



“I failed in some subjects in exam,
but my friend passed in all.

Now he is an engineer in Microsoft
and I am the owner of Microsoft”

Bill Gates





Overview





Course Information

COE 354 Operating
Systems OS

Requirements



3hrs Teaching

**Strong Programming Knowledge
in C or C++**

2hrs Practicals

Basic Knowledge in Computers

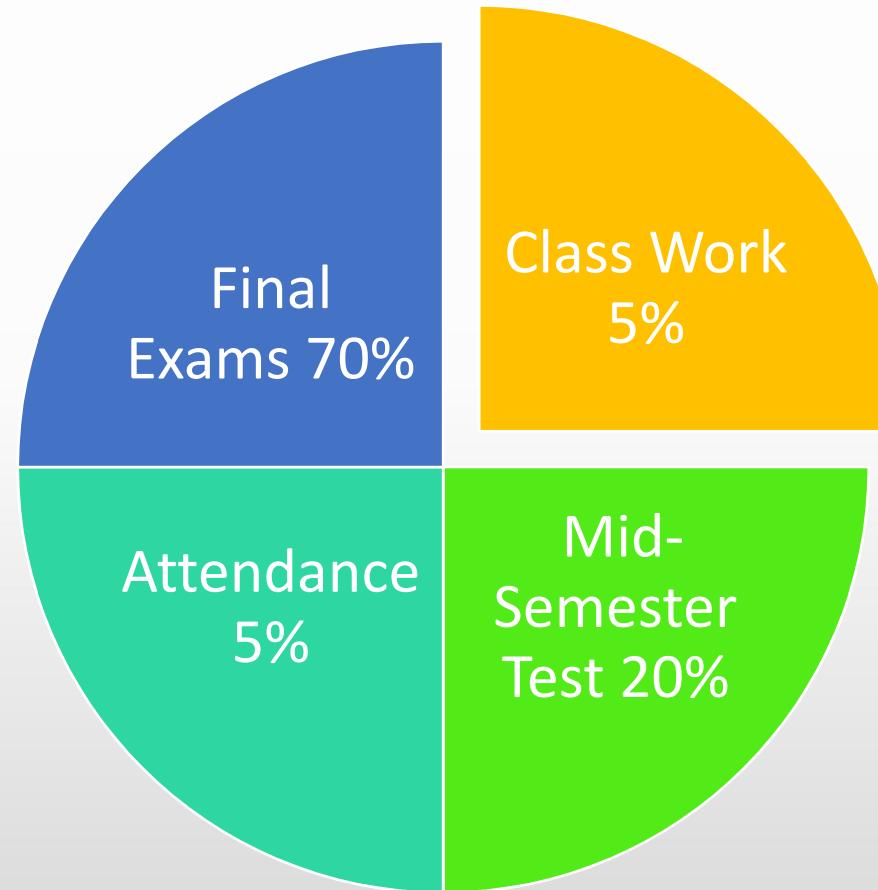
3 Credit Hours

Laptop (optional)





Course Information





Course Outline

Introduction to Operating Systems OS – Week 1

Structure of Operating Systems – Week 2

Processes – Week 3

Threads – Week 4





Course Outline

Scheduling – Week 5

System Overview – Week 5

Synchronization – Week 6





Course Outline

Deadlocks – Week 7

Memory Management – Week 8

Virtual Memory – Week 9

File Systems – Week 10





Course Outline

IO Systems – Week 11

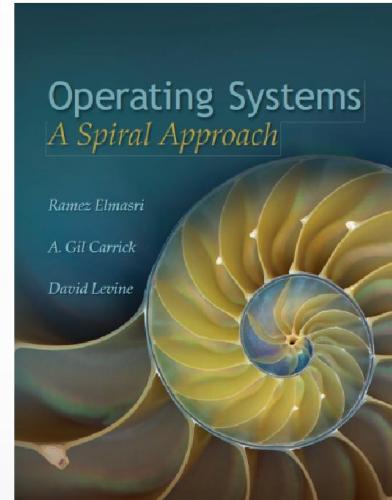
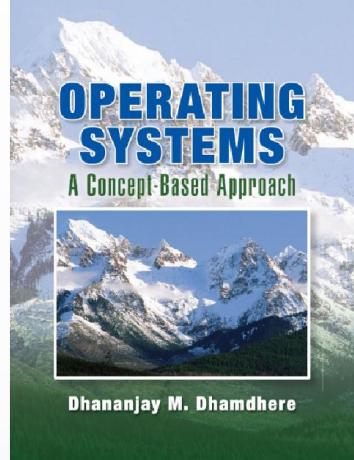
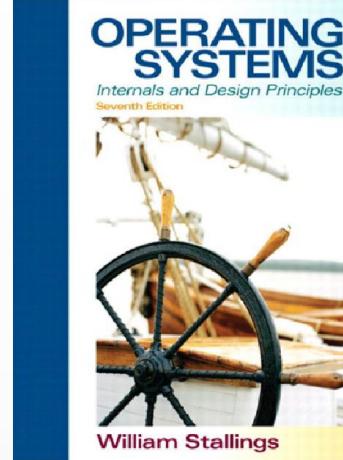
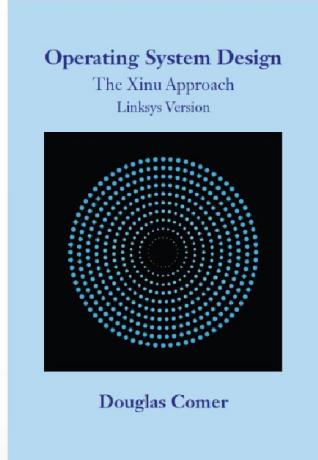
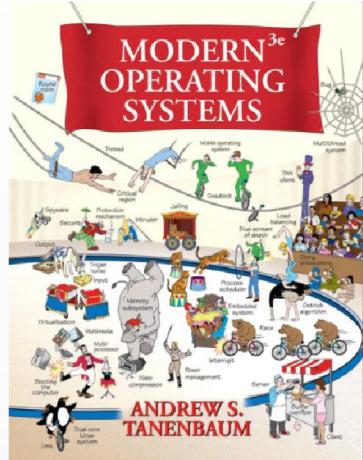
Real Time Operating Systems – Week 12

Revisions – Week 13



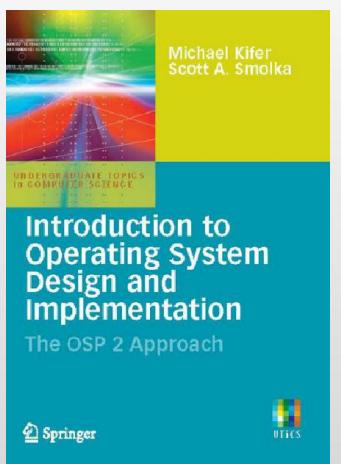
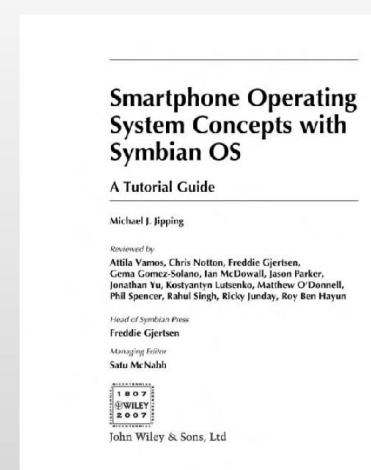
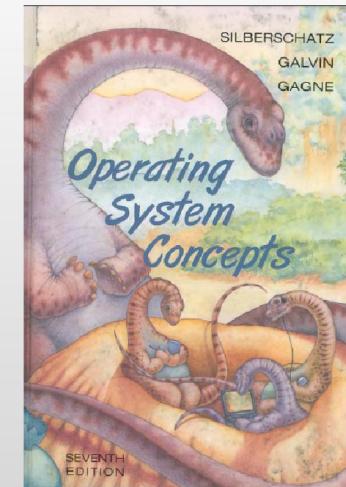
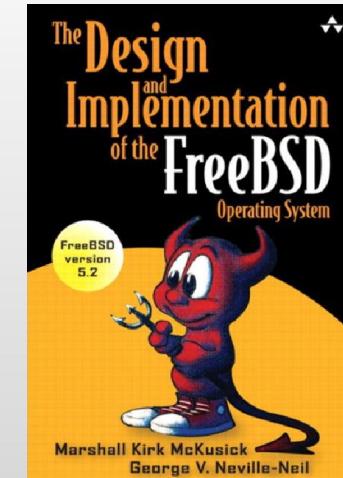
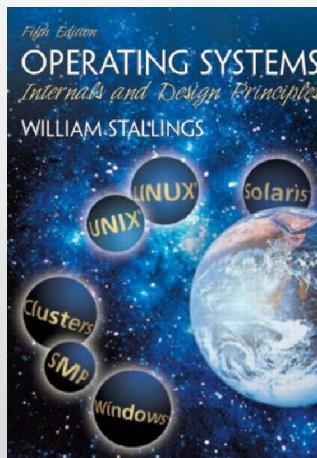
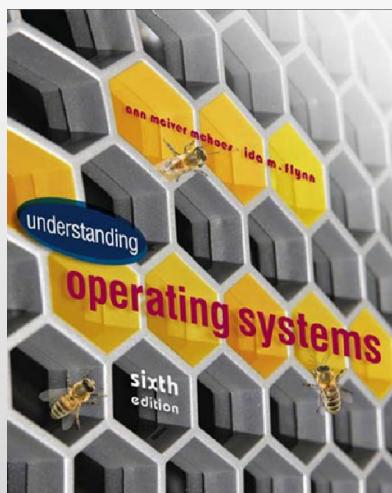


References Books



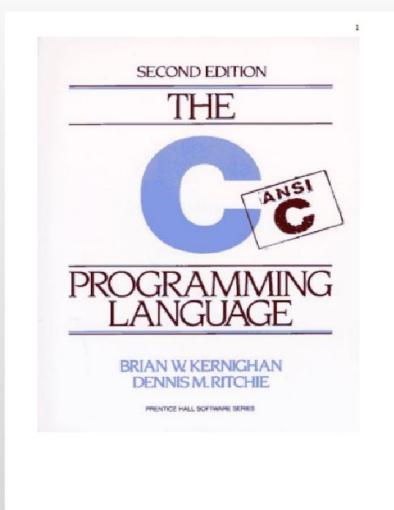
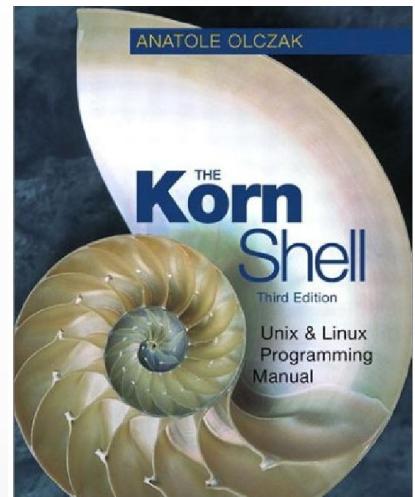
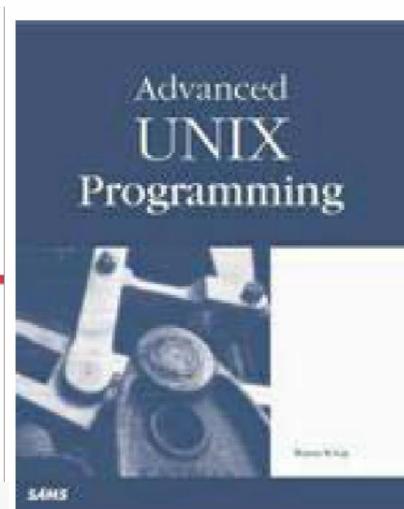
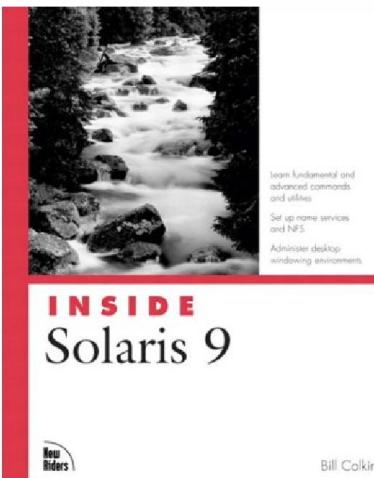
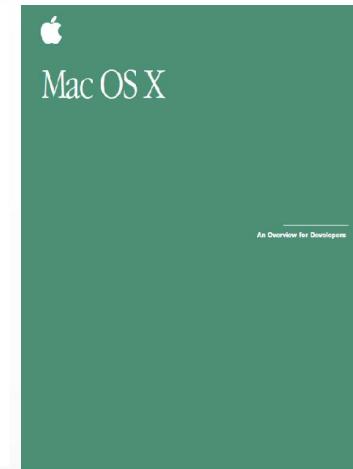
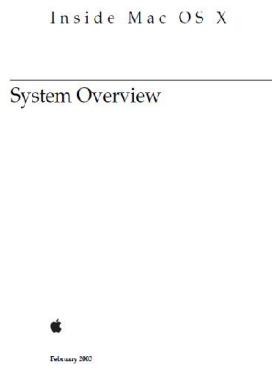
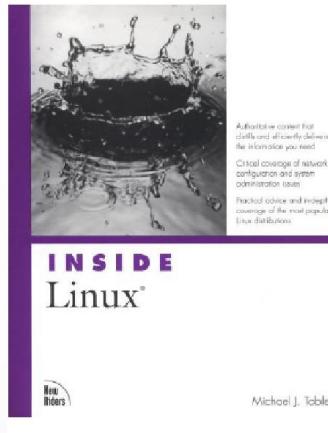
Realtime Operating Systems

Concepts and Implementation of Microkernels
for Embedded Systems
Dr. Jürgen Seemann, Melanie Thielicke





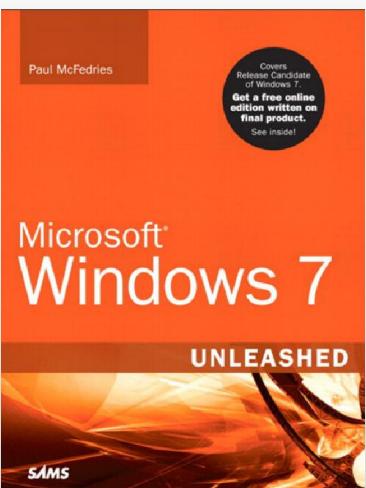
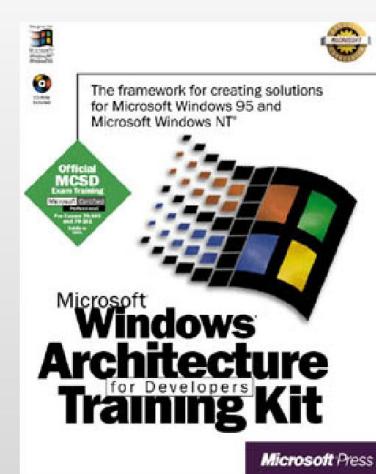
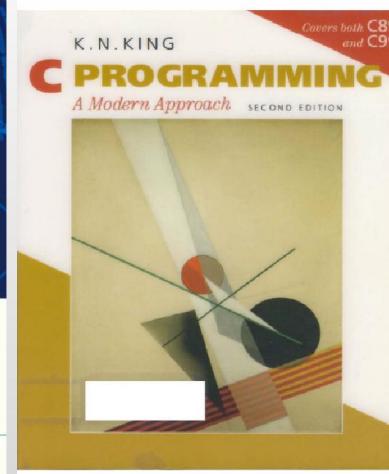
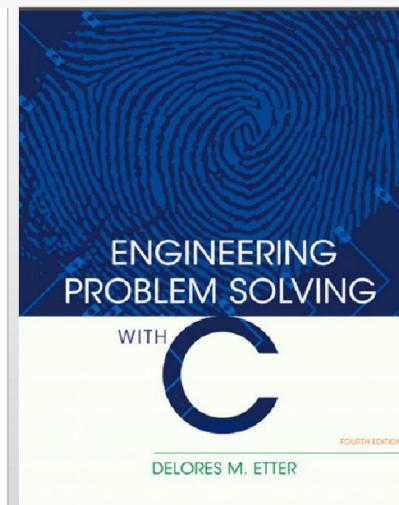
References Books



Let Us C

Fifth Edition

Yashavant P. Kanetkar





Introduction to Operating Systems OS

Operating System OS

An OS is a program which acts as an *interface* between computer system users and the computer hardware

- It provides a user-friendly environment in which a user may easily develop and execute programs.





Operating System OS

Otherwise, hardware knowledge would be mandatory for computer programming.

- So, it can be said that an OS hides the complexity of hardware from uninterested users.





Introduction to Operating Systems OS

OS Examples:

Unix,

Linux,

Solaris,

VxWorks,

MacOS,

Andriod,

Windows

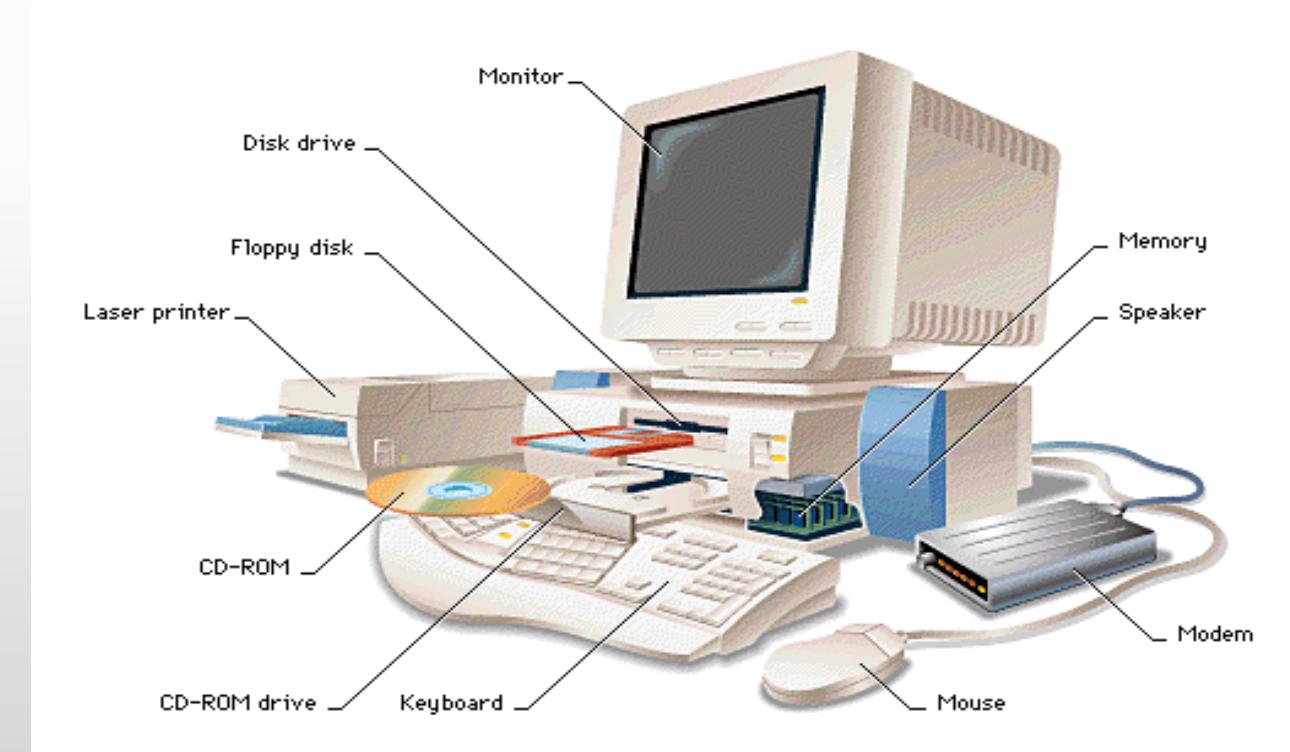




Introduction to Operating Systems OS

A computer system has some resources which may be utilized to solve a problem.

They are
Memory
Processor(s)
I/O
File System
etc.





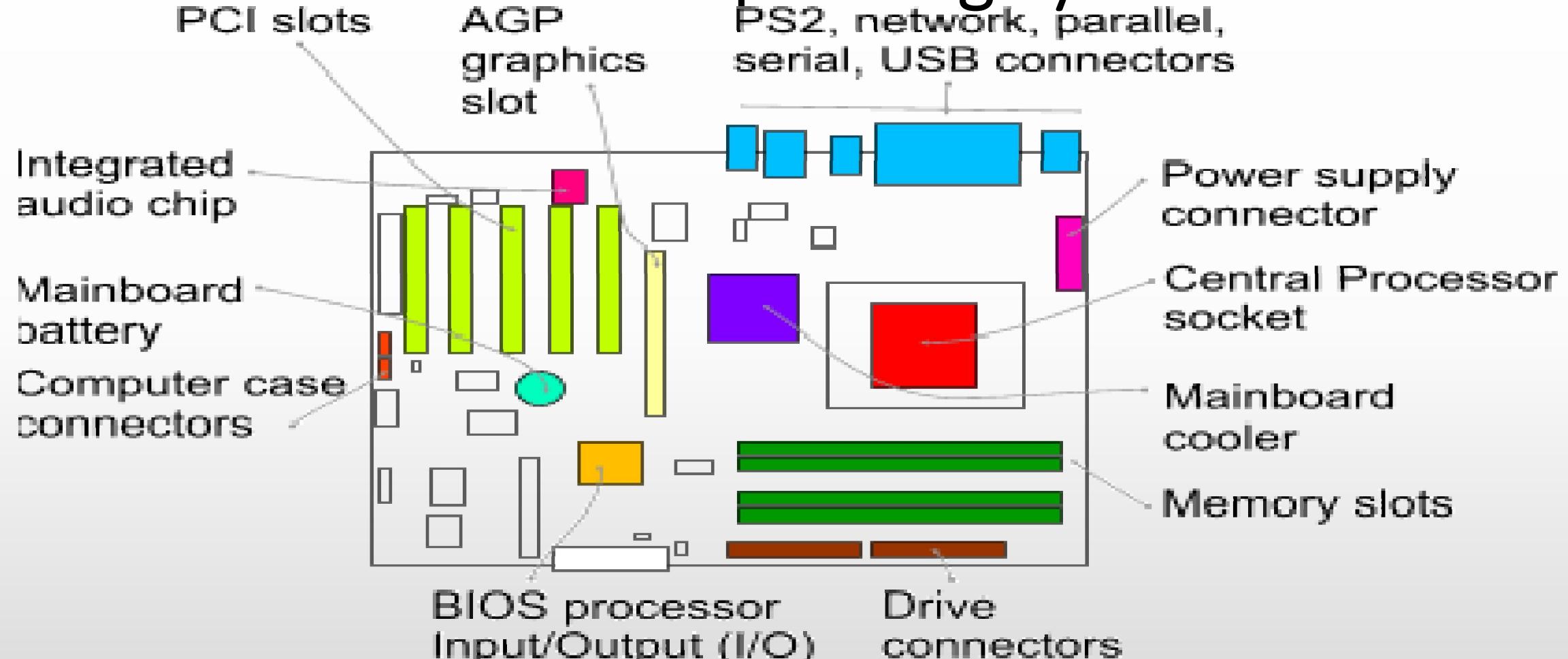
Introduction to Operating Systems OS

The mainboard of a computer





Introduction to Operating Systems OS



Mainboard Blockdiagram





Introduction to Operating Systems OS

The Processor





Introduction to Operating Systems OS

The Random Access Memory RAM





Introduction to Operating Systems OS

The OS manages these resources and allocates them to specific programs and users.

With the management of the OS, a programmer is not bothered with difficult hardware considerations.





Introduction to Operating Systems OS

An OS provides services for

Processor Management

Memory Management

File Management

Device Management

Concurrency Control





Introduction to Operating Systems OS

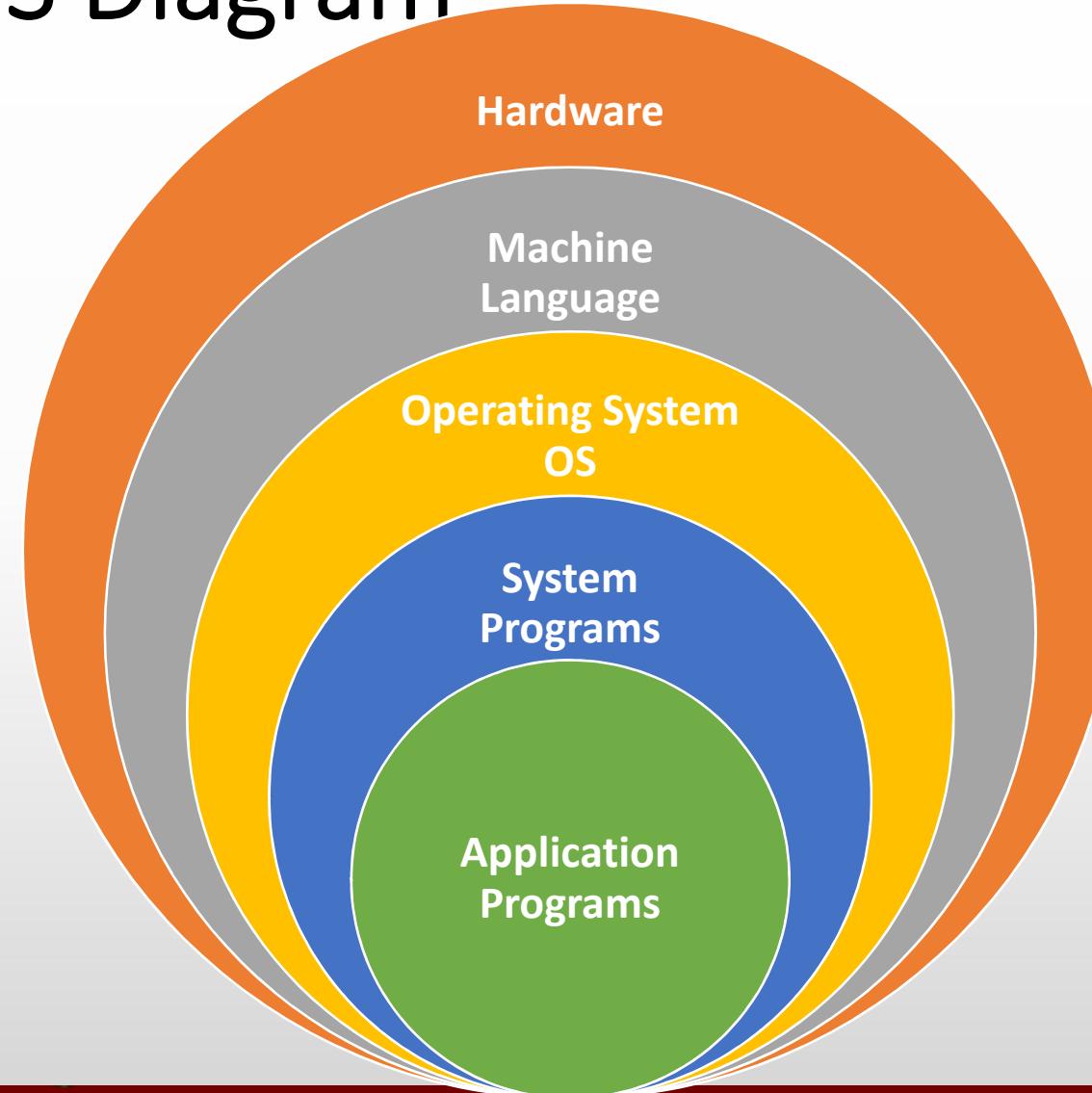
Another aspect for the usage of OS is that; it is used as a *predefined library* for hardware-software interaction.

This is why, system programs apply to the installed OS since they cannot reach hardware directly.





Hardware – OS Diagram





Introduction to Operating Systems OS

Since we have an already written library, namely the OS, to add two numbers we simply write the following line to our program:

$c = a + b ;$

in a system where there is no OS installed, we should consider some hardware work as:

(Assuming an MC 6800 computer hardware)

LDAA \$80 → Loading the number at memory location 80

LDAB \$81 → Loading the number at memory location 81

ADDB → Adding these two numbers

STAA \$55 → Storing the sum to memory location 55

memory locations and hardware knowledge of the system are required.

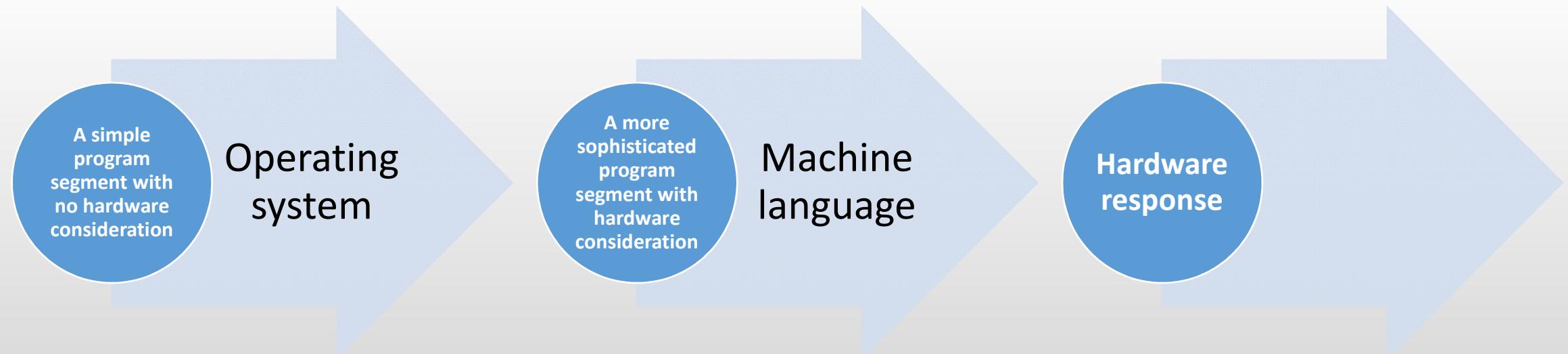




Introduction to Operating Systems OS

In an OS installed machine, since we have an intermediate layer, our programs obtain *some advantage of mobility* by not dealing with hardware.

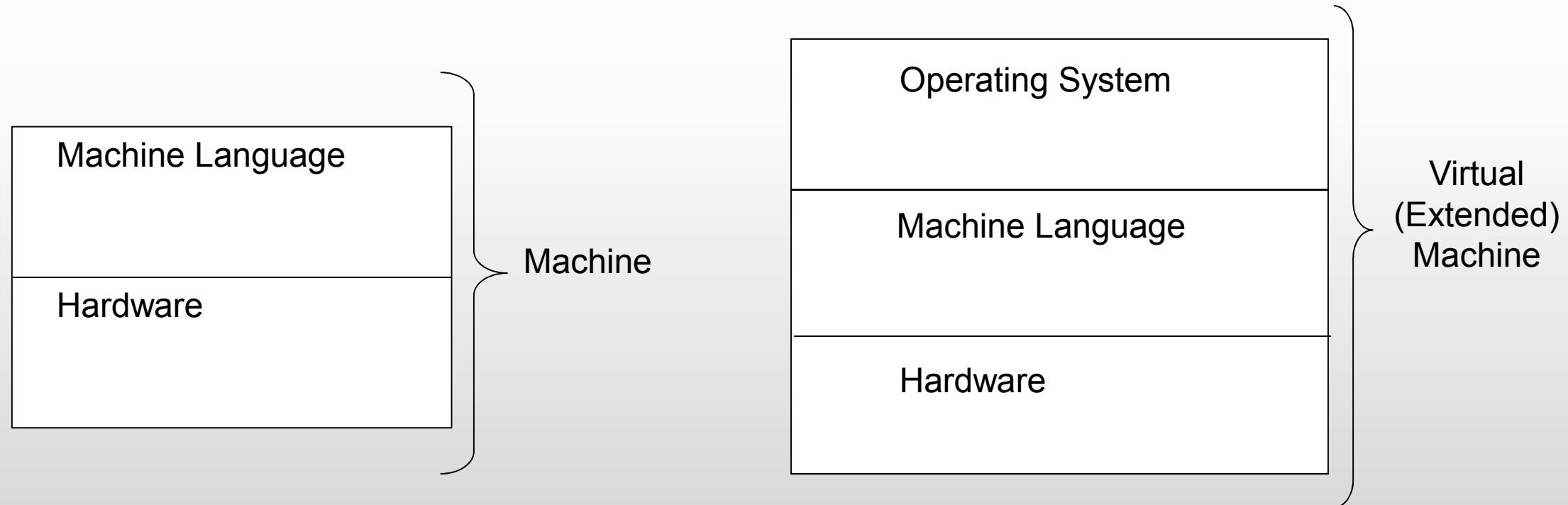
For example, the above program segment would not work for an 8086 machine, whereas the “ $c = a + b ;$ ” syntax will be suitable for both.





Introduction to Operating Systems OS

With the advantage of easier programming provided by the OS, the hardware, its machine language and the OS constitutes a new combination called a **virtual (extended) machine**.





Introduction to Operating Systems OS

The OS is itself a program, but it has a priority which application programs don't have.

OS uses the **kernel mode** of the microprocessor, whereas other programs use the **user mode**.

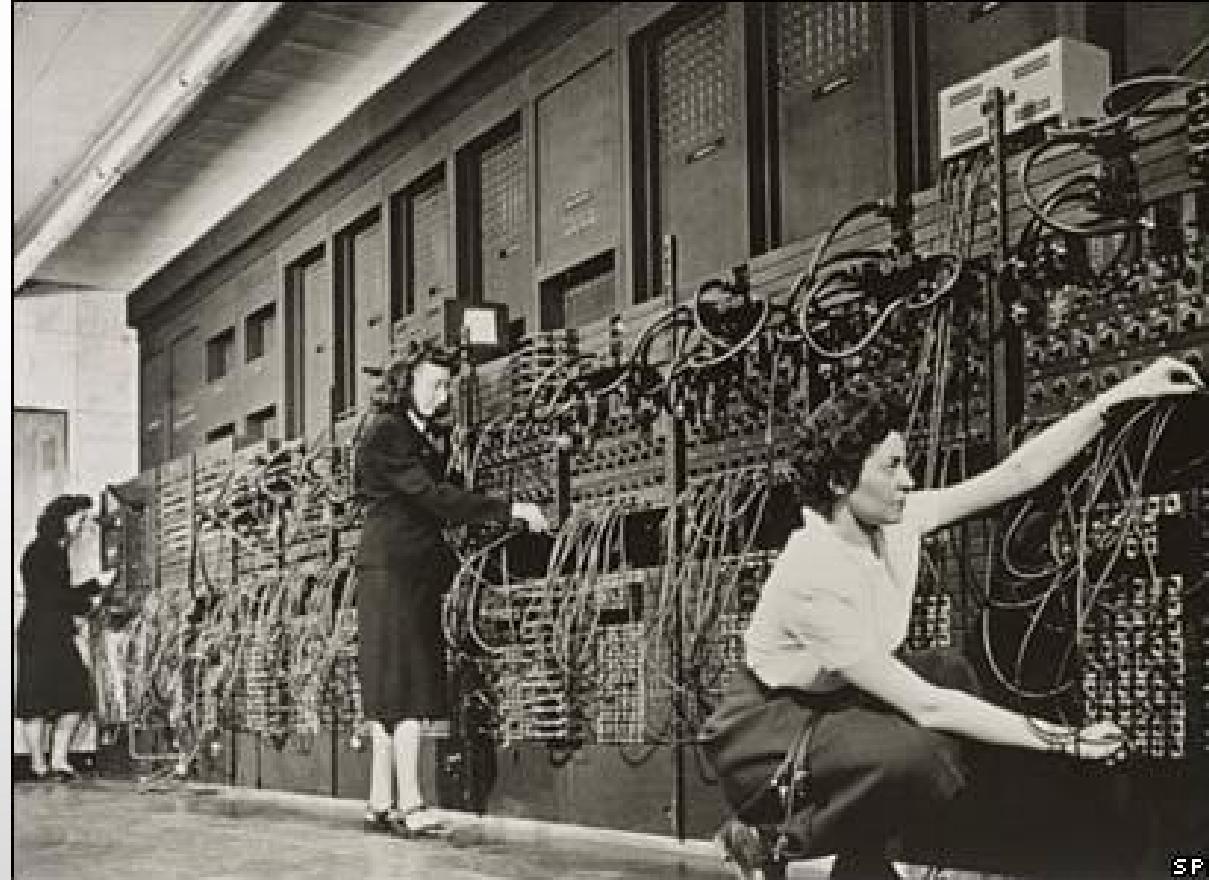
The difference between the two is that; all hardware instructions are valid in kernel mode, where some of them cannot be used in the user mode.





Operating System History

It all started with computer hardware in about 1940s



ENIAC 1943



Operating System History

ENIAC (Electronic Numerical Integrator and Computer),
at the U.S. Army's Aberdeen Proving Ground in Maryland.

built in the 1940s,
weighed 30 tons,
was eight feet high, three feet deep, and 100 feet long
contained over 18,000 vacuum tubes that were cooled
by 80 air blowers.



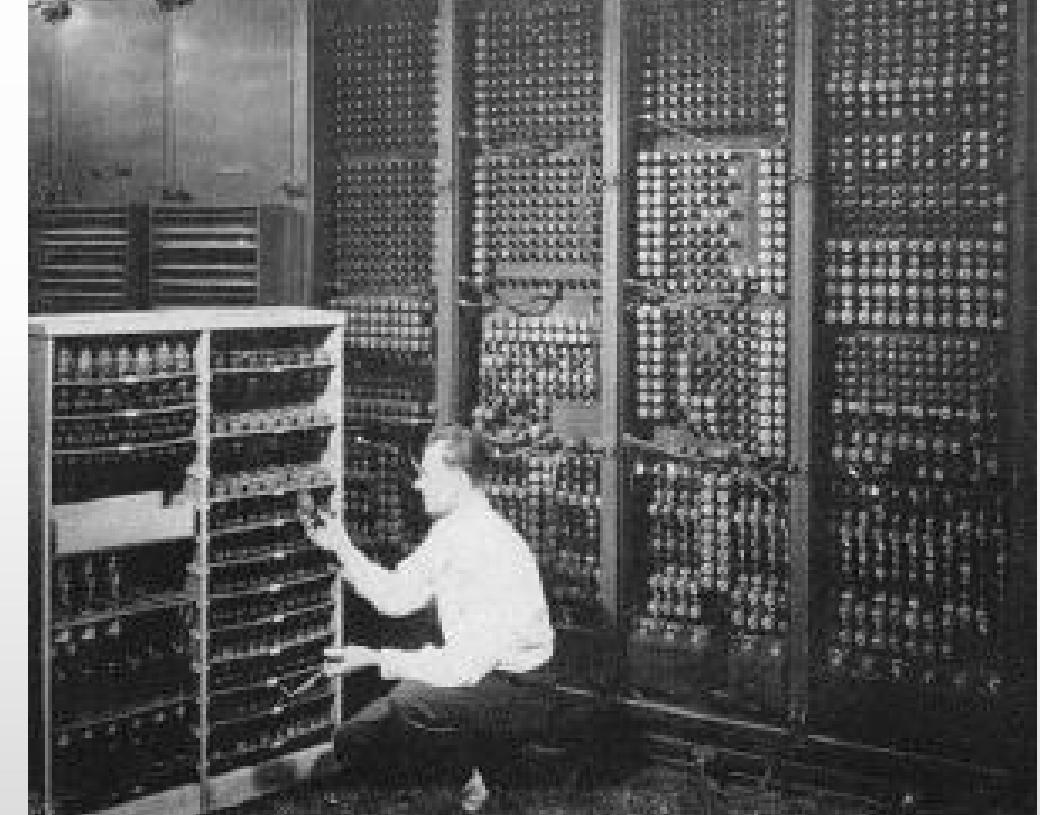


Operating System History

Computers were using vacuum tube technology.



ENIAC's vacuum tubes

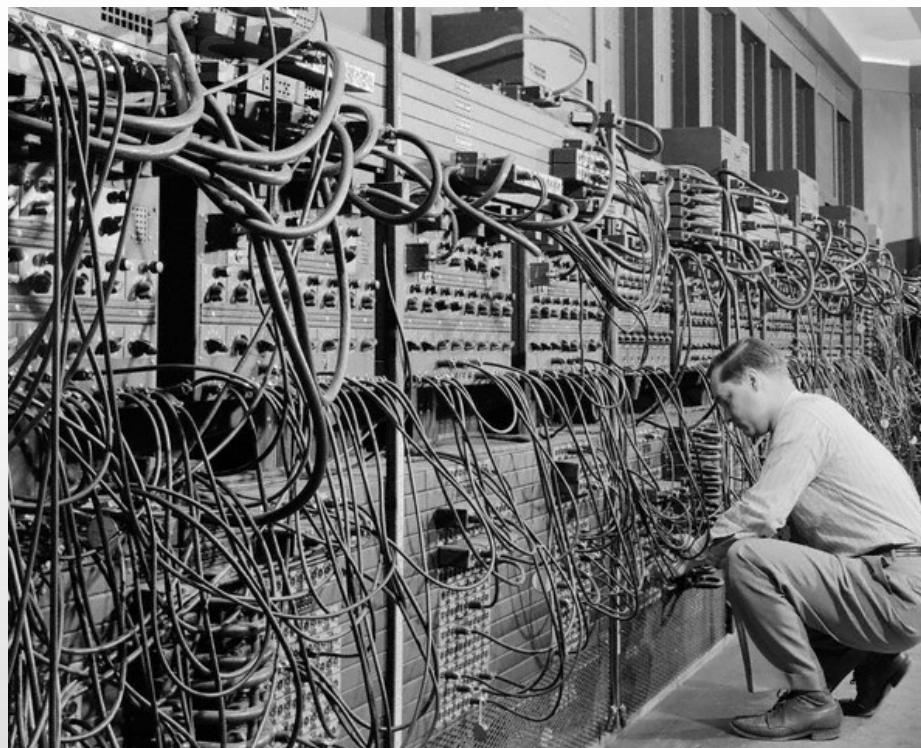


ENIAC's backside



Operating System History

Programs were loaded into memory manually using switches, punched cards, or paper tapes.



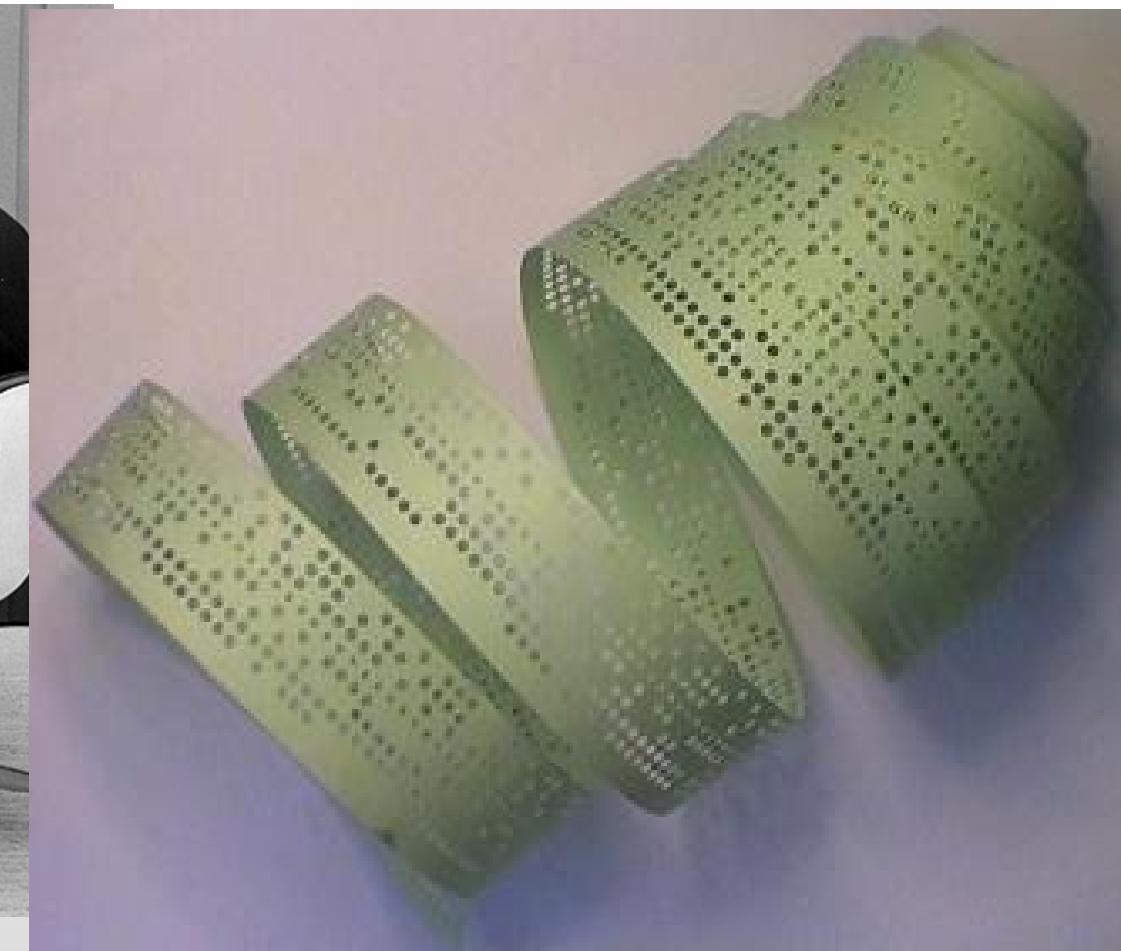
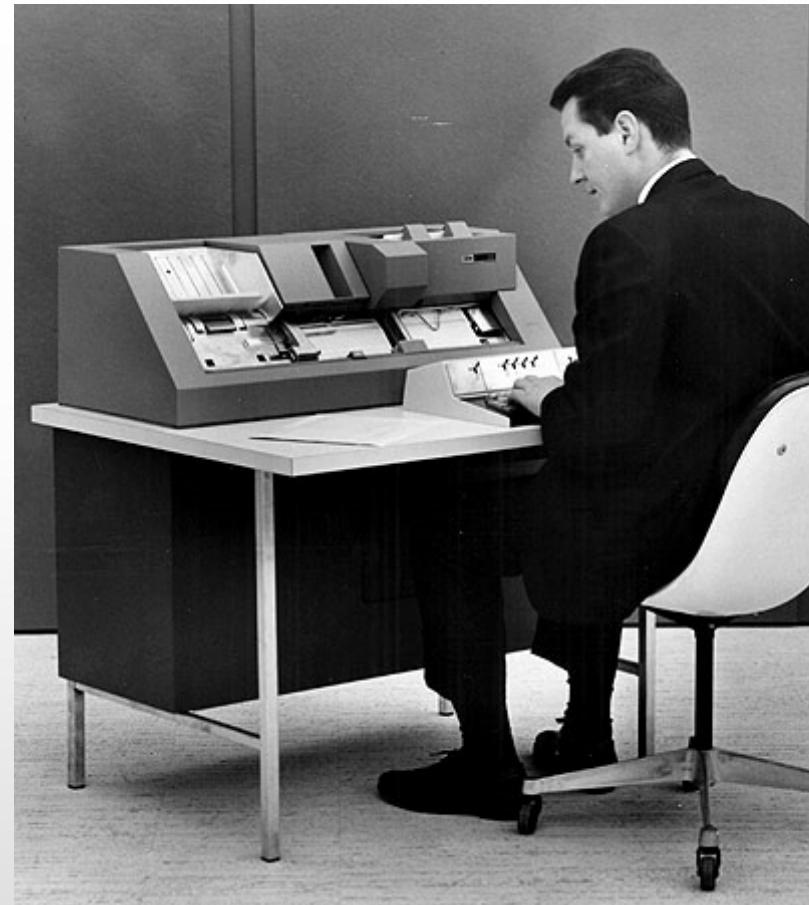
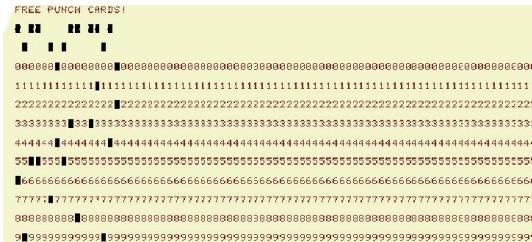
ENIAC : coding by cable connections

FREE PUNCH CARDS!





Operating System History



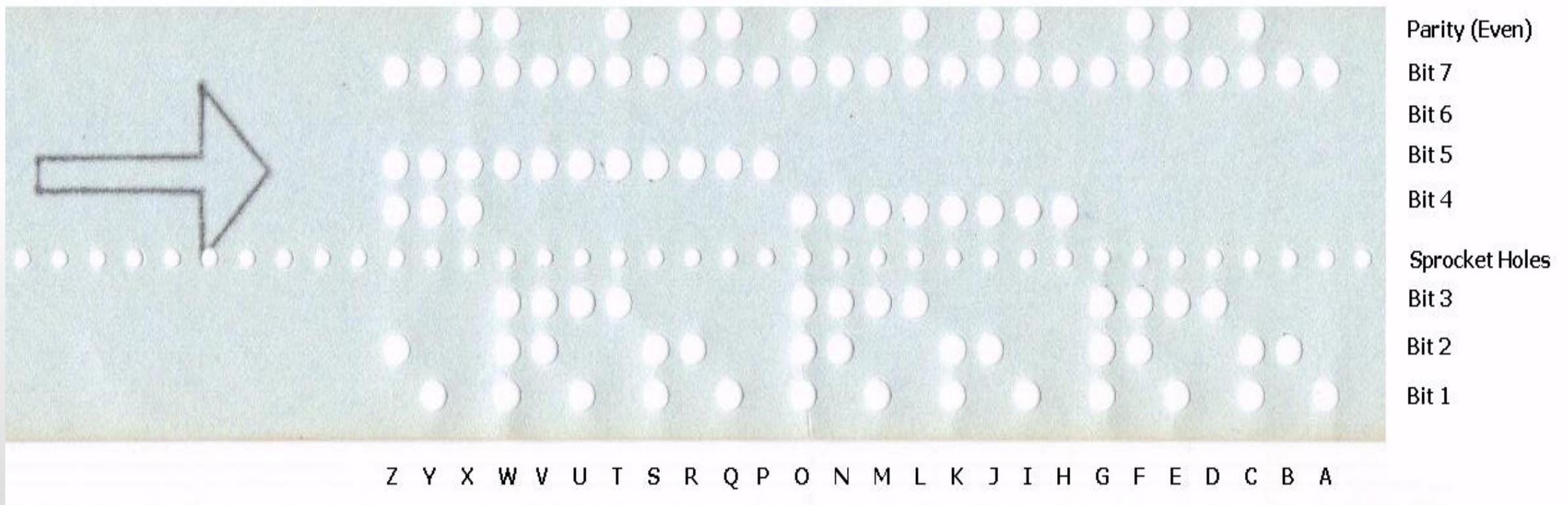
Paper tape





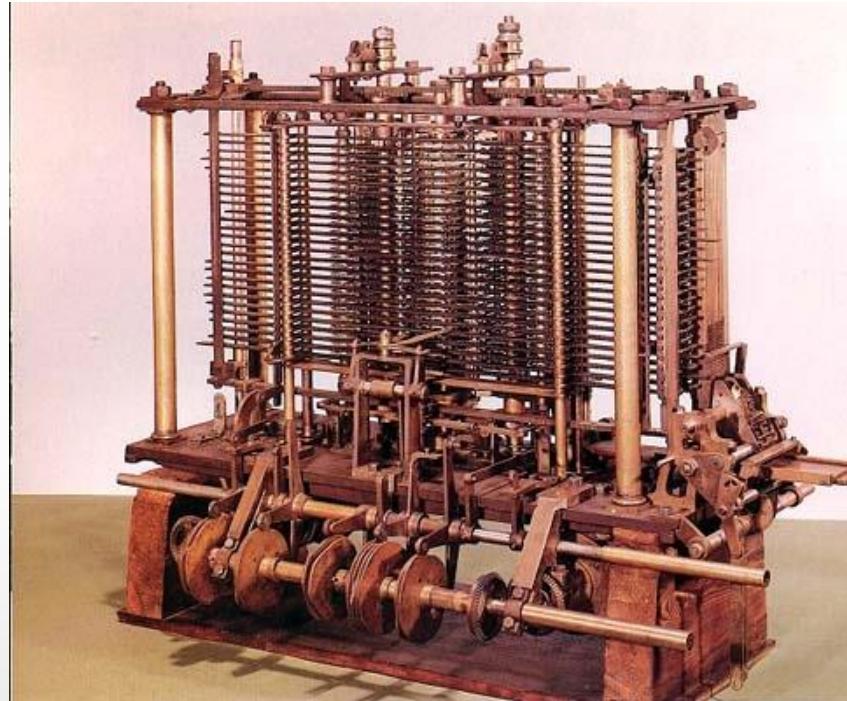
Operating System History

Punched Paper Tape 25.4 mm wide. Ascii 7-bit character code. Even Parity.





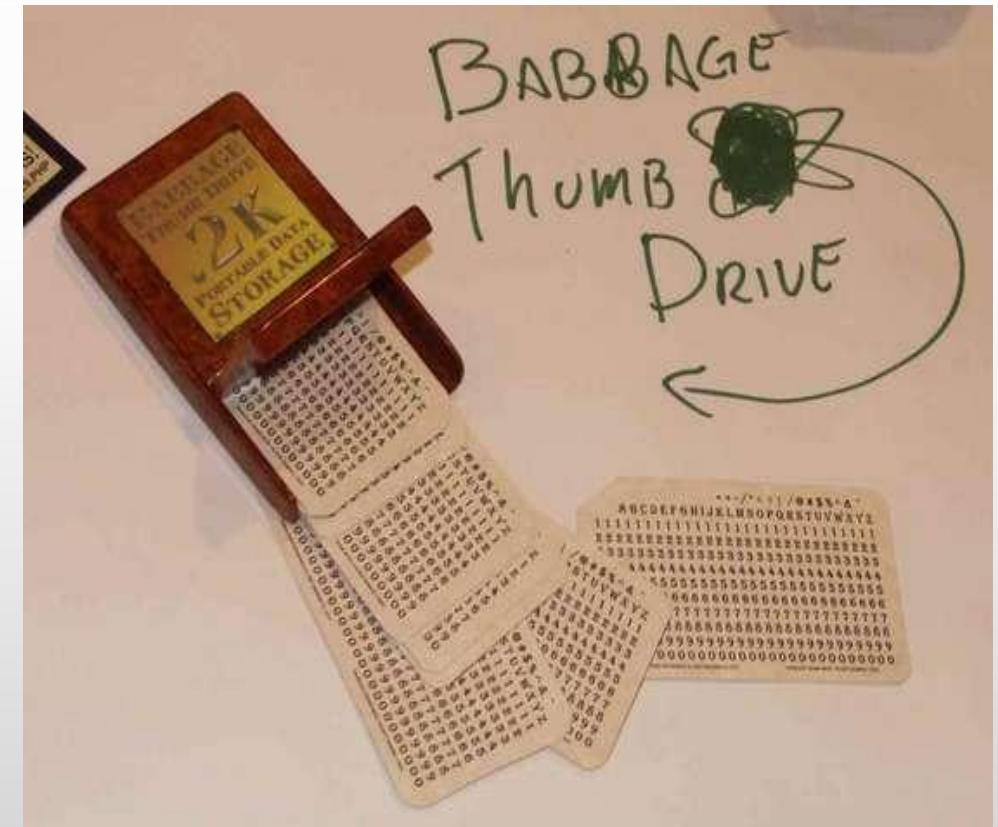
Operating System History



Babbage's analytical engine
(designed in 1840's by Charles Babbage, but could not be constructed by him.
An earlier and simpler version is constructed in 2002, in London)

<http://www.computerhistory.org/babbage/>

Ada Lovelace (at time of Charles Babbage)
wrote code for analytical engine to compute
Bernulli Numbers





Operating System History

As time went on, card readers, printers, and magnetic tape units were developed as additional hardware elements.

Assemblers, loaders and simple utility libraries were developed as software tools.
Later, off-line spooling and channel program methods were developed sequentially.



Commodore PET, 1977





Operating System History

Finally, the idea of **multiprogramming** came.

Multiprogramming means sharing of resources between more than one processes.

By multiprogramming the CPU time is not wasted, because, while one process moves on some I/O work, the OS picks another process to execute till the current one passes to I/O operation.





Operating System History

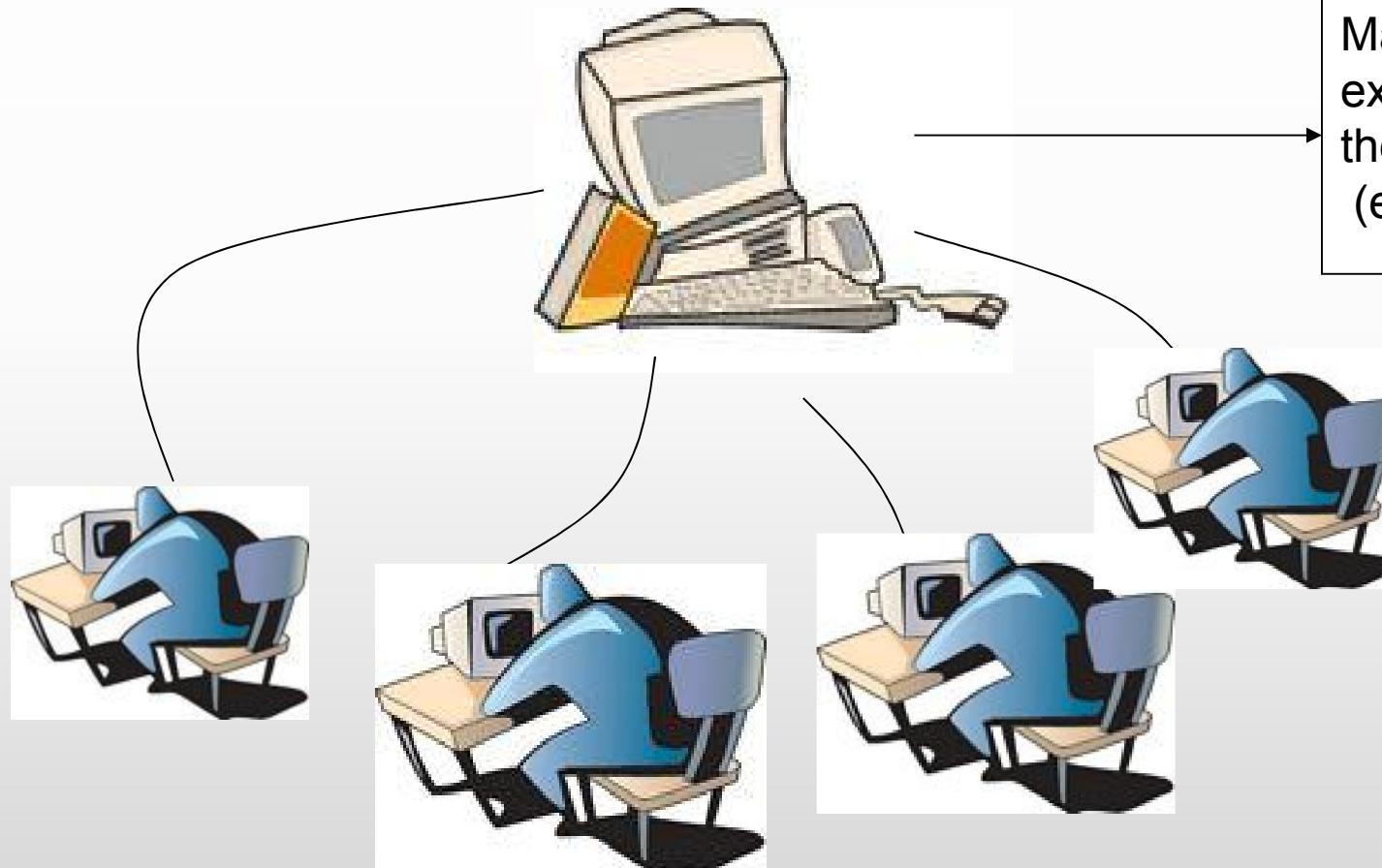
With the development of interactive computation in 1970s, **time-sharing systems** emerged.

