1

AN INTRODUCTION TO SOLARIS

The UNIX system is very successful. At the time of writing there are over 3000 UNIX systems in active use throughout the world.

—S.R. Bourne, *The UNIX System*, 1983

Sun systems have shipped with a UNIX-based operating system since the first Sun-1 workstation was introduced in 1982. Steve Bourne's quote indicates how relatively small the UNIX market was at that time. Today, millions of UNIX systems are deployed, running a variety of applications ranging from single-user systems, to real-time control systems, to mission- and business-critical environments—and Solaris represents a large percentage of these systems.

The Solaris installed base has rapidly increased its since its inception. It is available on SPARC processor architectures from Sun and OEMs and on standard Intel-based systems. Solaris scales from single-processor systems to the 64-processor Sun Enterprise 10000 system.

1.1 A Brief History

Sun's UNIX operating environment began life as a port of BSD UNIX to the Sun-1 workstation. The early versions of Sun's UNIX were known as SunOS, which is the name used for the core operating system component of Solaris.

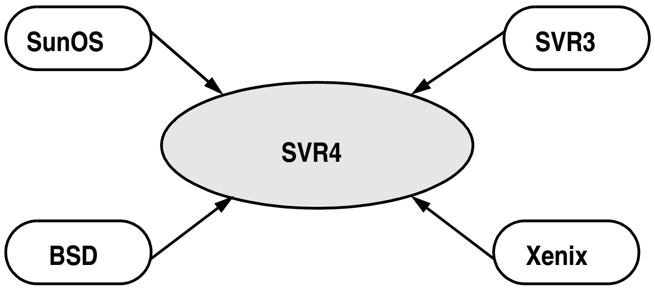
SunOS 1.0 was based on a port of BSD 4.1 from Berkeley labs in 1982. At that time, SunOS was implemented on Sun's Motorola 68000-based uniprocessor workstations. SunOS was small and compact, and the workstations had only a few MIPS of processor speed and around one megabyte of memory.

In the early to mid-1980s, networked UNIX systems were growing in popularity; networking was becoming ubiquitous and was a major part of Sun's computing strategy. Sun invested significant resources in developing technology that enabled distributed, network-based computing. These technologies included interfaces for building distributed applications (remote procedure calls, or RPC), and operating system facilities for the sharing of data over networks (Network Information System, or NIS) and a distributed computing file system: NFS. The incorporation of remote file sharing into SunOS required extensive operating system changes. In 1984, SunOS 2.0 offered the virtual file system framework to implement multiple file system types, which allowed support for the NFS file system. The network file system source was made openly licensable and has subsequently been ported to almost every modern operating system platform in existence today.

The volume of applications running on the Sun platform increased steadily, with each new application placing greater demand on the system, providing the catalyst for the next phase of innovation. Applications needed better facilities for the sharing of data and executable objects. The combination of the need for shared program libraries, memory mapped files, and shared memory led to a major re-architecting of the SunOS virtual memory system. The new virtual memory system, introduced as SunOS version 4, abstracted various devices and objects as virtual memory, facilitating the mapping of files, sharing of memory, and mapping of hardware devices into a process.

During the 1980s, the demand for processing capacity outpaced the industry's incremental improvements in processor speed. To satisfy the demand, systems were developed with multiple processors sharing the same system memory and Input/Output (I/O) infrastructure, an advance that required further operating system changes. An asymmetric multiprocessor implementation first appeared in SunOS 4.1—the kernel could run on only one processor at a time, while user processors could be scheduled on any of the available processors. Workloads with multiple processes could often obtain greater throughput on systems with more than one processor. The asymmetric multiprocessor implementation was a great step forward; however, scalability declined rapidly as additional processors were added. The need for a better multiprocessor implementation was obvious.

At this time, Sun was participating in a joint development with AT&T, and the SunOS virtual file system framework and virtual memory system became the core of UNIX System V Release 4 (SVR4). SVR4 UNIX incorporated the features from SunOS, SVR3, BSD UNIX, and Xenix, as shown below. International Computers Limited (ICL) ported the new SVR4 UNIX to the SPARC processor architecture and delivered the reference source for SVR4 on SPARC.



With the predicted growth in multiprocessor systems, Sun invested heavily in the development of a new operating system kernel with a primary focus on multiprocessor scalability. The new kernel allowed multiple threads of execution and provided facilities for threading at the process (application) level. Together with fine-grained locking, the new kernel provided the foundation for the scalability found in Solaris today. The new kernel and the SVR4 operating environment became the basis for Solaris 2.0.

This change in the base operating system was accompanied by a new naming convention; the Solaris name was introduced to describe the operating environment, of which SunOS, the base operating system, is a subset. Thus, the older SunOS retained the SunOS 4.X versioning and adopted Solaris 1.X as the operating environment version. The SVR4-based environment adopted a SunOS 5.X versioning (SunOS 5.0 being the first release) with the Solaris 2.X operating environment. The naming convention has resulted in most people referring to the pre-SVR4 releases as SunOS, and the SVR4-based releases as Solaris. Table 1-1 traces the development of Solaris from its roots to Solaris 7.

The new Solaris 2.0 operating environment was built in a modular fashion, which made possible its implementation on multiple platforms with different instruction set architectures. In 1993, Solaris was made available for Intel PC-based architectures, greatly expanding the platforms on which Solaris is available. In October 1999, Sun announced support for Solaris on the Intel Itanium processor.

The next major milestone was the introduction of a 64-bit implementation, in Solaris 7. Full 64-bit support allows the kernel and processes to access large address spaces and to use extended 64-bit data types. Solaris 7 also provides full compatibility for existing 32-bit applications, supporting concurrent execution of 32-bit and 64-bit applications.

