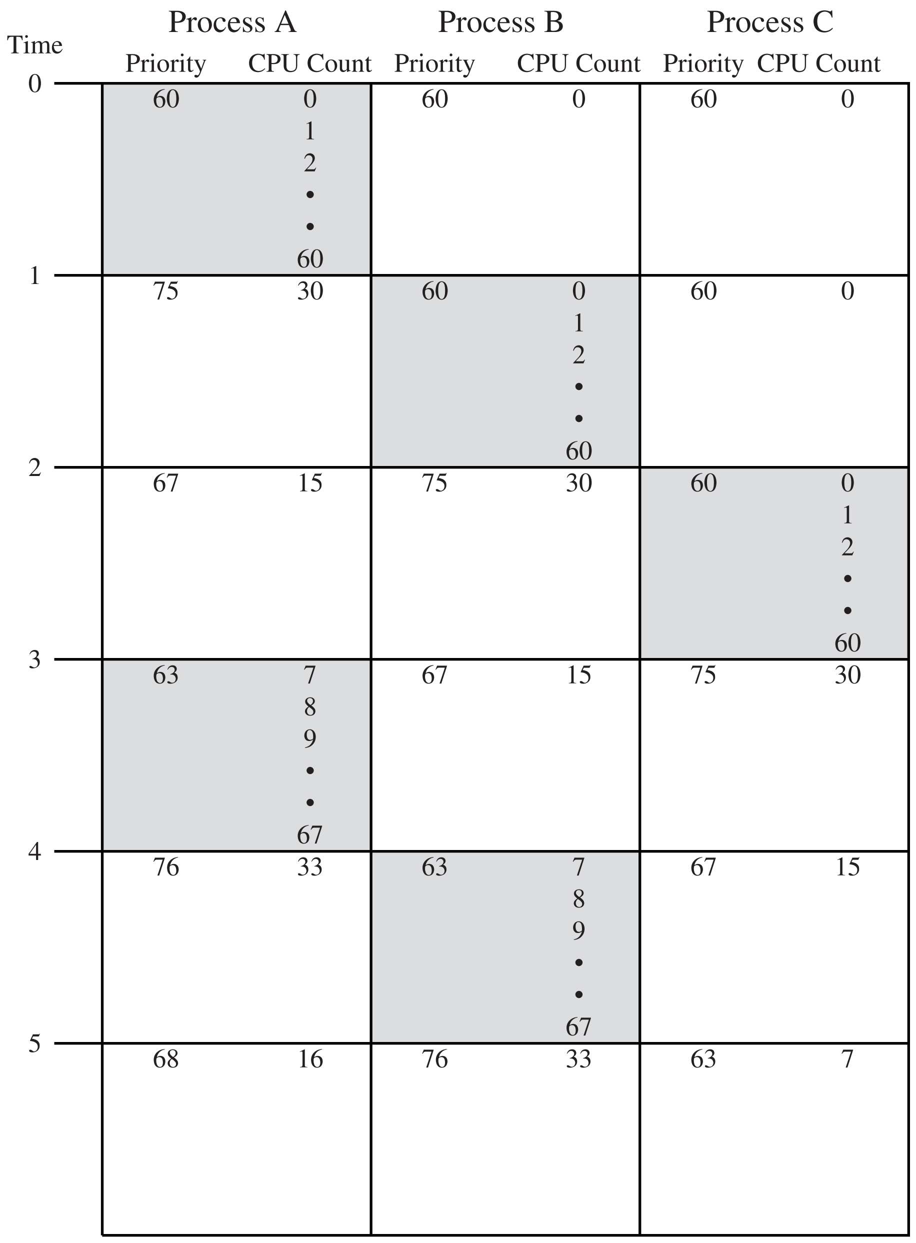
$CPU_j(i)$	=	Measure of processor utilization by process *j* through interval *i*
$P_j(i)$	=	Priority of process *j* at beginning of interval *i*; lower values equal higher priorities
Base_j	=	Base priority of process *j*
nice_j	=	user-controllable adjustment factor

The priority of each process is recomputed once per second, at which time a new scheduling decision is made. The purpose of the base priority is to divide all processes into fixed bands of priority levels. The CPU and nice components are restricted to prevent a process from migrating out of its assigned band (assigned by the base priority level). These bands are used to optimize access to block devices (e.g., disk) and to allow the operating system to respond quickly to system calls. In decreasing order of priority, the bands are:

· Swapper
· Block I/O device control
· File manipulation
· Character I/O device control
· User processes

This hierarchy should provide the most efficient use of the I/O devices. Within the user process band, the use of execution history tends to penalize processor-bound processes at the expense of I/O-bound processes. Again, this should improve efficiency. Coupled with the round robin preemption scheme, the scheduling strategy is well equipped to satisfy the requirements for general-purpose time sharing.

An example of process scheduling is shown in Figure 9.17. Processes A, B, and C are created at the same time with base priorities of 60 (we will ignore the nice value). The clock interrupts the system 60 times per second and increments a counter for the running process. The example assumes that none of the process block themselves and that no other processes are ready to run. Compare this with Figure 9.16.



Time	Process A		Process B		Process C	
	Priority	CPU Count	Priority	CPU Count	Priority	CPU Count
0	60	0 1 2 • • 60	60	0	60	0
1	75	30	60	0 1 2 • • 60	60	0
2	67	15	75	30	60	0 1 2 • • 60
3	63	7 8 9 • • 67	67	15	75	30
4	76	33	63	7 8 9 • • 67	67	15
5	68	16	76	33	63	7

Shaded rectangle represents executing process

Figure 9.17   Example of Traditional UNIX Process Scheduling

10.3 LINUX SCHEDULING

Linux builds on the traditional UNIX scheduling described in Section 9.3 by adding two scheduling classes for soft real-time processing. The three Linux scheduling classes are:

· **SCHED_FIFO:** First-in-first-out real-time threads
· **SCHED_RR:** Round-robin real-time threads
· **SCHED_OTHER:** Other, non-real-time threads

Within each class, multiple priorities may be used, with priorities in the real-time classes higher than the priorities for the SCHED_OTHER class. For FIFO threads, the following rules apply:

1. The system will not interrupt an executing FIFO thread except in the following cases:
   a. Another FIFO thread of higher priority becomes ready.
   b. The executing FIFO thread becomes blocked waiting for an event, such as I/O.
   c. The executing FIFO thread voluntarily gives up the processor following a call to the primitive *sched_yield*.

