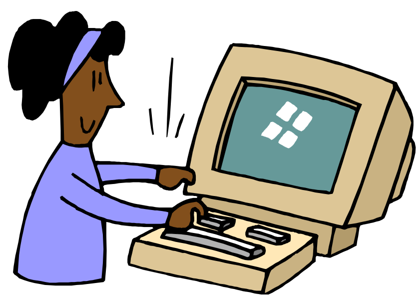
Introduction to Operating Systems

- Introduction
- Processes and Threads
- Memory Management
- File Systems
- Input / Output
- Deadlocks

Computer Basic Elements

Processor

I/O Modules



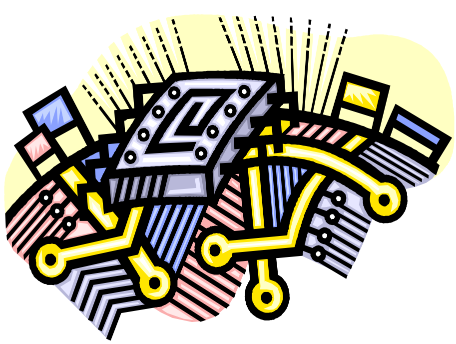
Main Memory

System Bus

Processor

Controls the operation of the computer

Performs the data processing functions

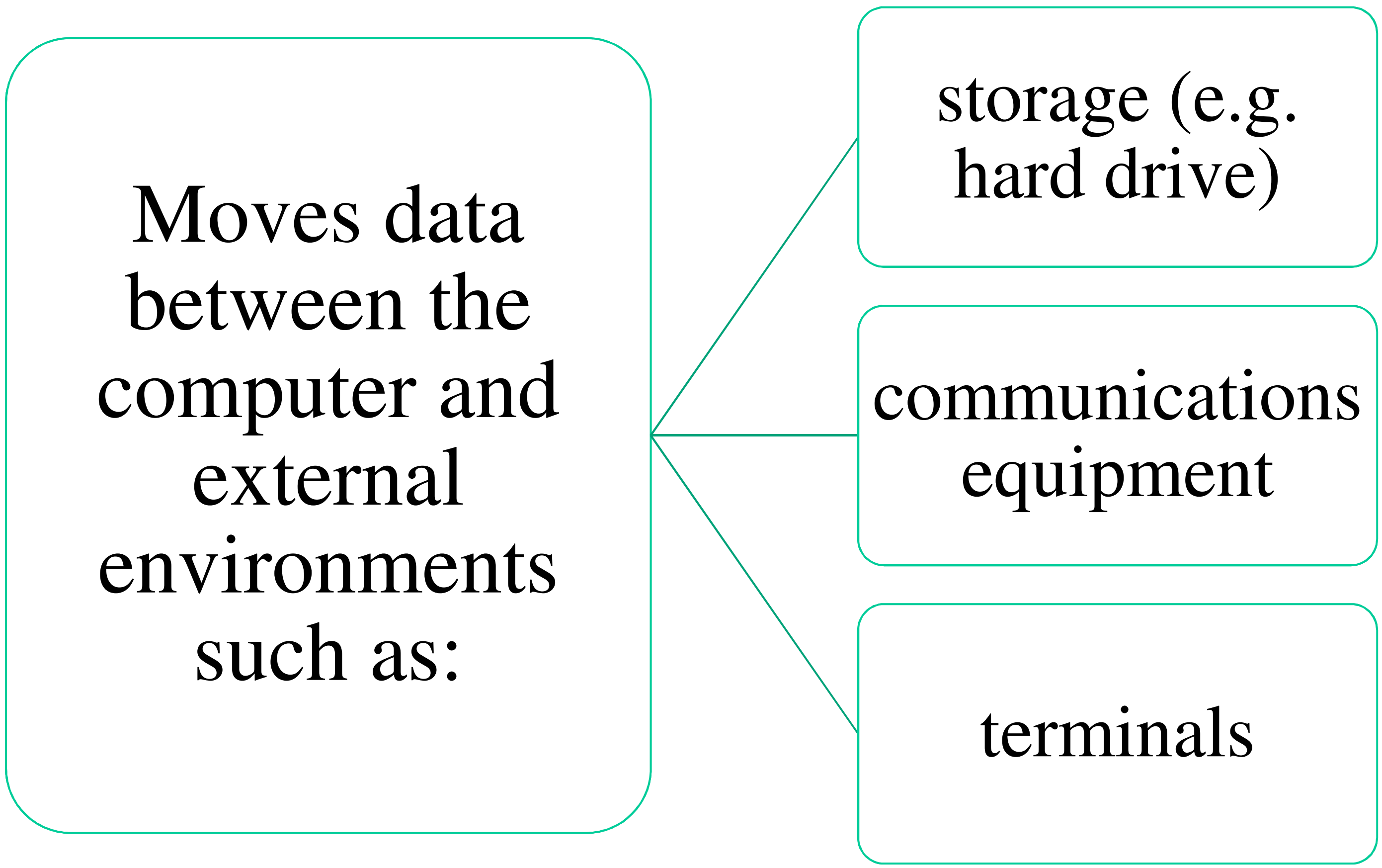


Referred to as the *Central Processing Unit* (CPU)

Main Memory

- Volatile
- Contents of the memory is lost when the computer is shut down
- Referred to as real memory or primary memory

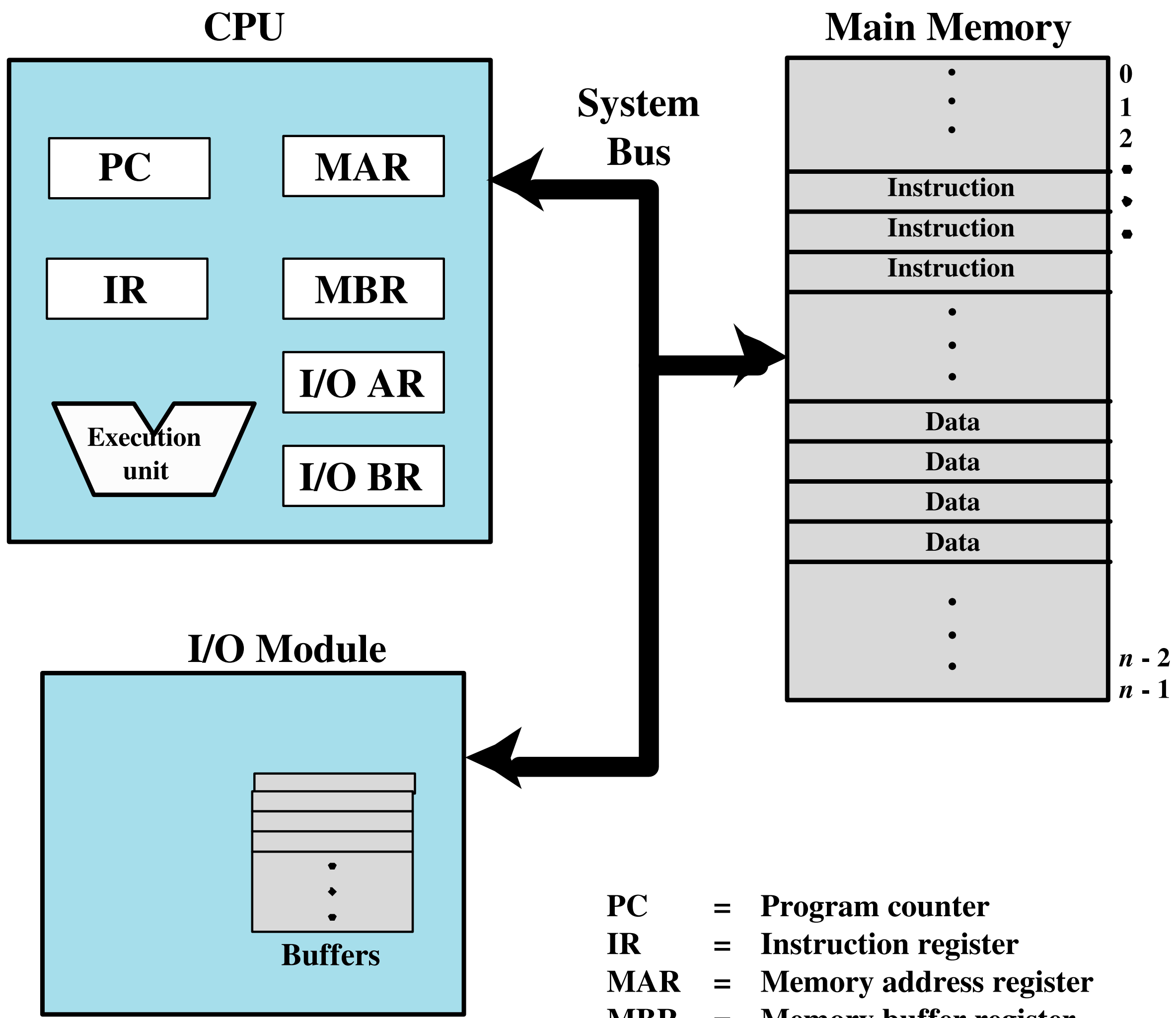
I/O Modules


Moves data between the computer and external environments such as:
storage (e.g. hard drive)
communications equipment
terminals

System Bus

- Provides for communication among processors, main memory, and I/O modules

Computer Components



PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

Figure 1.1 Computer Components: Top-Level View

Processor Registers

- User-visible registers
  - *Enable programmer to minimize main-memory references by optimizing register use*
- Control and status registers
  - *Used by processor to control operating of the processor*
  - *Used by privileged operating-system routines to control the execution of programs*

User Visiable Registers

- May be referenced by machine language
- Available to all programs - application programs and system programs
- Types of registers
    - *Data*
    - *Address*
        - *Index*
        - *Segment pointer*
        - *Stack pointer*

Control and Status Registers

- Program Counter (PC)
  - *Contains the address of an instruction to be fetched*
- Instruction Register (IR)
  - *Contains the instruction most recently fetched*
- Program Status Word (PSW)
  - *Condition codes*
  - *Interrupt enable/disable*
  - *Supervisor/user mode*

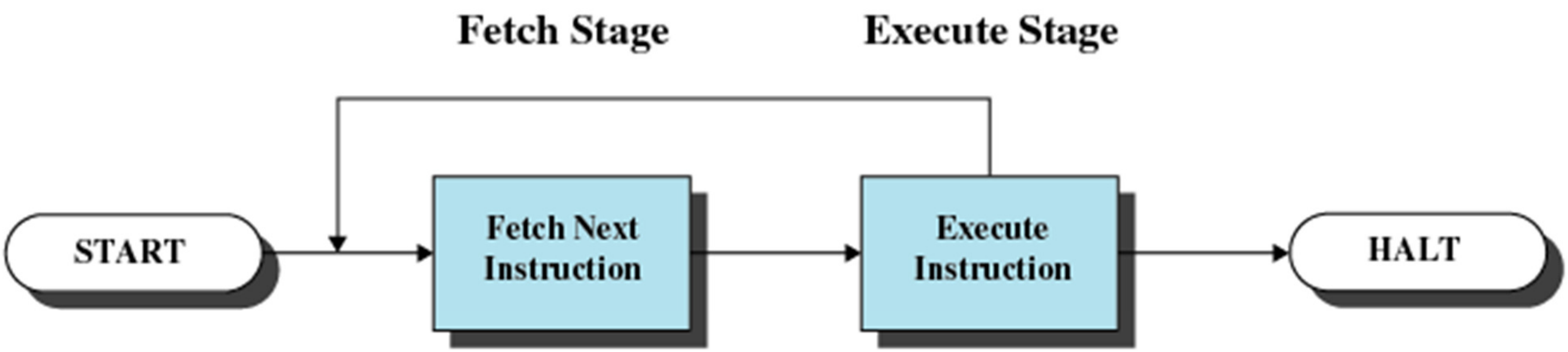


Figure 1.2 Basic Instruction Cycle

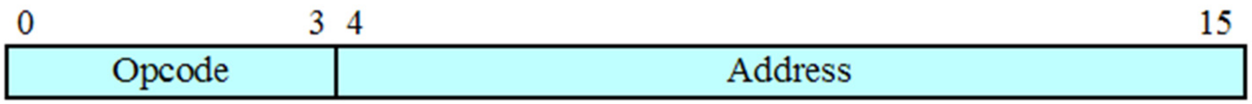
Instruction Fetch and Execute

- The processor fetches the instruction from memory
- Program counter (PC) holds address of the instruction to be fetched next
    - *PC is incremented after each fetch*

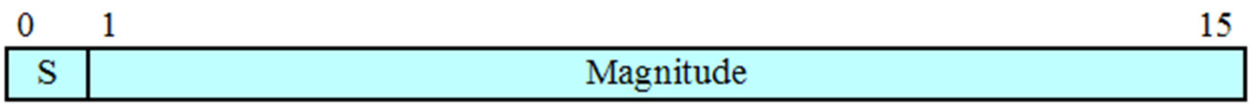
Instruction Register (IR)

- Fetched instruction is placed in the instruction register
- Categories
  - *Processor-memory*
    - *Transfer data between processor and memory*
  - *Processor-I/O*
    - *Data transferred to or from a peripheral device*
  - *Data processing*
    - *Arithmetic or logic operation on data*
  - *Control*
    - *Alter sequence of execution*

Characteristics of a Hypothetical Machine



(a) Instruction format



(b) Integer format

Program Counter (PC) = Address of instruction
Instruction Register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(c) Internal CPU registers

0001 = Load AC from Memory
0010 = Store AC to Memory
0101 = Add to AC from Memory