In these systems, multiple users have *terminals* (not computers) connected to a *main computer* and execute her task in the main computer.





Operating System History



Main computer; having a CPU executing processes by utilization of the OS,
(e.g. UNIX).

Terminals are connected to the main computer and used for input and output.
No processing is made.
They do not have CPUs.



Operating System History

Another computer system is the **multiprocessor system** having multiple processors sharing memory and peripheral devices.

With this configuration, they have greater computing power and higher reliability.

Multiprocessor systems are classified into two as tightly-coupled and loosely-coupled (distributed).





Operating System History

In the tightly-coupled one, each processor is assigned a specific duty but processors work in close association, possibly sharing the same memory.

In the loosely coupled one, each processor has its own memory and copy of the OS.





Operating System History

Use of the networks required OSs appropriate for them.

In **network systems**, each process runs on its own machine but the OS have access to other machines.

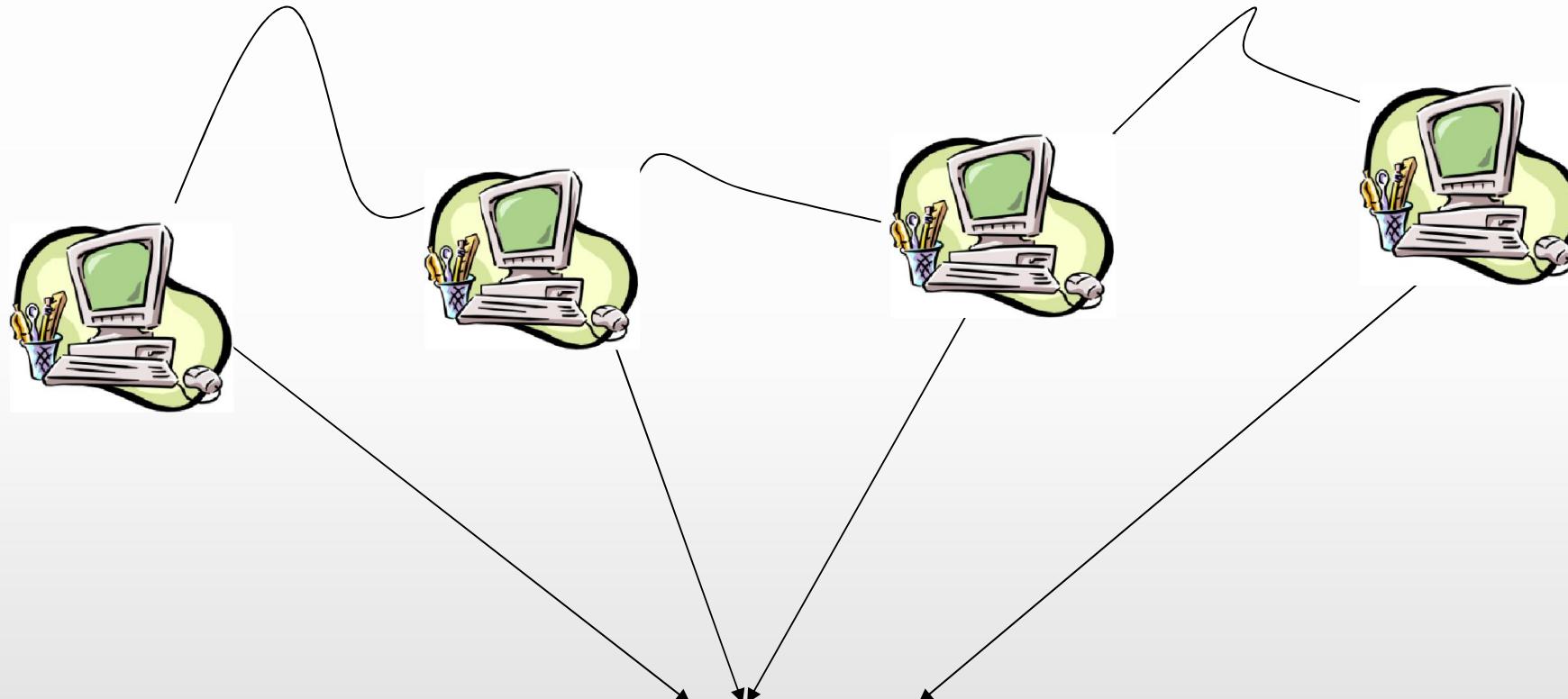
By this way, file sharing, messaging, etc. became possible.

In networks, users are aware of the fact that s/he is working in a network and when information is exchanged. The user explicitly handles the transfer of information.





Operating System History



Each is a computer having its own CPU, RAM, etc. An OS supporting networks is installed on them.





Operating System History

Distributed systems are similar to networks. However in such systems, there is no need to exchange information explicitly, it is handled by the OS itself whenever necessary.

With continuing innovations, new architectures and compatible OSs were developed.





Operating Systems Summary

In summary, OS is an interface between users and hardware - an environment "architecture"

Allows convenient usage; hides the tedious stuff

Allows efficient usage; parallel activity, avoids wasted cycles





Operating Systems Summary

Provides information protection

Gives each user a slice of the resources

Acts as a control program.





Thank You Very Much

Benjamin Kommey

bkommey.coe@knust.edu.gh

nii_kommey@msn.com

050 770 32 86

Skype_id: calculus.affairs





“Life is not divided into semesters.
You don't get summers off and very
few employers are interested in
helping you FIND YOURSELF.
Do that on your own time”

Bill Gates





Why Operating Systems?

We need a mechanism for scheduling jobs or processes.

Scheduling can be as simple as running the next process, or it can use relatively complex rules to pick a running process.

We need a method for simultaneous CPU execution and IO handling.

Processing is going on even as IO is occurring in preparation for future CPU work.

Off Line Processing; not only are IO and CPU happening concurrently, but some off-board processing is occurring with the IO.





Why Operating Systems?

The CPU is wasted if a job waits for I/O. This leads to:

Multiprogramming (dynamic switching). While one job waits for a resource, the CPU can find another job to run. It means that several jobs are ready to run and only need the CPU in order to continue.

There is therefore the need for:

memory management
resource scheduling
deadlock protection

which are the subject of the rest of this course.



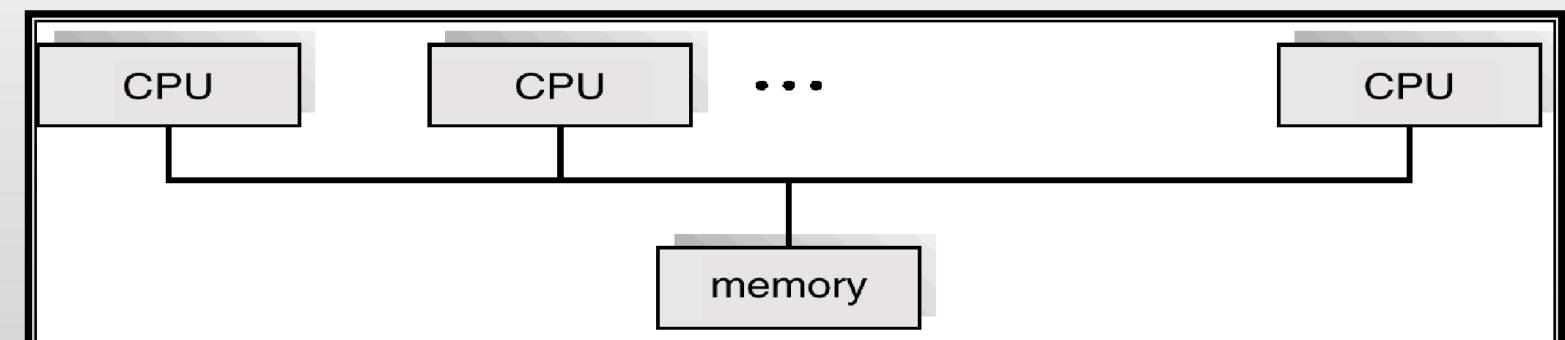


Why Operating Systems?

Other Characteristics include:

Time Sharing - multiprogramming environment that's also interactive.

Multiprocessing - Tightly coupled systems that communicate via shared memory. Used for scientific applications. Used for speed improvement by putting together a number of off-the-shelf processors.

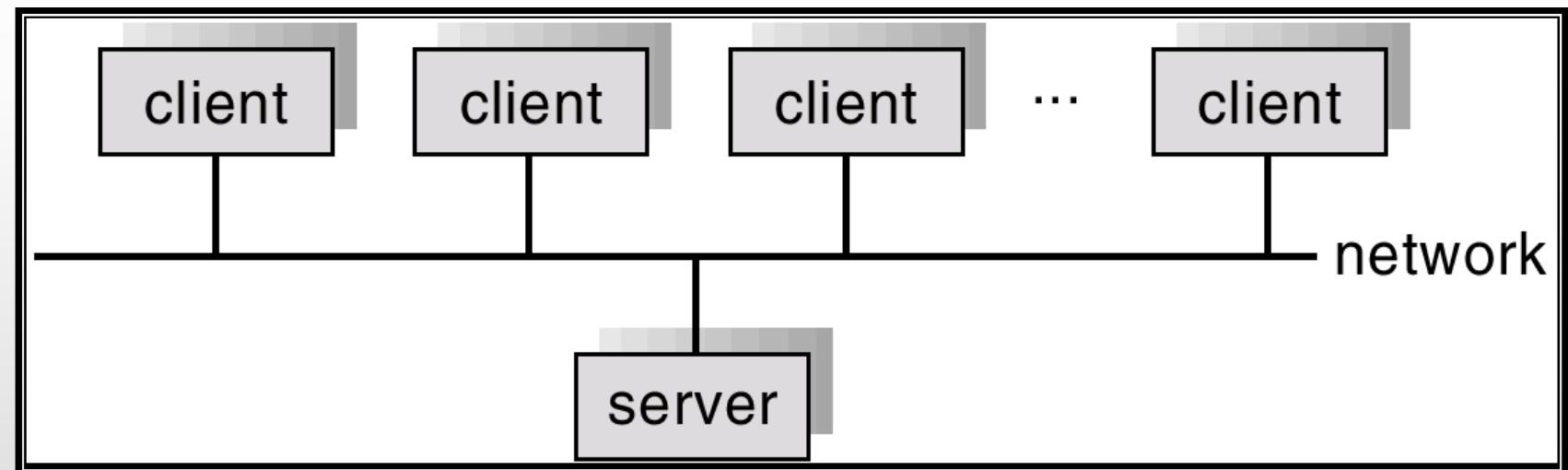




Why Operating Systems?

Other Characteristics include:

Distributed Systems - Loosely coupled systems that communicate via message passing. Advantages include resource sharing, speed up, reliability, communication.



Real Time Systems - Rapid response time is main characteristic. Used in control of applications where rapid response to a stimulus is essential.



Why Operating Systems?

Interrupts:

Interrupt transfers control to the interrupt service routine generally, through the *interrupt vector*, which contains the addresses of all the service routines.

Interrupt architecture must save the address of the interrupted instruction.

Incoming interrupts are *disabled* while another interrupt is being processed to prevent a *lost interrupt*.



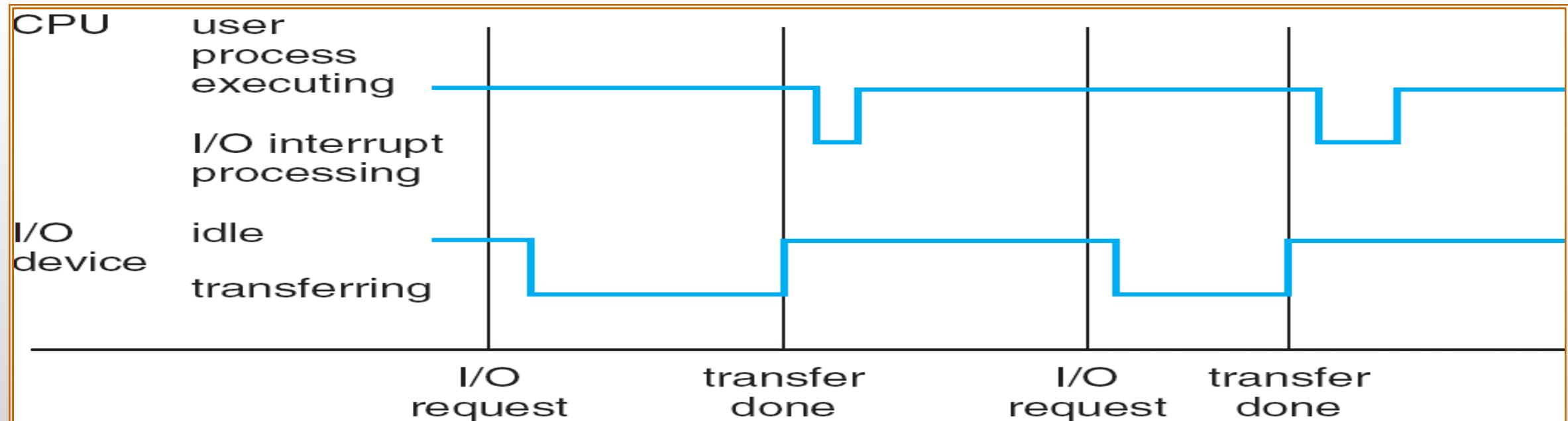


Why Operating Systems?

Interrupts:

A *trap* is a software-generated interrupt caused either by an error or a user request.

An operating system is *interrupt driven*.



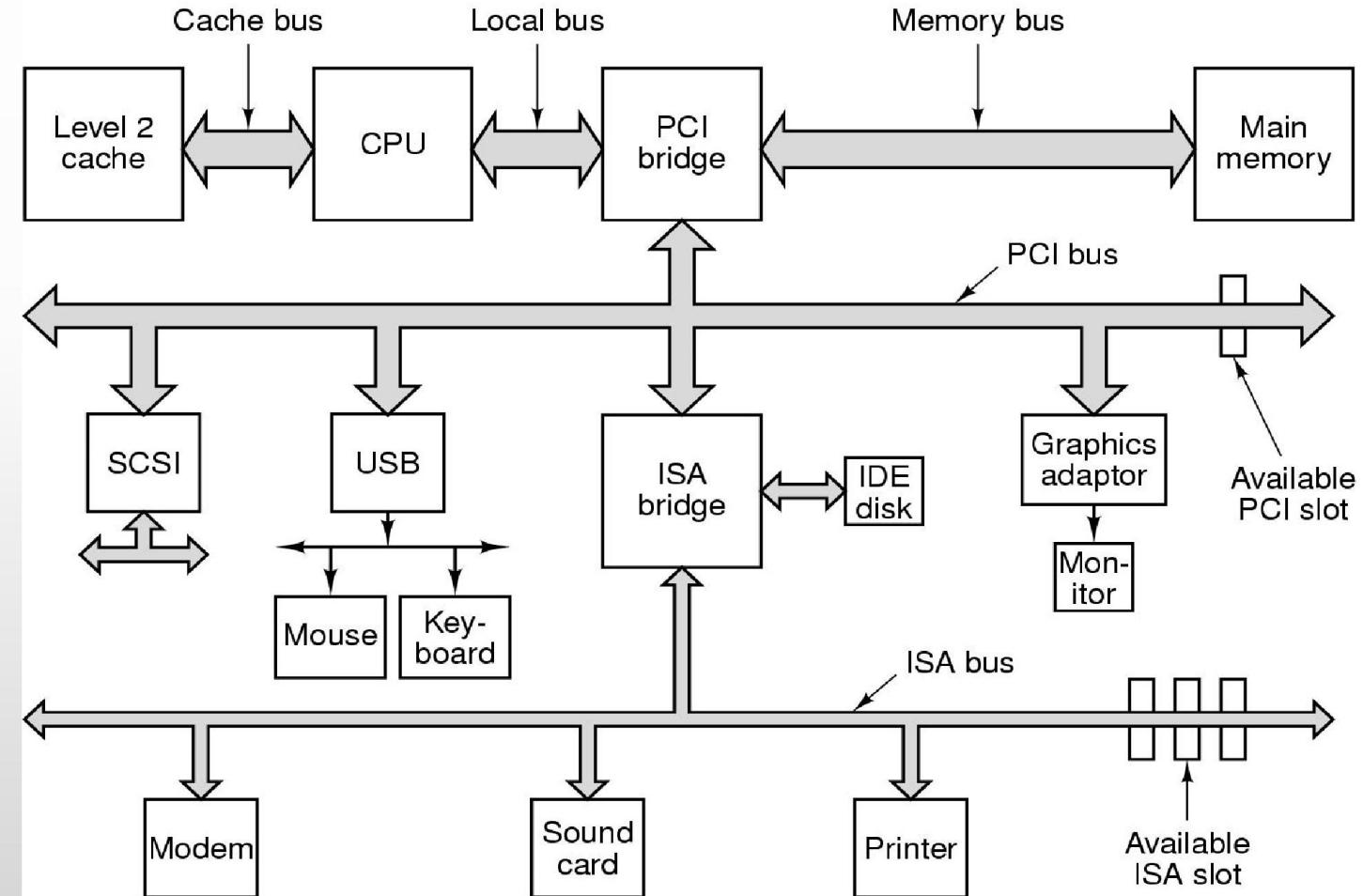


Why Operating Systems?

Hardware Support

These are the devices that make up a typical system.

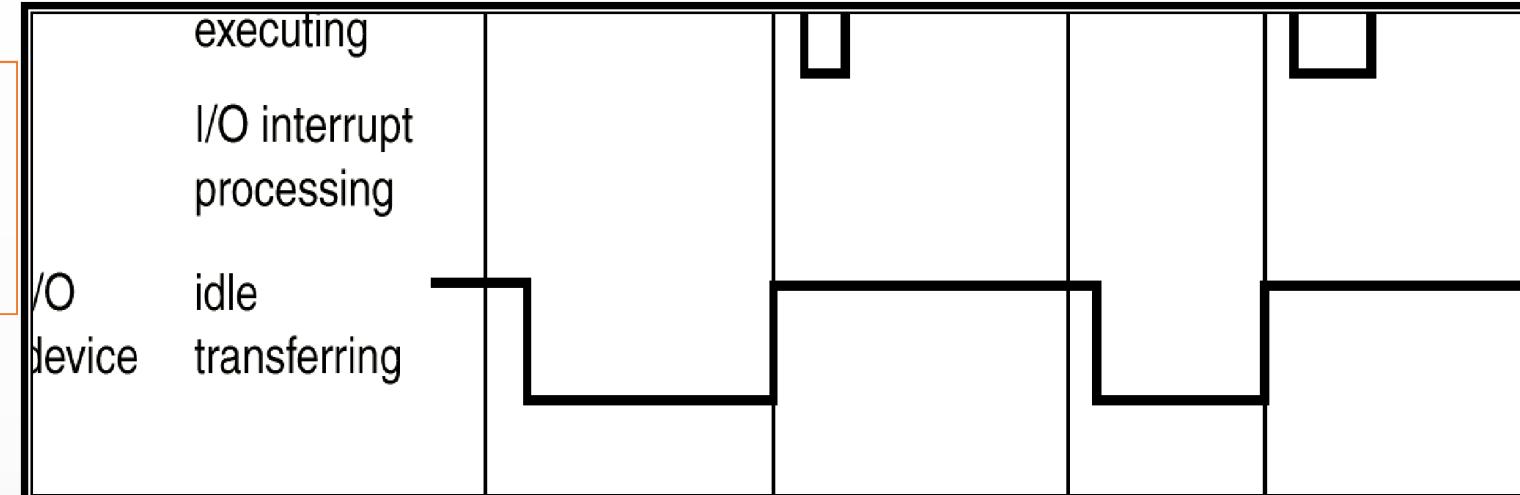
Any of these devices can cause an electrical interrupt that grabs the attention of the CPU.



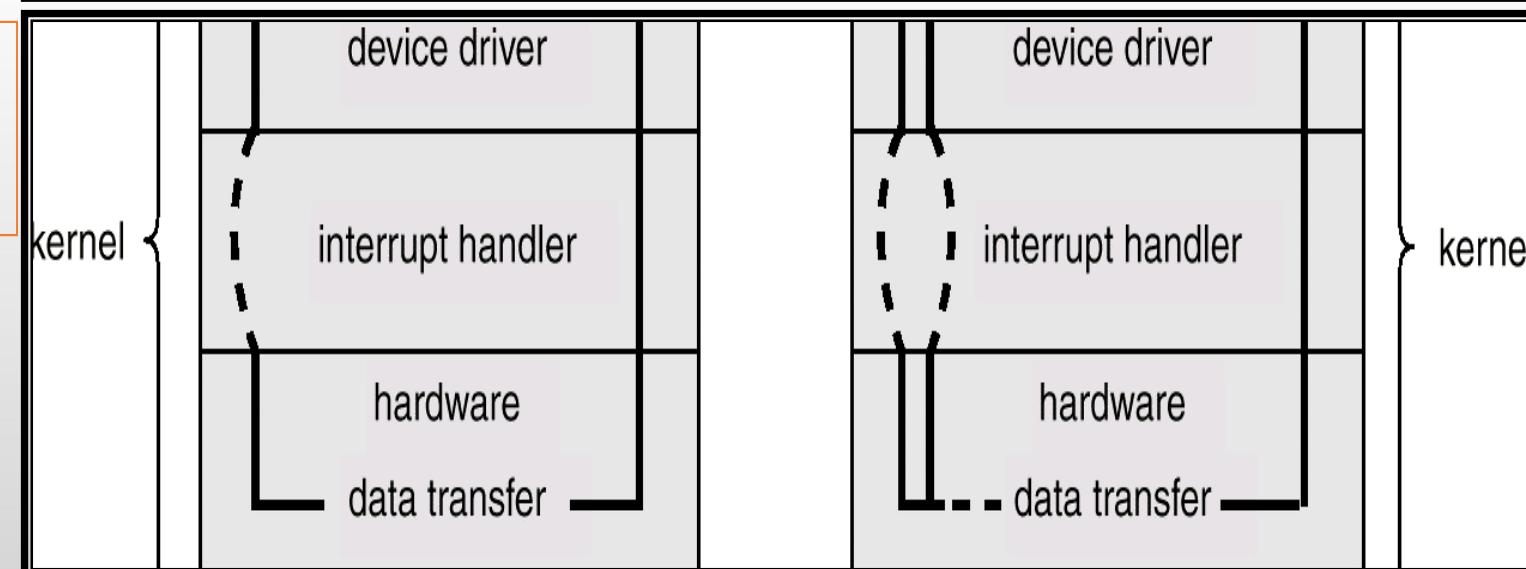


Why Operating Systems?

Sequence of events for processing an IO request.

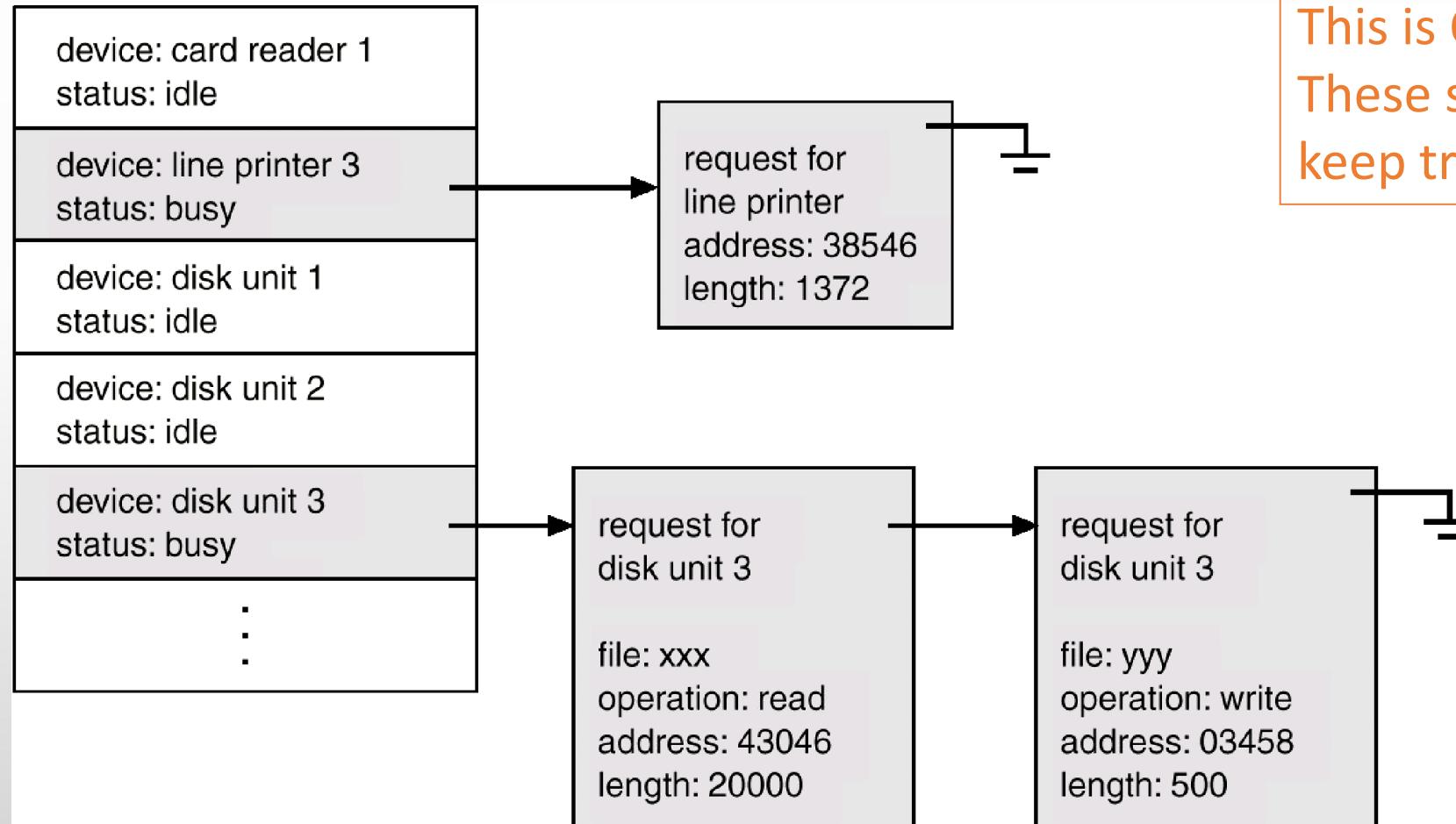


Comparing Synchronous and Asynchronous IO Operations





Why Operating Systems?



This is O.S. Bookkeeping.
These structures are necessary to keep track of IO in progress.



Why Operating Systems?

Storage Hierarchy

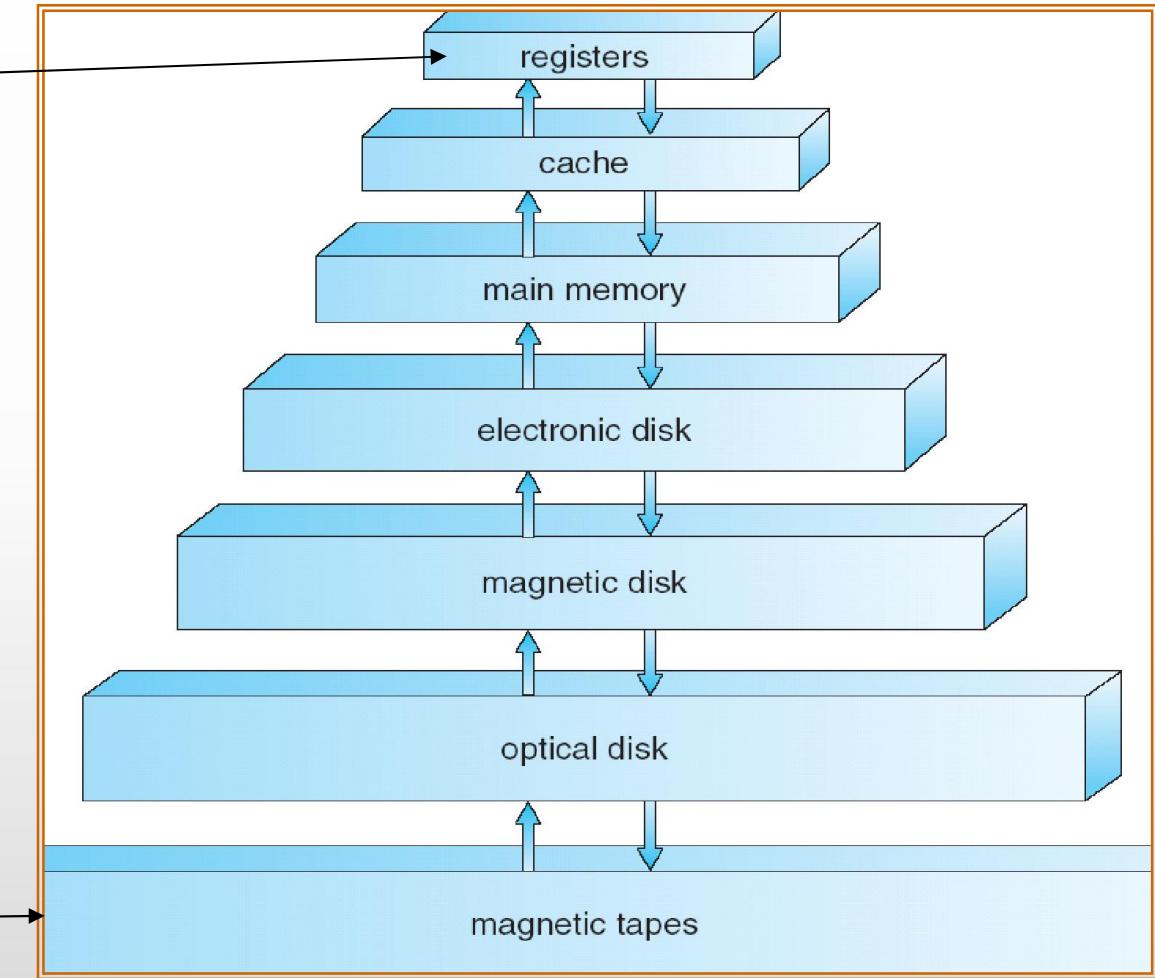
Very fast storage is very expensive.

So the Operating System manages a hierarchy of storage devices in order to make the best use of resources.

In fact, **considerable** effort goes into this support.

Fast and Expensive

Slow an Cheap



Why Operating Systems?

Performance:

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape





Why Operating Systems?

Caching:

Important principle, performed at many levels in a computer
(in hardware, operating system, software)

Information in use copied from slower to faster storage temporarily

Faster storage (cache) checked first to determine if information is there

- If it is, information used directly from the cache (fast)
- If not, data copied to cache and used there

Cache smaller than storage being cached

Cache management important design problem

Cache size and replacement policy





Overview

Operating System Structure

System Components

System Calls

How Components Fit

Virtual Machine





Operating System Structure

System Components

**Process
Management**

**Main Memory
Management**

File Management

**I/O System
Management**





Operating System Structure

System Components

Secondary Storage Management

Networking

Protection System

Command-Interpreter System





System Components

Process Management

- A **process** is a **program** in execution: (A program is passive, a process active.)
- A process has resources (CPU time, files) and attributes that must be managed.



System Components

Management of processes includes

- **Process Scheduling (priority, time management, . .)**
- **Creation/termination**
- **Block/Unblock (suspension/resumption)**
- **Synchronization**
- **Communication**
- **Deadlock handling**
- **Debugging**



System Components

Main Memory Management

- Allocation/de-allocation for processes, files, I/O.
- Maintenance of several processes at a time
- Keep track of who's using what memory
- Movement of process memory to/from secondary storage





System Components

File Management

A file is a collection of related information defined by its creator

- Commonly, files represent programs (both source and object forms) and data.

OS

is responsible for the following activities in connections with file management

- **File creation and deletion**
- **Directory creation and deletion**
- **Support of primitives for manipulating files and directories**
- **Mapping files onto secondary storage**
- **File backup on stable (nonvolatile) storage media.**





System Components

IO Management

- Buffer caching system
- Generic device driver code
- Drivers for each device - translate read/write requests into disk position commands.





System Components

Secondary Storage Management

- Disks, tapes, optical, ...
- Free space management (paging/swapping)
- Storage allocation (what data goes where on disk)
- Disk scheduling





System Components

Networking

- Communication system between distributed processors
- Getting information about files/processes/etc. on a remote machine
- Can use either a message passing or a shared memory model





System Components

Protection

- Of files, memory, CPU, etc.
- Means controlling of access
- Depends on the attributes of the file and user





System Components

Command Interpreter System

- Command Interpreters - Program that accepts control statements (shell, GUI interface, etc.)
- Compilers/ linkers
- Communications (ftp, telnet, etc.)

How Do These All Fit Together?

In essence, they all provide services for each other.





Operating System Tailoring

Modifying the Operating System program for a particular machine.

The goal is to include all the necessary pieces, but not too many extra ones.

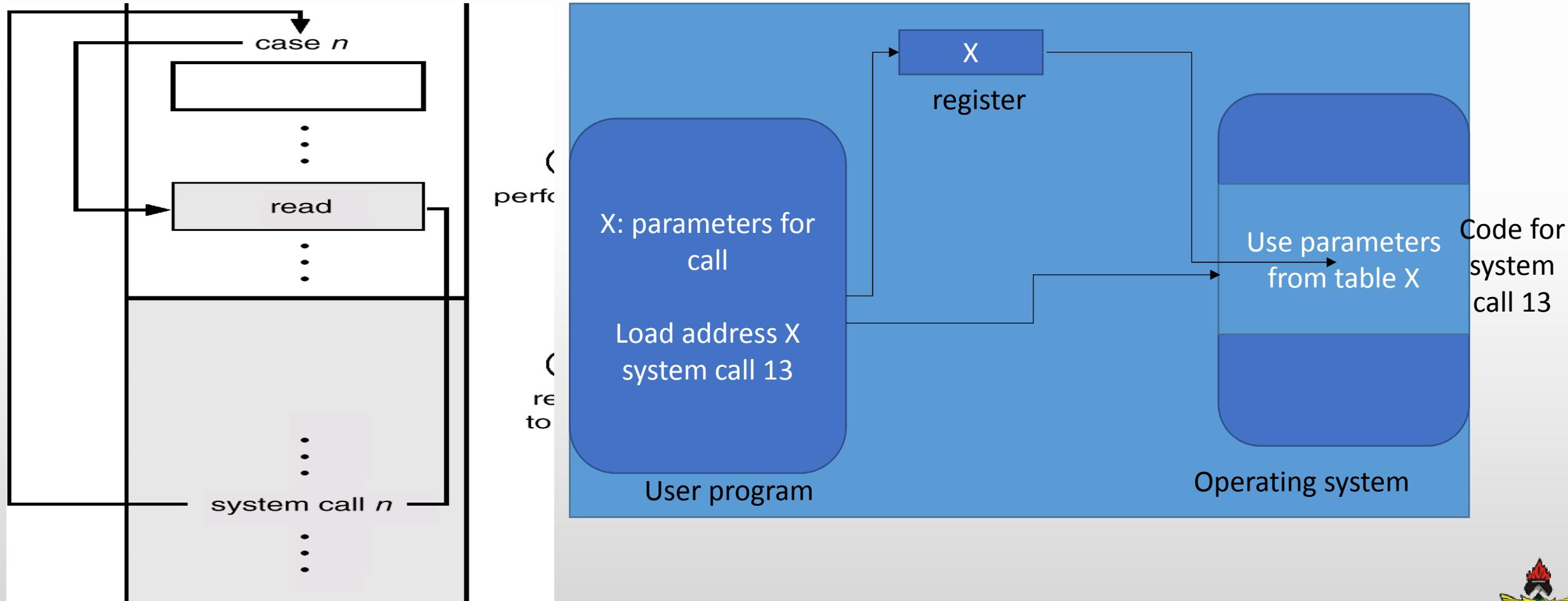
- Typically a System can support many possible devices, but any one installation has only a few of these possibilities.
- **Plug and play** allows for detection of devices and automatic inclusion of the code (drivers) necessary to drive these devices.
- A **sysgen** is usually a link of many OS routines/modules in order to produce an executable containing the code to run the drivers.



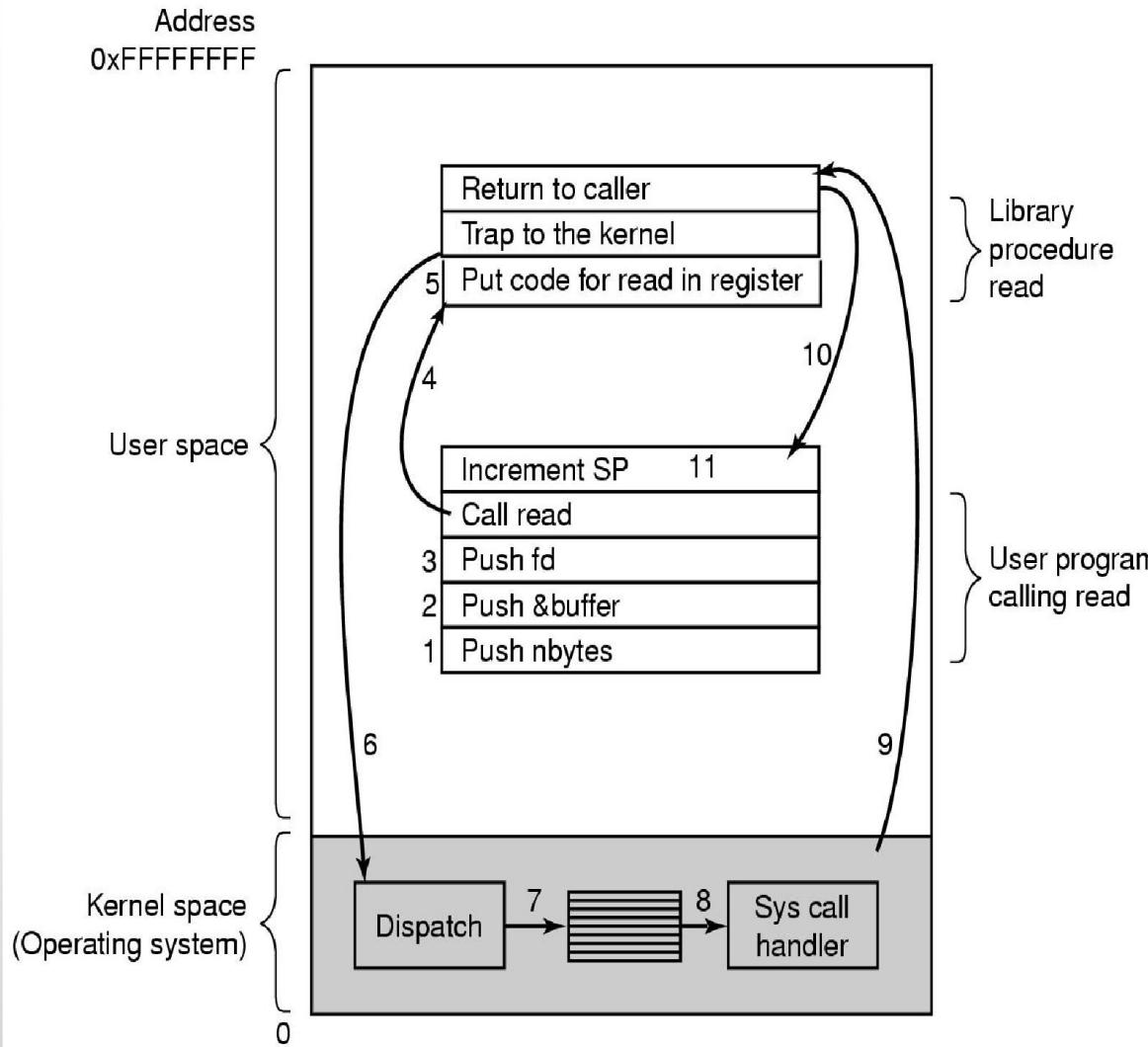


Operating System Calls

A System Call is the main way a user program interacts with the Operating System.



Operating System Calls



HOW A SYSTEM CALL WORKS

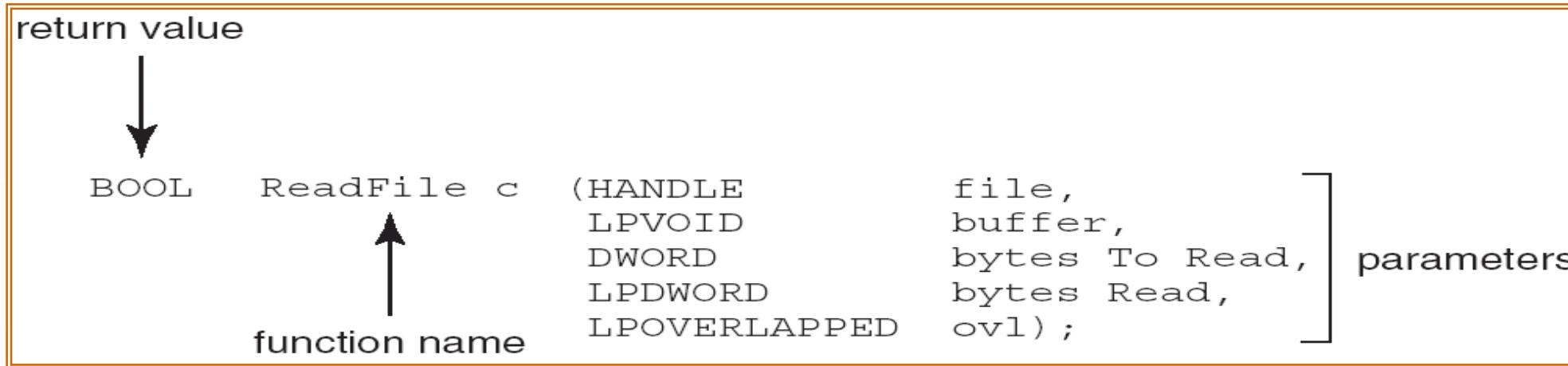
- Obtain access to system space
 - Do parameter validation
 - System resource collection (locks on structures)
 - Ask device/system for requested item
 - Suspend waiting for device
 - Interrupt makes this thread ready to run
 - Wrap-up
 - Return to user

There are 11 (or more) steps in making the system call
read (fd, buffer, nbytes) ←





System Calls – Windows API Example



Consider the `ReadFile()` function in the Win32 API—a function for reading from a file.

A description of the parameters passed to `ReadFile()`

`HANDLE file`—the file to be read

`LPVOID buffer`—a buffer where the data will be read into and written from

`DWORD bytesToRead`—the number of bytes to be read into the buffer

`LPDWORD bytesRead`—the number of bytes read during the last read

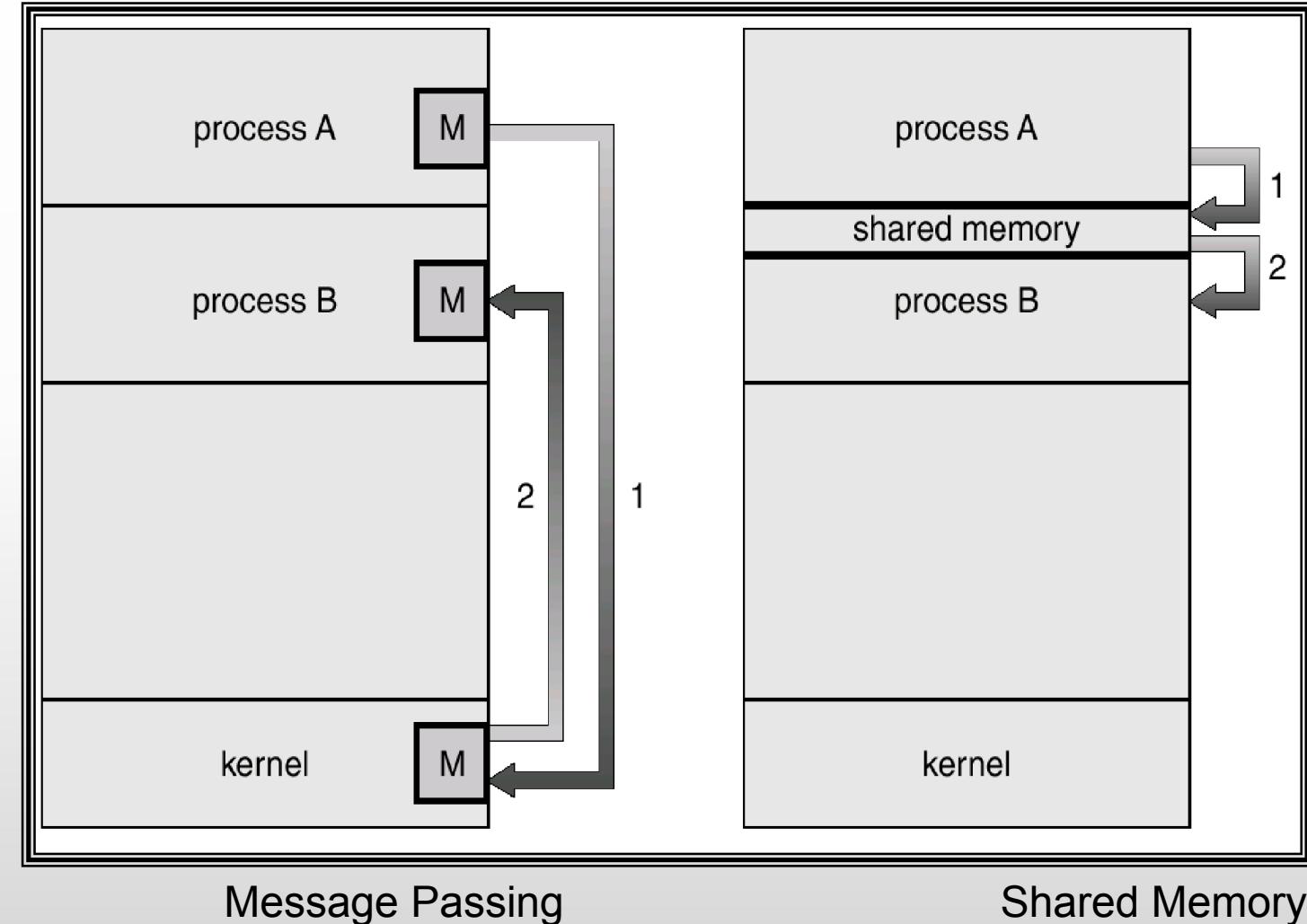
`LPOVERLAPPED ovl`—indicates if overlapped I/O is being used





Two ways of passing data
between programs.

Operating System Calls





These are examples of various system calls.

Operating 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

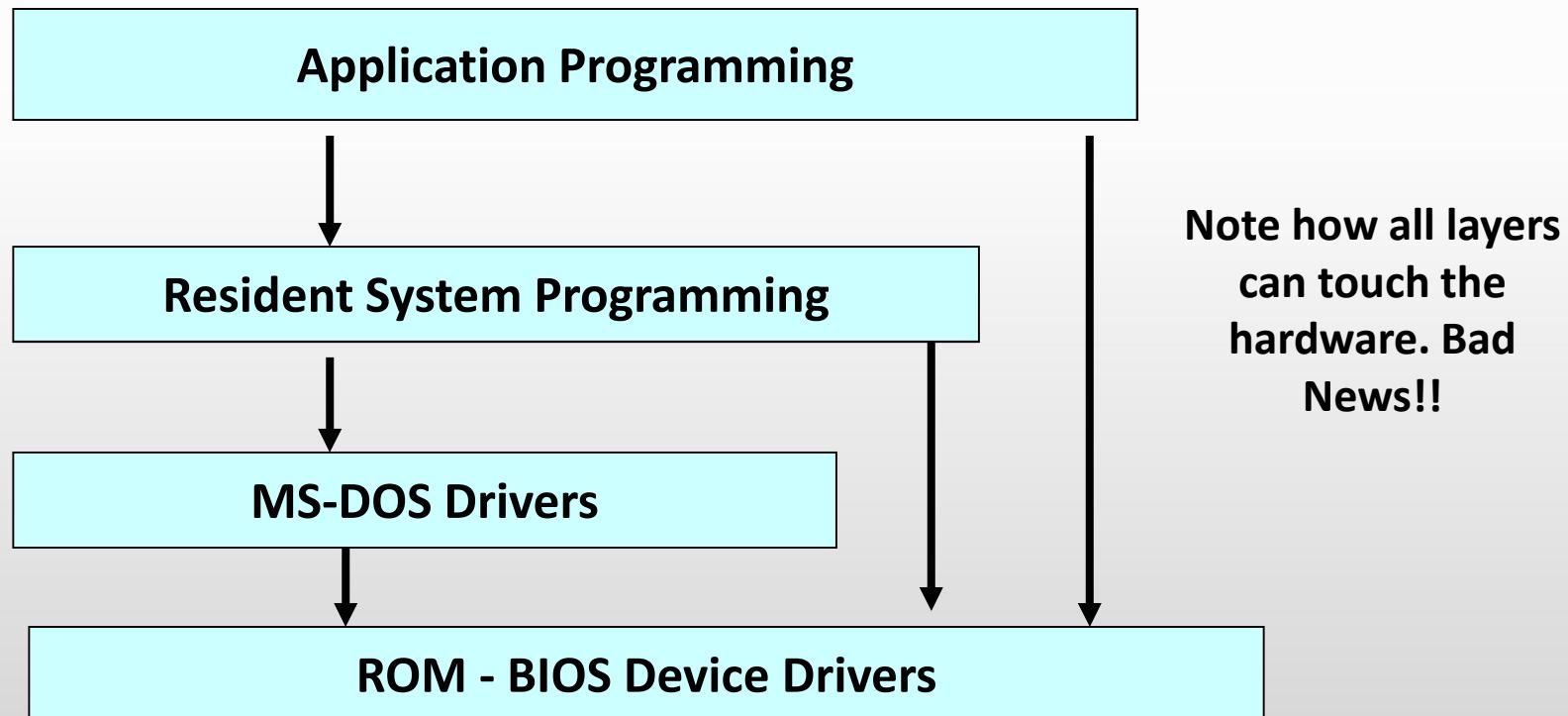




Putting Things Together

A SIMPLE STRUCTURE:

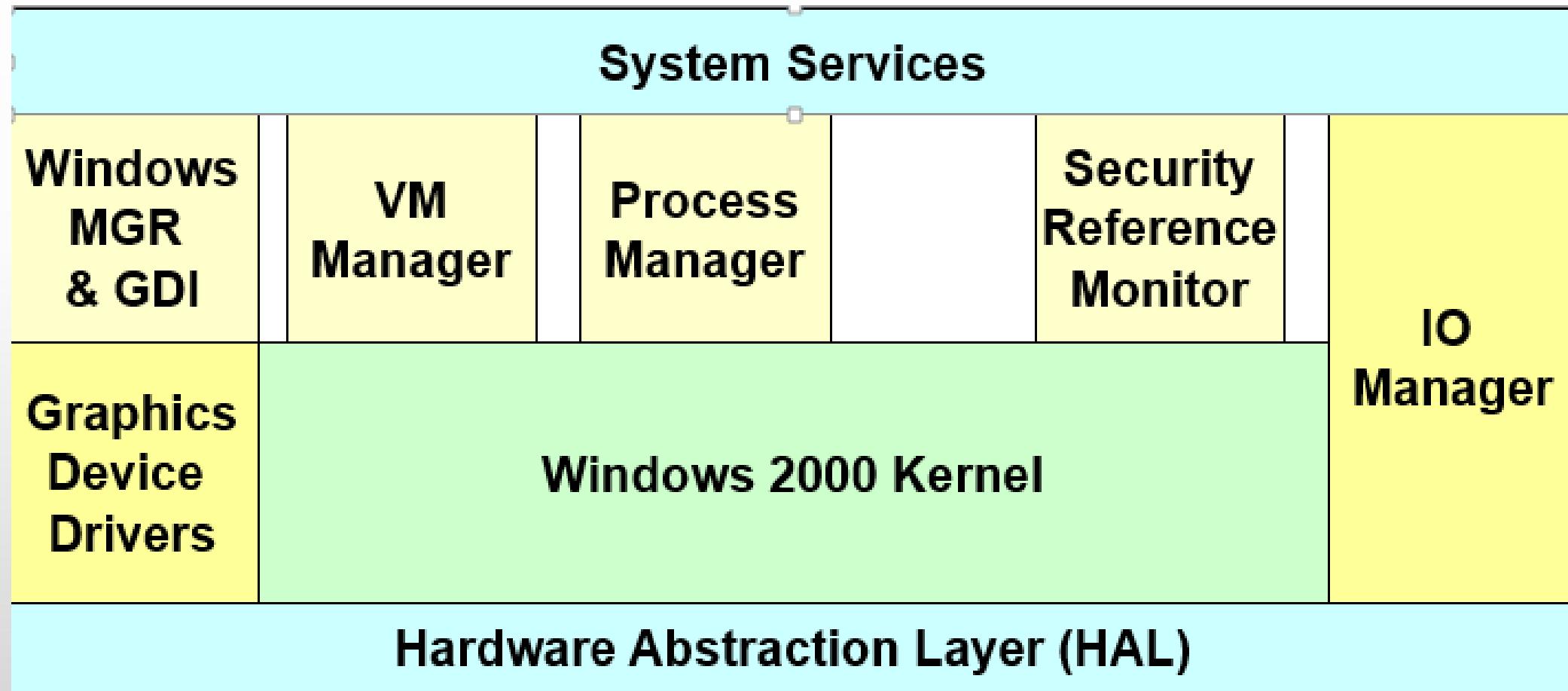
Example of MS-DOS.





Putting Things Together

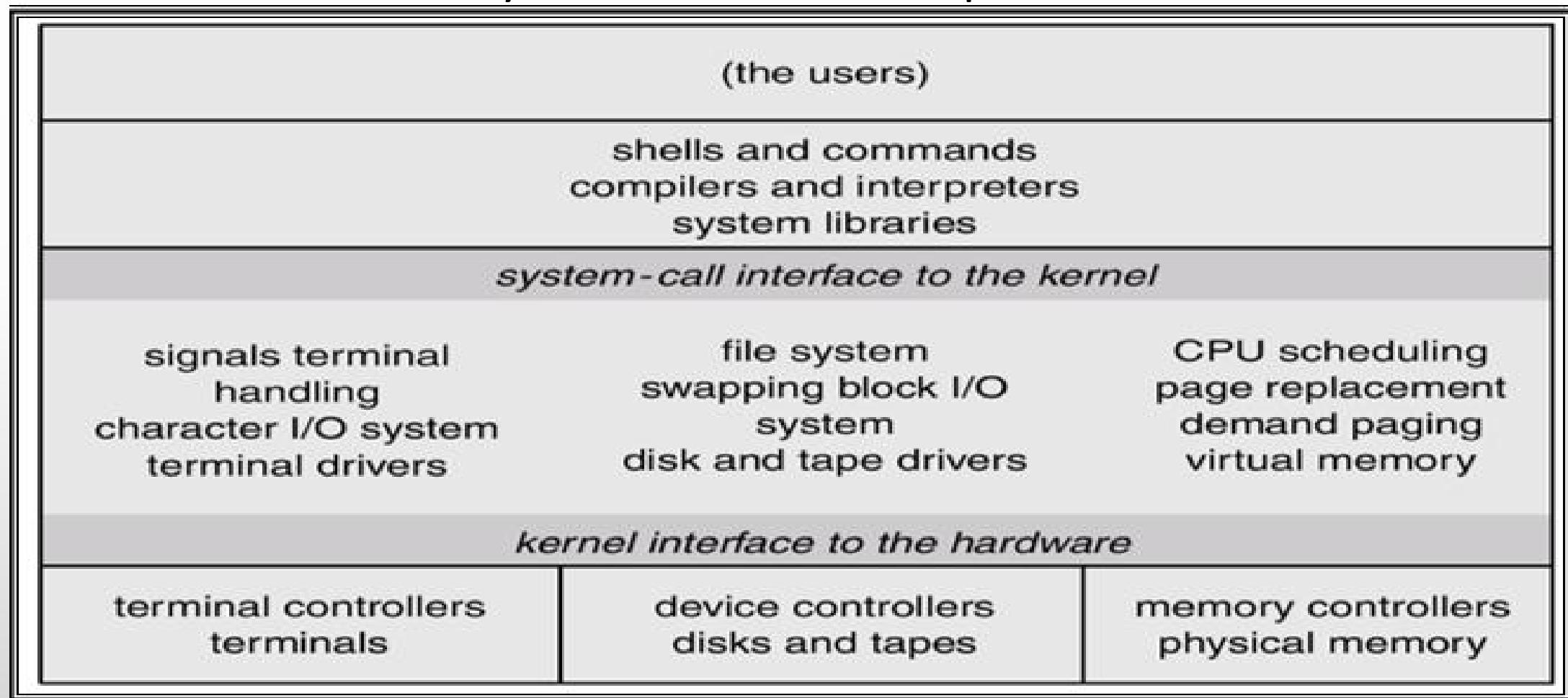
A layered structure example of Windows 2000





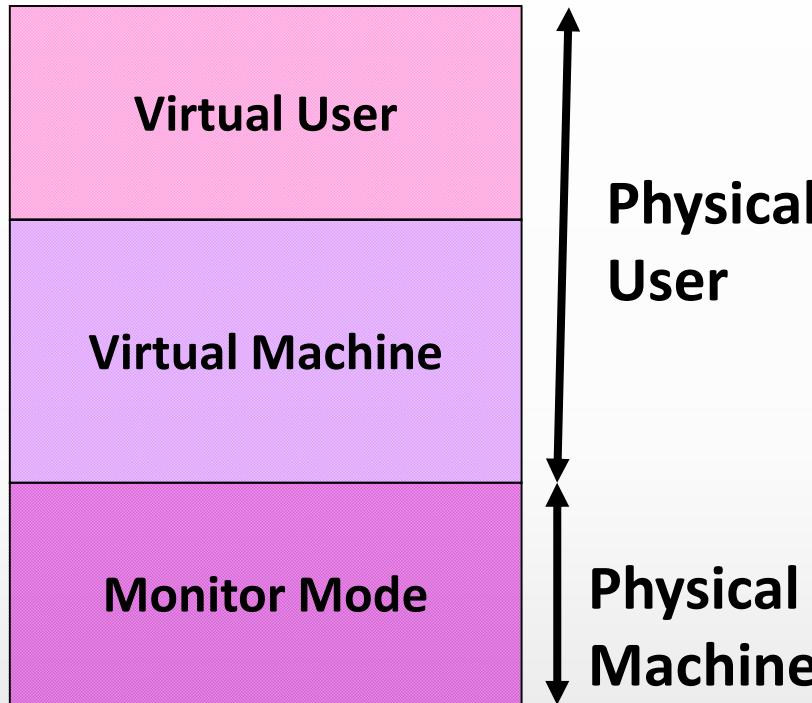
Putting Things Together

A layered structure example of Unix





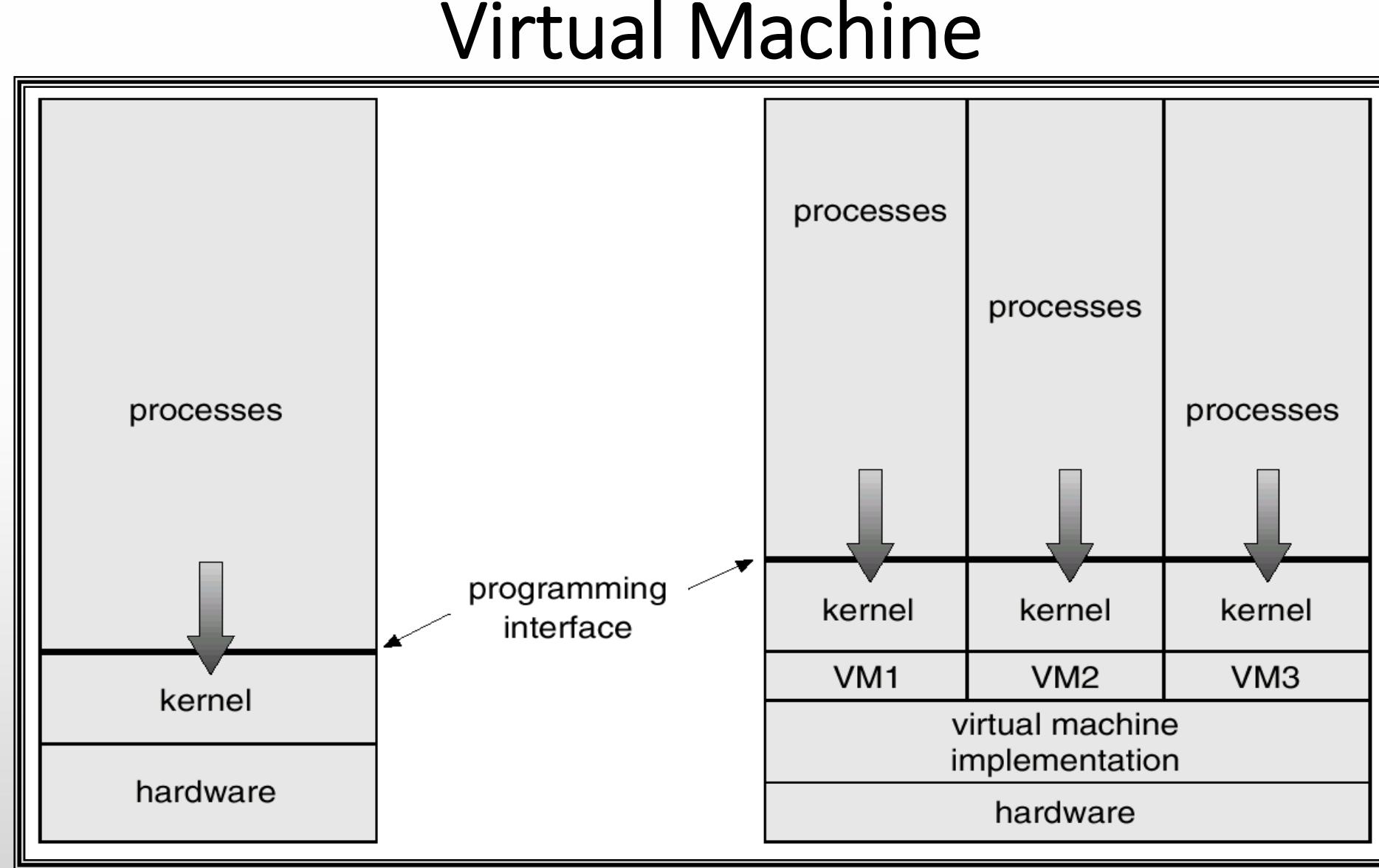
Virtual Machine



In a Virtual Machine (eg. VMware - each process "seems" to execute on its own processor with its own memory, devices, etc.

