

Advanced Operating Systems
高等作業系統
Fall 2013

Yuan-Hao Chang (張原豪)
johnson@iis.sinica.edu.tw
Institute of Information Science
Academia Sinica

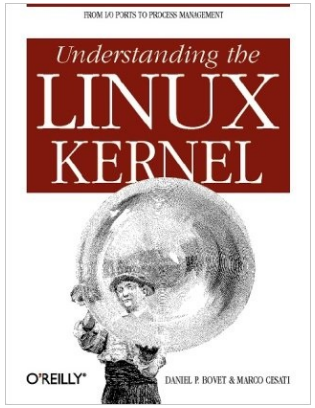


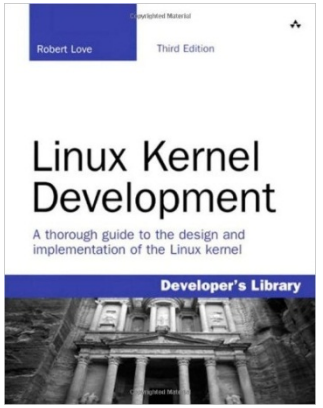
Course Information

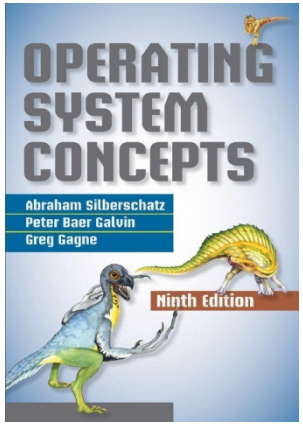
- **Lecturer:**
  - Yuan-Hao Chang
  - Office: R612, IIS, Academia Sinica
  - Phone: +886-2-2788-3799 ext. 1612
- **Teaching Assistant (TA):**
  - 柯奇恩
  - Email: bibi630[at]gmail.com
- **Lecturing hours:**
  - 9 am ~ 12:00 pm, Monday
- **Classroom:**
  - **S602,**
- **Course webpage:**
  - http://www.iis.sinica.edu.tw/~johnson/courses/AOS201308/
  - **Grading：(subject to changes)**
  - Projects: (30%), Midterm exam (30%), Final exam (30%), In-class performance (10%)
- **Prerequisite**
  - C programming language
  - Operating system

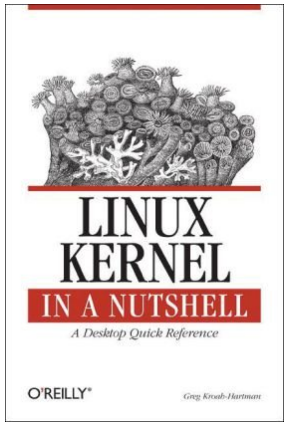
Course Information (Cont.)

- **Textbooks:**
  - Understanding the Linux Kernel 3/e, 2005
    Authors: Daniel Bovet and Marco Cesati
    Publisher: O'Reilly
    ISBN: 0596005652
  - Professional Linux Kernel Architecture , 2008
    Authors: Wolfgang Mauerer
    Publisher: Wrox Press
    ISBN: 0470343435
  - Linux Kernel Development 3/e, 2010
    Authors: Robert Love
    Publisher: Addison Wesley
    ISBN: 0672329468
  - Operating System Concepts 9/e, 2012
    Authors: Abraham Silberschat
    Publisher: Wiley
    ISBN: 1118063333
  - Linux Kernel in a Nutshell, 2006
    Authors: Greg Kroah-Hartman
    Publisher: O'Reilly Media, Inc.
    ISBN: 0596100795











Syllabus

1. Introduction
2. Memory Addressing
3. Processes
4. Interrupts and Exceptions
5. Kernel Synchronization
6. Timing Requirements
7. Architecture
8. Process Scheduling
9. Memory Management
10. Process Address Space
11. System Calls
12. Signals



01 Introduction

Unix-Like Operating Systems

- Linux is a member of the large family of Unix-like operating systems.
- Other Unix-like operating systems:
  - System V Release 4 (SVR4), from AT&T
  - 4.4BSD, from University of California at Berkeley
  - Digital Unix, from Digital (now Hewlett-Packard)
  - AIX, from IBM
  - HP-UX, from Hewlett-Packard
  - Solaris, from Sun
  - Mac OS X, from Apple
  - Others: FreeBSD, NetBSD, and OpenBSD

Linux

This course is based on Linux 2.6.11

- Developed by Linus Torvalds in 1991 for Intel 80386 microprocessor.
- Ported to various architectures, including Hewlett-Packard's Alpha, Intel's Itanium, AMD's AMD64, PowerPC, and IBM's zSeries, etc.
- Compliant to IEEE's Portable Operating Systems based on Unix (*POSIX*)
- Opened under GNU General Public License (GPL)
  - The GNU project is coordinated by the Free Software Foundation, Inc. (*http://www.gnu.org*); its aim is to implement a whole operating system freely usable by everyone.
- Linux kernel source: http://www.kernel.org
  - Some distributions put the kernel source in the /usr/src/linux directory



Linux Versus Other Unix-Link Kernels

Linux Features

- *Monolithic kernel*
  - Most commercial Unix variants are monolithic.
- *Dynamically linked Linux kernels*
  - Linux's support for modules is very good, because it is able to automatically load and unload modules on demand.
- *Kernel threading*
  - A kernel thread is an execution context that can be independently scheduled; it may be associated with a user program, or it may run only some kernel functions.
  - Context switches between kernel threads are usually much less expensive than context switches between ordinary processes.