(d) Partial list of opcodes

Figure 1.3 Characteristics of a Hypothetical Machine

Example of Program Execution

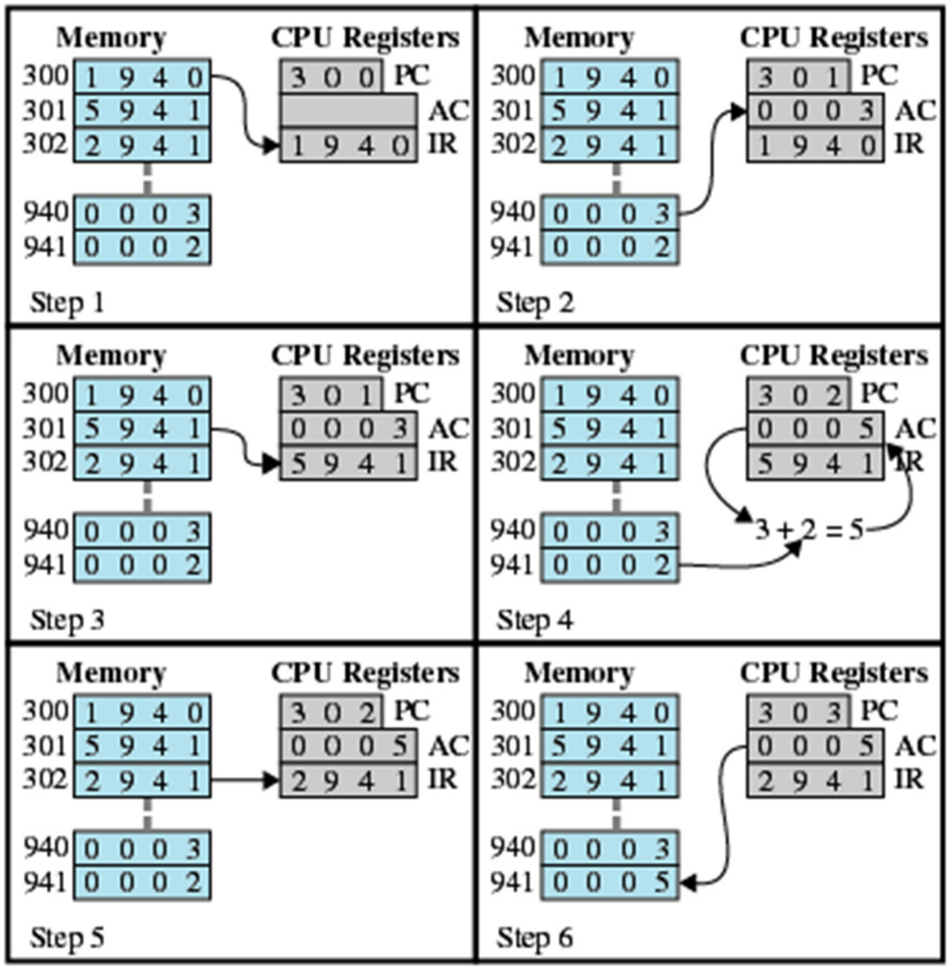


Figure 1.4 Example of Program Execution (contents of memory and registers in hexadecimal)

Interrupts

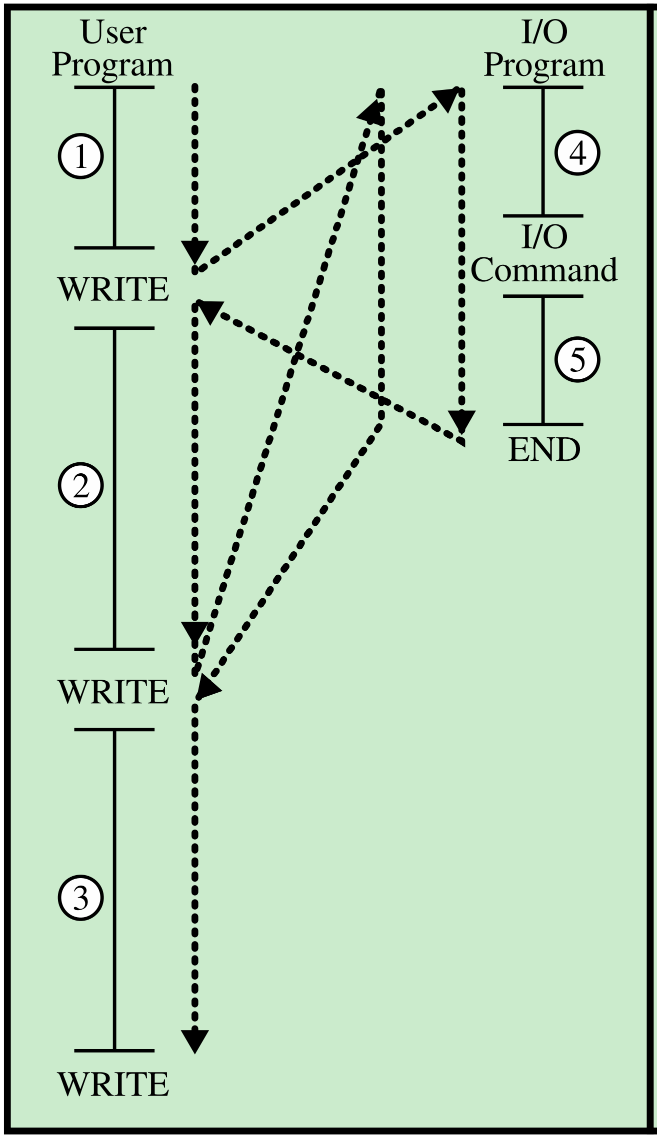
- Interrupt the normal sequencing of the processor
- Provided to improve processor utilization
  - *most I/O devices are slower than the processor*
  - *processor must pause to wait for device*
  - *wasteful use of the processor*

Classes of Interrupts

- **Program**
  - *Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a user's allowed memory space.*
- **Timer**
  - *Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.*
- **I/O**
  - *Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.*
- **Hardware failure**
  - *Generated by a failure, such as power failure or memory parity error.*