The resources of the physical machine are shared.
Virtual devices are sliced out of the physical ones.
Virtual disks are subsets of physical ones.
Useful for running different OS simultaneously on the same machine.
Protection is excellent, but no sharing possible.
Virtual privileged instructions are trapped.

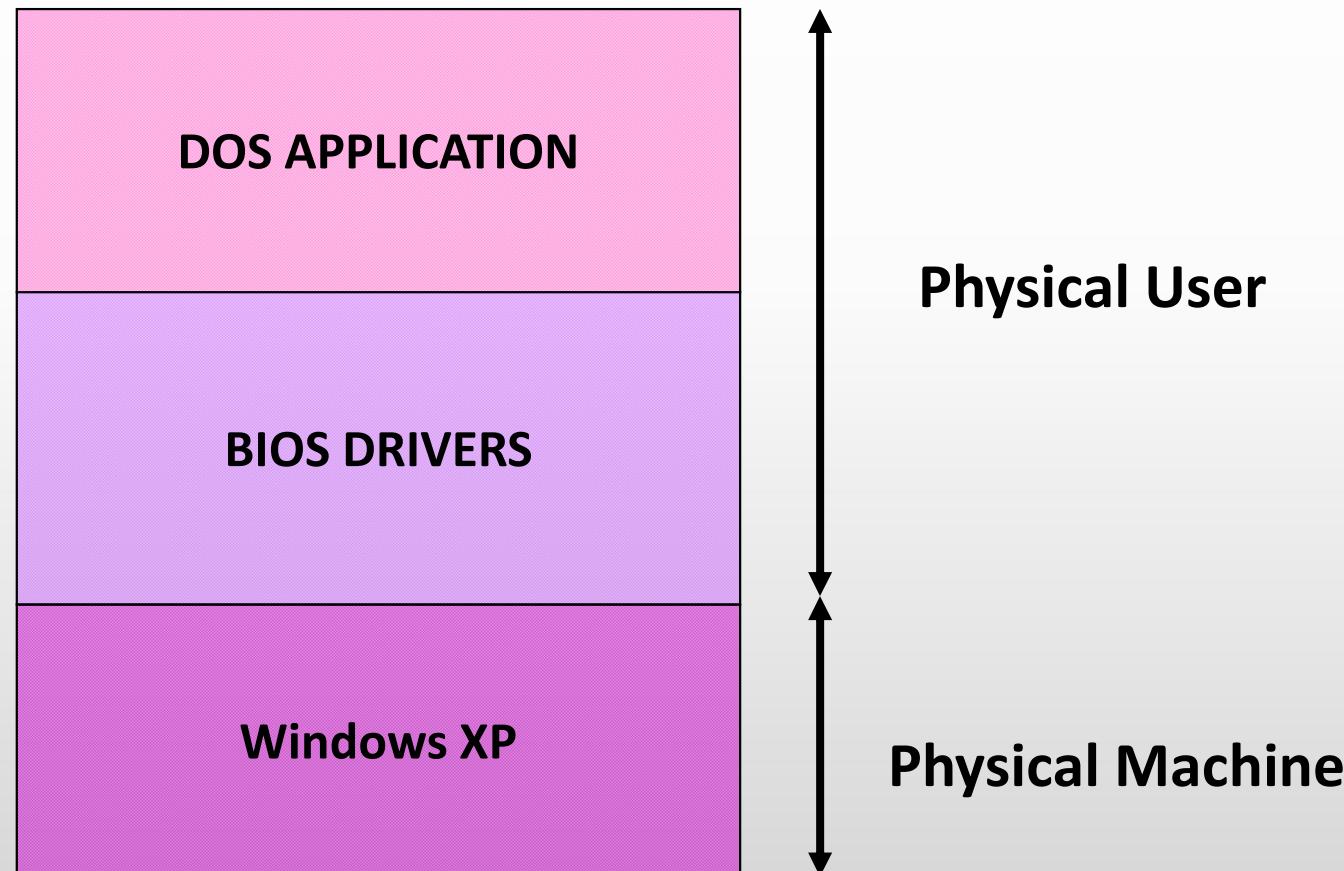






Virtual Machine

Example of MS-DOS on top of Windows XP.

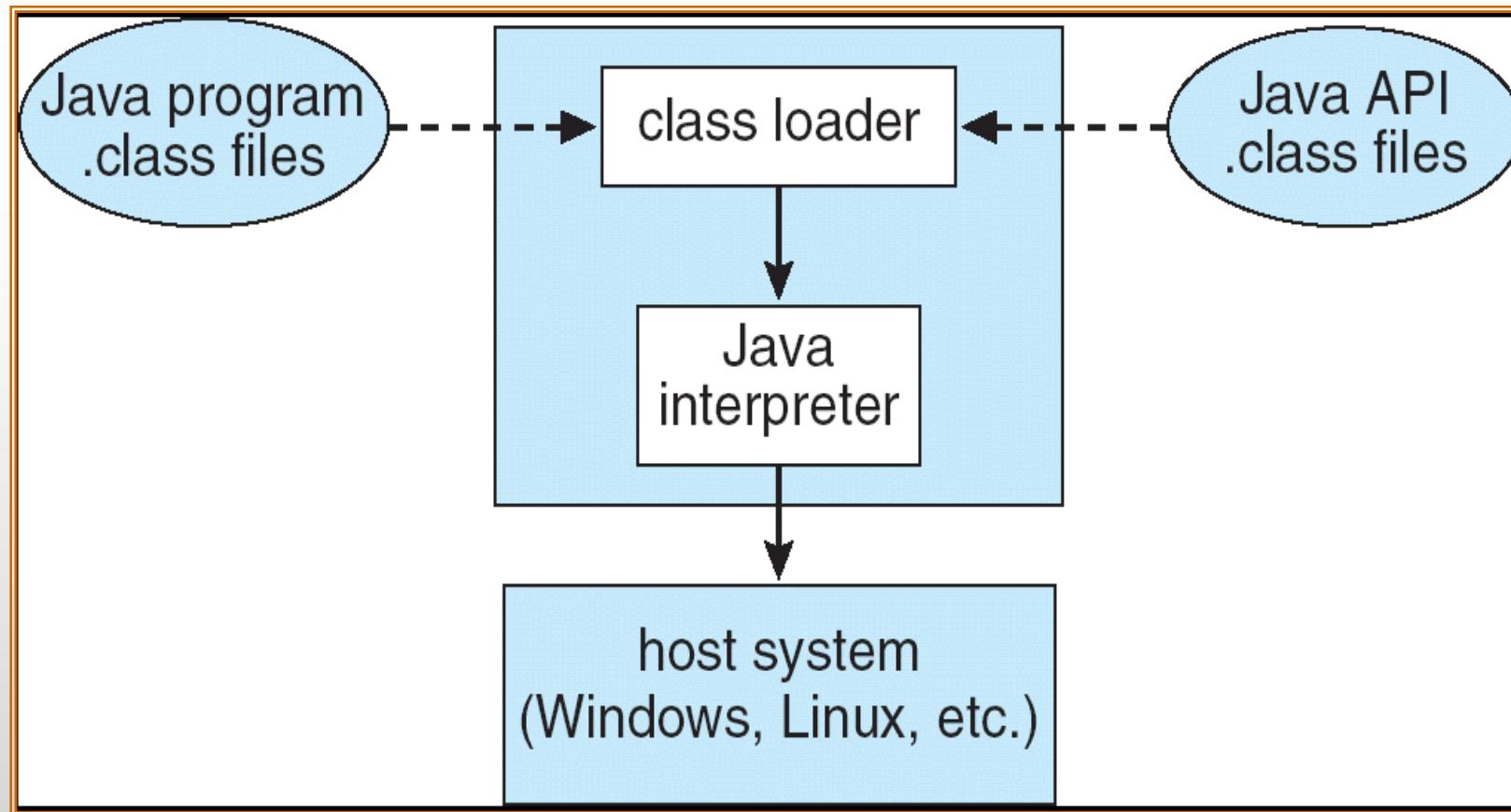




Virtual Machine

Example of Java Virtual Machine

The Java Virtual Machine allows Java code to be portable between various hardware and OS platforms.





Thank You Very Much

Benjamin Kommey

bkommey.coe@knust.edu.gh

nii_kommey@msn.com

050 770 32 86

Skype_id: calculus.affairs





“ I studied every thing but never topped.... But today the toppers of the best universities are my employees ”

Bill Gates





Overview

Processes





Processes

Process Concept

- A **program** is passive; a **process** active

Process Attributes

- hardware state,
- memory,
- CPU,
- progress (executing)

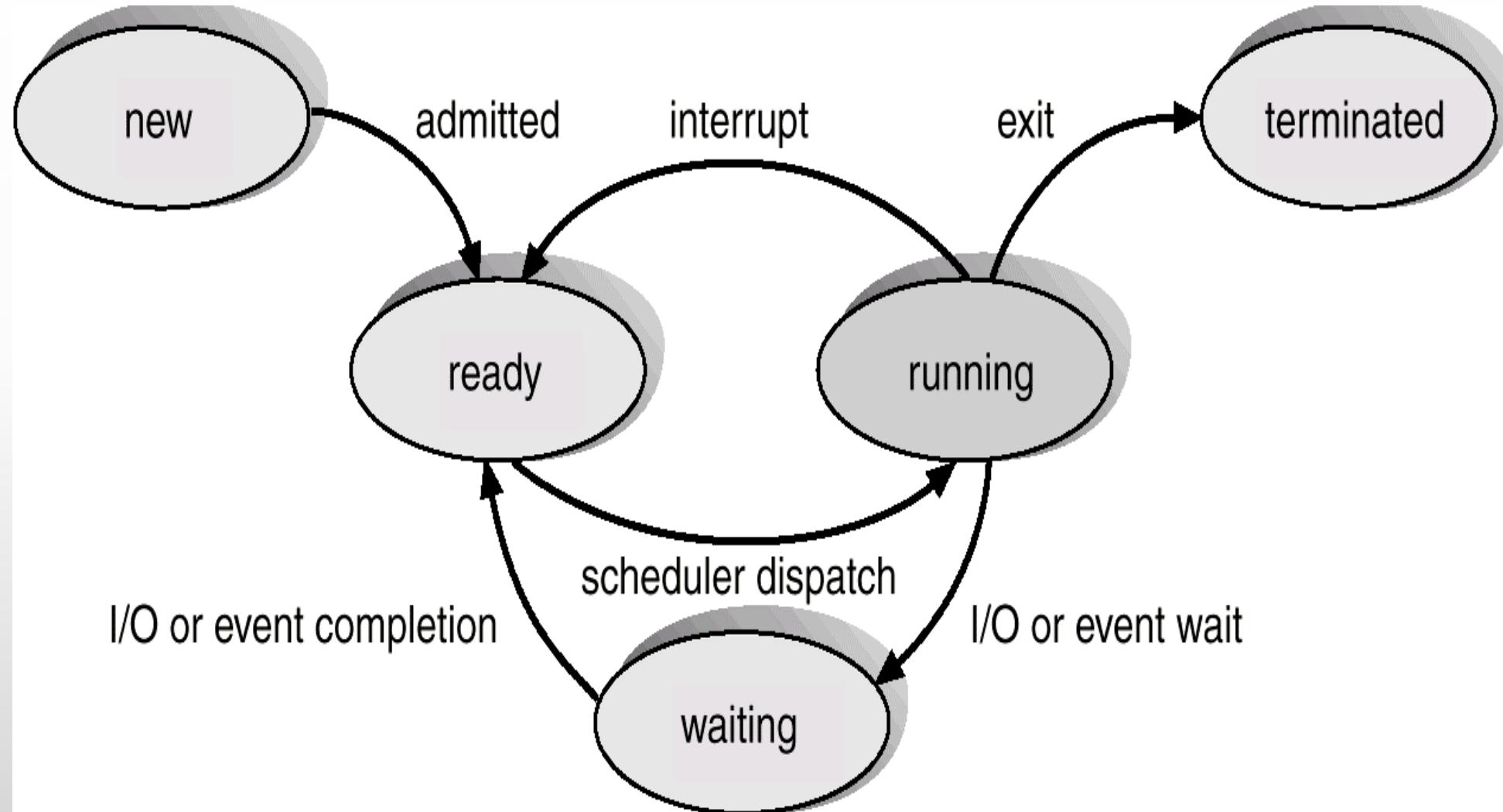
Why Processes?

- Resource sharing (logical (files) and physical(hardware)).
- Computation speedup - taking advantage of multiprogramming – i.e. example of a customer/server database system.
- Modularity for protection.





Process States



Process State

new

- The process is just being put together

Running

- Instructions being executed. This running process holds the CPU

Waiting

- For an event (hardware, human, or another process.)





Process State

Ready

- The process has all needed resources - waiting for CPU only

Suspended

- Another process has explicitly told this process to sleep. It will be awakened when a process explicitly awakens it

Terminated

- The process is being torn apart



Process States

Process Control Block PCB

- Contains information associated with each process

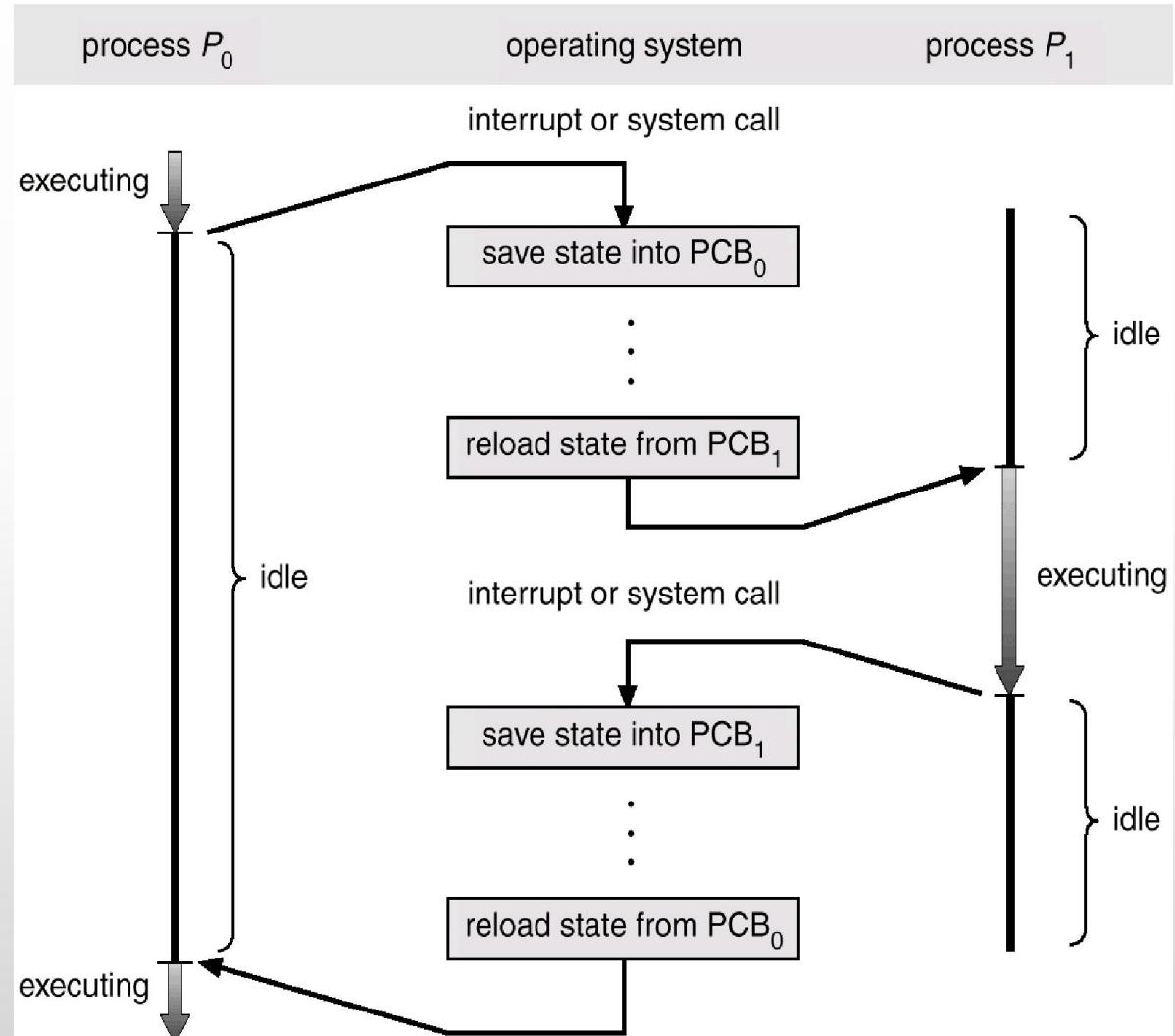
PCB Structure holds

- PC, CPU registers,
- memory management information,
- accounting (time used, ID, ...)
- I/O status (such as file resources),
- scheduling data (relative priority, etc.)
- Process State (so running, suspended, etc. is simply a field in the PCB).





Scheduling Components



The act of **Scheduling** a process means changing the active PCB pointed to by the CPU. Also called a **context switch**.

A context switch is essentially the same as a process switch

- it means that the memory, as seen by one process is changed to the memory seen by another process.



Scheduling Components

Scheduling Queues

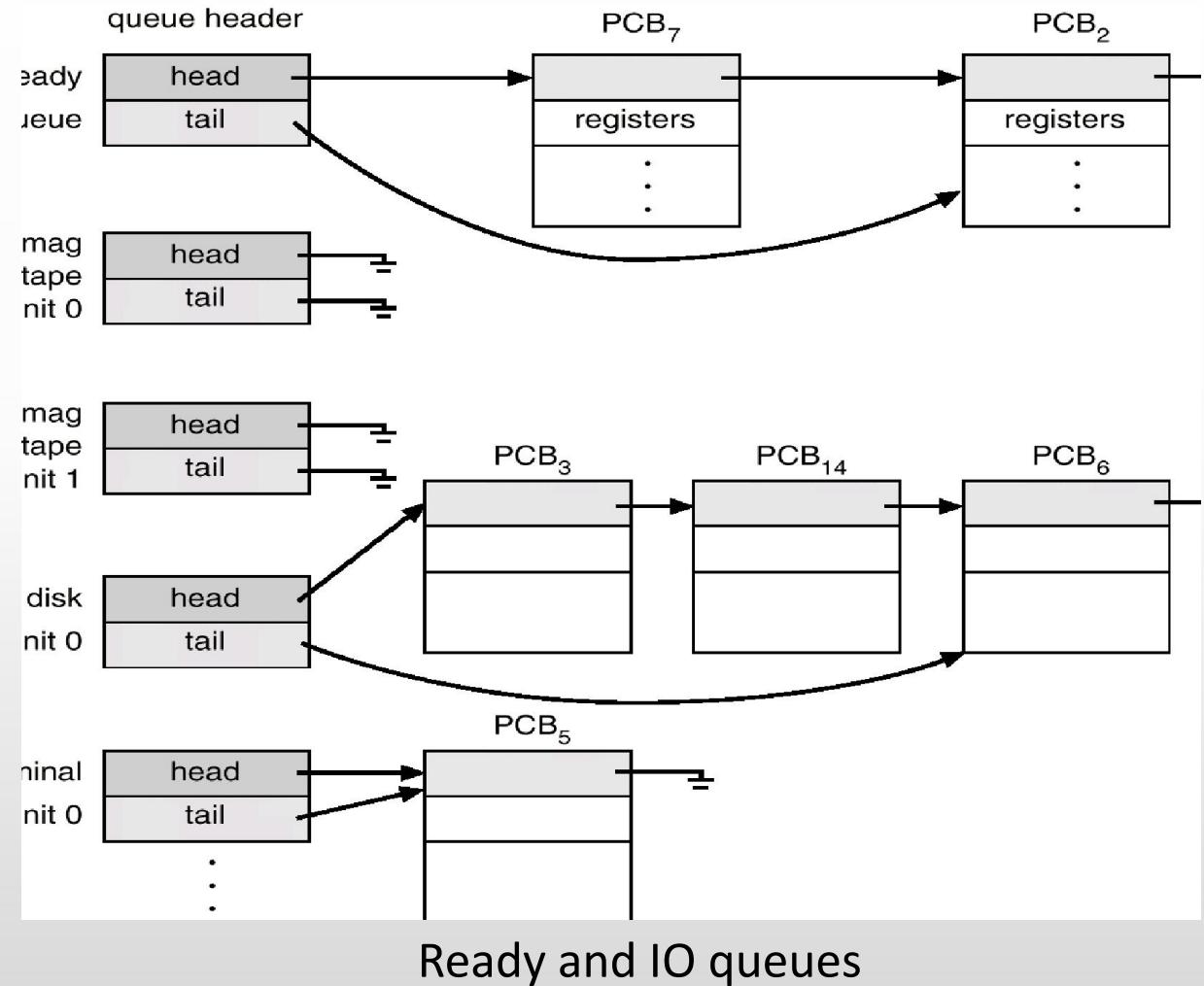
- Process is driven by events that are triggered by needs and availability

Ready Queue

- Contains those processes that are ready to run

IO Queue (Waiting State)

- Holds those processes waiting for IO service





Scheduling Components

Long Term Scheduler

- Run seldom (when job comes into memory)
- Controls degree of multiprogramming
- Tries to balance arrival and departure rate through an appropriate job mix



Scheduling Components

Short Term Scheduler

- Contains three functions:
- Code to remove a process from the processor at the end of its run.
 - a) Process may go to ready queue or to a wait state.
- Code to put a process on the ready queue –
 - a) Process must be ready to run.
 - b) Process placed on queue based on priority.
- Code to take a process off the ready queue and run that process (also called **dispatcher**).
 - a) Always takes the first process on the queue (no intelligence required)
 - b) Places the process on the processor.
- This code runs frequently and so should be as short as possible.





Scheduling Components

Medium Term Scheduler

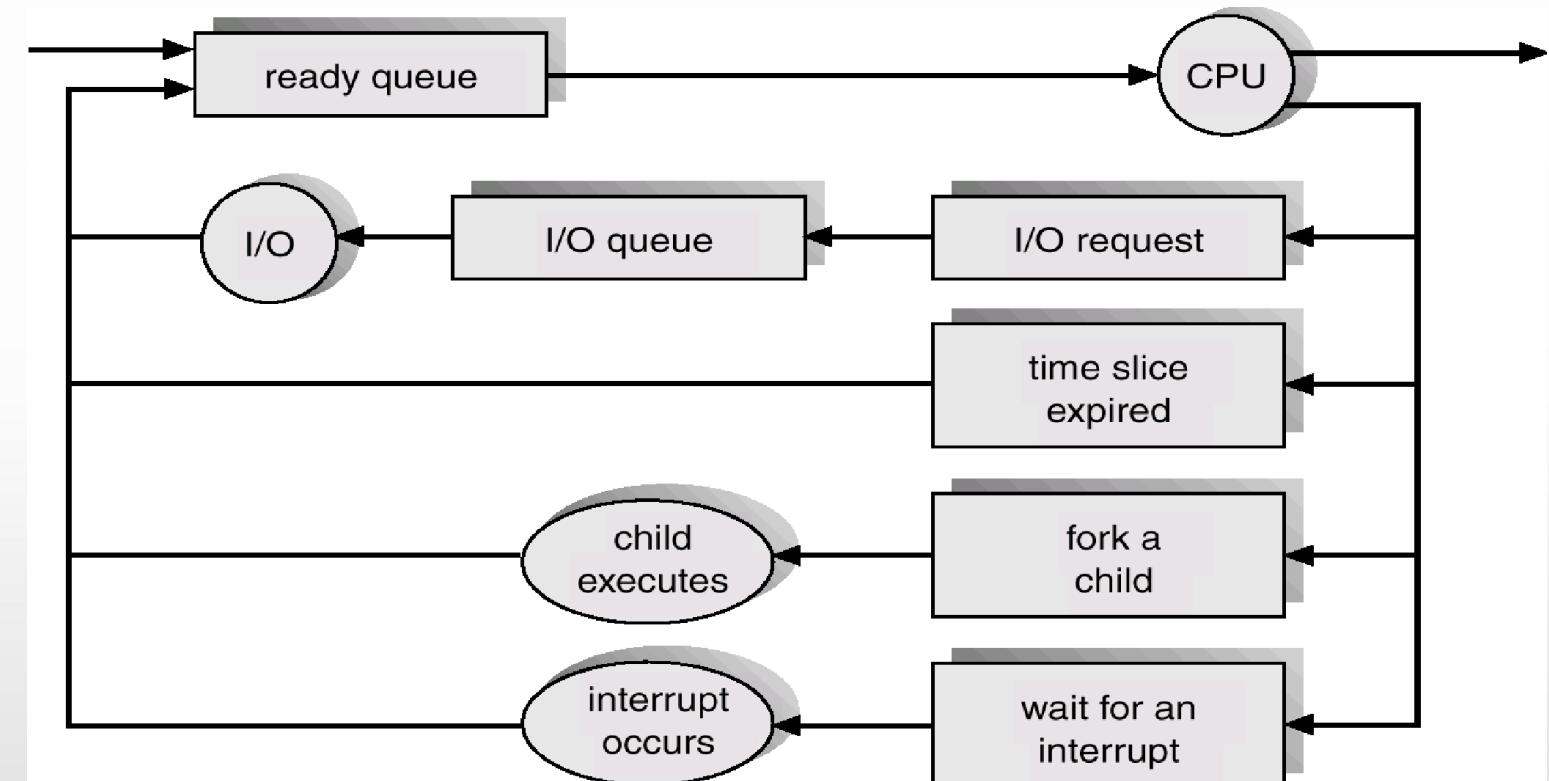
- Mixture of CPU and memory resource management.
- Swap out/in jobs to improve mix and to get memory.
- Controls change of priority





Scheduling Components

Short Term Scheduler



INTERRUPT HANDLER

In addition to doing device work, it also readies processes, moving them, for instance, from waiting to ready.





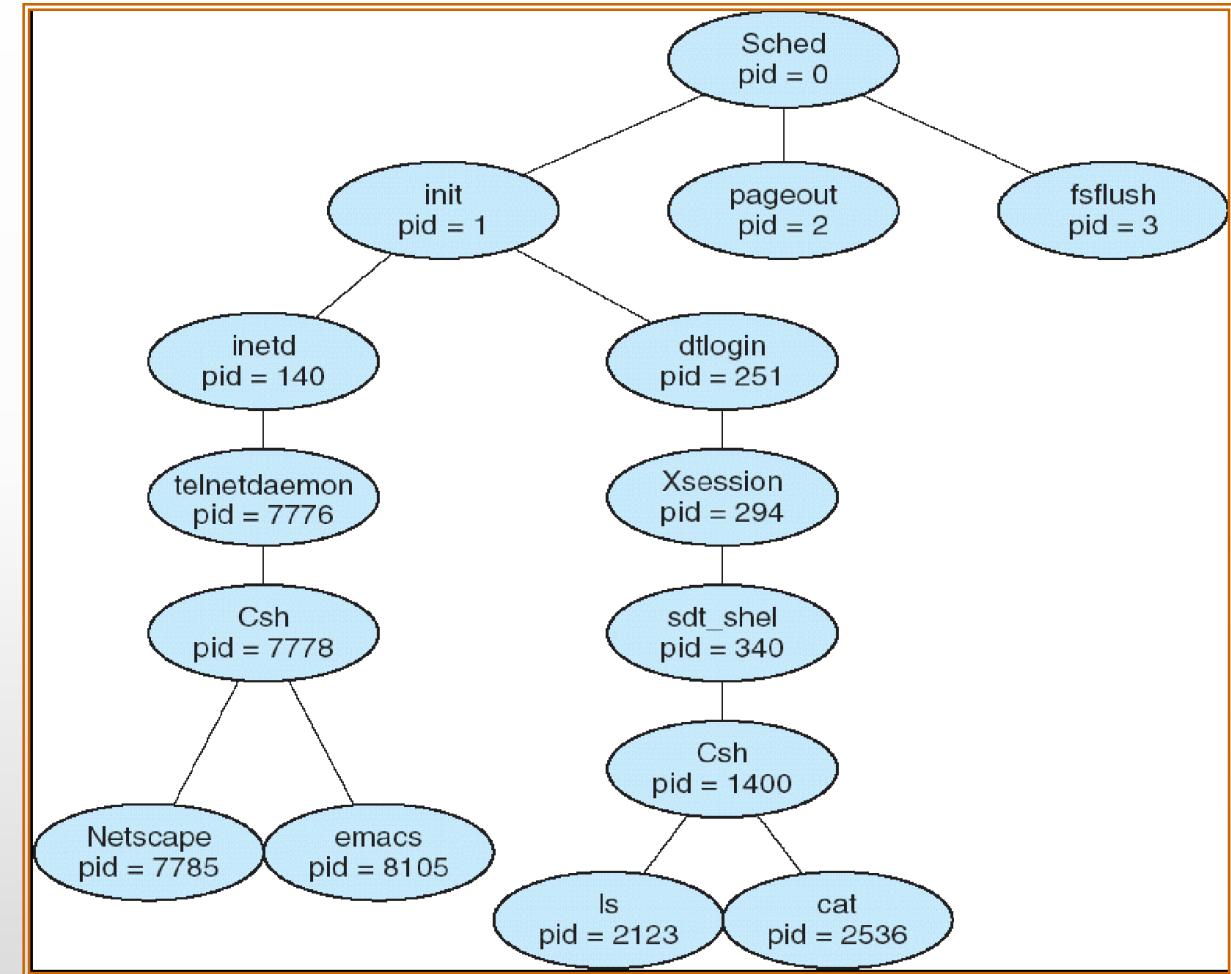
Process Relationships

Parent can run concurrently with child, or wait for completion.

Child may share all (fork/join) or part of parent's variables.

Death of parent may force death of child.

Processes are static (never terminate) or dynamic (can terminate).





Process Relationships

Independent

Execution is deterministic and reproducible. Execution can be stopped/ started without affecting other processes.

Cooperating

Execution depends on other processes or is time dependent. Here the same inputs won't always give the same outputs; the process depends on other external states.





Interprocess Communication IPC

IPC is how processes talk to each other.

There are basically two methods:

Shared memory (with a process "kick") -- fast/ no data transfer.

Message Passing -- distributed/ better isolation.

DIRECT COMMUNICATION:

Need to know name of sender/receiver. Mechanism looks like this:

send (Process_P, message) ;

receive (Process_Q , message);

receive (id, message) <-- from any sender





Interprocess Communication IPC

The Producer/Consumer Problem is a standard mechanism.

One process produces items that are handed off to the consumer where they are "used".

repeat

produce item

send(consumer, nextp)

until false

repeat

receive(producer, nextp)

consume item

until false





Interprocess Communication IPC

Other properties of Direct Communication:

Link established automatically (when send or receive requested.)

Only two processes in this form.

One link per pair of processes.

Generally Bi-directional

Receiver may not need ID of sender.

Disadvantage of Direct Communication:

The names of processes must be known - they can't be easily changed since they are explicitly named in the send and receive.





Interprocess Communication IPC

INDIRECT COMMUNICATION

Processes communicate via a named mailbox rather than via a process name.

Mechanism looks like this:

```
open( mailbox_name );  
send ( mailbox_name, message );  
receive ( mailbox_name, message);
```





Interprocess Communication IPC

INDIRECT COMMUNICATION

Link is established if processes have a shared mailbox.
So mailbox must be established before the send/receive.

More than two processes are allowed to use the same mailbox.

May cause confusion with multiple receivers
- if several processes have outstanding receives on a mailbox, which one gets a message?





Interprocess Communication IPC

BUFFERING:

Options include:

Zero - sender must wait for recipient to get message. Provides a rendezvous.

Bounded - sender must wait for recipient if more than n messages in buffer.

Unbounded - sender is never delayed.

MESSAGE FORMAT:

Fixed, Variable, or Typed (as in language typing) size messages.

Send reference rather than copy (good for large messages).

Suspended vs. unsuspended sends.



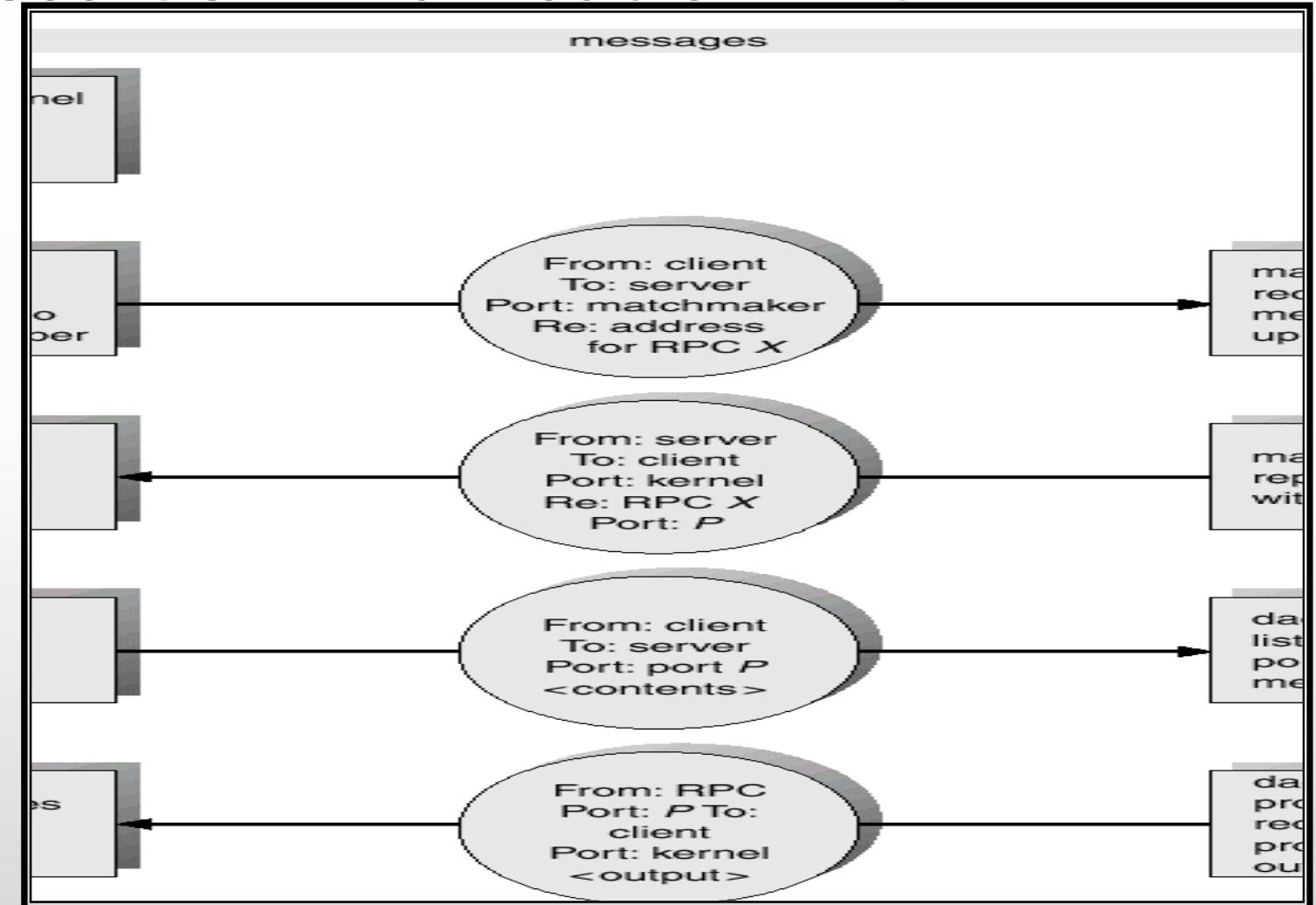


Interprocess Communication IPC

Remote procedure call

(RPC) abstracts procedure calls between processes on networked systems.

Stubs – client-side proxy for the actual procedure on the server.

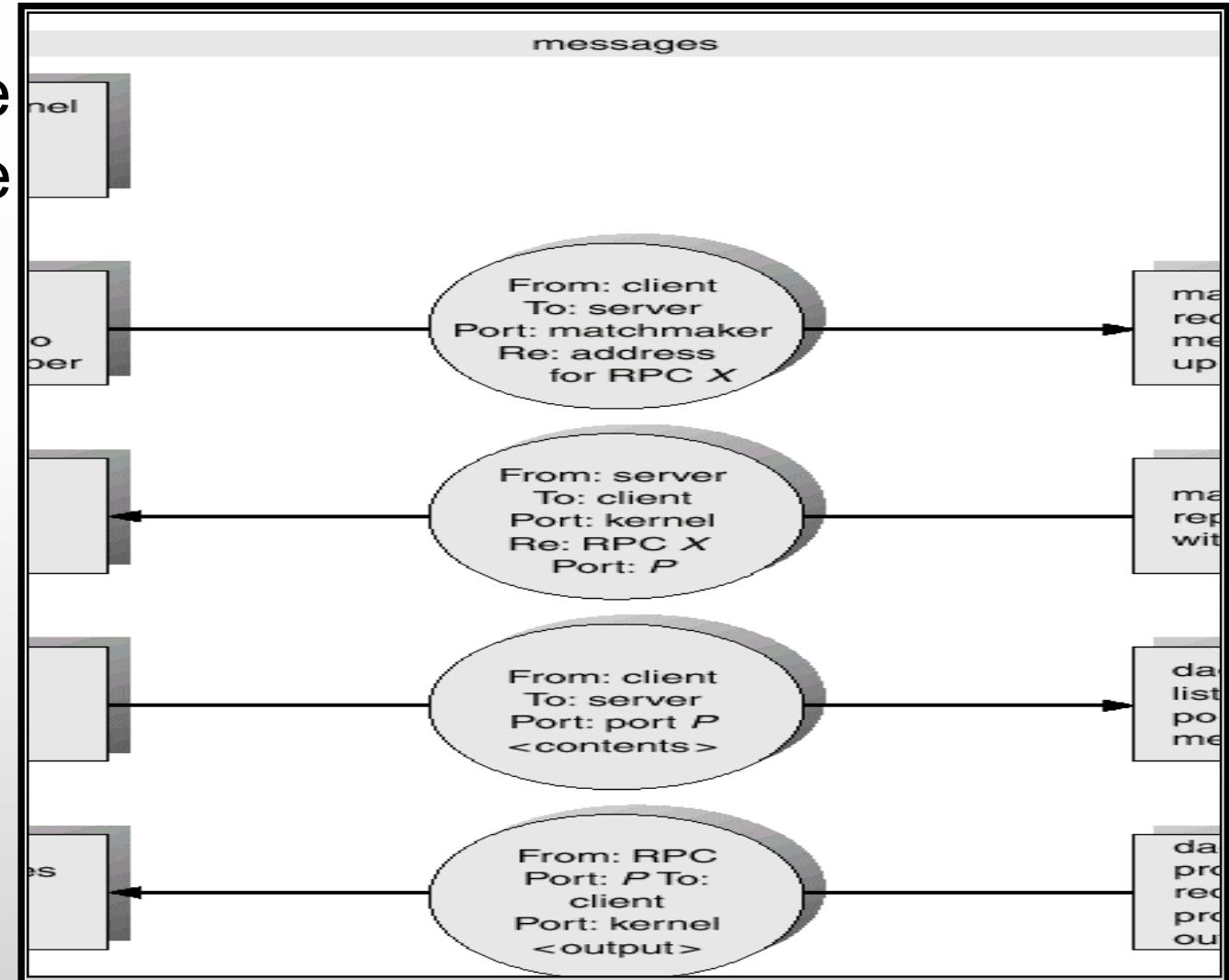




Interprocess Communication IPC

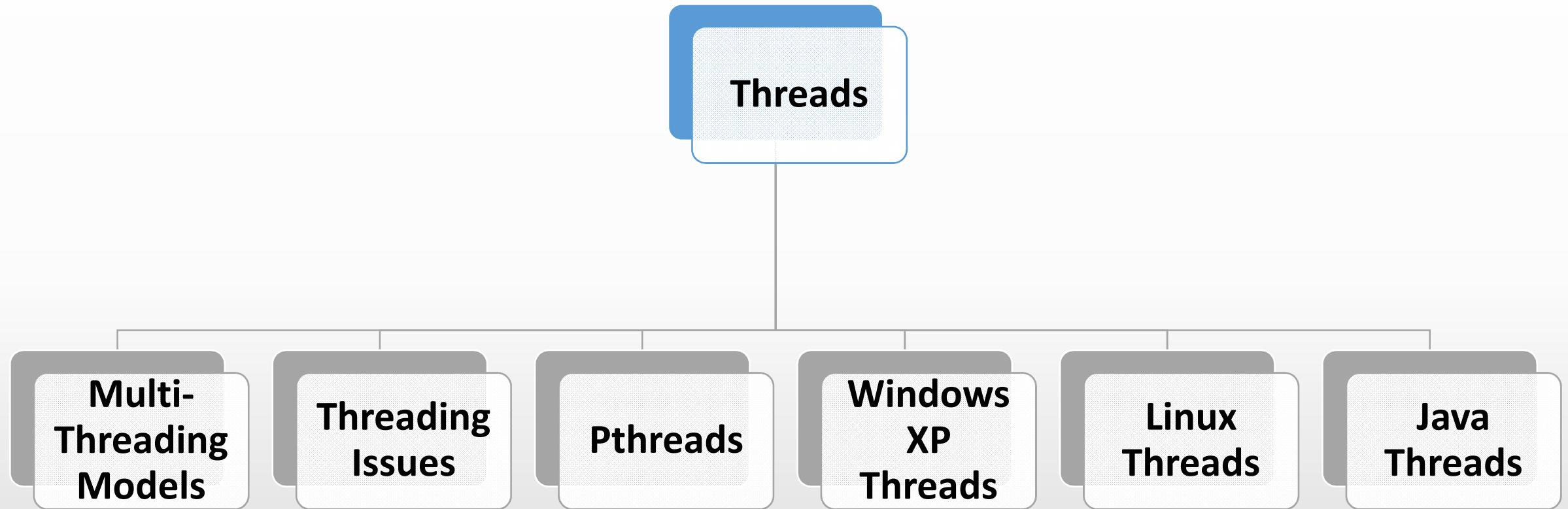
The client-side stub locates the server and *marshalls* the parameters.

The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server.



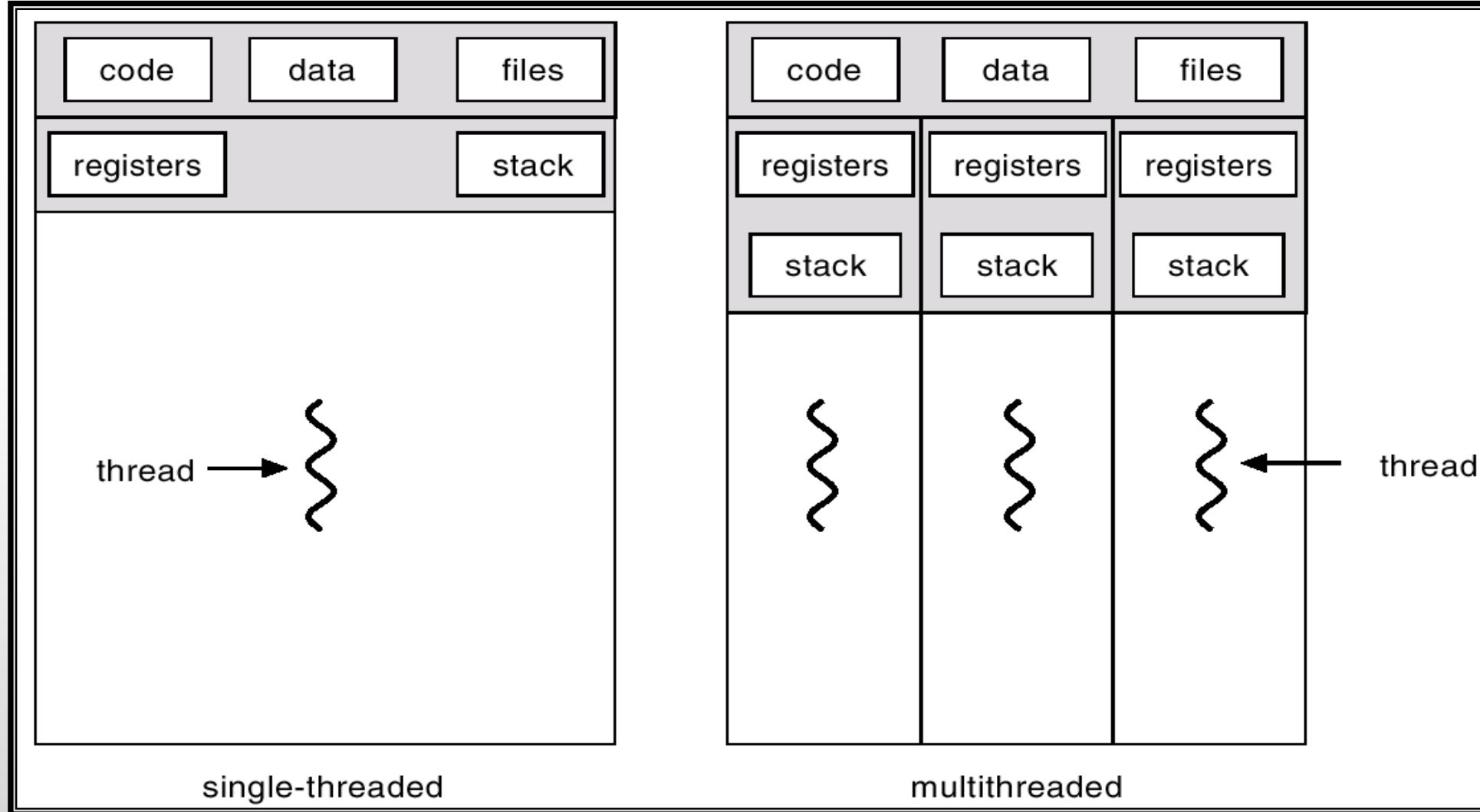


Overview





Single and Multithreaded Processes





Threads Benefits

Responsiveness

Resource Sharing

Economy

Utilization of MP Architectures



Thread Types

User threads - Thread management done by user-level threads library

Examples

- POSIX *Pthreads*
- Mach *C-threads*
- Solaris *threads*

Kernel threads - supported by the Kernel

Examples

- Windows 95/98/NT/2000
- Solaris
- Tru64 UNIX
- BeOS
- Linux





Multithreading Models

Many-to-one

One-to-one

Many-to-many

How do user and kernel threads map into each other?





Many-to-One Thread

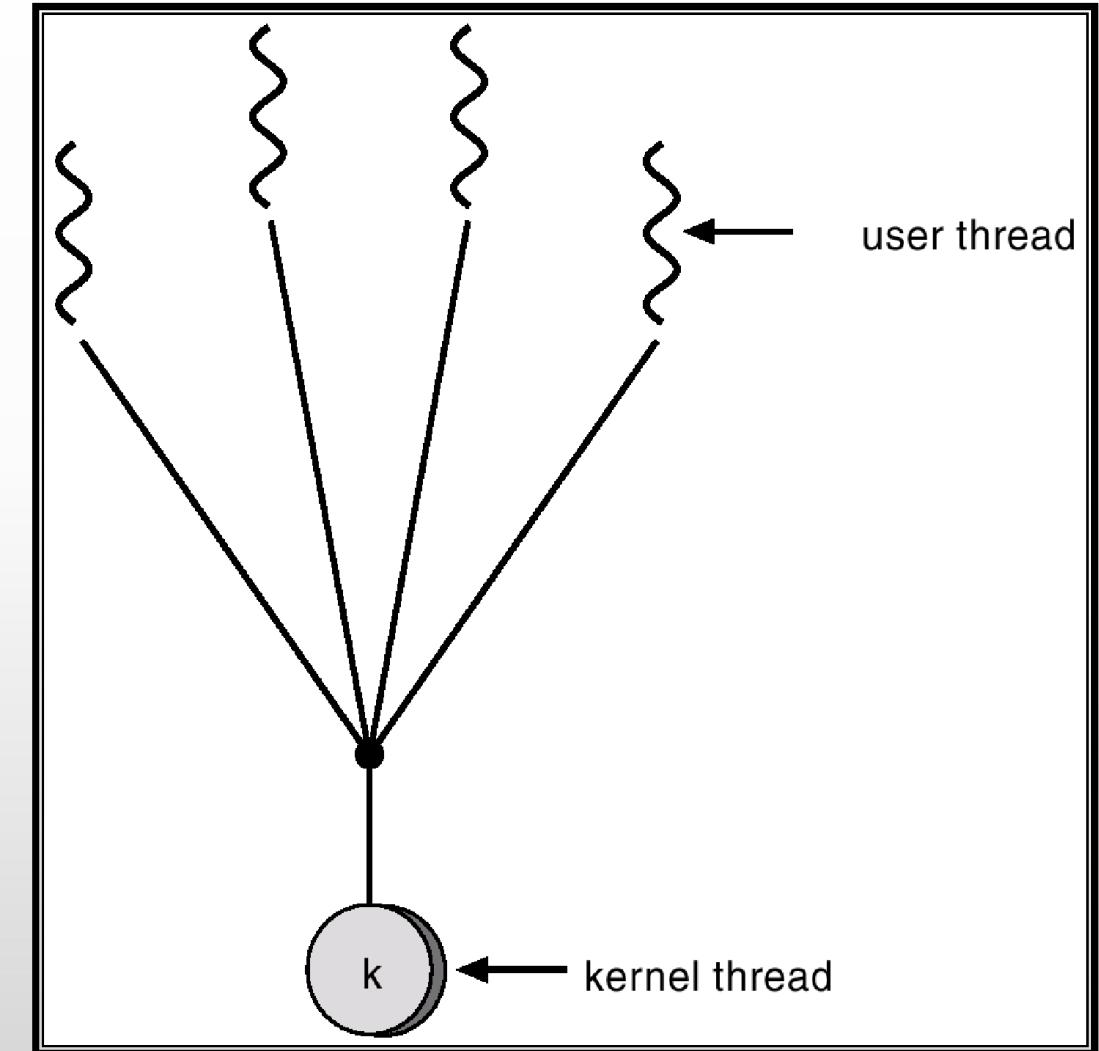
Many user-level threads mapped to single kernel thread.

Used on systems that do not support kernel threads.

Examples:

Solaris Green Threads

GNU Portable Threads



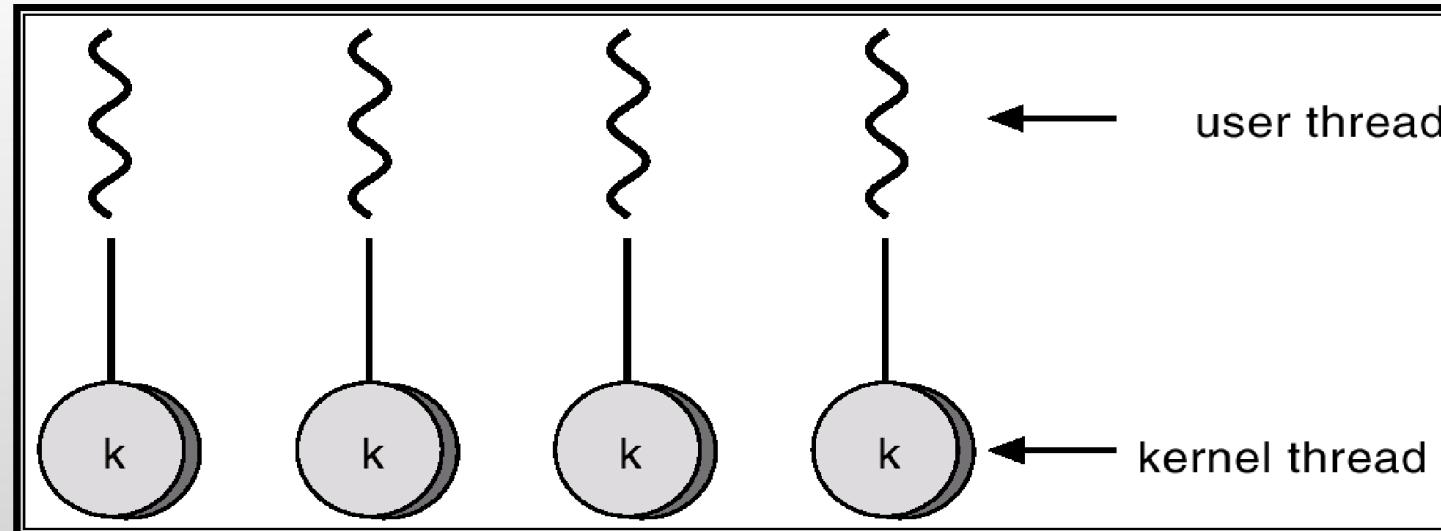


One-to-One

Each user-level thread maps to kernel thread.

Examples

- Windows 95/98/NT/2000
- Linux





Threading Issues

Semantics of fork() and exec() system calls

Does **fork()** duplicate only the calling thread or all threads?

