Linux Kernel Introduction

Wang Xiaolin
wx672ster@gmail.com

June 10, 2013

Contents

References

[BC05] D.P. Bovet and M. Cesatí. *Understanding The Linux Kernel*. 3rd ed. O'Reilly, 2005.

[Lov10] R. Love. *Linux Kernel Development*. Developer's Library. Addison-Wesley, 2010.

[SGG11] Silberschatz, Galvin, and Gagne. *Operating System Concepts Essentials*. John Wiley & Sons, 2011.

1 Basic Operating System Concepts

Basic Operating System Concepts

Two main objectives of an OS:

- Interact with the hardware components
- Provide an execution environment to the applications

Why?

Unix hides all low-level details from applications

User mode vs. Kernel mode

MS-DOS allows user programs to directly play with the hardware components

UNIX The original elegant design of the Unix system, along with the years of innovation and evolutionary improvement that followed, have made Unix a powerful, robust, and stable operating system. A handful of characteristics of Unix are responsible for its resilience.[Lov10]

- First, Unix is simple: Whereas some operating systems implement thousands of system calls and have unclear design goals, Unix systems typically implement only hundreds of system calls and have a very clear design.
- Next, in Unix, *everything is a file*. This simplifies the manipulation of data and devices into a set of simple system calls: `open()`, `read()`, `write()`, `ioctl()`, and `close()`.
- In addition, the Unix kernel and related system utilities are written in C — a property that gives Unix its amazing portability and accessibility to a wide range of developers.
- Next, Unix has fast process creation time and the unique `fork()` system call. This encourages strongly partitioned systems without gargantuan multi-threaded monstrosities.
- Finally, Unix provides simple yet robust interprocess communication (IPC) primitives that, when coupled with the fast process creation time, allow for the creation of simple utilities that do one thing and do it well, and that can be strung together to accomplish more complicated tasks.

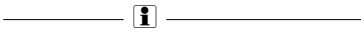
Typical Components of a Kernel

Interrupt handlers: to service interrupt requests

Scheduler: to share processor time among multiple processes

Memory management system: to manage process address spaces

System services: Networking, IPC...



Kernel And Processes

Kernel space

- a protected memory space
- full access to the hardware

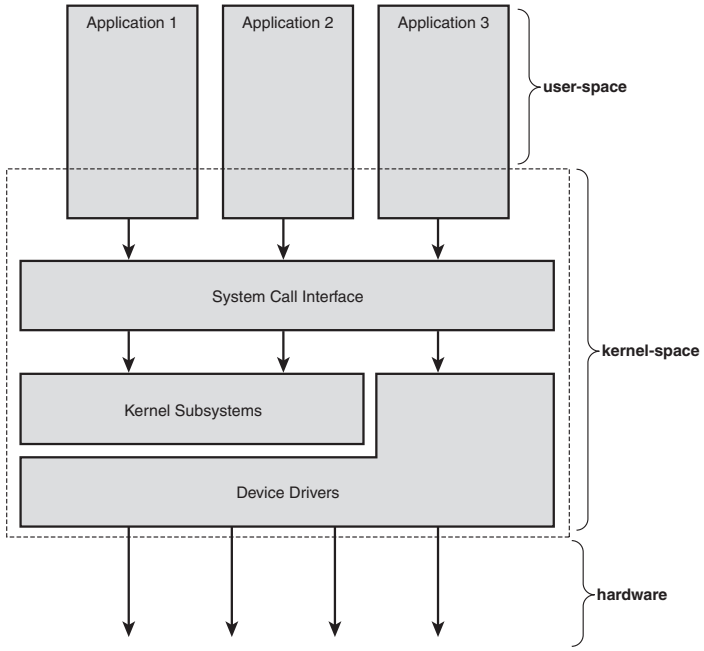
When executing the kernel, the system is in kernel-space executing in *kernel mode*.

User Space

- Can only see a subset of available resources
- unable to perform certain system functions, nor directly access hardware

Normal user execution in user-space executing in *user mode*.

$$\text{user mode} \xrightarrow{system\ calls} \text{kernel mode}$$



C library and system calls An application typically calls functions in a library, for example, the *C library*, that in turn rely on the system call interface to instruct the kernel to carry out tasks on their behalf. Some library calls provide many features not found in the system call, and thus, calling into the kernel is just one step in an otherwise large function. For example, consider the familiar `printf()` function. It provides formatting and buffering of the data and only eventually calls `write()` system call to write the data to the console. Conversely, some library calls have a one-to-one relationship with the kernel. For example, the `open()` library function does nothing except call the `open()` system call. Still other C library functions, such as `strcpy()`, should

(you hope) make no use of the kernel at all. When an application executes a system call, it is said that the *kernel is executing on behalf of the application*. Furthermore, the application is said to be *executing a system call in kernel-space*, and the kernel is running in *process context*. This relationship that applications *call into* the kernel via the system call interfaceis the fundamental manner in which applications get work done.[Lov10]

Kernel And Hardware

Interrupts
Whenever hardware wants to communicate with the system, it issues an interrupt that asynchronously interrupts the kernel.