Table 1-1 Solaris Release History

Date	Release	Notes
1982	Sun UNIX 0.7	• First version of Sun's UNIX, based on 4.BSD from UniSoft. • Bundled with the Sun-1, Sun's first workstation based on the Motorola 68000 processor; SunWindows GUI.
1983	SunOS 1.0	• Sun-2 workstation, 68010 based.
1985	SunOS 2.0	• Virtual file system (VFS) and `vnode` framework allows multiple concurrent file system types. • NFS implemented with the VFS/`vnode` framework.
1988	SunOS 4.0	• New virtual memory system integrates the file system cache with the memory system. • Dynamic linking added. • The first SPARC-based Sun workstation, the Sun-4. Support for the Intel-based Sun 386i.
1990	SunOS 4.1	• Supports the SPARCstation1+, IPC, SLC. • OpenWindows graphics environment
1992	SunOS 4.1.3	• Asymmetric multiprocessing (ASMP) for sun4m systems (SPARCstation-10 and -600 series MP (multiprocessor) servers).
1992	Solaris 2.0	• Solaris 2.x is born, based on a port of System V Release 4.0. • VFS/`vnode`, VM system, intimate shared memory brought forward from SunOS. • Uniprocessor only. • First release of Solaris 2, version 2.0, is a desktop-only developers release.
1992	Solaris 2.1	• Four-way symmetric multiprocessing (SMP).
1993	Solaris 2.2	• Large (> 2 Gbyte) file system support. • SPARCserver 1000 and SPARCcenter 2000 (sun4d architecture).
1993	Solaris 2.1-x86	• Solaris ported to the Intel i386 architecture.
1993	Solaris 2.3	• 8-way SMP. • Device power management and system suspend/resume functionality added. • New directory name lookup cache.

Table 1-1 Solaris Release History (Continued)

Date	Release	Notes
1994	Solaris 2.4	• 20-way SMP. • New kernel memory allocator (slab allocator) replaces SVR4 buddy allocator. • Caching file system (cachefs). • CDE windowing system.
1995	Solaris 2.5	• Large-page support for kernel and System V shared memory. • Fast local interprocess communication (Doors) added. • NFS Version 3. • Supports sun4u (UltraSPARC) architecture. UltraSPARC-I-based products introduced—the Ultra-1 workstation.
1996	Solaris 2.5.1	• First release supporting multiprocessor Ultra-SPARC-based systems. • 64-way SMP. • Ultra-Enterprise 3000–6000 servers introduced.
1996	Solaris 2.6	• Added support for large (> 2 Gbyte files). • Dynamic processor sets. • Kernel-based TCP sockets. • Locking statistics. • UFS direct I/O. • Dynamic reconfiguration.
1998	Solaris 7	• 64-bit kernel and process address space. • Logging UFS integrated. • Priority Paging memory algorithm.

The information in Table 1-1 shows the significant features incorporated in each major release of Solaris. Details of *all* of the features can be found in the Solaris release *What's New* document, which is part of the documentation supplied with Solaris.

1.2 Key Differentiators

Solaris development continued aggressively throughout the 1990s. Several key features distinguish Solaris from earlier UNIX implementations.

- **Symmetric multiprocessing** — Solaris is implemented on systems ranging from single-processor systems to 64-processor symmetric multiprocessor servers. Solaris provides linear scalability up to the currently supported maximum of 64 processors.
- **64-bit kernel and process address space** — A 64-bit kernel for 64-bit platforms provides an LP64 execution environment. (LP64 refers to the data model: long and pointer data types are 64 bits wide.) A 32-bit application environment is also provided so that 32-bit binaries execute on a 64-bit Solaris kernel alongside 64-bit applications.
- **Multiple platform support** — Solaris supports a wide range of SPARC and Intel x86 microprocessor-based architectures. A layered architecture means that over 90 percent of the Solaris source is platform independent.
- **Modular binary kernel** — The Solaris kernel uses dynamic linking and dynamic modules to divide the kernel into modular binaries. A core kernel binary contains central facilities; device drivers, file systems, schedulers, and some system calls are implemented as dynamically loadable modules. Consequently, the Solaris kernel is delivered as a binary rather than source and object, and kernel compiles are not required upon a change of parameters or addition of new functionality.
- **Multithreaded process execution** — A process can have more than one thread of execution, and these threads can run concurrently on one or more processors. Thus, a single process can use multiple processors for concurrent thread execution, thereby using multiprocessor platforms more efficiently.
- **Multithreaded kernel** — The Solaris kernel uses threads as the entity for scheduling and execution: The kernel schedules interrupts and kernel services as regular kernel threads. This key feature provides interrupt scalability and low-latency interrupt response.

  Previous UNIX implementations manipulated processor priority levels to ensure exclusive access to critical interrupt data structures. As a result, the inability of interrupt code to block led to poor scalability. Solaris provides greater parallelism by scheduling interrupts as threads, which can then use regular kernel locks to ensure exclusive access to data structures.
- **Fully preemptable kernel** — The Solaris kernel is fully preemptable and does not require manipulation of hardware interrupt levels to protect critical data—locks synchronize access to kernel data. This means threads that need to run can interrupt another, lower-priority thread; hence, low latency scheduling and low latency interrupt dispatch become possible. For example, a process waking up after sleeping for a disk I/O can be scheduled immediately,

rather than waiting until the scheduler runs. Additionally, by not raising priority levels and blocking interrupts, the system need not periodically suspend activity during interrupt handling, so system resources are used more efficiently.

- **Support for multiple schedulers** — Solaris provides a configurable scheduler environment. Multiple schedulers can operate concurrently, each with its own scheduling algorithms and priority levels. Schedulers are supplied as kernel modules and are dynamically loaded into the operating system. Solaris offers a table-driven, usage-decayed, timesharing user scheduler (TS); a window system optimized timeshare scheduler (IA); and a real-time fixed priority scheduler (RT). An optional fair-share scheduler class can be loaded with the Solaris Resource Manager package.
- **Support for multiple file systems** — Solaris provides a virtual file system (VFS) framework that allows multiple file systems to be configured into the system. The framework implements several disk-based file systems (UNIX File System, MS-DOS file system, CD-ROM file system, etc.) and the network file system (NFS V2 and V3). The virtual file system framework also implements pseudo file systems, including the process file system, procfs, a file system that abstracts processes as files. The virtual file system framework is integrated with the virtual memory system to provide dynamic file system caching that uses available free memory as a file system cache.
- **Processor partitioning and binding** — Special facilities allow fine-grained processor control, including binding processes to processors. Processors can be configured into scheduling groups to partition system resources.
- **Demand-paged virtual memory system** — This feature allows systems to load applications on demand, rather than loading whole executables or library images into memory. Demand-paging speeds up application startup and potentially reduces memory footprint.
- **Modular virtual memory system** — The virtual memory system separates virtual memory functions into distinct layers; the address space layer, segment drivers, and hardware-specific components are consolidated into a hardware address translation (HAT) layer. Segment drivers can abstract memory as files, and files can be memory-mapped into an address space. Segment drivers enable different abstractions, including physical memory and devices, to appear in an address space.
- **Modular device I/O system** — Dynamically loadable device and bus drivers allow a hierarchy of buses and devices to be installed and configured. A device driver interface (DDI) shields device drivers from platform-specific infrastructure, thus maximizing portability of device drivers.
- **Integrated networking** — With the data link provider interface (DLPI), multiple concurrent network interfaces can be configured, and a variety of different protocols—including Ethernet, X.25, SDLC, ISDN, FDDI, token bus, bi-sync, and other datalink-level protocols—can be configured upon them.