Linux Features (Cont.)

- *Multithreaded application support*
  - A multithreaded user application could be composed of many lightweight processes (LWP), which are processes that can operate on a common address space, common physical memory pages, common opened files, and so on.
  - Linux regards lightweight processes as the basic execution context.
- *Preemptive kernel*
  - Linux can arbitrarily interleave execution flows while they are in privileged mode.
- *Multiprocessor support*
  - Linux supports symmetric multiprocessing (SMP) for different memory models, including NUMA.
- *Filesystem*
  - Due toe the design of virtual filesystems, Linux is easy to support most of the existing filesystem designs.

Advantages of Linux

- Linux is cost-free
- Linux is fully customizable in all its components
  - Due to the GPL
- Linux runs on low-end, inexpensive hardware platforms.
  - E.g., Running with 4MB RAM
- Linux is powerful
  - Fully exploit the features of the hardware components.
- Linux developers are excellent programmers
  - Linux systems are very stable.
- The Linux kernel can be very small and compact
  - It is possible to fit in a 1.44MB floppy disk.
- Linux is highly compatible with many common operating systems' applications
- Linux is well supported

Linux Versions

- UP to kernel version 2.5:
  - Each version was characterized by three numbers, separated by periods.
    - Version: the first two numbers
      - The second number identifies the type of kernel.
        - » Even number: a stable version
        - » Odd number: a development version
    - Release: the third number
- Kernel version 2.6 and up:
  - The second number no longer identifies stable or development versions.
  - Two kernels having the same version but different release numbers (e.g., 2.6.10 and 2.6.11) can differ significantly.
  - The fourth number identifies the patched version
    - E.g., 2.6.11.12 is the 12th patch to make 2.6.11 more stable.

Basic Operating System Concepts

- Kernel: The most important program of an operating system.
  - Loaded into RAM when the system boots and contains many critical procedures that are needed for the system to operate.
  - The kernel provides key facilities to everything else on the system and determines many of the characteristics of higher software.
  - We use the term “operating system” as a synonym for “kernel”.
- Main objectives of operating systems
  - 1. Interact with the hardware components:
    - Servicing all low-level programmable elements included in the hardware platform.
  - 2. Provide an execution environment to user programs (or applications) that run on the computer system.

MS-DOS vs. Linux

- MS-DOS:
  - Allow all user programs to directly play with the hardware components.
- Linux:
  - Hides all low-level details concerning the physical organization of the computer from applications run by the user.
    - When a program wants to use a hardware resource, it must issue a request to the operating system.
    - The kernel evaluates the request and interacts with the proper hardware components on behalf of the user program.

User Mode vs. Kernel Mode

- Modern operating systems rely on the availability of specific hardware features that forbid user programs to directly interact with low-level hardware components or to access arbitrary memory locations.
- Two running modes:
  - User mode
    - Nonprivileged mode for user programs.
  - Kernel mode
    - Privileged mode for the kernel.

Multiuser System

- A *multiuser system* is a computer that is able to concurrently and independently execute several applications belonging to two or more users.
  - Concurrently:
    - Applications can be active at the same time and contend for the various resources such as CPU, memory, hard disks.
  - Independently:
    - Each application can perform its task with no concern for what the applications of the other users are doing.
- Features of multiuser operating systems:
  - An authentication mechanism
  - A protection mechanism against buggy user programs
  - A protection mechanism against malicious user programs
    - Might interfere with or spy on the activity of other users.
  - An accounting mechanism

Users and Groups

- Users and groups:
  - Users:
    - All users are identified by a unique number called the *User ID*, or *UID*.
  - Groups:
    - Each user is a member of one or more *user groups*, which are identified by a unique number called a *user group ID*.
- Reason:
  - The operating system must ensure that the private portion of a user space is visible only to its owner. In particular, it must ensure that no user can exploit a system application for the purpose of violating the private space of another user.

```
-rw-r-----  1 johnson johnson 84590989 Aug 27 14:20 linux-3.8.10.tar.bz2
```

- *root* or *superuser*:
  - The root user can do almost everything, because the operating system does not apply the usual protection mechanisms to her.

Processes

- A process can be defined either as “an instance of a program in execution” or as the “execution context” of a running program.
- A process executes a single sequence of instructions in an *address space*; the address space is the set of memory addresses that the process is allowed to reference.
  - Modern operating systems allow processes with multiple execution flow
- Systems that allow concurrent active processes are said to be multiprogramming or multiprocessing.
- It is important to distinguish programs from processes:
  - Several processes can execute the same program concurrently, while the same process can execute several programs sequentially.

Processes (Cont.)

- Scheduler
  - Choose the process that can progress.
  - Some operating systems only allow nonpreemptable processes which means that the scheduler is invoked only when a process voluntarily relinquishes the CPU.
  - Linux is a preemptable operating system.
- Example:
  - When a user is logged in, the process creates another process that runs a shell into which commands are entered.
  - If a graphical display is activated, one process runs the window manager, and each window on the display is usually run by a separate process.
  - When a user creates a graphics shell, one pone process runs the graphics windows and a second process runs the shell into which the user can enter the commands.

The Flow of a System Call

- 1. Whenever a process makes a system call (i.e., a request to the kernel), the hardware changes the privilege mode from User Mode to Kernel Mode, and the process starts the execution of a kernel procedure with a strictly limited purpose.
- 2. Then, the operating system acts within the execution context of the process in order to satisfy its request.
- 3. Whenever the request is fully satisfied, the kernel procedure forces the hardware to return to User Mode and the process continues its execution.

