Advanced System Software

Introduction

What is an OS kernel?

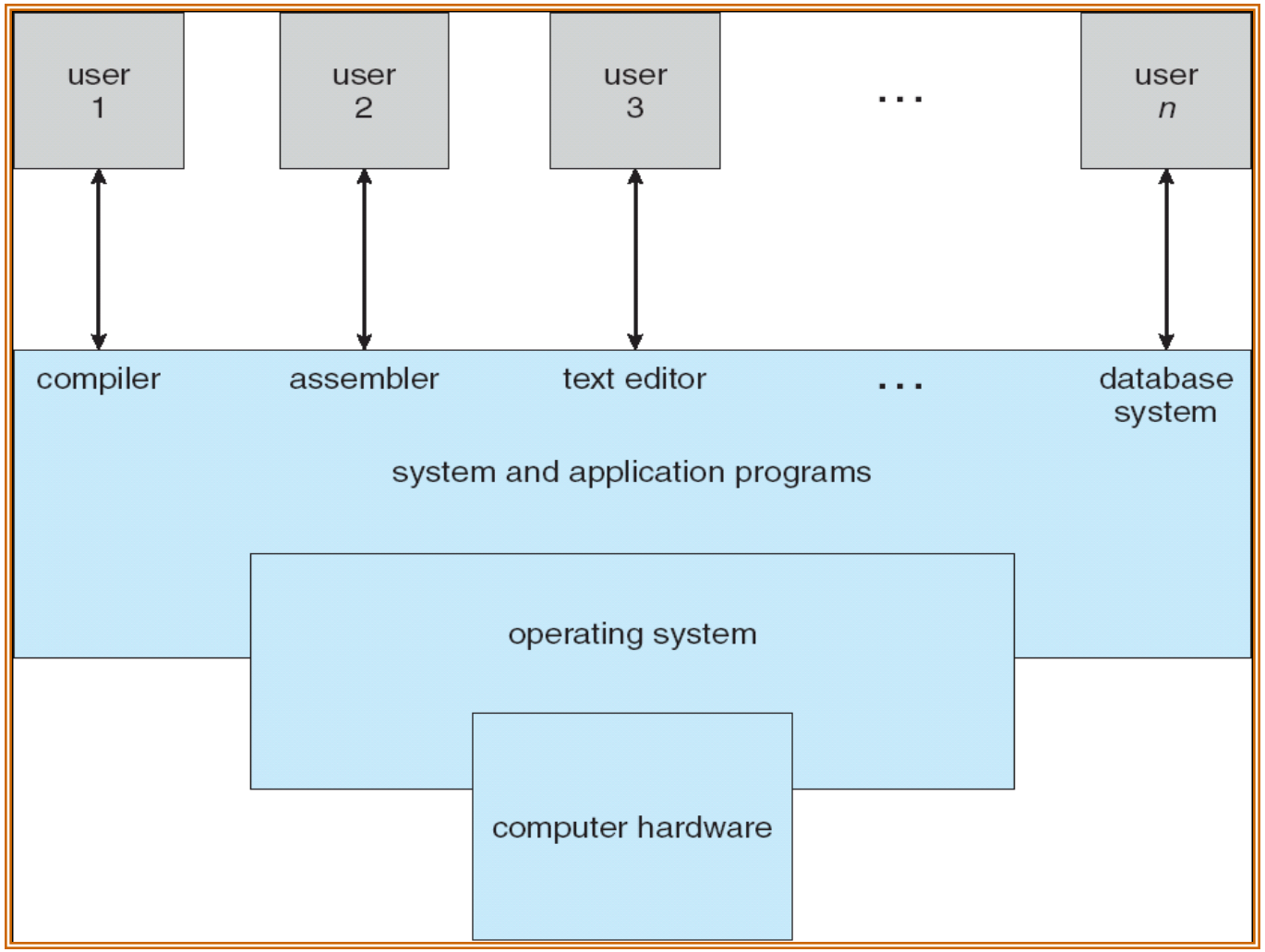
Program that is always running.

- Manages resources.
- Provides services.

Layering

- Layer between programs and hardware.
- Layer between users (multiuser OS).
- Layer between programs (multitasking OS).

What is an OS kernel?



Resource Management

Allocation

- Allocates finite resources among competing processes.
- CPU, memory, disk, network

Protection

- Prevents processes from interfering with each other.

Reclamation

- Voluntary at runtime; automatic at termination.

Virtualization

- Provides illusion of private unshared resources
- Timeshared CPU, Virtual Memory, Virtual Machines

What is the Linux kernel vs Linux OS?

Linux is a member of the large family of Unix-like operating systems.

Free open source UNIX-compatible kernel.

- Created by Linus Torvalds.
- Developed by thousands across the world.
- Coordinated via linux-kernel mailing list.

Kernel History

0.01 First version released by Linus (1991).

1.0 First release (x86 only) in 1994.

1.2 Supports other CPUs (Alpha, MIPS) in 1995.

2.0 SMP support, more architectures (1996).

2.2 Efficient SMP, more hardware support (1999).

2.4 LVM, Plug-n-Play, USB, etc. (2001).

2.6 Scalability (embedded, NUMA, PAE, sched), kernel pre-emption, User-mode linux (2003).

Version Numbering: A.B.C.D

A: Major version

Changed twice: 1.0 (1994), 2.0 (1996)

B: Minor version

Even numbers are stable releases

Odd numbers are development releases

C: Minor revision

Not so minor in 2.6 as development continues.

D: Bug-fix / security patch release

First occurred with NFS bug in 2.6.8.1

Official policy as of 2.6.11

Kernel Versions

mm: Andrew Morton tree

- New patches, almost ready for distribution.

ac: Alan Cox tree

Distribution trees

- RedHat
- Mandrake
- Debian
- Gentoo, etc.

- Linux is a true Unix kernel, although it is not a full Unix operating system, because it does not include all the applications such as filesystem utilities, windowing systems and graphical desktops, system administrator commands, text editors, compilers, and so on.

Assessment of Linux Vs Commercial Unix Kernels

- Linux kernel is monolithic.
- Traditional Unix kernels are compiled and linked statically.
- Kernel threading.
- Multithreaded application support.
- Linux is a nonpreemptive kernel.
- Multipurpose support
- File Systems
- STREAMS

Advantages of Linux

- Linux is fully customizable in all its components.
- Linux runs on low-end, cheap hardware platforms.
- Linux is powerful.
- Linux has a high standard for source code quality.
- The Linux kernel can be very small and compact.
- Linux is highly compatible with many common operating systems.
- Linux is well supported.

Hardware Dependency

- Linux tries to maintain a neat distinction between hardware-dependent and hardware independent source code.
  - arm - Acorn personal computers
  - i386- IBM-compatible personal computers based on Intel 80x86 or Intel 80x86-compatible microprocessors.
  - sparc - Workstations based on Sun Microsystems SPARC microprocessors.
  - m68k - Personal computers based on Motorola MC680x0 microprocessors

Multiusers Systems

- A multiuser system is a computer that is able to concurrently and independently execute several applications belonging to two or more users.
- Multiuser operating systems must include several features:
- An authentication mechanism for verifying the user identity.
- A protection mechanism against buggy user programs that could block other applications running in the system.
- A protection mechanism against malicious user programs that could interfere with, or spy on, the activity of other users.
- An accounting mechanism that limits the amount of resource units assigned to each user.
- In order to ensure safe protection mechanisms, operating systems must make use of the hardware protection associated with the CPU privileged mode. Otherwise, a user program would be able to directly access the system circuitry and overcome the imposed bounds.
- Unix is a multiuser system that enforces the hardware protection of system resources.