- **Integrated Internet protocol** — Solaris implements TCP/IP by use of the DLPI interfaces.
- **Real-time architecture** — The Solaris kernel was designed and implemented to provide real-time capabilities. The combination of the preemptive kernel, kernel interrupts as threads, fixed priority scheduling, high-resolution timers, and fine-grained processor control makes Solaris an ideal environment for real-time applications.

The differentiators listed above represent many innovative features integrated in the Solaris kernel. In the remaining chapters, we closely examine the core modules and major subsystems of the kernel.

1.3 Kernel Overview

The Solaris kernel is the core of Solaris. It manages the system hardware resources and provides an execution environment for user programs. The Solaris kernel supports an environment in which multiple programs can execute simultaneously. The primary functions of the kernel can be divided into two major categories: managing the hardware by allocating its resources among the programs running on it; and supplying a set of system services for those programs to use.

The Solaris kernel, like that of other operating systems implementations, provides a virtual machine environment that shields programs from the underlying hardware and allows multiple programs to execute concurrently on the hardware platform. Each program has its own virtual machine environment, with an execution context and state.

The basic unit that provides a program's environment is known as a *process*; it contains a virtual memory environment that is insulated from other processes on the system. Each Solaris process can have one or more *threads of execution* that share the virtual memory environment of the process, and each thread in effect executes concurrently within the process's environment. The Solaris kernel scheduler manages the execution of these threads (as opposed to management by scheduling processes) by transparently time-slicing them onto one or more *processors*. The threads of execution start and stop executing as they are moved on and off the processors, but the user program is unaware of this. Each time a thread is moved off a processor, its complete execution environment (program counter, stack pointers, registers, etc.) is saved, so when it is later rescheduled onto a processor, its environment can be restored and execution can resume. Processes and scheduling are covered in detail in Part 3 of this book.

The kernel provides mechanisms to access operating system services, such as file I/O, networking, process and thread creation and termination, process control and signaling, process memory management, and interprocess communication. A process accesses these kernel services through the use of *system calls*. System calls are programming interfaces through which the operating system is entered so that the kernel can perform work on behalf of the calling thread.

1.3.1 Solaris Kernel Architecture

The Solaris kernel is grouped into several key components and is implemented in a modular fashion. The key components of the Solaris kernel are described in the following list and illustrated in Figure 1.1.

- **System Call Interface** — The system call interface allows user processes to access kernel facilities. The system call layer consists of a common system call handler, which vectors system calls into the appropriate kernel modules.
- **Process Execution and Scheduling** — Process management provides facilities for process creation, execution, management, and termination. The scheduler implements the functions that divide the machine's processor resources among threads on the system. The scheduler allows different scheduling classes to be loaded for different behavior and scheduling requirements.
- **Memory Management** — The virtual memory system manages mapping of physical memory to user processes and the kernel. The Solaris memory management layer is divided into two layers: the common memory management functions and the hardware-specific components. The hardware-specific components are located in the hardware address translation (HAT) layer.
- **File Systems** — Solaris implements a virtual file system framework, by which multiple types of file system can be configured into the Solaris kernel at the same time. Regular disk-based file systems, network file systems, and pseudo file systems are implemented in the file system layer.
- **I/O Bus and Device Management** — The Solaris I/O framework implements bus nexus node drivers (bus-specific architectural dependencies, e.g., a PCI bus) and device drivers (a specific device on a bus, e.g., an Ethernet card) as a hierarchy of modules, reflecting the physical layout of the bus/device interconnect.
- **Kernel Facilities (Clocks, timers, etc.)** — Central kernel facilities, including regular clock interrupts, system timers, synchronization primitives, and loadable module support.
- **Networking** — TCP/IP protocol support and related facilities. The Solaris networking subsystem is implemented as streams-based device drivers and streams modules.