Kernel Architecture

- Linux is a monolithic kernel
  - Each kernel layer is integrated into the whole kernel program and runs in Kernel Mode on behalf of the current process.
- In contrast, microkernel demands a very small set of functions from the kernel.
  - Some system processes that run on top of the microkernel implement other OS functions, e.g., memory allocators, device driers, and system call handlers.
  - Advantages:
    - Force the system programmers to adopt a modularized approach.
    - Port to other architectures fairly easily .
    - Make better use of random access memory.

Kernel Architecture - Modules

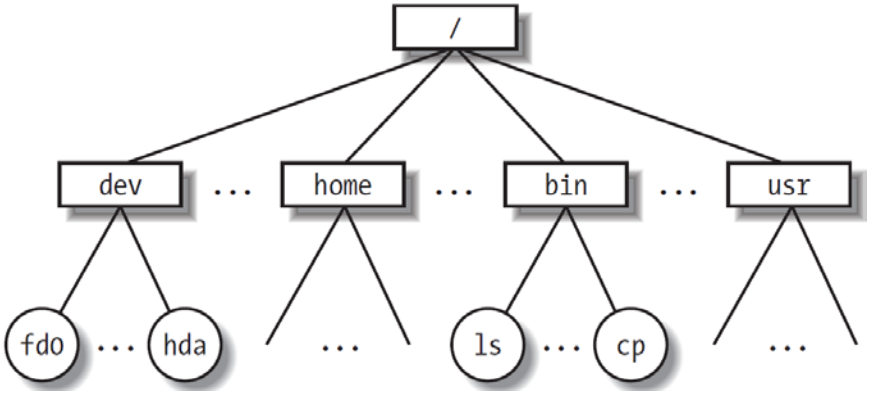
- By taking advantage of microkernels, Linux offers modules
  - A module is an object file whose code can be linked to the kernel at runtime.
  - Modules are executed in Kernel Mode on behalf of the current process, like any other statically linked kernel functions.
- The advantages of using modules:
  - A modularized approach
    - Linked/unlinked at runtime with well-defined software interfaces.
  - Platform independence
  - Frugal main memory usage
  - No performance penalty
    - No explicit message passing is required.



An Overview of the Unix Filesystem

Files

- Files are organized in a tree-structured namespace.
- A directory node contains information about the files and directories just beneath it.
- A file or directory name consists of a sequence of arbitrary ASCII characters (or Unicode), with the exception of / and of the null character \0.
- Most filesystems place a limit on the length of a filename.
- Root directory is denoted as /



Working Directory

- Linux associates a *current working directory* with each process.
- The process uses a *pathname*, which consists of slashes alternating with a sequence of directory names that lead to the file.
  - If the first item in the pathname is a slash, the pathname is said to be absolute, because its starting point is the root directory.
  - If the first item is a directory name or filename, the pathname is said to be relative.
- Special notation:
  - "." denotes the current working directory.
  - ".." denotes the parent directory of the current working directory.

Hard and Soft Links

- Hard link
  - A file name includes in a directory is called a file hard link.
  - **$ ln p1 p2** // Create a new hard link that has the pathname p2 for a file identified by the pathname p1.
  - Limitation:
    - It is not possible to create hard links for directories.
    - Links can be created only among files included in the same filesystem.
- Soft link (symbolic link)
  - Symbolic links are short files that contain an arbitrary pathname of another file.
  - $ ln –s p1 p2 // Create a new soft link with p2 that refers to p1.

File Types

- Linux file types:
  - Regular file
  - Directory
  - Symbolic link

  } Constituents of a filesystem

  - Block-oriented device file
  - Character-oriented device file

  } Related to I/O devices

  - Pipe and named pipe (called FIFO)
  - Socket

  } For interprocess communication

File Descriptor and Inode

- Linux makes a clear distinction between the contents of a file and the information about a file.
- All the information needed by the filesystem to handle a file is included in a data structure is called *inode*.
- Each file has its own inode.
- Information about a file:
  - File type
  - Number of hard links associated with the file
  - File length in bytes
  - Device ID (i.e., an identifier of the device containing the file)
  - Inode number that identifies the file within the filesystem
  - UID of the file owner
  - User group ID of the file
  - Several timestamps that specify the inode status change time, the last access time, and the last modify time
  - Access rights and file mode

Access Rights and File Mode

- The potential users of a file:
  - The user who is the owner of the file
  - The users who belong to the same group as the file.
  - All remaining users (others)
- Three types of access rights:
  - *Read*, *write* and *execute*
- Three additional flags
  - suid (Set User ID)
    - If the executable file has the *suid* flag set, the process gets the UID of the file owner.
  - sgid (Set Group ID)
    - If the executable file has the *sgid* flag set, the process gets the user group ID of the file.
  - sticky
    - An executable file with the *sticky* flag set corresponds to a request to the kernel to keep the program in memory after its termination.

File-Handling System Calls

- The reason to defined system calls:
  - A filesystem is a user-level view of the physical organization of a hard disk partition.
  - A process in User Mode cannot interact with the low-level hardware components, so each actual file operation must be performed in Kernel Mode.
- Opening a file with system call *open()*
  - **fd = open(path, flag, mode)**
    - path: Denotes the pathname of the file to be opened.
    - flag: Specifies how the file must be opened (e.g., read, write, read/write, append)
    - mode: Specifies the access rights of a newly created file.
  - The system call open() creates an “open file” object and returns an identifier called a *file descriptor*, which contains:
    - 1. Some file-handling data structures: e.g., offset field (i.e., file pointer)
    - 2. Some pointers to kernel functions that the process can invoke.
      - The set of permitted functions depends on the value of the flag parameter.