Thread cancellation

Terminating a thread before it has finished

Two general approaches:

Asynchronous cancellation terminates the target thread immediately

Deferred cancellation allows the target thread to periodically check if it should be cancelled





Threading Issues

Signal handling

Signals are used in UNIX systems to notify a process that a particular event has occurred

A **signal handler** is used to process signals

- 1.Signal is generated by particular event
- 2.Signal is delivered to a process
- 3.Signal is handled

Options:

- Deliver the signal to the thread to which the signal applies
- Deliver the signal to every thread in the process
- Deliver the signal to certain threads in the process
- Assign a specific thread to receive all signals for the process





Threading Issues

Thread pools

Create a number of threads in a pool where they await work

Advantages:

Usually slightly faster to service a request with an existing thread than create a new thread

Allows the number of threads in the application(s) to be bound to the size of the pool





Threading Issues

Thread specific data

Allows each thread to have its own copy of data

Useful when you do not have control over the thread creation process (i.e. when using a thread pool)





Threading Issues

Scheduler activations

Many-to-Many models require communication to maintain the appropriate number of kernel threads allocated to the application

Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library

This communication allows an application to maintain the correct number kernel threads





Thread Implementations

PThreads

A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

API specifies behavior of the thread library, implementation is up to development of the library

Common in UNIX operating systems (Solaris, Linux, Mac OS X)





Thread Implementations

Windows Threads

Implements the one-to-one mapping

Each thread contains

- A thread id

- Register set

- Separate user and kernel stacks

- Private data storage area

The register set, stacks, and private storage area are known as the **context** of the threads



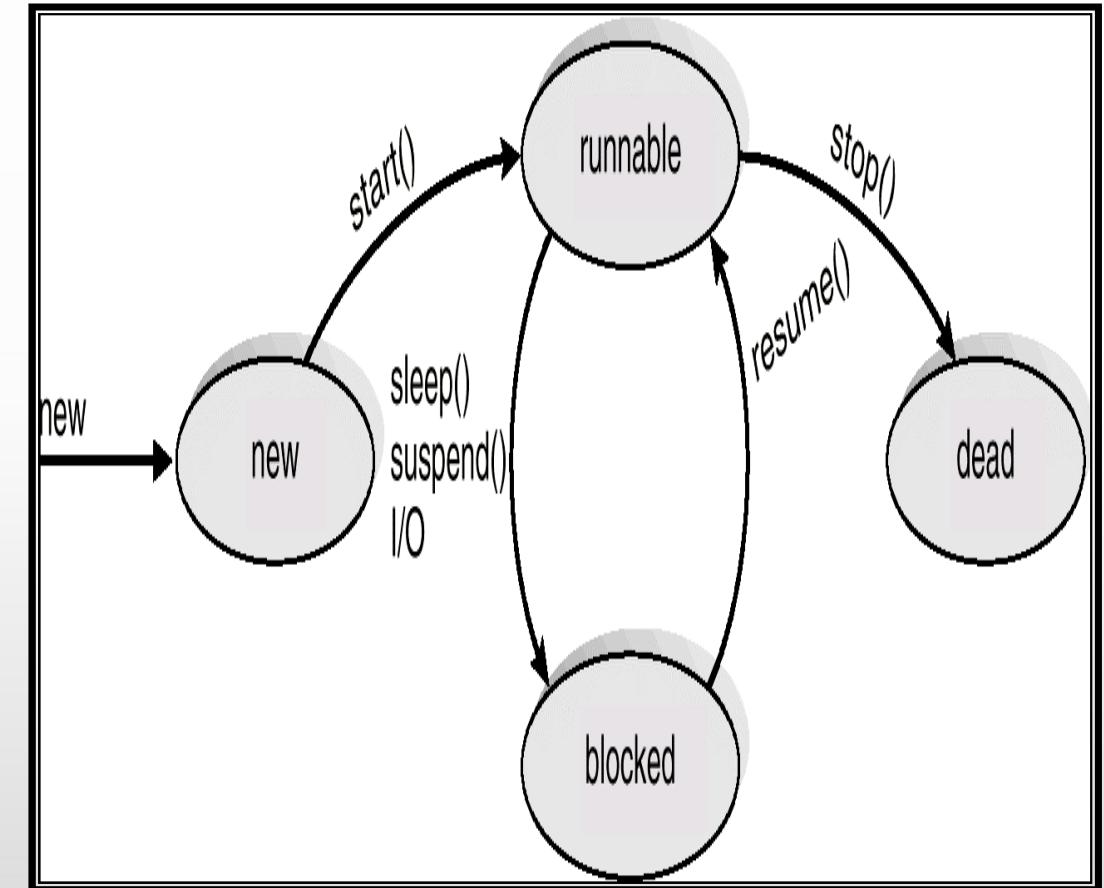


Thread Implementations

Linux Threads

Linux refers to them as *tasks* rather than *threads*

Thread creation is done through **clone()** system call **clone()** allows a child task to share the address space of the parent task (process)





Thread Implementations

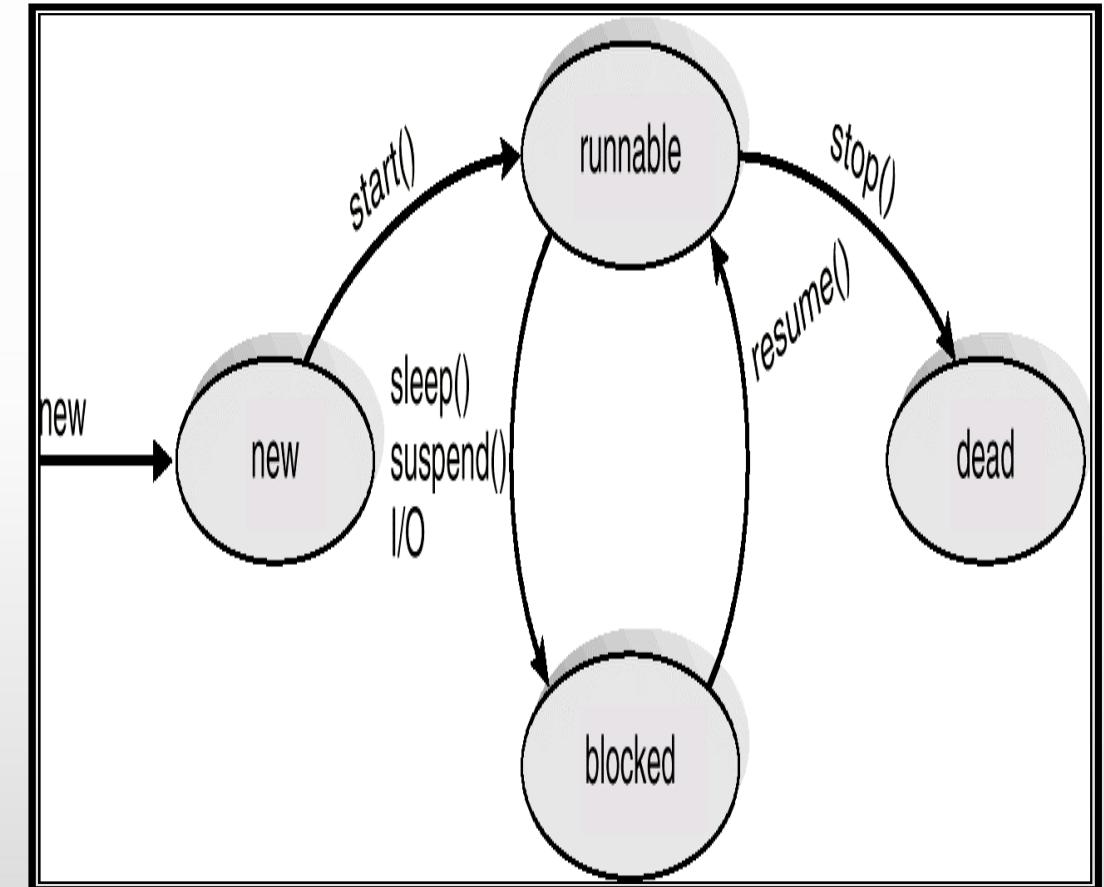
Java Threads

Java threads may be created by:

Extending Thread class

Implementing the Runnable interface

Java threads are managed by the JVM.





Thank You Very Much

Benjamin Kommey

bkommey.coe@knust.edu.gh

nii_kommey@msn.com

050 770 32 86

Skype_id: calculus.affairs





“ I choose a lazy person to do a hard job. Because a lazy person will find an easy way to do it. ”

Bill Gates





Overview

Scheduling

Basic Concepts

Scheduling Criteria

Scheduling Algorithms

Multi-processor Scheduling





Overview

Scheduling

Realtime
Scheduling

Thread
Scheduling

Operating
Systems
Example

Java
Thread
Scheduling

Algorithm
Evaluation





Scheduling

CPU
Scheduling

is about how to get a process attached to a processor

- It centers around efficient algorithms that perform well.
- The design of a scheduler is concerned with making sure all users get their fair share of the resources.





Scheduling Concepts

Multiprogramming

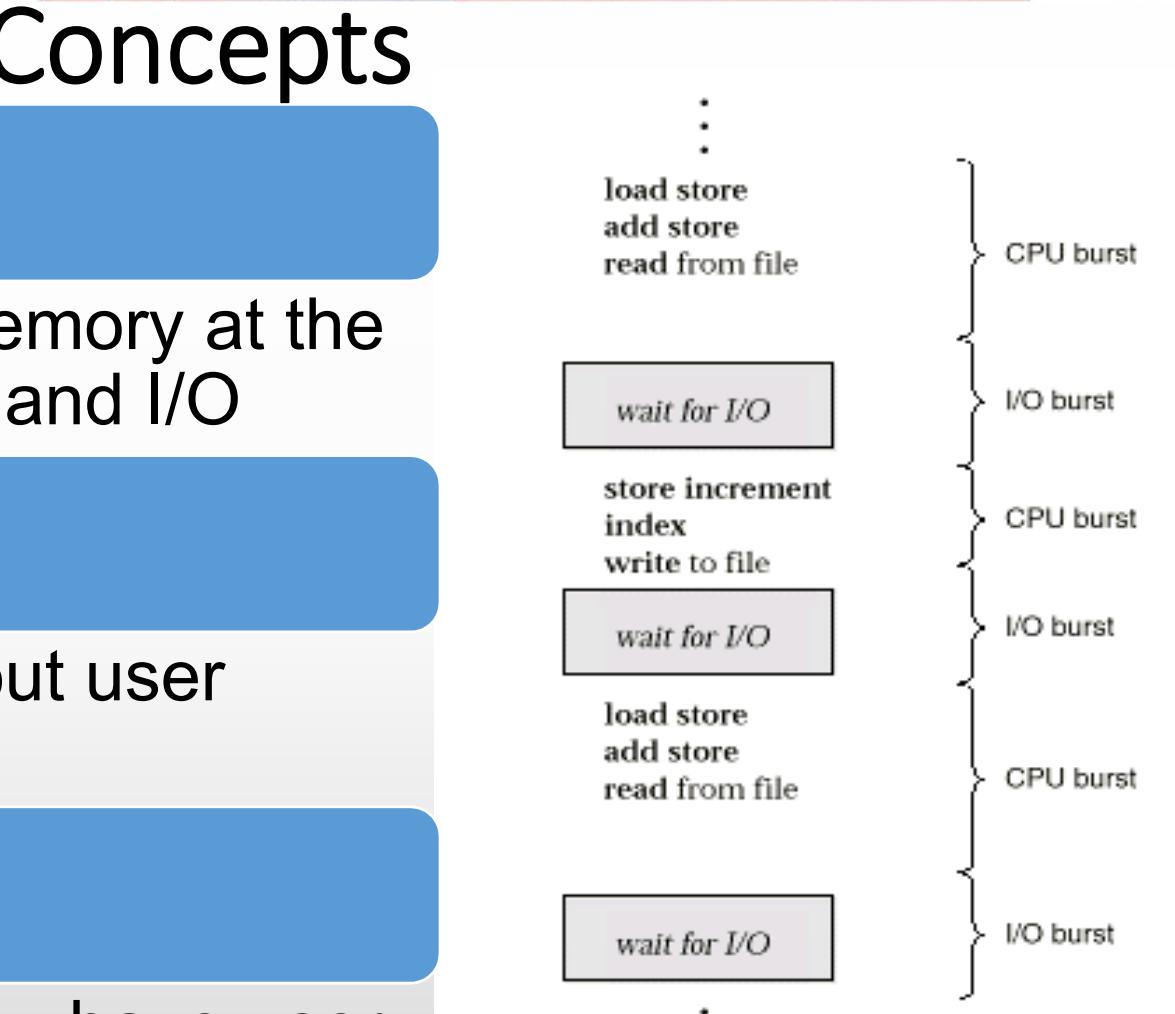
- A number of programs can be in memory at the same time. Allows overlap of CPU and I/O

Jobs

- (batch) are programs that run without user interaction

User

- (time shared) are programs that may have user interaction





Scheduling Concepts

Process

Process

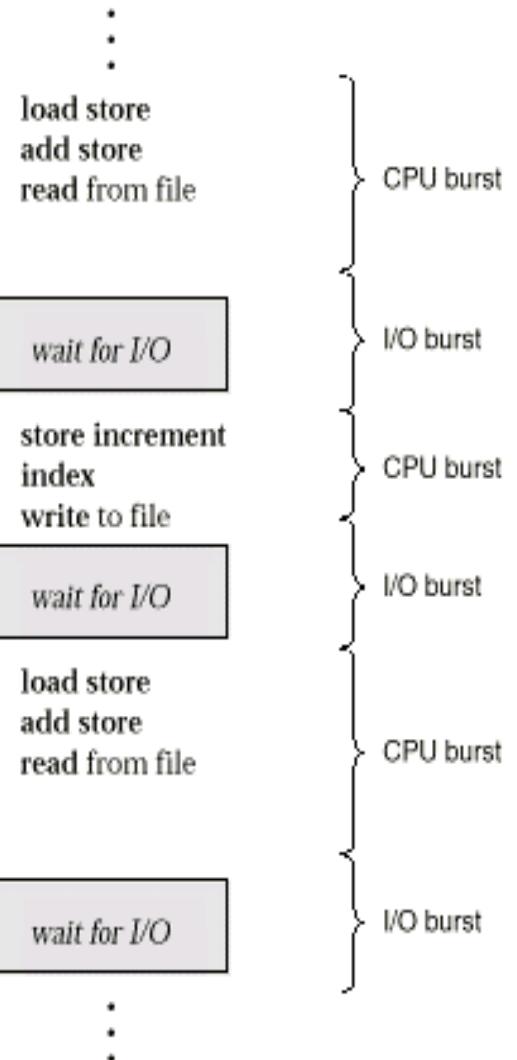
- is the common name for both

CPU – IO burst cycle

- Characterizes process execution, which alternates, between CPU and I/O activity. CPU times are generally much shorter than I/O times

Preemptive Scheduling

- An interrupt causes currently running process to give up the CPU and be replaced by another process





The Scheduler

Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them

CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

Scheduling under 1 and 4 is *nonpreemptive*

All other scheduling is *preemptive*



The Dispatcher

Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:

switching context

switching to user mode

jumping to the proper location in the user program to restart that program

Dispatch latency – time it takes for the dispatcher to stop one process and start another running



Criteria For Performance Evaluation

Note usage of the words **DEVICE**, **SYSTEM**, **REQUEST**, **JOB**.

UTILIZATION

The fraction of time a device is in use.

(ratio of in-use time / total observation time)

THROUGHPUT

The number of job completions in a period of time. (jobs / second)

SERVICE TIME

The time required by a device to handle a request. (seconds)

QUEUEING TIME

Time on a queue waiting for service from the device. (seconds)





Criteria For Performance Evaluation

RESIDENCE TIME

The time spent by a request at a device.

$$\text{RESIDENCE TIME} = \text{SERVICE TIME} + \text{QUEUEING TIME}.$$

RESPONSE TIME

Time used by a system to respond to a User Job. (seconds)

THINK TIME

The time spent by the user of an interactive system to figure out the next request. (seconds)

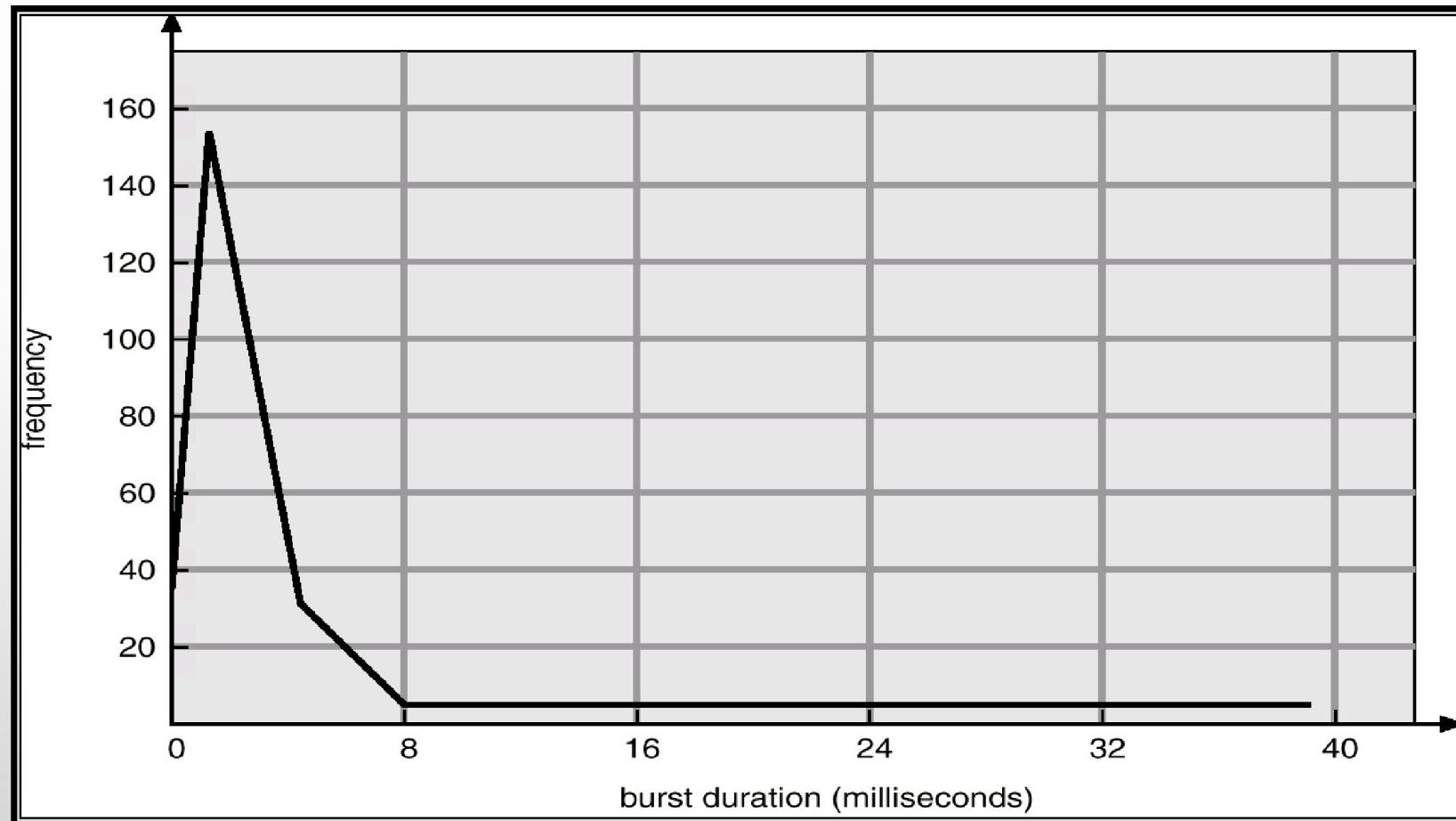
The goal is to optimize both the average and the amount of variation. (but beware the ogre predictability.)





Scheduling Behaviour

Most Processes Don't Use Up Their Scheduling Quantum!





Scheduling Algorithms

First-Come, First Served FCFS

- FCFS is same as FIFO (First-In, First-Out)
- Simple, fair, but poor performance. Average queueing time may be long.
- What are the average queueing and residence times for this scenario?
- How do average queueing and residence times depend on ordering of these processes in the queue?



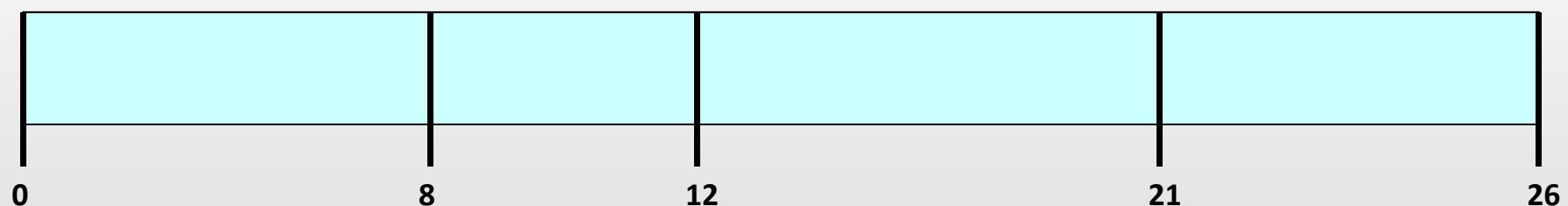


Scheduling Algorithms

EXAMPLE DATA:

Process	Arrival Time	Service Time
1	0	8
2	1	4
3	2	9
4	3	5

FCFS



Residence Time at the CPU

$$\text{Average wait} = ((8-0) + (12-1) + (21-2) + (26-3)) / 4 = 61 / 4 = 15.25$$



Scheduling Algorithms

SHORTEST JOB FIRST (SJF)

Optimal for minimizing queueing time, but impossible to implement.

Tries to predict the process to schedule based on previous history.

Predicting the time the process will use on its next schedule:

$$t(n+1) = w * t(n) + (1 - w) * T(n)$$

whereby $t(n+1)$ is time of next burst.

$t(n)$ is time of current burst.

$T(n)$ is average of all previous bursts .

w is a weighting factor emphasizing current or previous bursts.





Scheduling Algorithms

Preemptive Algorithms

- Yank the CPU away from the currently executing process when a higher priority process is ready
- Can be applied to both Shortest Job First or to Priority scheduling
- Avoids "hogging" of the CPU
- On time sharing machines, this type of scheme is required because the CPU must be protected from a run-away low priority process
- Give short jobs a higher priority – perceived response time is thus better
- What are average queueing and residence times? Compare with FCFS



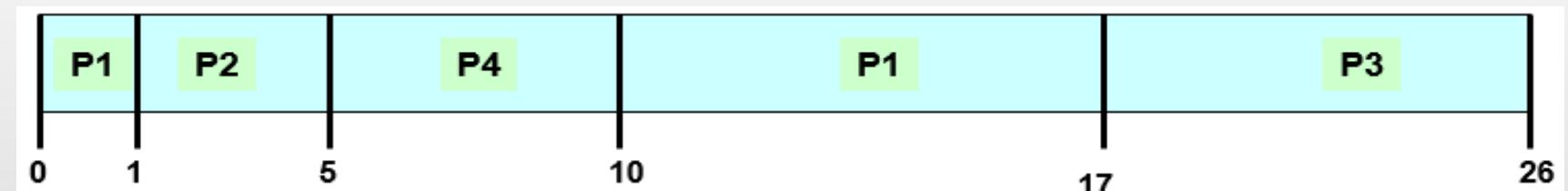


Scheduling Algorithms

Example Data

Process	Arrival Time	Service Time
1	0	8
2	1	4
3	2	9
4	3	5

Preemptive Shortest Job First



$$\text{Average wait} = ((5-1) + (10-3) + (17-0) + (26-2))/4 = 52/4 = 13.0$$



Scheduling Algorithms

PRIORITY BASED SCHEDULING:

Assign each process a priority. Schedule highest priority first. All processes within same priority are FCFS.

Priority may be determined by user or by some default mechanism. The system may determine the priority based on memory requirements, time limits, or other resource usage.

Starvation occurs if a low priority process never runs. Solution: build aging into a variable priority.

Delicate balance between giving favorable response for interactive jobs, but not starving batch jobs.





Scheduling Algorithms

ROUND ROBIN:

Use a timer to cause an interrupt after a predetermined time.
Preempts if task exceeds it's quantum.

Train of events

 Dispatch

 Time slice occurs OR process suspends on event

 Put process on some queue and dispatch next

Use numbers in last example to find queueing and residence times. (Use quantum = 4 sec.)





Scheduling Algorithms

Definitions:

Context Switch

Changing the processor from running one task (or process) to another. Implies changing memory.

Processor Sharing

Use of a small quantum such that each process runs frequently at speed $1/n$.

Reschedule latency How long it takes from when a process requests to run, until it finally gets control of the CPU.





Scheduling Algorithms

ROUND ROBIN:

Choosing a time quantum

Too short - inordinate fraction of the time is spent in context switches.

Too long - reschedule latency is too great. If many processes want the CPU, then it's a long time before a particular process can get the CPU. This then acts like FCFS.

Adjust so most processes won't use their slice.

As processors have become faster, this is less of an issue.



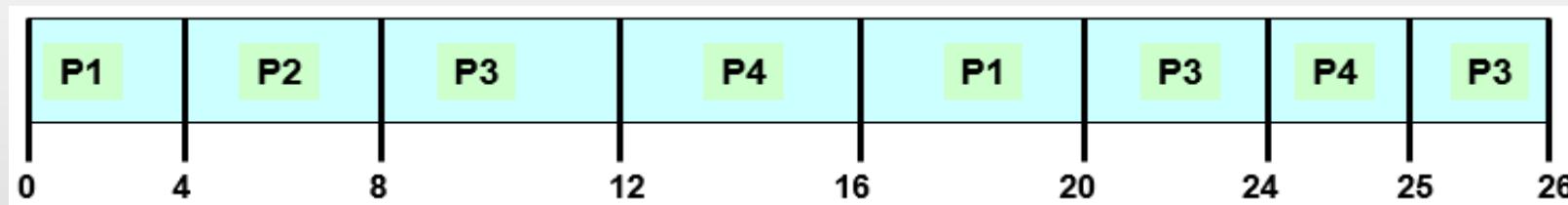


Scheduling Algorithms

Example data:

Process	Arrival Time	Service Time
1	0	8
2	1	4
3	2	9
4	3	5

Round Robin, quantum = 4, no priority-based preemption



Note:

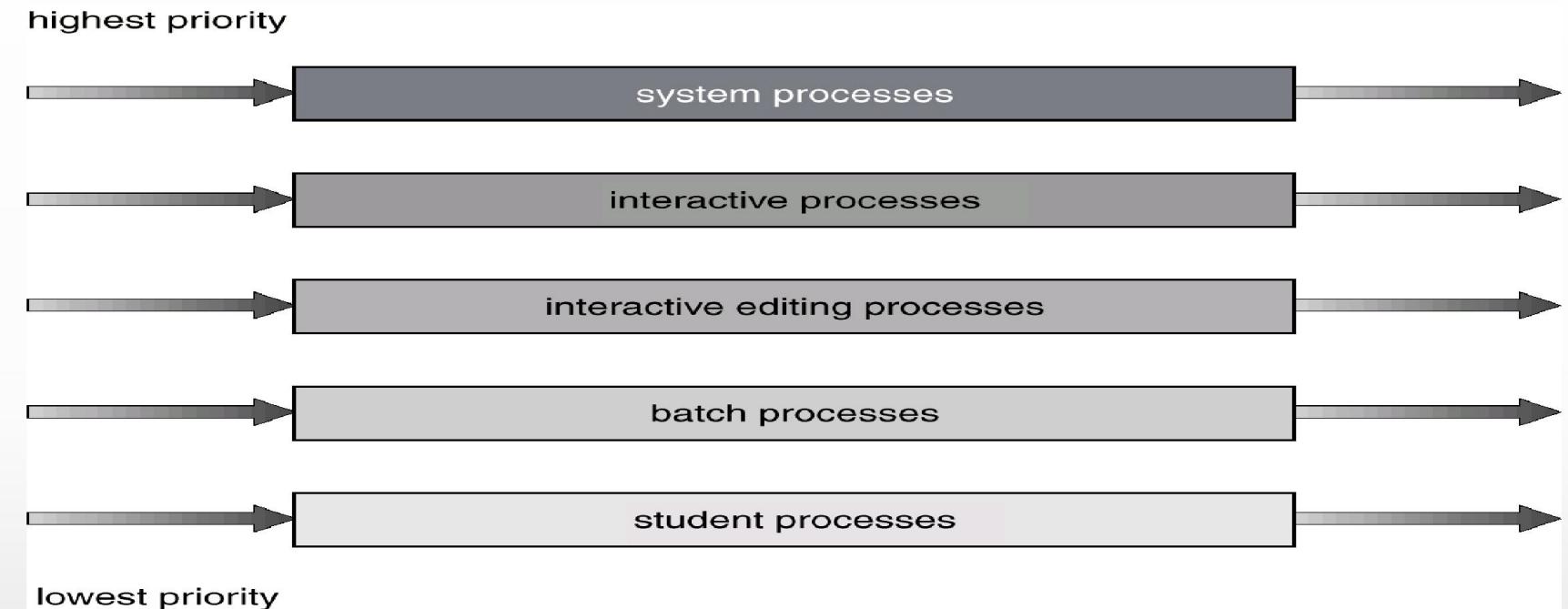
Example violates rules for quantum size since most processes don't finish in one quantum

$$\text{Average wait} = ((20-0) + (8-1) + (26-2) + (25-3))/4 = 74/4 = 18.5$$



MULTI-LEVEL QUEUES

Scheduling Algorithms



Each queue has its scheduling algorithm.

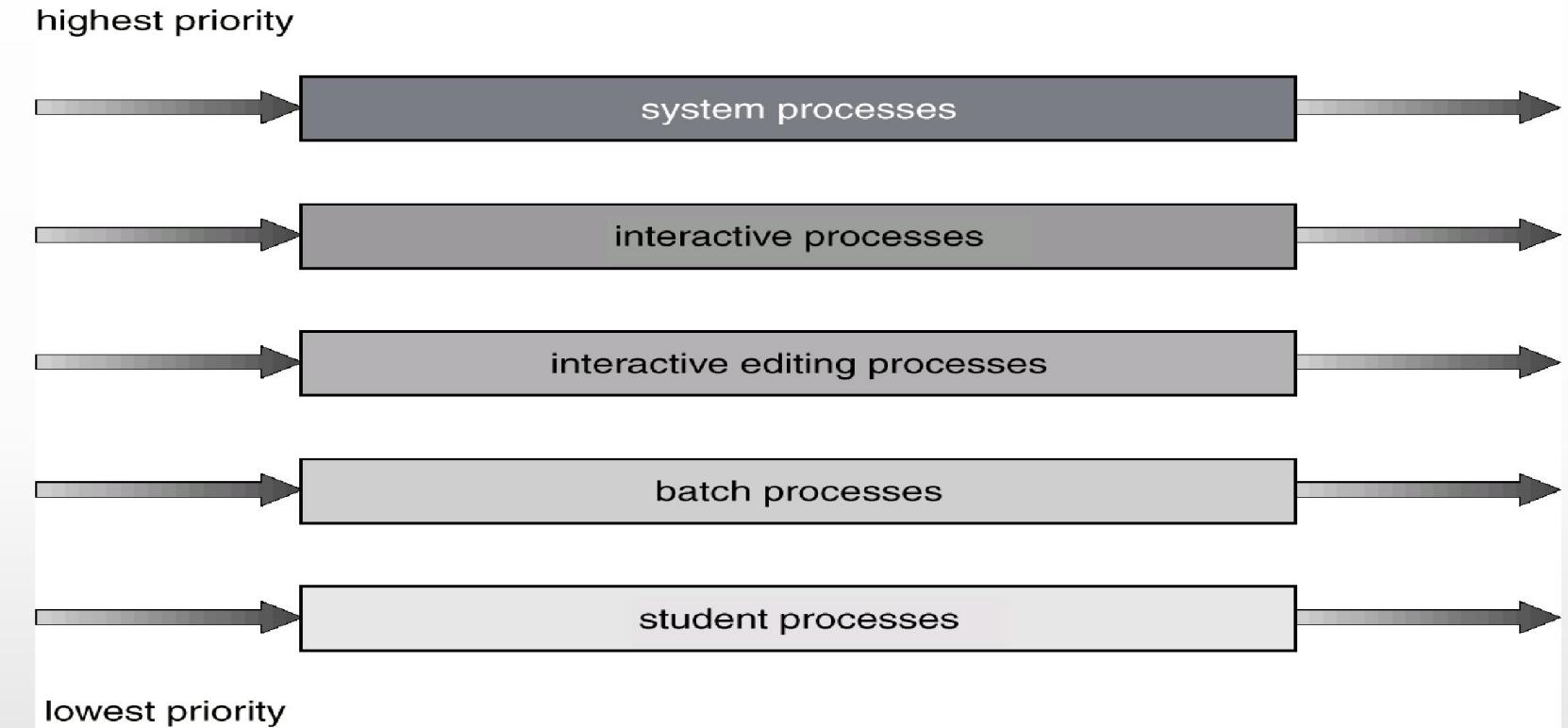
Then some other algorithm (perhaps priority based) arbitrates between queues.

Can use feedback to move between queues





MULTI-LEVEL QUEUES



Method is complex but flexible.

For example, could separate system processes, interactive, batch, favored, unfavored processes



Using Priorities

Priorities used in Windows OS

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1





Scheduling Algorithms

Multi Processor Scheduling

- Different rules for homogeneous or heterogeneous processors.
- Load sharing in the distribution of work, such that all processors have an equal amount to do.
- Each processor can schedule from a common ready queue (equal machines) OR can use a master slave arrangement





Scheduling Algorithms

Real Time Scheduling

- *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time.
- *Soft real-time* computing – requires that critical processes receive priority over less fortunate ones





Linux Scheduling

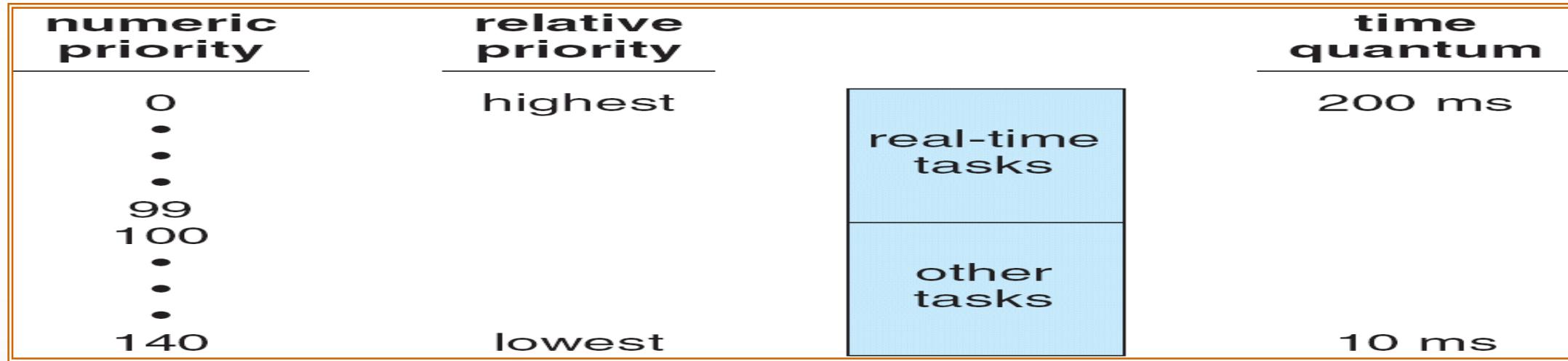
Time-sharing

- Prioritized credit-based – process with most credits is scheduled next
- Credit subtracted when timer interrupt occurs
- When credit = 0, another process chosen
- When all processes have credit = 0, re-crediting occurs based on factors including priority and history





Linux Scheduling

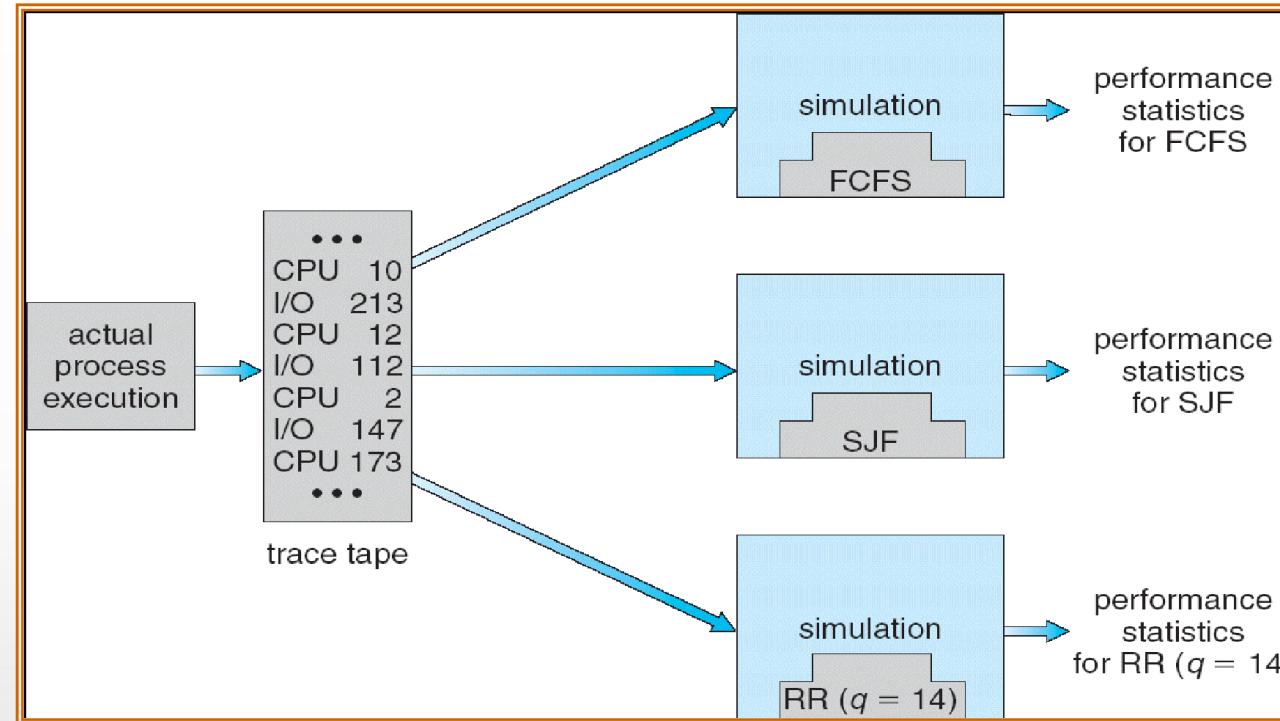


Real-time

- Soft real-time
- Posix.1b compliant – two classes (FCFS and RR)
- Highest priority process runs first



Algorithm Evaluation



How do we decide which algorithm is best for a particular environment?

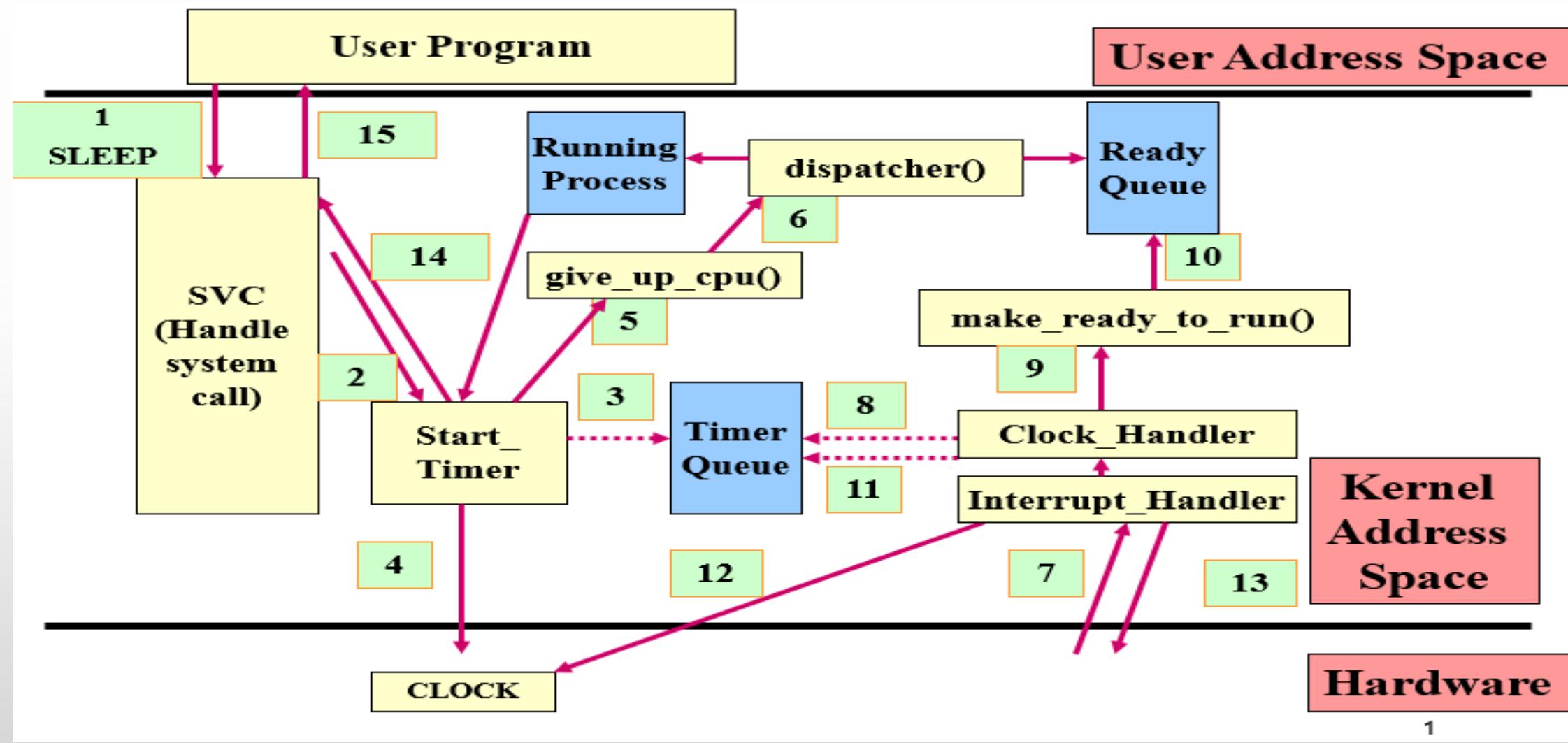
Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload.

Queueing models.





Sleep() Execution





Sleep() Execution

Step 1

- The user level program does a sleep() system call

Step 2

- The system call traps to the kernel, ending up in a predetermined location that's equipped to handle system calls. The system call handler determines the request is a sleep and passes the request to a routine Start_Timer





Sleep() Execution

Step 3

- The Start_Timer routine enques the PCB onto the timer queue

Step 4

- The Start_Timer routine invokes the hardware clock and asks it to interrupt at some time in the future

Step 5

- The process has nothing to do until the sleep completes, so it calls give_up_cpu().



Sleep() Execution

Step 6

- `give_up_cpu()` unloads the process from the processor and calls the dispatcher to find someone else to run. The dispatcher looks on the ready Q to find the next process to run. If there's no one to run, then it loops waiting for something to appear on the ready Q. If there is something there, run it

Step 7

- At some time, the hardware clock interrupts. The hardware interrupt routine determines that it's the clock that'd done the interrupt



Sleep() Execution

Step 8

- The clock interrupt handler goes to the Timer Queue and extracts the PCB for the process that originally generated the interrupt

Step 9

- The clock handler takes that PCB and calls `make_ready_to_run`

Step 10

- `make_ready_to_run` puts that PCB on the ready queue in the appropriate order





Sleep() Execution

Step 11

- The clock handler looks on the timer queue to see if some other process wants to use the timer

Step 12

- If yes, start the timer with the new request

Step 13

- The interrupt is now complete. The user process that was interrupted can now continue



Sleep() Execution

Step 14

- At some point the dispatcher is called on to find the next process to run. The PCB of the process that was sleeping is taken off the Ready Queue and is loaded on the processor

Step 15

- The subroutine calls now unwind, going back through start timer, back through the system call handler, and finally back to the user process. It starts there at the line after the sleep() call

Note the names used here are generic and are not meant to imply a particular routine on a particular OS.





Thank You Very Much

Benjamin Kommey

bkommey.coe@knust.edu.gh

nii_kommey@msn.com

050 770 32 86

Skype_id: calculus.affairs





“The world won't care about your self-esteem. The world will expect you to accomplish something BEFORE you feel good about yourself.”

Bill Gates





Overview

Synchronization

Background

Critical Section
Problem

Peterson's
Solution

Hardware
Synchronization





Overview

Synchronization

Semaphores

Classic
Synchronization
Problems

Synchronization
Examples

Atomic
Transactions





Background

Process Synchronization

is about getting processes to coordinate with each other.

How do processes work with resources that must be shared between them?

How do we go about acquiring locks to protect regions of memory?

How is synchronization really used?

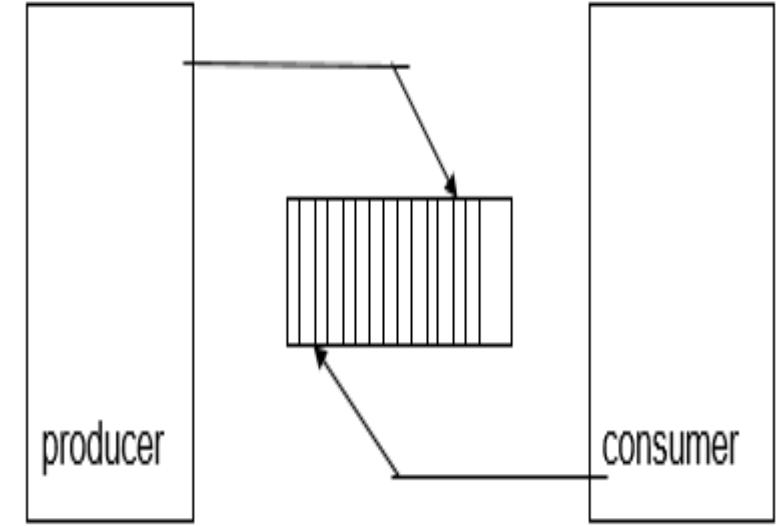




The Producer Consumer Problem

A **producer** process "produces" information "consumed" by a **consumer** process.

Here are the variables needed to define the problem:



```
#define BUFFER_SIZE 10
typedef struct {
    DATA          data;
} item;
item   buffer[BUFFER_SIZE];
int    in = 0;           // Location of next input to buffer
int    out = 0;          // Location of next removal from buffer
int    counter = 0;      // Number of buffers currently full
```



The Producer Consumer Problem

A producer process "produces" information "consumed" by a consumer process.

```
item    nextProduced;
while (TRUE) {
    while (counter ==
BUFFER_SIZE);
    buffer[in] =
nextProduced;
in = (in + 1) %
BUFFER_SIZE;
counter++;
}
```

```
item    nextConsumed;
while (TRUE) {
    while (counter == 0);
    nextConsumed =
buffer[out];
out = (out + 1) %
BUFFER_SIZE;
counter--;
}
```





The Producer Consumer Problem

Note that **counter++;** ← this line is NOT what it seems!!

is really --> register = counter

register = register + 1

counter = register

At a micro level, the following scenario could occur using this code:

TO;	Producer	Execute	register1 = counter	register1 = 5
T1;	Producer	Execute	register1 = register1 + 1	register1 = 6
T2;	Consumer	Execute	register2 = counter	register2 = 5
T3;	Consumer	Execute	register2 = register2 - 1	register2 = 4
T4;	Producer	Execute	counter = register1	counter = 6
T5;	Consumer	Execute	counter = register2	counter = 4





Critical Sections

A section of code, common to n cooperating processes, in which the processes may be accessing common variables.

A Critical Section Environment contains:

Entry Section

Code requesting entry into the critical section.

Critical Section

Code in which only one process can execute at any one time.

Exit Section The end of the critical section, releasing or allowing others in.

Remainder Section Rest of the code AFTER the critical section.





Critical Sections

The critical section must ENFORCE ALL THREE of the following rules:

Mutual Exclusion:

No more than one process can execute in its critical section at one time.

Progress:

If no one is in the critical section and someone wants in, then those processes not in their remainder section must be able to decide in a finite time who should go in.

Bounded Wait:

All requesters must eventually be let into the critical section.





Two Processes Software

Here's an example of a simple piece of code containing the components required in a critical section.

In this example, i is the current project, j is the other project. Same code running on two processors at the same time

Algorithm 1 : Toggled access

```
do {
```

```
    while ( turn ^= i ),
```

```
    /* critical section */
```

```
    turn = j;
```

```
    /* remainder section */
```

```
} while(TRUE);
```

Entry Section

Critical Section

Exit Section

Remainder Section

Are the three critical section requirements met?





Two Processes Software

Algorithm 2 Flag for each process gives state:

Each process maintains a flag that it wants to get into the critical section.

It checks the flag of the other process and doesn't enter the critical section if that other process wants to get in

Shared variables

- ☞ **boolean flag[2];**
initially **flag [0] = flag [1] = false.**
- ☞ **flag [i] = true** $\Rightarrow P_i$ ready to enter its critical section

```
do {  
    flag[i] := true;  
    while (flag[j]) ;  
    critical section  
    flag [i] = false;  
    remainder section  
} while (1);
```

Algorithm 2

Are the three Critical
Section Requirements Met?





Two Processes Software

Each processes sets a flag to request entry.

Then each process toggles a bit to allow the other in first. This code is executed for each process i.

Shared variables

- ☞ **boolean flag[2];**
initially **flag [0] = flag [1] = false.**
- ☞ **flag [i] = true** $\Rightarrow P_i$ ready to enter its critical section

```
do {  
    flag [i]:= true;  
    turn = j;  
    while (flag [j] and turn == j) ;  
    critical section  
    flag [i] = false;  
    remainder section  
} while (1);
```

Algorithm 3

Are the three Critical Section Requirements Met?

This is Peterson's Solution

12



Critical Sections

The hardware required to support critical sections must have (minimally):

- Indivisible instructions (what are they?)
- Atomic load, store, test instruction. For instance, if a store and test occur simultaneously, the test gets EITHER the old or the new, but not some combination.
- Two atomic instructions, if executed simultaneously, behave as if executed sequentially.





Hardware Solutions

Disabling Interrupts: Works for the Uni Processor case only.

WHY?

Atomic test and set: Returns parameter and sets parameter to true atomically.

```
while ( test_and_set ( lock ) );  
      /* critical section */  
lock = false;
```





Hardware Solutions

Example of Assembler code:

```
GET_LOCK:  IF_CLEAR_THEN_SET_BIT_AND_SKIP <bit_address>
            BRANCH  GET_LOCK          /* set failed */
            -----
                    /* set succeeded */
```

Must be careful if these approaches are to satisfy a bounded wait condition
- must use round robin - requires code built around the lock instructions.





Hardware Solutions

```
Boolean    waiting[N];  
int     j;           /* Takes on values from 0 to N - 1 */  
Boolean    key;  
do {  
    waiting[i] = TRUE;  
    key      = TRUE;  
    while( waiting[i] && key )  
        key = test_and_set( lock ); /* Spin lock */  
    waiting[ i ] = FALSE;
```





Hardware Solutions

***** CRITICAL SECTION *****

```
j = ( i + 1 ) mod N;
```

```
while( ( j != i ) && ( ! waiting[ j ] ) )
```

```
    j = ( j + 1 ) % N;
```

```
if ( j == i )
```

```
    lock = FALSE;
```

```
else
```

```
    waiting[ j ] = FALSE;
```

***** REMAINDER SECTION *****

```
} while (TRUE);
```





Current Hardware Dilemmas

We first need to define, for multiprocessors:

caches,

shared memory (for storage of lock variables),

write through cache,

write pipes.

The last software solution we did (the one we thought was correct) may not work on a cached multiprocessor. Why? { Hint, is the write by one processor visible immediately to all other processors? }

What changes must be made to the hardware for this program to work?





Current Hardware Dilemmas

a: A0

b: B0

Does the sequence (a , b) work on a cached multiprocessor?

Initially, location **a** contains A0 and location **b** contains B0.

- a) Processor 1 writes data A1 to location **a**.
- b) Processor 1 sets **b** to B1 indicating data at **a** is valid.
- c) Processor 2 waits for **b** to take on value B1 and loops until that change occurs.
- d) Processor 2 reads the value from **a**.

What value is seen by Processor 2 when it reads **a**?

How must hardware be specified to guarantee the value seen?





Current Hardware Dilemmas

We need to discuss:

Write Ordering:

The first write by a processor will be visible before the second write is visible. This requires a write through cache.

Sequential Consistency:

If Processor 1 writes to Location a "before" Processor 2 writes to Location b, then a is visible to ALL processors before b is. To do this requires NOT caching shared data.

The software solutions discussed earlier should be avoided since they require write ordering and/or sequential consistency.





Current Hardware Dilemmas

Hardware test and set on a multiprocessor causes

- an explicit flush of the write to main memory and
- the update of all other processor's caches.

Imagine needing to write **all** shared data straight through the cache.

With test and set, **only** lock locations are written out explicitly.

In not too many years, hardware will no longer support software solutions because of the performance impact of doing so.



Semaphores

PURPOSE:

We want to be able to write more complex constructs and so need a language to do so.

We thus define semaphores which we assume are atomic operations:

As given here, these are not atomic as written in "macro code". We define these operations, however, to be atomic (Protected by a hardware lock.)

FORMAT:

wait(mutex); <-- Mutual exclusion: mutex init to 1.

CRITICAL SECTION

signal(mutex);

REMAINDER



Semaphores

Semaphores can be used to force synchronization

(precedence) if the **preceeder** does a signal at the end, and the **follower** does wait at beginning.

For example, here we want P1 to execute before P2.

P1:

statement 1;
signal (synch);

P2:

wait (synch);
statement 2;



Semaphores

We don't want to loop on busy, so will suspend instead:

- Block on semaphore == False,
- Wakeup on signal (semaphore becomes True),
- There may be numerous processes waiting for the semaphore, so keep a list of blocked processes,
- Wakeup one of the blocked processes upon getting a signal

(choice of who depends on strategy).





Semaphores

To PREVENT looping,
we redefine the semaphore structure as:

```
typedef struct {  
    int             value;  
    struct process *list; /* linked list of PTBL waiting on S */  
} SEMAPHORE;
```



Semaphores

```
typedef struct {  
    int      value;  
    struct process *list; /* linked list of PTBL waiting on S */  
} SEMAPHORE;
```

- It's critical that these be atomic - in uniprocessors we can disable interrupts, but in multiprocessors other mechanisms for atomicity are needed.
- Popular incarnations of semaphores are as "event counts" and "lock managers".





Semaphores

DEADLOCKS:

May occur when two or more processes try to get the same multiple resources at the same time.

P1:

wait(S);

wait(Q);

.....

signal(S);

signal(Q);

P2:

wait(Q);

wait(S);

.....

signal(Q);

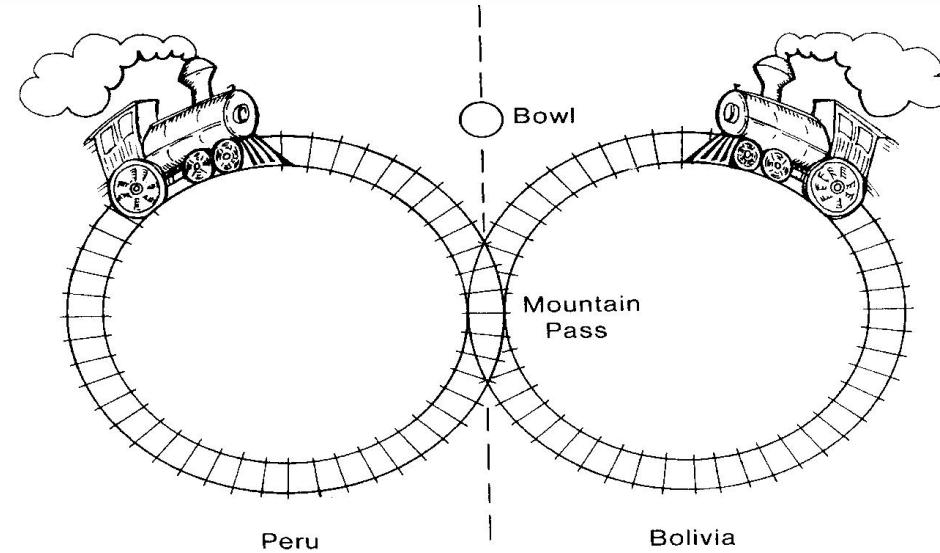
signal(S);

How can this be fixed?





A Practical Problem – Railways in the Andes



High in the Andes mountains, there are two circular railway lines. One line is in Peru, the other in Bolivia. They share a common section of track where the lines cross a mountain pass that lies on the international border (near Lake Titicaca?).

Unfortunately, the Peruvian and Bolivian trains occasionally collide when simultaneously entering the common section of track (the mountain pass). The trouble is, alas, that the drivers of the two trains are both **blind** and **deaf**, so they can neither see nor hear each other.





A Practical Problem – Railways in the Andes

The two drivers agreed on the following method of preventing collisions.

They set up a large bowl at the entrance to the pass.

Before entering the pass, a driver must stop his train, walk over to the bowl, and reach into it to see if it contains a rock.

If the bowl is empty, the driver finds a rock and drops it in the bowl, indicating that his train is entering the pass;

once his train has cleared the pass, he must walk back to the bowl and remove his rock, indicating that the pass is no longer being used.

Finally, he walks back to the train and continues down the line.



A Practical Problem – Railways in the Andes

If a driver arriving at the pass finds a rock in the bowl, he leaves the rock there; he repeatedly takes a siesta and rechecks the bowl until he finds it empty.

Then he drops a rock in the bowl and drives his train into the pass.

A smart graduate from the University of La Paz (Bolivia) claimed that subversive train schedules made up by Peruvian officials could block the train forever.

Explain

The Bolivian driver just laughed and said that could not be true because it never happened.





A Practical Problem – Railways in the Andes

Explain

Unfortunately, one day the two trains crashed.

Explain

Following the crash, the graduate was called in as a consultant to ensure that no more crashes would occur.

He explained that the bowl was being used in the wrong way.

The Bolivian driver must wait at the entry to the pass until the bowl is empty, drive through the pass and walk back to put a rock in the bowl.

The Peruvian driver must wait at the entry until the bowl contains a rock, drive through the pass and walk back to remove the rock from the bowl.





A Practical Problem – Railways in the Andes

Sure enough, his method prevented crashes.

Prior to this arrangement, the Peruvian train ran twice a day and the Bolivian train ran once a day. The Peruvians were very unhappy with the new arrangement.

Explain

The graduate was called in again and was told to prevent crashes while avoiding the problem of his previous method.

He suggested that two bowls be used, one for each driver.

When a driver reaches the entry, he first drops a rock in his bowl, then checks the other bowl to see if it is empty. If so, he drives his train through the pass.





A Practical Problem – Railways in the Andes

Stops and walks back to remove his rock.

But if he finds a rock in the other bowl, he goes back to his bowl and removes his rock. Then he takes a siesta, again drops a rock in his bowl and re-checks the other bowl, and so on, until he finds the other bowl empty.

This method worked fine until late in May, when the two trains were simultaneously blocked at the entry for many siestas.

Why? What is the problem? Explain – Homework!!!





The Bounded Buffer Problem

This is the same producer / consumer problem as before. But now we'll do it with signals and waits.

Remember: a **wait decreases** its argument and a **signal increases** its argument.

BINARY_SEMAPHORE mutex = 1;

// Can only be 0 or 1

COUNTING_SEMAPHORE empty = n; full = 0;

// Can take on any integer value

producer:

```
do {  
    /* produce an item in nextp */  
    wait (empty); /* Do action */  
    wait (mutex); /* Buffer guard */  
    /* add nextp to buffer */  
    signal (mutex);  
    signal (full);  
} while(TRUE);
```

consumer:

```
do {  
    wait (full);  
    wait (mutex);  
    /* remove an item from buffer to nextc */  
    signal (mutex);  
    signal (empty);  
    /* consume an item in nextc */  
} while(TRUE);
```



The Readers/ Writers Problem

This is the same as the Producer / Consumer problem except - we now can have many concurrent readers and one exclusive writer.

Locks: are **shared** (for the readers) and **exclusive** (for the writer).

Two possible (contradictory) guidelines can be used:

No reader is kept waiting unless a writer holds the lock (the readers have precedence).

If a writer is waiting for access, no new reader gains access (writer has precedence).
(NOTE: starvation can occur on either of these rules if they are followed rigorously.)





The Readers/ Writers Problem

```
BINARY_SEMAPHORE wrt      = 1;  
BINARY_SEMAPHORE mutex     = 1;  
int               readcount = 0;
```

Reader:

```
do {  
    wait( mutex );                      /* Allow 1 reader in entry*/  
    readcount = readcount + 1;  
    if readcount == 1 then wait(wrt); /* 1st reader locks writer */  
    signal( mutex );  
        /* reading is performed */  
    wait( mutex );  
    readcount = readcount - 1;  
    if readcount == 0 then signal(wrt); /*last reader frees writer */  
    signal( mutex );  
} while(TRUE);
```

Writer:

```
do {  
    wait( wrt );  
    /* writing is performed */  
    signal( wrt );  
} while(TRUE);
```

WAIT (S):

```
while ( S <= 0 );  
S = S - 1;
```

SIGNAL (S):

```
S = S + 1;
```



The Dining Philosophers Problems

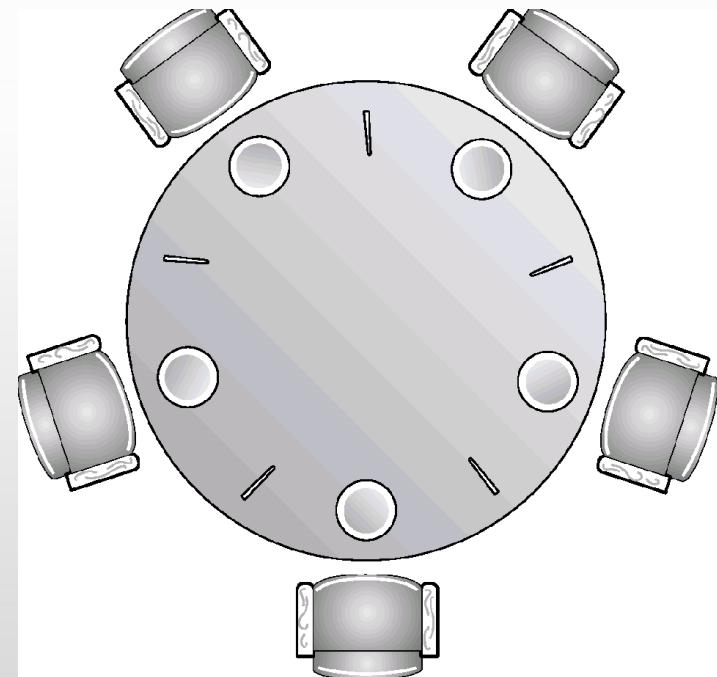
5 philosophers with 5 chopsticks sit around a circular table.