Figure 1.5a

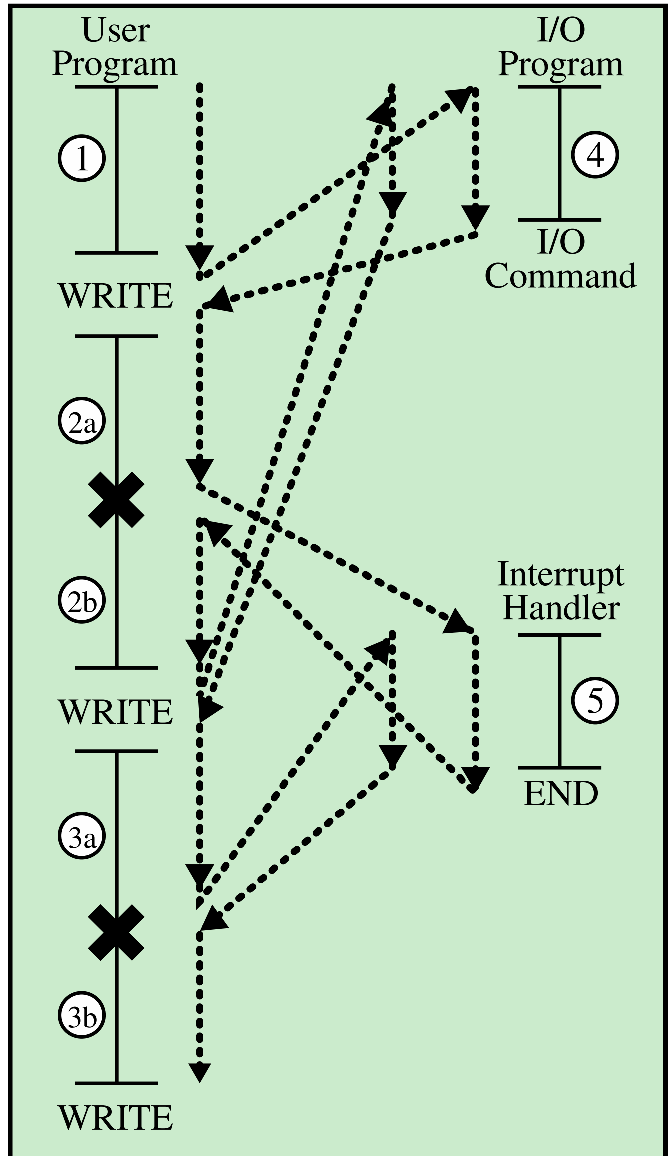
Flow of Control Without Interrupts



(a) No interrupts

Figure 1.5b

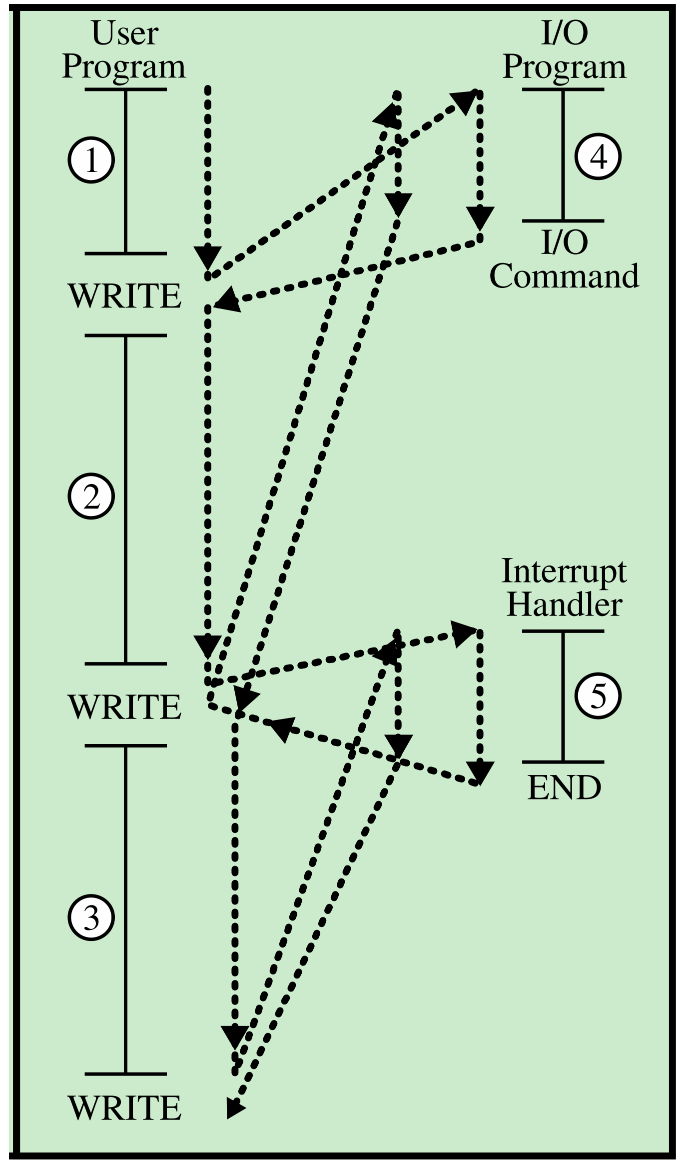
Short I/O Wait



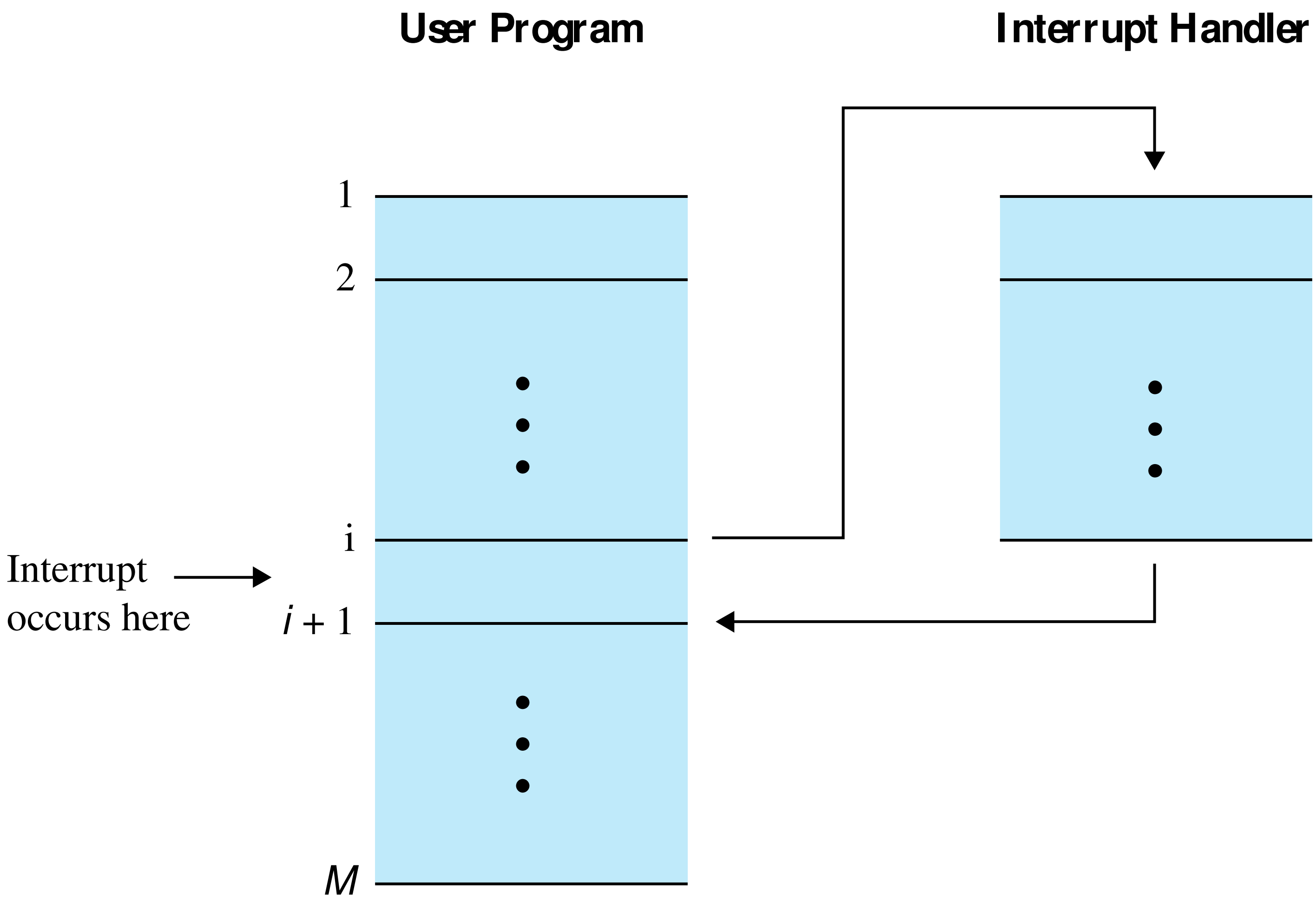
(b) Interrupts; short I/O wait

Figure 1.5c

Long I/O Wait



(c) Interrupts; long I/O wait



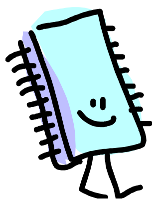


Figure 1.6 Transfer of Control via Interrupts

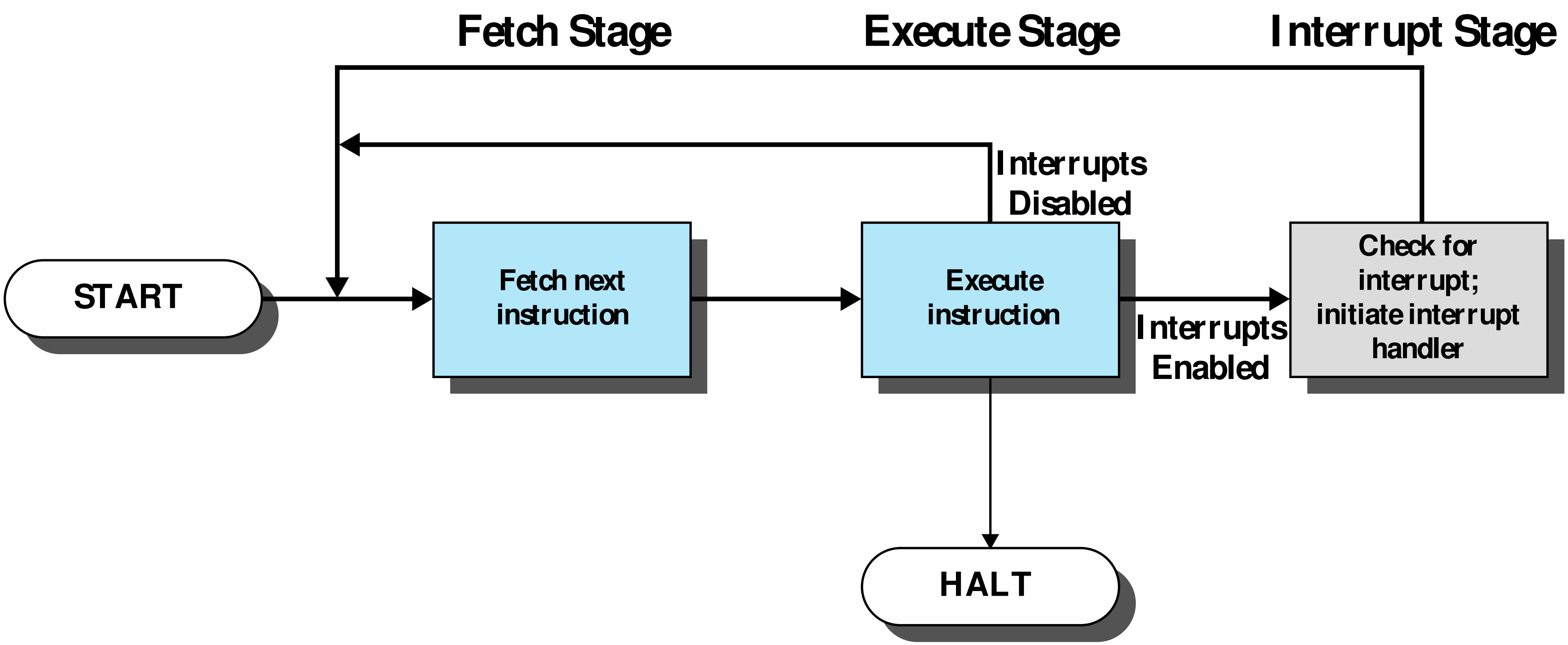


Figure 1.7 Instruction Cycle with Interrupts

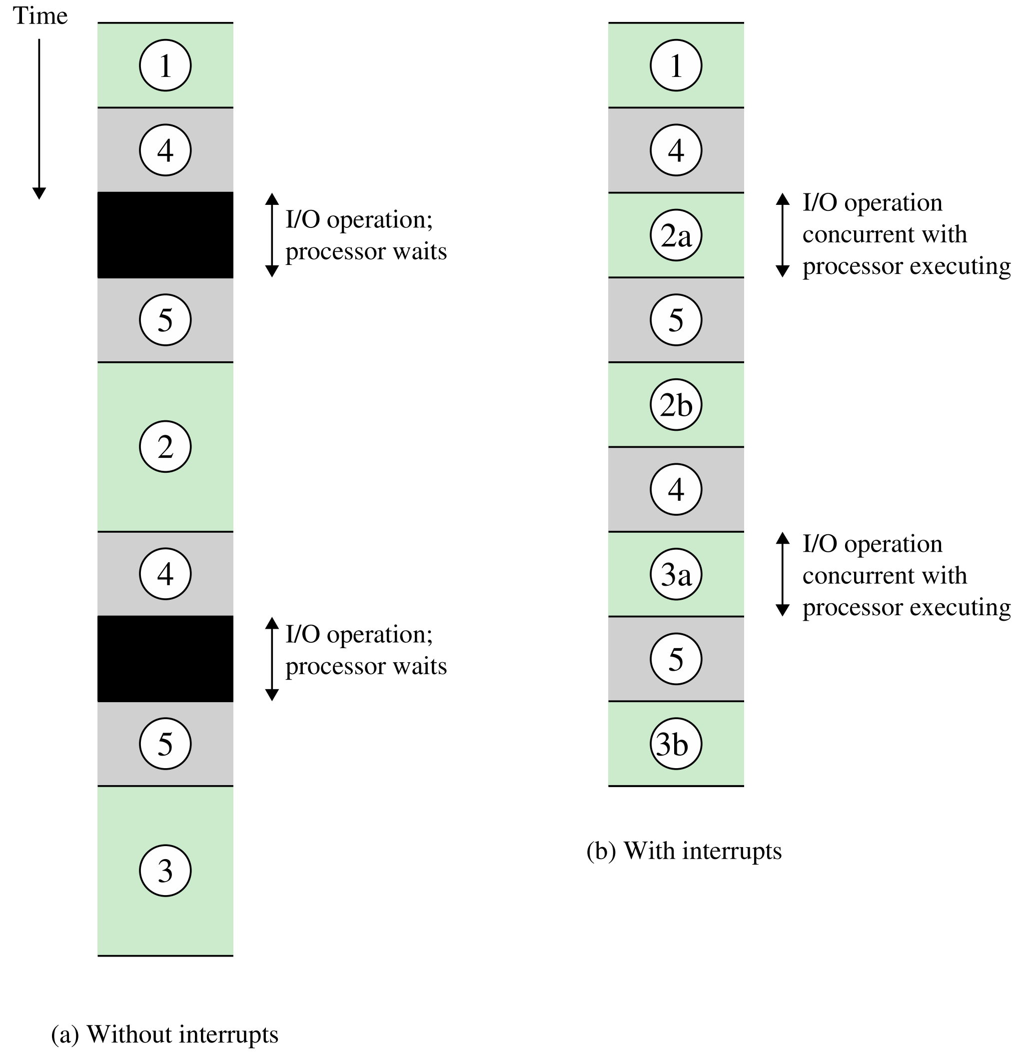


Figure 1.8 Program Timing: Short I/O Wait

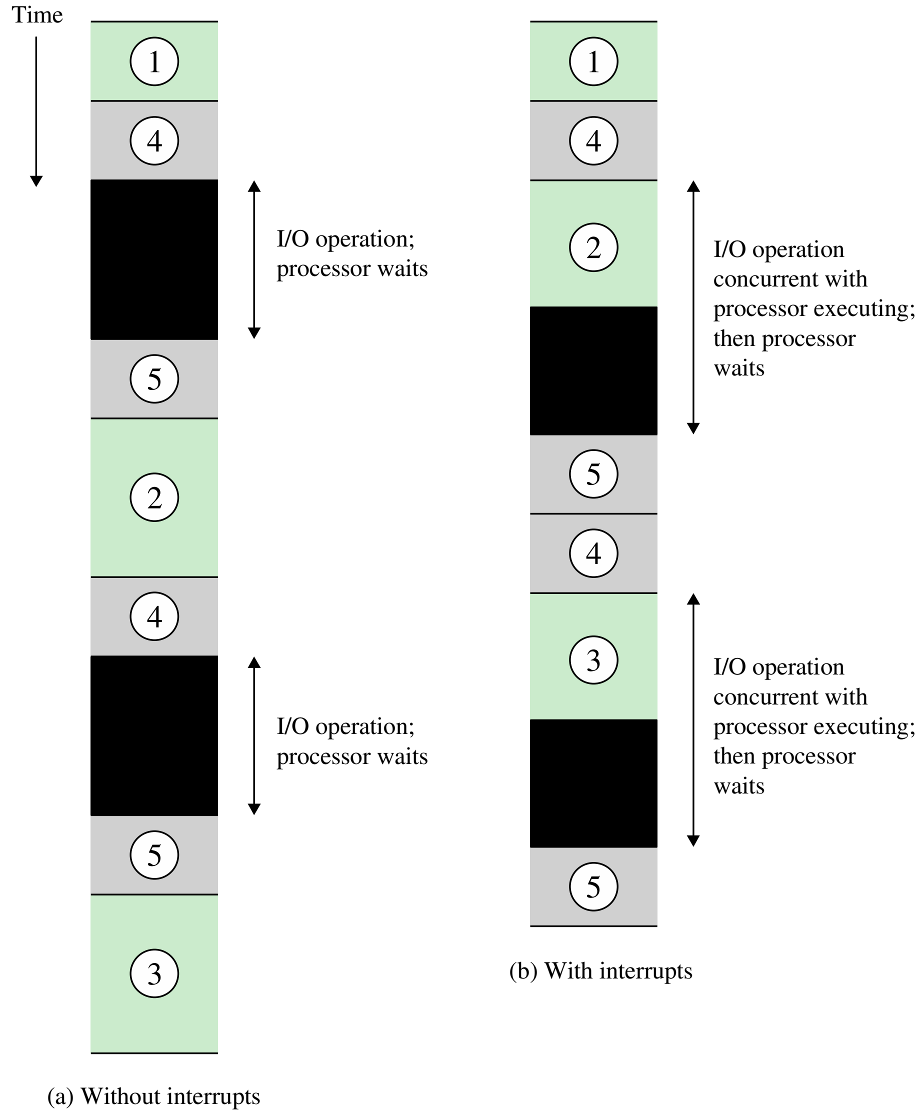


Figure 1.9 Program Timing: Long I/O Wait

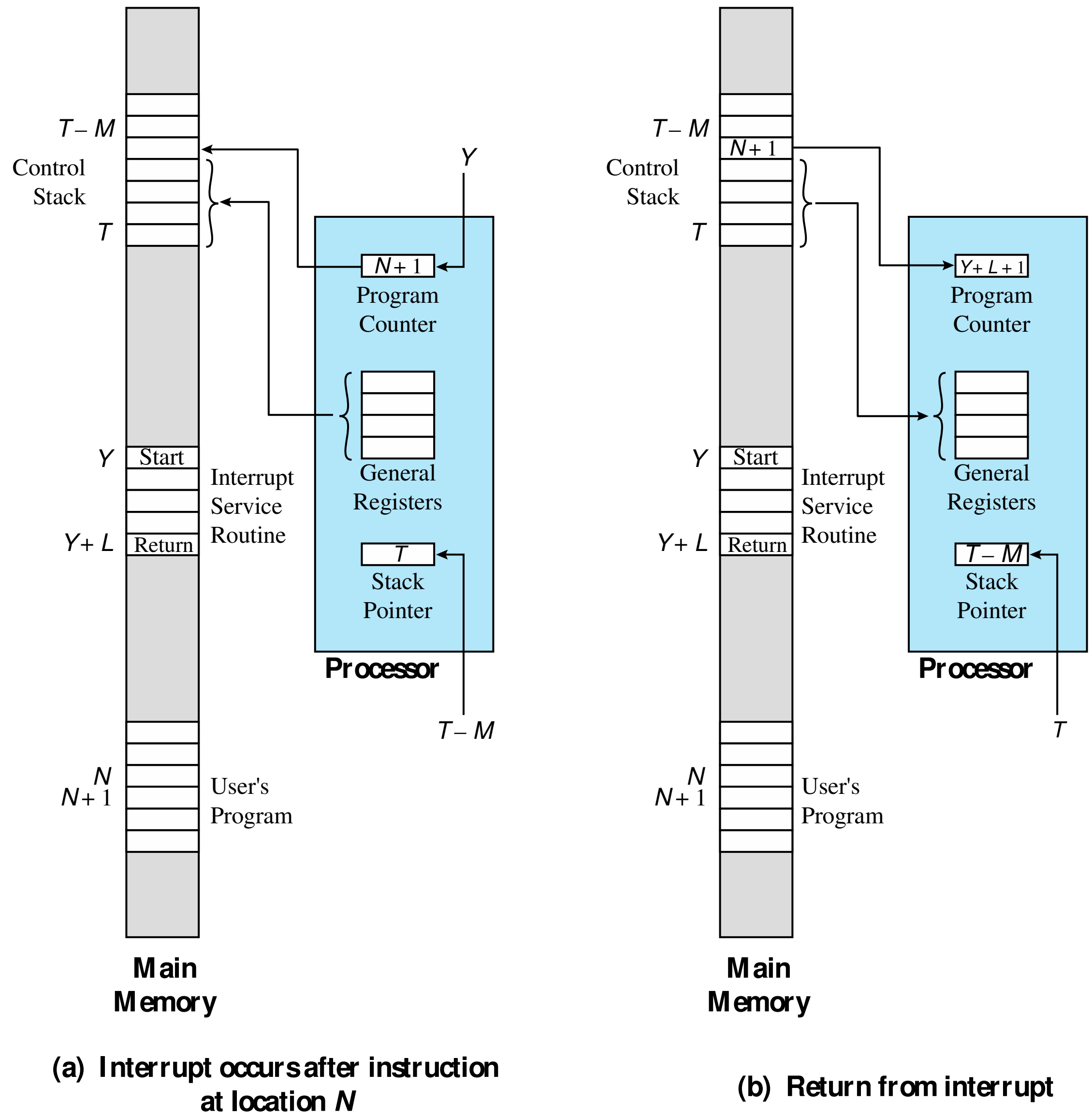
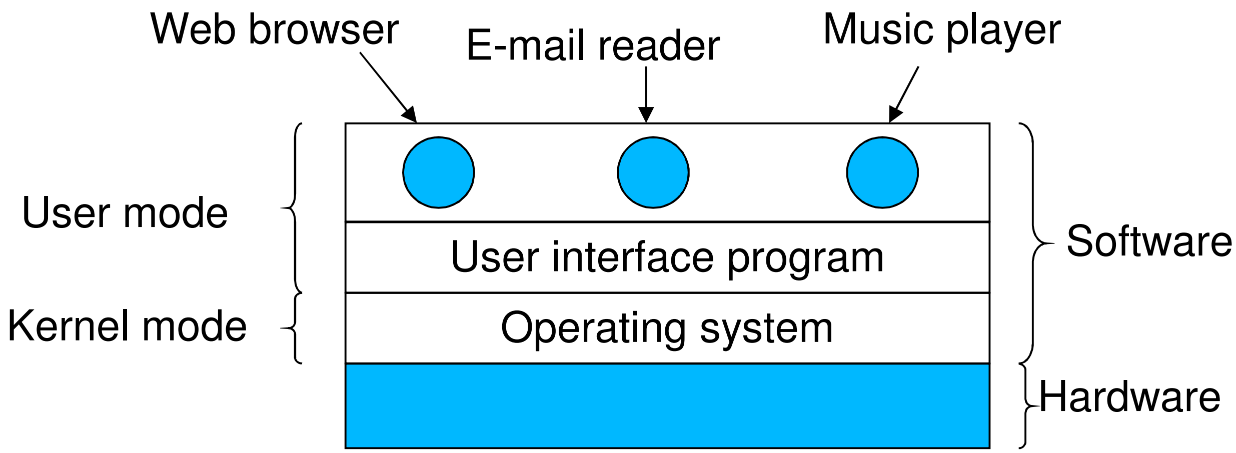