File Descriptor in POSIX Semantics

- A *file descriptor* represents an interaction between a process and an opened file.
- An *open file object* contains data related to that interaction.
  - The same open file object may be identified by several file descriptors in the same process.
- Several processes may concurrently open the same file.
  - The filesystem assigns a separate file descriptor to each opened file, along with a separate open file object.
  - Filesystems does not provide any kind of synchronization among the I/O operation issued by the processes on the same file.
  - Several systems calls such as flock() are available to allow processes to synchronize on the file.
  - To create a new file, the process may invoke the create() system call, which is handled by the kernel exactly like open().

Accessing an Opened File

- Two major types of files
  - Regular Linux file can be addressed sequentially or randomly.
  - Devices and named pipes are usually accessed sequentially.
- In the open file object, Linux kernel stores the file pointer to indicate the current position at which the next read or write operation will take place.
  - Sequential access is implicitly assumed:
    - Read() and write() system calls refer to the position of the current file pointer.
    - To modify the current file pointer, a program must invoke lseek() system call.
    - When a file is opened, the kernel sets the file pointer to the first position of the file.

Accessing an Opened File (Cont.)

- The lseek() system call:
  - newoffset = lseek(fd, offset, whence)
    - fd: Indicate the file descriptor
    - offset: Specify a signed integer value to compute the new position of the file pointer
    - whence: Specify whether the new position should be computed by adding the offset value to the number 0 (beginning of the file), the current file pointer, or the position of the last byte.
- The read() system call:
  - nread= read(fd, buf, count)
    - fd: Indicate the file descriptor
    - buf: Specify the address of the buffer in the process's address space
    - count: Denotes the number of bytes to read
    - The returned *nread* value specifies the number of bytes effectively read.
- The close() system call:
  - res = close(fd)
    - Release the open file object corresponding to the file descriptor.
    - When a process terminates, the kernel closes all its remaining opened files.

Renaming and Deleting a File

- To rename or delete a file, a process does not need to open it.
  - The rename() system call:
    - res = rename(oldpath, newpath)
      - Change the name of a file link
  - The unlink() system call:
    - res = unlink(pathname)
      - Decrease the file link count and remove the corresponding directory entry.
      - The file is deleted only when the link count assumes the value 0.



An Overview of Unix Kernels

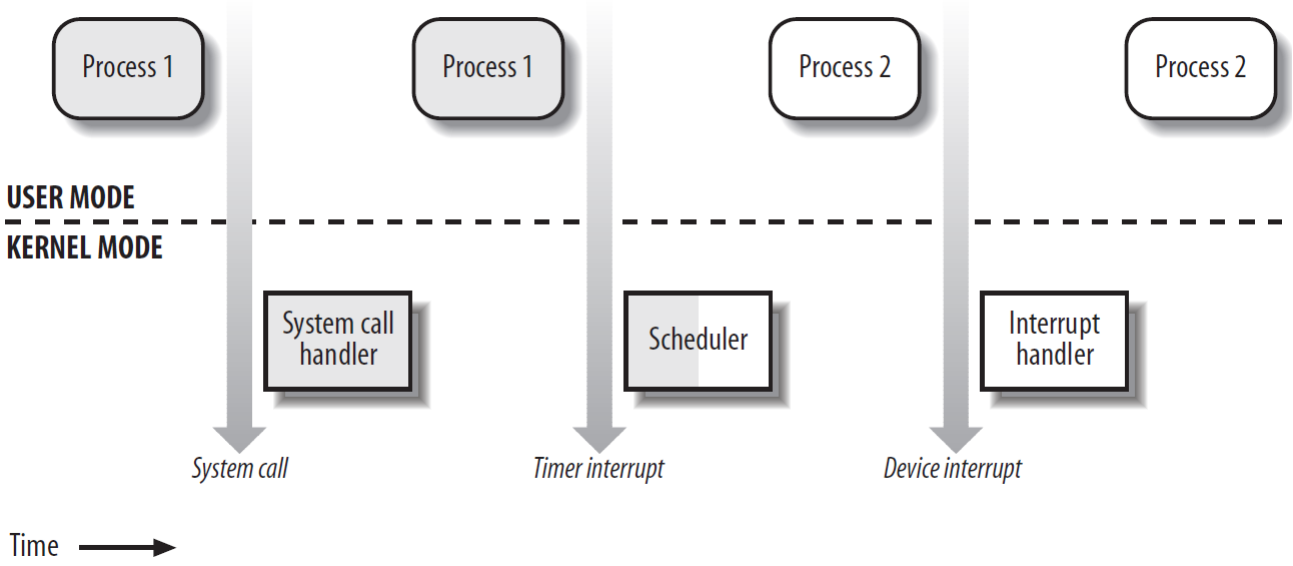
Overview

- Kernels provide an execution environment in which applications may run.
- The kernel must implement a set of services and corresponding interfaces.
- Applications use those interfaces and do not usually interact directly with hardware resources.
- A program executes in User Mode and switches to Kernel Mode only when requesting a service provided by the kernel. It is put back to User Mode when the kernel satisfies the program’s request.

The Process/Kernel Model

- Processes are dynamic entities that usually have a limited life span within the system.
  - The task of creating, eliminating, and synchronizing the existing processes is delegated to a group of routines in the kernel.
- Kernel threads are a few privileged processes:
  - Run in Kernel Mode in the kernel address space
  - Do not interact with users
  - Created at system startup and remain alive
- **Kernel is not a process but is a process manager**.
  - Processes that require a kernel service use specific programming constructs "system calls".