They each want to eat at random times and must pick up the chopsticks on their right and on their left.

Clearly deadlock is rampant (and starvation possible.)

Several solutions are possible:

- Allow only 4 philosophers to be hungry at a time.
- Allow pickup only if both chopsticks are available.
(Done in critical section)
- Odd # philosopher always picks up left chopstick 1st, even # philosopher always picks up right chopstick 1st.





Critical Regions

High Level synchronization construct implemented in a programming language.

A shared variable v of type T , is declared as:

```
var v; shared T
```

Variable v is accessed only inside a statement:

```
region v when B do S;
```

where B is a Boolean expression.

Entry Section

Shared Data

Exit Section

While statement S is being executed, no other process can access variable v .

Critical Region





Critical Regions

Regions referring to the same shared variable exclude each other in time.

When a process tries to execute the region statement, the Boolean expression B is evaluated.

If B is true, statement S is executed.

If it is false, the process is delayed until B is true and no other process is in the region associated with v.

Entry Section

Shared Data

Exit Section

Critical Region



Critical Regions

EXAMPLE: Bounded Buffer:

Shared variables declared as:

```
struct buffer {
    int      pool[n];
    int      count, in, out;
}
```

Producer process inserts **nextp** into the shared buffer:

```
region buffer when( count < n) {
    pool[in] = nextp;
    in:= (in+1) % n;
    count++;
}
```

Consumer process removes an item from the shared buffer and puts it in **nextc**.

```
Region buffer when (count > 0) {
    nextc = pool[out];
    out = (out+1) % n;
    count--;
}
```





Monitors

High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) { ... }
    procedure body P2 (...) { ... }
    procedure body Pn (...) { ... }
    {
        initialization code
    }
}
```





Monitors

To allow a process to wait within the monitor, a **condition** variable must be declared, as

condition x, y;

Condition variable can only be used with the operations **wait** and **signal**.

The operation

x.wait();

means that the process invoking this operation is suspended until another process invokes

x.signal();

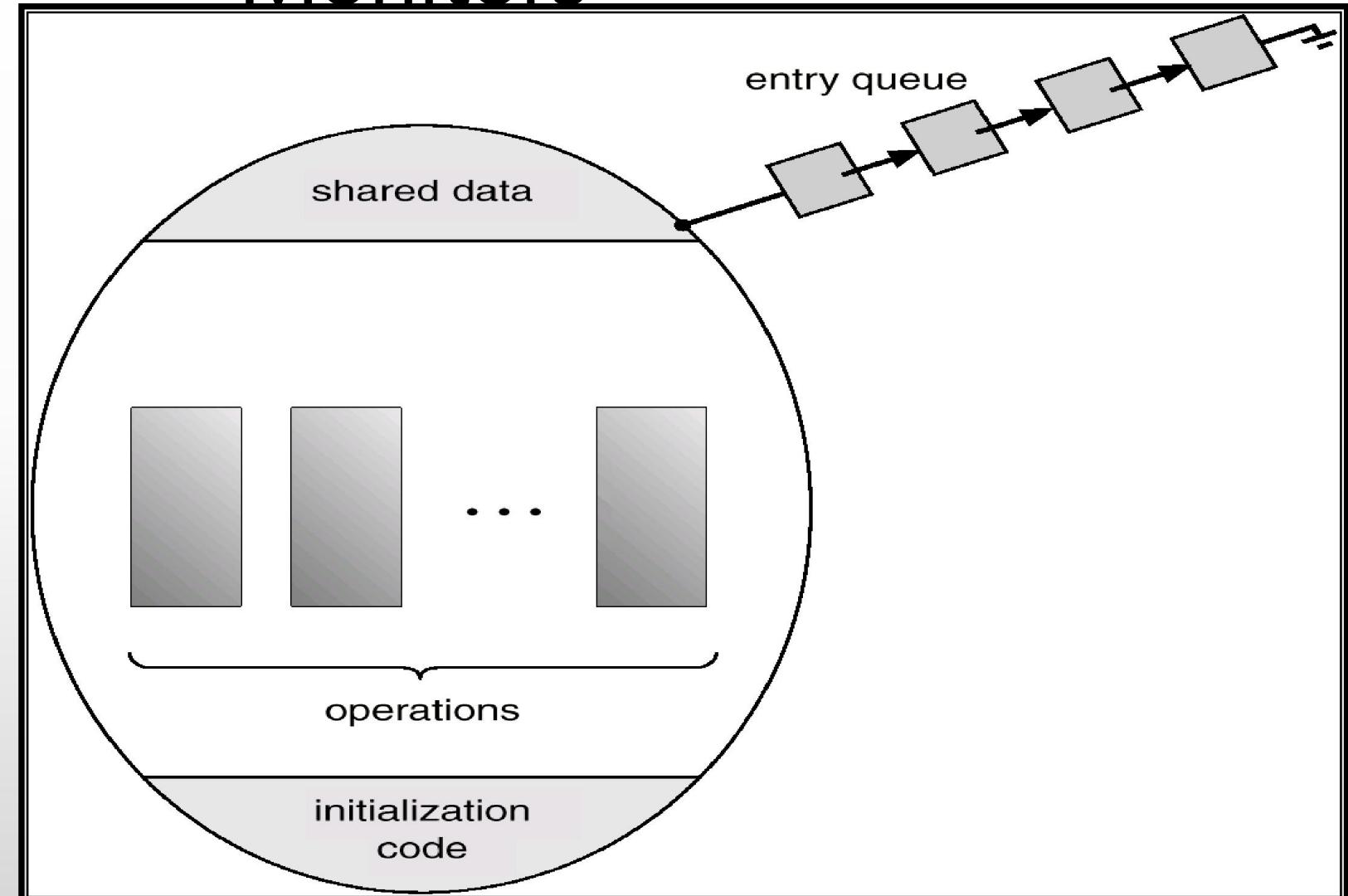
The **x.signal** operation resumes exactly one suspended process.

If no process is suspended, then the **signal** operation has no effect.





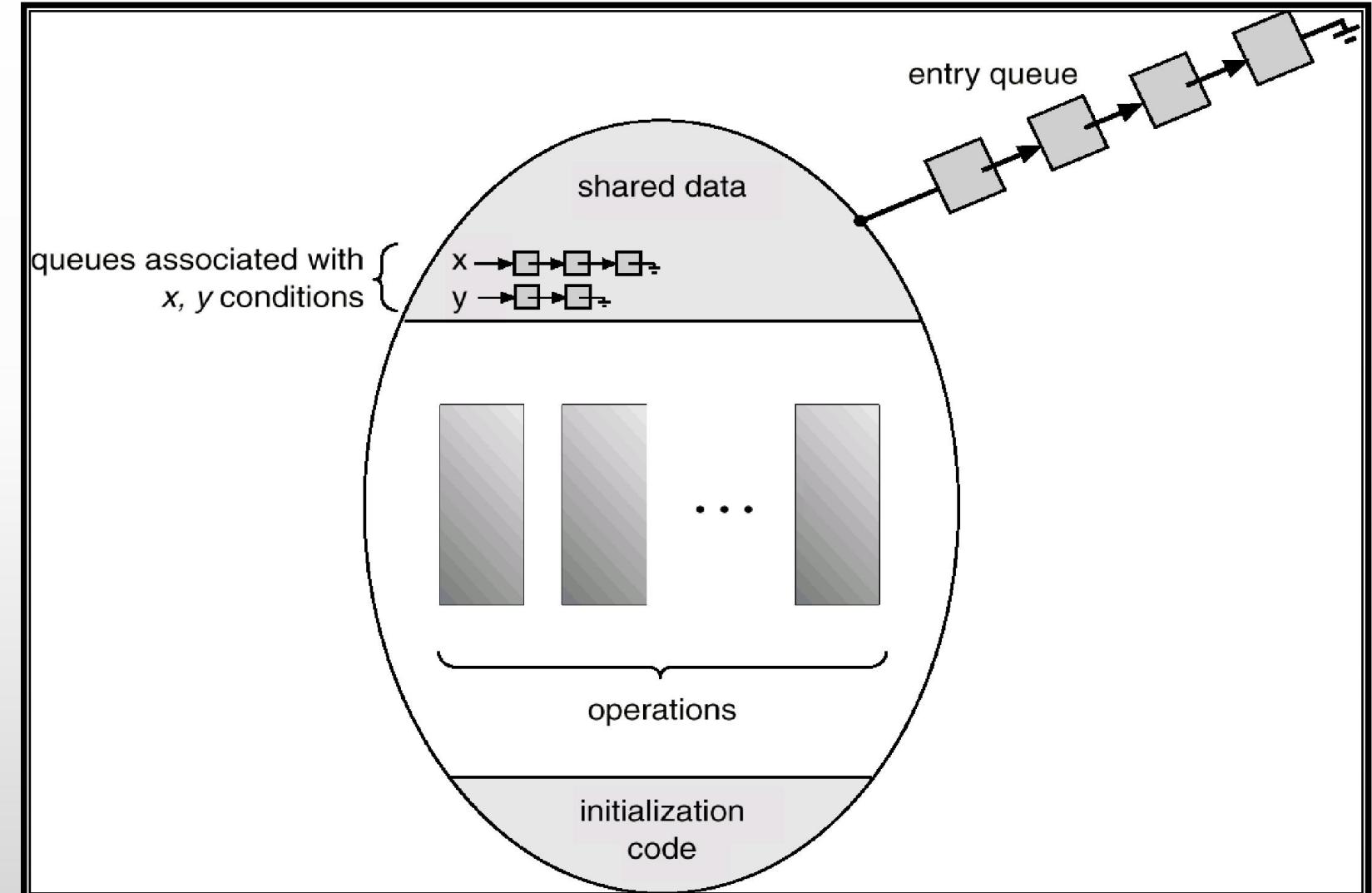
Schematic View of a Monitor





Monitors

Monitor With
Condition Variables





Monitors

```
monitor dp {  
    enum state{thinking, hungry, eating};  
    state[5];  
    condition self[5];  
}
```

```
initializationCode() {  
    for ( int i = 0; i < 5; i++ )  
        state[i] = thinking;  
}
```

```
void test(int i) {  
    if ( (state[(i + 4) % 5] != eating) && (state[i] == hungry) && (state[(i + 1) % 5] != eating)) {  
        state[i] = eating;  
        self[i].signal();  
    }  
}
```

Dining Philosophers Example

```
void putdown(int i) {  
    state[i] = thinking;  
    // test left & right neighbors  
    test((i+4) % 5);  
    test((i+1) % 5);  
}
```

```
void pickup(int i) {  
    state[i] = hungry;  
    test[i];  
    if (state[i] != eating)  
        self[i].wait();  
}
```

Windows XP Synchronization

Uses interrupt masks to protect access to global resources on uniprocessor systems.

Uses *spinlocks* on multiprocessor systems.

Also provides *dispatcher objects* which may act as either mutexes or semaphores.

Dispatcher objects may also provide *events*. An event acts much like a condition variable.





Thank You Very Much

Benjamin Kommey

bkommey.coe@knust.edu.gh

nii_kommey@msn.com

050 770 32 86

Skype_id: calculus.affairs





“ If you think your teacher is tough,
wait till you get a boss. ”

Bill Gates





Overview

Deadlock

Background

Bridge
Crossing
Example

Deadlock
Characterisa-
tion

Resource
Allocation
Graph

Deadlock
Handling
Strategies





Overview

Deadlock Handling Strategies

Deadlock
Prevention

Deadlock
Avoidance

Deadlock
Detection

Deadlock
Recovery





Deadlocks

What is a deadlock?

**Staying Safe:
Preventing and Avoiding Deadlocks**

**Living Dangerously:
Let the deadlock happen, then detect it and recover from it.**





Deadlocks

EXAMPLES:

"It takes money to make money".

You can't get a job without experience; you can't get experience without a job.





Deadlocks

BACKGROUND:

The cause of deadlocks:

Each process needing what another process has.

This results from sharing resources such as memory, devices, links.

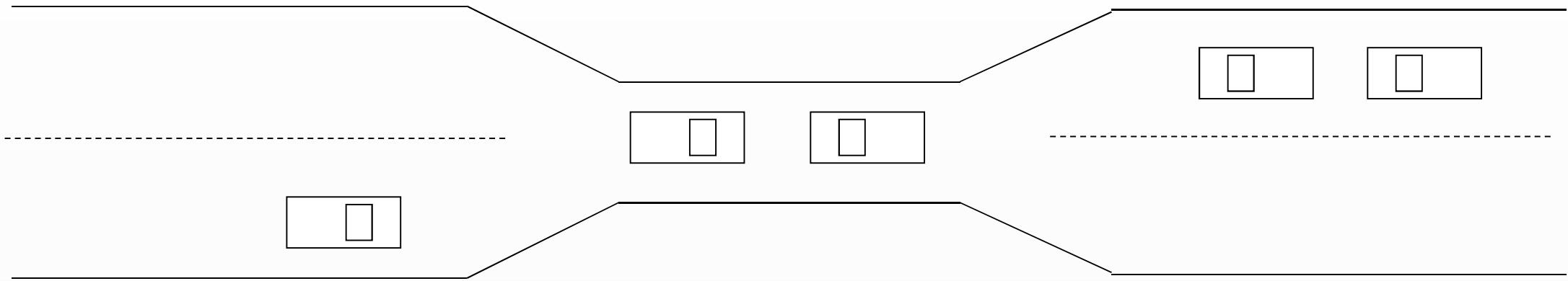
Under normal operation, a resource allocations proceed like this::

1. Request a resource (suspend until available if necessary).
2. Use the resource.
3. Release the resource.





Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.





Deadlock Characterisation

Deadlock
Necessary
Conditions

All of the following four conditions must happen simultaneously for deadlock to occur

Mutual
exclusion

One or more than one resource must be held by a process in a non-sharable (exclusive) mode.

Hold and
Wait

A process holds a resource while waiting for another resource





Deadlock Characterisation

No
Preemption

There is only voluntary release of a resource - nobody else can make a process give up a resource

Circular
Wait

Process A waits for Process B waits for Process C waits for Process A





Resource Allocation Graph

A visual (mathematical) way to determine if a deadlock has, or may occur.

$G = (V, E)$ The graph contains nodes and edges.

V Nodes consist of processes = { P1, P2, P3, ... } and resource types { R1, R2, ... }

E Edges are (Pi, Rj) or (Ri, Pj)

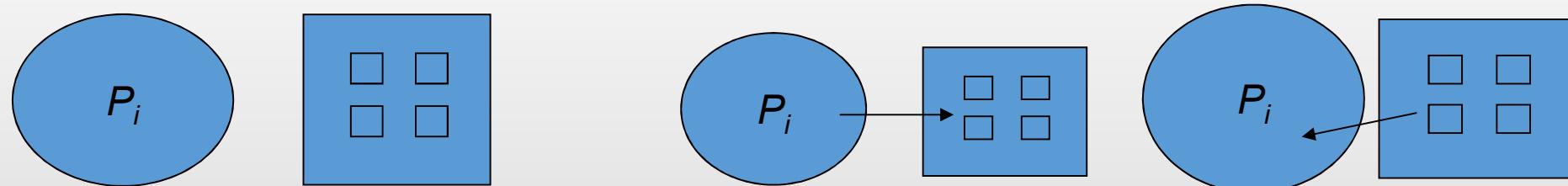
An arrow from the **process** to **resource** indicates the process is **requesting** the resource. An arrow from **resource** to **process** shows an instance of the resource has been **allocated** to the process.





Resource Allocation Graph

Process is a circle, resource type is square; dots represent number of instances of resource in type. Request points to square, assignment comes from dot.





Resource Allocation Graph

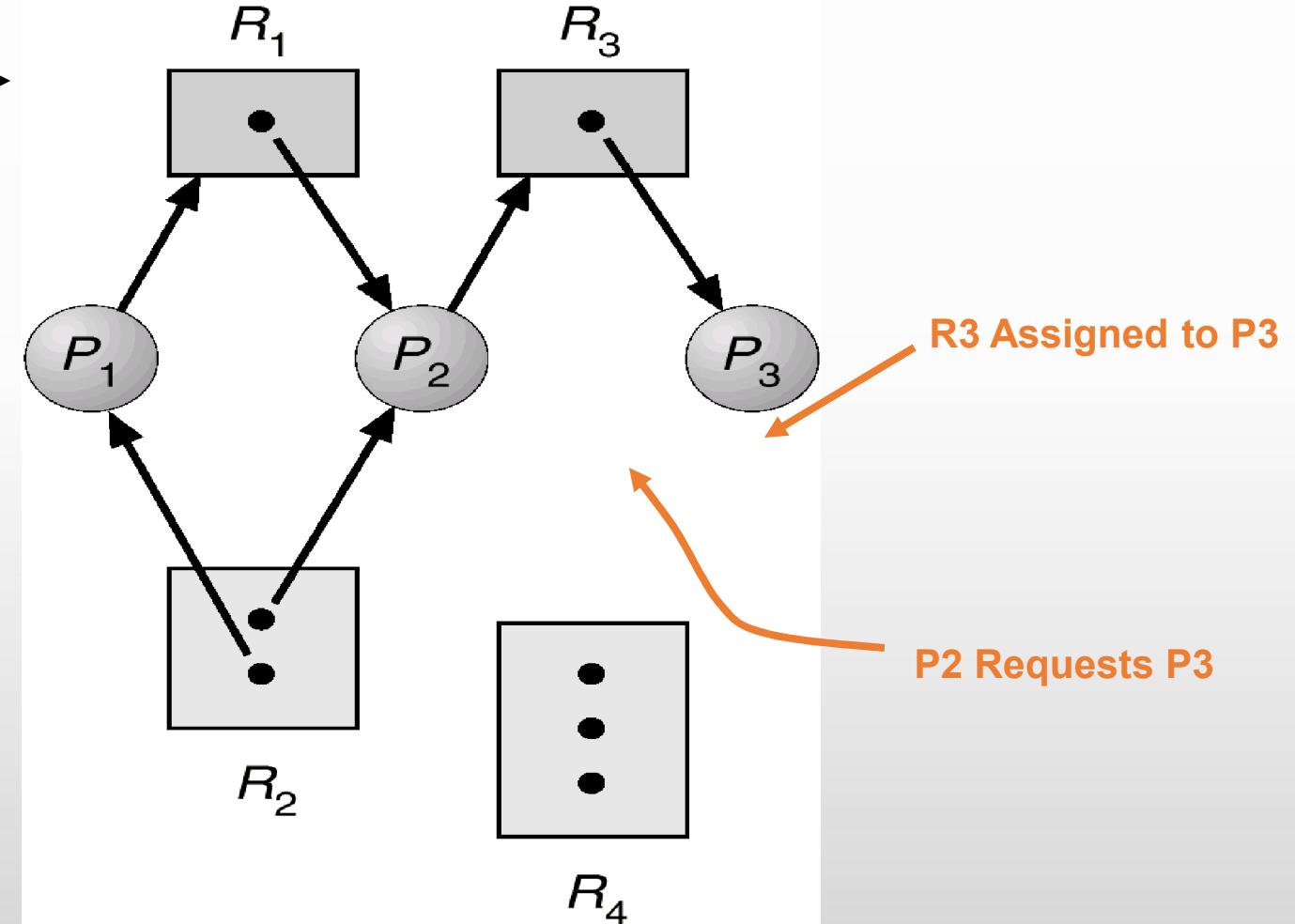
- If the graph contains no cycles, then no process is deadlocked.
- If there is a cycle, then:
 - a) If resource types have multiple instances, then deadlock MAY exist.
 - b) If each resource type has 1 instance, then deadlock has occurred.





Resource Allocation Graph

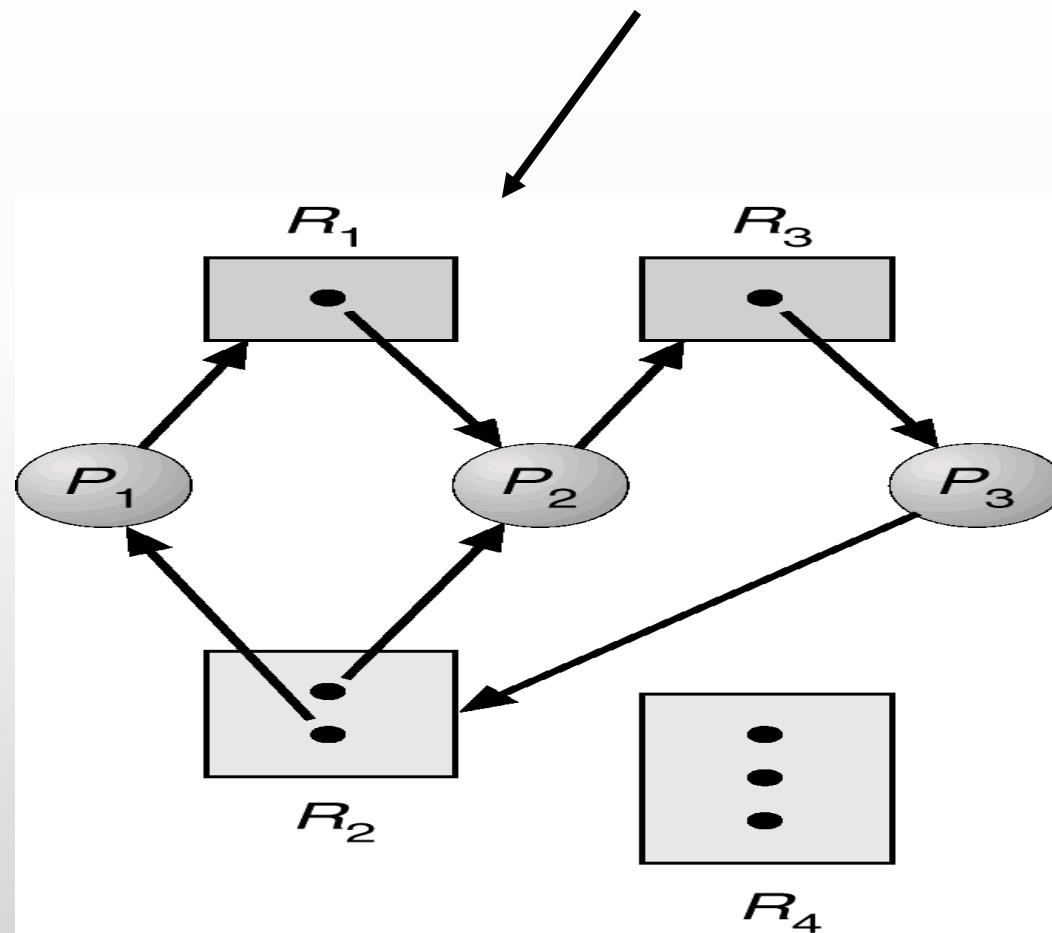
Resource allocation graph →



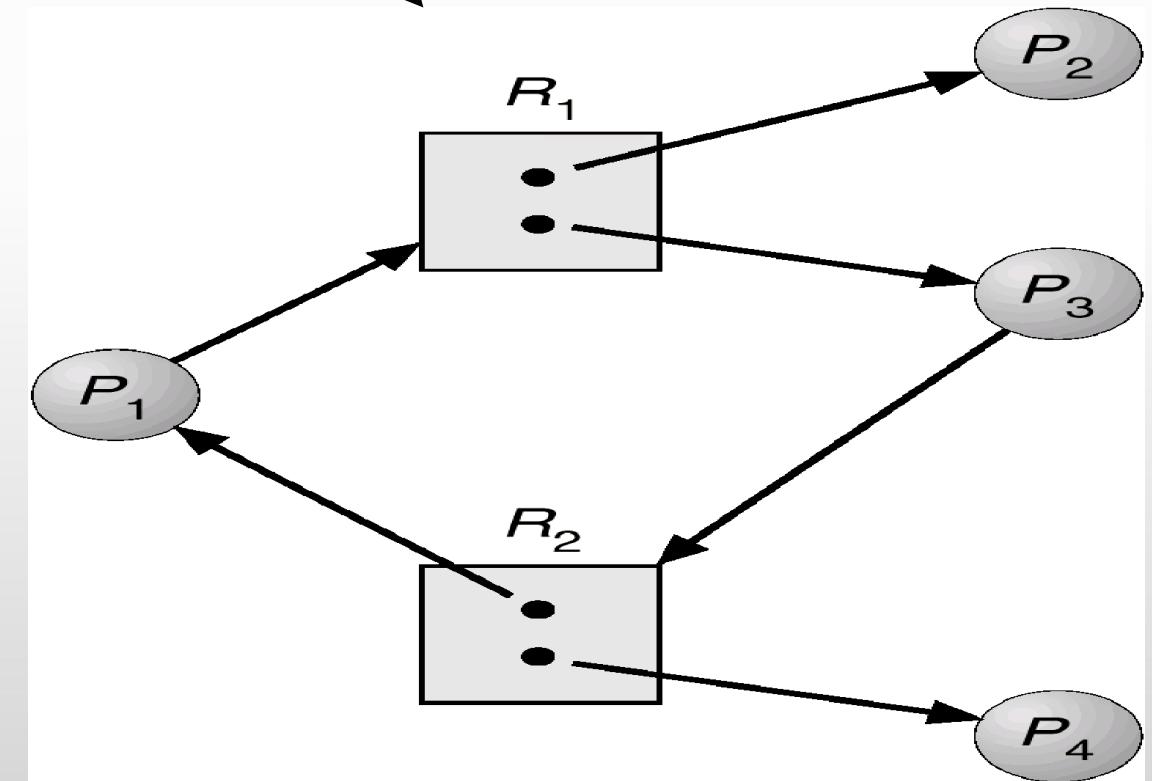


Resource Allocation Graph

Resource allocation graph
with a deadlock.



Resource allocation graph
with a cycle but no deadlock.





Deadlock Handling Strategies

There are three methods:

Ignore Deadlocks: (done by most operating systems)

Ensure deadlock **never** occurs using either

Prevention Prevent any one of the 4 conditions from happening.

Avoidance Allow all deadlock conditions, but calculate cycles about to happen and stop dangerous operations..

Allow deadlock to happen.

This requires using both:

Detection Know a deadlock has occurred.

Recovery Regain the resources.





Deadlock Prevention

Do not allow one of the four conditions to occur.

Mutual exclusion:

- a) Automatically holds for printers and other non-shareables.
- b) Shared entities (read only files) don't need mutual exclusion
(and aren't susceptible to deadlock.)
- c) Prevention not possible, since some devices are intrinsically non-shareable.





Deadlock Prevention

Hold and wait:

- a) Collect all resources before execution.
- b) A particular resource can only be requested when no others are being held.

A sequence of resources is always collected beginning with the same one.

- c) Utilization is low, starvation possible.





Deadlock Prevention

Do not allow one of the four conditions to occur.

No preemption:

- a) Release any resource already being held if the process can't get an additional resource.
- b) Allow preemption - if a needed resource is held by another process, which is also waiting on some resource, steal it. Otherwise wait.





Deadlock Prevention

Circular wait:

- a) Number resources and only request in ascending order.
- b) EACH of these prevention techniques may cause a decrease in utilization and/or resources. For this reason, prevention isn't necessarily the best technique.
- c) Prevention is generally the easiest to implement.





Deadlock Avoidance

If we have prior knowledge of how resources will be requested, it's possible to determine if we are entering an "unsafe" state.

Possible states are:

Deadlock No forward progress can be made.

Unsafe state A state that **may** allow deadlock.

Safe state A state is safe if a sequence of processes exist such that there are enough resources for the first to finish, and as each finishes and releases its resources there are enough for the next to finish.





Deadlock Avoidance

The rule is simple:

If a request allocation would cause an unsafe state, do not honor that request.

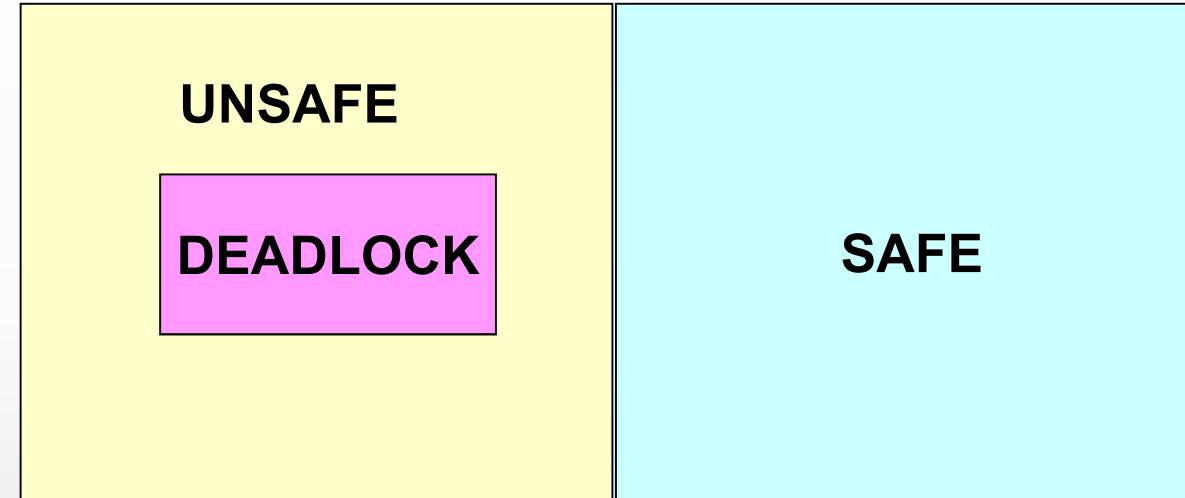
NOTE: All deadlocks are unsafe, but all unsafes are NOT deadlocks.





Deadlock Avoidance

NOTE: All deadlocks are unsafe, but all unsafes are NOT deadlocks.



Only with luck will processes avoid deadlock.

SAFE

O.S. can avoid deadlock.

Let's assume a very simple model: each process declares its maximum needs. In this case, algorithms exist that will ensure that no unsafe state is reached.



Deadlock Avoidance

EXAMPLE:

There are multiple instances of the resource in these examples.

There exists a total of 12 tape drives. The current state looks like this:

In this example, $\langle p_1, p_0, p_2 \rangle$ is a workable sequence.

Suppose p_2 requests and is given one more tape drive. What happens then?

Process	Max Needs	Allocated	Current Needs
P0	10	5	5
P1	4	2	2
P2	9	2	7





Deadlock Avoidance

Safety Algorithm

A method used to determine if a particular state is safe.

It's safe if there exists a sequence of processes such that for all the processes, there's a way to avoid deadlock:

The algorithm uses these variables:

Need[I]

the remaining resource needs of each process.

Work

Temporary variable – how many of the resource are currently available.





Deadlock Avoidance

Safety Algorithm

Finish[I] – flag for each process showing we've analyzed that process or not.

$\text{need} \leq \text{available} + \text{allocated}[0] + \dots + \text{allocated}[I-1]$

<- Sign of success

Let **work** and **finish** be vectors of length **m** and **n** respectively.





Deadlock Avoidance

Safety Algorithm

1. Initialize work = available
Initialize finish[i] = false, for i = 1,2,3,..n

2. Find an i such that:
finish[i] == false and need[i] <= work
If no such i exists, go to step 4.

3. work = work + allocation[i]
finish[i] = true
goto step 2

4. if finish[i] == true for all i, then the system is in a safe state.





Deadlock Avoidance

Safety Algorithm

Consider a system with five processes, P0 – P4, three resource types A, B and C.

Type A has 10 instances, B has 5 instances, C has 7 instances.

At time T0 the following snapshot of the system is taken

Is the system in a safe state?

Max Needs = allocated + can-be-requested

	Alloc			Req			Avail		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	3	3	2
P1	2	0	0	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			





Deadlock Avoidance

Safety Algorithm

Now try it again with only a slight change in the request by P1.

P1 requests one additional resource of type A, and two more of type C.

$$\text{Request1} = (1,0,2)$$

Is Request1 < available?

Produce the state chart as if the request is Granted and see if it's safe. (We've drawn the chart as if it's granted.)

Can the request be granted?

	←	Alloc	→	←	Req	→	←	Avail	→
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	1#	3	0#
P1	3#	0	2#	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			





Deadlock Detection

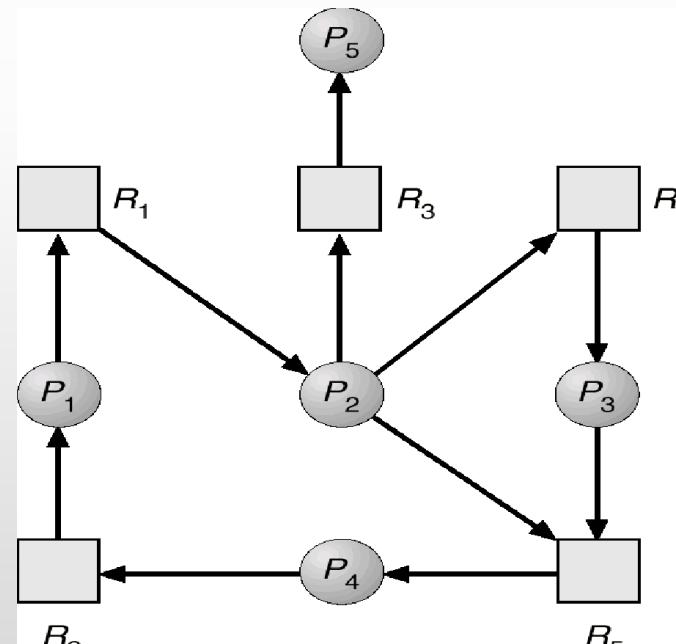
Need an algorithm that determines if deadlock occurred.

Also need a means of recovering from that deadlock.

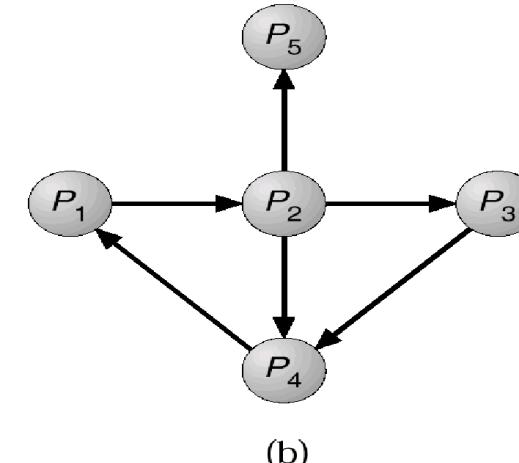
SINGLE INSTANCE OF A RESOURCE TYPE

Wait-for graph == remove the resources from the usual graph and collapse edges.

An edge from $p(j)$ to $p(i)$ implies that $p(j)$ is waiting for $p(i)$ to release.



(a)



(b)



Deadlock Detection

SEVERAL INSTANCES OF A RESOURCE TYPE

Complexity is of order $m * n * n$.

We need to keep track of:

- | | |
|-------------------|--|
| available | - records how many resources of each type are available. |
| allocation | - number of resources of type m allocated to process n. |
| request | - number of resources of type m requested by process n. |

Let **work** and **finish** be vectors of length **m** and **n** respectively.





Deadlock Detection

1. Initialize **work[] = available[]**
For i = 1,2,...n, if allocation[i] != 0 then
finish[i] = false; otherwise, finish[i] = true;
2. Find an i such that:
finish[i] == false and request[i] <= work
If no such i exists, go to step 4.
3. **work = work + allocation[i]**
finish[i] = true
goto step 2
4. **if finish[i] == false for some i, then the system is in deadlock state.**
IF finish[i] == false, then process p[i] is deadlocked.





Deadlock Detection

Example

We have three resources A, B, and C.

A has 7 instances, B has 2 instances and C has 6 instances.

At this time, the allocation, etc. Looks like this:

Is there a sequence that will allow deadlock to be avoided?

Is there more than one sequence that will work?

	←	Alloc	→	←	Req	→	←	Avail	→
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			





Deadlock Detection

Example

Suppose the Request matrix is changed like this.

In other words, the maximum amounts to be allocated are initially declared so that this request matrix results

Is there now a sequence that will allow deadlock to be avoided?

USAGE OF THIS DETECTION ALGORITHM

Frequency of check depends on how often a deadlock occurs and how many processes will be affected.

	←	Alloc	→	←	Req	→	←	Avail	→
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	1#			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			



Deadlock Recovery

So, the deadlock has occurred. Now, how do we get the resources back and gain forward progress?

PROCESS TERMINATION:

- Could delete all the processes in the deadlock -- this is expensive.
- Delete one at a time until deadlock is broken (time consuming).
- Select who to terminate based on priority, time executed, time to completion, needs for completion, or depth of rollback
- In general, it's easier to preempt the resource, than to terminate the process.





Deadlock Recovery

RESOURCE PREEMPTION:

- Select a victim - which process and which resource to preempt.
- Rollback to previously defined "safe" state.
- Prevent one process from always being the one preempted (starvation).





Deadlock Recovery

COMBINED APPROACH TO DEADLOCK HANDLING:

- Type of resource may dictate best deadlock handling.
 Look at ease of implementation, and effect on performance.
- In other words, there is no one best technique.
- Cases include:

Preemption for memory,

Preallocation for swap space,

Avoidance for devices (can extract Needs from process.)





Thank You Very Much

Benjamin Kommey

bkommey.coe@knust.edu.gh

nii_kommey@msn.com

050 770 32 86

Skype_id: calculus.affairs





“ It's fine to celebrate success, but it is more important to heed the lessons of failure.”

Bill Gates





Overview

Memory Management

Definitions

Logical-
Physical
Binding

Allocation

Long Term
Scheduling





Overview

Memory Management

Compaction

Paging

Segmentation

Intel Memory Management



Overview

Memory Management

Allocation

Single Partition Allocation

Contiguous Allocation





Memory Management

Just as processes share the CPU, they also share physical memory.
This section is about mechanisms for doing that sharing.

EXAMPLE OF MEMORY USAGE:

Calculation of an **effective address**

- Fetch from instruction
- Use index offset





Memory Management

Example: (Here index is a pointer to an address)

loop:

load	register, index
add	42, register
store	register, index
inc	index
skip_equal	index, final_address
branch	loop
... continue	



Definitions

The concept of a logical *address space* that is bound to a separate *physical address space* is central to proper memory management.

Logical address

generated by the CPU;
also referred to as *virtual address*

Physical address

address seen by the memory unit



Definitions

Logical and physical addresses are the same in compile-time and load-time address-binding schemes;

Logical (virtual) and physical addresses differ in execution-time address-binding scheme





Definitions

Relocatable

- Means that the program image can reside anywhere in physical memory

Binding

- Programs need real memory in which to reside. When is the location of that real memory determined?
- This is called **mapping** logical to physical addresses.
- This binding can be done at compile/link time. Converts symbolic to relocatable. Data used within compiled source is offset within object module





Definitions

Compiler

- If it's known where the program will reside, then absolute code is generated. Otherwise compiler produces re-locatable code

Load

- Binds relocatable to physical. Can find best physical location

Execution

- The code can be moved around during execution. Means flexible virtual mapping



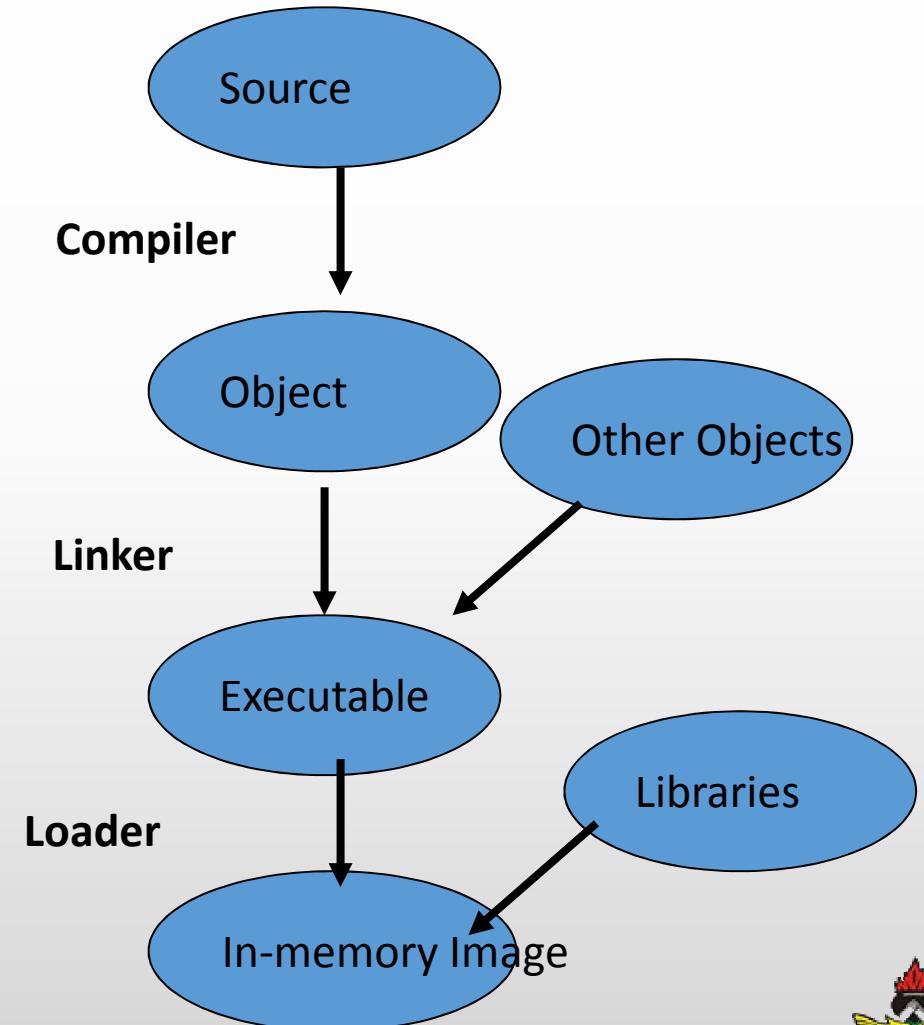


Binding Logical To Physical

This binding can be done at compile/link time. Converts symbolic to relocatable. Data used within compiled source is offset within object module.

- Can be done at load time. Binds relocatable to physical.
- Can be done at run time. Implies that the code can be moved around during execution.

The next example shows how a compiler and linker actually determine the locations of these effective addresses.





Binding Logical To Physical

```
void main()
{
    printf( "Hello, from main\n" );
    b();
}

void b()
{
    printf( "Hello, from 'b'\n" );
}
```



Binding Logical To Physical

ASSEMBLY LANGUAGE LISTING

000000B0:	6BC23FD9	stw	%r2,-20(%sp) ; main()
000000B4	37DE0080	ldo	64(%sp),%sp
000000B8	E8200000	bl	0x000000C0,%r1 ; get current addr=BC
000000BC	D4201C1E	depi	0,31,2,%r1
000000C0	34213E81	ldo	-192(%r1),%r1 ; get code start area
000000C4	E8400028	bl	0x000000E0,%r2 ; call printf
000000C8	B43A0040	addi	32,%r1,%r26 ; calc. String loc.
000000CC	E8400040	bl	0x000000F4,%r2 ; call b
000000D0	6BC23FD9	stw	%r2,-20(%sp) ; store return addr
000000D4	4BC23F59	ldw	-84(%sp),%r2
000000D8	E840C000	bv	%r0(%r2) ; return from main
000000DC	37DE3F81	ldo	-64(%sp),%sp
			STUB(S) FROM LINE 6
000000E0:	E8200000	bl	0x000000E8,%r1
000000E4	28200000	addil	L%0,%r1
000000E8:	E020E002	be,n	0x00000000(%sr7,%r1)





Binding Logical To Physical

ASSEMBLY LANGUAGE LISTING

000000EC	08000240	nop	void b()
000000F0:	6BC23FD9	stw	%r2,-20(%sp)
000000F4:	37DE0080	ldo	64(%sp),%sp
000000F8	E8200000	bl	0x00000100,%r1 ; get current addr=F8
000000FC	D4201C1E	depi	0,31,2,%r1
00000100	34213E01	ldo	-256(%r1),%r1 ; get code start area
00000104	E85F1FAD	bl	0x000000E0,%r2 ; call printf
00000108	B43A0010	addi	8,%r1,%r26
0000010C	4BC23F59	ldw	-84(%sp),%r2
00000110	E840C000	bv	%r0(%r2) ; return from b
00000114	37DE3F81	ldo	-64(%sp),%sp





Binding Logical To Physical

EXECUTABLE IS DISASSEMBLED HERE

00002000	0009000F	;		
00002004	08000240	;	. . . @		
00002008	48656C6C	;	H e l l		
0000200C	6F2C2066	;	o , f		
00002010	726F6D20	;	r o m		
00002014	620A0001	;	b . . .		
00002018	48656C6C	;	H e l l		
0000201C	6F2C2066	;	o , f		
00002020	726F6D20	;	r o m		
00002024	6D61696E	;	m a i n		
000020B0	6BC23FD9	stw	%r2,-20(%sp)	;	main



Binding Logical To Physical

EXECUTABLE IS DISASSEMBLED HERE

000020B0	6BC23FD9	stw	%r2,-20(%sp)	; main
000020B4	37DE0080	ldo	64(%sp),%sp	
000020B8	E8200000	bl	0x000020C0,%r1	
000020BC	D4201C1E	depi	0,31,2,%r1	
000020C0	34213E81	ldo	-192(%r1),%r1	
000020C4	E84017AC	bl	0x00003CA0,%r2	
000020C8	B43A0040	addi	32,%r1,%r26	
000020CC	E8400040	bl	0x000020F4,%r2	
000020D0	6BC23FD9	stw	%r2,-20(%sp)	
000020D4	4BC23F59	ldw	-84(%sp),%r2	
000020D8	E840C000	bv	%r0(%r2)	
000020DC	37DE3F81	ldo	-64(%sp),%sp	
000020E0	E8200000	bl	0x000020E8,%r1	; stub
000020E4	28203000	addil	L%6144,%r1	
000020E8	E020E772	be,n	0x000003B8(%sr7,%r1)	
000020EC	08000240	nop		



Binding Logical To Physical

EXECUTABLE IS DISASSEMBLED HERE

```

000020F0 6BC23FD9 stw    %r2,-20(%sp)      ; b
000020F4 37DE0080 ldo      64(%sp),%sp
000020F8 E8200000 bl       0x00002100,%r1
000020FC D4201C1E depi    0,31,2,%r1
00002100 34213E01 ldo      -256(%r1),%r1
00002104 E840172C bl       0x00003CA0,%r2
00002108 B43A0010 addi    8,%r1,%r26
0000210C 4BC23F59 ldw     -84(%sp),%r2
00002110 E840C000 bv      %r0(%r2)
00002114 37DE3F81 ldo     -64(%sp),%sp

```

EXECUTABLE IS DISASSEMBLED HERE