Figure 1.11 Changes in Memory and Registers for an Interrupt

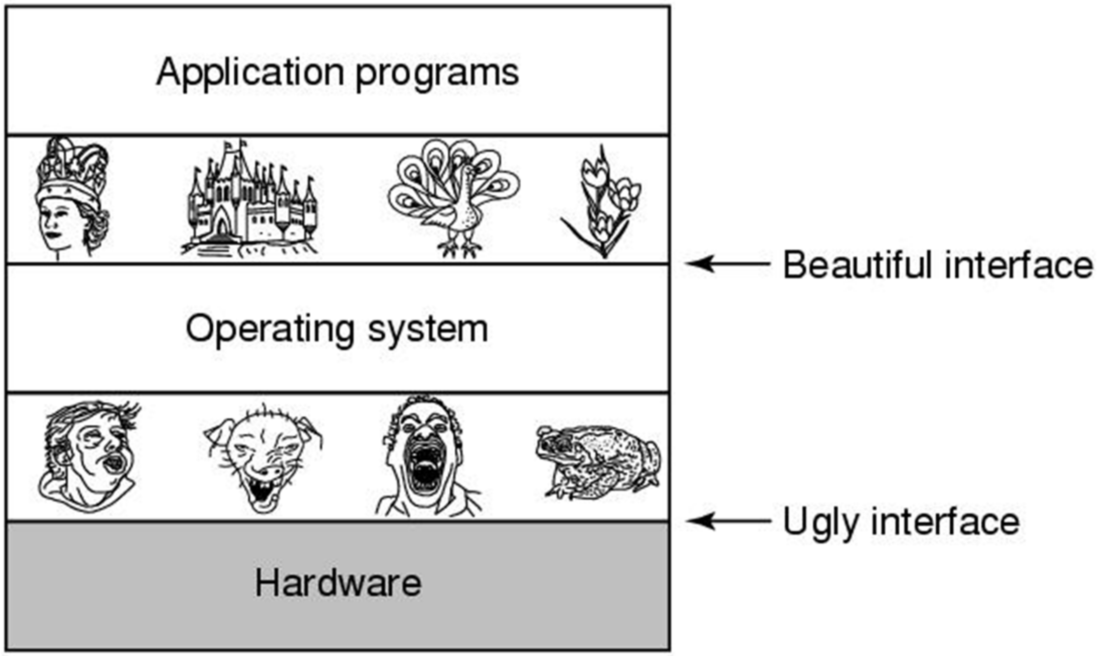
What is an operating system?



- A software layer to provide user programs with a better, simpler, cleaner model of the computer and to handle managing all the hardware resources.
- Most computers has two modes of operation: **kernel mode** and **user mode**.
  - *The operating system runs in* ***kernel mode*** *in which it has complete access to all the hardware and can execute any instruction the machine capable of executing.*
  - *The rest of the software runs in* ***user mode****, in which only a subset of the machine instruction available.*

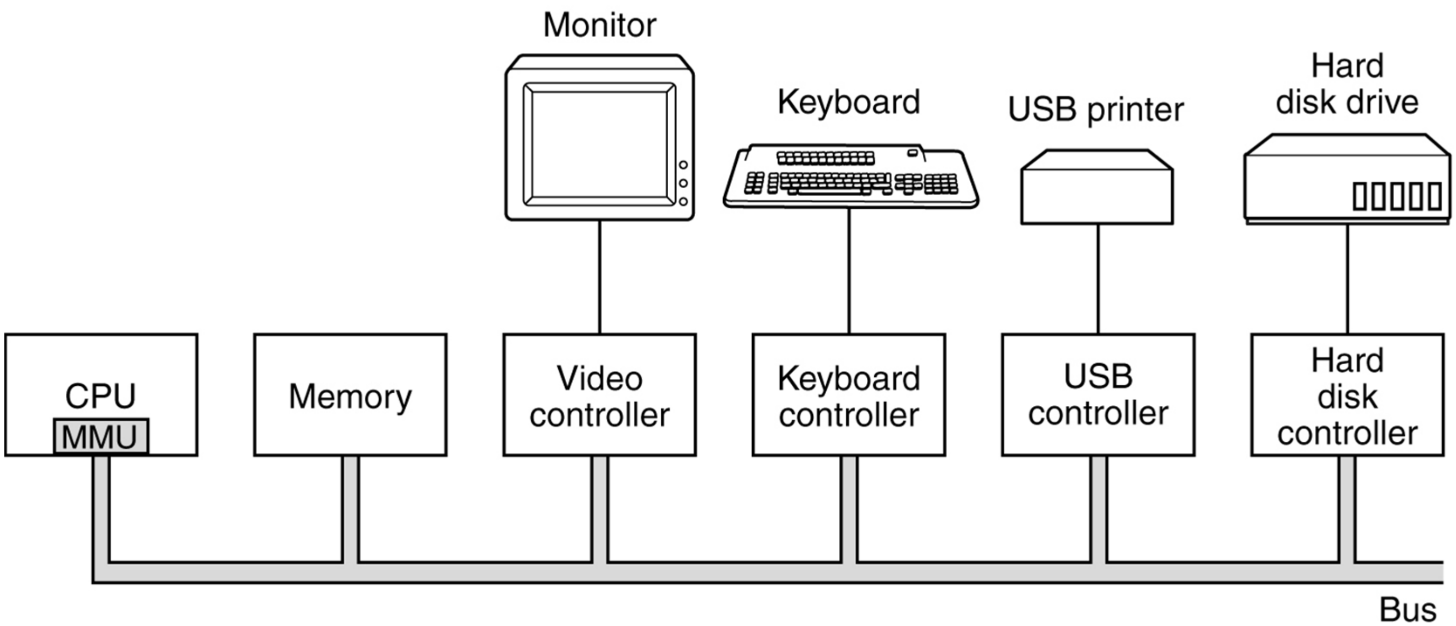
What is an operating system?

- A **virtual machine** (or extended) that hides the messy details which must be performed and is easier to program than the raw hardware.



What is an operating system?

- A **resource manager** that manages and allocates resources for the user programs.



The OS and Hardware

- An OS mediates programs' access to hardware resources
  - *Computation (CPU)*
    - *Process Management, CPU Scheduling, Synchronization*
  - *Volatile storage (memory)*
    - *Memory Management*
  - *Persistent storage (disks)*
    - *File Systems*
  - *Network communications (TCP/IP stacks, ethernet cards, etc.)*
    - *Communication Subsystem – mostly covered in networking course*

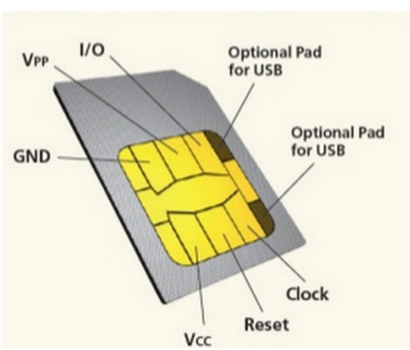
Why bother with an OS?

- Application benefits
  - *programming simplicity*
    - *see high-level abstractions (files) instead of low-level hardware details (device registers)*
  - *portability (across machine configurations or architectures)*
    - *device independence: 3Com card or Intel card?*

- User benefits
  - *safety*
    - *program "sees" own virtual machine, thinks it owns computer*
    - *OS protects programs from each other*
    - *OS fairly multiplexes resources across programs*
  - *efficiency (cost and speed)*
    - *share one computer across many users*
    - *concurrent execution of multiple programs*

The Operating System Zoo

- Mainframe operating systems
- Server operating systems
- Multiprocessor operating systems
- Personal computer operating systems
- Handheld operating systems
- Embedded operating systems
- Sensor node operating systems
- Real-time operating systems
- Smart card operating systems
- …



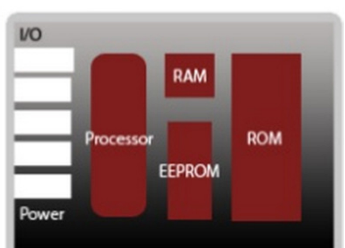


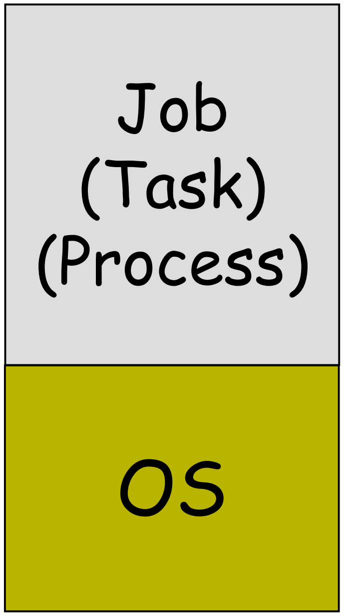
Figure 1: The smart chip in a card

OS History

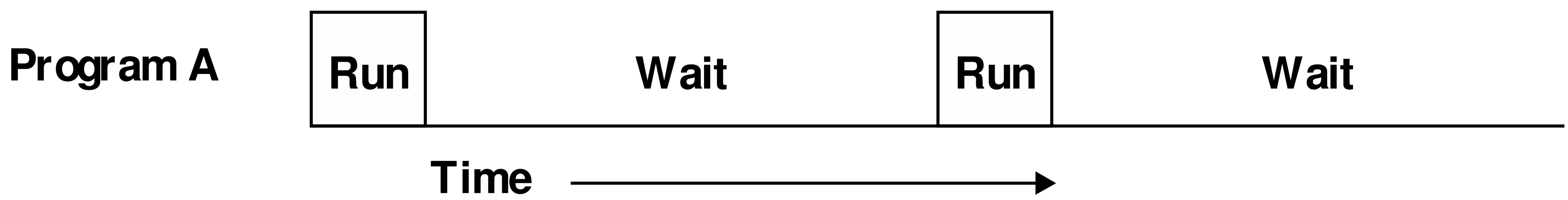
- **In the very beginning…**
  - *OS (not actually an OS) was just a library of code that you linked into your program; programs were loaded entirely into memory, and executed*
  - *What you do in your microprocessors course!*

Single-Tasking Systems (MS-DOS)

- And then came **single-tasking systems**
  - *OS was stored in a portion of primary memory*
  - *OS loaded the next job into memory from the disk*
    - *Job (task, process) gets executed until termination*
    - *repeat…*
- Problem: CPU is idle when a program interacts with a peripheral (I/O) during execution



Single-Tasking Systems



(a) Uniprogramming

The processor spends a certain amount of time executing, until it reaches an I/O instruction; it must then wait until that I/O instruction concludes before proceeding

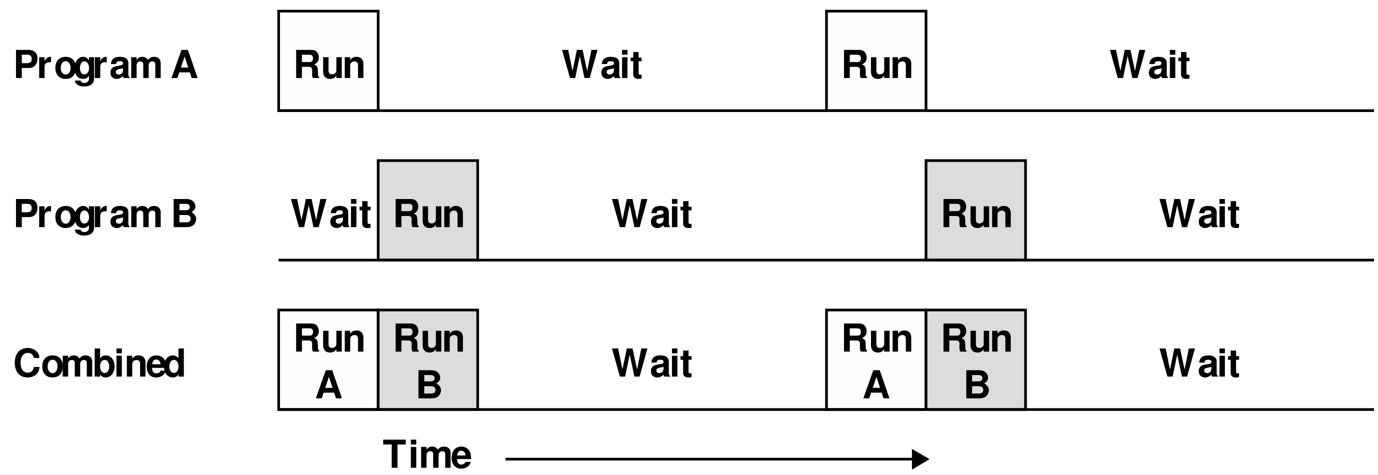
Multi-Tasking Systems

- To increase system utilization, multi-tasking OS's were invented
  - *keeps multiple runnable jobs loaded in memory at once*
  - *overlaps I/O of a job with computing of another*
    - *while one job waits for I/O completion, OS runs instructions from another job*
  - *to benefit, need asynchronous I/O devices*
    - *need some way to know when devices are done*

      interrupts

      polling
  - *goal: optimize system throughput*
    - *perhaps at the cost of response time…*