Users and Groups

- In a multiuser system, each user has a private space on the machine: typically, he owns some quota of the disk space to store files, receives private mail messages, and so on.
- The operating system must ensure that the private portion of a user space is visible only to its owner.
- All users are identified by a unique number called the User ID , or UID.
- In order to selectively share material with other users, each user is a member of one or more groups, which are identified by a unique number called a Group ID , or GID. Each file is also associated with exactly one group.
- Any Unix-like operating system has a special user called root, superuser, or supervisor. The system administrator must log in as root in order to handle user accounts, perform maintenance tasks like system backups and program upgrades, and so on.

Processes

- All operating systems make use of one fundamental abstraction: the process .
- A process can be defined either as "an instance of a program in execution," or as the "execution context" of a running program.
- In traditional operating systems, 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 flows,i.e., multiple sequences of instructions executed in the same address space.
- Systems that allow concurrent active processes are said to be multiprogramming or multiprocessing.
- Several processes can execute the same program concurrently, while the same process can execute several programs sequentially.

Identifying the Running Kernel

```
> uname
Linux
> uname -r
2.6.10
> cat /proc/version
Linux version 2.6.10(jw@csc660)
  (gcc version 3.3.5) #3 Sun Dec
  25 10:22:50 EST 2005
```

Investigating the Running Kernel: /proc

###: directory for each running process
cpuinfo: processor information
devices: supported hardware
diskstats: disk performance statistics
meminfo: memory usage information
modules: linux kernel modules
net: directory of network information
partitions: linux disk partitions
swaps: swap files/partitions in use by kernel
self: link to ### directory for current process

Process information

```
> ls -alF /proc/self
dr-xr-xr-x  2 jw jw 0 2005-12-29 13:46 attr/
-r--------  1 jw jw 0 2005-12-29 13:46 auxv
-r--r--r--  1 jw jw 0 2005-12-29 13:46 cmdline
lrwxrwxrwx  1 jw jw 0 2005-12-29 13:46 cwd -> /proc/20041/
-r--------  1 jw jw 0 2005-12-29 13:46 environ
lrwxrwxrwx  1 jw jw 0 2005-12-29 13:46 exe -> /bin/bash*
dr-x------  2 jw jw 0 2005-12-29 13:46 fd/
-r--r--r--  1 jw jw 0 2005-12-29 13:46 maps
-rw-------  1 jw jw 0 2005-12-29 13:46 mem
-r--r--r--  1 jw jw 0 2005-12-29 13:46 mounts
lrwxrwxrwx  1 jw jw 0 2005-12-29 13:46 root -> //
-r--r--r--  1 jw jw 0 2005-12-29 13:46 stat
-r--r--r--  1 jw jw 0 2005-12-29 13:46 statm
-r--r--r--  1 jw jw 0 2005-12-29 13:46 status
dr-xr-xr-x  3 jw jw 0 2005-12-29 13:46 task/
-r--r--r--  1 jw jw 0 2005-12-29 13:46 wchan
```

Process information

```
> cd /proc/self
> cat cmdline ; echo
-bash
> cat environ | tr '\0' '\n' | head -8
ENV_SET=1
MANPATH=/usr/local/man:/usr/man:/usr/share/man
PATH=/usr/ucb:/usr/bin:/bin:/sbin:/usr/sbin:/usr/local/bin
TERM=xterm
SHELL=/bin/bash
EDITOR=vim
VISUAL=vim
PAGER=less
> ls -l fd
total 4
lrwx------  1 jw jw 64 2005-12-29 13:50 0 -> /dev/pts/3
lrwx------  1 jw jw 64 2005-12-29 13:50 1 -> /dev/pts/3
lrwx------  1 jw jw 64 2005-12-29 13:50 2 -> /dev/pts/3
```

The Linux Kernel Archives

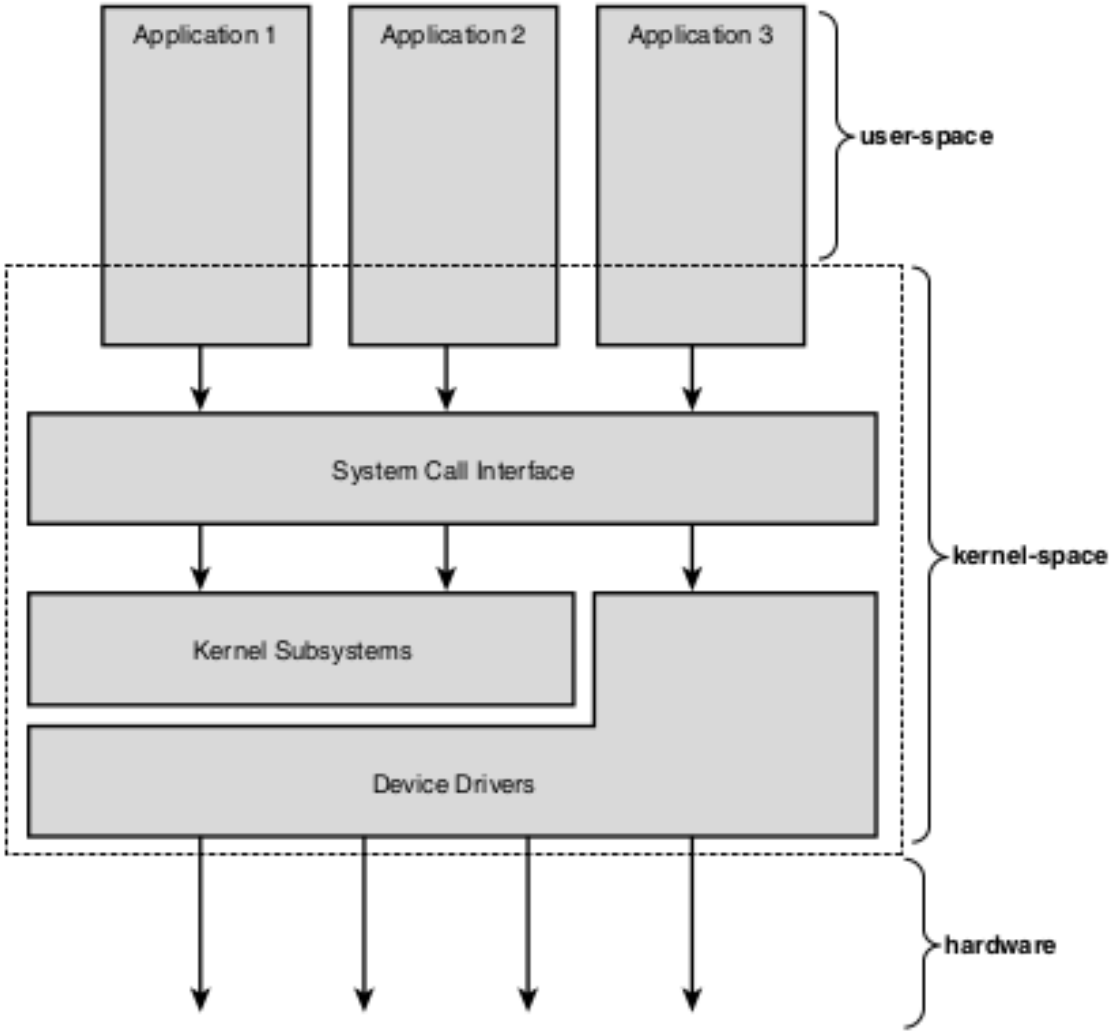
Welcome to the Linux Kernel Archives. This is the primary site for the Linux kernel source, but it has much more than just Linux kernels.