Transitions between User and Kernel Mode



- 1. Process 1 in User Mode issues a system call, after which the process switches to Kernel Mode, and the system call is serviced.
- 2. Process 1 then resumes execution in User Mode until a timer interrupt occurs, and the scheduler is activated in Kernel Mode.
- 3. A process switch takes place, and Process 2 starts its execution in User Mode until a hardware device raises an interrupt.
- 4. As a consequence of the interrupt, Process 2 switches to Kernel Mode and services the interrupt.

Kernel Routine Activation

- Kernel routine can be activated in several ways:
  - A process invokes a system call.
  - The CPU executing the process signals an exception, which is an unusual condition, e.g., an invalid instruction.
  - A peripheral device issues an interrupt signal to the CPU.
    - Each interrupt signal is dealt by a kernel program called interrupt handler.
  - A kernel thread is executed
    - Because kernel threads are run in Kernel Mode and are part of the kernel.

Process Implementation

- Each process is represented by a process descriptor that maintains the current state of the process.
- When the kernel stops the execution of a process, it saves the current contents of several processor registers in the process descriptor. These include:
  - The program counter (PC) and stack pointer (SP) registers
  - The general purpose registers
  - The floating point registers
  - The processor control registers (Processor Status Word)
  - The memory management registers (to keep track of the RAM accessed by the process)
- When a process is not executing, its process descriptor is put in the queue corresponding to its waiting event.

Reentrant Kernels

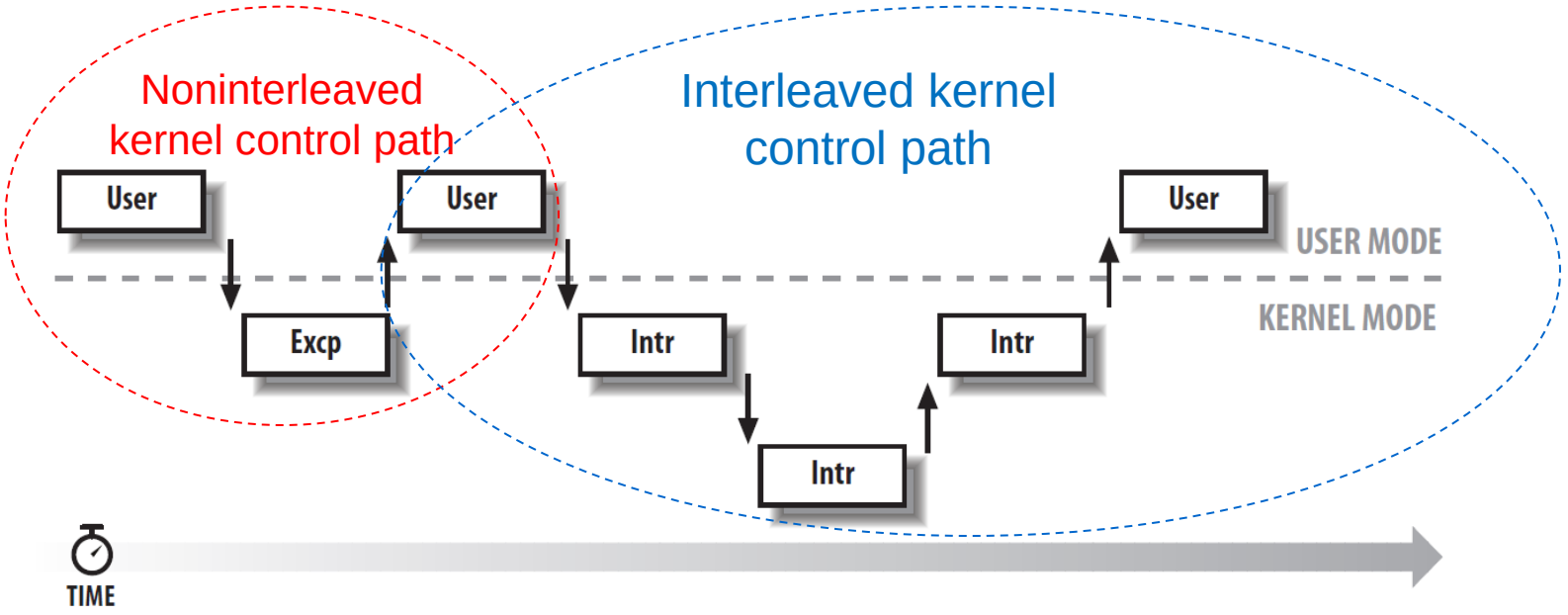
- Linux kernels are reentrant.
  - Several processes can be executing in Kernel Mode at the same time.
  - On uniprocessor systems, only one process can progress, but many can be blocked in Kernel Mode when waiting for the CPU or the completion of I/O.
    - E.g., Issuing a read to a disk on behalf of a process.
  - One way to provide reentrancy is to write functions (i.e., reentrant functions) that modify only local variable.
  - Reentrant kernels can include nonreentrant functions with *locking mechanisms* to ensure that only one process can execute a nonreentrant function at a time.
  - If a hardware interrupt occurs, a reentrant kernel is able to suspend the current running process even if that process is in the Kernel Mode.
    - This capability improves the throughput of the device controllers that issue interrupts.
      - Once a device has issued an interrupt, it waits until the CPU acknowledges.