Multiprogramming

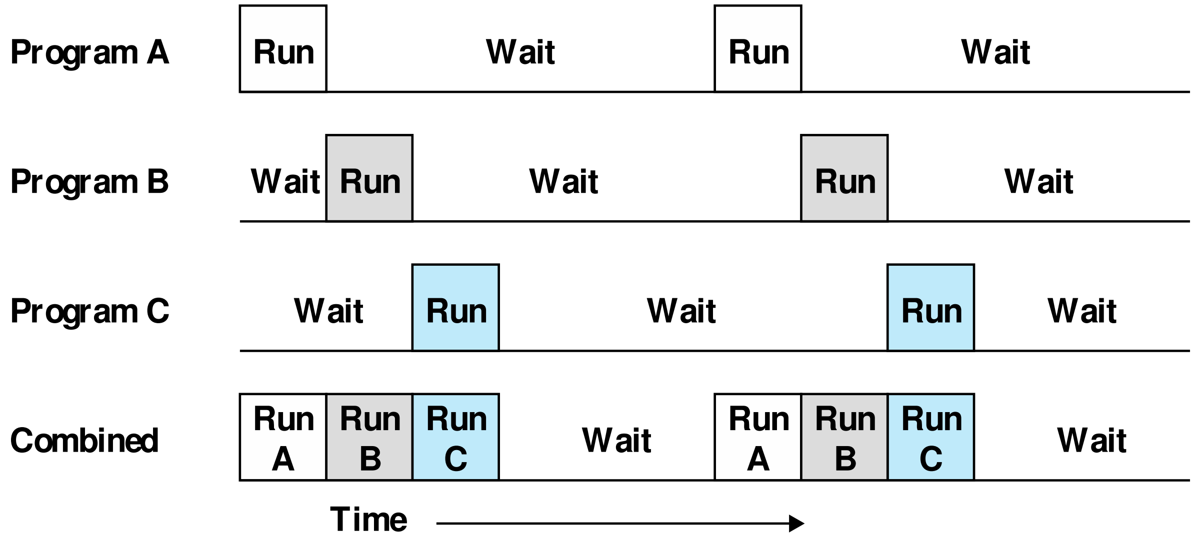


(b) Multiprogramming with two programs

There must be enough memory to hold the OS (resident monitor) and one user program

When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O

Multiprogramming



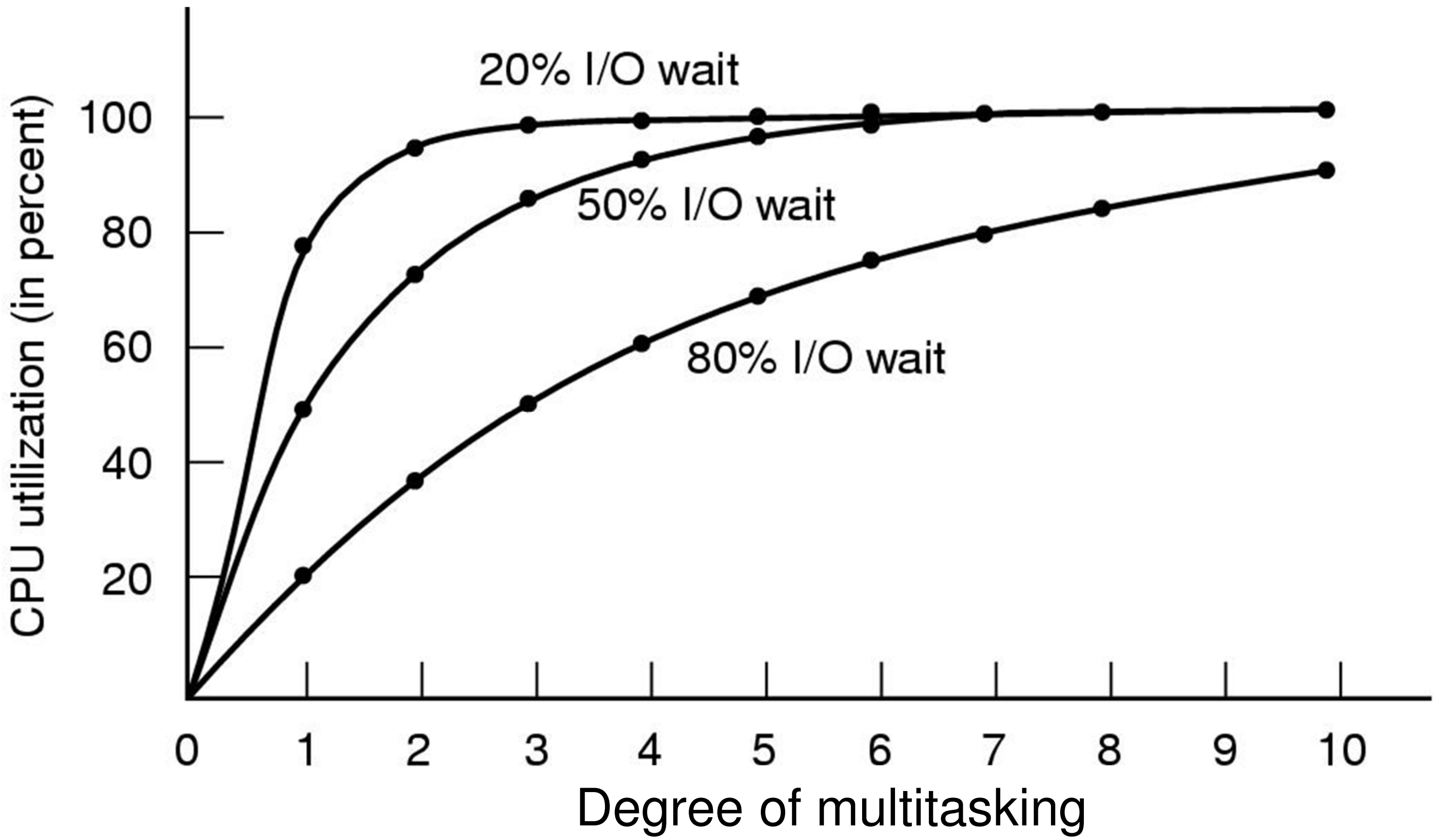
(c) Multiprogramming with three programs

Multiprogramming

also known as multitasking

memory is expanded to hold three, four, or more programs and switch among all of them

Why Multi-tasking?



- CPU utilization as a function of number of processes in memory

Effects on Resource Utilization

	Uniprogramming	Multiprogramming
Processor use	20%	40%
Memory use	33%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput	6 jobs/hr	12 jobs/hr
Mean response time	18 min	10 min

Timesharing

- To support interactive use, create a timesharing OS:
  - *multiple terminals into one machine*
  - *each user has illusion of entire machine to him/herself*
  - *optimize response time, perhaps at the cost of throughput*

- Time slicing
  - *divide CPU equally among the users*
  - *if job is truly interactive (e.g. editor), then can jump between programs and users faster than users can generate load*
  - *permits users to interactively view, edit, debug running programs (why does this matter?)*

- MIT Multics system (mid-1960's) was the first large timeshared system
  - *nearly all OS concepts can be traced back to Multics*

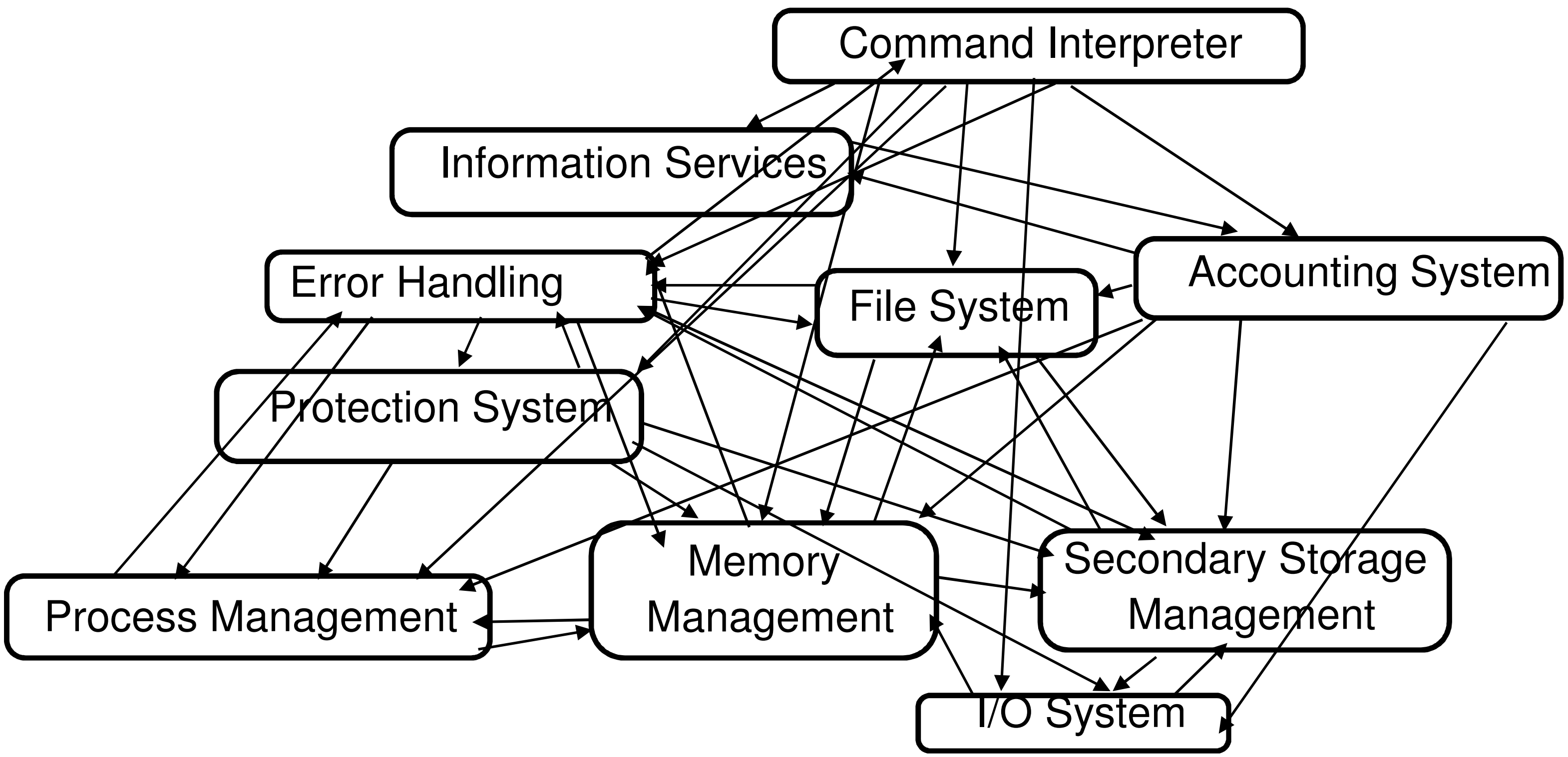
Major OS components

- Processes manager
- Memory manager
- I/O manager
    - *File systems (Abstractions of disks)*

- protection
- accounting
- shells (command interpreter, or OS UI)

OS Structure

- It's not always clear how to switch OS modules together:

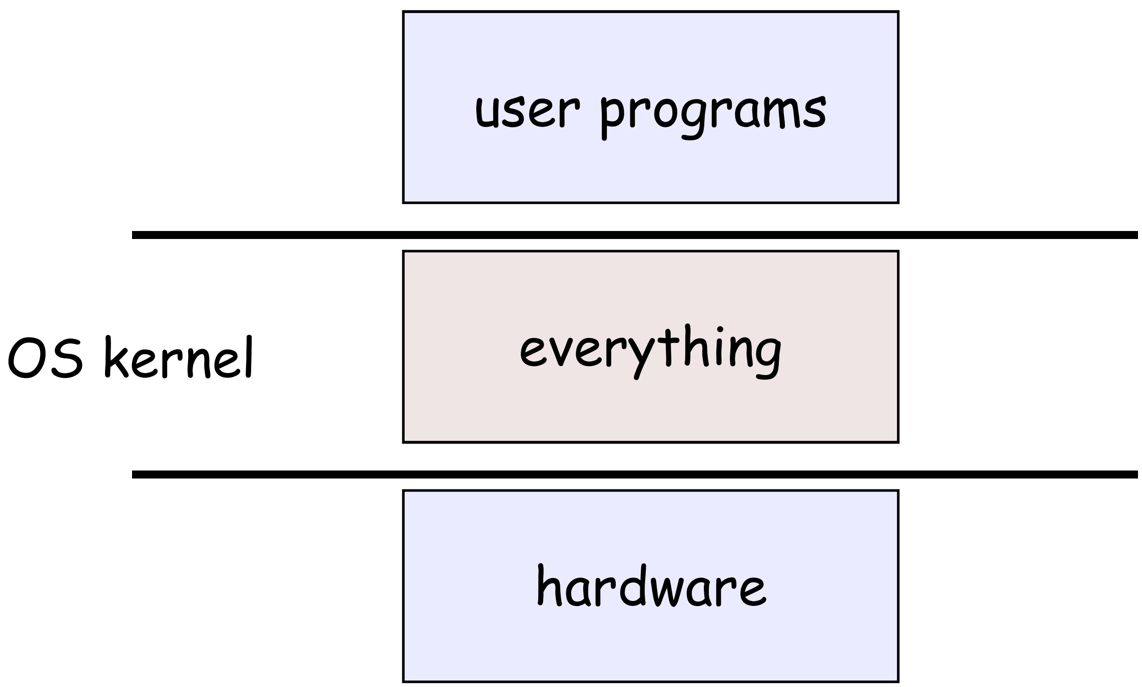


OS Structure

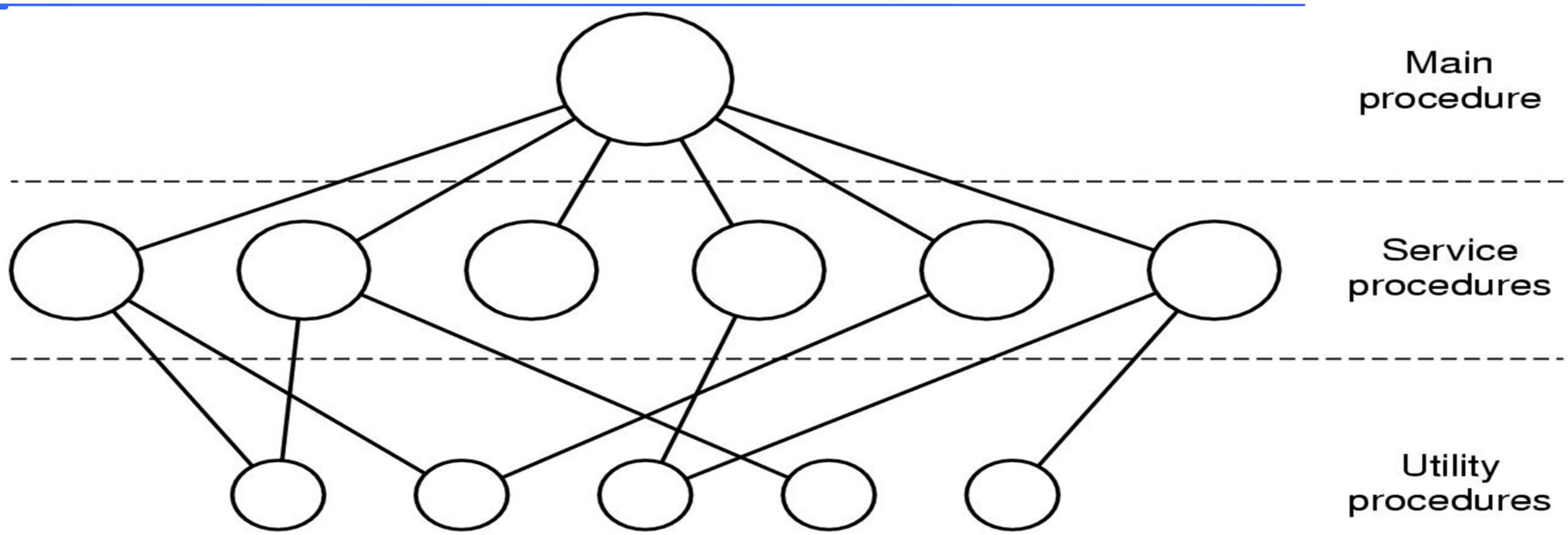
- An OS consists of all of these components, plus:
  - *system programs (privileged and non-privileged)*
    - *e.g. bootstrap code, the init program, …*
- Major issue:
  - *how do we implement the modules?*
    - *Which programming language?*
  - *how do the modules cooperate?*
  - *how do we organize all this?*
- Massive software engineering and design problem
  - *design a large, complex program that:*
    - *performs well, is reliable, is extensible, is backwards compatible, …*