Protocol	Location
HTTP	http://www.kernel.org/pub/
FTP	ftp://ftp.kernel.org/pub/
RSYNC	rsync://rsync.kernel.org/pub/

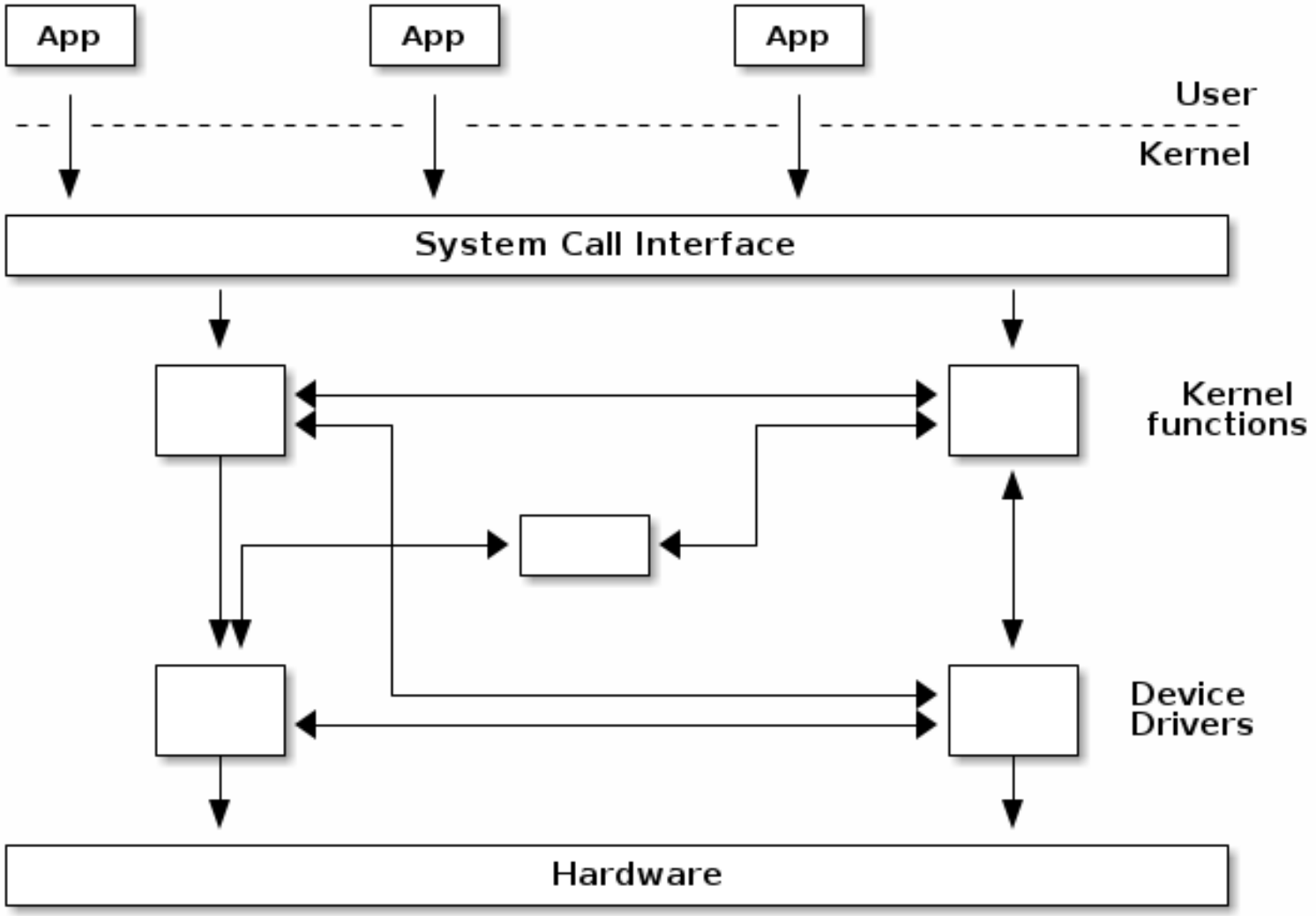
The latest stable version of the Linux kernel is:	**2.6.14.5**	2005-12-27 00:29 UTC	F	V	VI	C	Changelog
The latest prepatch for the stable Linux kernel tree is:	**2.6.15-rc7**	2005-12-25 03:39 UTC		V	VI	C	Changelog
The latest snapshot for the stable Linux kernel tree is:	**2.6.15-rc7-git3**	2005-12-29 08:01 UTC		V		C	
The latest 2.4 version of the Linux kernel is:	**2.4.32**	2005-11-16 19:13 UTC	F	V	VI	C	Changelog
The latest prepatch for the 2.4 Linux kernel tree is:	**2.4.33-pre1**	2005-12-29 16:18 UTC		V		C	Changelog
The latest 2.2 version of the Linux kernel is:	**2.2.26**	2004-02-25 00:28 UTC	F	V			Changelog
The latest prepatch for the 2.2 Linux kernel tree is:	**2.2.27-rc2**	2005-01-12 23:55 UTC		V	VI		Changelog
The latest 2.0 version of the Linux kernel is:	**2.0.40**	2004-02-08 07:13 UTC	F	V	VI		Changelog
The latest -ac patch to the stable Linux kernels is:	**2.6.11-ac7**	2005-04-11 18:36 UTC		V			
The latest -mm patch to the stable Linux kernels is:	**2.6.15-rc5-mm3**	2005-12-15 06:51 UTC		V			Changelog

F = full source, **V** = view patch, **VI** = view incremental, **C** = current changesets
Changelogs are provided by the kernel authors directly. Please don't write the webmaster about them.
Customize the patch viewer

