Linux Kernel Introduction

Wang Xiaolin

June 10, 2013

✉ wx672ster@gmail.com

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

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

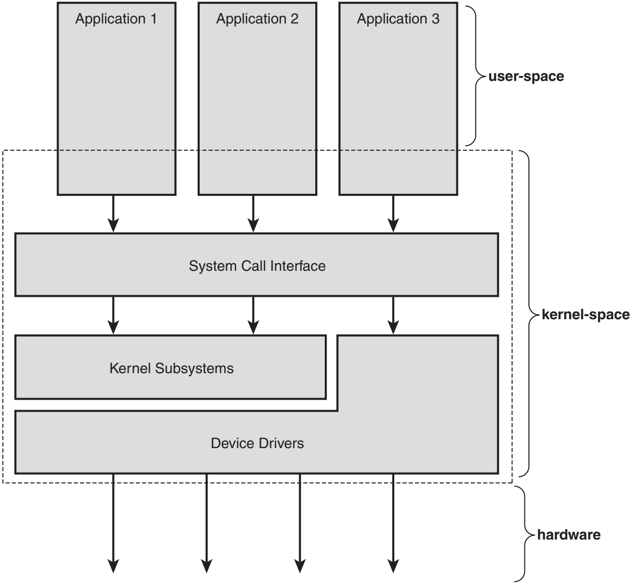
When executing the kernel, the system is in kernel-space executing in *kernel mode*.

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*.

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


Application 1
Application 2
Application 3
user-space
System Call Interface
kernel-space
Kernel Subsystems
Device Drivers
hardware

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

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.

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

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 $\Leftrightarrow$ 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.

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 ...

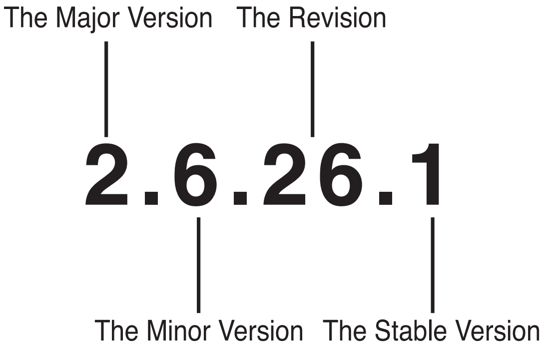
Linux Kernel Features

- 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



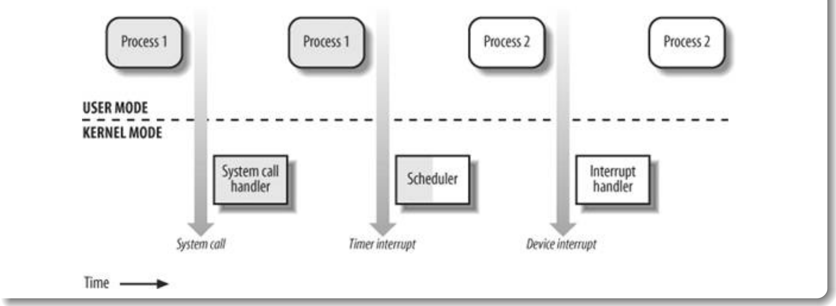
The Unix Process/Kernel Model

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

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

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

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

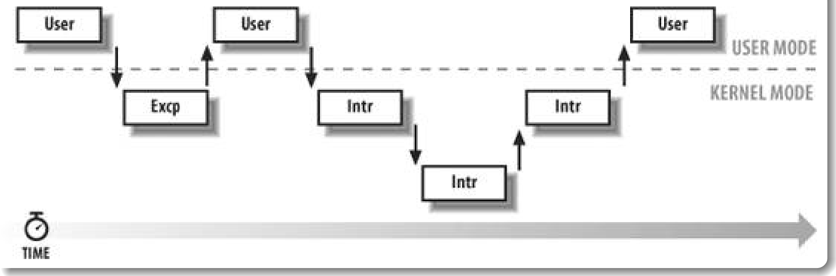
Reentrant Kernels

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



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()`

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

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()`

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

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*.

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

Memory Management

Virtual memory

Application memory requests
Virtual memory
MMU

- 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
- ...

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

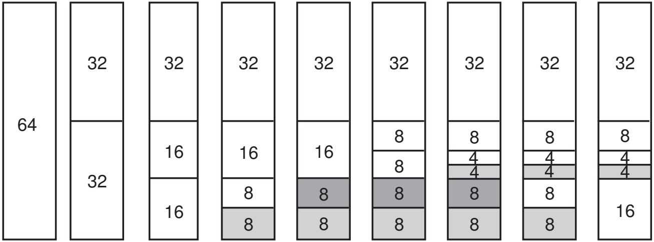
Because:

- 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.

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

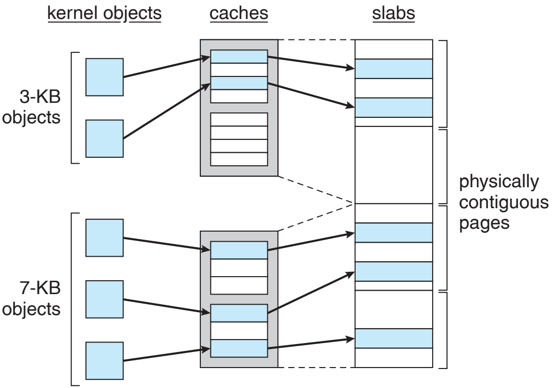
- No memory is wasted due to fragmentation
- Memory request can be satisfied quickly

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()`

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