2. When an executing FIFO thread is interrupted, it is placed in the queue associated with its priority.

3. When a FIFO thread becomes ready and if that thread has a higher priority than the currently executing thread, then the currently executing thread is preempted and the highest priority ready FIFO thread is executed. If more than one thread has that highest priority, the thread that has been waiting the longest is chosen.

The SCHED_RR policy is similar to the SCHED_FIFO policy, except for the addition of a time quota associated with each thread. When a SCHED_RR thread has executed for its time quota, it is suspended and a real-time thread of equal or higher priority is selected for running.

Figure 10.9, from [COMP98], is an example that illustrates the distinction between FIFO and RR scheduling. Assume a program has four threads with three relative priorities assigned as shown in Figure 10.9a. Assume that all waiting threads are ready to execute when the current thread waits or terminates and that no higher-priority thread is awakened while a thread is executing. Figure 10.9b shows a flow in which all of the threads are in the SCHED_FIFO class. Thread D executes until it waits or terminates. Next, although threads B and C have the same priority, thread B starts because it has been waiting longer than thread C. Thread B executes until it waits or terminates, then thread C executes until it waits or terminates. Finally, thread A executes.

Figure 10.9c shows a sample flow if all of the threads are in the SCHED_RR class. Thread D executes until it waits or terminates. Next, threads B and C are time sliced, because they both have the same priority. Finally, thread A executes.

The final scheduling class is SCHED_OTHER. A thread in this class can only execute if there are no real-time threads ready to execute. Within the SCHED_OTHER class, the traditional UNIX scheduling algorithm described in Section 9.3 is used.

A	minimum
B	middle
C	middle
D	maximum

(a) Relative thread priorities

$$\begin{tikzcd}D \arrow[r] & B \arrow[r] & C \arrow[r] & A \arrow[r] & {}\end{tikzcd}$$

(b) Flow with FIFO scheduling

$$\begin{tikzcd}D \arrow[r] & B \arrow[r] & C \arrow[r] & B \arrow[r] & C \arrow[r] & A \arrow[r] & {}\end{tikzcd}$$

(c) Flow with RR scheduling

Figure 10.9 Example of Linux Scheduling

10.4 UNIX SVR4 SCHEDULING

The scheduling algorithm used in UNIX SVR4 is a complete overhaul of the scheduling algorithm used in earlier UNIX systems (described in Section 9.3). The new algorithm is designed to give highest preference to real-time processes, next-highest preference to kernel-mode processes, and lowest preference to other user-mode processes, referred to as time-shared processes.

The two major modifications implemented in SVR4 are:

1. The addition of a preemptable static priority scheduler and the introduction of a set of 160 priority levels divided into three priority classes.
2. The insertion of preemption points. Because the basic kernel is not preemptive, it can only be split into processing steps that must run to completion without interruption. In between the processing steps, safe places known as preemption points have been identified where the kernel can safely interrupt processing and schedule a new process. A safe place is defined as a region of code where all kernel data structures are either updated and consistent or locked via a semaphore.

Figure 10.10 illustrates the 160 priority levels defined in SVR4. Each process is defined to belong to one of three priority classes and is assigned a priority level within that class. The classes are:

- **Real time (159-100):** Processes at these priority levels are guaranteed to be selected to run before any kernel or time-sharing process. In addition, real-time processes can make use of preemption points to preempt kernel processes and user processes.
- **Kernel (99-60):** Processes at these priority levels are guaranteed to be selected to run before any time-sharing process but must defer to real-time processes.
- **Time-shared (59-0):** The lowest-priority processes, intended for user applications other than real-time applications.

Figure 10.11 indicates how scheduling is implemented in SVR4. A dispatch queue is associated with each priority level, and processes at a given priority level are executed in round-robin fashion. A bit-map vector, `dqactmap`, contains one bit for each priority level; the bit is set to one for any priority level with a nonempty queue. Whenever a running process leaves the Running state, due to a block, time-slice expiration, or preemption, the dispatcher checks `dqactmap` and dispatches a ready process from the highest-priority nonempty queue. In addition, whenever a defined preemption point is reached, the kernel checks a flag called `kprunrun`. If set, this indicates that at least one real-time process is in the Ready state, and the kernel preempts the current process if it is of lower priority than the highest-priority real-time ready process.

Within the time-sharing class, the priority of a process is variable. The scheduler reduces the priority of a process each time it uses up a time quantum, and it raises its priority if it blocks on an event or resource. The time quantum allocated to a time-sharing process depends on its priority, ranging from 100 ms for priority 0 to 10 ms for priority 59. Each real-time process has a fixed priority and a fixed time quantum.