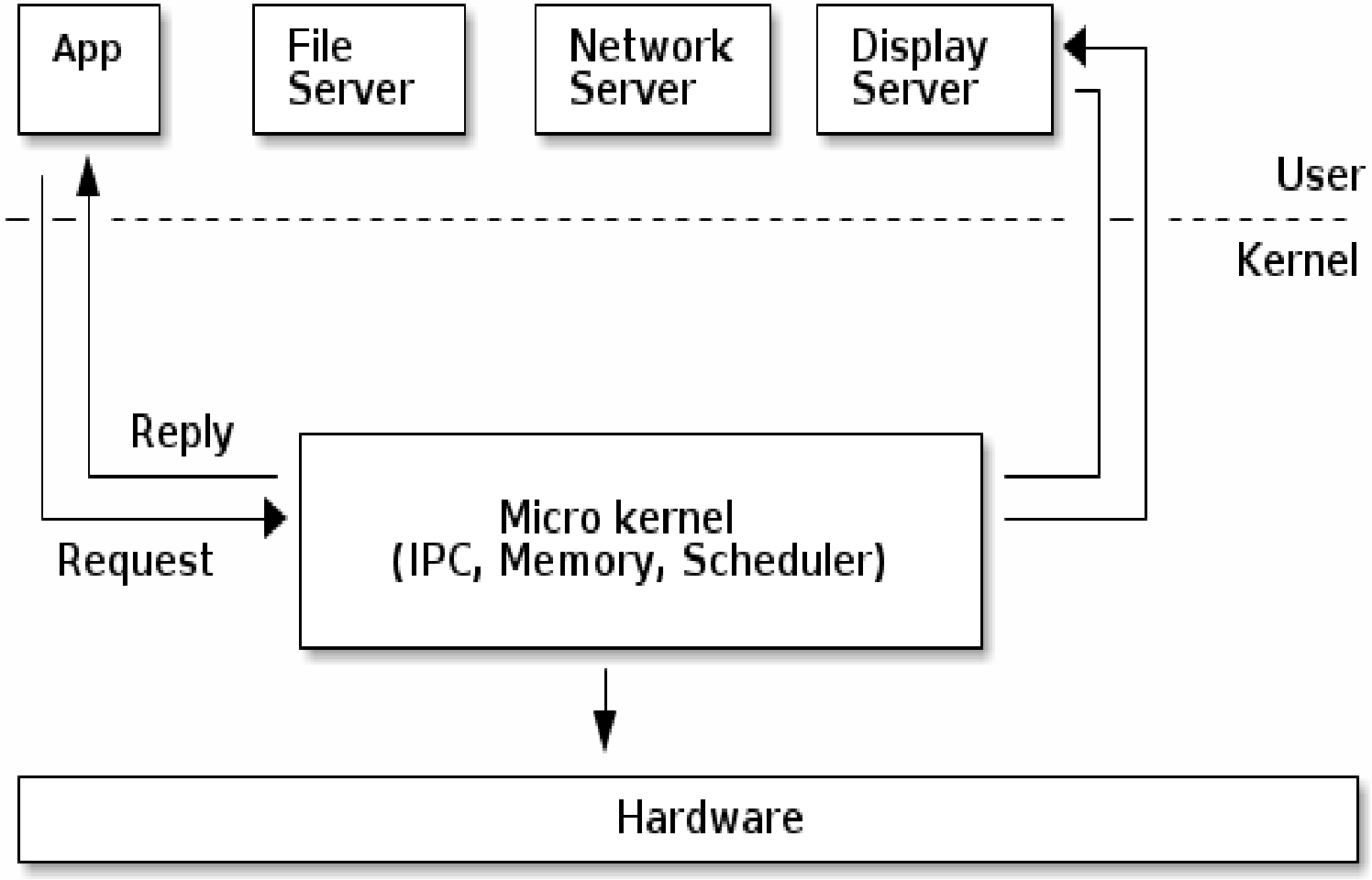
Kernel Architecture



Monolithic kernel



Micro Kernel



Monolithic v/s Micro

Parameters	Microkernel	Monolithic
Address space	User services and kernel services are kept in separate address space.	Both user services and kernel services are kept in the same address space.
Design	Complex	Easy
Size	Smaller	Larger
Functionality	Easier to add new functionality	Difficult to add new functionality
Coding	More Code is required to design a microkernel	Less code is required

Parameters	Microkernel	Monolithic
Failure	Failure of one component does not effect the working of micro kernel.	Failure of one component in a monolithic kernel leads to the failure of the entire system.
Processing speed	Execution speed is low	Execution speed is high
Extend	Easy to extend	Not easy to extend
Communication	To implement IPC messaging queues are used by the communication microkernels.	Signals and Sockets are utilized to implement IPC in monolithic kernels.

Parameters	Microkernel	Monolithic
Debugging	Simple	Difficult
Maintain	Simple	Extra time and resources are needed
Message passing and context switching	Required by the microkernel	Not required by the microkernel
Services	The kernel only offers IPC and low-level device management services.	The Kernel contains all of the operating system's services.
Performance	Lower due to message passing and more overhead	High due to direct function calls and less overhead
Examples	Mac OS	Microsoft 95

- Each kernel layer is integrated into the whole kernel program and runs in Kernel Mode on behalf of the current process.
- Microkernel operating systems demand a very small set of functions from the kernel, generally including a few interrupt handlers, sevice interrupts requests, synchronization primitives, a simple scheduler, a memory management and an interprocess communication mechanism.
- Several system processes that run on top of the microkernel implement other operating system-layer functions, like memory allocators, device drivers, system call handlers, and so on.
- Microkernels force the system programmers to adopt a modularized approach, since any operating system layer is a relatively independent program that must interact with the other layers through well-defined and clean software interfaces.

- An existing microkernel operating system can be fairly easily ported to other architectures, since all hardware- dependent components are generally encapsulated in the microkernel code.
- Finally, microkernel operating systems tend to make better use of random access memory (RAM) than monolithic ones, since system processes that aren't implementing needed functionalities might be swapped out or destroyed.
- Modules are kernel features that has advantage of microkernels without introducing performance penalties.
- A module is an object file whose code can be linked to (and unlinked from) the kernel at runtime.
- The object code usually consists of a set of functions that implements a filesystem, a device driver, or other features at the kernel's upper layer.

- The module, unlike the external layers of microkernel operating systems, does not run as a specific process. Instead, it is executed in Kernel Mode on behalf of the current process, like any other statically linked kernel function.
- The main advantages of using modules include -
- Modularized approach
- Platform independence
- Frugal main memory usage
- No performance penalty

Prepatches and Snapshots

Prepatches

Alpha versions of the kernel, located in the testing/ subdirectory of kernel.org.

Snapshots

Automatically created images of the kernel development tree. May not work or compile.