Kernel Control Path

- A kernel control path denotes the sequence of instructions executed by the kernel to handle (1) a system call, (2) an exception, or (3) an interrupt.
- The CPU interleaves the kernel control paths:
  - A process in User Mode invokes a system call, and the corresponding kernel control path verify that the request cannot be satisfied.
    - It invokes the scheduler to select a new process (in the Kernel Mode) to run. Thus, the kernel control path is switched due to the process switch.
  - The CPU detects an exception, e.g., a page fault, while running a kernel control path.
    - The first control path is suspended, and the CPU starts the execution of a suitable procedure.
      - E.g., This type of procedure can allocate a new page for the process and read the content from disk, and then the first control path is resumed. In this case, the two kernel control paths run in the execution context of the same process.
  - 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 start processing another kernel control path to handle the interrupt.
      In this case, the two kernel control paths run in the execution context of the same process.
  - An interrupt occurs while the CPU is running with kernel preemption enabled and a higher priority process is runnable.
    - The first kernel control path is left unfinished, and the CPU resumes executing another kernel control path on behalf on the higher priority process.

Kernel Control Path (Example)

- Example:
  - 1. Running a process in User Mode (User)
  - 2. Running an exception or a system call handler (Excp)
  - 3. Running an interrupt handler (Intr)



Process Address Space

- Each process runs in its private address space.
  - When running in User Mode, a process refers to its private stack, data, and code areas.
  - When running in Kernel Mode, a process addresses the kernel data and code areas, and uses another private stack.
- Because the kernel is reentrant, several kernel control paths (each of which is related to a different process) may be executed in turn. Thus, each kernel control path refers to private kernel stack of its corresponding process.
- Shared address space
  - Sometimes, part of the private address space of some processes are shared to reduce memory usage.
    - E.g., if a program, say an editor, is needed by several users, the program is loaded into memory once shared by all of the users, but its data are not shared. (Automatically by kernel)
    - Processes share parts of their address space (called shared memory) as a kind of interprocess communication (IPC). (Shared by user)
  - The mmap() system call allows part of a file or the information stored in a block device to be mapped into a part of a process address.

Synchronization and Critical Regions

- Implementing a reentrant kernel requires the use of synchronization.
- Race condition occurs when the outcome of a computation depends on how two or more processes are scheduled.
  - Safe access to global variable is ensured by using atomic operations.
  - To achieve the atomicity, any section of code (called critical section/region) should be finished by each process that begins it before another process can enter it.

Kernel Preemption and Interrupt Disabling

- Kernel preemption disabling
  - To solve the synchronization problem, some traditional kernels are nonpreemptive: when a process executes in Kernel Mode, it cannot be arbitrarily suspended and substituted with another process.
  - Nonpreemptability is not enough for multiprocessor systems, because two kernel control paths running on different CPUs can concurrently access the same data structure.
- Interrupt disabling
  - A simple solution by disabling all hardware interrupts before entering a critical region and reenabling them right after leaving it.
  - Disabling interrupts on the local CPU is not sufficient on a multiprocessor system.

Semaphores

- A semaphore is simply a counter associated with a data structure. It is checked by all kernel threads before they try to access the data structure.
- A semaphore is composed of
  - An integer variable
  - A list of waiting processes
  - Two atomic methods: down() and up()
    - The down() method decreases the value of the semaphore. IF the new value is less than 0, the method adds the running process to the semaphore list and blocks.
    - The up() method increases the value of the semaphore. If the value of the new semaphore isn't negative, access to the data structure is granted.

Spin Locks

- If the time required to update the data structure is short, a semaphore could be very inefficient.
- Multiprocessor operating systems use spin locks.
  - A spin lock is very similar to a semaphore, but it has no process list.
  - When a process finds the lock closed by another process, it “spins” around until the lock becomes open.

Avoiding Deadlocks

- Processes or kernel control paths that synchronize with other control paths may easily enter a deadlock state.
  - E.g., Process p1 gain access to data structure a and process p2 gains access to b, but p1 then waits for b and p2 waits for a.
- Deadlocks become an issue when the number of kernel locks used is high.
- Several operating systems, including Linux, avoid this problem by requesting locks in a predefined order.

Signals

- Signals provide a mechanism for notifying processes of system events.
- Each event has its own signal number, which is referred to by a symbolic constant. There are two kinds of system events:
  - Asynchronous notifications
    - E.g., a user can send the interrupt signal SIGINT to a foreground process by pressing Ctrl+C at the terminal.
  - Synchronous notifications
    - E.g., the kernel sends the signal SIGSEGV (segmentation violation) to a process when it accesses a memory location at an invalid address.
- The POSIX standard defines about 20 different signals.
- A process may react to a signal delivery in two possible ways:
  - Ignore the signal.
  - Asynchronously execute a specified procedure (the signal handler).

Signals (Cont.)

- If no signal handler is defined in the process, the kernel performs a default action. Five possible default actions:
  - Terminate the process.
  - Write the execution context and the contents of the address space in a file (i.e., core dump) and terminate the process.
  - Ignore the signal.
  - Suspend the process.
  - Resume the process's execution if it was not stopped.
- The SIGKILL and SIGSTOP signals cannot be handled by the process or ignored.

Interprocess Communication (IPC)

- Interprocess communication includes semaphores, message queues, and shared memory, which are implemented as IPC resources.
  - Semaphores are used to synchronize among processes in User Mode.
  - Message queues allow processes to exchange messages by using the msgsnd() and msgrcv() system calls.
  - Shard memory provides the fastest way for processes to exchange and share data.
    - 1. A process starts by issuing a shmget() system call to create a new shared memory having a required size.
    - 2. After obtaining the IPC resource identifier, the process invokes the shmat() system call to have the starting address of the new region within the process address space.
    - When the process wishes to detach the shared memory, it invokes the shmdt() system call.