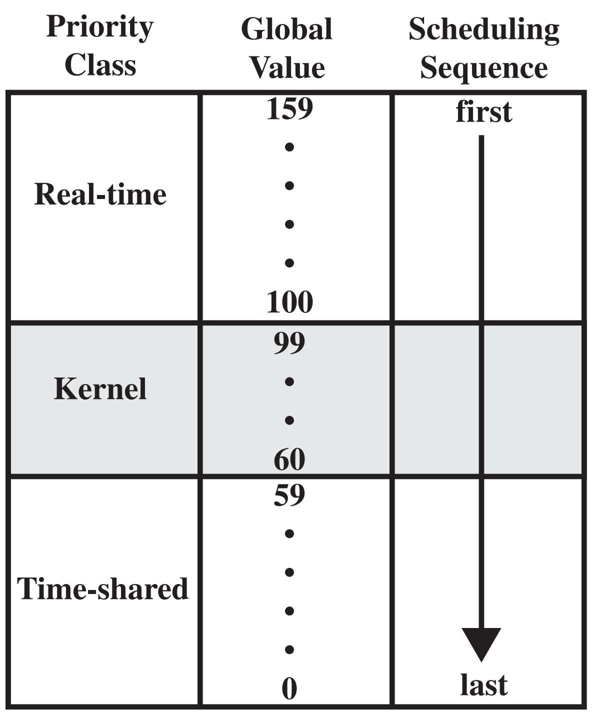


Figure 10.10 SVR4 Priority Classes

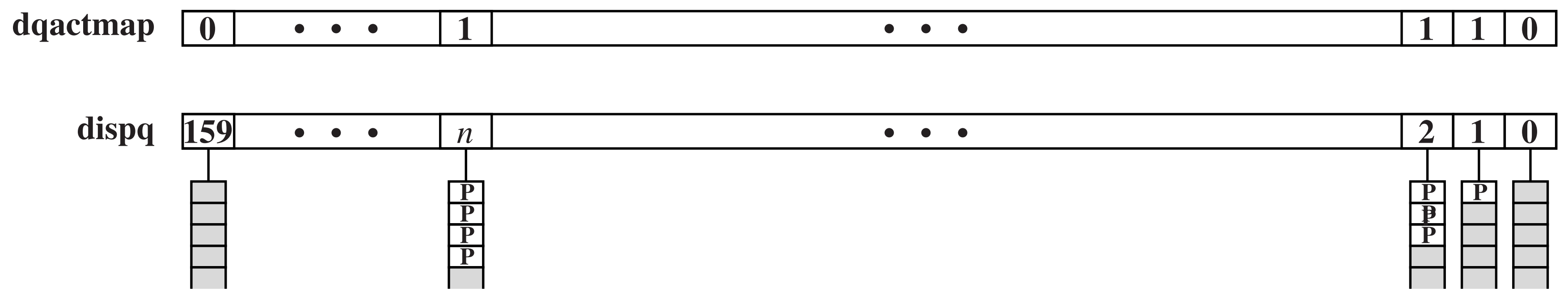


Figure 10.11 SVR4 Dispatch Queues

11.8 UNIX SVR4 I/O

In UNIX, each individual I/O device is associated with a special file. These are managed by the file system and are read and written in the same manner as user data files. This provides a clean, uniform interface to users and processes. To read from or write to a device, read and write requests are made for the special file associated with the device.

Figure 11.14 illustrates the logical structure of the I/O facility. The file subsystem manages files on secondary storage devices. In addition, it serves as the process interface to devices, because these are treated as files.

There are two types of I/O in UNIX: buffered and unbuffered. Buffered I/O passes through system buffers, whereas unbuffered I/O typically involves the DMA facility, with the transfer taking place directly between the I/O module and the process I/O area. For buffered I/O, two types of buffers are used: system buffer caches and character queues.

Buffer Cache

The buffer cache in UNIX is essentially a disk cache. I/O operations with disk are handled through the buffer cache. The data transfer between the buffer cache and the user process space always occurs using DMA. Because both the buffer cache and the process I/O area are in main memory, the DMA facility is used in this case to perform a memory-to-memory copy. This does not use up any processor cycles, but it does consume bus cycles.

To manage the buffer cache, three lists are maintained:

- **Free list:** List of all slots in the cache (a slot is referred to as a buffer in UNIX; each slot holds one disk sector) that are available for allocation
- **Device list:** List of all buffers currently associated with each disk
- **Driver I/O queue:** List of buffers that are actually undergoing or waiting for I/O on a particular device

All buffers should be on the free list or on the driver I/O queue list. A buffer, once associated with a device, remains associated with the device even if it is on the free list, until is actually reused and becomes associated with another device. These lists are maintained as pointers associated with each buffer rather than physically separate lists.

When a reference is made to a physical block number on a particular device, the operating system first checks to see if the block is in the buffer cache. To minimize the search time, the device list is organized as a hash table, using a technique similar to the overflow with chaining technique discussed in Appendix 8A (Figure 8.26b). Figure 11.15 depicts the general organization of the buffer cache. There is a hash table of fixed length that contains pointers into the buffer cache. Each reference to a (device#, block#) maps into a particular entry in the hash table. The pointer in that entry points to the first buffer in the chain. A hash pointer associated with each buffer points to the next buffer in the chain for that hash table entry. Thus, for all (device#, block#) references that map into the same hash table entry, if the corresponding block is in the buffer cache, then that buffer will be in the chain for that hash table entry. Thus, the length of the search of the buffer cache is reduced by a factor of on the order of *N*, where *N* is the length of the hash table.

For block replacement, a least-recently-used algorithm is used: After a buffer has been allocated to a disk block, it cannot be used for another block until all other buffers have been used more recently. The free list preserves this least-recently-used order.

Character Queue

Block-oriented devices, such as disk and tape, can be effectively served by the buffer cache. A different form of buffering is appropriate for character-oriented devices, such as terminals and printers. A character queue is either written by the I/O device and read by the process or written by the process and read by the device. In both cases, the producer/consumer model introduced in Chapter 5 is used. Thus, character queues may only be read once; as each character is read, it is effectively destroyed. This is in contrast to the buffer cache, which may be read multiple times and hence follows the readers/writers model (also discussed in Chapter 5).

Unbuffered I/O

Unbuffered I/O, which is simply DMA between device and process space, is always the fastest method for a process to perform I/O. A process that is performing unbuffered I/O is locked in main memory and cannot be swapped out. This reduces the opportunities for swapping by tying up part of main memory, thus reducing the overall system performance. Also, the I/O device is tied up with the process for the duration of the transfer, making it unavailable for other processes.

UNIX Devices

UNIX recognizes five types of devices:

- Disk drives
- Tape drives
- Terminals
- Communication lines
- Printers

Table 11.5 shows the types of I/O suited to each type of device. Disk drives are heavily used in UNIX, are block oriented, and have the potential for reasonable high throughput. Thus, I/O for these devices tends to be unbuffered or via buffer cache. Tape drives are functionally similar to disk drives and use similar I/O schemes.

Because terminals involve relatively slow exchange of characters, terminal I/O typically makes use of the character queue. Similarly, communication lines require serial processing of bytes of data for input or output and are best handled by character queues. Finally, the type of I/O used for a printer will generally depend on its speed. Slow printers will normally use the character queue, while a fast printer might employ unbuffered I/O. A buffer cache could be used for a fast printer. However, because data going to a printer are never reused, the overhead of the buffer cache is unnecessary.