Obtaining the Kernel

Select the kernel version you need

- 2.6.10 for our course

Download the kernel

- wget ftp://ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.10.tar.bz2

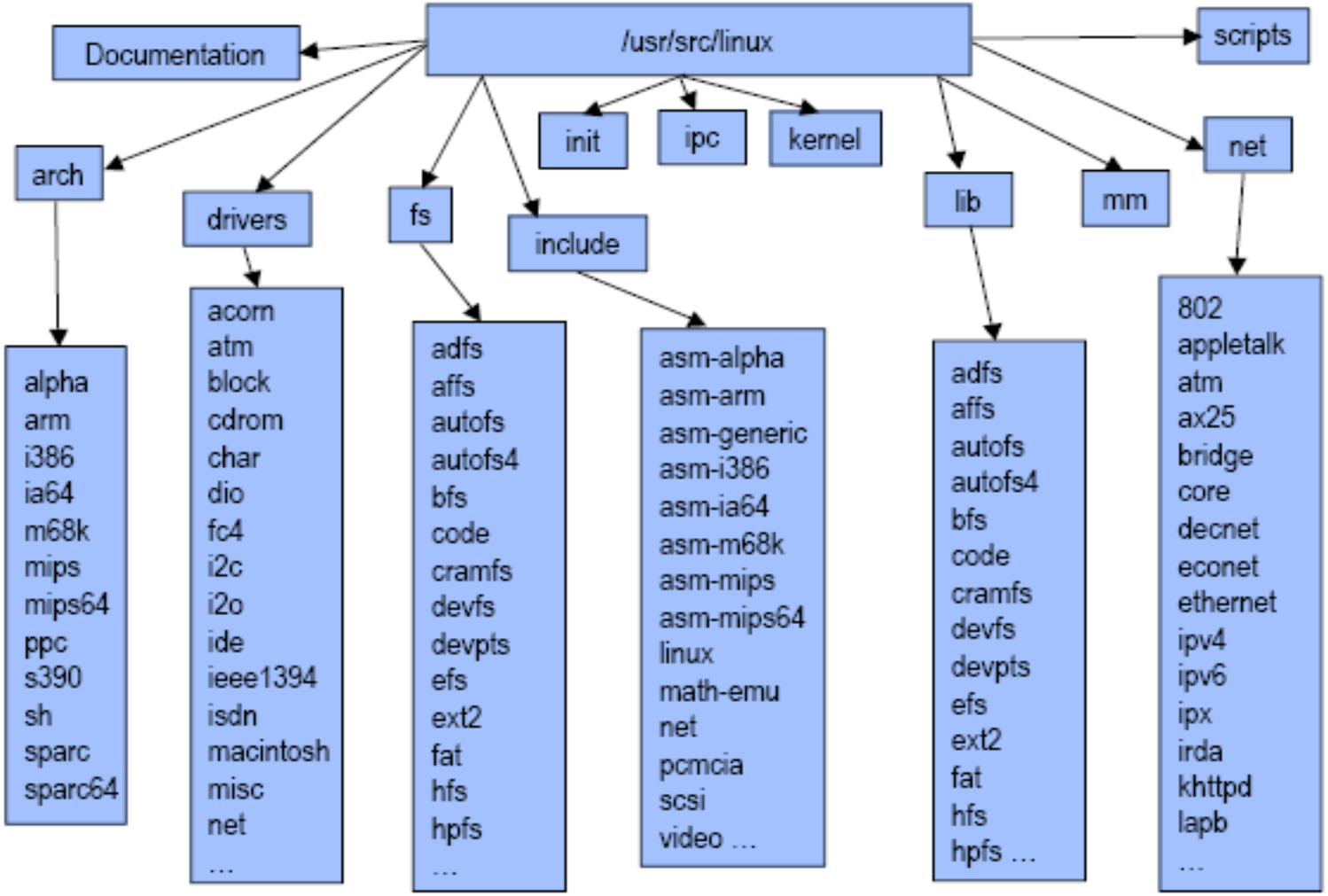
Unpack the kernel

- tar –xjf linux-2.6.10.tar.bz2

Overview of a Linux File system

- As in Unix file system, Linux file is an information container structured as a sequence of bytes; the kernel does not interpret the contents of a file.
- Many programming libraries implement higher-level abstractions, such as records structured into fields and record addressing based on keys.
- However, the programs in these libraries must rely on system calls offered by the kernel.
- From the user's point of view, files are organized in a tree-structured.

Linux Source Layout



Documentation

Text files documenting various aspects of kernel

- Can be very useful.
- Not well organized.
- Not always up to date.

arch

Architecture dependent code.

- Subdirectories for all current archs (22 in 2.6.10)
- i386 is subdirectory for x86 architecture

Contains architecture-dependent subdirs

- boot/
- kernel/
- lib/
- mm/

arch/i386

boot/	x86-specific boot code
crypto/	x86-optimized crypto functions
kernel/	x86-specific kernel core code
lib/	x86-optimized library code
mach-*/	x86 sub-architectures
math-emu/	FPU emulation code
oprofile/	kernel profiling tool
pci/	x86 PCI drivers
power/	x86 power management

drivers

Hardware device drivers.

Largest division of kernel code.

Major subdirectories

- char/
- block/
- net/
- scsi/
- buses (mca/, nubus/, pci/, sbus/)

fs

Virtual filesystem framework code.

Subdirectories for actual filesystems.

- ext2/
- ext3/
- hfs/
- isofs/
- nfs/
- ntfs/
- proc/
- ufs/

include

include/asm-*

Architecture dependent subdirectories.

include/linux

Headers needed by both kernel and user apps.

User programs use /usr/include/linux

Kernel parts guarded by

```
#ifdef __KERNEL__
…
#endif
```

init

Kernel startup code

version.c: kernel version banner; prints at boot.

main.c: architecture independent boot code.

- See arch/i386/boot for arch-dependent code.
- start_kernel() is entry point.

ipc

System V UNIX IPC

Interprocess Communication Facilities

sem.c	semaphors
shm.c	shared memory
msg.c	messages

kernel

Core kernel code.

Scheduler

- sched.c

Process Control

- fork.c, exit.c, signal.c, wait.c

Kernel module support

- kmod.c, module.c

Other facilities

- dma.c, ptrace.c, softirq.c, spinlock.c, timer.c

lib

Kernel cannot call standard C library.

lib/ contains kernel's own library routines

Optimized libraries found under arch/i386/lib

Examples

inflate.c	Unpacks kernel at boot.
rbtree.c	Red-black tree implementation.
string.c	String-handling routines.
vsprintf.c	Replacement for libc vsprintf().

mm

Memory management

memory.c	Page table setup, page faults
filemap.c	File memory mapping

Allocation and Deallocation

slab.c	Slab allocator
vmalloc.	Kernel virtual memory allocator

Paging and Swapping

vmscan.c	Paging policies
page_io.c	Low level paging I/O
swap.c	Swapping

net

Low level networking protocols

atm/, bluetooth/, decnet/ ethernet/

TCP/IP networking: ipv4/ and ipv6/

ipv4/icmp.c	ICMP
ipv4/netfilter/*.c	Netfilter firewall
ipv4/route.c	Routing code
ipv4/tcp.c	TCP
ipv4/udp.c	UDP

scripts

Kernel scripts for

- Building documentation
- Kernel configuration

Hard and Soft Links

- A filename included in a directory is called a file hard link, or more simply a link. The same file may have several links included in the same directory or in different ones, thus several filenames.
- The Unix command:

$ ln f1 f2

is used to create a new hard link that has the pathname f2 for a file identified by the pathname f1 .

- Hard links have two limitations:
  - Users are not allowed to create hard links for directories.
  - Links can be created only among files included in the same filesystem.
- In order to overcome these limitations, soft links (also called symbolic links) have been introduced.
- Symbolic links are short files that contain an arbitrary pathname of another file.
- The pathname may refer to any file located in any filesystem.

- The Unix command:

$ ln -s f1 f2

- creates a new soft link with pathname f2 that refers to pathname f1 .

File types

- Unix files may have one of the following types:
- Regular file
- Directory
- Symbolic link
- Block-oriented device file
- Character-oriented device file
- Pipe and named pipe (also called FIFO)
- Socket

File descriptor and Inode

- There is a clear distinction between a file and file descriptor.
- All information needed by the filesystem to handle a file is included in a data structure called an inode.
- Each file has its own inode, which the filesystem uses to identify the file.
- The filesystems and kernel functions handling them can vary from one system to another, but they must always provide atleast the following attributes:-

- 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
- User ID of the file owner
- 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 fall into three classes:
- The user who is the owner of the file
- The users who belong to the same group as the file, not including the owner
- All remaining users (others)
- Three types of access rights - Read, Write and Execute, for each of these three classes. The set of access rights associated with a file consists of nine different binary flags. Three additional flags include -

- suid - A process executing a file normally keeps the User ID (UID) of the process owner. However, if the executable file has the suid flag set, the process gets the UID of the file owner.
- sgid - A process executing a file keeps the Group ID (GID) of the process group. However, if the executable file has the sgid flag set, the process gets the ID of the file group.
- sticky - An executable file with the sticky flag set corresponds to a request to the kernel to keep the program in memory after its execution terminates.
- When a file is created by a process, its owner ID is the UID of the process.
- Its owner group ID can be either the GID of the creator process or the GID of the parent directory, depending on the value of the sgid flag of the parent directory.