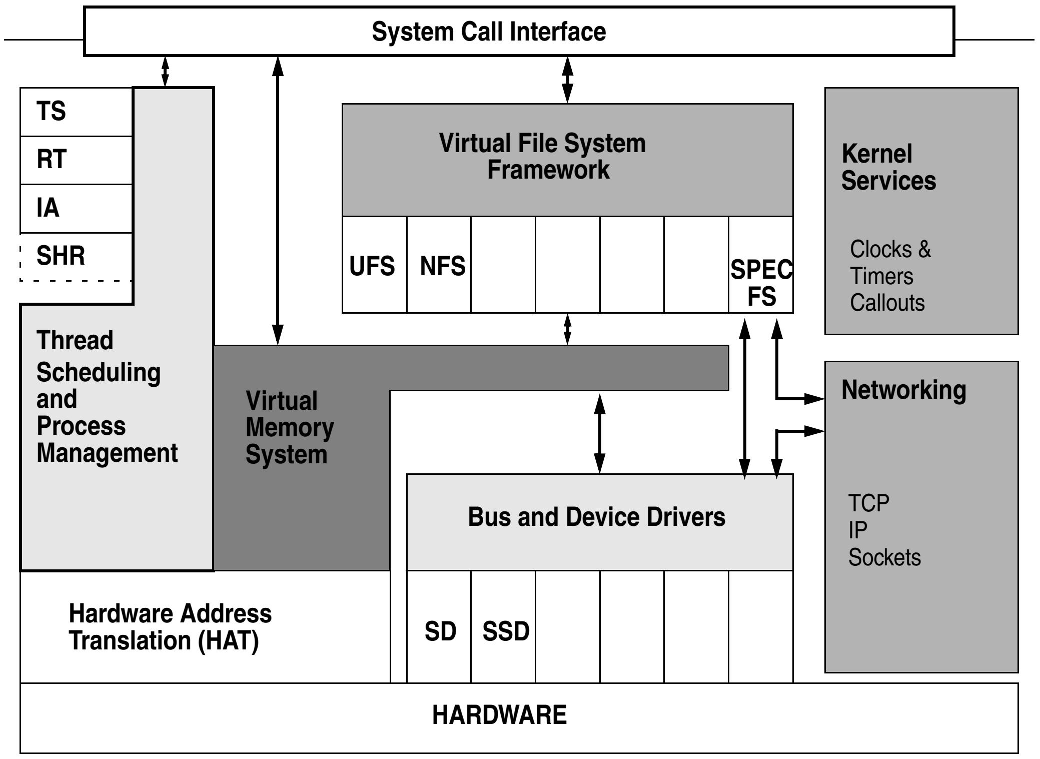


Figure 1.1 Solaris Kernel Components

1.3.2 Modular Implementation

The Solaris kernel is implemented as a core set of operating system functions, with additional kernel subsystems and services linked in as dynamically loadable modules. This implementation is facilitated by a module loading and kernel runtime linker infrastructure, which allows kernel modules to be added to the operating system either during boot or on demand while the system is running.

The Solaris 7 module framework supports seven types of loadable kernel modules: scheduler classes, file systems, loadable system calls, loaders for executable file formats, streams modules, bus or device drivers, and miscellaneous modules. Figure 1.2 shows the facilities contained in the core kernel and the various types of kernel modules that implement the remainder of the Solaris kernel.

Core Kernel	Module Types	Module Examples
System Calls Scheduler Memory Mgmt Proc Mgmt VFS Framework Kernel Locking Clock & Timers Interrupt Mgmt Boot & Startup Trap Mgmt CPU Mgmt	**Scheduler Classes**	TS –Time Share
		RT – Real Time
		IA – Interactive Class
		SRM – Resource Manager Class
	File Systems	UFS – UNIX File System
		NFS – Network File System
		PROCFS – Process File System
		Etc.…
	Loadable System Calls	shmsys – System V Shared Memory
		semsys – Semaphores
		msgsys – Messages
		Other loadable system calls …
	Executable Formats	ELF – SVR4 Binary Format
		COFF – BSD Binary Format
	Streams Modules	pipemod – Streams Pipes
		ldterm – Terminal Line Disciplines
		Other loadable streams modules …
	Misc Modules	NFSSRV – NFS Server
		IPC – Interprocess Communication
		Other loadable kernel code …
	Device and Bus Drivers	SBus – SBus Bus Controller
		PCI – PCI Bus Controller
		sd – SCSI I/O Devices
		Many other devices …

Figure 1.2 Core Kernel and Loadable Modules

1.4 Processes, Threads, and Scheduling

The Solaris kernel is multithreaded; that is, it is implemented with multiple threads of execution to allow concurrency across multiple processors. This architecture is a major departure from the traditional UNIX scheduling model. In Solaris, threads in the kernel, or *kernel threads*, are the fundamental unit that is scheduled and dispatched onto processors. Threads allow multiple streams of execution within a single virtual memory environment; consequently, switching execution between threads is inexpensive because no virtual memory context switch is required.

Threads are used for kernel-related tasks, for process execution, and for interrupt handling. Within the kernel, multiple threads of execution share the kernel's environment. Processes also contain one or more threads, which share the virtual memory environment of the process.

A process is an abstraction that contains the environment for a user program. It consists of a virtual memory environment, resources for the program such as an open file list, and at least one thread of execution. The virtual memory environment, open file list, and other components of the process environment are shared by the threads within each process.

Within each process is a *lightweight process*, a virtual execution environment for each kernel thread within a process. The lightweight process allows each kernel thread within a process to make system calls independently of other kernel threads within the same process. Without a lightweight process, only one system call could be made at a time. Each time a system call is made by a thread, its registers are placed on a stack within the lightweight process. Upon return from a system call, the system call return codes are placed in the lightweight process. Figure 1.3 shows the relationship between kernel threads, processes, and lightweight processes.

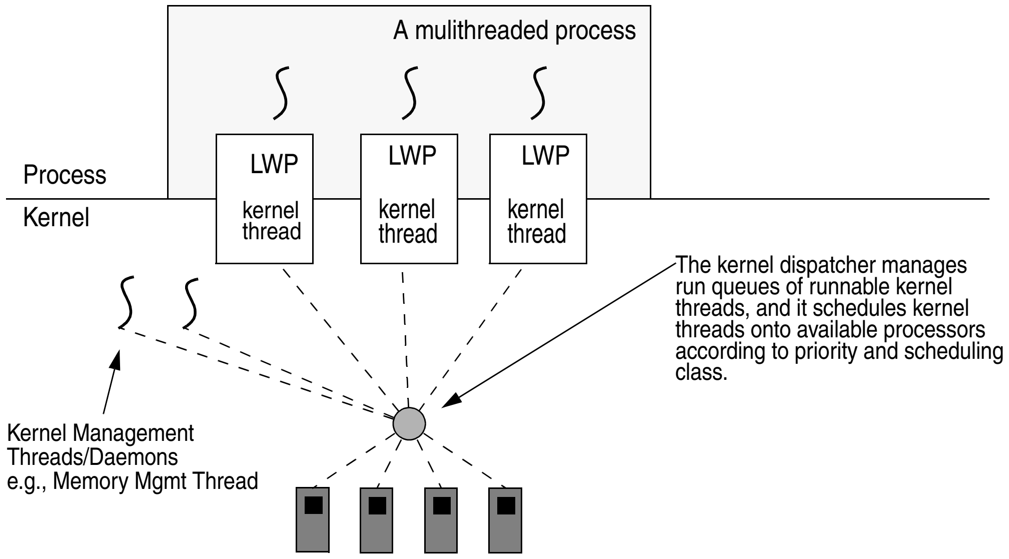

A mulithreaded process
LWP
kernel thread
LWP
kernel thread
LWP
kernel thread
Process
Kernel
The kernel dispatcher manages run queues of runnable kernel threads, and it schedules kernel threads onto available processors according to priority and scheduling class.
Kernel Management Threads/Daemons e.g., Memory Mgmt Thread

Figure 1.3 Kernel Threads, Processes, and Lightweight Processes

1.4.1 Two-Level Thread Model

Although it is relatively inexpensive to switch between multiple threads within a process, it is still relatively expensive to create and destroy threads. In addition, each kernel thread within a process requires a lightweight process containing a stack that consumes kernel resources. For these reasons, an additional level of thread management is implemented within each process to manage *user threads*, as shown in Figure 1.4.