Process Management

- The fork() system call is used to create a new process.
  - The process that invokes a fork() is the parent, and the new process is its child.
    - Parent: returned value = child's PID
    - Child: returned value = 0
  - The current kernels that can rely on hardware paging units follow the Copy-On-Write (COW) approach, which defers page duplication until the last moment.
- The _exit() system call is used to terminate a process.
  - Release the resources owned by the process and send the parent process a SIGCHILD signal.
- The exec()-like system call is invoked to load a new program.
  - After the exec()-like system call is executed, the process resumes execution with a brand new address space containing the loaded program.

Zombie Processes

- Zombie Process:
  - The process that has terminated, but whose parent hasn't yet waited for it.
- A special zombie process state is introduced to represent terminated processes:
  - A process remains in that state until its parent process executes a wait4() system call on it.
  - If no child process has already terminated when the wait4() system call is executed, the kernel usually puts the process in a wait state until a child terminates.
  - Many kernels also implement a waitpid() system call, which allows a process to wait for a specific child process.
- If the parent process terminates without issuing that call, the information takes up valuable memory slots that could be used to serve living processes.
  - E.g., many shells allow the user to start a command in the background and then log out.
- The solution to zombie processes lies in a special system process called *init*, which is created during system initialization.
  - When a process terminates, all of its children are set as the children of *init*, which routinely issues wait4() system calls.

Process Groups and Login Sessions

- A job is composed of a process group.
  - $ ls | sort | more
    - This command creates a new group for the three processes corresponding ls, sort, and more.
    - The shell acts on the three processes as if there were a single entity (i.e., job).
    - Each group of processes may have a group leader, which is the process whose PID coincides with the process group ID.
- Login session
  - A login session contains all processes that are descendants of the process that has started a working session on a specific terminal.
  - All processes in a process group must be in the same login session.
  - A login session may have several process groups.
  - The terminal command *bg* and *fg* can be used to put a process group in the background or the foreground.

Virtual Memory

- Virtual memory acts as a logical layer between the application memory requests and the hardware memory management unit (MMU).
  - Several processes can be executed concurrently.
  - Applications' memory needs could be larger than the physical memory.
  - Processes can execute a program whose code is partially loaded in memory.
  - Each process is allowed to access a subset of the available physical memory.
  - Processes can share a single memory image of a library program.
  - Programs can be relocatable.
  - Programmers can write machine-independent code.

Virtual Memory (Cont.)

- The set of memory references that a process can use is the *virtual address space*.
  - When a process uses a virtual address, the kernel and the MMU cooperate to find the actual physical location of the requested memory item.
  - CPUs include hardware circuits that automatically translate the virtual addresses into physical ones.
  - The available RAM is partitioned into page frames (typically 4KB or 8KB).
  - A set of Page Tables is introduced to specify how virtual addresses correspond to physical addresses.
  - A request for contiguous virtual addresses can be satisfied by allocating a group of page frames having noncontiguous physical addresses.

Random Access Memory Usage

- A few megabytes of RAM are dedicated to storing the kernel image (i.e., the kernel code and the kernel static data structures).
- The remaining RAM is usually handled by the virtual memory system in three possible ways:
  - To satisfy kernel requests
    - for buffers, descriptors, and other dynamic kernel data structures.
  - To satisfy process requests
    - for generic memory areas and for memory mapping of files.
  - To get better performance from disks and other buffered devices by means of cache.
- A page-frame-reclaiming algorithm is invoked to free additional memory when the free memory is low.
- Memory fragmentation
  - Since the kernel is often forced to use physically contiguous memory areas, the memory request could fail even if there is enough memory available but not contiguous.

Kernel Memory Allocator (KMA)

- Kernel memory allocator (KMA) is a subsystem that tries to satisfy the requests for memory areas from all parts of the system.
  - Some requests come from kernel subsystems needing memory for kernel use.
  - Some requests come via systems calls from user processes to increase their processes' address space.
- Features of a good KMA:
  - Be fast. This is the most crucial attribute.
  - Minimize the amount of wasted memory.
  - Reduce the memory fragmentation problem.
  - Cooperate with the other memory management subsystems to borrow or release pages frames from them.
- Popular algorithms for KMA
  - E.g., resource map allocator, power-of-two free lists, McKusick-Karels allocator, buddy system, Mach's zone allocator, Dynix allocator, and slab allocator.

Process Virtual Address Space

- The address of a process contains all the virtual memory addresses that the process is allowed to reference.
- When a process starts the execution of some program via an exec()-like system call, the kernel assigns to the process a virtual address space that is maintained with a list of memory area descriptors and is comprised of memory areas for:
  - The executable code of the program
  - The initialized data of the program
  - The uninitialized data of the program
  - The executable code and data of needed shared libraries
  - The initial program stack (i.e., the User Mode stack)
  - The heap

Process Virtual Address Space (Cont.)

- Linux adopts a memory allocation strategy called demand paging.
  - 1. A process can start program execution with none of its pages in physical memory.
  - 2. As it accesses a nonpresent page, the MMU generates an exception.
  - 3. The exception handler finds the affected memory region, allocates a free page, and initializes it with the appropriate data.
    - A page frame is assigned to the process only when it generates an exception by trying to refer its virtual memory addresses.
- When the process dynamically requires memory by using malloc() or brk() system call, the kernel just updates the size of the heap memory region of the process.
- Virtual address spaces allow other efficient strategies such as copy on write (COW):
  - E.g., when a new process is created, the kernel just assigns the parent's page frames to the child address space but marks them read-only.
    - An exception is raised as soon as the parent or the child tries to modify the contents of a page. The exception handler assigns a new page frame to the affected process and initializes it with the contents of the original page.