Early structure

- Traditionally, OS's (like UNIX) were built as a monolithic (macro) kernel:



Monolithic Kernels



- Simple structuring model for a monolithic OS
  - *All system calls TRAP to a main procedure*
  - *Main procedure looks at the system call number, typically passed in a register, and invokes the appropriate service procedure*
  - *A service procedure may use one or more utility procedures to do its job*
  - *Utility procedures may be shared by different system call service procedures*

Monolithic Kernels

- Major advantage:
  - *cost of module interactions is low (procedure call)*
- Disadvantages:
  - *hard to understand*
  - *hard to modify*
  - *unreliable (no isolation between system modules)*
  - *hard to maintain*
- What is the alternative?
  - *find a way to organize the OS in order to simplify its design and implementation*

Monolithic Kernel Examples

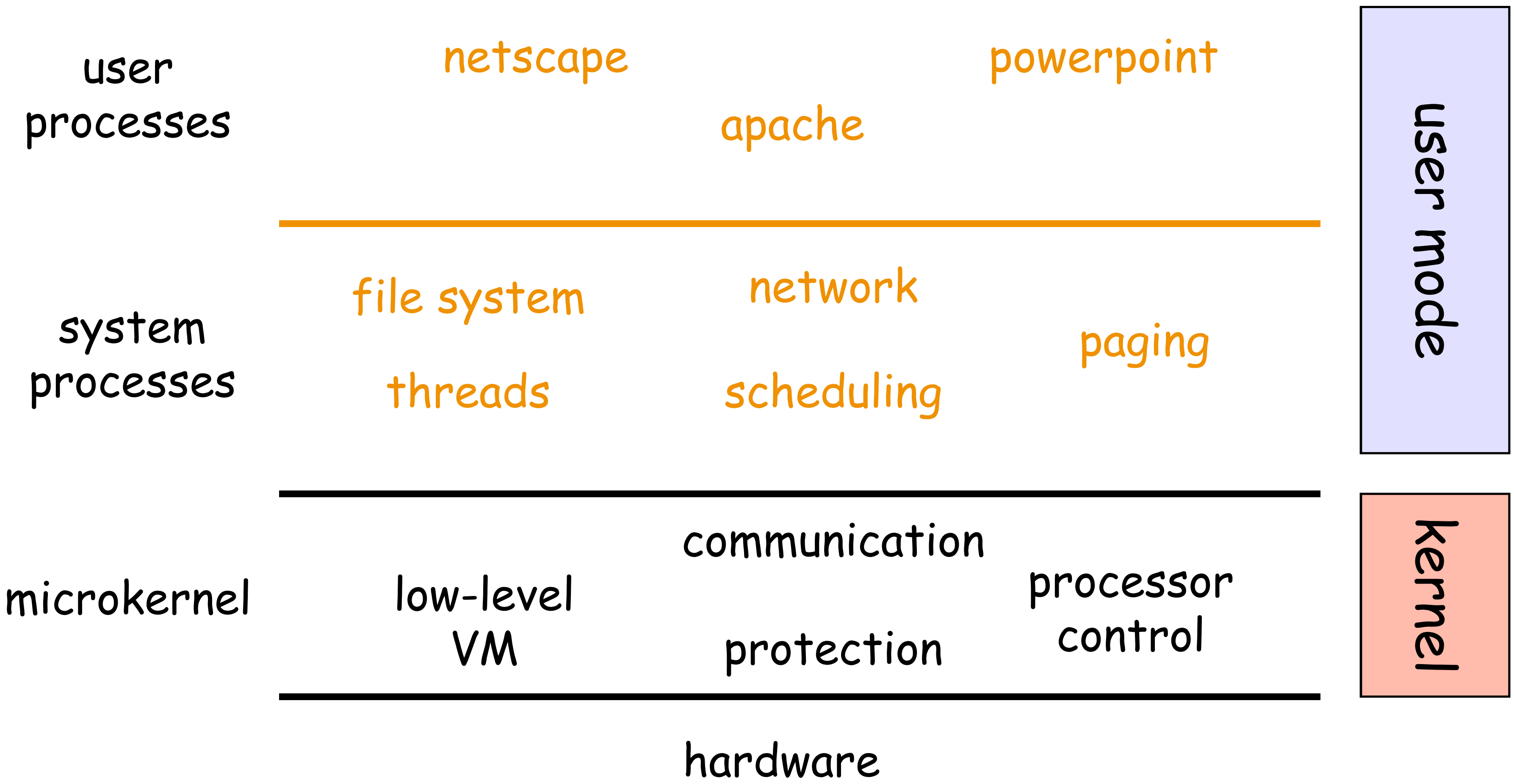
- Unix kernels
  - *BSD*
    - *FreeBSD*
    - *NetBSD*
    - *OpenBSD*
    - *Solaris 1 / SunOS 1.x-4.x*
  - *UNIX System V*
    - *AIX*
    - *HP-UX*
- Unix-like kernels
  - *Linux*
- DOS
  - *DR-DOS*
  - *MS-DOS*
    - *Microsoft Windows 9x series (95, 98, Windows 98SE, Me)*

Microkernels

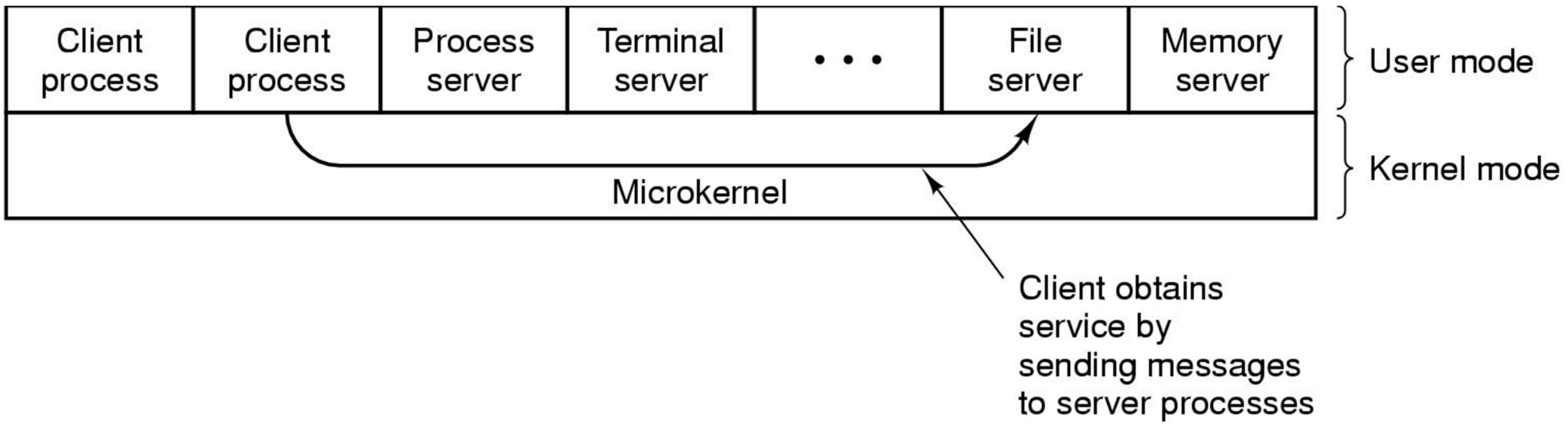
- Motivation:
  - *Monolithic kernels are fast, but very hard to maintain, extend, debug*
- Goal:
  - *minimize what goes in kernel*
  - *organize rest of OS as user-level processes*

- This results in:
  - *better reliability (isolation between components)*
  - *ease of extension and customization*

Microkernels



MicroKernels



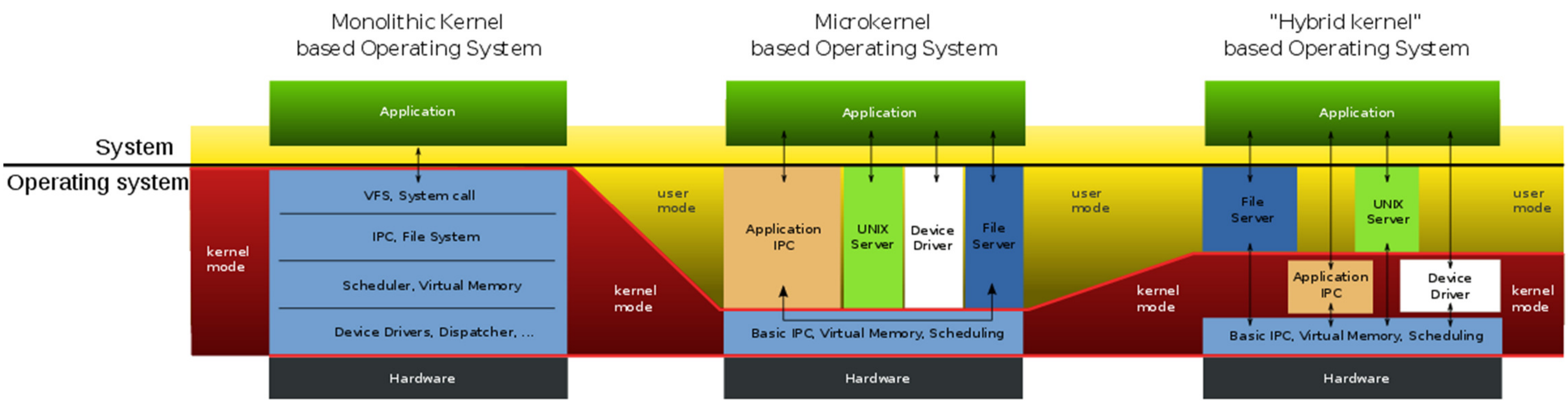
- To request a service, such as reading a block of a file, a user process (now called the client), sends the request to a server process, which then does the work and sends back the answer
- Kernel's job is to handle communication between client & server processes
- Very small kernel BUT leads to poor performance!

- First microkernel system was Hydra (CMU, 1970)

Hybrid Kernel

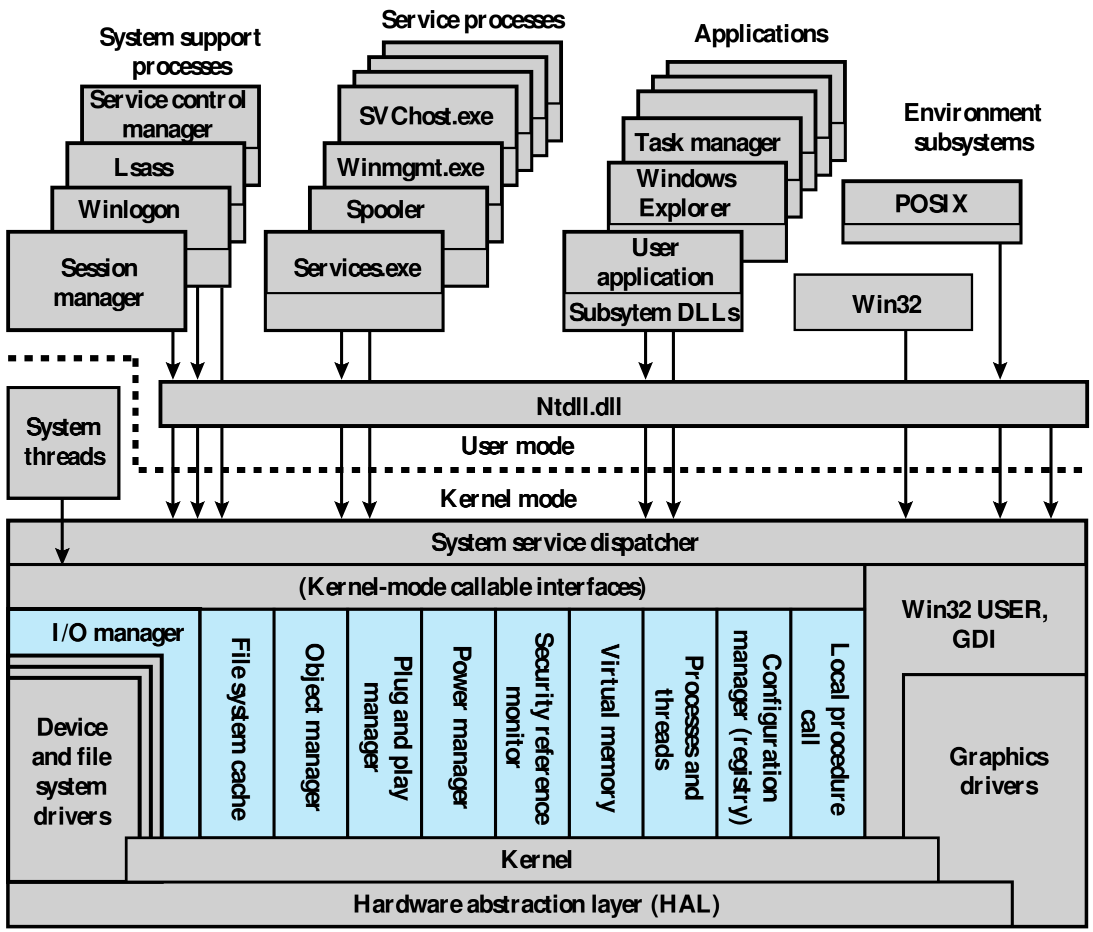
- A hybrid kernel is a kernel architecture based on combining aspects of microkernel and monolithic kernel architectures used in computer operating systems
- Examples
  - *BSD-based*
    - *DragonFly BSD (first non-Mach BSD OS to use a hybrid kernel, concepts inspired by AmigaOS)*
    - *XNU kernel (core of Darwin, used in Mac OS X and iOS)*
  - *NetWare kernel*
  - *NT kernel (used in Windows NT 3.1, Windows NT 3.5, Windows NT 4.0, Windows 2000, Windows Server 2003, Windows XP, Windows Vista, Windows Server 2008, Windows 7)*