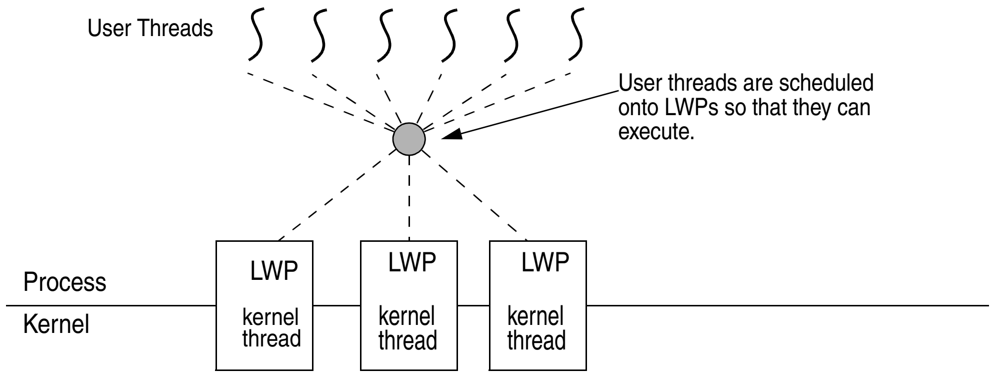


Figure 1.4 Two-Level Thread Model

Solaris exposes user threads as the primary thread abstraction for multithreaded programs. User threads are implemented in a thread library and can be created and destroyed without kernel involvement. User threads are scheduled on and off the lightweight processes. As a result, only a subset of the user threads is active at any one time—those threads that are scheduled onto the lightweight processes. The number of lightweight processes within the process affects the degree of parallelism available to the user threads and is adjusted on-the-fly by the user thread library

1.4.2 Global Process Priorities and Scheduling

The Solaris kernel implements a global thread priority model for kernel threads. The kernel scheduler, or *dispatcher*, uses the model to select which kernel thread of potentially many runnable kernel threads executes next. The kernel supports the notion of *preemption*, allowing a better-priority thread to cause the preemption of a running thread, such that the better- (higher) priority thread can execute. The kernel itself is preemptable, an innovation providing for time-critical scheduling of high-priority threads. There are 170 global priorities; numerically larger priority values correspond to better thread priorities. The priority name space is partitioned by different *scheduling classes*, as illustrated in Figure 1.5.

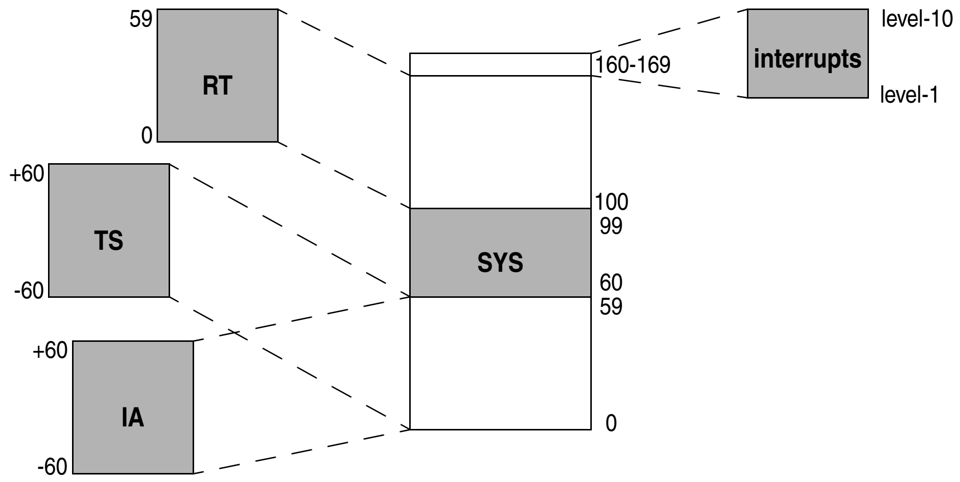


Figure 1.5 Global Thread Priorities

The Solaris dispatcher implements multiple scheduling classes, which allow different scheduling policies to be applied to threads. The three primary scheduling classes—TS (IA is an enhanced TS), SYS, and RT—shown in Figure 1.5 are described below.

- **TS** — The timeshare scheduling class is the default class for processes and all the kernel threads within the process. It changes process priorities dynamically according to recent processor usage in an attempt to evenly allocate processor resources among the kernel threads in the system. Process priorities and time quantums are calculated according to a timeshare scheduling table at each clock tick, or during wakeup after sleeping for an I/O. The TS class uses priority ranges 0 to 59.
- **IA** — The interactive class is an enhanced TS class used by the desktop windowing system to boost priority of threads within the window under focus. IA shares the priority numeric range with the TS class.
- **SYS** — The system class is used by the kernel for kernel threads. Threads in the system class are bound threads; that is, there is no time quantum—they run until they block. The system class uses priorities 60 to 99.
- **RT** — The realtime class implements fixed priority, fixed time quantum scheduling. The realtime class uses priorities 100 to 159. Note that threads in the RT class have a higher priority over kernel threads in the SYS class.

The interrupt priority levels shown in Figure 1.5 are not available for use by anything other than interrupt threads. The intent of their positioning in the priority scheme is to guarantee that interrupt threads have priority over all other threads in the system.

1.5 Interprocess Communication

Processes can communicate with each other by using one of several types of interprocess communication (IPC). IPC allows information transfer or synchronization to occur between processes. Solaris supports four different groups of interprocess communication: basic IPC, System V IPC, POSIX IPC, and advanced Solaris IPC.

1.5.1 Traditional UNIX IPC

Solaris implements traditional IPC facilities such as local sockets and pipes. A local socket is a network-like connection using the `socket(2)` system call to directly connect two processes.

A pipe directly channels data flow from one process to another through an object that operates like a file. Data is inserted at one end of the pipe and travels to the receiving processes in a first-in, first-out order. Data is read and written on a pipe with the standard file I/O system calls. Pipes are created with the `pipe(2)` system call or by a special pipe device created in the file system with `mknod(1)` and the standard file `open(2)` system call.

1.5.2 System V IPC

Three types of IPC originally developed for System V UNIX have become standard across all UNIX implementations: shared memory, message passing, and semaphores. These facilities provide the common IPC mechanism used by the majority of applications today.