File Handling System Calls

- Opening a file - Processes can access only "opened" files. In order to open a file, the process invokes the system call:

  fd = open(path, flag, mode)

- The three parameters have the following meanings:
- path - Denotes the pathname (relative or absolute) of the file to be opened.
- flag - Specifies how the file must be opened (e.g., read, write, read/write, append). It can also specify whether a nonexisting file should be created.
- mode - Specifies the access rights of a newly created file.

- This system call creates an "open file" object and returns an identifier called file descriptor .
- An open file object contains:
- Some file-handling data structures, like a pointer to the kernel buffer memory area where file data will be copied; an offset field that denotes the current position in the file from which the next operation will take place (the so-called file pointer); and so on.
- Some pointers to kernel functions that the process is enabled to invoke. The set of permitted functions depends on the value of the flag parameter.
- In order to create a new file, the process may also invoke the create( ) system call, which is handled by the kernel exactly like open( ) .

- Accessing an opened file - The kernel stores the file pointer in the open file object, that is, the current position at which the next read or write operation will take place.
- Sequential access is implicitly assumed: the read( ) and write( ) system calls always refer to the position of the current file pointer.
- In order to modify the value, a program must explicitly invoke the lseek( ) system call. When a file is opened, the kernel sets the file pointer to the position of the first byte in the file (offset 0).

- The lseek( ) system call requires the following parameters:

  newoffset = lseek(fd, offset, whence);
- which have the following meanings:
- fd - Indicates the file descriptor of the opened file.
- offset - Specifies a signed integer value that will be used for computing the new position of the file pointer.
- whence - Specifies whether the new position should be computed by adding the offset value to the number (offset from the beginning of the file), the current file pointer, or the position of the last byte (offset from the end of the file).

- The read( ) system call requires the following parameters:

  nread = read(fd, buf, count);
- which have the following meaning:
- fd - Indicates the file descriptor of the opened file
- buf - Specifies the address of the buffer in the process's address space to which the data will be transferred
- count - Denotes the number of bytes to be read

- Closing a file - When a process does not need to access the contents of a file anymore, it can invoke the system call:

  res = close(fd);

- which releases the open file object corresponding to the file descriptor fd
- When a process terminates, the kernel closes all its still opened files.

Overview of Unix Kernel

- Unix kernels provide an execution environment in which applications may run.
- Therefore, the kernel must implement a set of services and corresponding interfaces.
- Applications use those interfaces and do not usually interact directly with hardware resources.

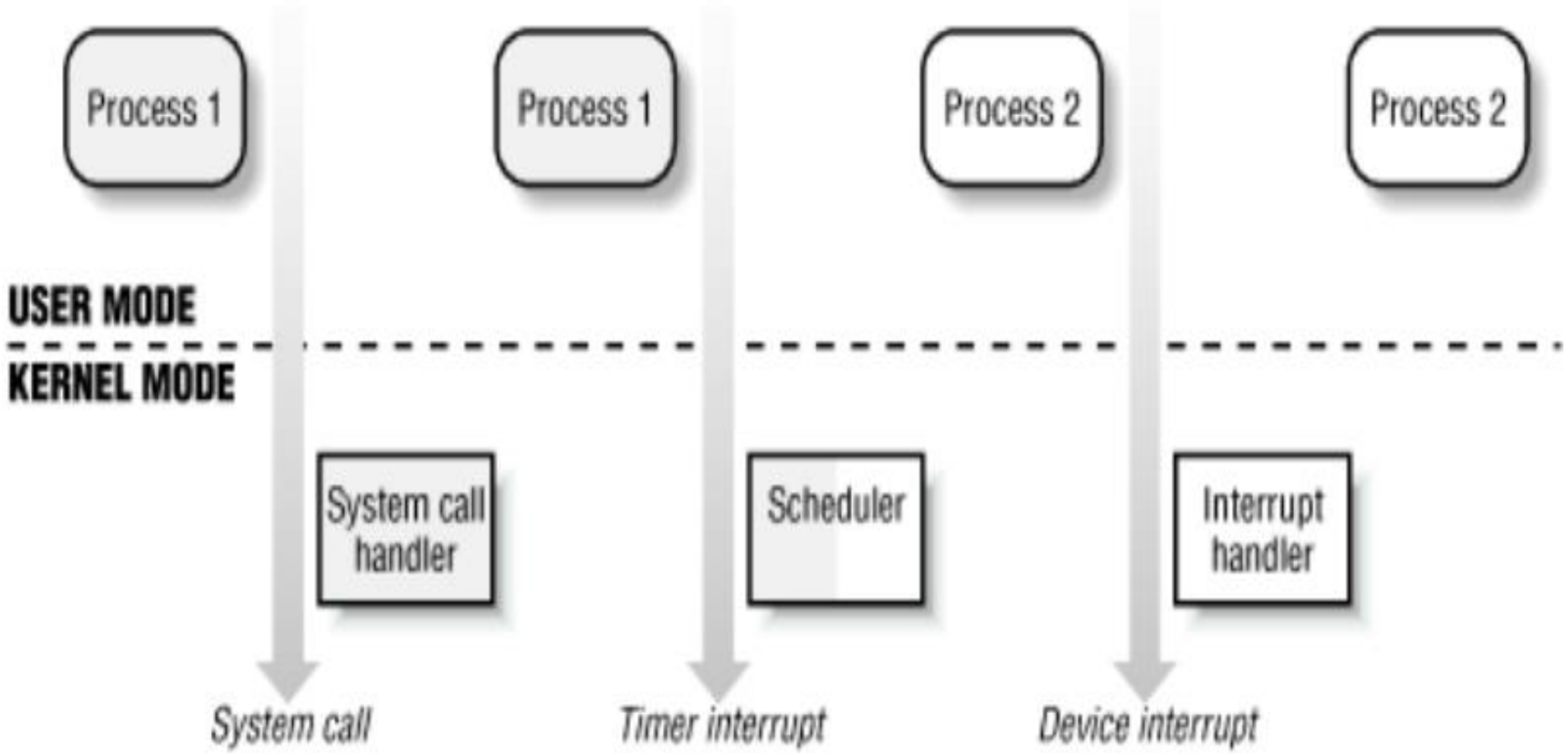
Process/Kernel Mode

- A CPU can run either in User Mode or in Kernel Mode.
- Some CPUs can have more than two execution states.
- Example - The Intel 80x86 microprocessors have four different execution states.
- When a program is executed in User Mode, it cannot directly access the kernel data structures or the kernel programs.
- When an application executes in Kernel Mode, however, these restrictions no longer apply.
- Each CPU model provides special instructions to switch from User Mode to Kernel Mode and vice versa.
- A program executes most of the time in User Mode and switches to Kernel Mode only when requesting a service provided by the kernel.
- When the kernel has satisfied the program's request, it puts the program back in User Mode.

- Processes are dynamic entities having 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.
- The kernel itself is not a process but a process manager.
- The process/kernel model assumes that processes that require a kernel service make use of specific programming constructs called system calls.
- Each system call sets up the group of parameters that identifies the process request and then executes the hardware-dependent CPU instruction to switch from User Mode to Kernel Mode.