Caching

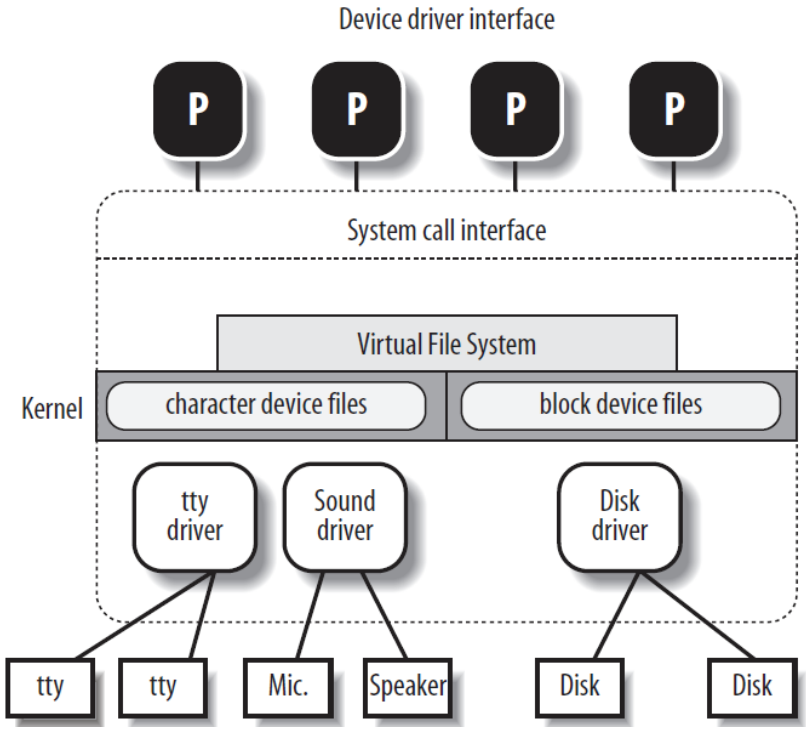
- Available physical can be used as cache for hard disks and other block devices because hard disks are very slow.
- The sync() system call forces disk synchronization by writing all of the “dirty” buffers into disk.

Device Drivers

- Device drivers:
  - The kernel interacts with I/O devices by means of device drivers.
  - Device drivers are included in the kernel and consist of data structures and functions that control one or more devices.
  - Each driver interacts with the kernel (and/or even with other drivers) through a specific interface.
- This approach has the following advantages:
  - Device-specific code can be encapsulated in a specific module.
  - Venders can add new devices without knowing the kernel source code.
  - The kernel deals with all devices in a uniform way and access them through the same interface.
  - A device driver implemented as a module can be dynamically loaded in the kernel without requiring the system to be rebooted.

Device Drivers (Cont.)

- Example:
  - 1. Some user program P wish to operate on hardware devices.
  - 2. It makes requests to the kernel using the usual file-related system calls and the device files normally found in the /dev directory.
    - Each device file refers to a specific device driver, which is invoked by the kernel to perform the requested operation on the hardware.





Spare Slides

An Example of mmap()

```
offset = 0;start = 0;

/* 打開檔案 */
fd = open( "/home/tester/a.txt", O_RDWR O_SYNC );

/* 作mmap動作，取得一個對應好的address */
ptr = mmap( 0, map_size, PROT_READPROT_WRITE,
MAP_SHARED, fd, offset );

/*假如一切順利的話 ptr 就會接到一個address，這個address會對
應到a.txt這個檔案所在的起始位置，如果這時候我們用*/
strcpy( ptr, "hello!!" );
/* a.txt裡面就會被寫入"hello!!"的字串*/
```

Semaphore

```
//The definition of wait() is as follows:
wait(S) {
    while (S <= 0); // busy wait
    S--;
}

//The definition of signal() is as follows:
signal(S) {
    S++;
}
```

An Example of fork() and exec()

```
int var_glb; /* A global variable*/
int main(void)
{
    pid_t childPID;

    childPID = fork();
    if(childPID == 0) // child process
        {
            printf("\n Child Process :: var_lcl = [%d], var_glb[%d]\n", var_lcl, var_glb);
            execl("/bin/ls", "/bin/ls", "-r", "-t", "-l", (char *) 0);
        }
    else //Parent process
        {
            printf("\n Parent process :: var_lcl = [%d], var_glb[%d]\n", var_lcl, var_glb);
        }
    }
    return 0;
}
```

An Signal Handler Example

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>

void sig_handler(int signo)
{
  if (signo == SIGINT)
    printf("received SIGINT\n");
}

int main(void)
{
  if (signal(SIGINT, sig_handler) == SIG_ERR)
  printf("\ncan't catch SIGINT\n");
  // A long long wait so that we can easily issue a signal to this process
  while(1)  sleep(1);
  return 0;
}
```

An Example of errno

```
#include <errno.h>
#include <string.h>

/* ... */

if(read(fd, buf, 1)==-1) {
    printf("Oh dear, something went wrong with read()! %s\n", strerror(errno));
}
```

```
if (somecall() == -1) {
        printf("somecall() failed\n");
        if (errno == ...) { ... }
    }
```