Table 11.5 Device I/O in UNIX

	Unbuffered I/O	Buffer Cache	Character Queue
Disk drive	X	X	
Tape drive	X	X	
Terminals			X
Communication lines			X
Printers	X		X

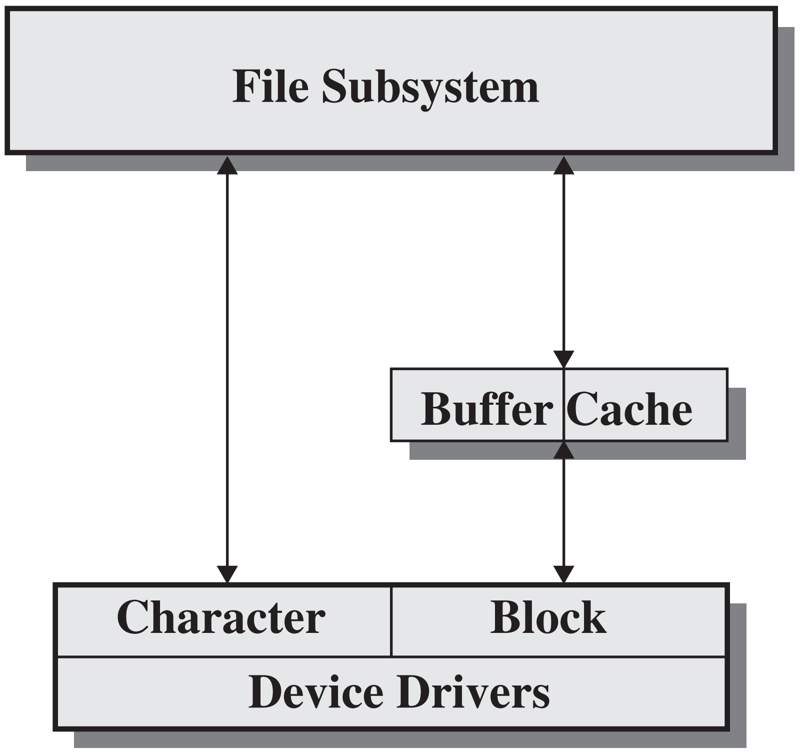


Figure 11.14 UNIX I/O Structure

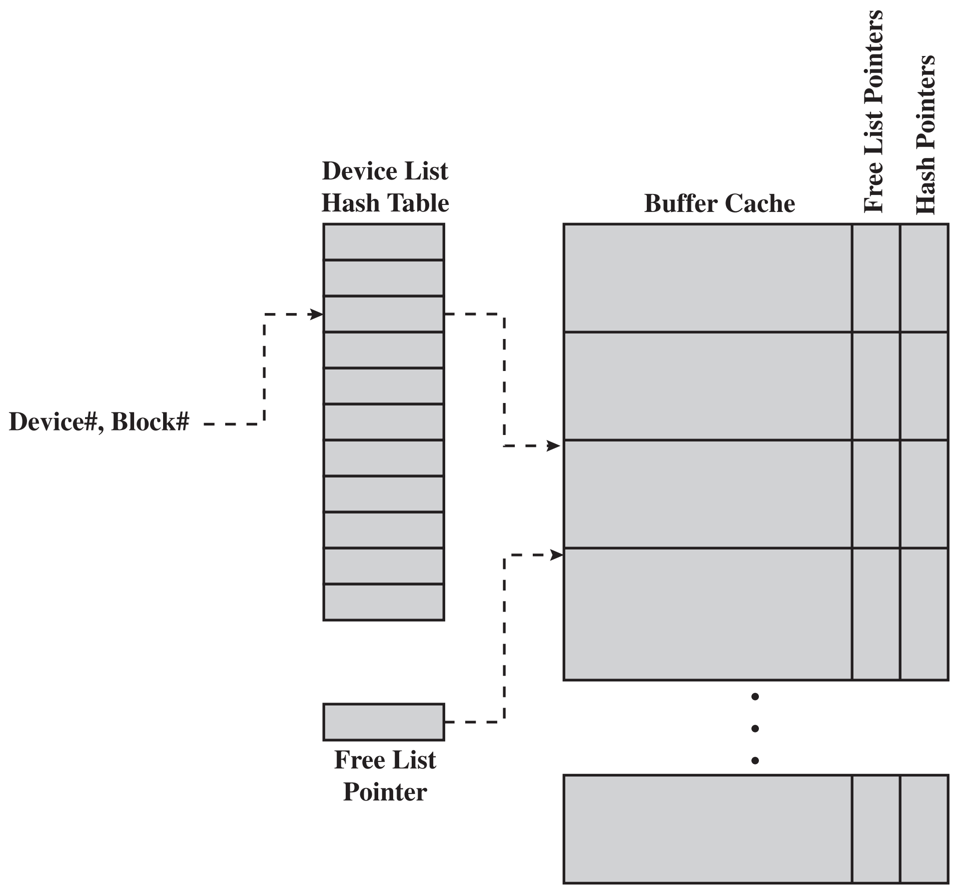


Figure 11.15 UNIX Buffer Cache Organization

12.7 UNIX FILE MANAGEMENT

The UNIX kernel views all files as streams of bytes. Any internal logical structure is application specific. However, UNIX is concerned with the physical structure of files.

Four types of files are distinguished:

- **Ordinary:** Files that contain information entered in them by a user, an application program, or a system utility program.
- **Directory:** Contains a list of file names plus pointers to associated inodes (index nodes), described later. Directories are hierarchically organized (Figure 12.4). Directory files are actually ordinary files with special write protection privileges so that only the file system can write into them, while read access is available to user programs.
- **Special:** Used to access peripheral devices, such as terminals and printers. Each I/O device is associated with a special file, as discussed in Section 11.7.
- **Named:** Named pipes, as discussed in Section 6.7.

In this section, we are concerned with the handling of ordinary files, which correspond to what most systems treat as files.

Inodes

All types of UNIX files are administered by the operating system by means of inodes. An inode (information node) is a control structure that contains the key information needed by the operating system for a particular file. Several file names may be associated with a single inode, but an active inode is associated with exactly one file, and each file is controlled by exactly one inode.

The attributes of the file as well as its permissions and other control information are stored in the inode. Table 12.4 lists the contents.