- Unix systems include a few privileged processes called kernel threads with the following characteristics:
  - They run in Kernel Mode in the kernel address space.
  - They do not interact with users, and thus do not require terminal devices.
  - They are usually created during system startup and remain alive until the system is shut down.
- The figure shows, Process 1 in User Mode issues a system call, after which the process switches to Kernel Mode and the system call is serviced.
- Process 1 then resumes execution in User Mode until a timer interrupt occurs and the scheduler is activated in Kernel Mode.
- A process switch takes place, and Process 2 starts its execution in User Mode until a hardware device raises an interrupt.
- As a consequence of the interrupt, Process 2 switches to Kernel Mode and services the interrupt.

Transistions between User mode and Kernel mode



- Unix kernels do much more than handle system calls -
  - A process invokes a system call.
  - The CPU executing the process signals an exception, which is some unusual condition such as an invalid instruction. The kernel handles the exception on behalf of the process that caused it.
  - A peripheral device issues an interrupt signal to the CPU to notify it of an event such as a request for attention, a status change, or the completion of an I/O operation. Each interrupt signal is dealt by a kernel program called an interrupt handler. Since peripheral devices operate asynchronously with respect to the CPU, interrupts occur at unpredictable times.
  - A kernel thread is executed; since it runs in Kernel Mode, the corresponding program must be considered part of the kernel, albeit encapsulated in a process.

Process Implementation

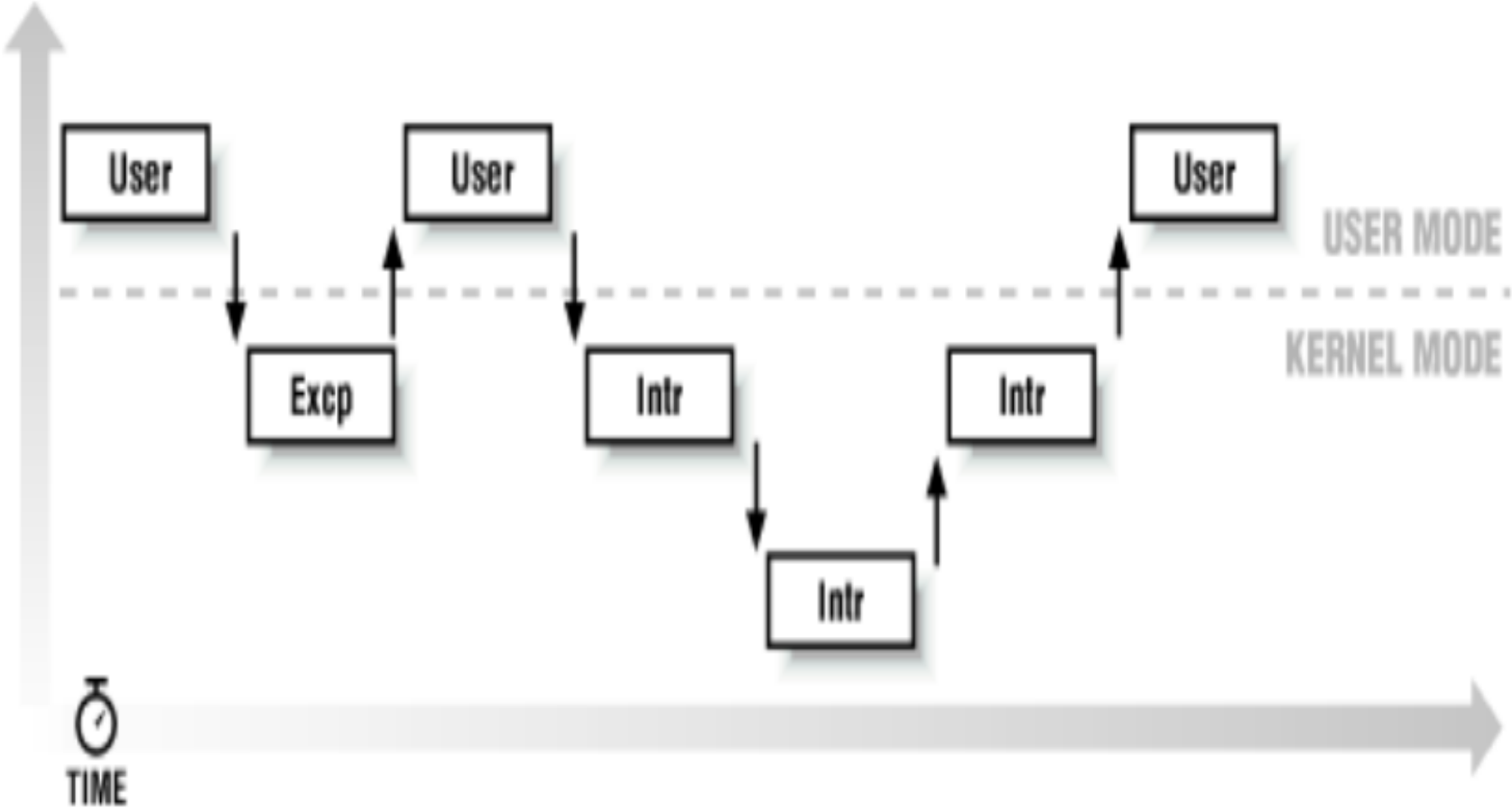
- To let the kernel manage processes, each process is represented by a process descriptor that includes information about the current state of the process.
- When the kernel stops the execution of a process, it saves the current contents of several processor registers in the process descriptor. These 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) containing information about the CPU state
  - The memory management registers used to keep track of the RAM accessed by the process

- When the kernel decides to resume executing a process, it uses the proper process descriptor fields to load the CPU registers.
- Since the stored value of the program counter points to the instruction following the last instruction executed, the process resumes execution from where it was stopped.
- When a process is not executing on the CPU, it is waiting for some event.

Reentrant Kernel

- All Unix kernels are reentrant.
- One way to provide reentrancy is to write functions so that they modify only local variables and do not alter global data structures. Such functions are called reentrant functions.
- If a hardware interrupt occurs, a reentrant kernel is able to suspend the current running process even if that process is in Kernel Mode.
- A kernel control path denotes the sequence of instructions executed by the kernel to handle a system call, an exception, or an interrupt.
- The CPU executes a kernel control path sequentially from the first instruction to the last.

- The following events occur when the CPU interleaves the kernel control paths:
  - 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. In this case, the two control paths are executed on behalf of two different processes.
  - The CPU detects an exception.
  - A hardware interrupt occurs while the CPU is running a kernel control path with the interrupts enabled.


User
User
User
USER MODE
KERNEL MODE
Excp
Intr
Intr
Intr
TIME

Process Address space

- Each process runs in its private address space.
- A process running in User Mode refers to private stack, data, and code areas.
- When running in Kernel Mode, the process addresses the kernel data and code area and makes use of another stack.
- There are times when part of the address space is shared among processes, i.e., explicitly requested by the processes or done automatically by the kernel to reduce memory usage.
- Processes can also share parts of their address space as a kind of interprocess communication.