Interrupt vector

Interrupt handlers

Interrupts The kernel also manages the system's hardware. Nearly all architectures, including all systems that Linux supports, provide the concept of *interrupts*. When hardware wants to communicate with the system, it issues an interrupt that asynchronously interrupts the kernel. Interrupts are identified by a number. The kernel uses the number to execute a specific *interrupt handler* to process and respond to the interrupt. For example, as you type, the keyboard controller issues an interrupt to let the system know that there is new data in the keyboard buffer. The kernel notes the interrupt number being issued and executes the correct interrupt handler. The interrupt handler processes the keyboard data and lets the keyboard controller know it is ready for more data. To provide synchronization, the kernel can usually disable interrupts, either all interrupts or just one specific interrupt number. In many operating systems, including Linux, the interrupt handlers do not run in a process context. Instead, they run in a special *interrupt context* that is not associated with any process. This special context exists solely to let an interrupt handler quickly respond to an interrupt, and then exit.[Lov10]

These contexts represent the breadth of the kernel's activities. In fact, in Linux, we can generalize that each processor is doing one of three things at any given moment:

- In kernel-space, in process context, executing on behalf of a specific process
- In kernel-space, in interrupt context, not associated with a process, handling an interrupt
- In user-space, executing user code in a process

This list is inclusive. Even corner cases fit into one of these three activities: For example, when idle, it turns out that the kernel is executing an *idle process* in process context in the kernel.

Kernel Architecture

Monolithic kernels
Simplicity and performance

- exist on disk as single static binaries

- All kernel services run in the kernel address space
- Communication within the kernel is trivial

Most Unix systems are monolithic in design.

———— ℹ ————

Microkernels

- are not implemented as single large processes
- break the kernel into separate processes (*servers*).
  - in the microkernel
    * a few synchronization primitives
    * a simple scheduler
    * an IPC mechanism
  - top of the microkernel
    * memory allocators
    * device drivers
    * system call handlers

———— ℹ ————

Advantages of microkernel OS

- modularized design
- easily ported to other architectures
- make better use of RAM

Performance Overhead

- Communication via *message passing*
- Context switch (kernel-space ⇔ user-space)
  - Windows NT and Mac OS X keep all servers in kernel-space. (defeating the primary purpose of microkernel designs)
- Microkernel OSes are generally slower than monolithic ones.
- Academic research on OS is oriented toward microkernels.

———— ℹ ————

Monolithic Kernel Versus Microkernel Designs Operating kernels can be divided into two main design camps: the monolithic kernel and the microkernel. (A third camp, exokernel, is found primarily in research systems but is gaining ground in real-world use.)

Monolithic kernels involve the simpler design of the two, and all kernels were designed in this manner until the 1980s. Monolithic kernels are implemented entirely as single large processes running entirely in a single address space. Consequently, such kernels typically exist on disk as single static binaries. All kernel services exist and execute in the large kernel address space. Communication within the kernel is

trivial because everything runs in kernel mode in the same address space: The kernel can invoke functions directly, as a user-space application might. Proponents of this model cite the simplicity and performance of the monolithic approach. Most Unix systems are monolithic in design.

Microkernels, on the other hand, are not implemented as single large processes. Instead, the functionality of the kernel is broken down into separate processes, usually called servers. Idealistically, only the servers *absolutely* requiring such capabilities run in a privileged execution mode. The rest of the servers run in user-space. All the servers, though, are kept separate and run in different address spaces. Therefore, direct function invocation as in monolithic kernels is not possible. Instead, communication in microkernels is handled via *message passing*: An interprocess communication (IPC) mechanism is built into the system, and the various servers communicate and invoke ”services” from each other by sending messages over the IPC mechanism. The separation of the various servers prevents a failure in one server from bringing down another.