File Allocation

File allocation is done on a block basis. Allocation is dynamic, as needed, rather than using preallocation. Hence, the blocks of a file on disk are not necessarily contiguous. An indexed method is used to keep track of each file, with part of the index stored in the inode for the file. The inode includes 39 bytes of address information that is organized as thirteen 3-byte addresses, or pointers. The first 10 addresses point to the first 10 data blocks of the file. If the file is longer than 10 blocks long, then one or more levels of indirection is used as follows:

- The eleventh address in the inode points to a block on disk that contains the next portion of the index. This is referred to as the single indirect block. This block contains the pointers to succeeding blocks in the file.
- If the file contains more blocks, the twelfth address in the inode points to a double indirect block. This block contains a list of addresses of additional single indirect blocks. Each of single indirect blocks, in turn, contains pointers to file blocks.
- If the file contains still more blocks, the thirteenth address in the inode points to a triple indirect block that is a third level of indexing. This block points to additional double indirect blocks.

All of this is illustrated in Figure 12.13. The total number of data blocks in a file depends on the capacity of the fixed-size blocks in the system. In UNIX System V, the length of a block is 1 Kbyte, and each block can hold a total of 256 block addresses. Thus, the maximum size of a file with this scheme is over 16 Gbytes (Table 12.5).

This scheme has several advantages:

1. The inode is of fixed size and relatively small and hence may be kept in main memory for long periods.
2. Smaller files may be accessed with little or no indirection, reducing processing and disk access time.
3. The theoretical maximum size of a file is large enough to satisfy virtually all applications.

Table 12.4 Information in a UNIX Disk-Resident Inode

File Mode	16-bit flag that stores access and execution permissions associated with the file.	
	12-14	File type (regular, directory, character or block special, FIFO pipe
	9-11	Execution flags
	8	Owner read permission
	7	Owner write permission
	6	Owner execute permission
	5	Group read permission
	4	Group write permission
	3	Group execute permission
	2	Other read permission
	1	Other write permission
	0	Other execute permission
Link Count	Number of directory references to this inode	
Owner ID	Individual owner of file	
Group ID	Group owner associated with this file	
File Size	Number of bytes in file	
File Addresses	39 bytes of address information	
Last Accessed	Time of last file access	
Last Modified	Time of last file modification	
Inode Modified	Time of last inode modification	

Table 12.5 Capacity of a UNIX File

Level	Number of Blocks	Number of Bytes
Direct	10	10K
Single Indirect	256	256K
Double Indirect	256 × 256 = 65K	65M
Triple Indirect	256 × 65K = 16M	16G

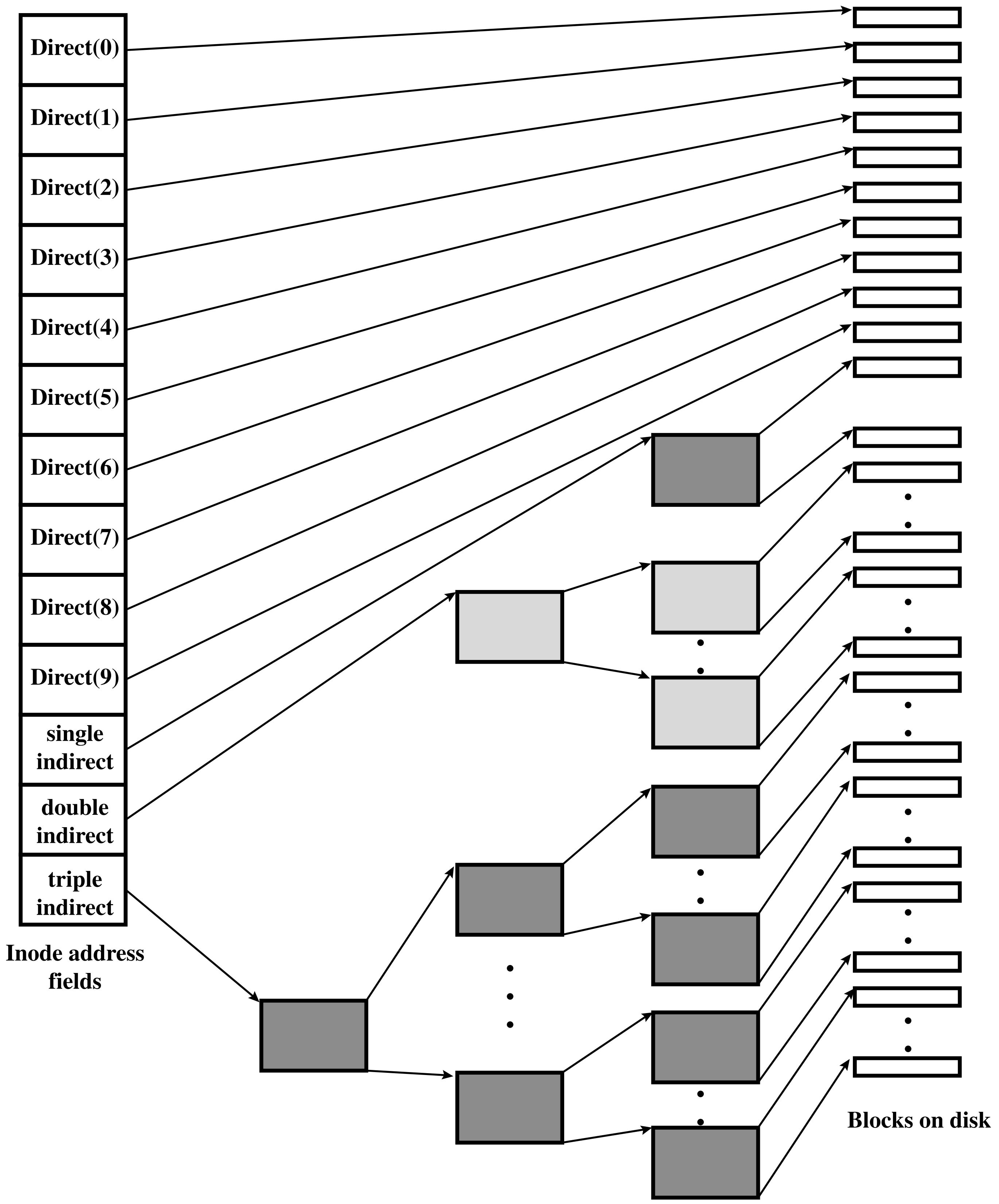


Figure 12.13 UNIX Block Addressing Scheme

13.6 SUN CLUSTER

Sun Cluster is a distributed operating system built as a set of extensions to the base Solaris UNIX system. It provides cluster with a single-system image; that is, the cluster appears to the user and applications as a single computer running the Solaris operating system.