```

00003CA0 6BC23FD9 stw    %r2,-20(%sp)      ; printf
00003CA4 37DE0080 ldo      64(%sp),%sp
00003CA8 6BDA3F39 stw    %r26,-100(%sp)
00003CAC 2B7CFFFF addil   L%-26624,%dp
00003CBO 6BD93F31 stw    %r25,-104(%sp)
00003CB4 343301A8 ldo     212(%r1),%r19
00003CB8 6BD83F29 stw    %r24,-108(%sp)
00003CBC 37D93F39 ldo     -100(%sp),%r25
00003CC0 6BD73F21 stw    %r23,-112(%sp)
00003CC4 4A730009 ldw     -8188(%r19),%r19
00003CC8 B67700D0 addi   104,%r19,%r23
00003CCC E8400878 bl      0x00004110,%r2
00003CD0 08000258 copy   %r0,%r24
00003CD4 4BC23F59 ldw     -84(%sp),%r2
00003CD8 E840C000 bv      %r0(%r2)
00003CDC 37DE3F81 ldo     -64(%sp),%sp
00003CEO E8200000 bl      0x00003CE8,%r1
00003CE8 E020E852 be,n    0x00000428(%sr7,%r1)

```





More Definitions

Dynamic loading

- + Routine is not loaded until it is called
- + Better memory-space utilization; unused routine is never loaded.
- + Useful when large amounts of code are needed to handle infrequently occurring cases.
- + No special support from the OS is required - implemented through program design.





More Definitions

Dynamic Linking

- + Linking postponed until execution time.
- + Small piece of code, *stub*, used to locate the appropriate memory-resident library routine.
- + Stub replaces itself with the address of the routine, and executes the routine.
- + Operating system needed to check if routine is in processes' memory address.
- + Dynamic linking is particularly useful for libraries.

Memory Management

Performs the above operations. Usually requires hardware support.





Single Partition Allocation

BARE MACHINE:

- No protection, no utilities, no overhead.
- This is the simplest form of memory management.
- Used by hardware diagnostics, by system boot code, real time/dedicated systems.
- logical == physical
- User can have complete control. Commensurably, the operating system has none.





Single Partition Allocation

DEFINITION OF PARTITIONS:

- Division of physical memory into fixed sized regions. (Allows addresses spaces to be distinct = one user can't muck with another user, or the system.)
- The number of partitions determines the level of multiprogramming. Partition is given to a process when it's scheduled.
- Protection around each partition determined by bounds (upper, lower) base / limit.
- These limits are done in hardware.



Single Partition Allocation

RESIDENT MONITOR:

- Primitive Operating System.
- Usually in low memory where interrupt vectors are placed.
- Must check each memory reference against fence (fixed or variable) in hardware or register. If user generated address < fence, then illegal.
- User program starts at fence -> fixed for duration of execution. Then user code has fence address built in. But only works for static-sized monitor.





Single Partition Allocation

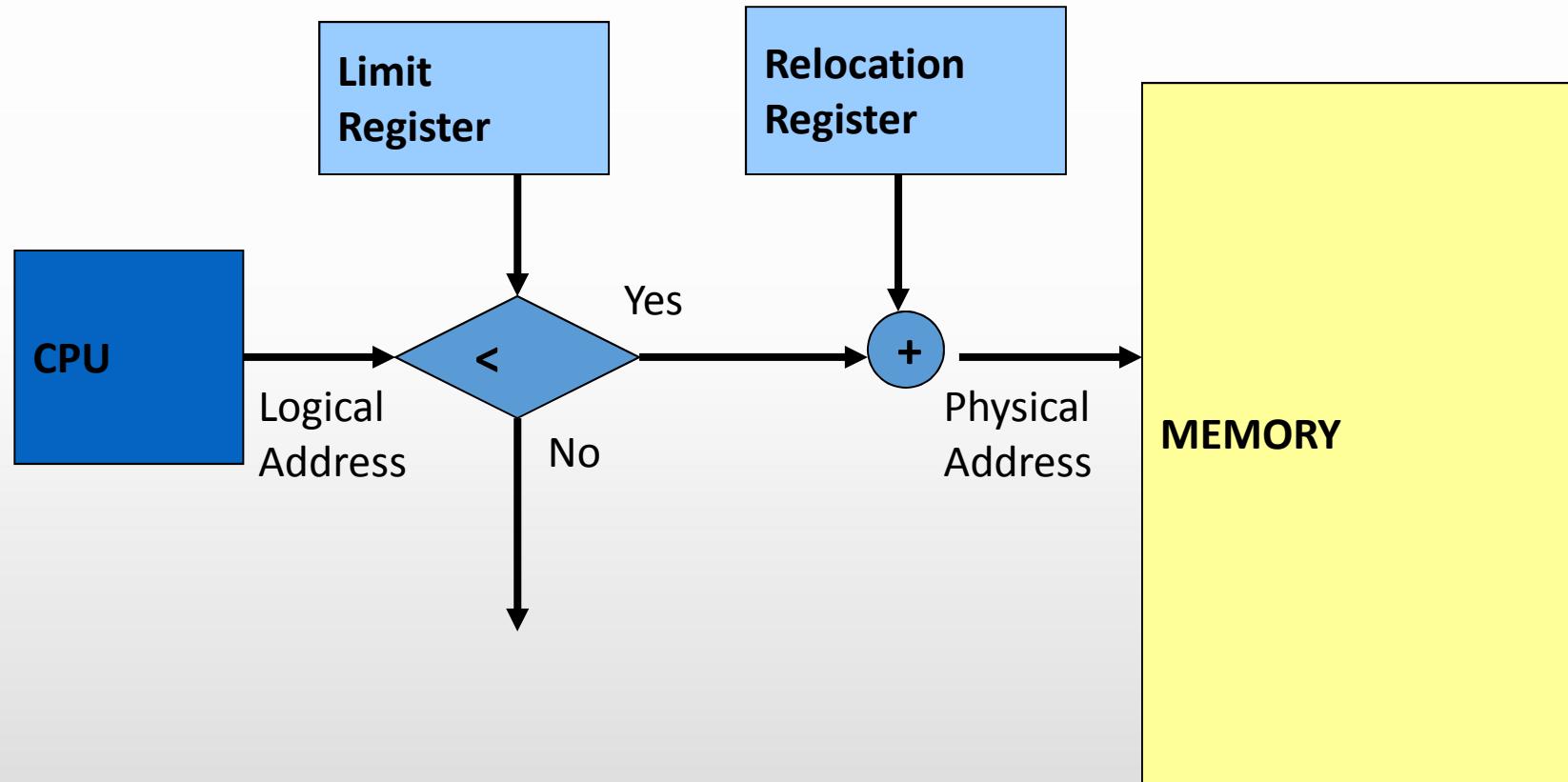
RESIDENT MONITOR:

- If monitor can change in size, start user at high end and move back, OR use fence as base register that requires address binding at execution time. Add base register to every generated user address.
- Isolate user from physical address space using logical address space.
- Concept of "mapping addresses"





Single Partition Allocation



Contiguous Allocation

All pages for a process are allocated together in one chunk.

JOB SCHEDULING

- Must take into account who wants to run, the memory needs, and partition availability. (This is a combination of short/medium term scheduling.)
- Sequence of events:
- In an empty memory slot, load a program
- THEN it can compete for CPU time.
- Upon job completion, the partition becomes available.
- Can determine memory size required (either user specified or "automatically").

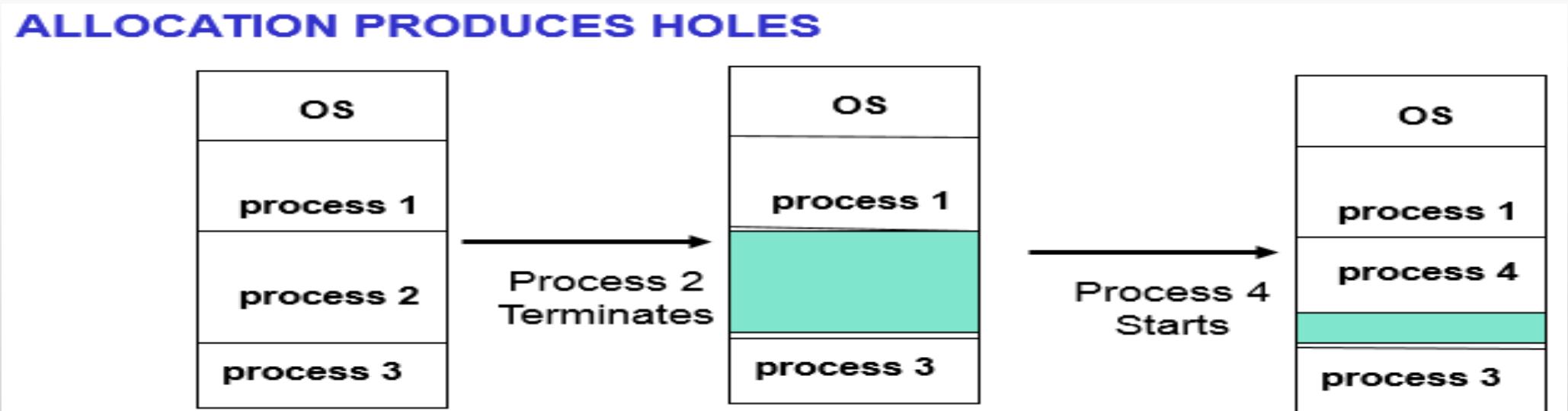




Contiguous Allocation

Dynamic Storage

- Variable sized holes in memory allocated on need
- Operating System keeps table of this memory – space allocated based on table
- Adjacent freed space merged to get largest holes –buddy system





Contiguous Allocation

HOW DO YOU ALLOCATE MEMORY TO NEW PROCESSES?

First fit - allocate the first hole that's big enough.

Best fit - allocate smallest hole that's big enough.

Worst fit - allocate largest hole.

(First fit is fastest, worst fit has lowest memory utilization.)

- Avoid small holes (**external fragmentation**). This occurs when there are many small pieces of free memory.
- What should be the minimum size allocated, allocated in what chunk size?
- Want to also avoid **internal fragmentation**. This is when memory is handed out in some fixed way (power of 2 for instance) and requesting program doesn't use it all.





Long Term Scheduling

If a job doesn't fit in memory, the scheduler can

wait for memory

skip to next job and see if it fits.

What are the pros and cons of each of these?

There's little or no internal fragmentation (the process uses the memory given to it - the size given to it will be a page.)

But there can be a great deal of external fragmentation. This is because the memory is constantly being handed cycled between the process and free.





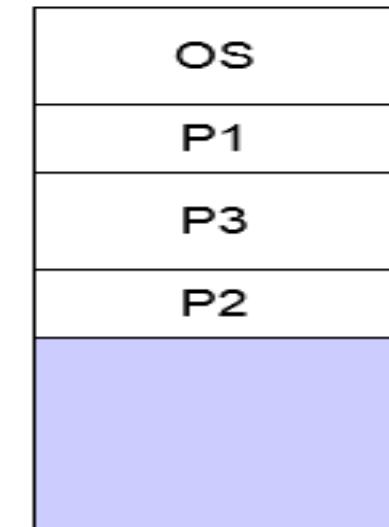
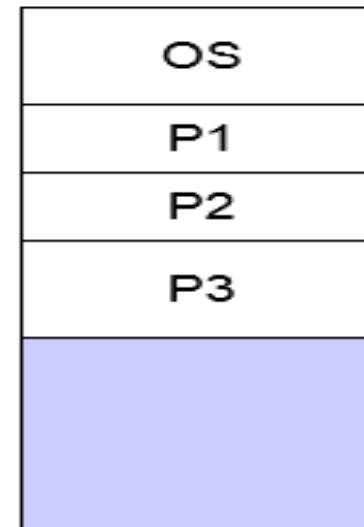
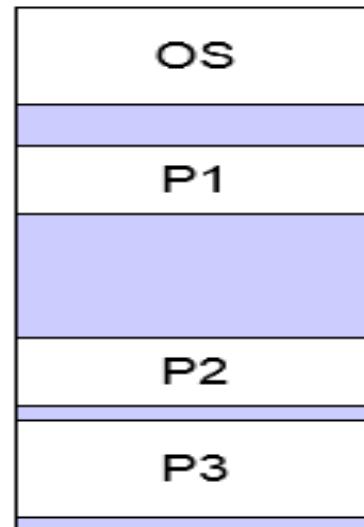
Compaction

Trying to move free memory to one large block

Only possible if programs linked with dynamic relocation (base and limit)

There are many ways to move programs in memory

Swapping: if using static relocation, code/data must return to same place. But if dynamic, can reenter at more advantageous memory



Paging

New Concept!!

Logical address space of a process can be noncontiguous; process is allocated physical memory whenever that memory is available and the program needs it.

Divide **physical** memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes).

Divide **logical** memory into blocks of same size called **pages**.

Keep track of all free frames.

To run a program of size n pages, need to find n free frames and load program.

Set up a page table to translate logical to physical addresses.

Internal fragmentation.





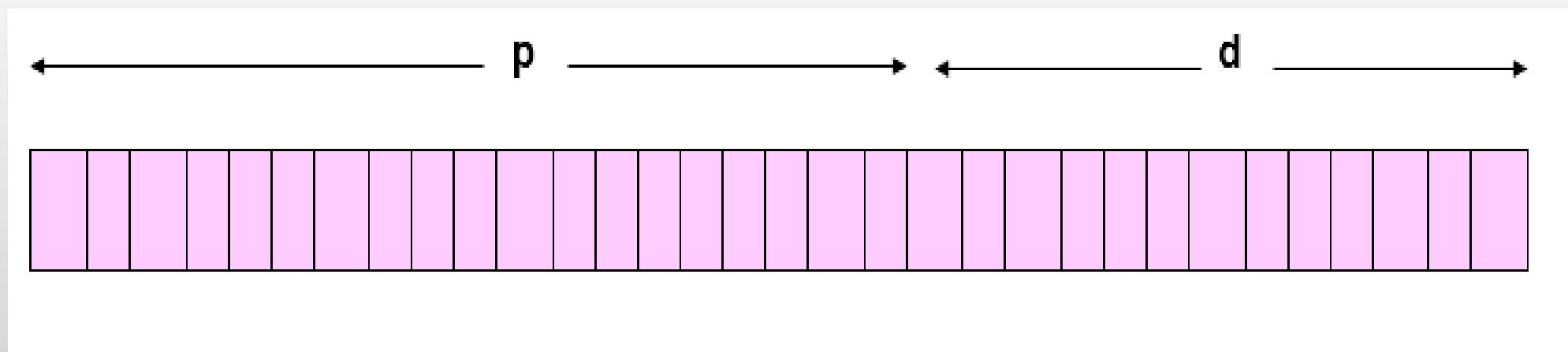
Paging

Address Translation Scheme

Address generated by the CPU is divided into:

Page number (p) – used as an index into a *page table* which contains base address of each page in physical memory.

Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit.





Paging

Permits a program's memory to be physically noncontiguous so it can be allocated from wherever available. This avoids fragmentation and compaction.

Frames = physical blocks
Pages = logical blocks

Size of frames/pages is defined by hardware
(power of 2 to ease calculations)

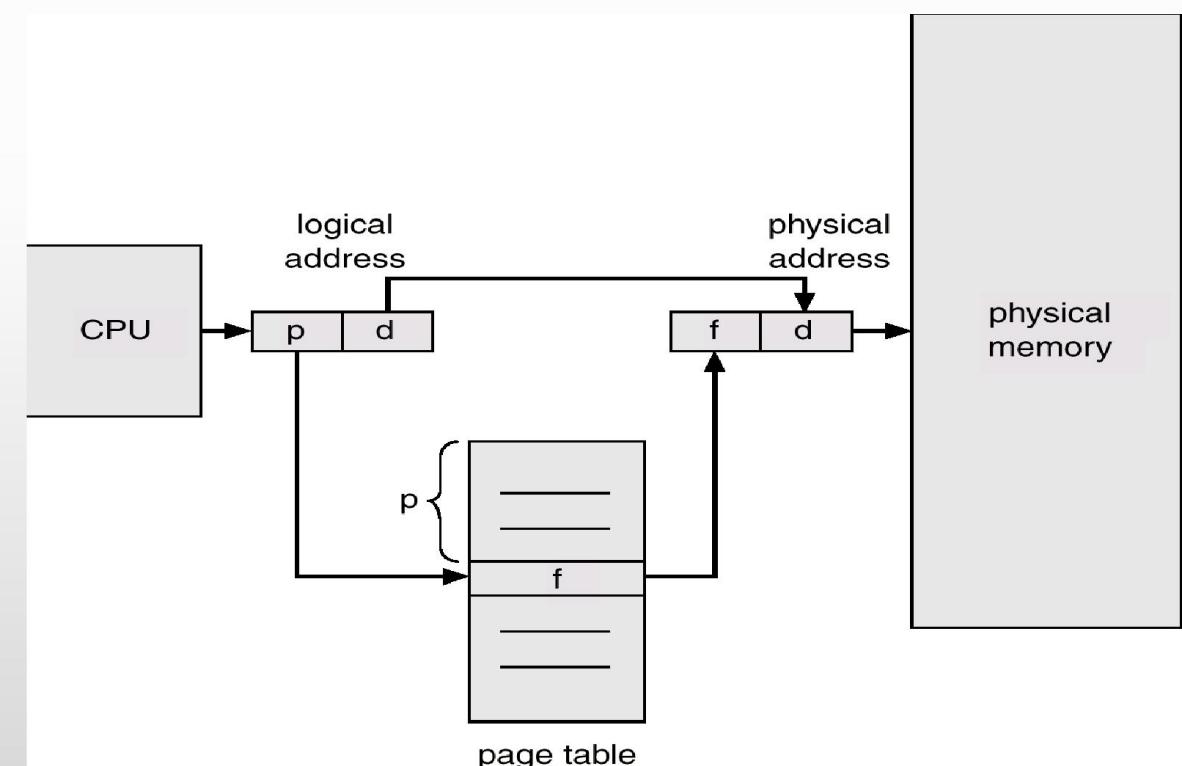
HARDWARE

An address is determined by:

page number (index into table) +
offset

---> mapping into --->

base address (from table) + offset.





Paging

Paging Example

32 byte memory with 4 byte pages

0 a	1 b	2 c	3 d
4 e	5 f	6 g	7 h
8 i	9 j	10 k	11 l
12 m	13 n	14 o	15 p

0	5
1	6
2	1
3	2

Page Table

Logical Memory

Physical Memory

0	
4	j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	



Paging

IMPLEMENTATION OF THE PAGE TABLE

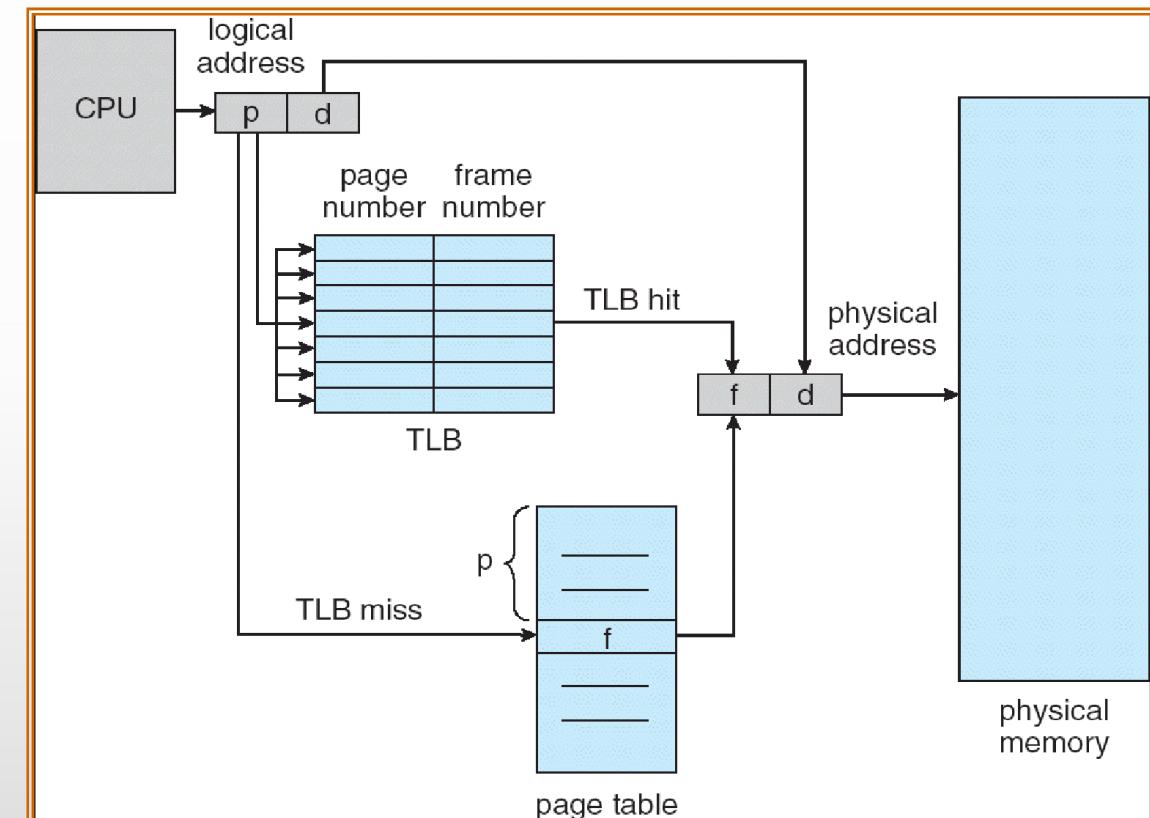
A 32 bit machine can address 4 gigabytes which is 4 million pages (at 1024 bytes/page). WHO says how big a page is, anyway?

Could use dedicated registers (OK only with small tables.)

Could use a register pointing to table in memory (slow access.)

Cache or associative memory
(TLB = Translation Lookaside Buffer):
simultaneous search is fast and uses only a few registers.

TLB = Translation Look-aside Buffer





Paging

IMPLEMENTATION OF THE PAGE TABLE

Issues include:

key and value

hit rate 90 - 98% with 100 registers

add entry if not found

$$\text{Effective access time} = \%{\text{fast}} * \text{time_fast} + \%{\text{slow}} * \text{time_slow}$$

Relevant times:

2 nanoseconds to search associative memory – the TLB.

20 nanoseconds to access processor cache and bring it into TLB for next time.

Calculate time of access:

hit = 1 search + 1 memory reference

miss = 1 search + 1 mem reference(of page table) + 1 mem reference.





Paging

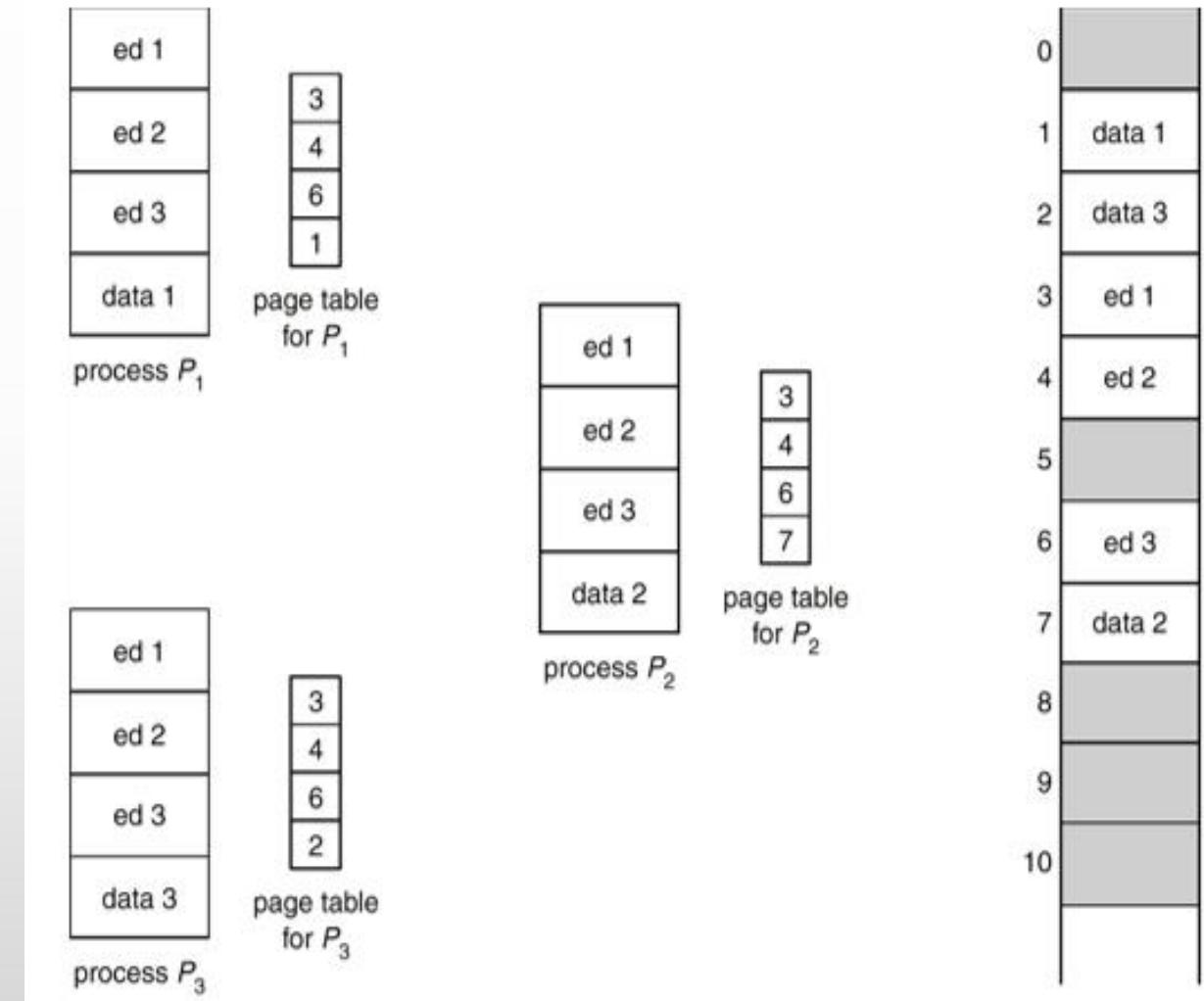
Shared Pages

Data occupying one physical page, but pointed to by multiple logical pages

Useful for common code – must be write protected

(no writeable data mixed with code)

Extremely useful for read/write communication between processes





Paging

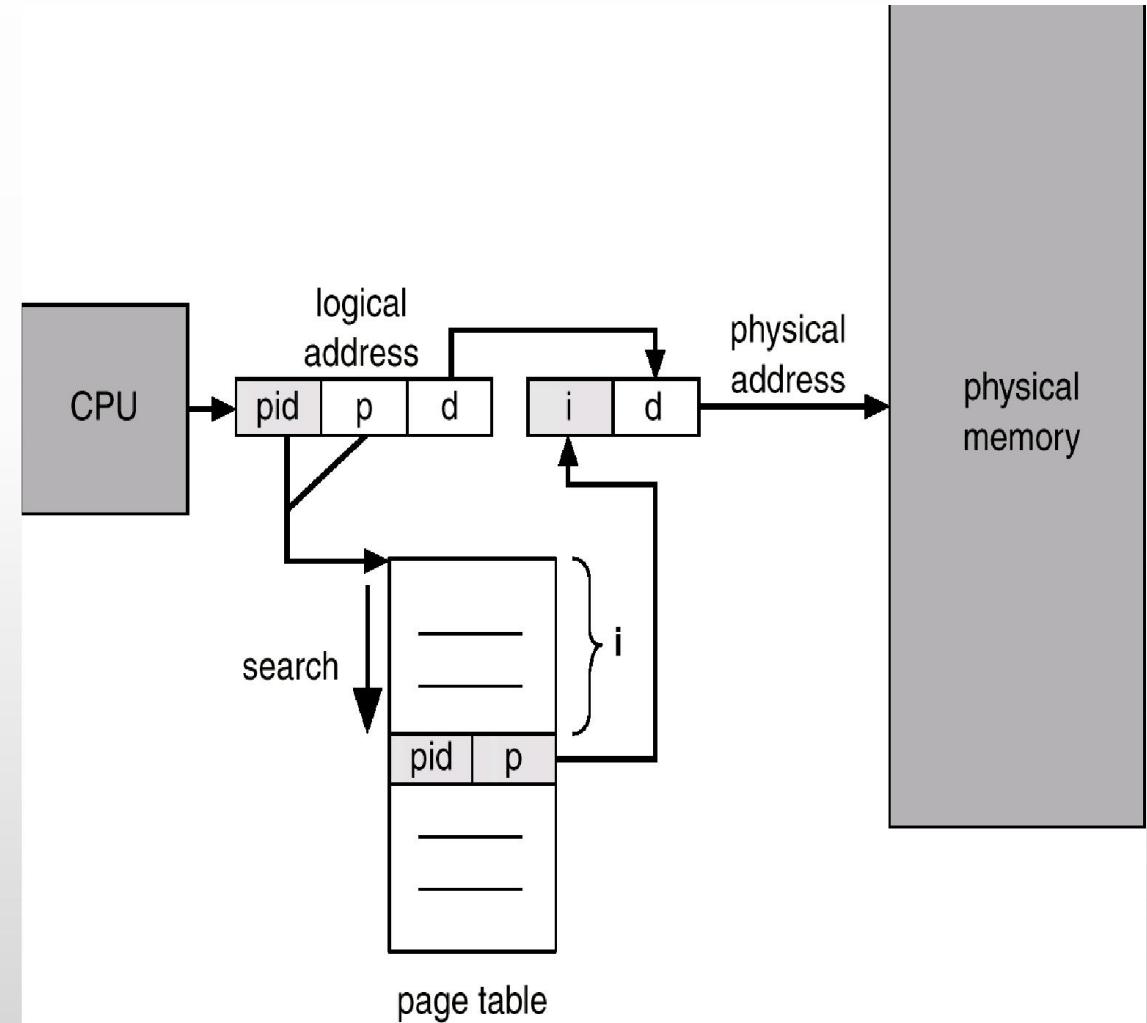
INVERTED PAGE TABLE:

One entry for each real page of memory.

Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.

Essential when you need to do work on the page and must find out what process owns it.

Use hash table to limit the search to one - or at most a few - page table entries.





Paging

PROTECTION:

Bits associated with page tables.

Can have read, write, execute, valid bits.

Valid bit says page isn't in address space.

Write to a write-protected page causes a fault. Touching an invalid page causes a fault.





Paging

ADDRESS MAPPING:

Allows physical memory larger than logical memory.

Useful on 32 bit machines with more than 32-bit addressable words of memory.

The operating system keeps a frame containing descriptions of physical pages; if allocated, then to which logical page in which process.

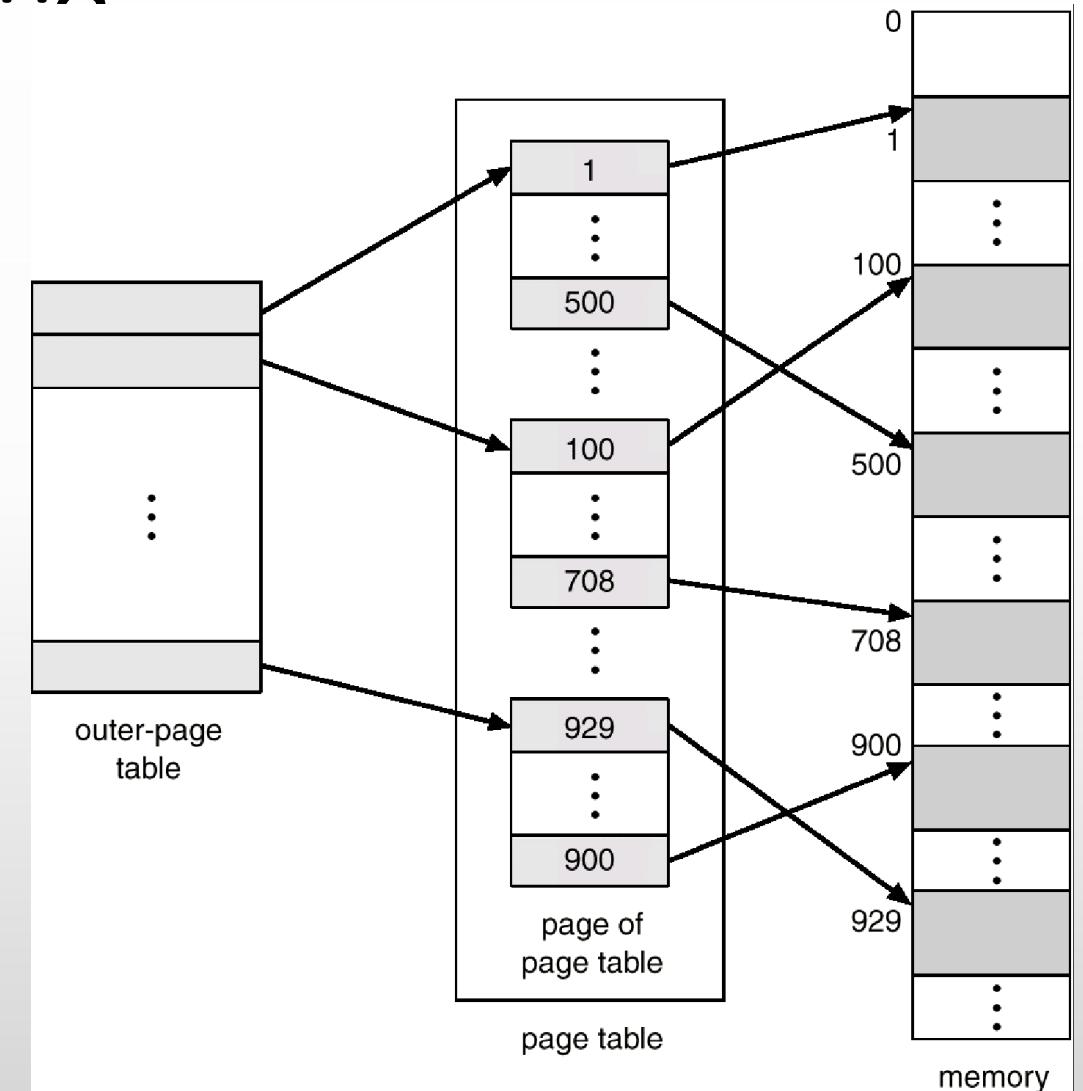
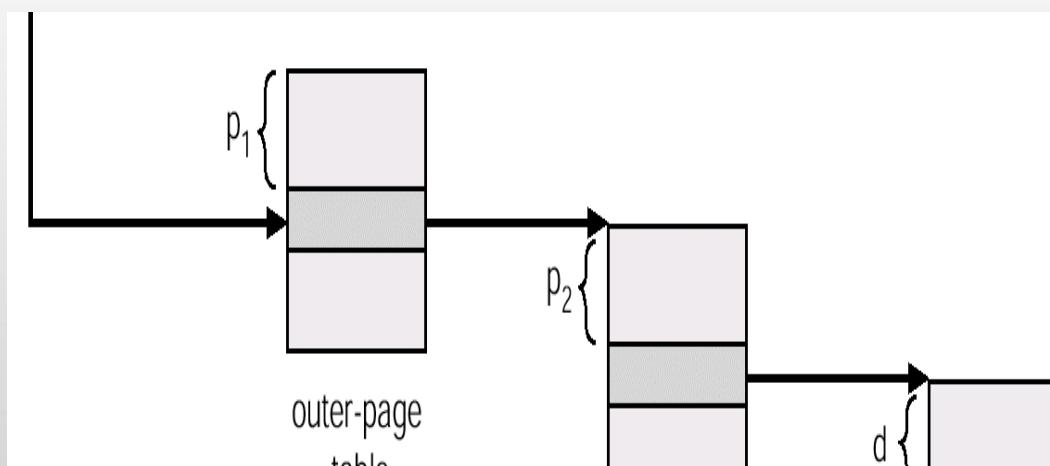




Paging

MULTILEVEL PAGE TABLE

A means of using page tables for large address spaces.





Segmentation

USER'S VIEW OF MEMORY

A programmer views a process consisting of unordered segments with various purposes. This view is more useful than thinking of a linear array of words. We really don't care at what address a segment is located.

Typical segments include

- global variables
- procedure call stack
- code for each function
- local variables for each
- large data structures

Logical address = segment name (number) + offset

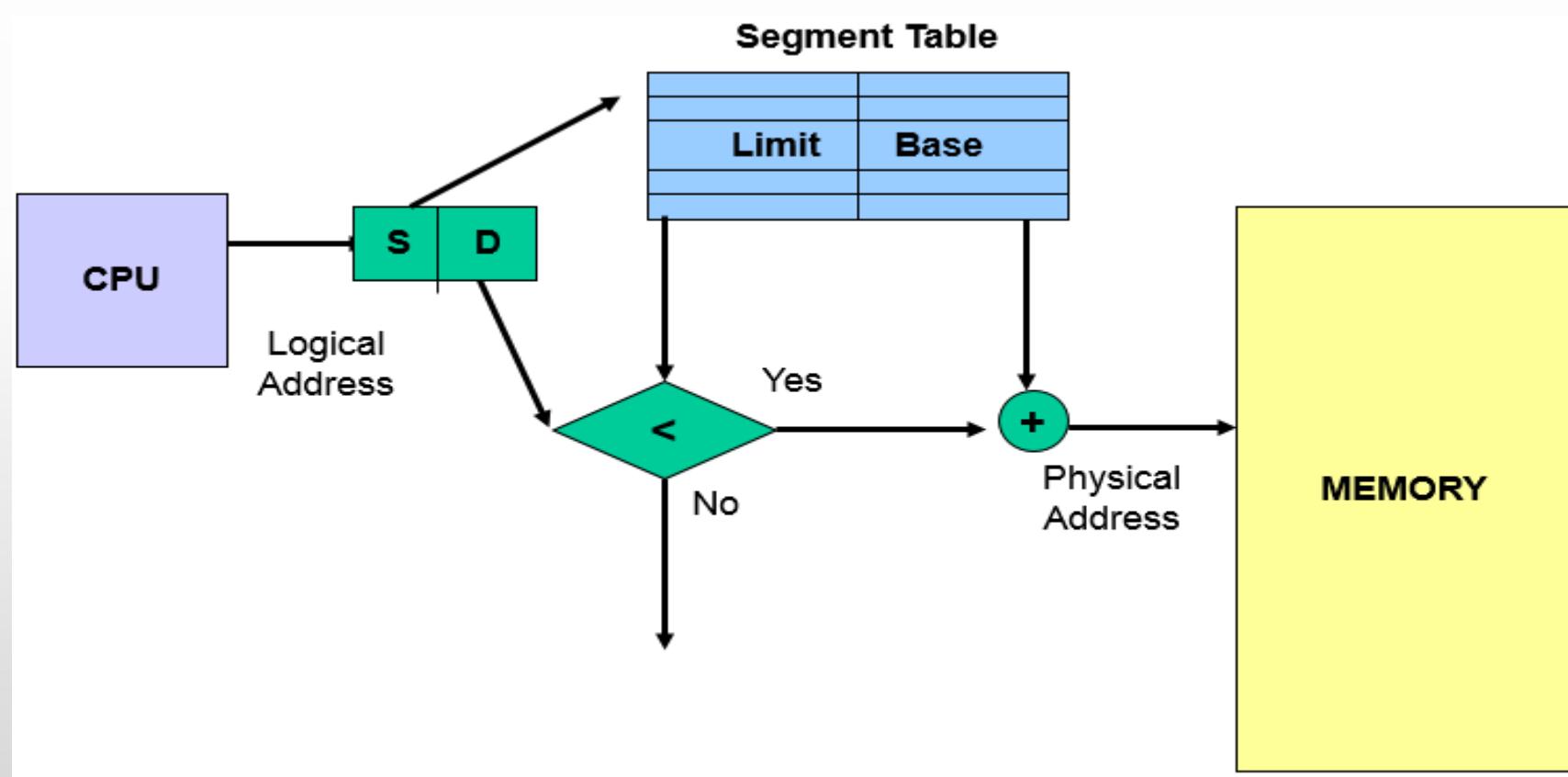
Memory is addressed by both segment and offset.





Segmentation

Hardware – must map a segment/offset into one-dimensional address

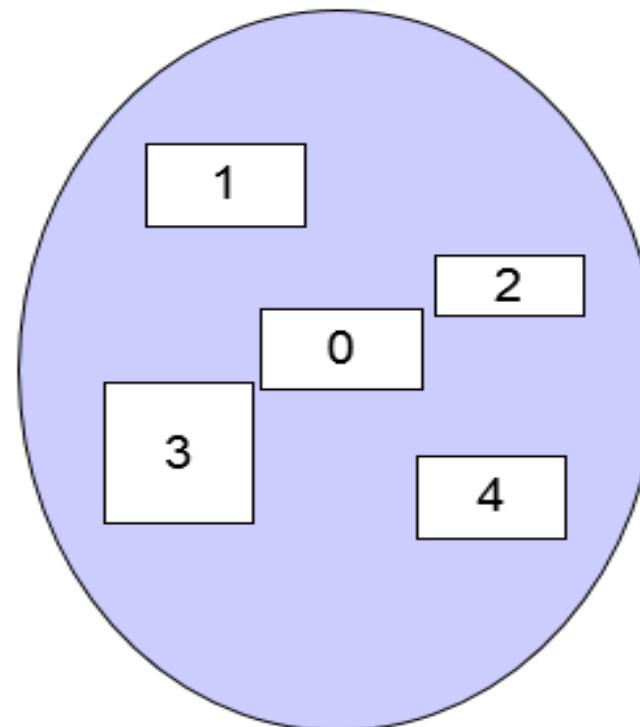




Segmentation

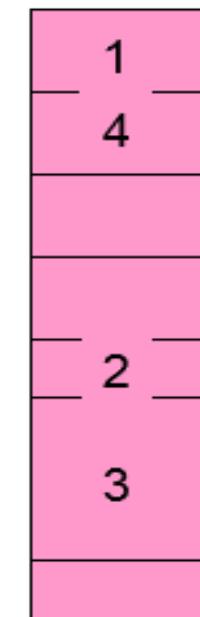
Hardware

Base/ limit pairs in a segment table



Logical Address Space

	Limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700



Physical Memory



Segmentation

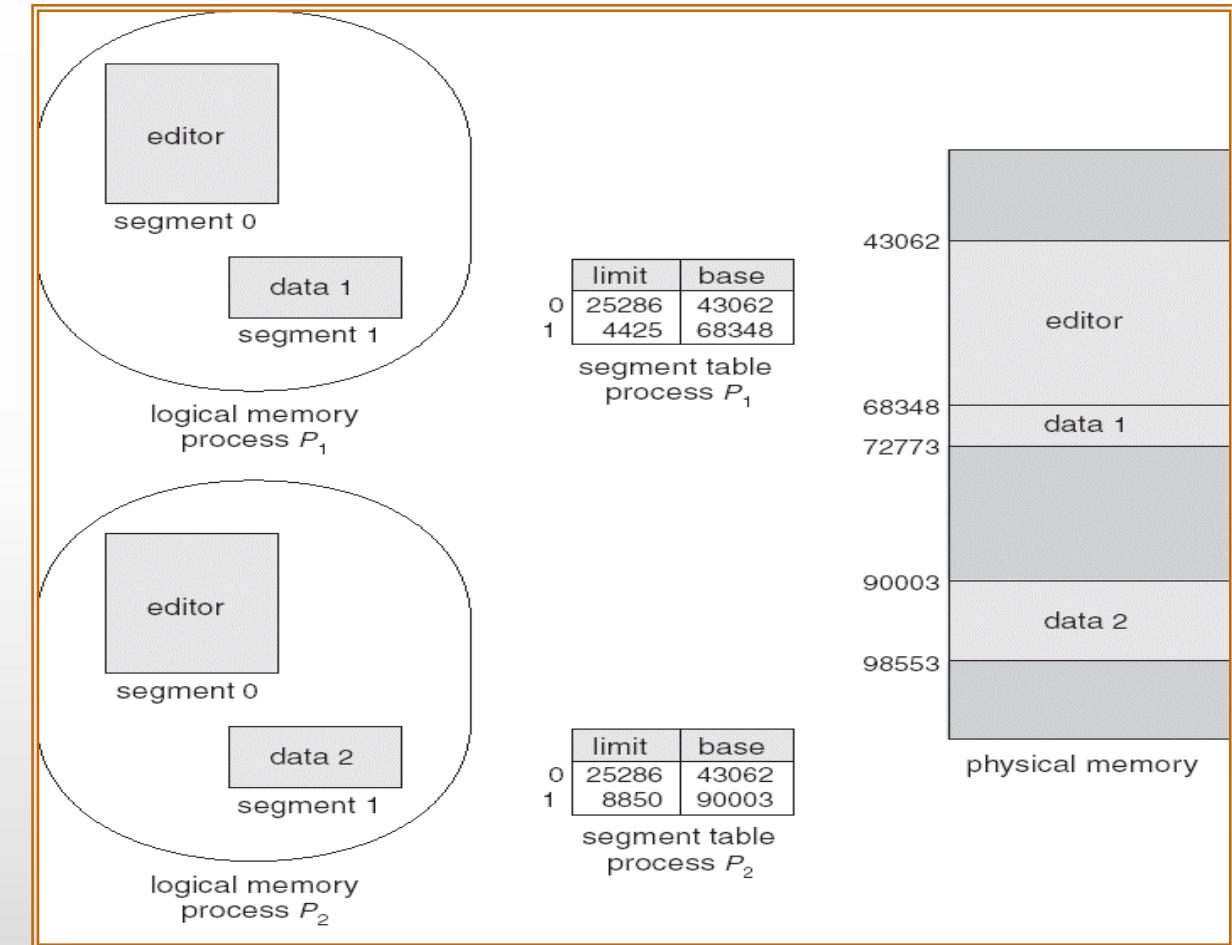
PROTECTION AND SHARING

Addresses are associated with a logical unit (like data, code, etc.) so protection is easy.

Can do bounds checking on arrays

Sharing specified at a logical level, a segment has an attribute called "shareable".

Can share some code but not all - for instance a common library of subroutines.



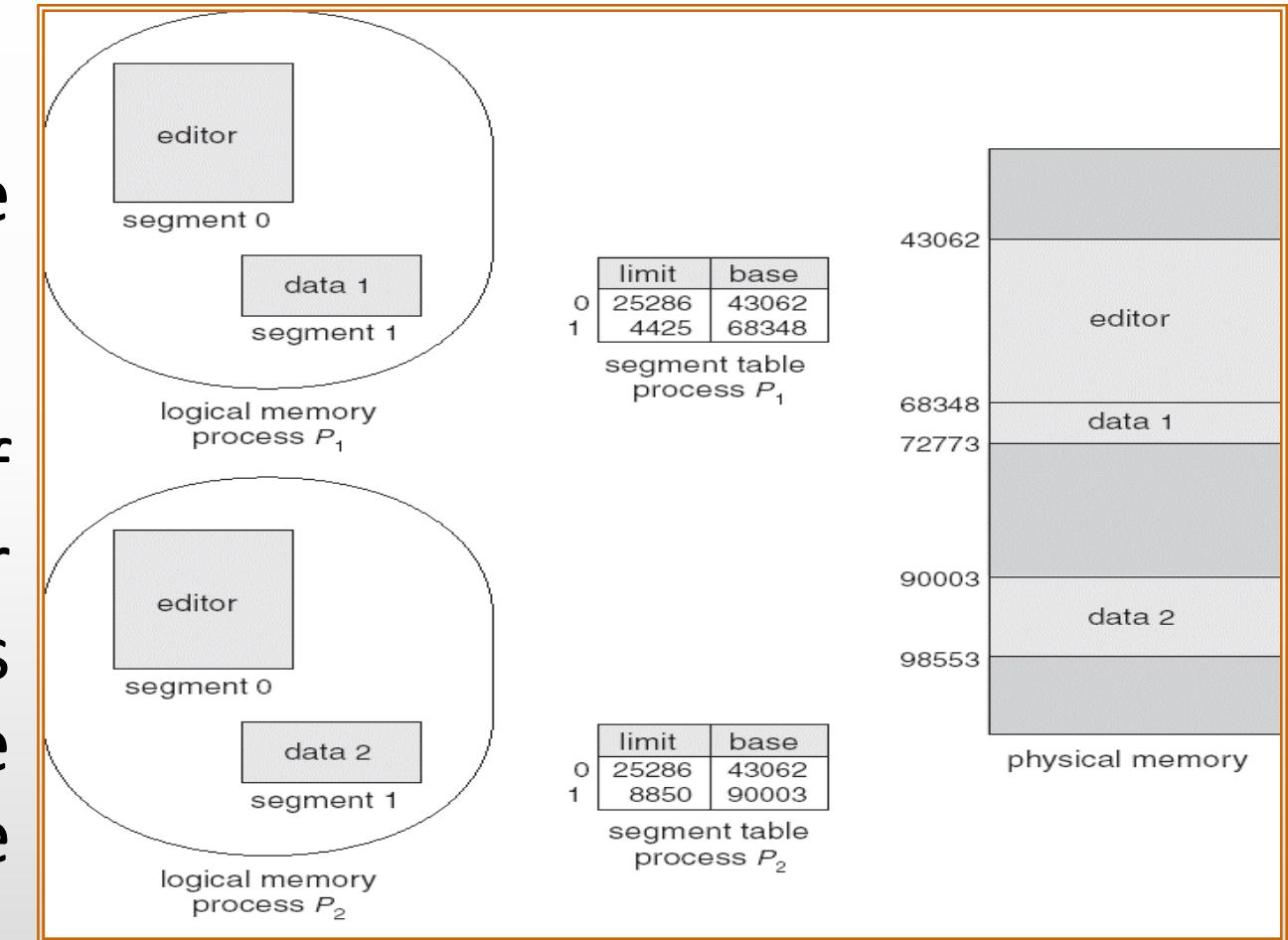


FRAGMENTATION

Use variable allocation since segment lengths vary.

Again have issue of fragmentation; Smaller segments means less fragmentation. Can use compaction since segments are relocatable.

Segmentation





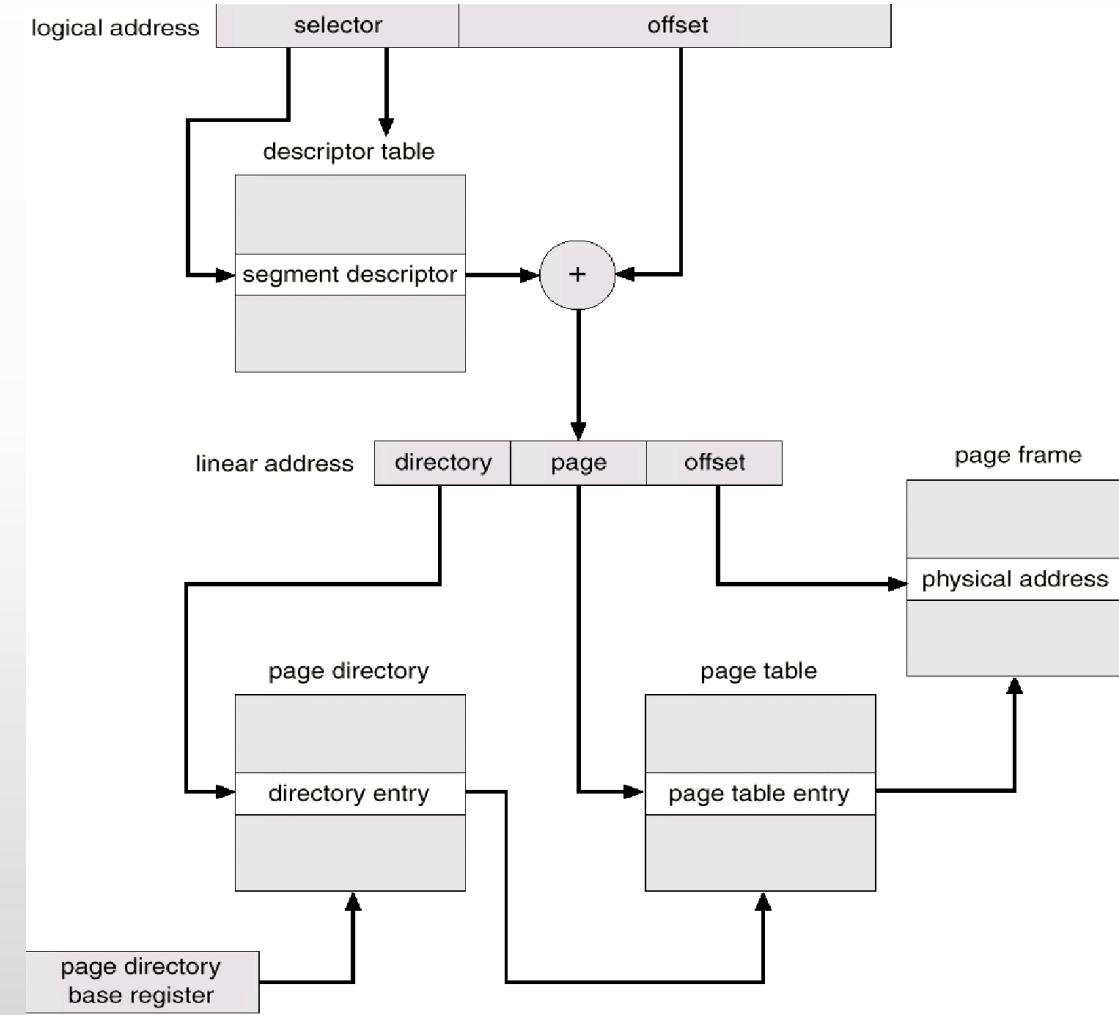
Segmentation

PAGED SEGMENTATION

Combination of paging and segmentation.

address = frame at (page table base for segment + offset into page table) + offset into memory

Look at example of Intel architecture.





Intel Memory Management

Memory Management Architecture used by Intel Pentium processors.

Intel document found at:

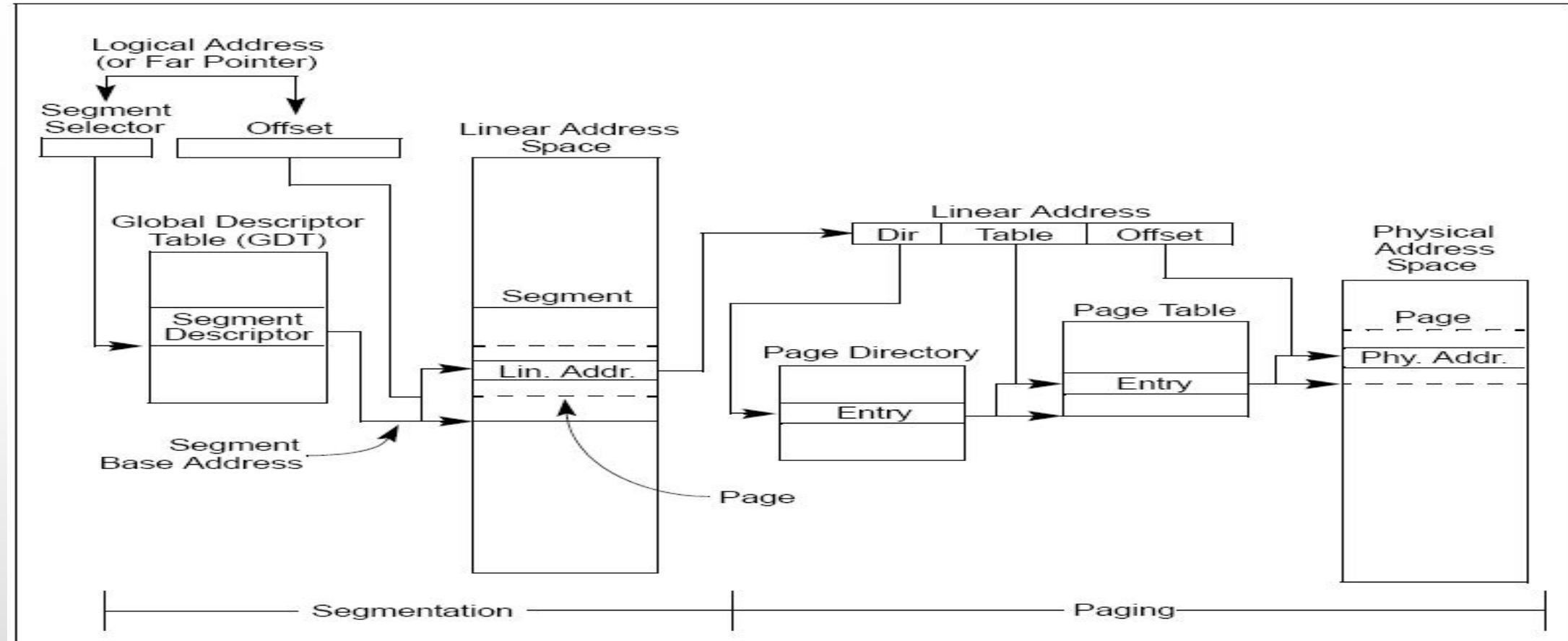
<http://www.intel.com/design/processor/manuals/253668.pdf>

Intel explains this document as a description of the hardware interface required by an Operating System in order to implement a Memory Management.





Intel Memory Management



This is an overview of the hardware pieces provided by Intel. It's what we have to work with if we're designing an O.S.





Intel Memory Management

The memory management facilities of the **IA-32** architecture are divided into two parts:

Segmentation

Segmentation provides a mechanism of isolating individual code, data, and stack modules so that multiple programs (or tasks) can run on the same processor without interfering with one another.

When operating in protected mode, some form of segmentation must be used.





Intel Memory Management

Paging.

Paging provides a mechanism for implementing a conventional demand-paged, virtual-memory system where sections of a program's execution environment are mapped into physical memory as needed. Paging can also be used to provide isolation between multiple tasks.

These two mechanisms (segmentation and paging) can be configured to support simple single program (or single-task) systems, multitasking systems, or multiple-processor systems that used shared memory.



Intel Memory Management

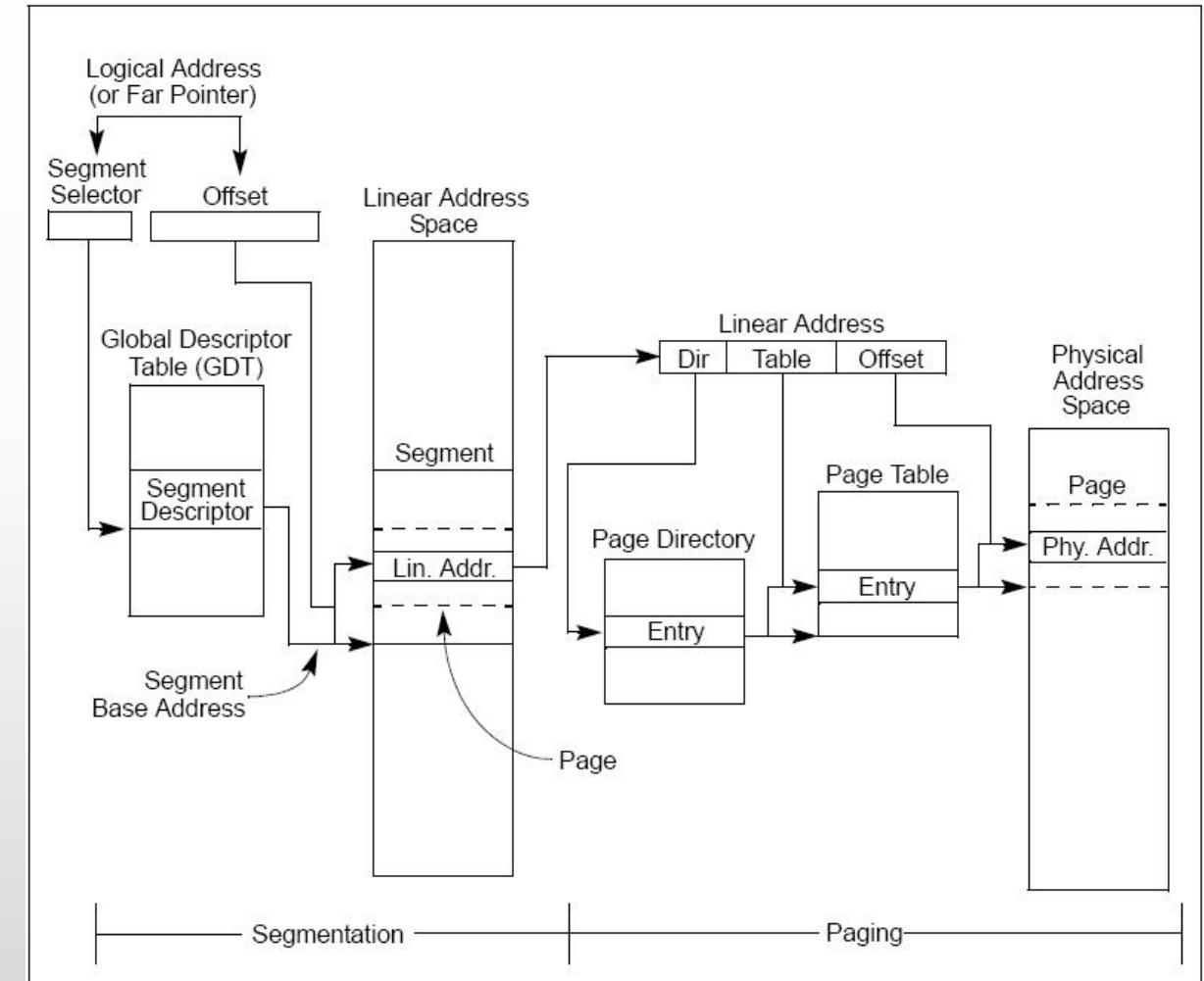
Segmentation gives a mechanism for dividing the processor's addressable memory space (**called the linear address space**) into smaller protected address spaces called **segments**.

Segments are used to hold code, data, and stack for a program and to hold system data structures (such as a TSS or LDT).

Each program running on a processor, is assigned its own set of segments.

The processor enforces the boundaries between segments and insures that one program doesn't interfere with the execution of another .

The segmentation mechanism allows typing of segments to restrict operations that can be performed.





Intel Memory Management

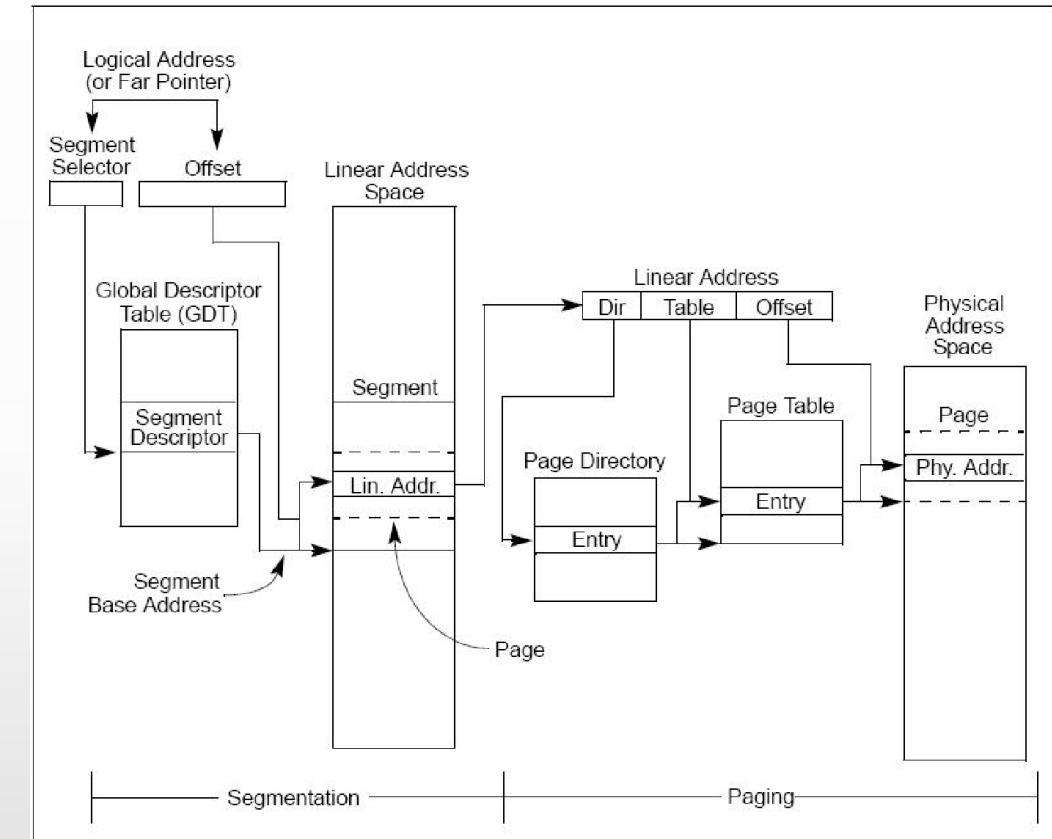
All the segments in a system are contained in the processor's **linear address space**.

To locate a byte in a particular segment, **a logical address** (also called a far pointer) must be provided.

A logical address has :

1. The segment selector – a unique identifier for a segment - provides an offset into a descriptor table (such as the global descriptor table, GDT) to a data structure called a segment descriptor.

This segment descriptor specifies the size of the segment, the access rights and privilege level for the segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment).



2. The offset part of the logical address -added to the base address for the segment to locate a byte within the segment.
The base address plus the offset thus forms a **linear address** in the processor's linear address space.





Intel Memory Management

Basic Flat Model

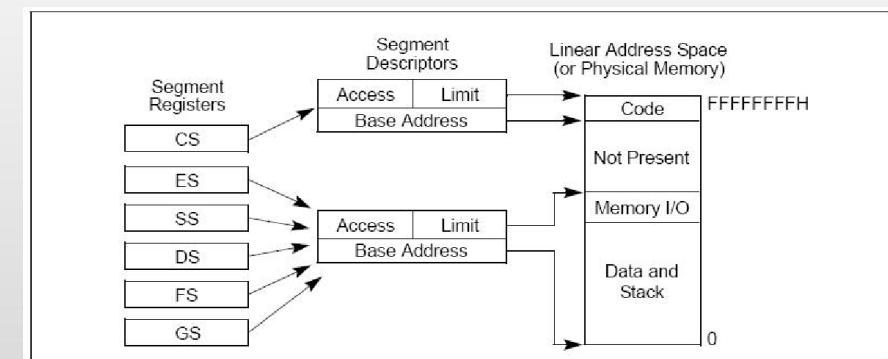
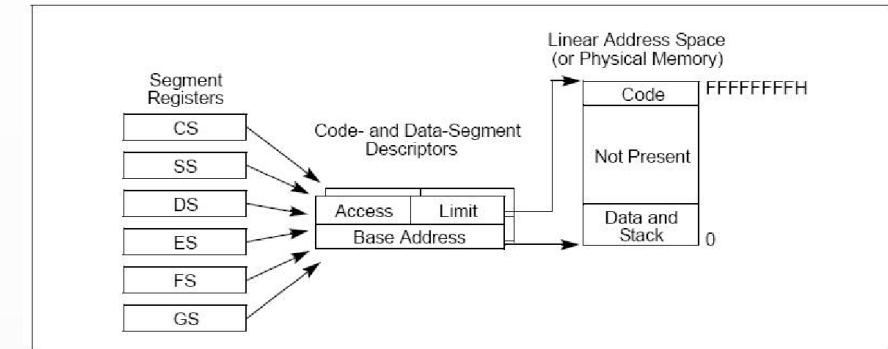
The simplest memory model for a system is the basic “flat model,”

the operating system and application programs have access to a continuous, unsegmented address space.

To implement a basic flat memory model with the IA-32 architecture, at least two segment descriptors must be created:

one for referencing a code segment and one for referencing a data segment (see Figure 3-2).

both segments, however, are mapped to the entire linear address space: that is, both segment descriptors have the same base address value of 0 and the same segment limit of 4 GBytes.





Intel Memory Management

3.2.2 Protected Flat Model

The protected flat model is similar to the basic flat model, except the segment limits are set to include only the range of addresses for which physical memory actually exists (see Figure 3-3).

A protection exception is generated on any attempt to access nonexistent memory. This model provides a minimum level of hardware protection against some kinds of program bugs.

More complexity can be added to this protected flat model to provide more protection.

Example: For the paging mechanism to provide isolation between user and supervisor code and data, four segments need to be defined:

code and data segments at privilege level 3 for the user,
and code and data segments at privilege level 0 for the supervisor.

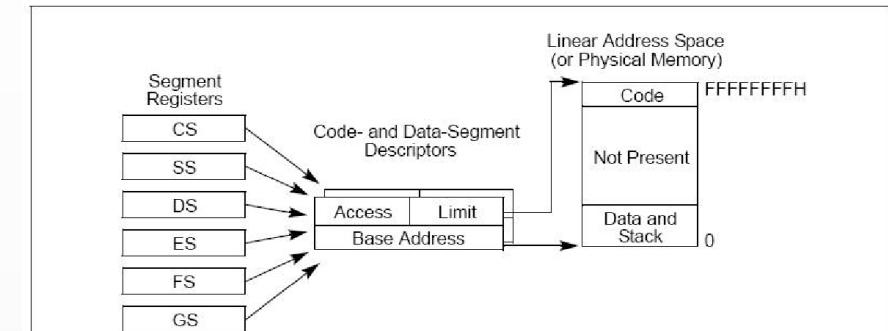


Figure 3-2. Flat Model

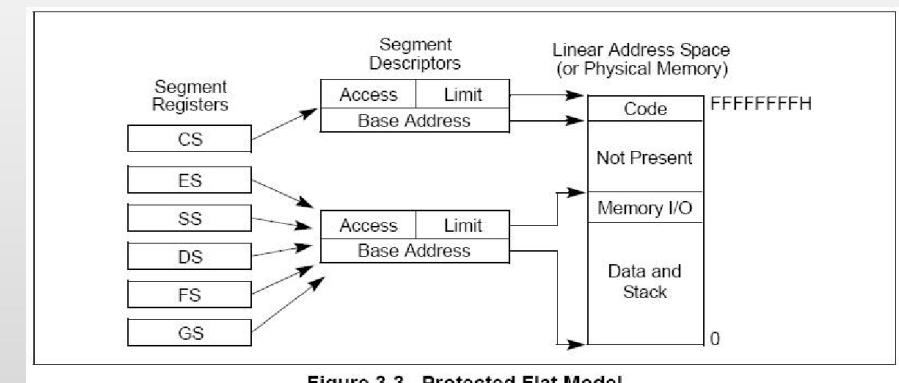


Figure 3-3. Protected Flat Model





Intel Memory Management

3.2.3 Multi-Segment Model

A multi-segment model (shown here) uses the full capabilities of segmentation to provide hardware enforced protection of code, data structures, and programs and tasks.