Kernel Structures



(Ref:wikimedia.org)

Windows Architecture



Lsass = local security authentication server
POSIX = portable operating system interface
GDI = graphics device interface
DLL = dynamic link libraries

Colored area indicates Executive

Figure 2.14 Windows Architecture

Traditional UNIX Kernel

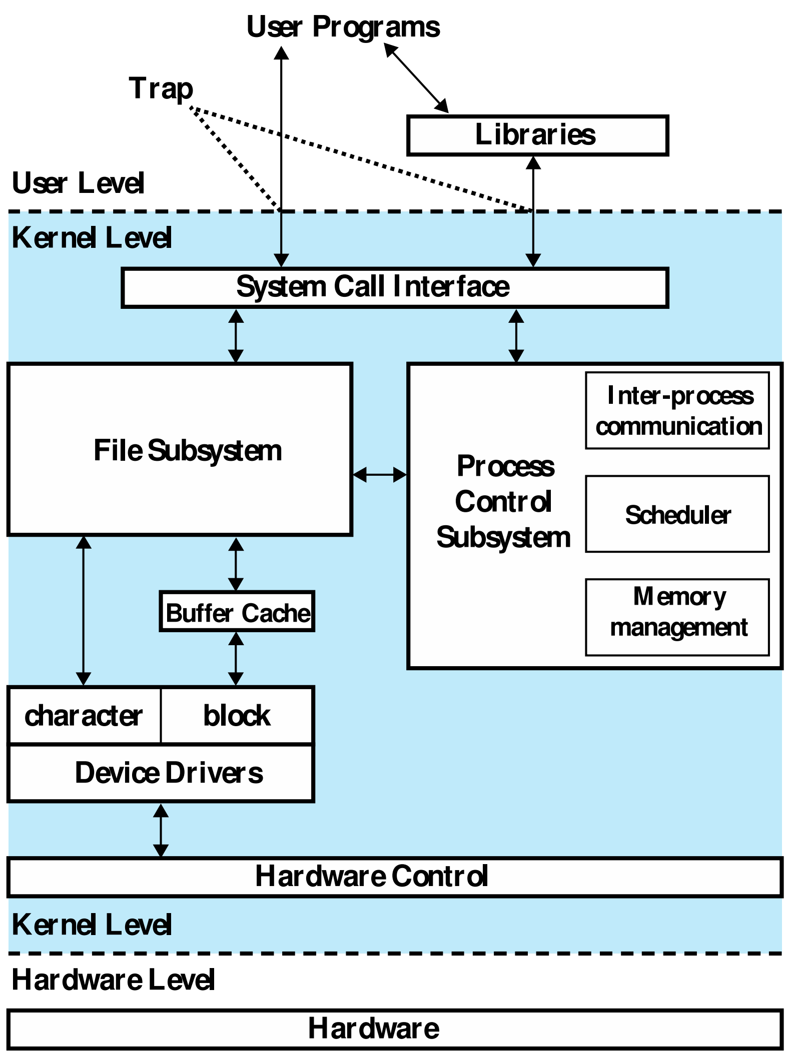


Figure 2.16 Traditional UNIX Kernel

LINUX Overview

- Started out as a UNIX variant for the IBM PC
- Linus Torvalds, a Finnish student of computer science, wrote the initial version
- Linux was first posted on the Internet in 1991
- Today it is a full-featured UNIX system that runs on several platforms
- Is free and the source code is available
- Key to success has been the availability of free software packages
- Highly modular and easily configured

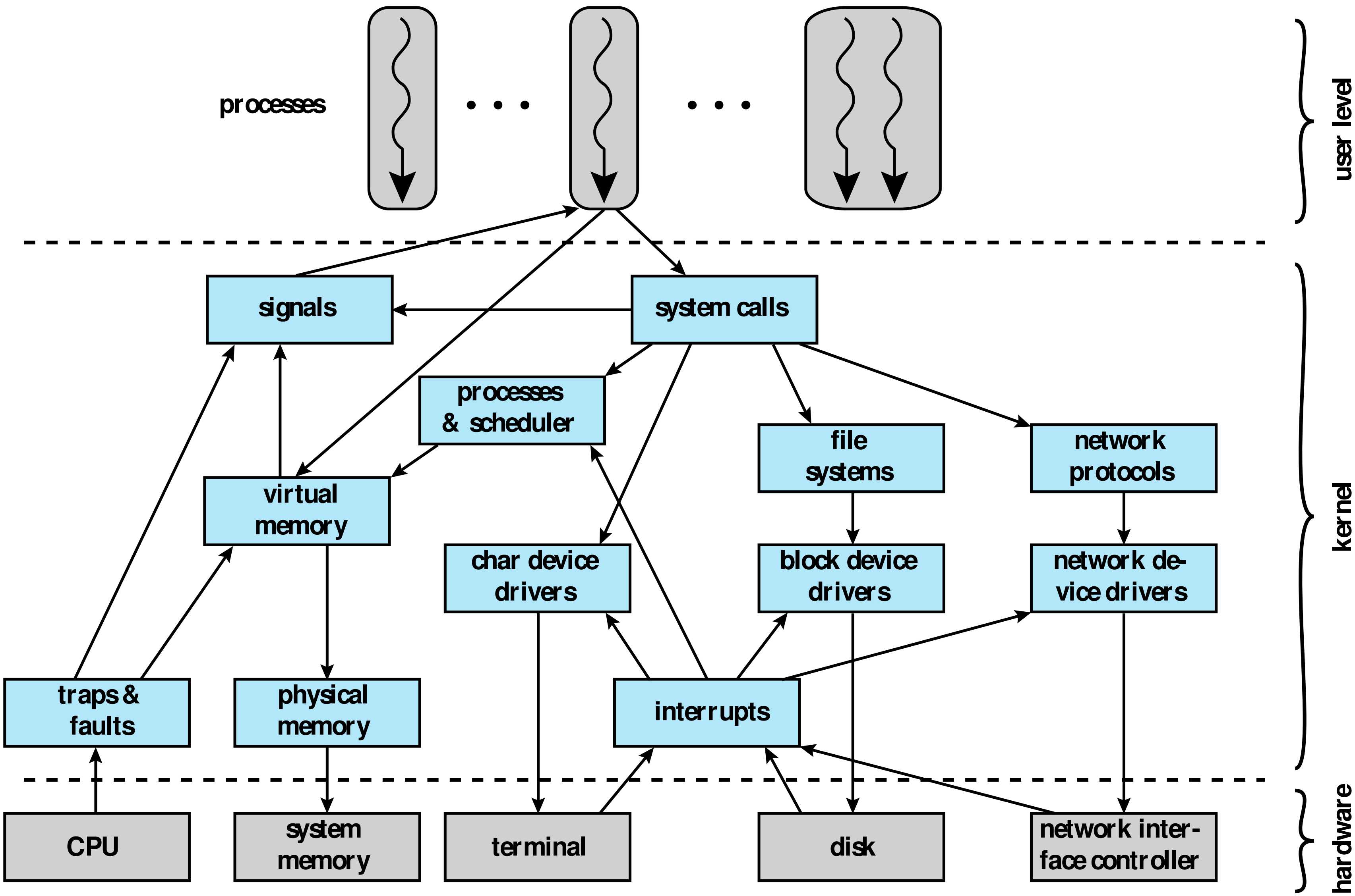
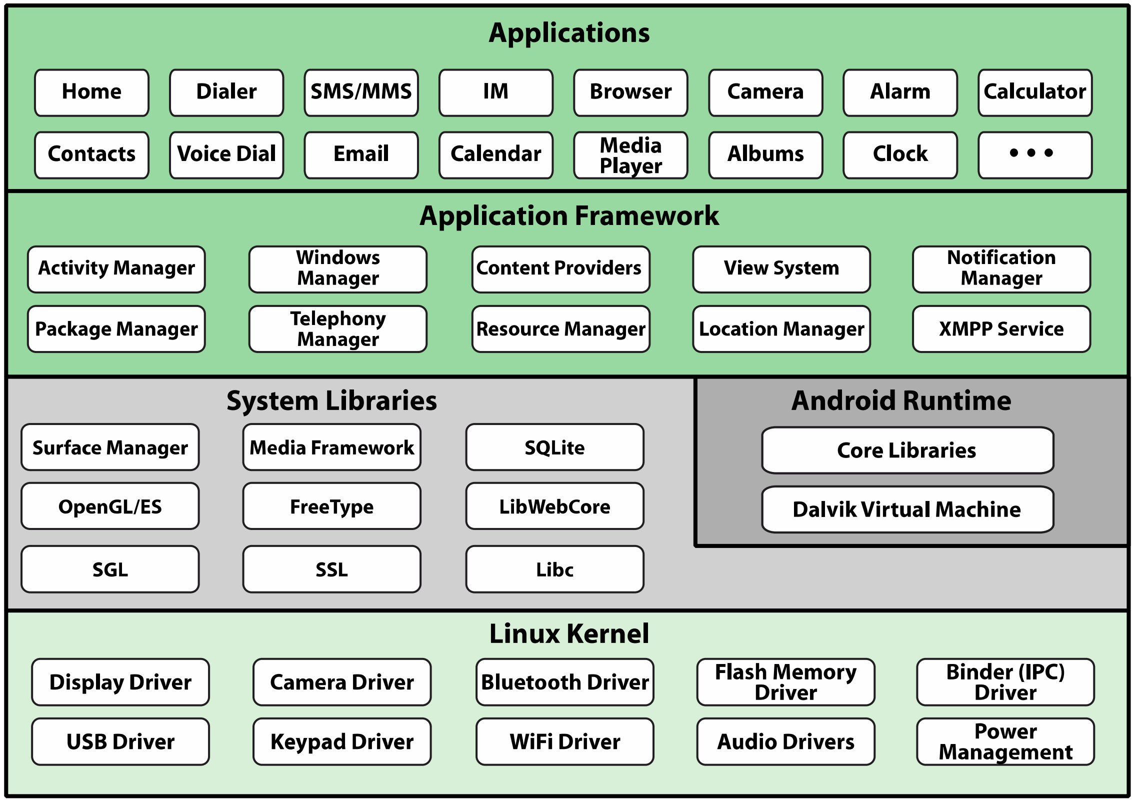


Figure 2.19 Linux Kernel Components

Android Operating System

- A Linux-based system originally designed for touchscreen mobile devices such as smartphones and tablet computers
- The most popular mobile OS
- Development was done by Android Inc., which was bought by Google in 2005
- 1st commercial version (Android 1.0) was released in 2008
- Most recent version is Android 4.3 (Jelly Bean)
- The Open Handset Alliance (OHA) was responsible for the Android OS releases as an open platform
- The open-source nature of Android has been the key to its success

Android



Implementation:

Applications, Application Framework: Java

System Libraries, Android Runtime: C and C++

Linux Kernel: C

Figure 2.20 Android Software Architecture

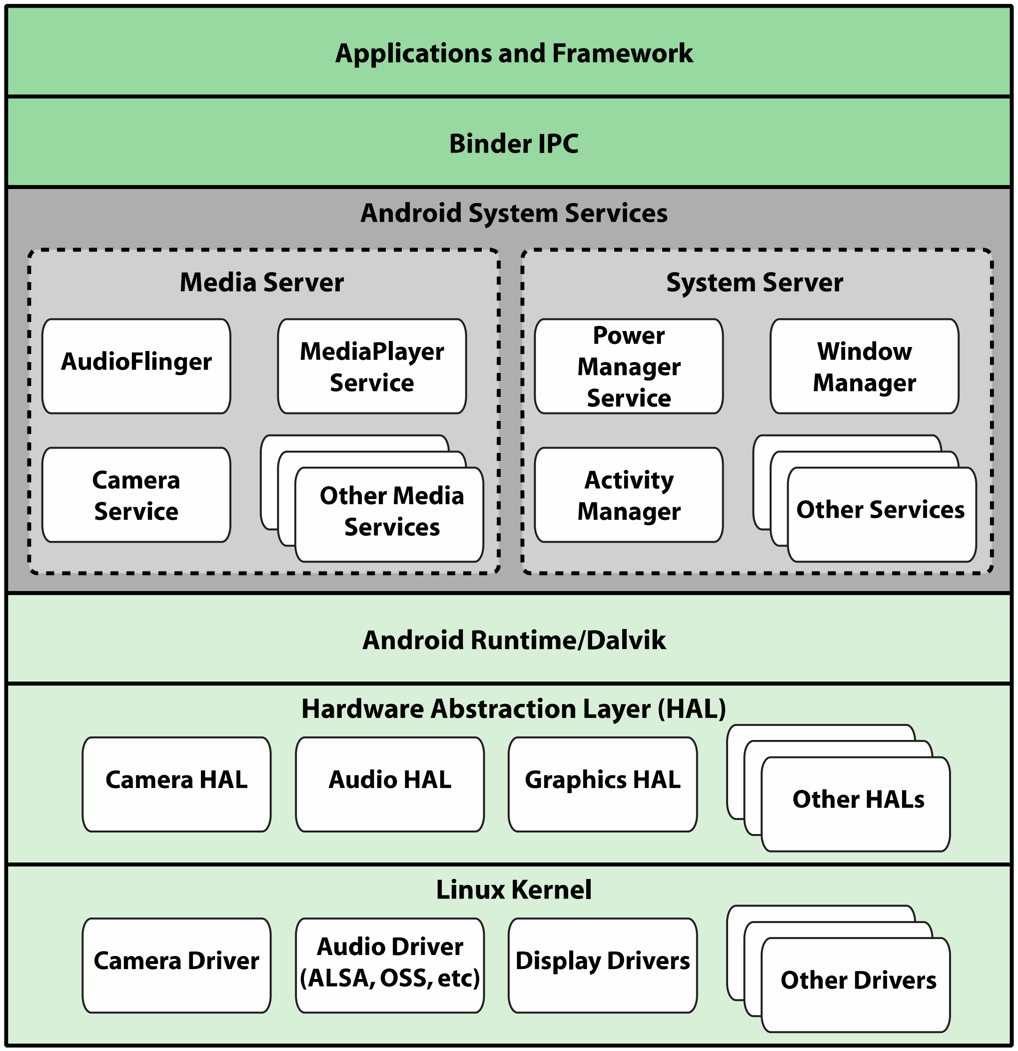
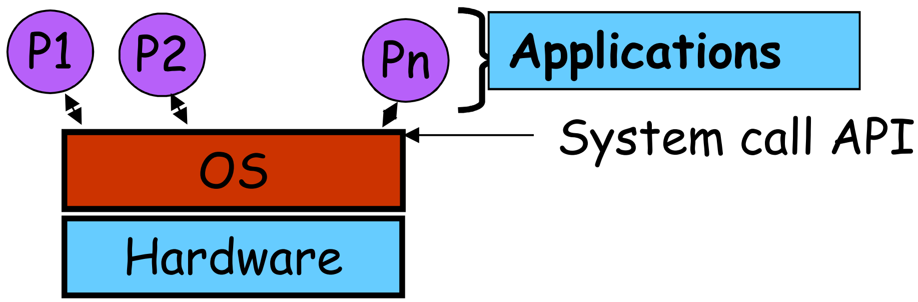