- **System V Shared Memory** — Processes can create a segment of shared memory. Changes within the area of shared memory are immediately available to other processes that attach to the same shared memory segment.
- **System V Message Queues —** A message queue is a list of messages with a head and a tail. Messages are placed on the tail of the queue and are received on the head. Each messages contains a 32-bit type value, followed by a data payload.
- **System V Semaphores —** Semaphores are integer-valued objects that support two atomic operations: increment or decrement the value of the integer. Processes can sleep on semaphores that are greater than zero, then can be awakened when the value reaches zero.

1.5.3 POSIX IPC

The POSIX IPC facilities are similar in functionality to System V IPC but are abstracted on top of memory mapped files. The POSIX library routines are called by a program to create a new semaphore, shared memory segment, or message queue using the Solaris file I/O system calls (`open`(2), `read`(2), `mmap`(2), etc.). Internally in the POSIX library, the IPC objects exist as files. The object type exported to the program through the POSIX interfaces is handled within the library routines.

1.5.4 Advanced Solaris IPC

A new, fast, lightweight mechanism for calling procedures between processes is available in Solaris: doors. Doors are a low-latency method of invoking a procedure in local process. A door server contains a thread that sleeps, waiting for an invocation from the door client. A client makes a call to the server through the door, along with a small (16 Kbyte) payload. When the call is made from a door client to a door server, scheduling control is passed directly to the thread in the door server. Once a door server has finished handling the request, it passes control and response back to the calling thread. The scheduling control allows ultra-low-latency turnaround because the client does not need to wait for the server thread to be scheduled to complete the request.

1.6 Signals

UNIX systems have provided a process signaling mechanism from the earliest implementations. The signal facility provides a means to interrupt a process or thread within a process as a result of a specific event. The events that trigger signals can be directly related to the current instruction stream. Such signals, referred to as synchronous signals, originate as hardware trap conditions arising from illegal address references (segmentation violation), illegal math operations (floating point exceptions), and the like.

The system also implements asynchronous signals, which result from an external event not necessarily related to the current instruction stream. Examples of asynchronous signals include job control signals and the sending of a signal from one process or thread to another, for example, sending a kill signal to terminate a process.

For each possible signal, a process can establish one of three possible signal dispositions, which define what action, if any, will be taken when the signal is received. Most signals can be *ignored*, a signal can be *caught* and a process-specific signal handler invoked, or a process can permit the *default* action to be taken. Every signal has a predefined default action, for example, terminate the process. Solaris provides a set of programming interfaces that allow signals to be masked or a specific signal handler to be installed.

The traditional signal model was built on the concept of a process having a single execution stream at any time. The Solaris kernel's multithreaded process architecture allows for multiple threads of execution within a process, meaning that a signal can be directed to specific thread. The disposition and handlers for signals are process-wide; every thread in a multithreaded process has the same signal disposition and handlers. However, the Solaris model allows for signals to be masked at the thread level, so different threads within the process can have different signals masked. (Masking is a means of blocking a signal from being delivered.)

1.7 Memory Management

The Solaris virtual memory (VM) system can be considered to be the core of the operating system—it manages the system's memory on behalf of the kernel and processes. The main task of the VM system is to manage efficient allocation of the system's physical memory to the processes and kernel subsystems running within the operating system. The VM system uses slower storage media (usually disk) to store data that does not fit within the physical memory of the system, thus accommodating programs larger than the size of physical memory. The VM system is what keeps the most frequently used portions within physical memory and the lesser-used portions on the slower secondary storage.

For processes, the VM system presents a simple linear range of memory, known as an *address space*. Each address space is broken into several *segments* that represent mappings of the executable, heap space (general-purpose, process-allocated memory), shared libraries, and a program stack. Each segment is divided into equal-sized pieces of virtual memory, known as *pages*, and a hardware memory management unit (MMU) manages the mapping of page-sized pieces of virtual memory to physical memory. Figure 1.6 shows the relationship between an address space, segments, the memory management unit, and physical memory.

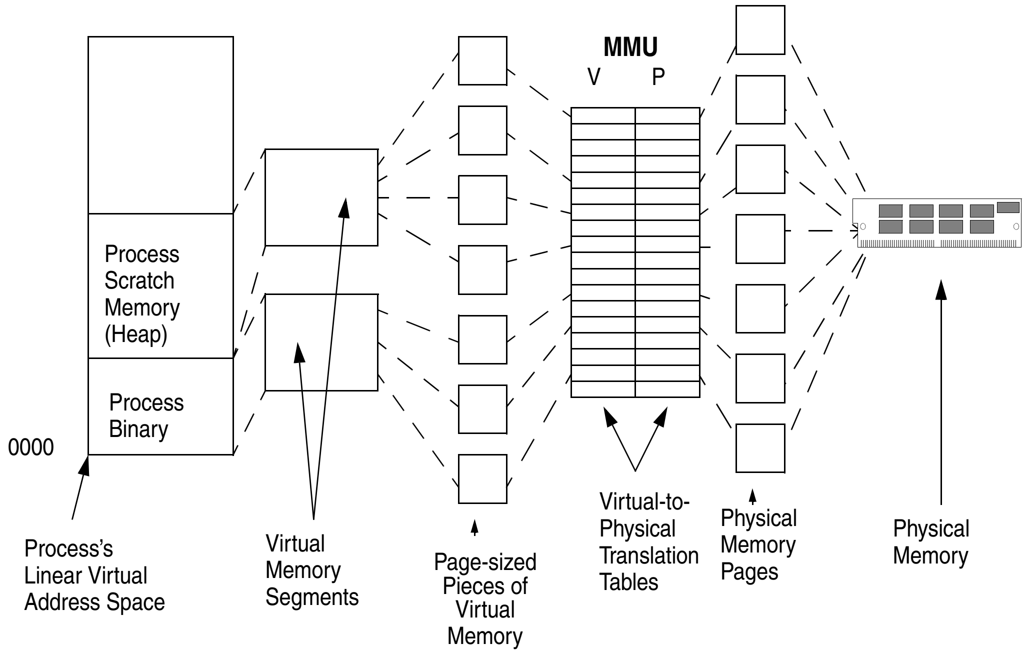

MMU
V P
Process Scratch Memory (Heap)
Process Binary
0000
Process's Linear Virtual Address Space
Virtual Memory Segments
Page-sized Pieces of Virtual Memory
Virtual-to-Physical Translation Tables
Physical Memory Pages
Physical Memory

Figure 1.6 Address Spaces, Segments, and Pages

The virtual memory system is implemented in a modular fashion. The components that deal with physical memory management are mostly hardware platform specific. The platform-dependent portions are implemented in the hardware address translation (HAT) layer.