Likewise, the modularity of the system allows one server to be swapped out for another. Because the IPC mechanism involves quite a bit more overhead than a trivial function call, however, and because a context switch from kernel-space to user-space or vice versa may be involved, message passing includes a latency and throughput hit not seen on monolithic kernels with simple function invocation. Consequently, all practical microkernel-based systems now place most or all the servers in kernel-space, to remove the overhead of frequent context switches and potentially allow for direct function invocation. The Windows NT kernel and Mach (on which part of Mac OS X is based) are examples of microkernels. Neither Windows NT nor Mac OS X run any microkernel servers in user-space in their latest versions, defeating the primary purpose of microkernel designs altogether.

Linux is a monolithic kernel, that is, the Linux kernel executes in a single address space entirely in kernel mode. Linux, however, borrows much of the good from microkernels: Linux boasts a modular design with kernel preemption, support for kernel threads, and the capability to dynamically load separate binaries (kernel modules) into the kernel. Conversely, Linux has none of the performance-sapping features that curse microkernel designs: Everything runs in kernel mode, with direct function invocation, not message passing, the method of communication. Yet Linux is modular, threaded, and the kernel itself is schedulable. Pragmatism wins again.[Lov10]

2 Linux Versus Other Unix-Like Kernels

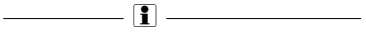
Linux is a monolithic kernel with modular design

Modularized approach — makes it easy to develop new modules

Platform independence — if standards compliant

Frugal main memory usage — run time (un)loadable

No performance penalty — no explicit message passing is required



Linux

A newcomer in the family of Unix-like OSes

AT&T Unix SVR4	UCB 4.4BSD	DEC Digital Unix
IBM AIX	HP HP-UX	Sun Solaris
Apple Mac OS X	FreeBSD	NetBSD
OpenBSD	Linux	

- Linus Torvalds, 1991
- A true Unix kernel
- available on many architectures
  - `ls /usr/src/linux/arch/`
- GPL, non-commercial

Linux Versus Other Unix-Like Kernels

- share fundamental design ideas and features
- from 2.6, Linux kernels are POSIX-compliant
  - Unix programs can be compiled and executed on Linux
- Linux includes all the features of a modern Unix
  - VM, VFS, LWP, SVR4 IPC, signals, SMP support ...

Linux Kernel Features

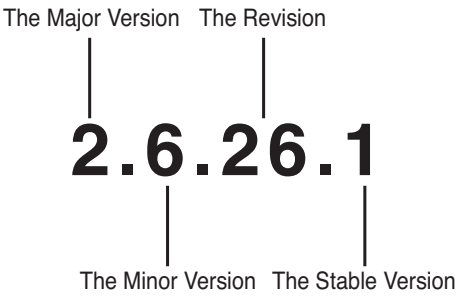
- Monolithic kernel
- loadable modules support
- Kernel threading
- Multithreaded application support
- Preemptive kernel
- Multiprocessor support
- File systems

Advantages Over Its Commercial Competitors

- cost-free
- fully customizable in all its components

- runs on low-end, inexpensive hardware platforms
- performance
- developers are excellent programmers
- kernel can be very small and compact
- highly compatible with many common operating systems
  - filesystems, network interfaces, wine ...
- well supported

Linux Versions



3 An Overview of Unix Kernels

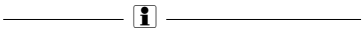
3.1 The Process/Kernel Model

The Unix Process/Kernel Model

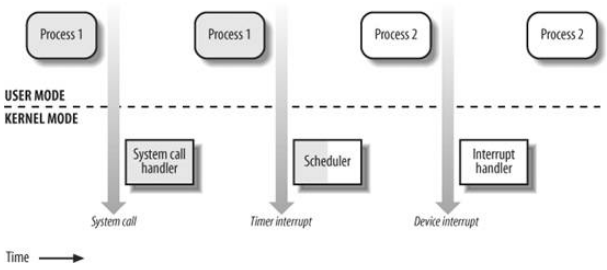
User Mode vs. Kernel Mode

- Processes can run in either user mode or kernel mode
- The kernel itself is not a process but a process manager

$$\text{processes} \xrightarrow{system\ calls} \text{process manager}$$



Kernel routines can be activated in several ways



Unix Kernel Threads

- run in Kernel Mode in the kernel address space
- no interact with users
- created during system startup and remain alive until the system is shut down

3.2 Process Implementation

Process Implementation

- Each process is represented by a *process descriptor (PCB)*
- Upon a process switch, the kernel
  - saves the current contents of several registers in the PCB
  - uses the proper PCB fields to load the CPU registers