Synchronization and Critical Regions

- Implementing a reentrant kernel requires the use of synchronization: if a kernel control path is suspended while acting on a kernel data structure, no other kernel control path will be allowed to act on the same data structure unless it has been reset to a consistent state.
- When the outcome of some computation depends on how two or more processes are scheduled, the code is incorrect: there is a race condition.
- Any section of code that should be finished by each process that begins it before another process can enter it is called a critical region.
- These problems occur not only among kernel control paths but also among processes sharing common data.
- Several synchronization techniques have been adopted.
- The following types will concentrate on how to synchronize kernel control paths.

- Non preemptive kernels - Nonpreemptability is ineffective in multiprocessor systems, since two kernel control paths running on different CPUs could concurrently access the same data structure.
- Interrupt disabling - Another synchronization mechanism consists of disabling all hardware interrupts before entering a critical region and reenabling them right after leaving it.
- Semaphores - A semaphore is simply a counter associated with a data structure; the semaphore is checked by all kernel threads before they try to access the data structure.
- 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 repeatedly, executing a tight instruction loop until the lock becomes open.

Signals and Interprocess communication

- Unix signals provide a mechanism for notifying processes of system events. Each event has its own signal number, which is usually referred to by a symbolic constant such as SIGTERM.
- There are two kinds of system events:
  - Asynchronous notifications - For instance, a user can send the interrupt signal SIGTERM to a foreground process by pressing the interrupt keycode (usually, CTRL-C) at the terminal.
  - Synchronous errors or exceptions - For instance, the kernel sends the signal SIGSEGV to a process when it accesses a memory location at an illegal address.

- In general, a process may react to a signal reception in two possible ways:
  - Ignore the signal.
  - Asynchronously execute a specified procedure (the signal handler).
- If the process does not specify one of these alternatives, the kernel performs a default action that depends on the signal number. The five possible default actions are:
  - Terminate the process.
  - Write the execution context and the contents of the address space in a file (core dump) and terminate the process.
  - Ignore the signal.
  - Suspend the process.
  - Resume the process's execution, if it was stopped.

Process management

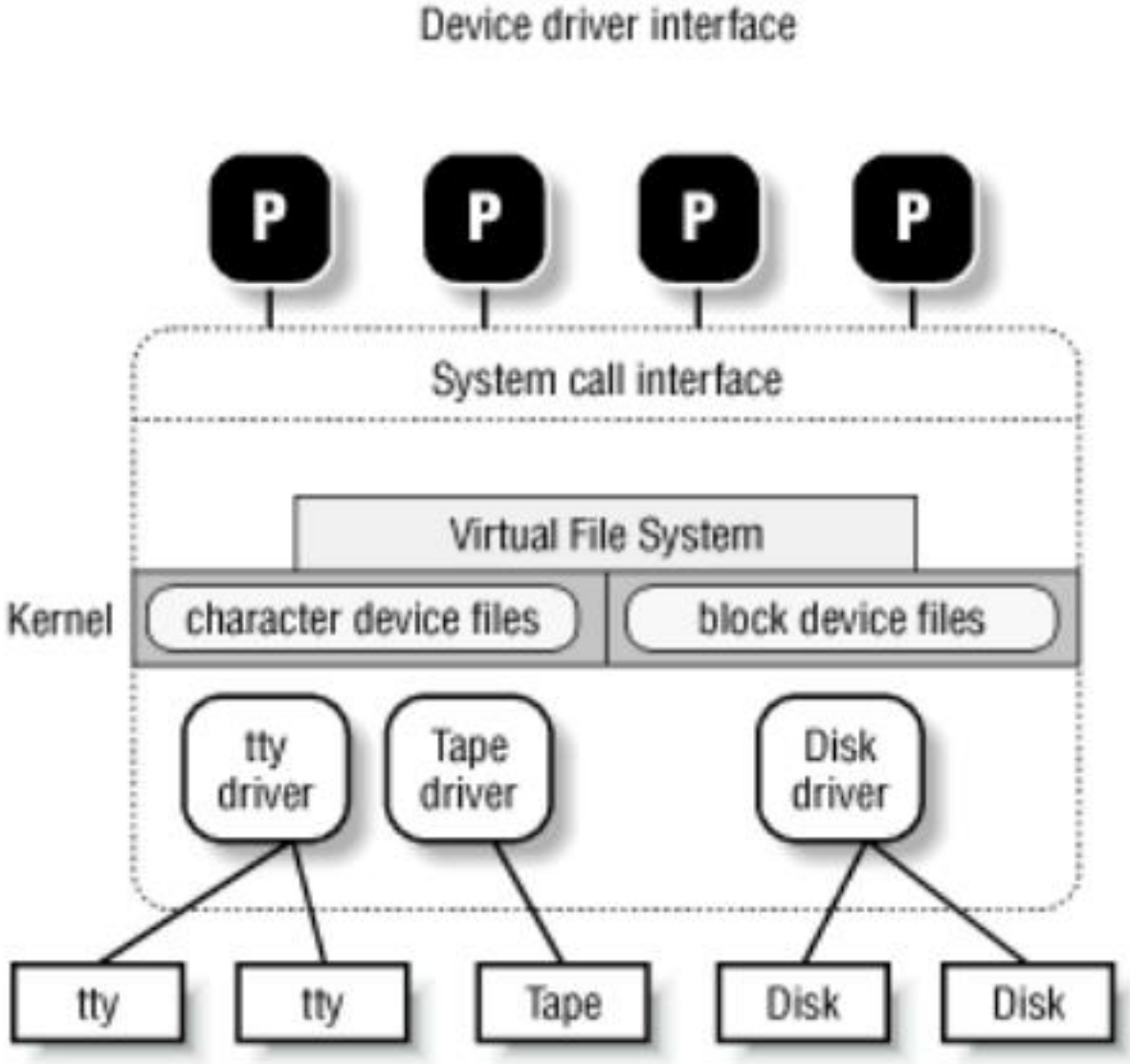
- The fork( ) and exit( ) system calls are used respectively to create a new process and to terminate it.
- While an exec( )-like system call is invoked to load a new program.
- The process that invokes a fork( ) is the parent while the new process is its child .
- The exit( ) system call terminates a process.
- The wait( ) system call allows a process to wait until one of its children terminates; it returns the process ID (PID) of the terminated child.

Kernel Memory Allocator

- The Kernel Memory Allocator (KMA) is a subsystem that tries to satisfy the requests for memory areas from all parts of the system.
- A good KMA should have the following features:
  - It must be fast. Actually, this is the most crucial attribute, since it is invoked by all kernel subsystems (including the interrupt handlers).
  - It should minimize the amount of wasted memory.
  - It should try to reduce the memory fragmentation problem.
  - It should be able to cooperate with the other memory management subsystems in order to borrow and release page frames from them.

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, such as hard disks, keyboards, mouses, monitors, network interfaces, and devices connected to a SCSI bus.
- Each driver interacts with the remaining part of the kernel (even with other drivers) through a specific interface.
- This approach has the following advantages:
  - Device-specific code can be encapsulated in a specific module.
  - Vendors can add new devices without knowing the kernel source code: only the interface specifications must be known.
  - The kernel deals with all devices in a uniform way and accesses them through the same interface.
  - It is possible to write a device driver as a module that can be dynamically loaded in the kernel without requiring the system to be rebooted. It is also possible to dynamically unload a module that is no longer needed, thus minimizing the size of the kernel image stored in RAM.


Device driver interface
P
P
P
P
System call interface
Virtual File System
Kernel
character device files
block device files
tty driver
Tape driver
Disk driver
tty
tty
Tape
Disk
Disk