each program (or task) has its own table of segment descriptors and its own segments.

segments can be completely private to their programs or shared among programs.

Access to segments and to program environments is controlled by hardware.

Access checks can be used to protect not only against referencing an address outside the limit of a segment, but also against performing disallowed operations in certain segments.

- The access rights information created for segments can also be used to set up protection rings or levels.
- Protection levels can be used to protect operating system procedures from unauthorized access by application programs.

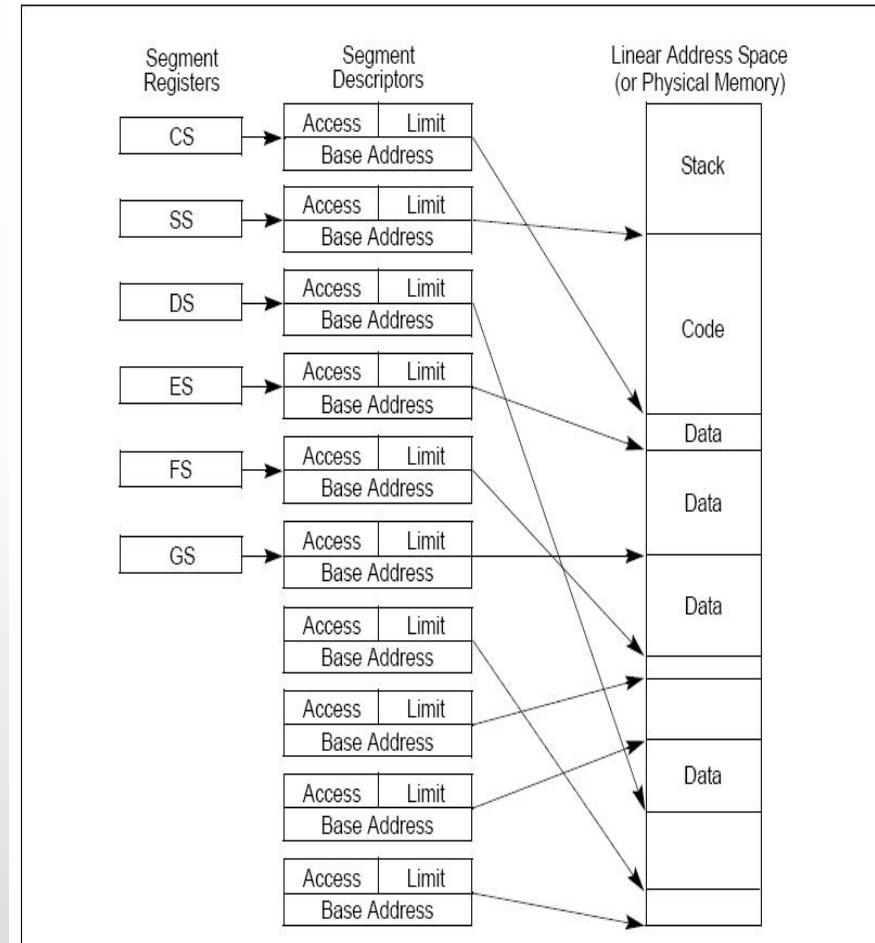


Figure 3-4. Multi-Segment Model



Physical Address Space

3.3 PHYSICAL ADDRESS SPACE

In protected mode, the IA-32 architecture provides a normal physical address space of 4 Gbytes (2^{32} bytes).

This is the address space that the processor can address on its address bus. This address space is flat (unsegmented), with addresses ranging continuously from 0 to FFFF,FFFFH. This physical address space can be mapped to read-write memory, read-only memory, and memory mapped I/O. The memory mapping facilities described in this chapter can be used to divide this physical memory up into segments and/or pages.

The IA-32 architecture also supports an extension of the physical address space to 2^{36} bytes (64 GBytes); with a maximum physical address of F,FFFF,FFFFH. This extension is invoked

- Using the physical address extension (PAE) flag, located in bit 5 of control register CR4.
-- Talked about later.





Logical and Linear Address

The processor uses two stages of address translation to arrive at a physical address: logical-address (via segments) translation and linear address space (via paging) translation.

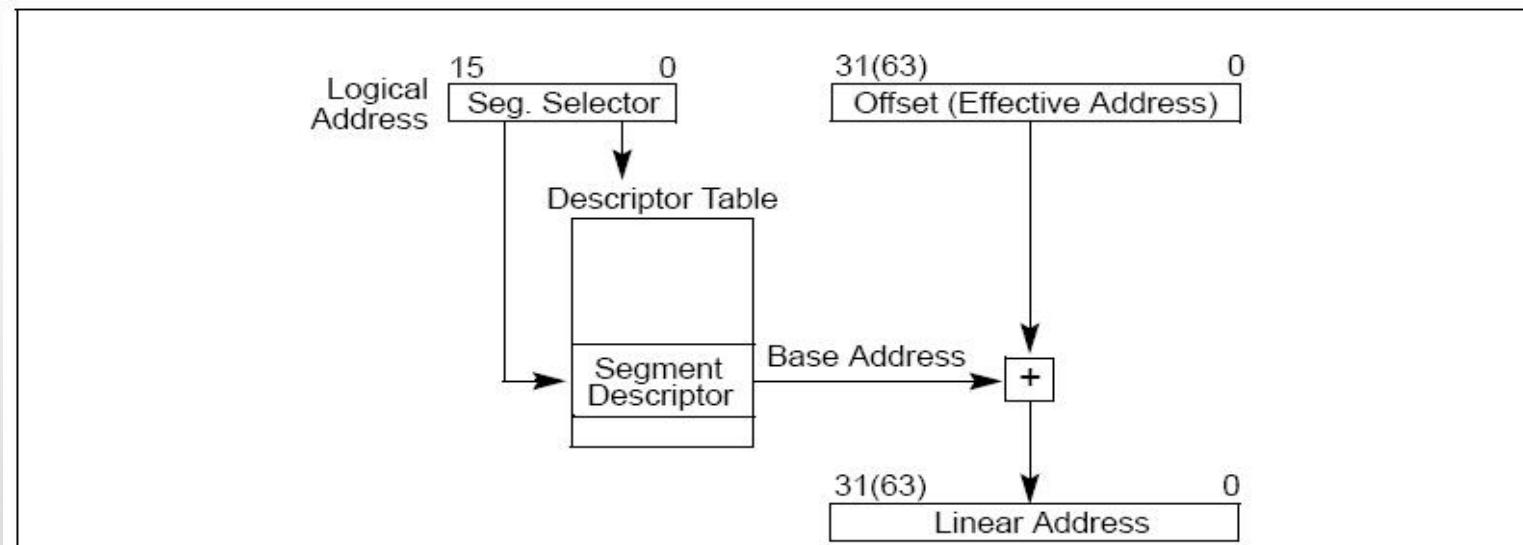


Figure 3-5. Logical Address to Linear Address Translation





Logical and Linear Address

Every byte in the processor's address space is accessed with a logical address. A **logical** address consists of a 16-bit segment selector and a 32-bit offset (see Figure 3-5).

.A **linear** address is a 32-bit address in the processor's linear address space. The linear address space is a flat (unsegmented), 2^{32} -byte address space, with addresses ranging from 0 to FFFF,FFFFH.

The linear address space contains all the segments and system tables defined for a system.

To translate a logical address into a linear address, the processor does the following:

1. Uses the offset in the segment selector to find the descriptor for the segment in the GDT or LDT and reads it into the processor, or uses the appropriate segment register.
2. Examines the segment descriptor to check the access rights and range of the segment – makes sure the segment is accessible and has legal offset.
3. Adds the base address of the segment to the offset to form a linear address.

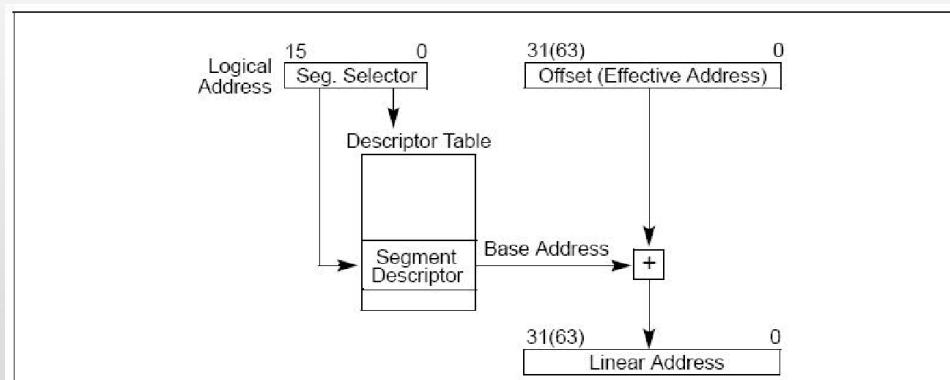


Figure 3-5. Logical Address to Linear Address Translation

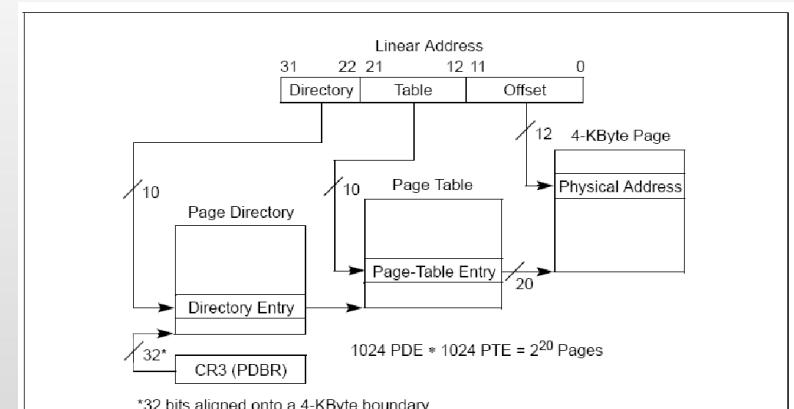


Figure 3-12. Linear Address Translation (4-KByte Pages)



Segment Selectors

3.4.2 Segment Selectors

A segment selector is a 16-bit identifier for a segment (see Figure 3-6). It does not point directly to the segment, but instead points to the segment descriptor that defines the segment. A segment selector contains the following items:

Index — Selects one of 8192 descriptors in the GDT or LDT.

TI (table indicator) flag — Specifies the descriptor table to use: GDT or LDT

Requested Privilege Level (RPL) — Specifies the privilege level of the selector. The privilege level can range from 0 to 3, with 0 being the most privileged level.





Segment Registers

3.4.3 Segment Registers

To reduce address translation time and coding complexity, the processor provides registers for holding up to 6 segment selectors (see Figure 3-7).

Each of these segment registers support a specific kind of memory reference (code, stack, or data).

At least the code-segment, data-segment, and stack-segment registers must be loaded for a program to run..

The processor provides three additional data-segment registers (ES, FS, and GS), which can be used to make other data segments available to the currently executing program (or task).

To access a segment, a program must get to it via a segment register.

Although a system can define thousands of segments, only 6 can be available for immediate use.

There are instructions available so the OS can set up segment registers.

Note how the address translation actually goes through the segment register rather than through the Descriptor Table.

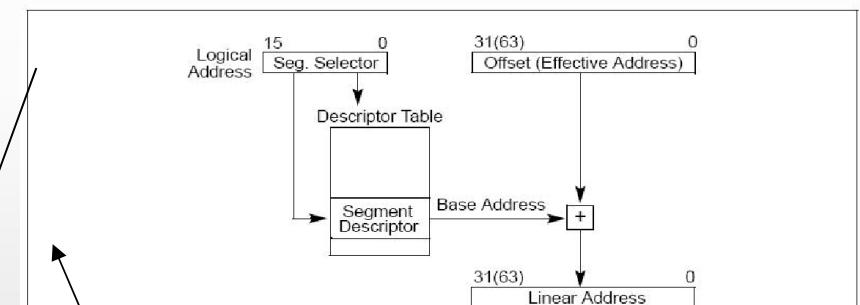
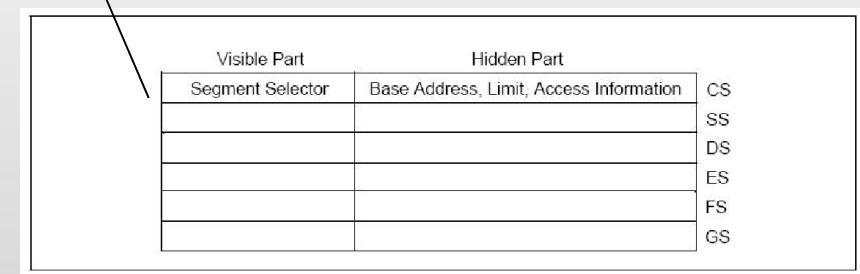


Figure 3-5. Logical Address to Linear Address Translation



Visible Part	Hidden Part
Segment Selector	Base Address, Limit, Access Information
	CS
	SS
	DS
	ES
	FS
	GS

Figure 3-7. Segment Registers



Segment Registers

Every segment register has a “visible” part and a “hidden” part.

When a segment selector is loaded, the processor also loads the hidden part of the segment register with the base address, segment limit, and access control information from the descriptor pointed to by the segment selector.

This allows the processor to translate addresses without taking extra bus cycles to read the base address and limit from the segment descriptor.

In systems in which multiple processors have access to the same descriptor tables, it is the responsibility of software to reload the segment registers when the descriptor tables are modified.

If this is not done, an old segment descriptor cached in a segment register might be used after its memory-resident version has been modified.

Two kinds of instructions are provided for loading the segment registers:

1. Direct load instructions such as the MOV, LES, LGS, and LFS instructions explicitly reference the segment registers.
2. Implied load instructions such as the far pointer versions of the CALL, JMP, and RET instructions, the SYSENTER and SYSEXIT instructions, and the IRET, INT n , INTO and INT3 instructions. These instructions change the contents of the CS register as an incidental part of their operation.

Visible Part	Hidden Part
Segment Selector	Base Address, Limit, Access Information
	CS
	SS
	DS
	ES
	FS
	GS

Figure 3-7. Segment Registers



System Descriptor Types

3.5 SYSTEM DESCRIPTOR TYPES

When the S (descriptor type) flag in a segment descriptor is clear, the descriptor type is a system descriptor. The processor recognizes the following types of system descriptors:

- Local descriptor-table (LDT) segment descriptor.
- Task-state segment (TSS) descriptor.
- Call-gate descriptor.
- Interrupt-gate descriptor.
- Trap-gate descriptor.
- Task-gate descriptor.

These descriptor types fall into two categories: system-segment descriptors and gate descriptors.

System-segment descriptors point to system segments (LDT and TSS segments). Gate descriptors are in themselves “gates,” which hold pointers to procedure entry points in code segments (call, interrupt, and trap gates) or which hold segment selectors for TSS’s (task gates).

Table 3-2 shows the encoding of the type field for system-segment descriptors and gate descriptors.



Segment Descriptor Tables

3.5.1 Segment Descriptor Tables

A segment descriptor table is an array of segment descriptors (see Figure 3-10). A descriptor table is variable in length and can contain up to 8192 (213) 8-byte descriptors.

There are two kinds of descriptor tables:

- The global descriptor table (GDT)
- The local descriptor tables (LDT)

Each system must have one GDT defined, which may be used for all programs and tasks in the system.

Optionally, one or more LDTs can be defined. For example, an LDT might be defined for each separate task being run.

The GDT is not a segment itself; instead, it is a data structure in linear address space. The base linear address and limit of the GDT must be loaded into the GDTR register.

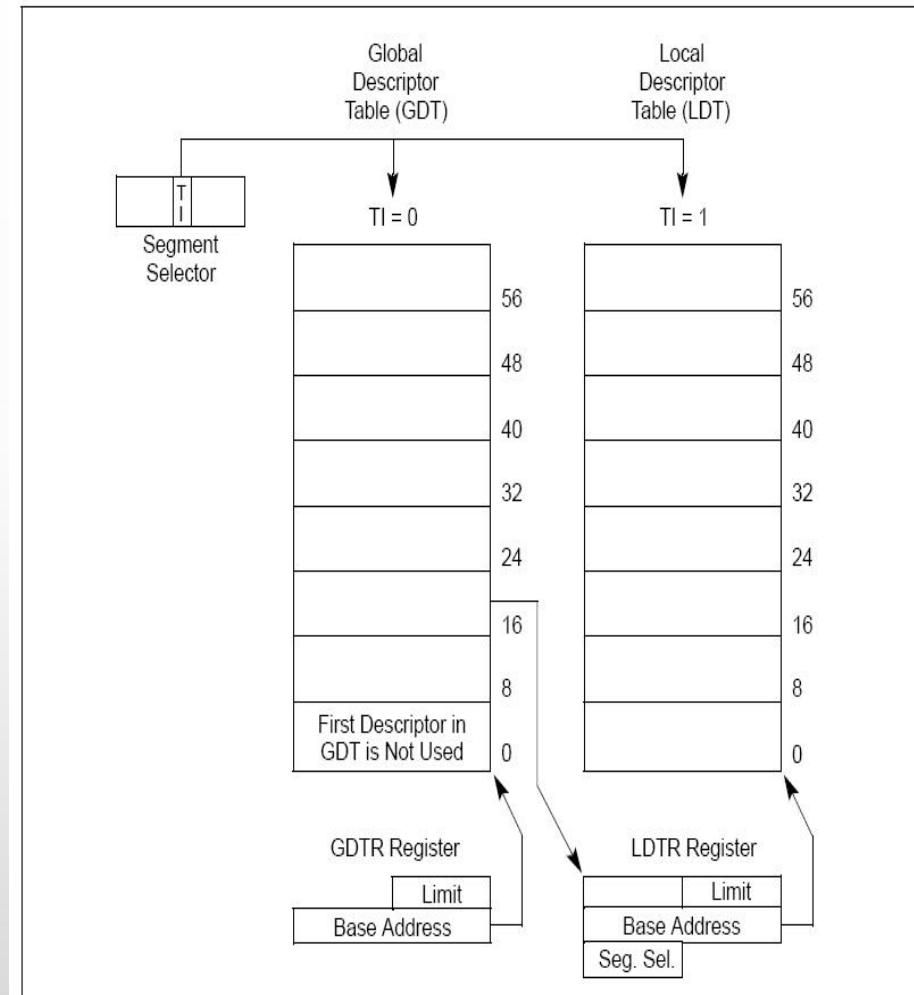


Figure 3-10. Global and Local Descriptor Tables



Segment Descriptor Tables

3.5.1 Segment Descriptor Tables

The LDT is located in a system segment of the LDT type.

The GDT must contain a segment descriptor for the LDT segment. If the system supports multiple LDTs, each must have a separate segment selector and segment descriptor in the GDT.

An LDT is accessed with its segment selector. To eliminate address translations when accessing the LDT, the segment selector, base linear address, limit, and access rights of the LDT are stored in the LDTR register.

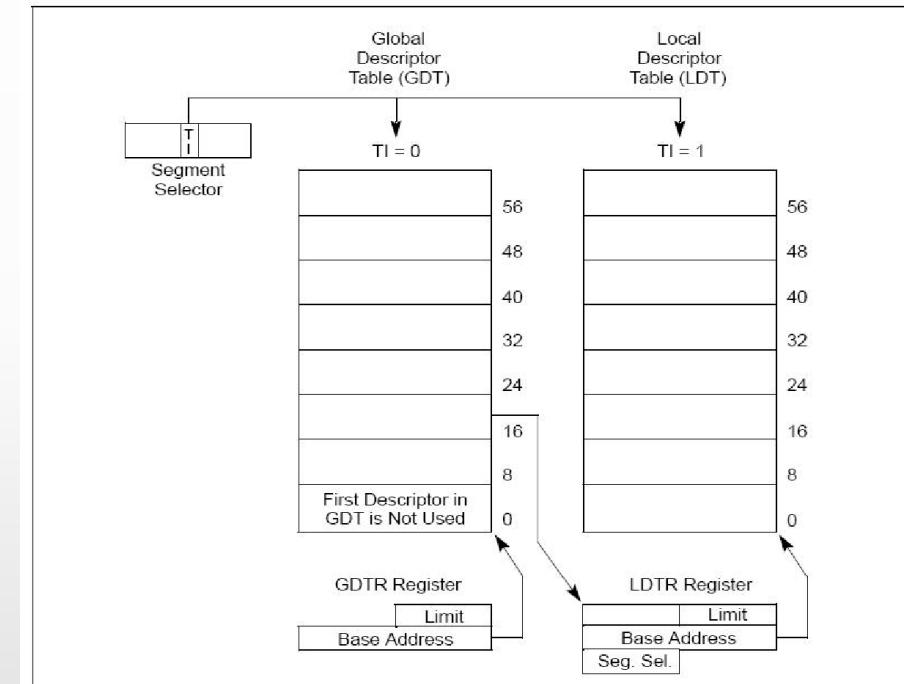


Figure 3-10. Global and Local Descriptor Tables





Thank You Very Much

Benjamin Kommey

bkommey.coe@knust.edu.gh

nii_kommey@msn.com

050 770 32 86

Skype_id: calculus.affairs





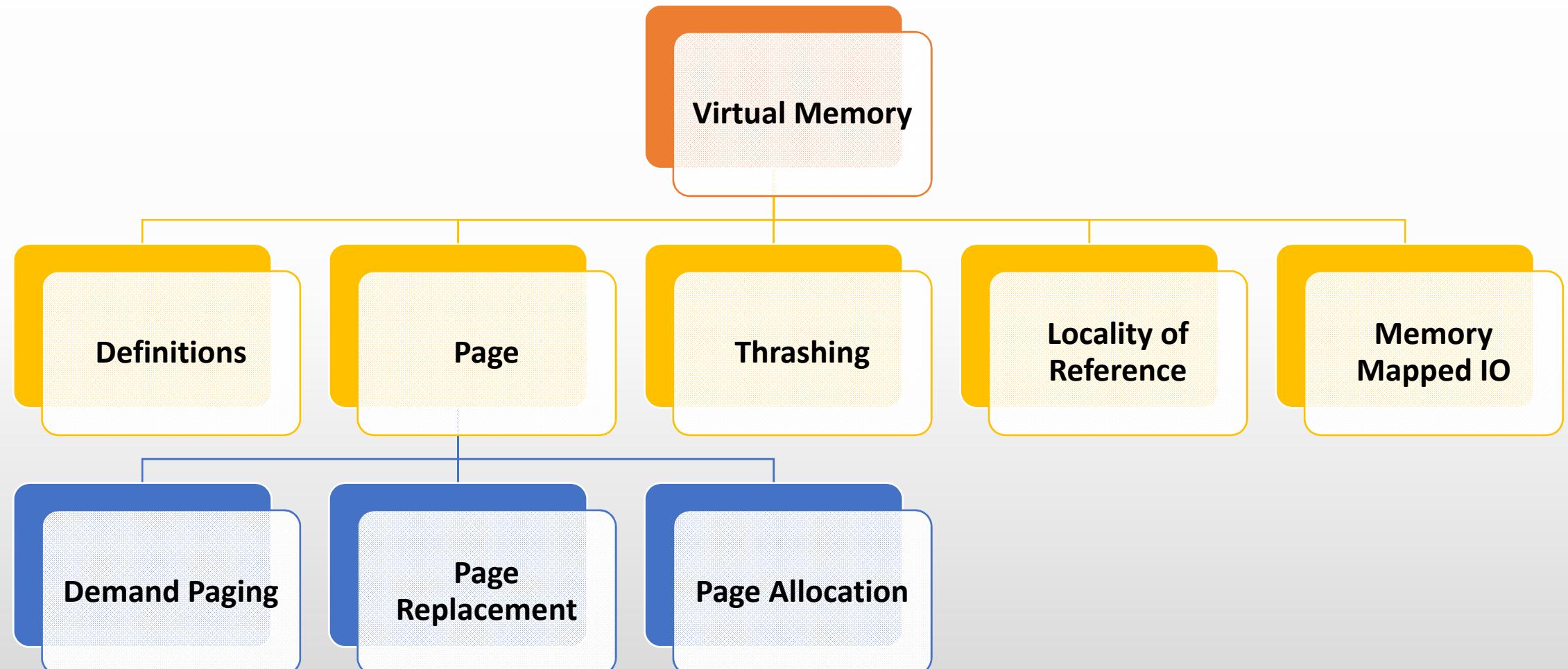
“Technology is just a tool. In terms of getting the kids working together and motivating them, the teacher is the most important.”

Bill Gates





Overview





Virtual Memory

WHY VIRTUAL MEMORY?

- We've previously required the entire logical space of the process to be in memory before the process could run.
We will now look at alternatives to this.
- Most code/data isn't needed at any instant, or even within a finite time - we can bring it in only as needed.

VIRTUES

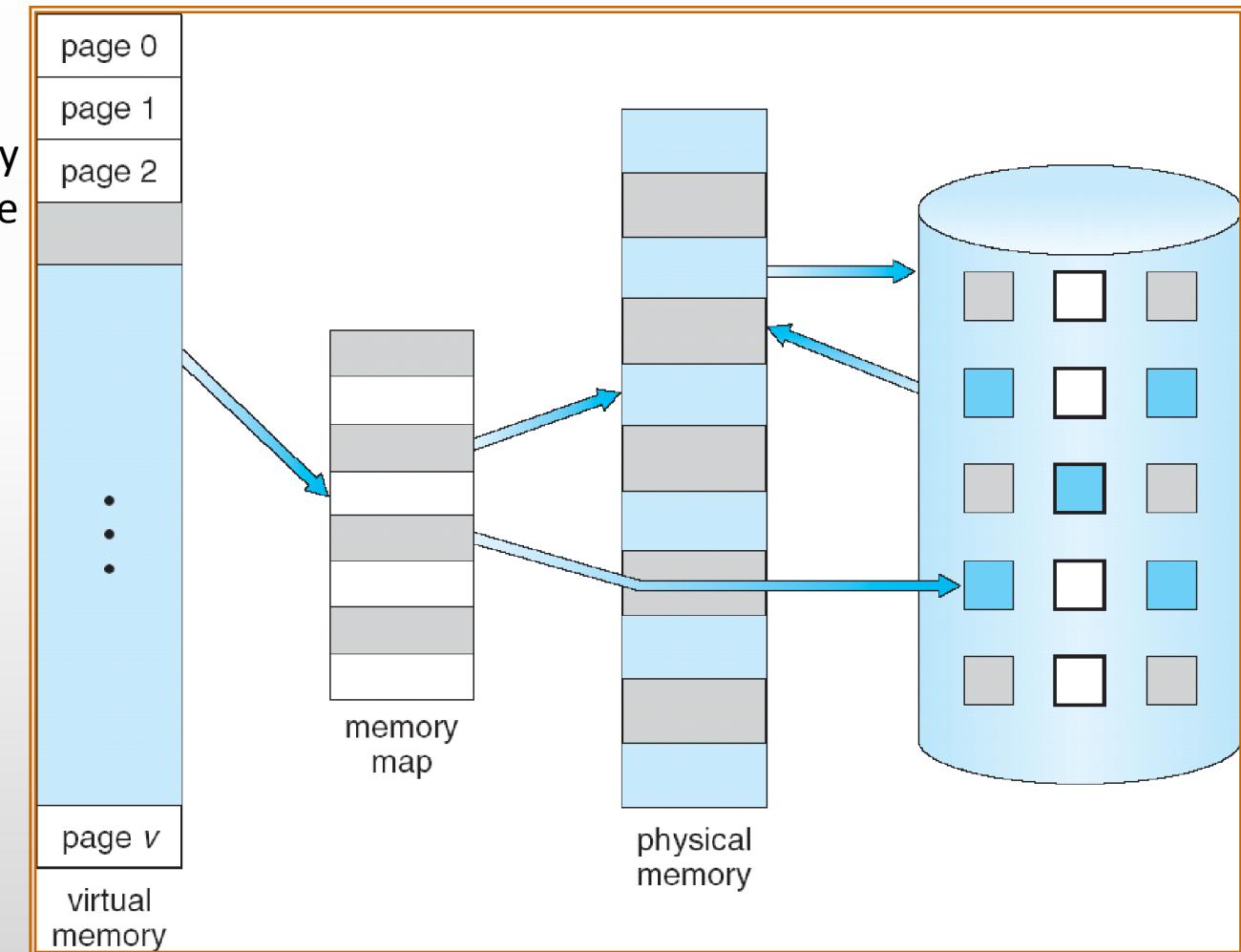
- Gives a higher level of multiprogramming
- The program size isn't constrained (thus the term 'virtual memory'). Virtual memory allows very large logical address spaces.
- Swap sizes smaller.





Virtual memory

The conceptual separation of user logical memory from physical memory. Thus we can have large virtual memory on a small physical memory.





Definitions

Demand paging

When a page is touched, bring it from secondary to main memory.

Overlays

Laying of code data on the same logical addresses - this is the reuse of logical memory. Useful when the program is in phases or when logical address space is small.

Dynamic loading

A routine is loaded only when it's called.

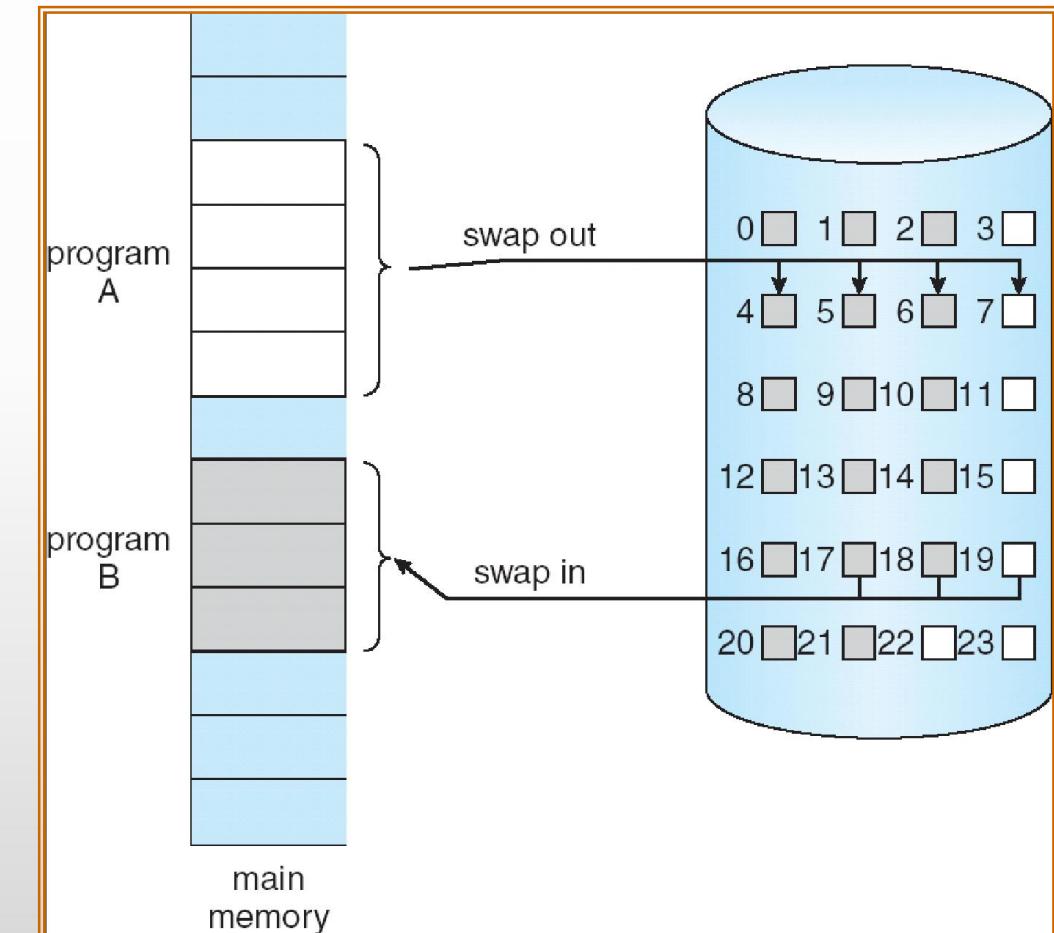




Demand Paging

When a page is referenced, either as code execution or data access, and that page isn't in memory, then get the page from disk and re-execute the statement.

Here's migration between memory and disk.





Demand Paging

One instruction may require several pages. For example, a block move of data.

May page fault part way through an operation - may have to undo what was done. Example: an instruction crosses a page boundary.

Time to service page faults demands that they happen only infrequently.

Note here that the page table requires a "resident" bit showing that page is/isn't in memory. (Book uses "valid" bit to indicate residency. An "invalid" page is that way because a legal page isn't resident or because the address is illegal.

It makes more sense to have two bits - one indicating that the page is legal (valid) and a second to show that the page is in memory.

Frame #	valid-invalid bit
	1
	1
	1
	1
	0
⋮	
	0

page table

Frame #	Resid ent	valid- invalid bit
	1	1
	0	0





Demand Paging

STEPS IN HANDLING A PAGE FAULT

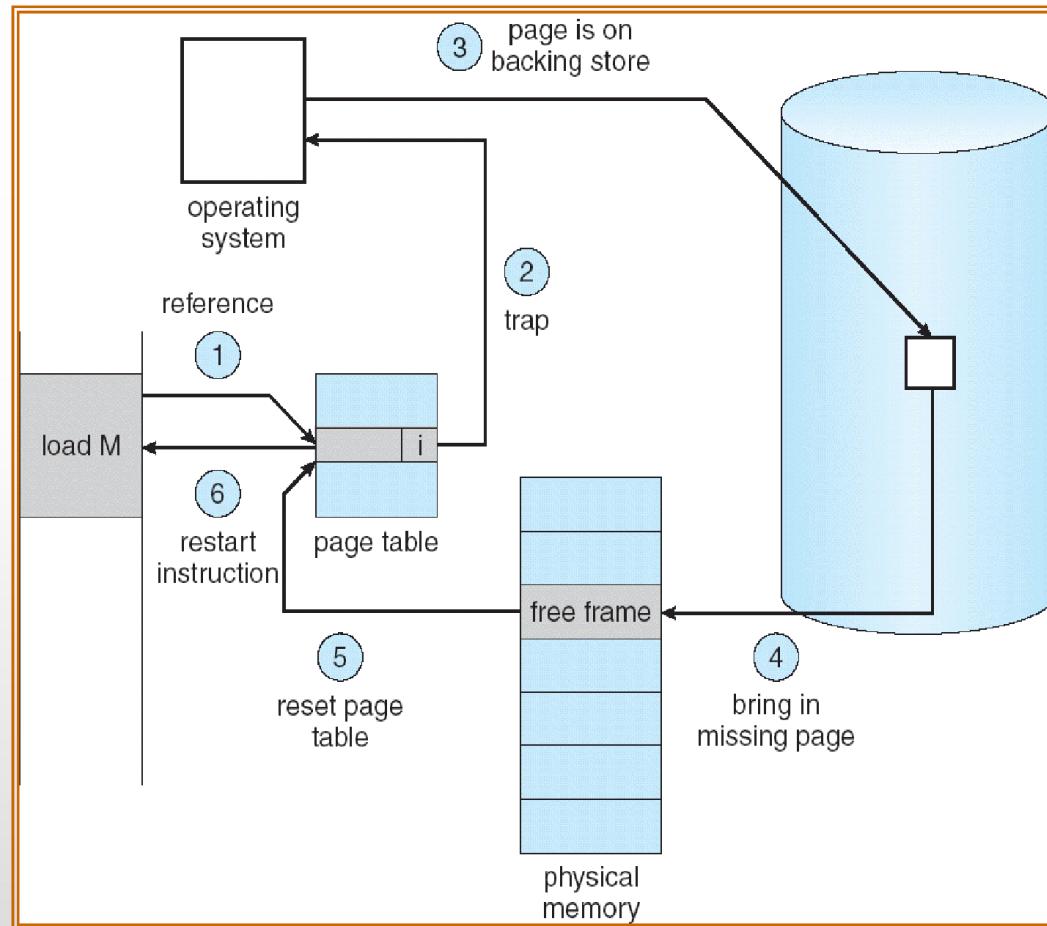
1. The process has touched a page not currently in memory.
2. Check an internal table for the target process to determine if the reference was valid (do this in hardware.)
3. If page valid, but page not resident, try to get it from secondary storage.
4. Find a free frame; a page of physical memory not currently in use. (May need to free up a page.)
5. Schedule a disk operation to read the desired page into the newly allocated frame.
6. When memory is filled, modify the page table to show the page is now resident.
7. Restart the instruction that failed

Do these steps using the figure you can see on the next page.





Demand Paging



REQUIREMENTS FOR DEMAND PAGING (HARDWARE AND SOFTWARE) INCLUDE:

Page table mechanism

Secondary storage (disk or network mechanism.)

Software support for fault handlers and page tables.

Architectural rules concerning restarting of instructions. (For instance, block moves across faulted pages.)





Demand Paging

PERFORMANCE OF DEMAND PAGING

We are interested in the effective access time: a combination of "normal" and "paged" accesses.

It's important to keep fraction of faults to a minimum. If fault ratio is "p", then

$$\text{effective_access_time} = (1 - p) * \text{memory_access_time} + p * \text{page_fault_time}.$$

Calculate the time to do a fault as shown in the text:

fault time = 10 milliseconds (why)

normal access = 100 nanoseconds (why)

How do these fit in the formula?



The Picture When Not All Pages Are In Memory

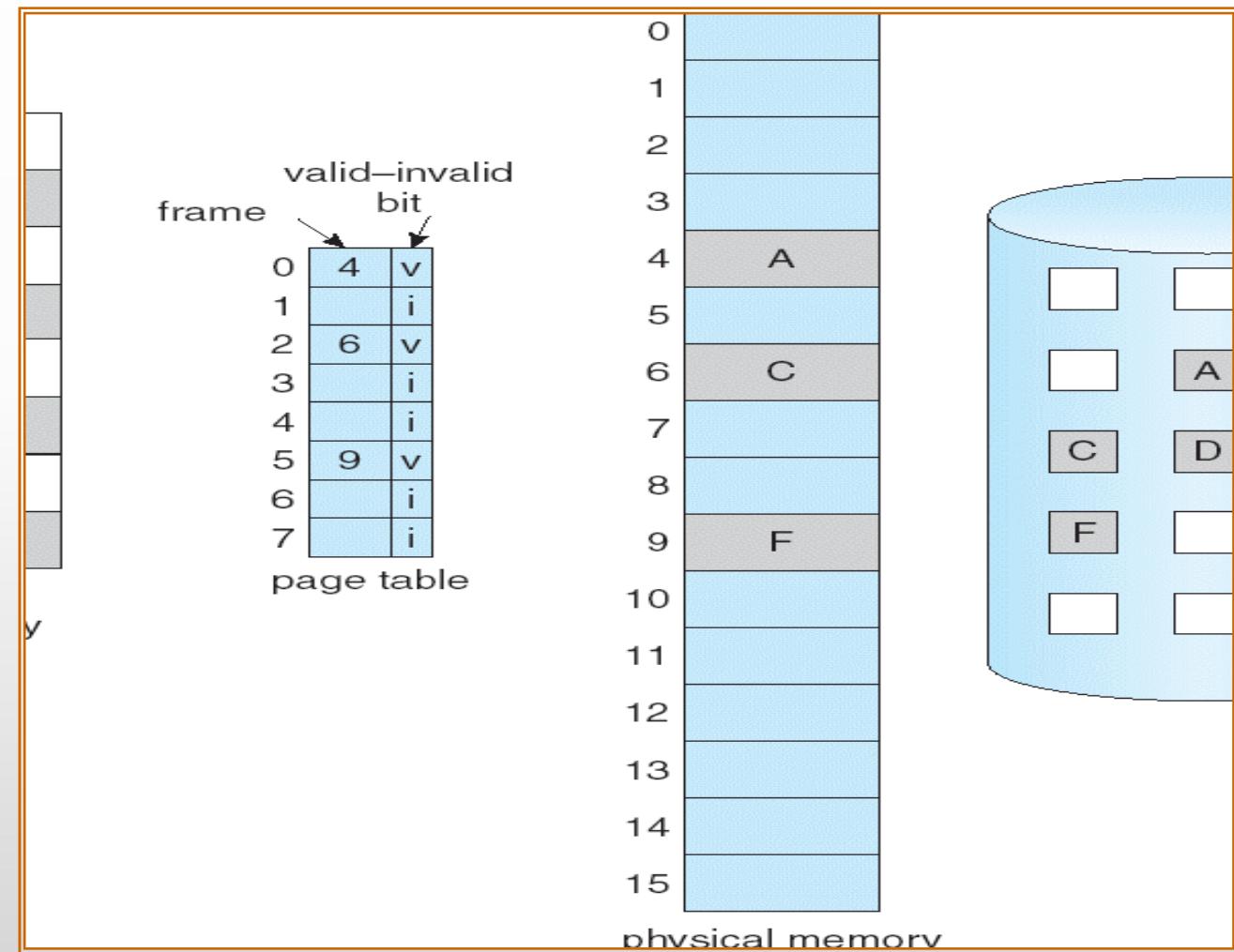
Some of the pages belonging to this process are in memory, and some are on the disk.

A bit in the page table tells where to find the page.

pmap on linux shows where these pages get mapped.

“pmap -x <pid>”

Check out also “Isof”





Page Replacement

When we over-allocate memory, we need to push out something already in memory.

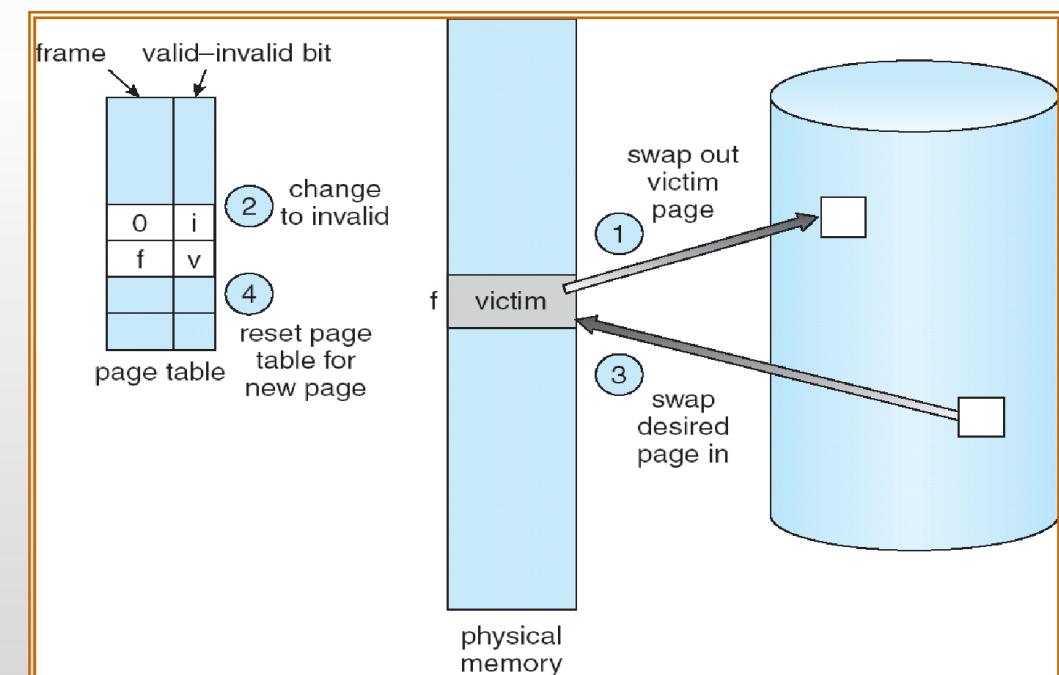
Over-allocation may occur when programs need to fault in more pages than there are physical frames to handle.

Approach: If no physical frame is free, find one not currently being touched and free it.

Steps to follow are:

1. Find requested page on disk.
2. Find a free frame.
 - a. If there's a free frame, use it
 - b. Otherwise, select a victim page.
 - c. Write the victim page to disk.
3. Read the new page into freed frame. Change page and frame tables.
4. Restart user process.

Hardware requirements include "dirty" or modified bit.





Page Replacement

PAGE REPLACEMENT ALGORITHMS:

When memory is overallocated, we can either swap out some process, or overwrite some pages. Which pages should we replace?? <--- here the goal is to minimize the number of faults.

Here is an example reference string we will use to evaluate fault mechanisms:

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

FIFO

Conceptually easy to implement; either use a time-stamp on pages, or organize on a queue.
(The queue is by far the easier of the two methods.)

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		





Page Replacement

OPTIMAL REPLACEMENT

- This is the replacement policy that results in the lowest page fault rate.
- Algorithm: Replace that page which will not be next used for the longest period of time.
- Impossible to achieve in practice; requires crystal ball.

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5





Page Replacement

LEAST RECENTLY USED (LRU)

- Replace that page which has not been used for the longest period of time.
- Results of this method considered favorable. The difficulty comes in making it work.
- Implementation possibilities:

Time stamp on pages - records when the page is last touched.

Page stack - pull out touched page and put on top

Both methods need hardware assist since the update must be done on every instruction.

So in practice this is rarely done.

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



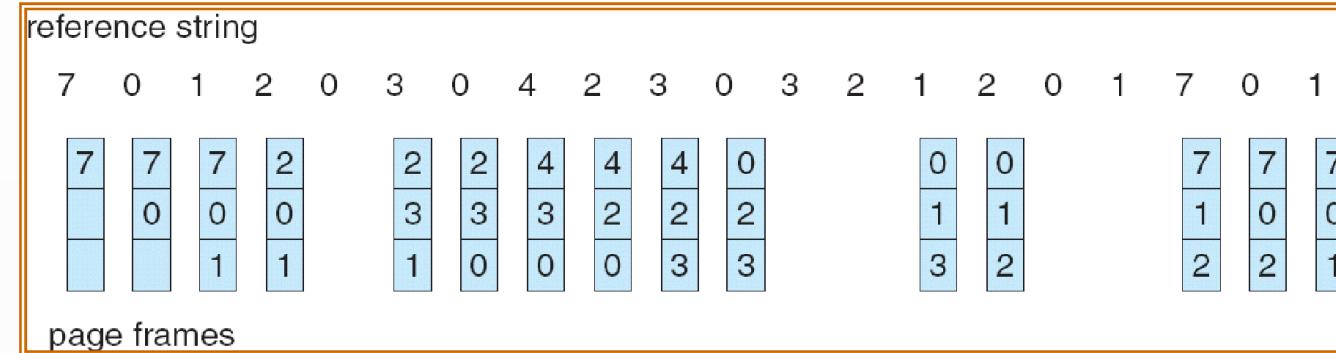


Page Replacement

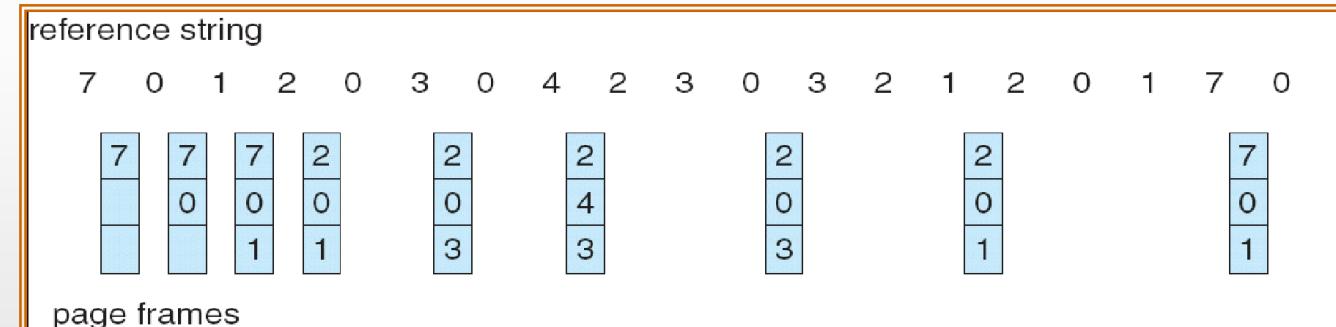
PAGE REPLACEMENT ALGORITHMS :

Using another string:

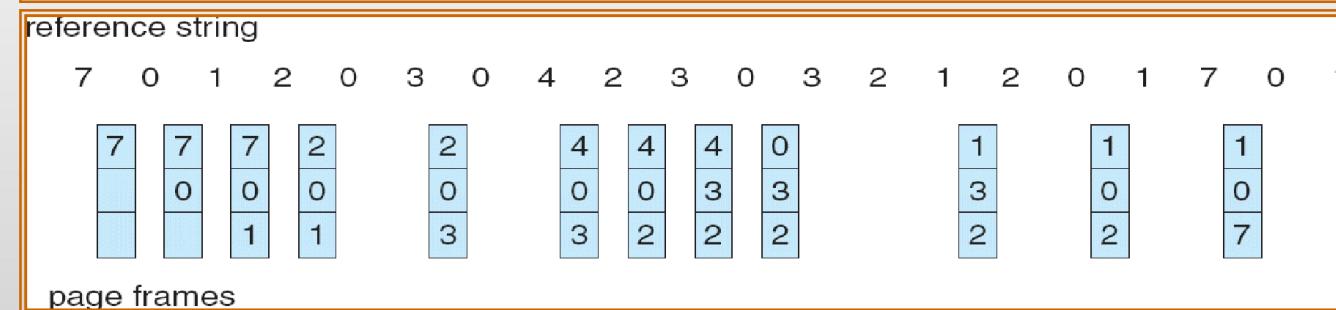
FIFO



OPTIMAL



LRU





Page Replacement

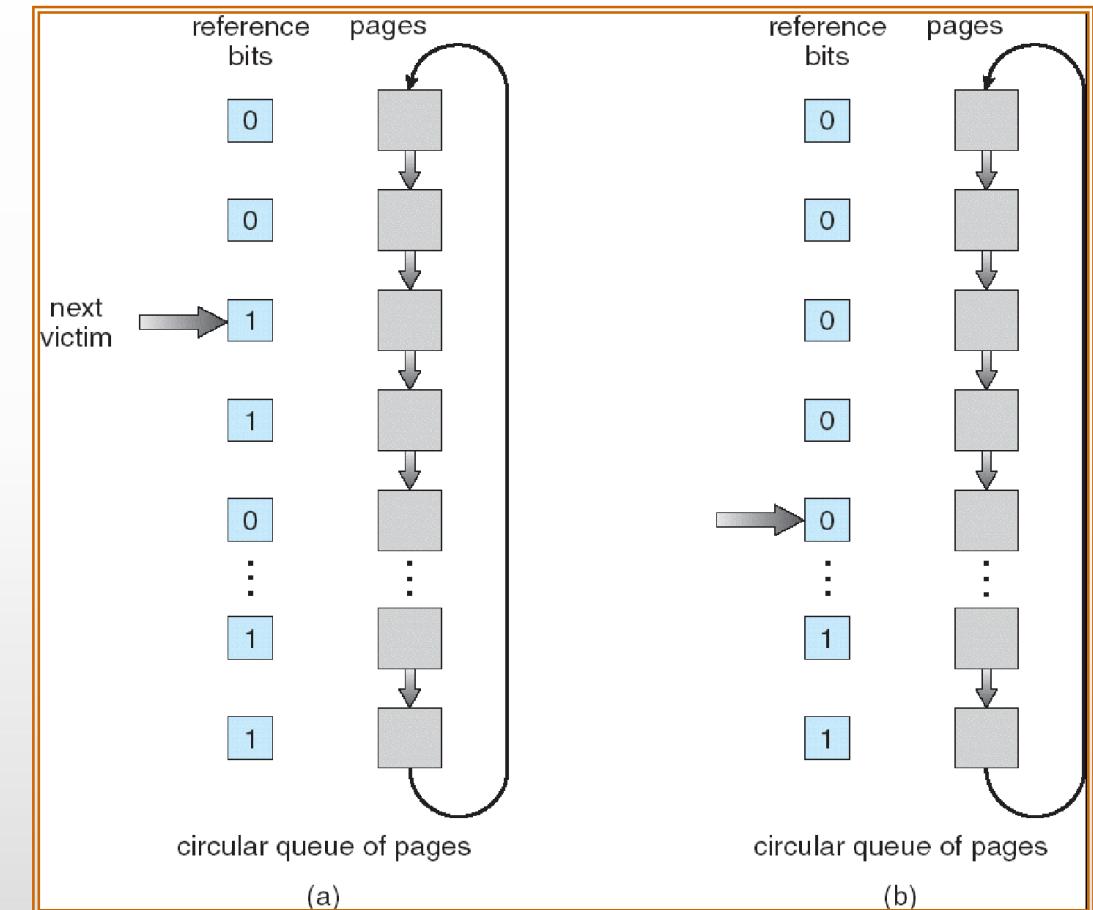
LRU APPROXIMATION

Uses a reference bit set by hardware when the page is touched. Then when a fault occurs, pick a page that hasn't been referenced.

Additional reference bits can be used to give some time granularity. Then pick the page with the oldest timestamp.

Second chance replacement: pick a page based on FIFO. If its reference bit is set, give it another chance. Envision this as a clock hand going around a circular queue. The faster pages are replaced, the faster the hand goes.

Maintain a modified bit, and preferentially replace unmodified pages.



**Second-Chance (clock)
Page-Replacement Algorithm**





Page Replacement

ADD HOC (OR ADD-ON) ALGORITHMS

These methods are frequently used over and above the standard methods given above.