Figure 13.17 shows the overall architecture of Sun Cluster. The major components are:

· Object and communication support
· Process management
· Networking
· Global distributed file system

Object and Communication Support

The Sun Cluster implementation is object oriented. The CORBA object model (see Appendix B) is used to define objects and the remote procedure call (RPC) mechanism implemented in Sun Cluster. The CORBA Interface Definition Language (IDL) is used to specify interfaces between MC components in different nodes. The elements of MC are implemented in the object-oriented language C++. The use of a uniform object model and IDL provides a mechanism for internode and intranode interprocess communication. All of this is built on top of the Solaris kernel with virtually no changes required to the kernel.

Process Management

Global process management extends process operations so that the location of a process is transparent to the user. Sun Cluster maintains a global view of processes so that there is a unique identifier for each process in the cluster and so that each node can learn the location and status of each process. Process migration (described in Chapter 14) is possible: a process can move from one node to another during its lifetime, to achieve load balancing or for failover. However, the threads of a single process must be on the same node.

Networking

The designers of Sun Cluster considered three approaches for handling network traffic:

1. Perform all network protocol processing on a single node. In particular, for a TCP/IP-based application, incoming (and outgoing) traffic would go through a network-connection node that for incoming traffic would analyze TCP and IP headers and route the encapsulated data to the appropriate node; and for outgoing traffic would encapsulate data from other nodes in TCP/IP headers. This approach is not scalable to a large number of nodes and so was rejected.

2. Assign a unique IP address to each node and run the network protocols over the external network directly to each node. One difficulty with this approach is that the cluster configuration is no longer transparent to the outside world. Another complication is the difficulty of failover when a running application moves to another node with a different underlying network address.

3. Use a packet filter to route packets to the proper node and perform protocol processing on that node. Externally, the cluster appears as a single server with a single IP address. Incoming connections (client requests) are load balanced among the available nodes of the cluster. This is the approach adopted in Sun Cluster.

The Sun Cluster networking subsystem has three key elements:

1. Incoming packets are first received on the node that has the network adapter physically attached to it; the receiving node filters the packet and delivers it to the correct target node over the cluster interconnect.
2. All outgoing packets are routed over the cluster interconnect to the node (or one of multiple alternative nodes) that has an external network physical connection. All protocol processing for outgoing packets is done by the originating node.
3. A global network configuration database is maintained to keep track of network traffic to each node.

Global File System

The most important element of Sun Cluster is the global file system, depicted in Figure 13.18, which contrasts MC file management with the basic Solaris scheme. Both are built on the use of vnode and virtual file system concepts.

In Solaris, the virtual node (vnode) structure is used to provide a powerful, general-purpose interface to all types of file systems. A vnode is used to map pages of memory into the address space of a process and to permit access to a file system. While an inode is used to map processes to UNIX files, a vnode can map a process to an object in any file system type. In this way, a system call need not understand the actual object being manipulated, only how to make the proper object-oriented type call using the vnode interface. The vnode interface accepts general-purpose file manipulation commands, such as read and write, and translates them into actions appropriate for the subject file system. Just as vnodes are used to describe individual file system objects, the virtual file system (vfs) structures are used to describe entire file systems. The vfs interface accepts general-purpose commands that operate on entire files and translates them into actions appropriate for the subject file system.

In Sun Cluster, the global file system provides a uniform interface to files distributed over the cluster. A process can open a file located anywhere in the cluster, and processes on all nodes use the same pathname to locate a file. To implement global file access, MC includes a proxy file system built on top of the existing Solaris file system at the vnode interface. The vfs/vnode operations are converted by a proxy layer into object invocations (see Figure 13.18b). The invoked object may reside on any node in the system. The invoked object performs a local vnode/vfs operation on the underlying file system. Neither the kernel nor the existing file systems have to be modified to support this global file environment.

To reduce the number of remote object invocations, caching is used. Sun Cluster supports caching of file contents, directory information, and file attributes.