Registers

- The program counter (PC) and stack pointer (SP) registers
- The general purpose registers
- The floating point registers
- The processor control registers (Processor Status Word) containing information about the CPU state
- The memory management registers used to keep track of the RAM accessed by the process

3.3 Reentrant Kernels

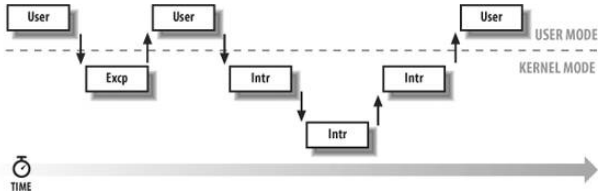
Reentrant Kernels

Reentrant Kernels several processes may be executing in Kernel Mode at the same time

i.e. several processes can wait in kernel mode

Kernel control path denotes the sequence of instructions executed by the kernel to handle a system call, an exception, or an interrupt.

Interleaving of kernel control paths



Re-entrant functions It's easier to remember when you understand what the term means.

The term "re-entrant" means that it is safe to "re-enter" the function while it is already executed, typically in a concurrent environment.

In other words, when two tasks can execute the function at the same time without interfering with each other, then the function is re-entrant. A function is not re-entrant when the execution by one task has an impact on the influence of another task. This typically is the case when a global state or data is used. A function that uses only local variables and arguments is typically re-entrant. [Stackoverflow]

About kernel control path (Sec 1.6.3 in [BC05]) In the simplest case, the CPU executes a kernel control path sequentially from the first instruction to the last. When one of the following events occurs, however, the CPU interleaves the kernel control paths :

case 1 A process executing in User Mode invokes a system call, and the corresponding kernel control path verifies that the request cannot be satisfied immediately; it then invokes the scheduler to select a new process to run. As a result, a process switch occurs. The first kernel control path is left unfinished, and the CPU resumes the execution of some other kernel control path. In this case, the two control paths are executed on behalf of two different processes.

case 2 The CPU detects an exception, for example, access to a page not present in RAM, while running a kernel control path. The first control path is suspended, and the CPU starts the execution of a suitable procedure. In our example, this type of procedure can allocate a new page for the process and read its contents from disk. When the procedure terminates, the first control path can be resumed. In this case, the two control paths are executed on behalf of the same process.

case 3 A hardware interrupt occurs while the CPU is running a kernel control path with the interrupts enabled. The first kernel control path is left unfinished, and the CPU starts processing another kernel control path to handle the interrupt. The first kernel control path resumes when the interrupt handler terminates. In this case, the two kernel control paths run in the execution context of the same process, and the total system CPU time is accounted to it. However, the interrupt handler doesn't necessarily operate on behalf of the process.

case 4 An interrupt occurs while the CPU is running with kernel preemption enabled, and a higher priority process is runnable. In this case, the first kernel control path is left unfinished, and the CPU resumes executing another kernel control path on behalf of the higher priority process. This occurs only if the kernel has been compiled with kernel preemption support.

- stackoverflow: User-space process preempts kernel thread?

3.4 Process Address Space

Each process runs in its private address space

- User-mode private stack (user code, data...)
- Kernel-mode private stack (kernel code, data...)

Sharing cases

- The same program is opened by several users
- Shared memory IPC
- `mmap()`

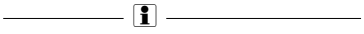
3.5 Synchronization and Critical Regions

Synchronization and Critical Regions

Re-entrant kernel requires synchronization
If a kernel control path is suspended while acting on a kernel data structure, no other kernel control path should be allowed to act on the same data structure unless it has been reset to a consistent state.

Race condition
When the outcome of a computation depends on how two or more processes are scheduled, the code is incorrect. We say that there is a *race condition*.

- Kernel preemption disabling
- Interrupt disabling
- Semaphores
- Spin locks
- Avoiding deadlocks



3.6 Signals and Interprocess Communication

Signals and IPC

Unix signals notifying processes of system events

`man 7 signal`

IPC semaphores , message queues , and shared memory

`man 5 ipc`
`shmget()`, `shmat()`, `shmdt()`
`semget()`, `semctl()`, `semop()`
`msgget()`, `msgsnd()`,`msgrcv()`

- Wikipedia: Unix signal
- Introduction to UNIX Signals and System Calls

3.7 Process Management

Process Management

`fork()` to create a new process

`wait()` to wait until one of its children terminates

`_exit()` to terminate a process

`exec()` to load a new program

`_exit()`: system call

`exit()`: library call

Zombie processes

Zombie a *process state* representing terminated processes

- a process remains in that state until its parent process executes a `wait()` system call on it

Orphaned processes become children of *init*.