Figure 2.21 Android System Architecture

OS – User Program Interface



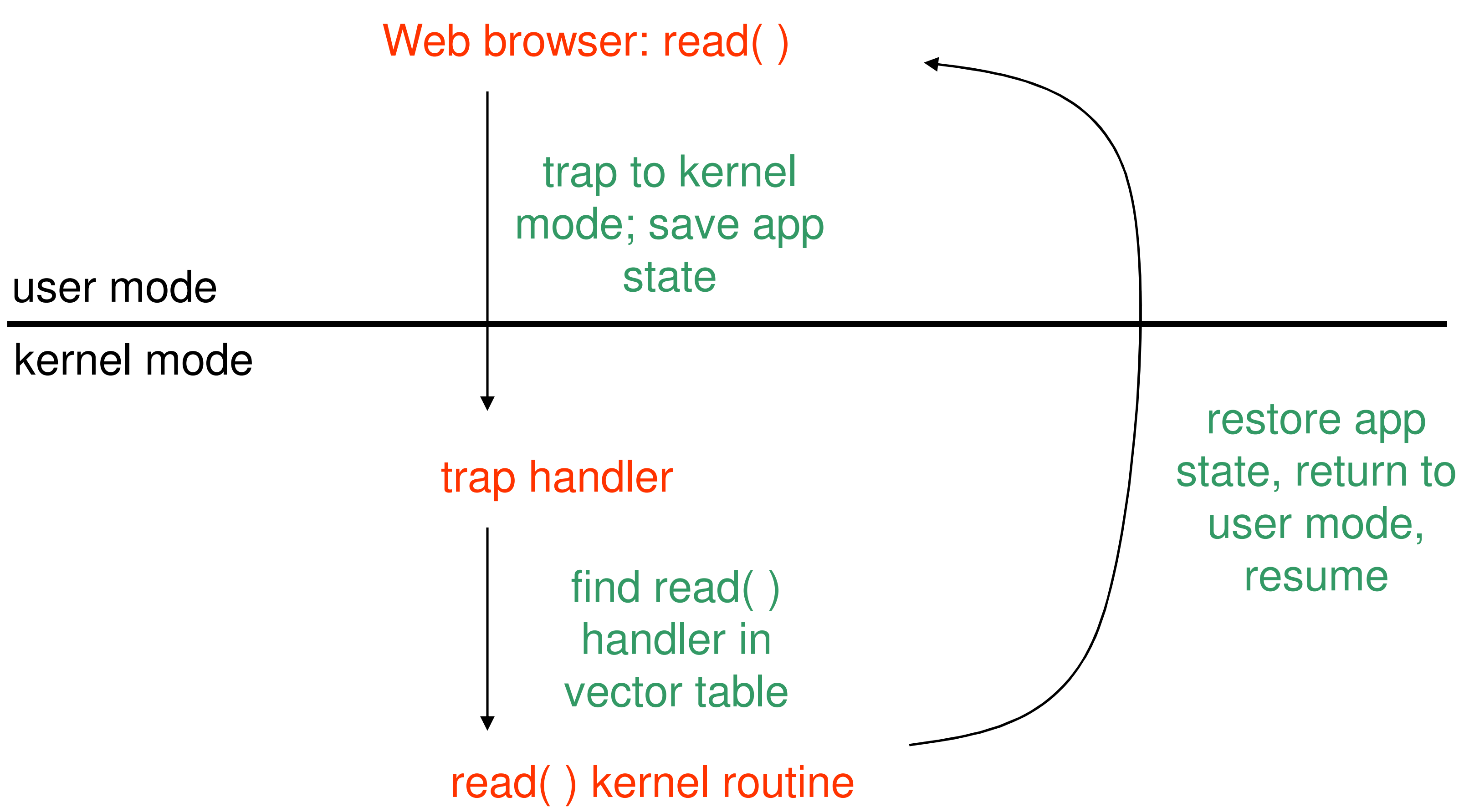
- How does a user program ask services from the OS?
  - *OS exports what's called a system call API to user programs*
  - *System call API consists of several functions that the user can call to ask certain services from the OS*
    - *Create a process, terminate a process, inter-process communication, read/write a file/directory, send/receive a network packet…*
  - *Each OS defines its own system call API*
    - *Win32 API for Windows*
    - *POSIX for Unix systems*

Example System Calls

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

- Some Unix & Win32 API calls

A Kernel Crossing (System Call) Illustrated



Dual Mode Operation

- Most CPUs, except very simple ones used in embedded systems, have two modes of operation
  - *User mode: execution done on behalf of the user*
  - *Kernel or system mode: execution done on behalf of OS*
  - *Usually a bit is PSW indicate the mode*

- In kernel mode:
  - *CPU can execute every instruction in the instruction set and use every feature of the hardware*
- In user mode:
  - *CPU can run only a subset of the instructions*
  - *Generally instructions involving I/O and memory protection are disallowed in user mode.*

Dual Mode Operation

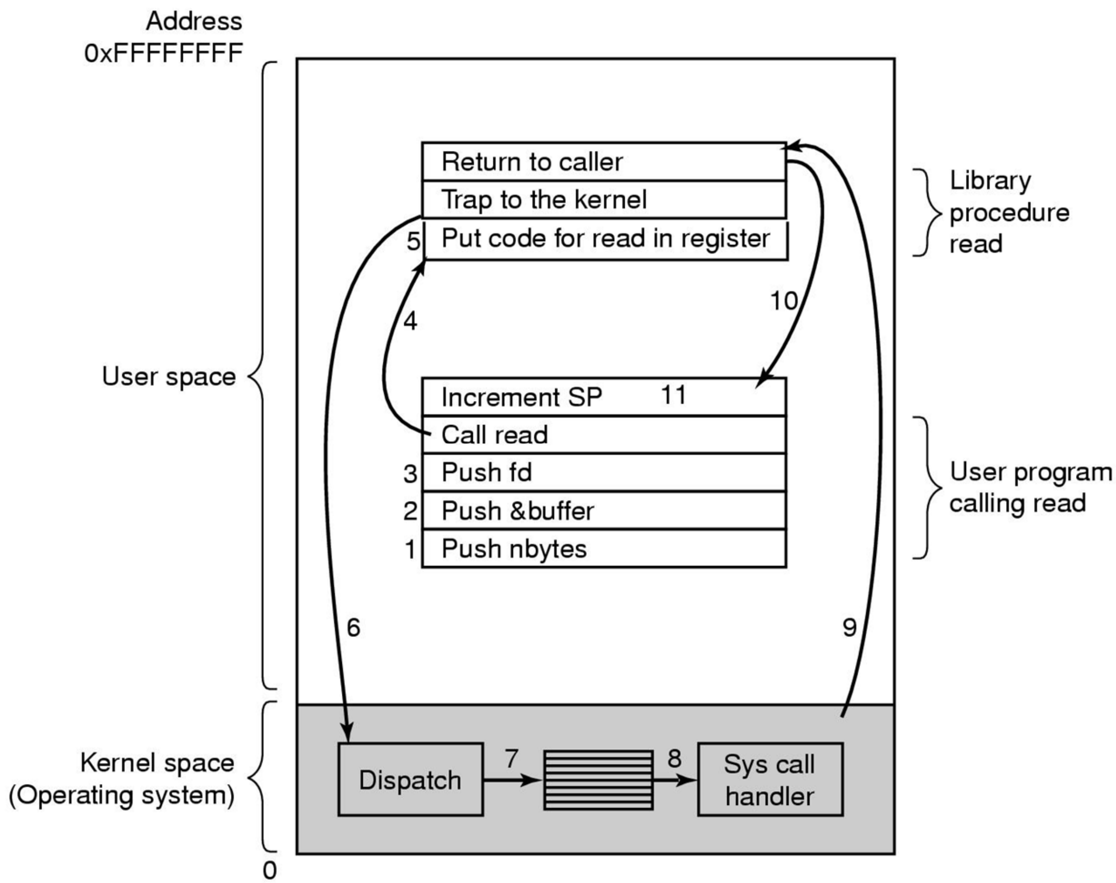
- How do we switch from user mode to kernel mode?
  - *A special "TRAP" instruction switches from user mode to kernel mode*
  - *This is how user programs ask services from the OS*
    - *User program makes a "system call", which TRAPs to the OS and invokes OS code (int 0x80 in Linux)*
    - *When the work is done, control is returned to the user program at the instruction following the system call*
  - *A user program may also TRAP to OS for other reasons*
    - *Divide by 0*
    - *Invalid memory reference*
    - *Floating point overflow/underflow*
    - *…*

More on System Calls

- Kernel must save process state before making the call.
- Within the call, kernel must verify arguments
- How can you reference kernel objects as arguments or results to/from system calls?
  - *E.g., how does a program reference a file object?*

- How does the user program pass arguments to the OS?
  - *Pass parameters in* registers. *-- Linux*
  - *Store the parameters in a table in memory, and the table address is passed as a parameter in a register.*
  - Push *(store) the parameters onto the* stack *by the program, and* pop *off the stack by operating system.*

Steps in Making a System Call



- There are 11 steps in making the system call
  - *read (fd, buffer, nbytes)*

System Calls on Linux

- Look into <include/asm-i386/unistd.h>
- Each system call has a number[0-MaxSysCall]
  - *E.g., exit: 1, read: 3, write: 4, … up to 273 system calls in Linux 2.6.3*
- First put system call number in register eax
- Then put each parameter into specific registers
  - *Passes parameters in registers*
  - *E.g.: read(fd, buffer, 5);*
    - *Eax = 3 // System call # for read system call*
    - *Ebx = fd // File Descriptor number*
    - *Ecx = buffer // pointer to buffer*
    - *Edx = 5 // # of bytes to read from the file*
- Finally, execute int 0x80 (trap to the kernel)

System Calls on Linux

- Interrupt 0x80 does the following
  - *changes the mode from user to kernel and calls the ISR (Interrupt service routine) that implements interrupt 0x80*
  - *Look into <arch/i386/kernel/entry.S> for the ISR of interrupt 0x80 (search for sysenter_entry)*
  - *The ISR first pushes all process registers onto the stack (SAVE_ALL)*
  - *Then uses system call number in eax as an index into a system call table (sys_call_table), where the addresses of the functions that implement system calls are stored*
    - *call *sys_call_table(,%eax, 4)*
    - *Look under <arch/i386/kernel/syscall_table.S> for sys_call_table*
  - *The function that implements the system call is now called*
  - *When the system call function is done executing, the OS restores the process state and returns from the ISR*
  - *The result of the system call is then put into variable "errno" and the system call returns to the user*

Linux Kernel Structure

- init/
  - *All functions needed to start the kernel*
  - *Kernel_start: init kernel, create "init" process*
- kernel/ & arch/i386/kernel
  - *Implementation of main system calls*
  - *Timers, schedulers, DMA, IRQ, signal management*
- mm/
  - *Memory management functions*
- net/
  - *IPv4, IPv6, ARP, TCP, UDP implementations*
- fs/
  - *Virtual File System (vfs) related functions*
- drivers/
  - *Drivers (file system, network, disk…)*
- arch/
  - *Arcthitecture dependant parts of the kernel*

Linux Kernel Startup Sequence

- After the kernel gets loaded up, the loader jumps to “start_32”
- start_32: [arch/i386/kernel/head.S]
  - ...
  - *jmp kernel_start*
- kernel_start: [init/main.c]
  - …
  - *Start “init” process*
    - */sbin/init*
- init process does the following: (for SUSE linux)
  - *Reads /etc/inittab (for runlevel)*
  - *Executes /etc/init.d/boot*
  - */etc/init.d/boot.local*
  - */etc/init.d/rc → This creates all daemons in the system*
  - *Finally, creates a “login” process for each tty*

Linux Kernel Startup Sequence

1. The BIOS performs hardware-platform specific startup tasks
2. The BIOS loads and executes the partition boot code from the designated boot device, which contains phase 1 of a Linux boot loader.
3. The boot loader often presents the user with a menu of possible boot options. It then loads the operating system, which decompresses into memory, and sets up system functions such as essential hardware and memory paging, before calling start_kernel().
4. start_kernel() then performs the majority of system setup (interrupts, the rest of memory management, device initialization, drivers, etc.) before spawning separately, the idle processand scheduler, and the Init process (which is executed in user space).
5. The Init process executes scripts as needed that set up all non-operating system services and structures in order to allow a user environment to be created, and then presents the user with a login screen.