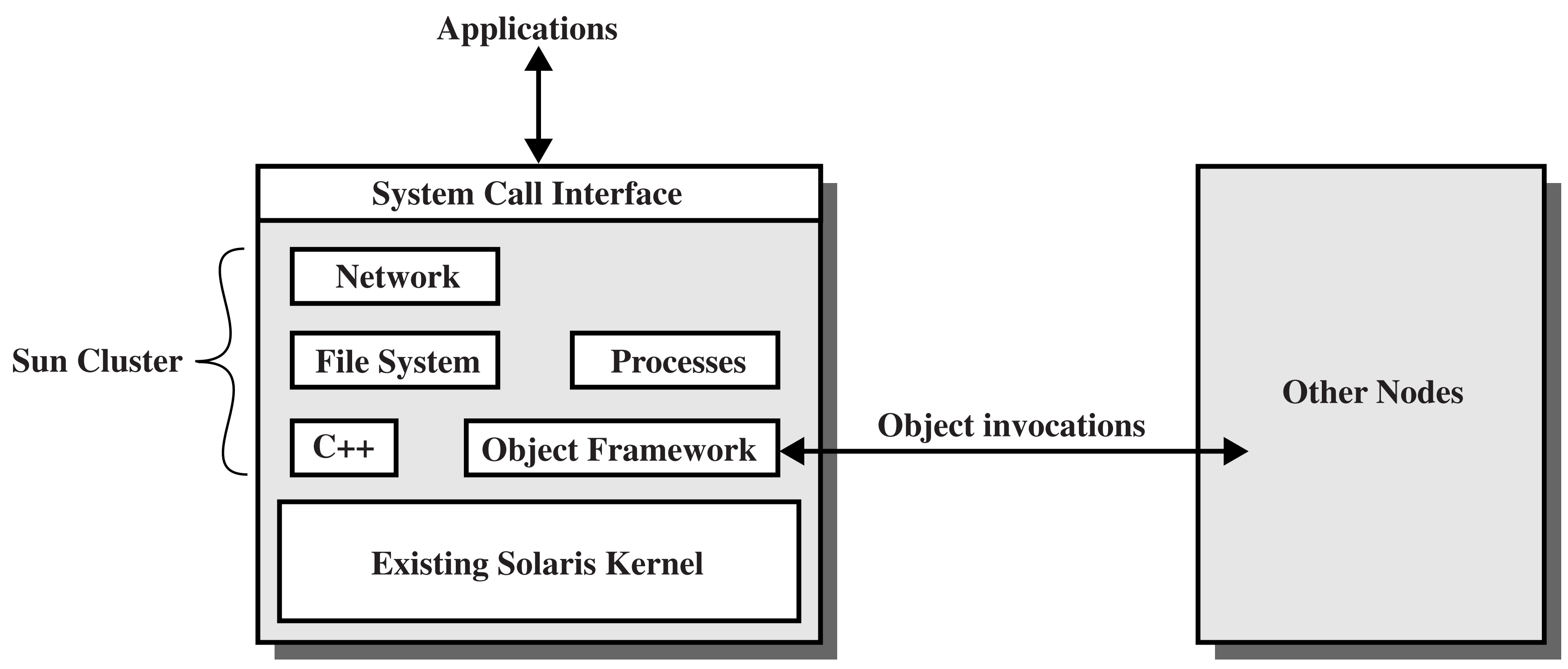


Figure 13.17   Sun Cluster Structure

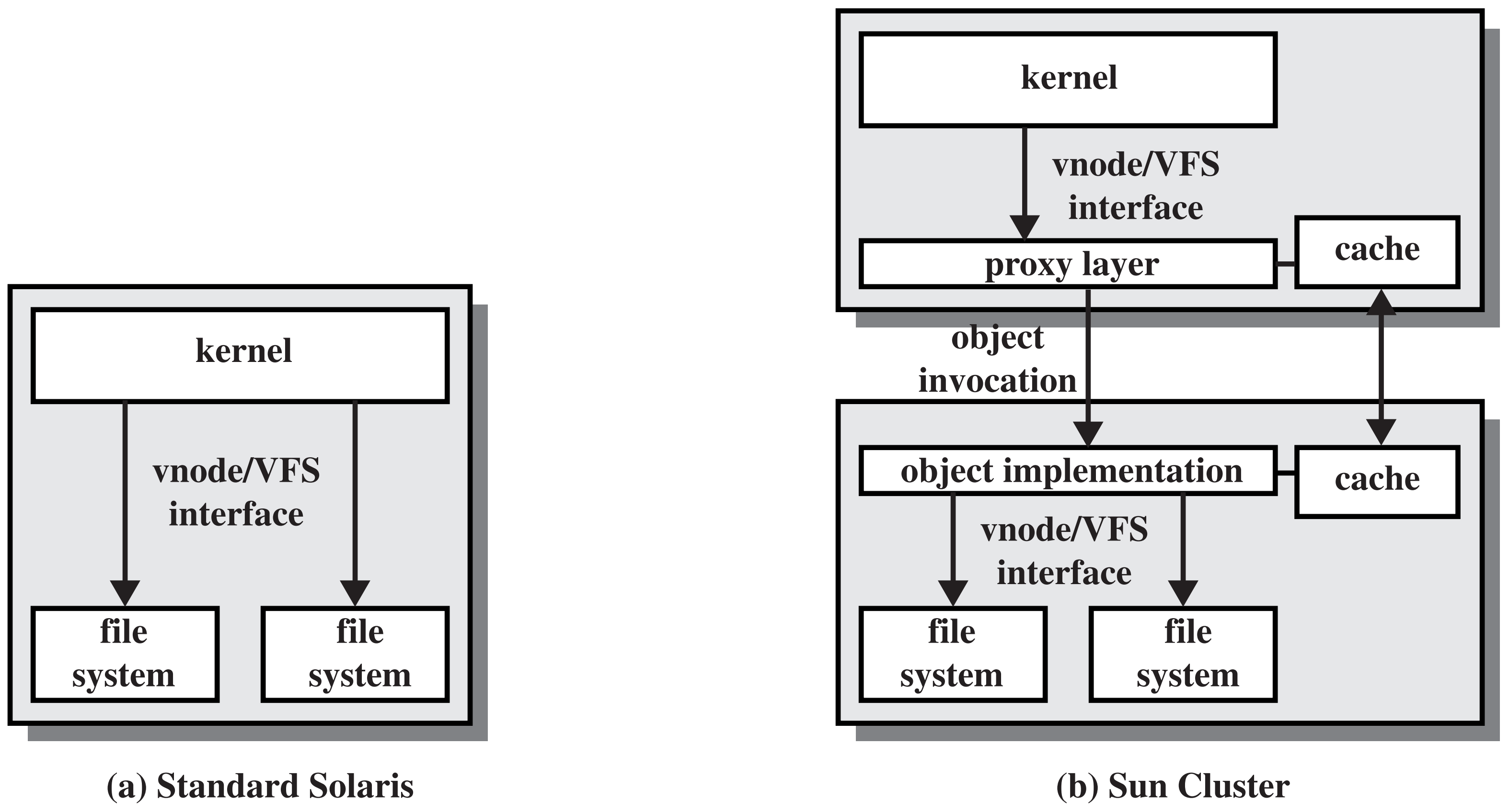


Figure 13.18 Sun Cluster File System Extensions

13.7 BEOWULF AND LINUX CLUSTERS

In 1994, the Beowulf project was initiated under the sponsorship of the NASA High Performance Computing and Communications (HPCC) project. Its goal was to investigate the potential of clustered PCs for performing important computation tasks beyond the capabilities of contemporary workstations at minimum cost. Today, the Beowulf approach is widely implemented and is perhaps the most important cluster technology available.