`man 2 wait`: read the **NOTES** section

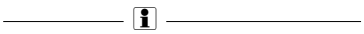
Process groups

```
~$ ls | sort | less
```

- `bash` creates a new group for these 3 processes
- each PCB includes a field containing the *process group ID*
- each group of processes may have a *group leader*
- a newly created process is initially inserted into the process group of its parent

login sessions

- All processes in a process group must be in the same login session
- A login session may have several process groups active simultaneously

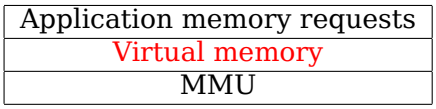


- `man 7 credentials`
- What are session leaders in PS?

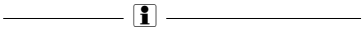
3.8 Memory Management

Memory Management

Virtual memory



- Several processes can be executed concurrently
- Virtual memory can be larger than physical memory
- Processes can run without fully loaded into physical memory
- Processes can share a single memory image of a library or program
- Easy relocation



RAM Usage

Physical memory

- A few megabytes for storing the kernel image
- The rest of RAM are handled by the virtual memory system
  - dynamic kernel data structures, e.g. buffers, descriptors ...
  - to serve process requests
  - caches of buffered devices

Problems faced:

- the available RAM is limited
- memory fragmentation
- ...

———— ℹ ————

Kernel Memory Allocation

User mode process memory

- Memory pages are allocated from the list of free page frames
- The list is populated using a page-replacement algorithm
- free frames scattered throughout physical memory

Kernel memory allocation

- Treated differently from user memory
  - allocated from a free-memory pool

Because:

- must be fast, i.e. avoid searching
- minimize waste, i.e. avoid fragmentation
- maximize contiguousness

Linux's KMA uses a Slab allocator on top of a buddy system.

———— ℹ ————

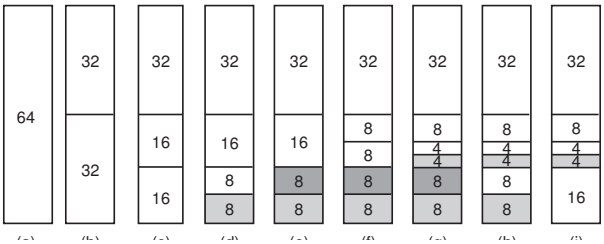
More info:

- sec 8.8 of [SGG11]

Buddy system

- By splitting memory into halves to try to give a best-fit

- Adjacent units of allocatable memory are paired together



Object creation and deletion

- are widely employed by the kernel
- more expensive than allocating memory to them

Slab allocation

- memory chunks suitable to fit data objects of certain type or size are preallocated
  - avoid searching for suitable memory space
  - greatly alleviates memory fragmentation
- Destruction of the object does not free up the memory, but only opens a slot which is put in the list of free slots by the slab allocator

Benefits

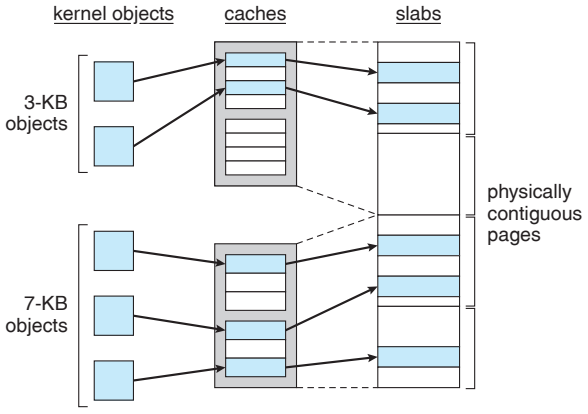
- No memory is wasted due to fragmentation
- Memory request can be satisfied quickly

Slab allocation

Slab is made up of several physically contiguous pages

Cache consists of one or more slabs.

- A storage for a specific type of object such as semaphores, process descriptors, file objects etc.



Process virtual address space handling

- demand paging
- copy on write

Caching

- hard drives are very slow
- to defer writing to disk as long as possible
- When a process asks to access a disk, the kernel checks first whether the required data are in the cache
- `sync()`

3.9 Device Drivers

Device Drivers
The kernel interacts with I/O devices by means of device drivers

The device files in `/dev`

- are the user-visible portion of the device driver interface
- each device file refers to a specific device driver