1.7.1 Global Memory Allocation

The VM system implements demand paging. Pages of memory are allocated on demand, as they are referenced, and hence portions of an executable or shared library are allocated on demand. Loading pages of memory on demand dramatically lowers the memory footprint and startup time of a process. When an area of virtual memory is accessed, the hardware MMU raises an event to tell the kernel that an access has occurred to an area of memory that does not have physical memory mapped to it. This event is a *page fault*. The heap of a process is also allocated in a similar way: initially, only virtual memory space is allocated to the process. When memory is first referenced, a page fault occurs and memory is allocated one page at a time.

The virtual memory system uses a global paging model that implements a single global policy to manage the allocation of memory between processes. A scanning algorithm calculates the least-used portion of the physical memory. A kernel thread (the page scanner) scans memory in physical page order when the amount of free memory falls below a preconfigured threshold. Pages that have not been used recently are stolen and placed onto a free list for use by other processes.

1.7.2 Kernel Memory Management

The Solaris kernel requires memory for kernel instructions, data structures, and caches. Most of the kernel's memory is not pageable; that is, it is allocated from physical memory which cannot be stolen by the page scanner. This characteristic avoids deadlocks that could occur within the kernel if a kernel memory management function caused a page fault while holding a lock for another critical resource. The kernel cannot rely on the global paging used by processes, so it implements its own memory allocation systems.

A core kernel memory allocator—the *slab allocator*—allocates memory for kernel data structures. As the name suggests, the allocator subdivides large contiguous areas of memory (slabs) into smaller chunks for data structures. Allocation pools are organized so that like-sized objects are allocated from the same continuous segments, thereby dramatically reducing fragmentation that could result from continuous allocation and deallocation.

1.8 Files and File Systems

Solaris provides facilities for storage and management of data, as illustrated in Figure 1.7. A *file* provides a container for data, a *directory* contains a number of files, and a *file system* implements files and directories upon a device, typically a storage medium of some type.

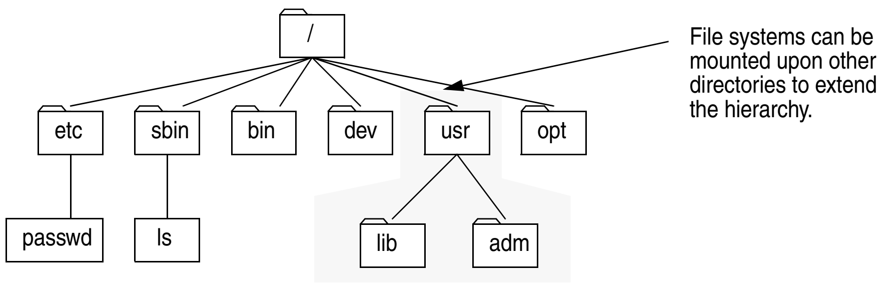

/
etc
sbin
bin
dev
usr
opt
passwd
ls
lib
adm
File systems can be mounted upon other directories to extend the hierarchy.

Figure 1.7 Files Organized in a Hierarchy of Directories

A file system can be mounted on a branch of an existing file system to extend the hierarchy. The hierarchy hides the mount so that it is transparent to users or applications that traverse the tree.

Solaris implements several different types of files:

- **Regular files** store data within the file system.
- **Special files** represent a device driver. Reads and writes to special files are handled by a device driver and translated into I/O of some type.
- **Pipes** are a special type of file that do not hold data but can be opened by two different processes so that data can be passed between them.
- **Hard links** link to the data of other files within the same file system. With hard links, the same data can have two different file names in the file system.
- **Symbolic links** point to other path names on any file system.
- **Sockets** in the file system enable local communication between two processes.

1.8.1 File Descriptors and File System Calls

Processes interface with files through file-related system calls. The file-related system calls identify files by two means: their path name in the file system and a *file descriptor*. A file descriptor is an integer number identifying an open file within a process. Each process has a table of open files, starting at file descriptor 0 and progressing upward as more files are opened. A file descriptor can be obtained with the `open()` system call, which opens a file named by a path name and returns a file descriptor identifying the open file.

```
fd = open("/etc/passwd",flag, mode);
```

Once a file has been opened, a file descriptor can be used for operations on the file. The `read`(2) and `write`(2) operations provide basic file I/O, along with several other advanced mechanisms for performing more complex operations. A file descriptor is eventually closed by the `close`(2) system call or by the process's exit. By default, file descriptors 0, 1, and 2 are opened automatically by the C runtime library and represent the standard input, standard output, and standard error streams for a process.

1.8.2 The Virtual File System Framework

Solaris provides a framework under which multiple file system types are implemented: the *virtual file system framework*. Earlier implementations of UNIX used a single file system type for all of the mounted file systems; typically, the UFS file system from BSD UNIX. The virtual file system framework, developed to enable the network file system (NFS) to coexist with the UFS file system in SunOS 2.0, became a standard part of System V in SVR4 and Solaris.

Each file system provides file abstractions in the standard hierarchical manner, providing standard file access interfaces even if the underlying file system implementation varies. The file system framework allows almost any objects to be abstracted as files and file systems. Some file systems store file data on storage-based media, whereas other implementations abstract objects other than storage as files. For example, the procfs file system abstracts the process tree, where each file in the file system represents a process in the process tree. We can categorize Solaris file systems into the following groups:

- **Storage Based** — Regular file systems that provide facilities for persistent storage and management of data. The Solaris UFS and PC/DOS file systems are examples.
- **Network File Systems** — File systems that provide files which appear to be in a local directory structure but are stored on a remote network server; for example, Sun's network file system (NFS).
- **Pseudo File Systems** — File systems that present various abstractions as files in a file system. The `/proc` pseudo file system represents the address space of a process as a series of files.

The framework provides a single set of well-defined interfaces that are file system independent; the implementation details of each file system are hidden behind these interfaces. Two key objects represent these interfaces: the virtual file, or *vnode*, and the virtual file system, or *vfs* objects. The `vnode` interfaces implement file-related functions, and the vfs interfaces implement file system management functions. The `vnode` and vfs interfaces call appropriate file system functions depending on the type of file system being operated on. Figure 1.8 shows the file system layers. File-related functions are initiated through a system call or from another kernel subsystem and are directed to the appropriate file system via the `vnode/vfs` layer.