Beowulf Features

Key features of Beowulf include [RIDG97]:

- Mass market commodity components
- Dedicated processors (rather than scavenging cycles from idle workstations)
- A dedicated, private network (LAN or WAN or internetted combination)
- No custom components
- Easy replication from multiple vendors
- Scalable I/O
- A freely available software base
- Use of freely available distribution computing tools with minimal changes
- Return of the design and improvements to the community

Although elements of Beowulf software have been implemented on a number of different platforms, the most obvious choice for a base is Linux, and most Beowulf implementations use a cluster of Linux workstations and/or PCs. Figure 13.19 depicts a representative configuration. The cluster consists of a number of workstations, perhaps of differing hardware platforms, all running the Linux operating system. Secondary storage at each workstation may be made available for distributed access (for distributed file sharing, distributed virtual memory, or other uses). The cluster nodes (the Linux boxes) are interconnected with a commodity networking approach, typically Ethernet. The Ethernet support may be in the form of a single Ethernet switch or an interconnected set of switches. Commodity Ethernet products at the standard data rates (10 Mbps, 100 Mbps, 1 Gbps) are used.

Beowulf Software

The Beowulf software environment is implemented as an add-on to commercially available, royalty-free base Linux distributions. The principal source of open-source Beowulf software is the Beowulf site at www.beowulf.org, but numerous other organizations also offer free Beowulf tools and utilities.

Each node in the Beowulf cluster runs its own copy of the Linux kernel and can function as an autonomous Linux system. To support the Beowulf cluster concept, extensions are made to the Linux kernel to allow the individual nodes to participate in a number of global namespaces. Some examples of Beowulf system software:

- **Beowulf distributed process space (BPROC):** This package allows a process ID space to span multiple nodes in a cluster environment and also provides mechanisms for starting processes on other nodes. The goal of this package is to provide key elements needed for a single system image on Beowulf cluster. BPROC provides a mechanism to start processes on remote nodes without ever logging into another node and by making all the remote processes visible in the process table of the cluster's front end node.
- **Beowulf Ethernet Channel Bonding:** This is a mechanism that joins multiple low-cost networks into a single logical network with higher bandwidth. The only additional work over a using single network interface is the computationally simple task of distributing the

packets over the available device transmit queues. This approach allows load balancing over multiple Ethernets connected to Linux workstations.

- **Pvmsync:** This is a programming environment that provides synchronization mechanisms and shared data objects for processes in a Beowulf cluster.
- **EnFuzion:** EnFuzion consists of a set of tools for doing parametric computing, as described in Section 13.4. Parametric computing involves the execution of a program as a large number of jobs, each with different parameters or starting conditions. EnFusion emulates a set of robot users on a single root node machine, each of which will log into one of the many client node machines the form a cluster. Each job is set up to run with a unique, programmed scenario, with an appropriate set of starting conditions [KAPP00].

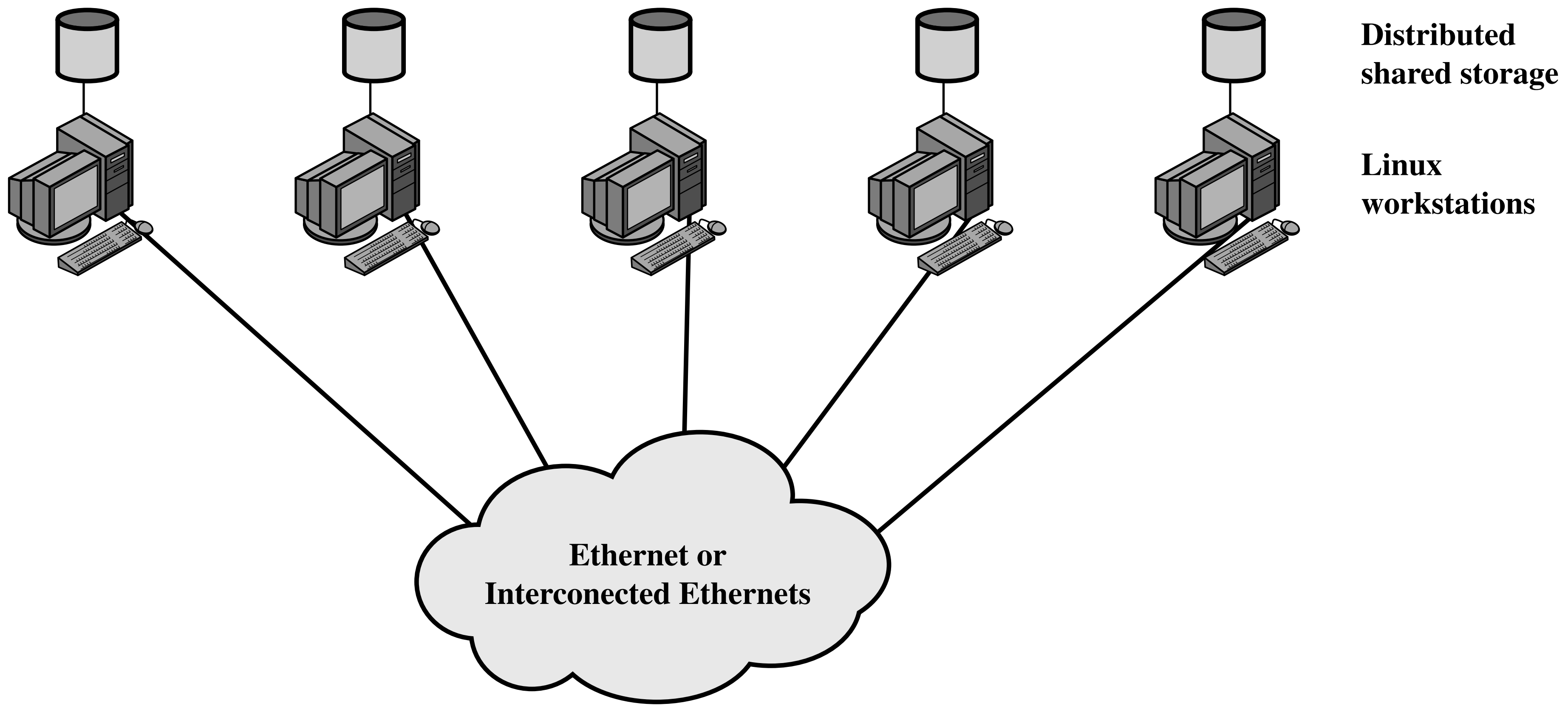


Figure 13.19 Generic Beowulf Configuration