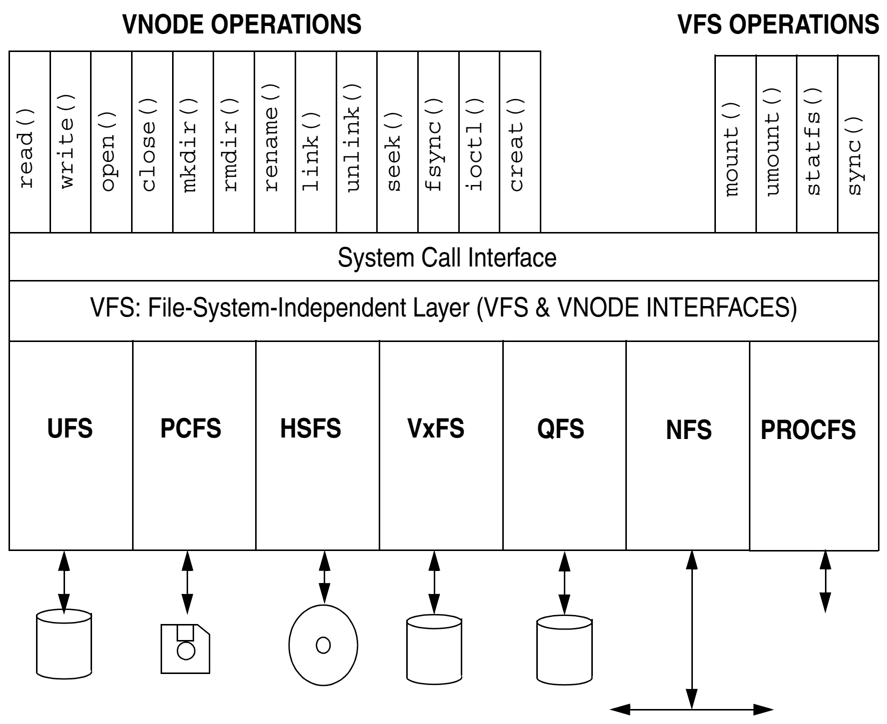


Figure 1.8 VFS/Vnode Architecture

Table 1-2 summarizes the major file system types that are implemented in Solaris.

Table 1-2 File Systems Available in Solaris File System Framework

File System	Type	Device	Description
`ufs`	Regular	Disk	UNIX Fast File system, default in Solaris
`pcfs`	Regular	Disk	MS-DOS file system
`hsfs`	Regular	Disk	High Sierra file system (CD-ROM)
`tmpfs`	Regular	Memory	Uses memory and swap
`nfs`	Pseudo	Network	Network file system
`cachefs`	Pseudo	File system	Uses a local disk as cache for another NFS file system
`autofs`	Pseudo	File system	Uses a dynamic layout to mount other file systems
`specfs`	Pseudo	Device Drivers	File system for the `/dev` devices
`procfs`	Pseudo	Kernel	`/proc` file system representing processes

Table 1-2 File Systems Available in Solaris File System Framework (Continued)

File System	Type	Device	Description
`sockfs`	Pseudo	Network	File system of socket connections
`fdfs`	Pseudo	File Descriptors	Allows a process to see its open files in `/dev/fd`
`fifofs`	Pseudo	Files	FIFO file system

1.9 I/O Architecture

Traditional UNIX implements kernel-resident device drivers to interface with hardware devices. The device driver manages data transfer and registers I/O and handles device hardware interrupts. A device driver typically has to know intimate details about the hardware device and the layout of buses to which the device is connected. Solaris extends traditional device driver management functions by using separate drivers for devices and buses: a *device driver* controls a device's hardware, and a *bus nexus driver* controls and translates data between two different types of buses.

Solaris organizes I/O devices in a hierarchy of bus nexus and instances of devices, according to the physical connection hierarchy of the devices. The hierarchy shown in Figure 1.9 represents a typical Solaris *device tree*.

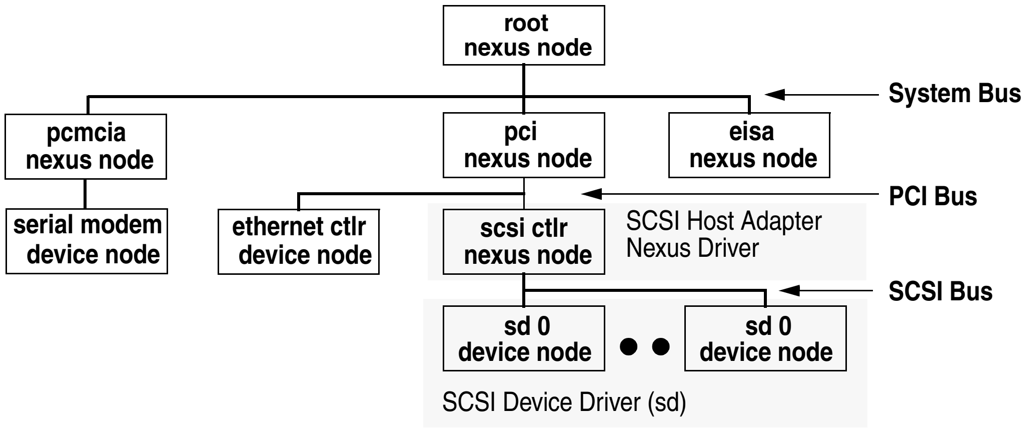


Figure 1.9 The Solaris Device Tree

Each bus connects to another bus through a bus nexus. In our example, nexus drivers are represented by the PCI, EISA, PCMCIA, and SCSI nodes. The SCSI host adapter is a bus nexus bridging the PCI and SCSI bus it controls, underneath which the SCSI disk (`sd`) device driver implements device nodes for each disk on the SCSI chain.

The Solaris device driver interface (DDI) hides the implementation specifics of the platform and bus hierarchy from the device drivers. The DDI provides interfaces for registering interrupts, mapping registers, and accessing DMA memory. In that way, the kernel can interface with the device.

Device drivers are implemented as loadable modules, that is, as separate binaries containing driver code. Device drivers are loaded automatically the first time their device is accessed.