Maintain pools of free frames; write out loser at leisure

Occasional writes of dirties - make clean and then we can use them quickly when needed.

Write out and free a page but remember a page id in case the page is needed again - even though a page is in the free pool, it can be recaptured by a process (a soft page fault.)





Page Allocation

ALLOCATION OF FRAMES:

What happens when several processes contend for memory?

What algorithm determines which process gets memory - is page management a global or local decision?

A good rule is to ensure that a process has at least a minimum number of pages.

This minimum ensures it can go about its business without constantly thrashing.

ALLOCATION ALGORITHMS

Local replacement -- the process needing a new page can only steal from itself.

(Doesn't take advantage of entire picture.)

Global replacement - sees the whole picture, but a memory hog steals from everyone else

Can divide memory equally, or can give more to a needier process.

Should high priority processes get more memory?



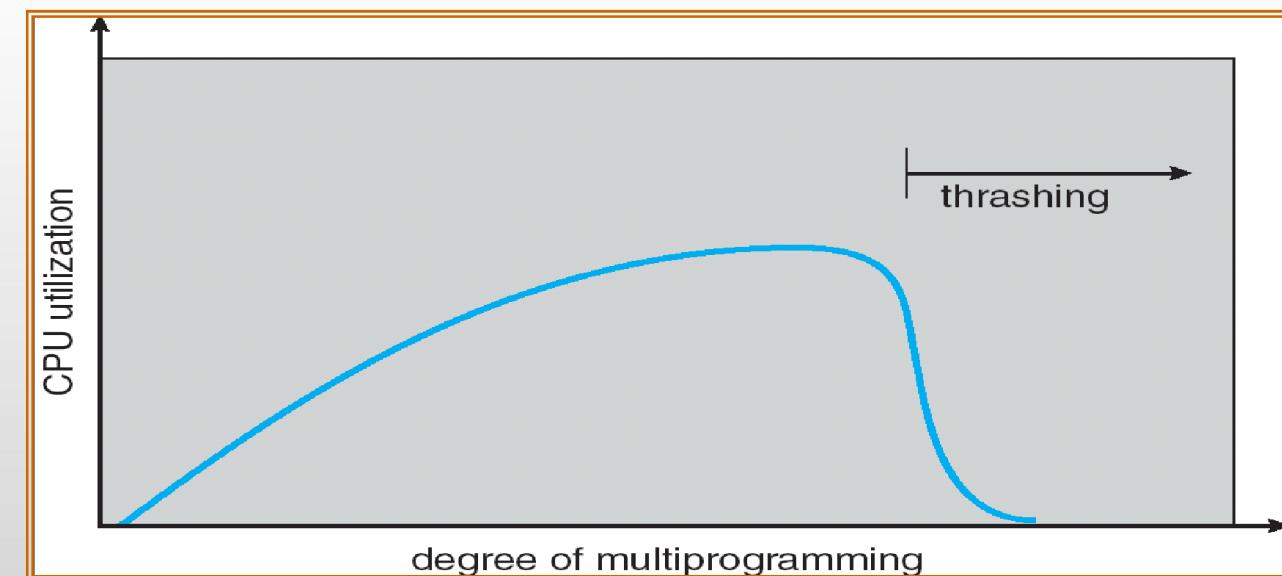


Thrashing

Suppose there are too few physical pages (less than the logical pages being actively used). This reduces CPU utilization, and may cause increase in multiprogramming needs defined by locality.

A program will thrash if all pages of its locality aren't present in the working set.

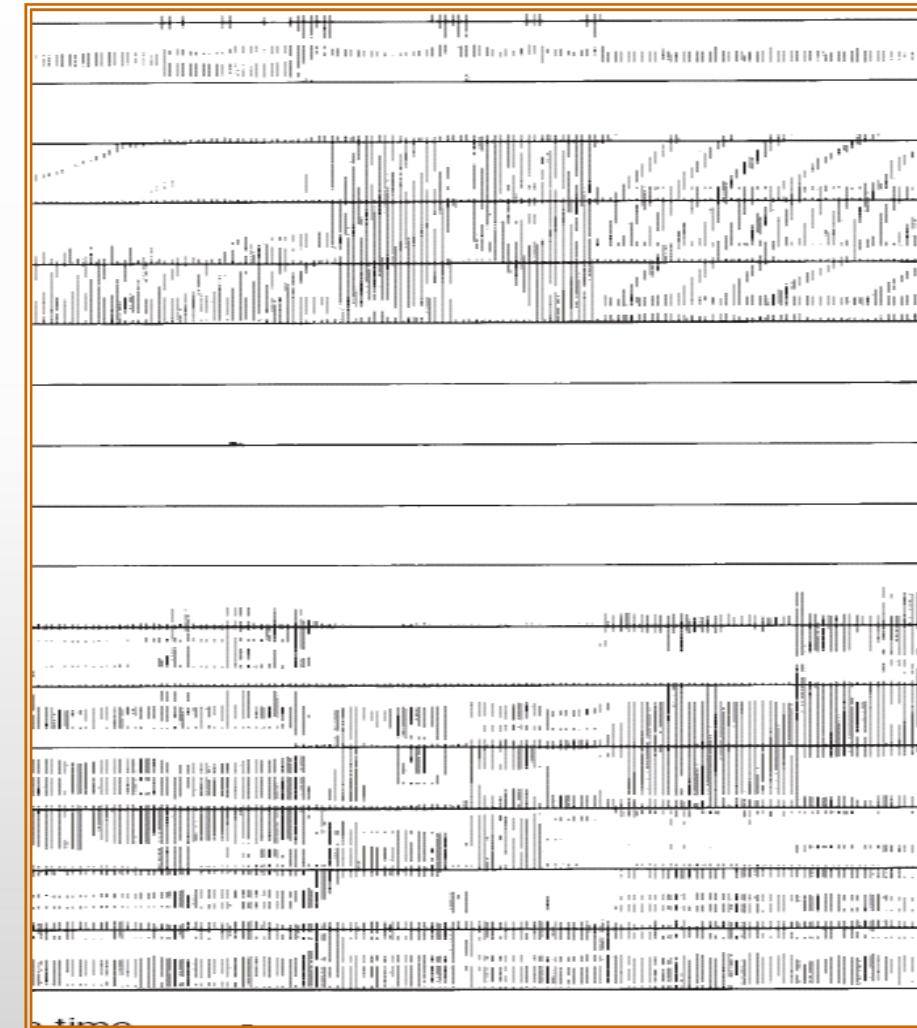
Two programs thrash if they fight each other too violently for memory.





Locality of Reference

Locality of reference: Programs access memory near where they last accessed it.

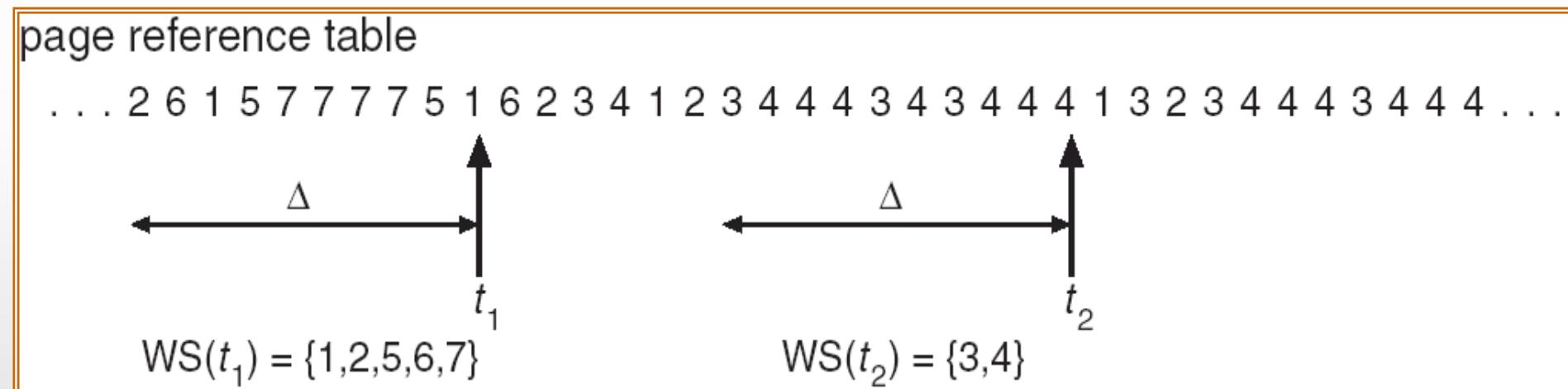




Working Set Model

WORKING SET MODEL

The pages used by a process within a window of time are called its working set.



Changes continuously - hard to maintain an accurate number. How can the system use this number to give optimum memory to the process?

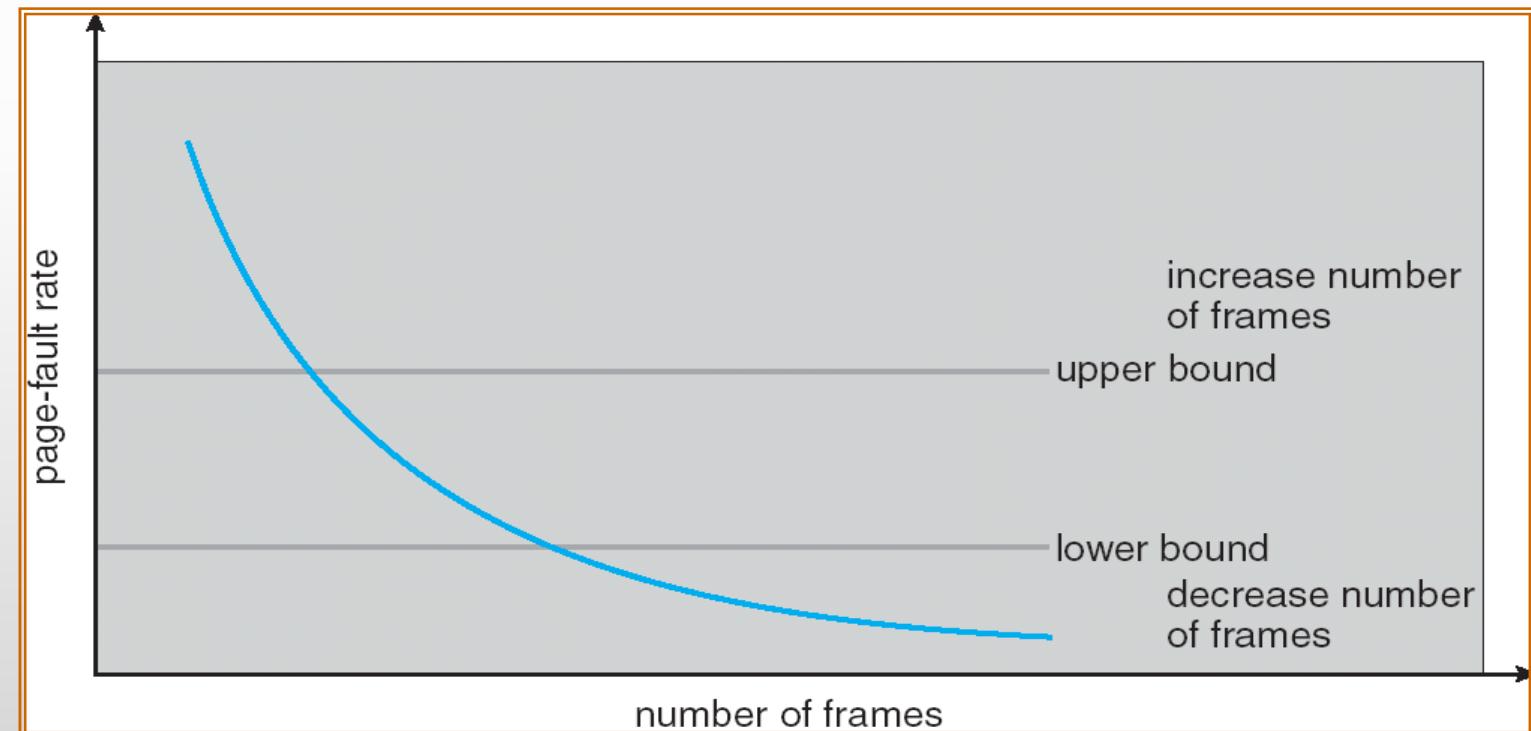




Working Set Model

PAGE FAULT FREQUENCY

This is a good indicator of thrashing. If the process is faulting heavily, allocate it more frames. If faulting very little, take away some frames.





Other Issues

PREPAGING

- Bring lots of pages into memory at one time, either when the program is initializing, or when a fault occurs.
- Uses the principle that a program often uses the page right after the one previously accessed (locality of reference.)

PAGE SIZE

- If too big, there's considerable fragmentation and unused portions of the page.
- If too small, table maintenance is high and so is I/O.
- Has ramifications in code optimization.





Memory Mapped IO

Memory Mapped IO

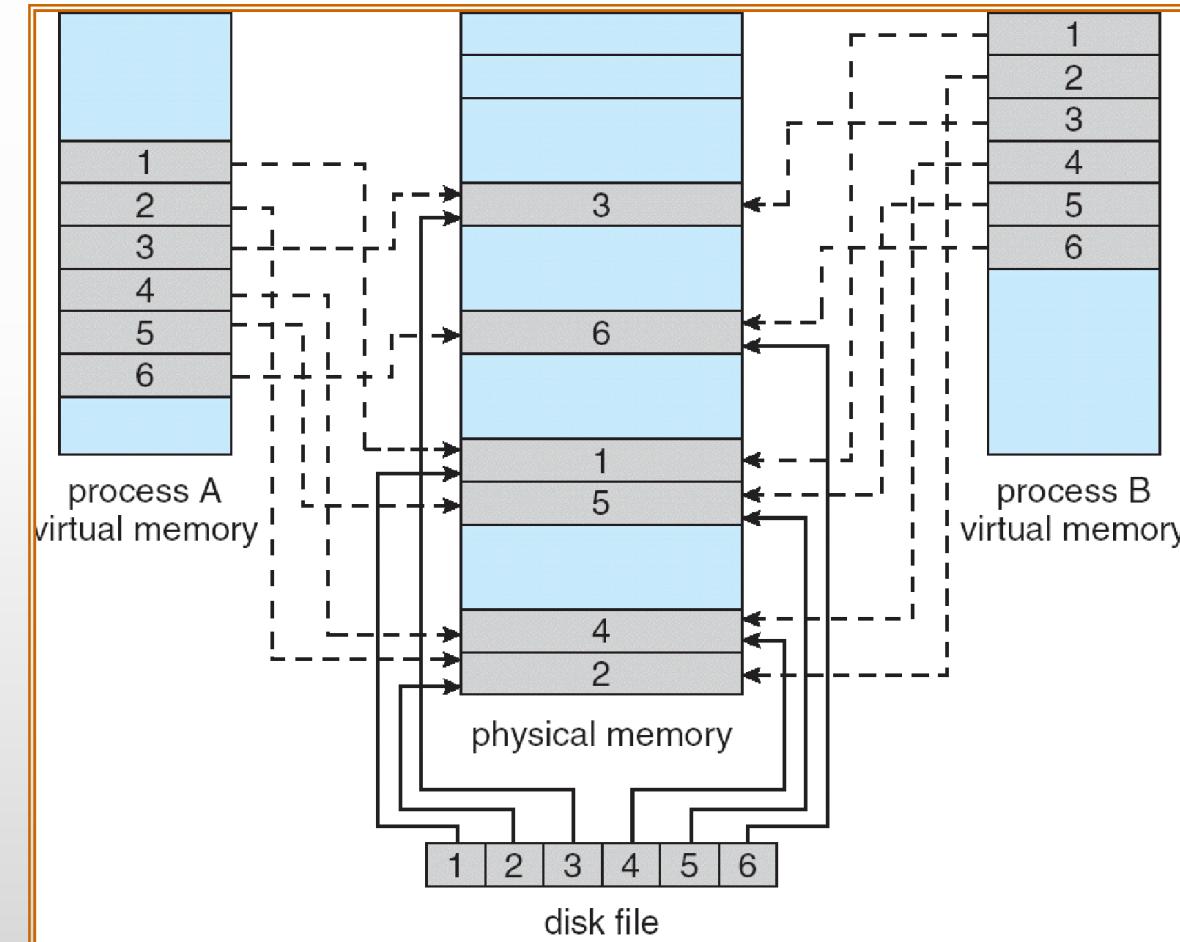
- Allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging. Multiple page-sized portions of the file are read from the file system into physical pages. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **read()** **write()** system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared





Memory Mapped IO

Memory Mapped IO





Thank You Very Much

Benjamin Kommey

bkommey.coe@knust.edu.gh

nii_kommey@msn.com

050 770 32 86

Skype_id: calculus.affairs

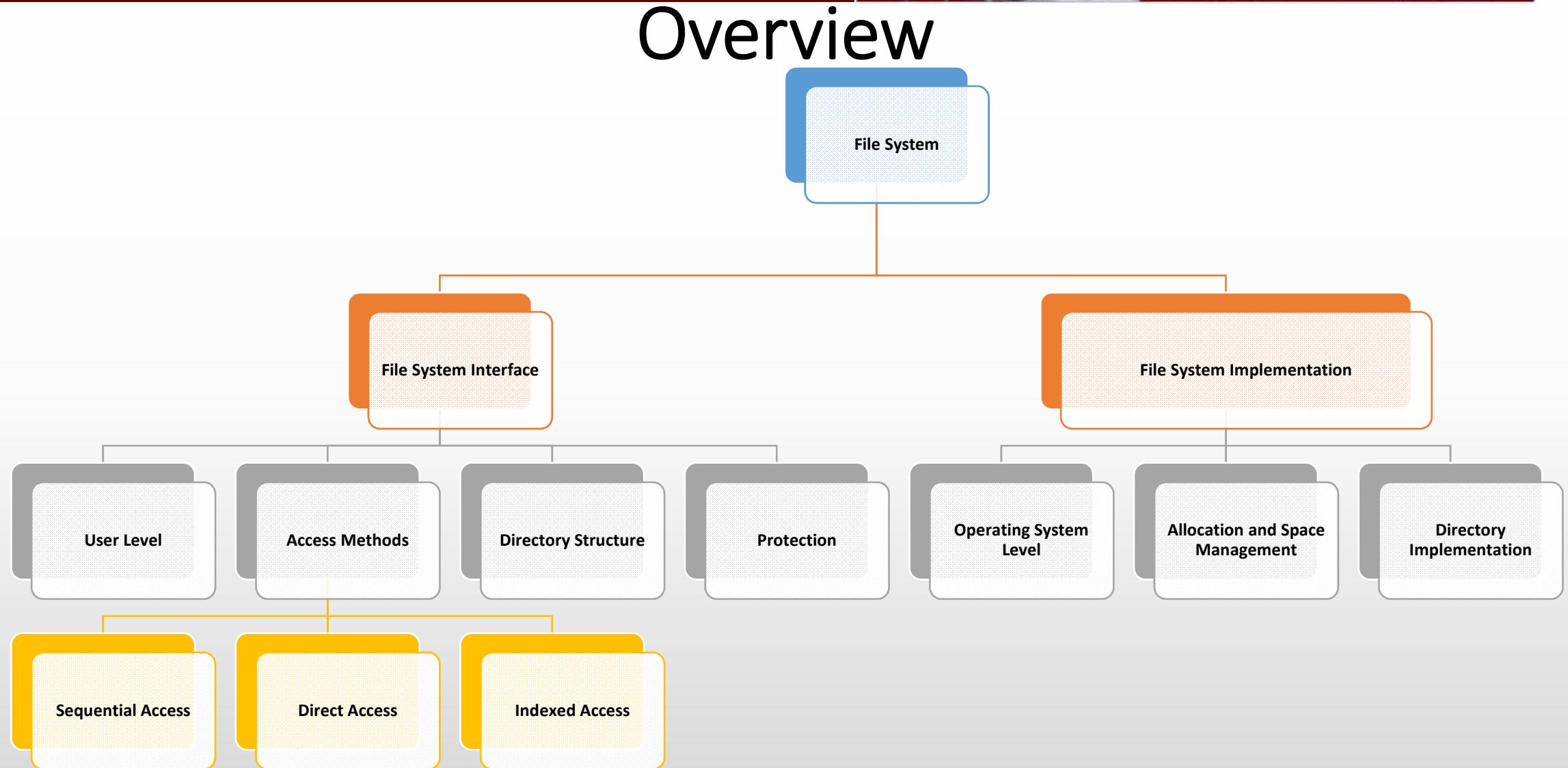




“ Television is NOT real life. In real life people actually have to leave the coffee shop and go to jobs.”

Bill Gates







File Concept

A file is a collection of related bytes having meaning only to the creator.

A file can be "free formed", indexed, or structured.

A file is an entry in a directory.

A file may have attributes (name, creator, date, type, permissions)

A file may have structure (O.S. may or may not know about this.) It's a tradeoff of power versus overhead.
For example,

- An Operating System understands program image format in order to create a process.
- The UNIX **shell** understands how directory files look.
(In general the UNIX **kernel** doesn't interpret files.)
- Usually the Operating System understands and interprets file types.





File System Interface

A file can have various kinds of structure

None - sequence of words, bytes

Simple record structure

- Lines

- Fixed length

- Variable length

Complex Structures

- Formatted document

- Relocatable load file

Who interprets this structure?

- Operating system

- Program





File Attributes

Name – only information kept in human-readable form

Identifier – unique tag (number) identifies file within file system

Type – needed for systems that support different types

Location – pointer to file location on device

Size – current file size

Protection – controls who can do reading, writing, executing

Time, date, and user identification – data for protection, security, and usage monitoring

Information about file is kept in the directory structure, which is maintained on the disk.





Blocking /Packing

Blocking (packing) occurs when some entity,
(either the user or the Operating System) must pack bytes into a physical block.

- a) Block size is fixed for disks, variable for tape
- b) Size determines maximum internal fragmentation
- c) We can allow reference to a file as a set of logical records (addressable units) and then divide (or pack) logical records into physical blocks.

What does it mean to “open” a file??





Access Methods

If files had only one "chunk" of data, life would be simple.

But for large files, the files themselves may contain structure, making access faster.

SEQUENTIAL ACCESS

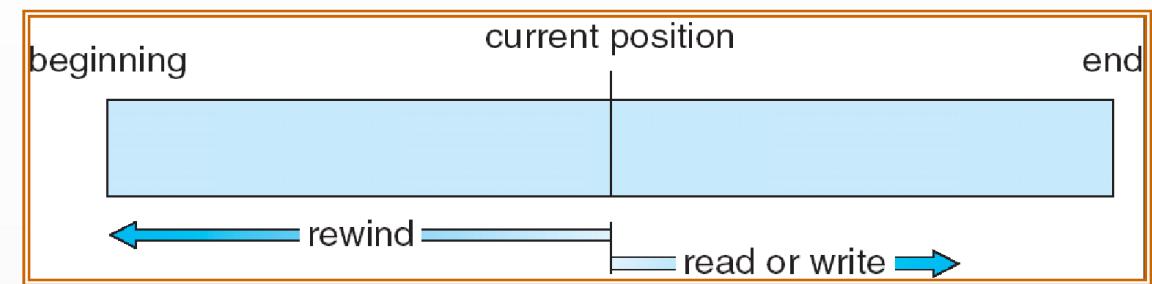
Implemented by the file system.

Data is accessed one record right after the last.

Reads cause a pointer to be moved ahead by one.

Writes allocate space for the record and move the pointer to the new End Of File.

Such a method is reasonable for tape





Direct Access Method

Method useful for disks.

The file is viewed as a numbered sequence of blocks or records.

There are no restrictions on which blocks are read/written in any order.

User now says "read n" rather than "read next".

"n" is a number relative to the beginning of file, not relative to an absolute physical disk location.





Indexed Access Method

Built on top of direct access and often implemented by a user utility.

Indexed ID plus pointer.

An index block says what's in each remaining block or contains pointers to blocks containing particular items. Suppose a file contains many blocks of data arranged by name alphabetically.

Example 1:

Index contains the name appearing as the first record in each block.

There are as many index entries as there are blocks.

Example 2:

Index contains the block number where "A" begins, where "B" begins, etc. Here there are only 26 index entries.



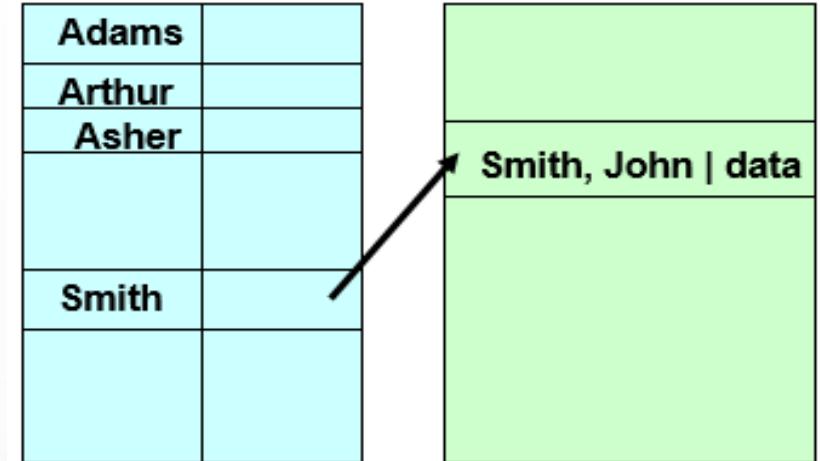


Indexed Access Method

Example 1

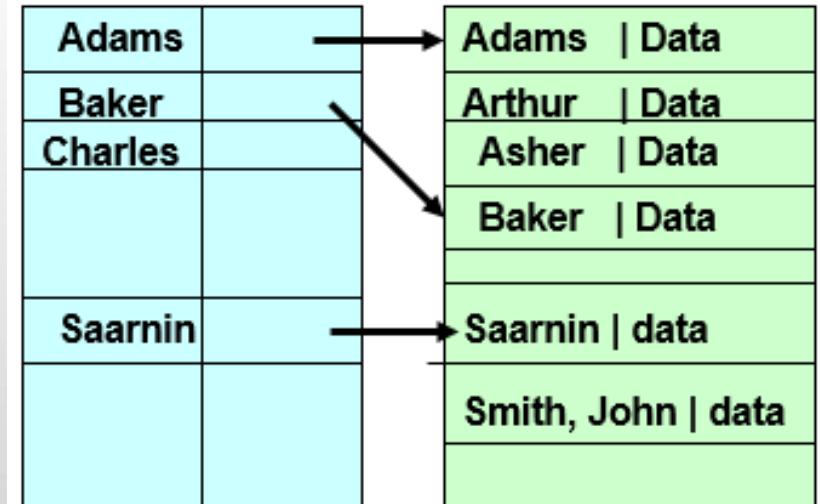
Index contains the name appearing as the first record in each block

There are as many index entries as there are blocks



Example 2

Index contains the block number where A begins, where B begins, etc. Here there are only 26 index entries





Directory Structure

Directories maintain information about files:

For a large number of files, may want a directory structure - directories under directories.

Information maintained in a directory:

Name	The user visible name.
Type	The file is a directory, a program image, a user file, a link, etc.
Location	Device and location on the device where the file header is located.
Size	Number of bytes/words/blocks in the file.
Position	Current next-read/next-write pointers.
Protection	Access control on read/write/ execute/delete.
Usage	Open count
Usage	time of creation/access, etc.
Mounting	a file system occurs when the root of one file system is "grafted" into the existing tree of another file system.

There is a need to **PROTECT** files and directories.

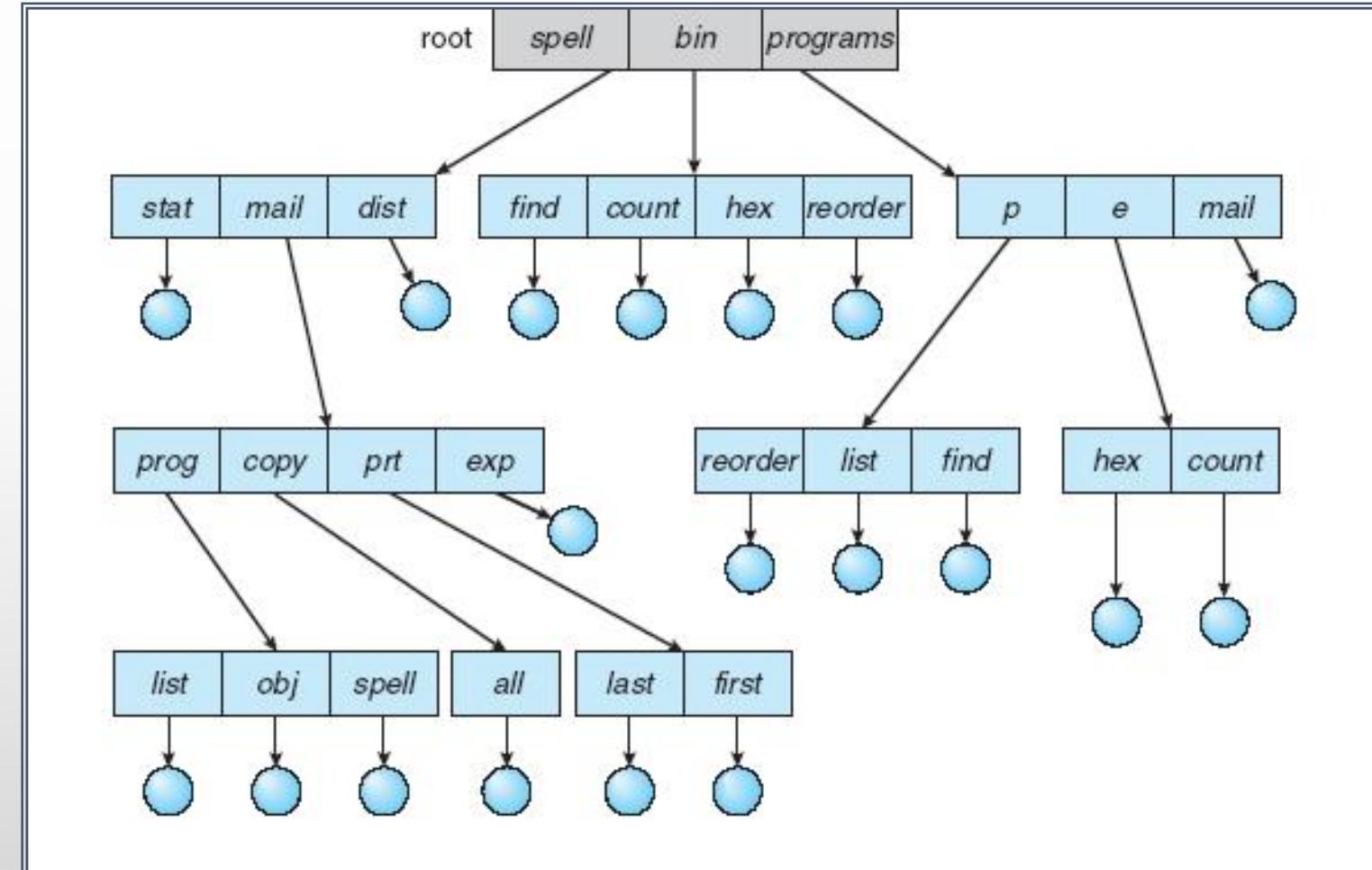
Actions that might be protected include: **read, write, execute, append, delete, list**





Tree-Structured Directory

Tree-Structured Directory





Other Issues

Mounting

Attaching portions of the file system into a directory structure.

Sharing:

Sharing must be done through a **protection** scheme

May use networking to allow file system access between systems

- Manually via programs like FTP or SSH

- Automatically, seamlessly using **distributed file systems**

- Semi automatically via the **world wide web**

Client-server model allows clients to mount remote file systems from servers

- Server can serve multiple clients

- Client and user-on-client identification is insecure or complicated

- NFS** is standard UNIX client-server file sharing protocol

- CIFS** is standard Windows protocol

- Standard operating system file calls are translated into remote calls





File Protection

File owner/creator should be able to control:

what can be done
by whom

Types of access

Read

Write

Execute

Append

Delete

List

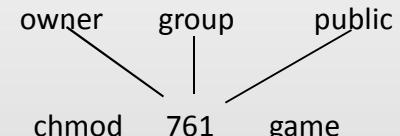
Mode of access: read, write, execute

Three classes of users

a) owner access 7	\Rightarrow	RWX
		1 1 1
b) group access 6	\Rightarrow	RWX
		1 1 0
c) public access 1	\Rightarrow	RWX
		0 0 1

Ask manager to create a group (unique name), say G, and add some users to the group.

For a particular file (say *game*) or subdirectory, define an appropriate access.



Attach a group to a file

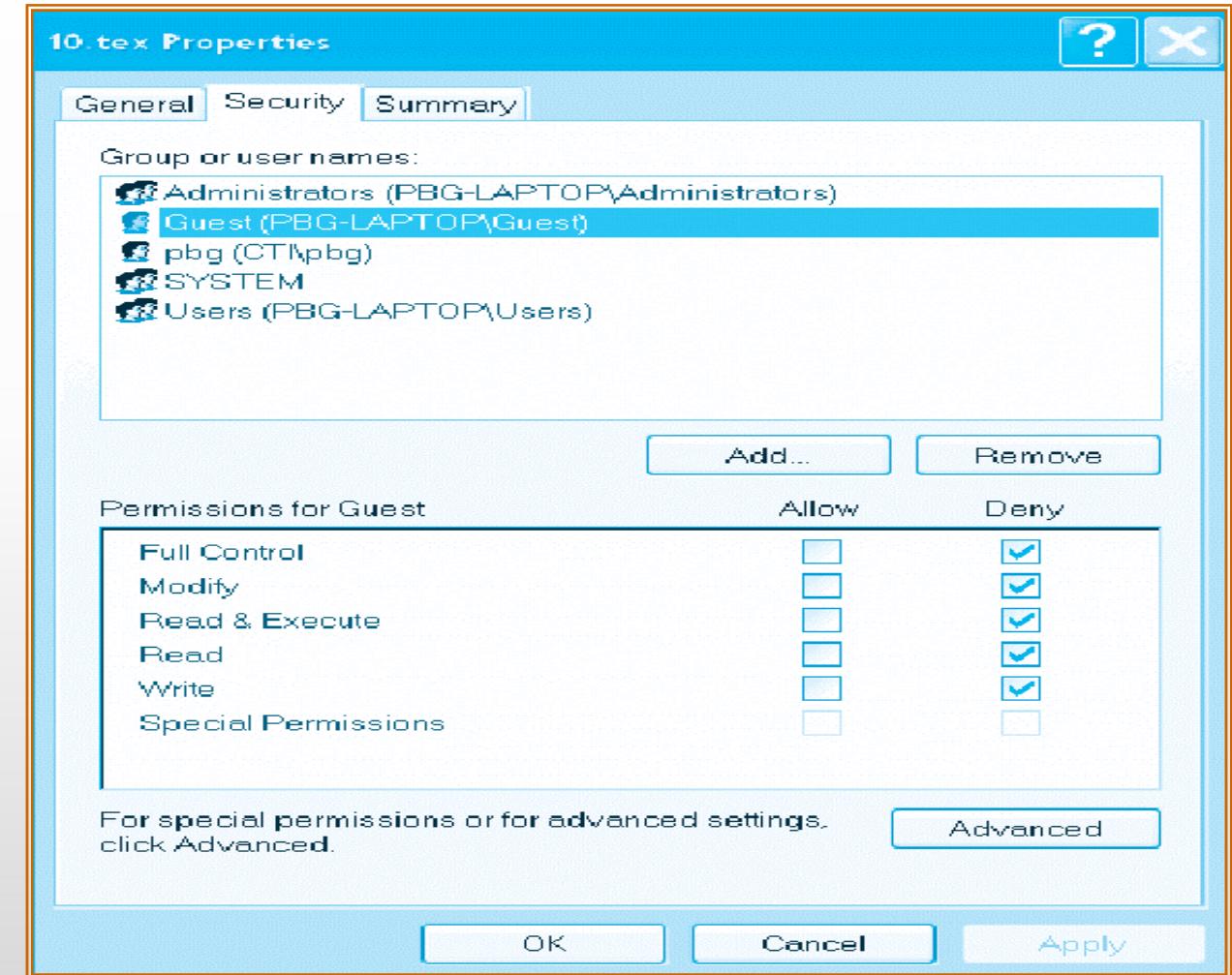
"chgrp G game"





File Protection

Example on Windows XP





File System Implementation

FILE SYSTEM STRUCTURE:

When talking about “the file system”, you are making a statement about both the rules used for file access, and about the algorithms used to implement those rules. Here’s a breakdown of those algorithmic pieces.

Application Programs The code that's making a file request.

Logical File System This is the highest level in the OS; it does protection, and security. Uses the directory structure to do name resolution.

File-organization Module Here we read the file control block maintained in the directory so we know about files and the logical blocks where information about that file is located.

Basic File System Knowing specific blocks to access, we can now make generic requests to the appropriate device driver.

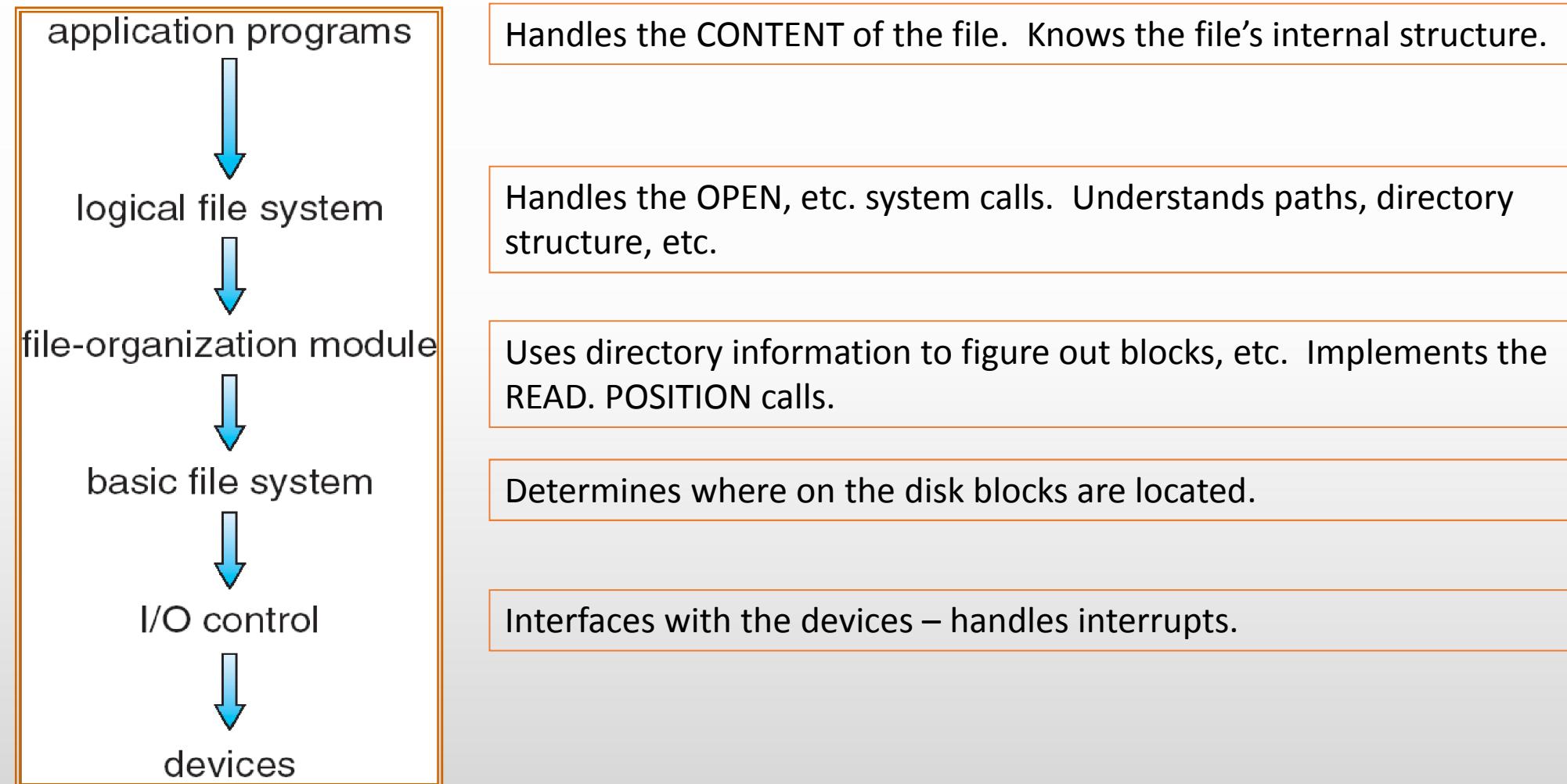
IO Control These are device drivers and interrupt handlers. They cause the device to transfer information between that device and CPU memory.

Devices The disks / tapes / etc.



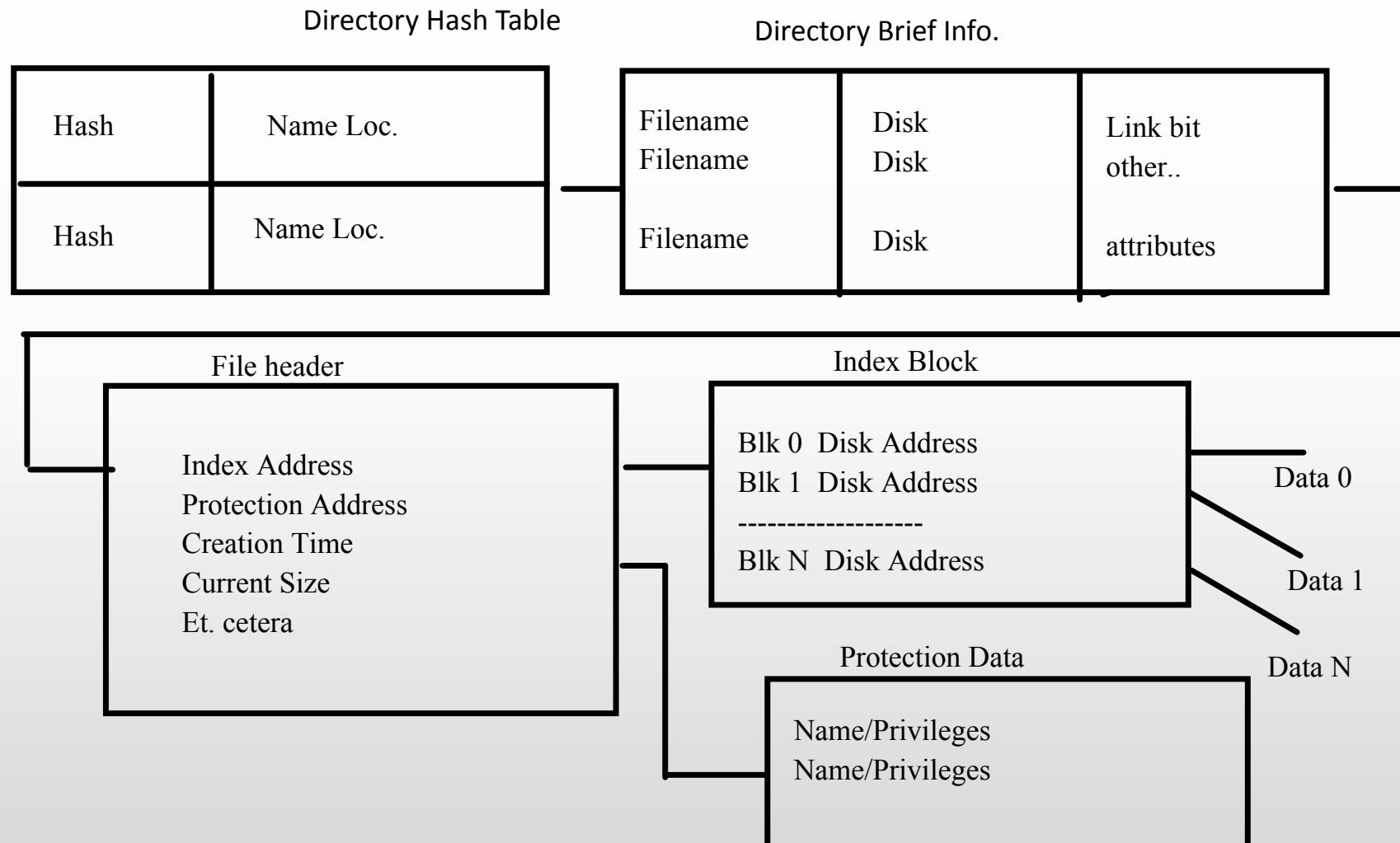


Layered File System





Directory and File Structure Example



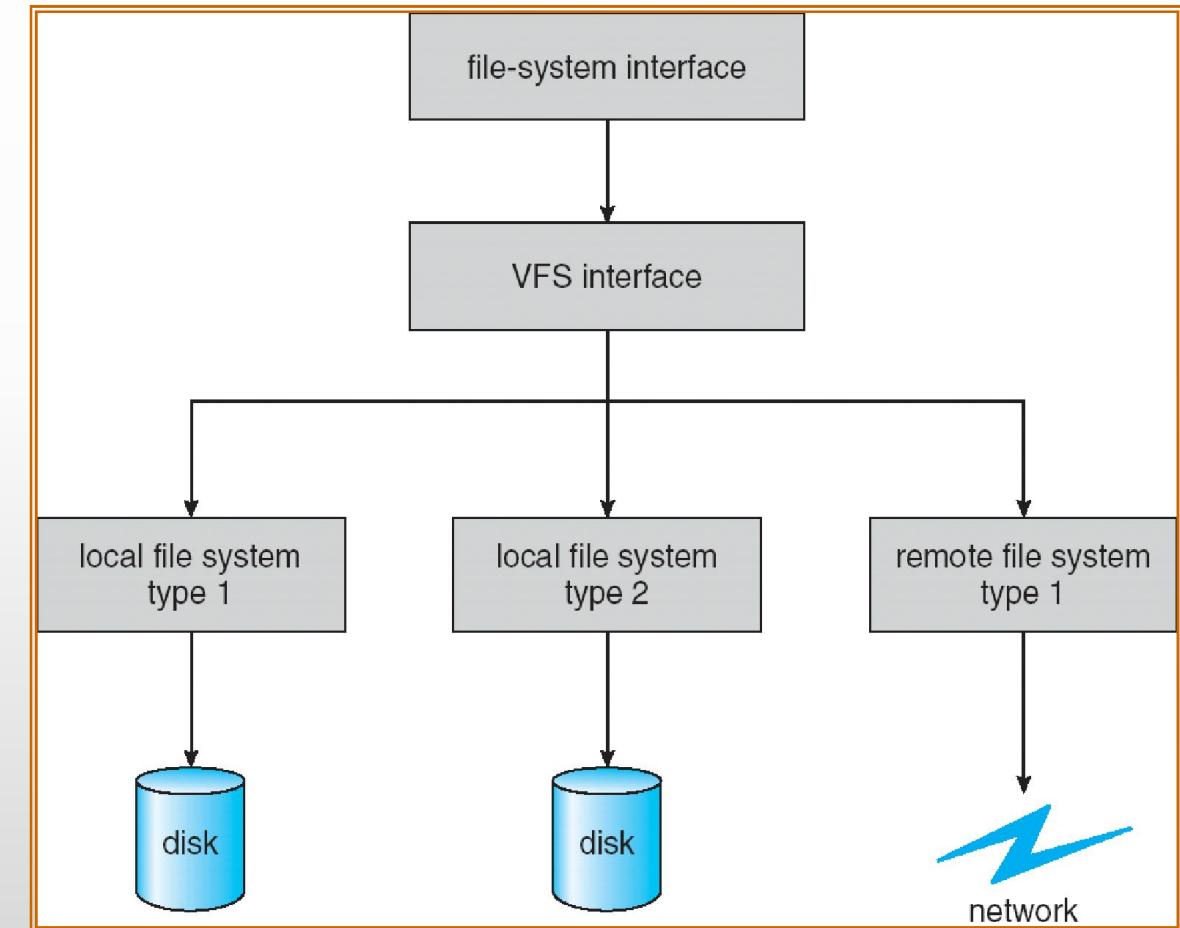


Virtual File System

Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.

VFS allows the same system call interface (the API) to be used for different types of file systems.

The API is to the VFS interface, rather than any specific type of file system.



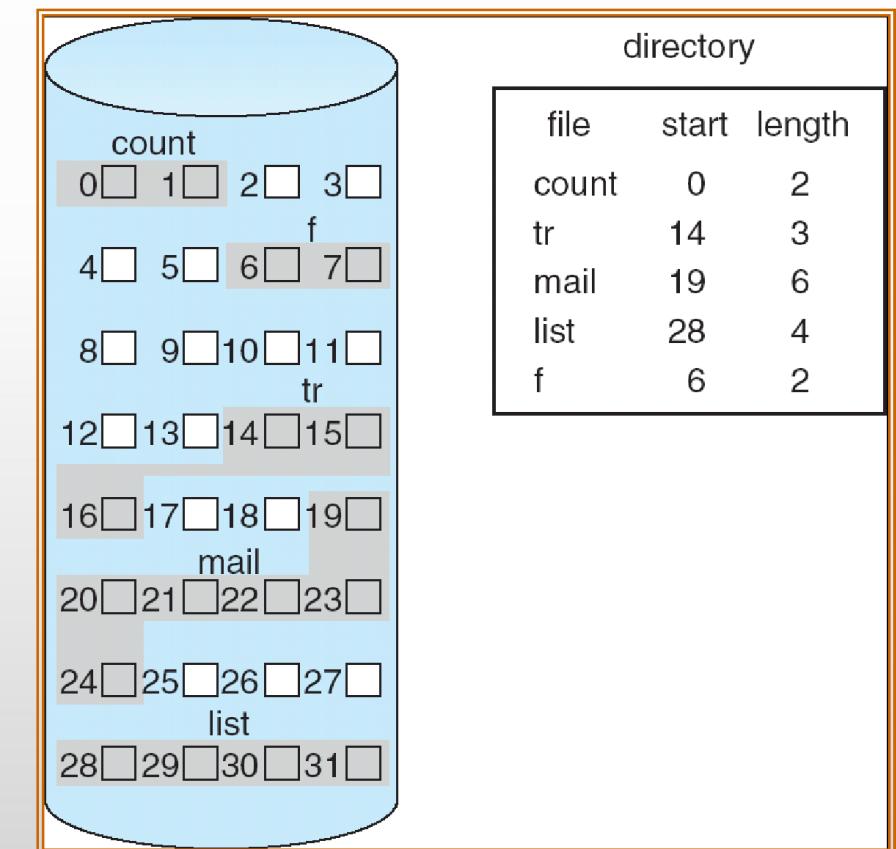


Allocation Methods

CONTIGUOUS ALLOCATION

Method: Lay down the entire file on contiguous sectors of the disk. Define by a <first block location, length >.

- a) Accessing the file requires a minimum of head movement.
- b) Easy to calculate block location: block i of a file, starting at disk address b , is $b + i$.
- c) Difficulty is in finding the contiguous space, especially for a large file. Problem is one of dynamic allocation (first fit, best fit, etc.) which has external fragmentation. If many files are created/deleted, compaction will be necessary.



It's hard to estimate at create time what the size of the file will ultimately be. What happens when we want to extend the file -- we must either terminate the owner of the file, or try to find a bigger hole.





Allocation Methods

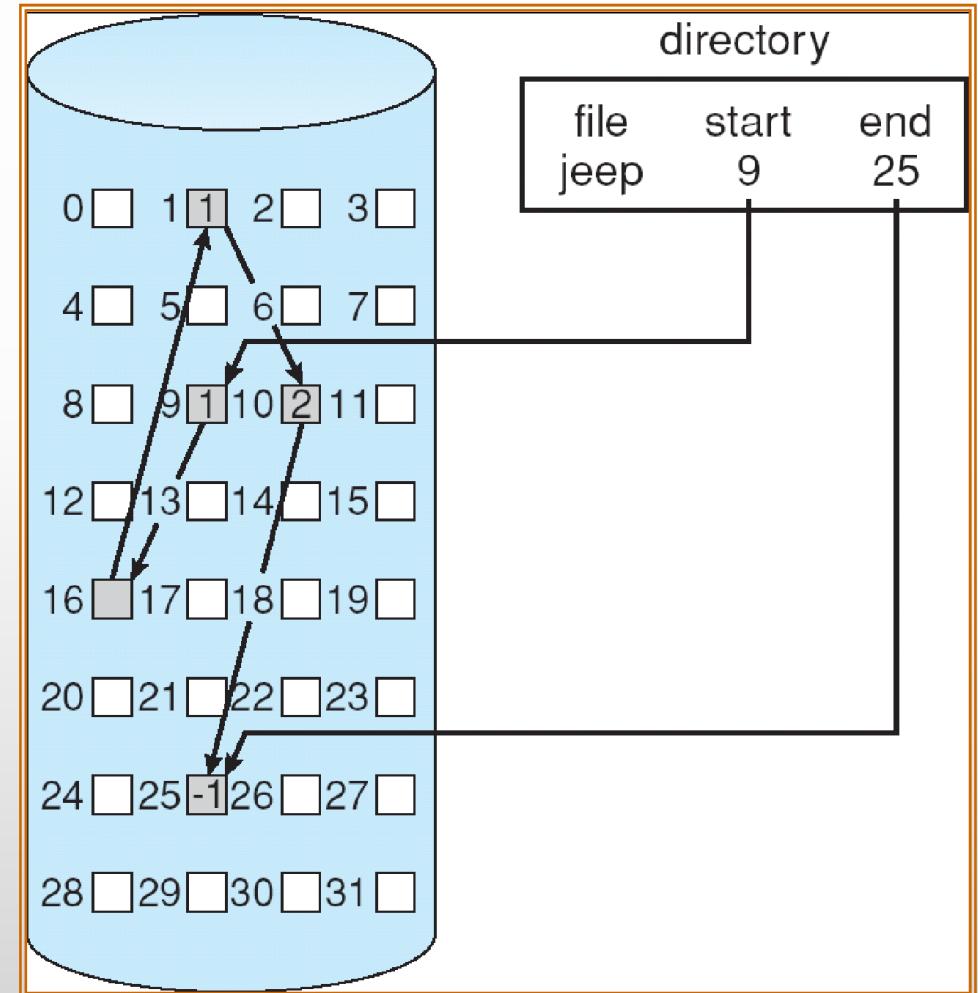
LINKED ALLOCATION

Each file is a linked list of disk blocks, scattered anywhere on the disk.

At file creation time, simply tell the directory about the file. When writing, get a free block and write to it, enqueueing it to the file header.

There's no external fragmentation since each request is for one block.

Method can only be effectively used for sequential files.





Allocation Methods

LINKED ALLOCATION

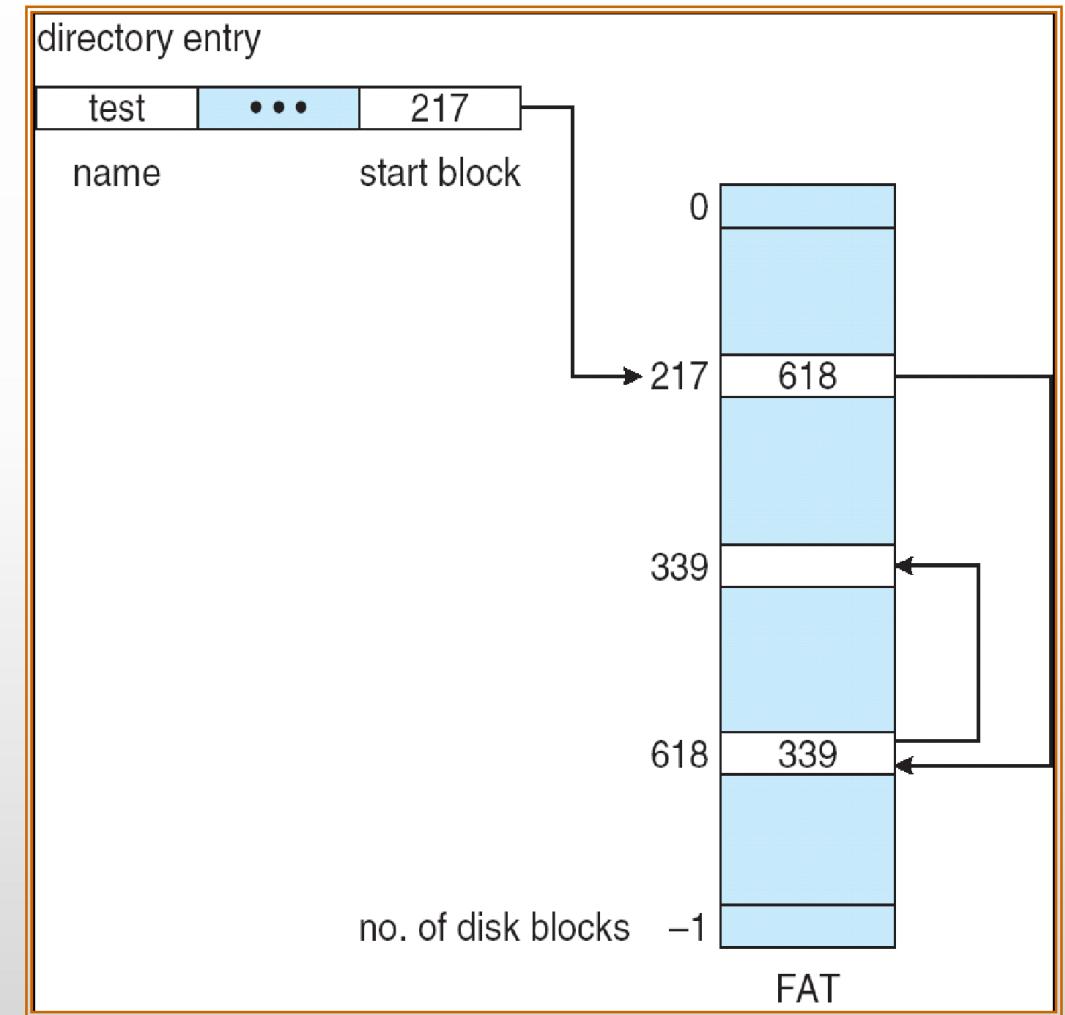
Pointers use up space in each block. Reliability is not high because any loss of a pointer loses the rest of the file.

A File Allocation Table is a variation of this.

It uses a separate disk area to hold the links.

This method doesn't use space in data blocks. Many pointers may remain in memory.

A FAT file system is used by MS-DOS.





Allocation Methods

INDEXED ALLOCATION

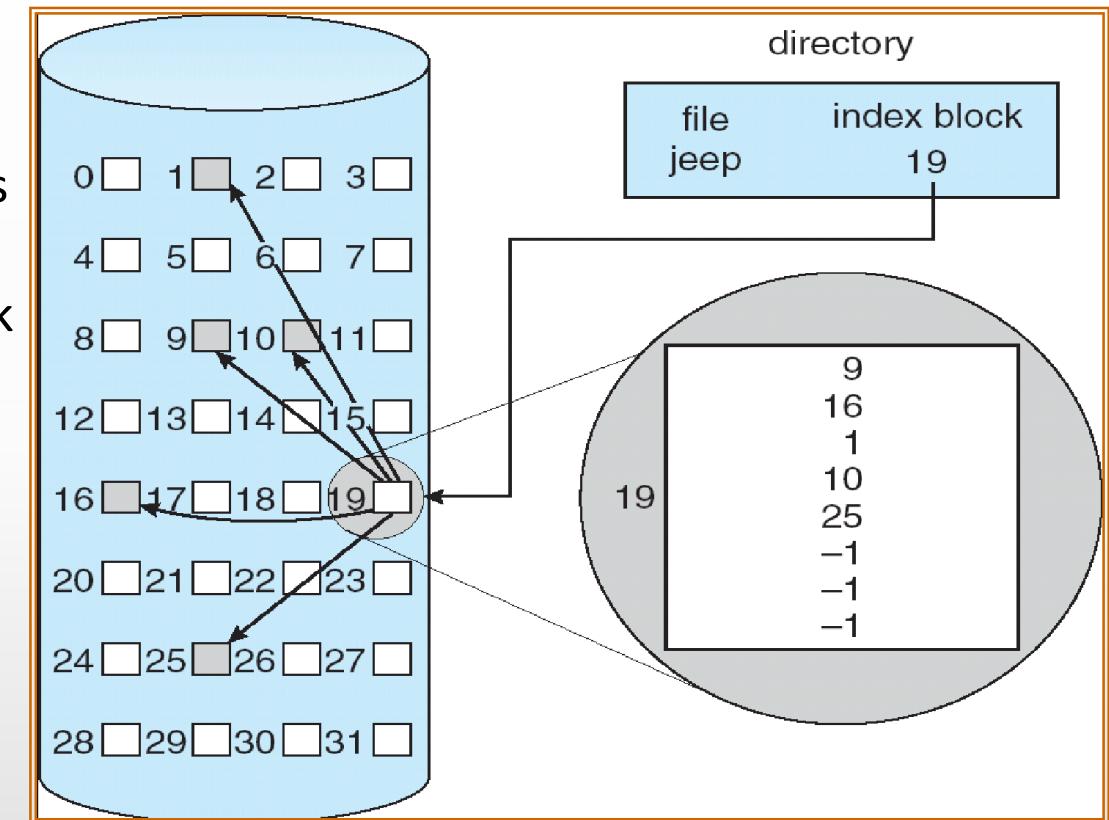
Each file uses an index block on disk to contain addresses of other disk blocks used by the file.

When the **i th** block is written, the address of a free block is placed at the **i th** position in the index block.

Method suffers from wasted space since, for small files, most of the index block is wasted. What is the optimum size of an index block?

If the index block is too small, we can:

- Link several together
- Use a multilevel index



UNIX keeps 12 pointers to blocks in its header. If a file is longer than this, then it uses pointers to single, double, and triple level index blocks.

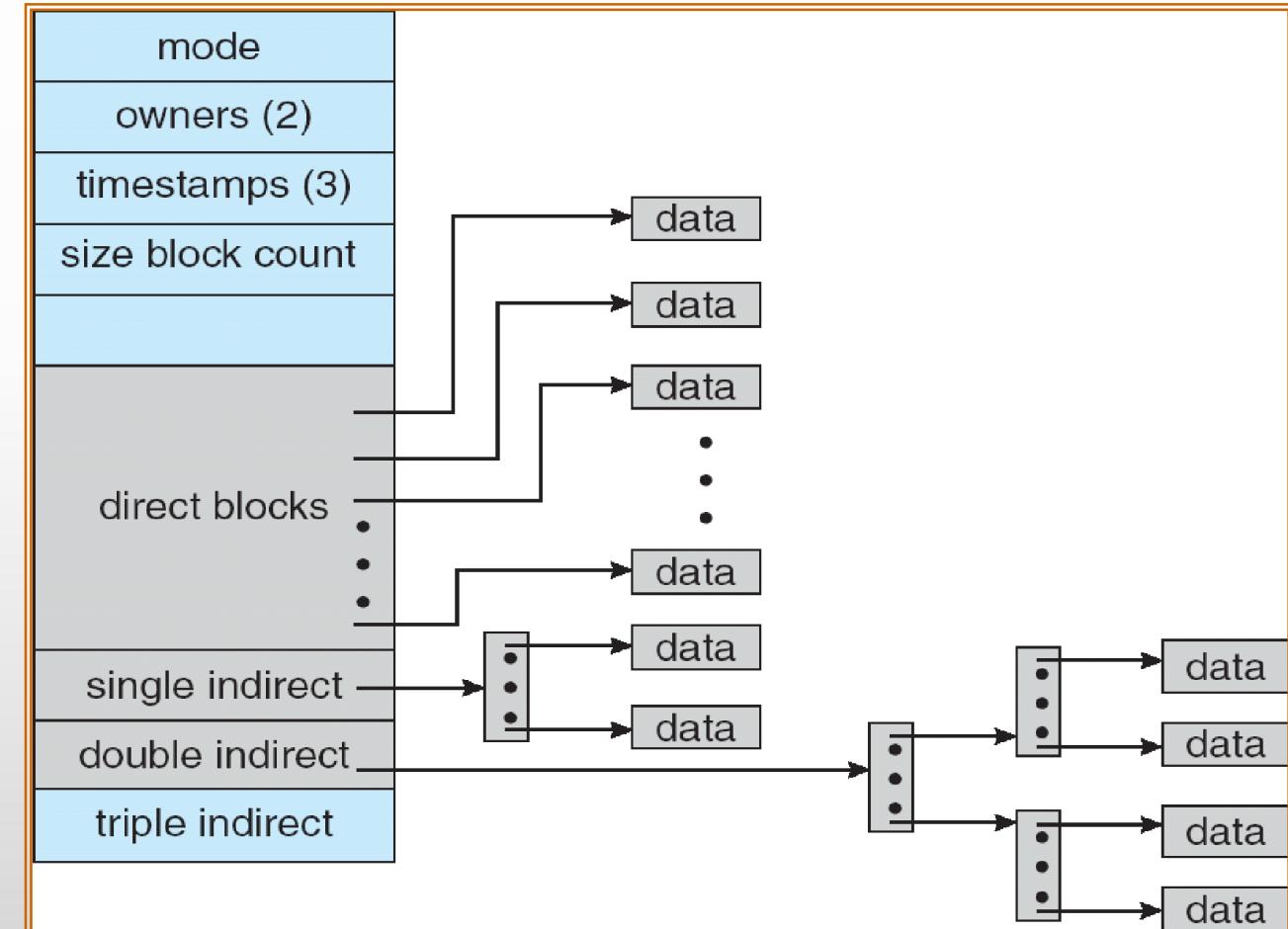




Allocation Methods

UNIX METHOD:

Note that various mechanisms are used here so as to optimize the technique based on the size of the file.





Performance Issues

It's difficult to compare mechanisms because usage is different. Let's calculate, for each method, the number of disk accesses to read block i from a file:

contiguous: 1 access from location **start + i**.

linked: $i + 1$ accesses, reading each block in turn.
(is this a fair example?)

index: 2 accesses, 1 for index, 1 for data.





Free Space Management

We need a way to keep track of space currently free. This information is needed when we want to create or add (allocate) to a file. When a file is deleted, we need to show what space is freed up.

BIT VECTOR METHOD

Each block is represented by a bit

1 1 0 0 1 1 0 means blocks 2, 3, 6 are free.

This method allows an easy way of finding contiguous free blocks. Requires the overhead of disk space to hold the bitmap.

A block is not REALLY allocated on the disk unless the bitmap is updated.

What operations (disk requests) are required to create and allocate a file using this implementation?





Free Space Management

FREE LIST METHOD

Free blocks are chained together, each holding a pointer to the next one free.

This is very inefficient since a disk access is required to look at each sector.

GROUPING METHOD

In one free block, put lots of pointers to other free blocks. Include a pointer to the next block of pointers.

COUNTING METHOD

Since many free blocks are contiguous, keep a list of dyads holding the starting address of a "chunk", and the number of blocks in that chunk.

Format < disk address, number of free blocks >





Directory Management

The issue here is how to be able to search for information about a file in a directory given its name.

Could have **linear list** of file names with pointers to the data blocks. This is:

simple to program **BUT** time consuming to search.

Could use hash table - a linear list with hash data structure.

- a) Use the filename to produce a value that's used as entry to hash table.
- b) Hash table contains where in the list the file data is located.
- c) This decreases the directory search time (file creation and deletion are faster.)
- d) Must contend with collisions - where two names hash to the same location.
- e) The number of hashes generally can't be expanded on the fly.





Directory/ File Management

GAINING CONSISTENCY

Required when system crashes or data on the disk may be inconsistent:

Consistency checker - compares data in the directory structure with data blocks on disk and tries to fix inconsistencies. For example, What if a file has a pointer to a block, but the bit map for the free-space-management says that block isn't allocated.

Back-up- provides consistency by copying data to a "safe" place.

Recovery - occurs when lost data is retrieved from backup.

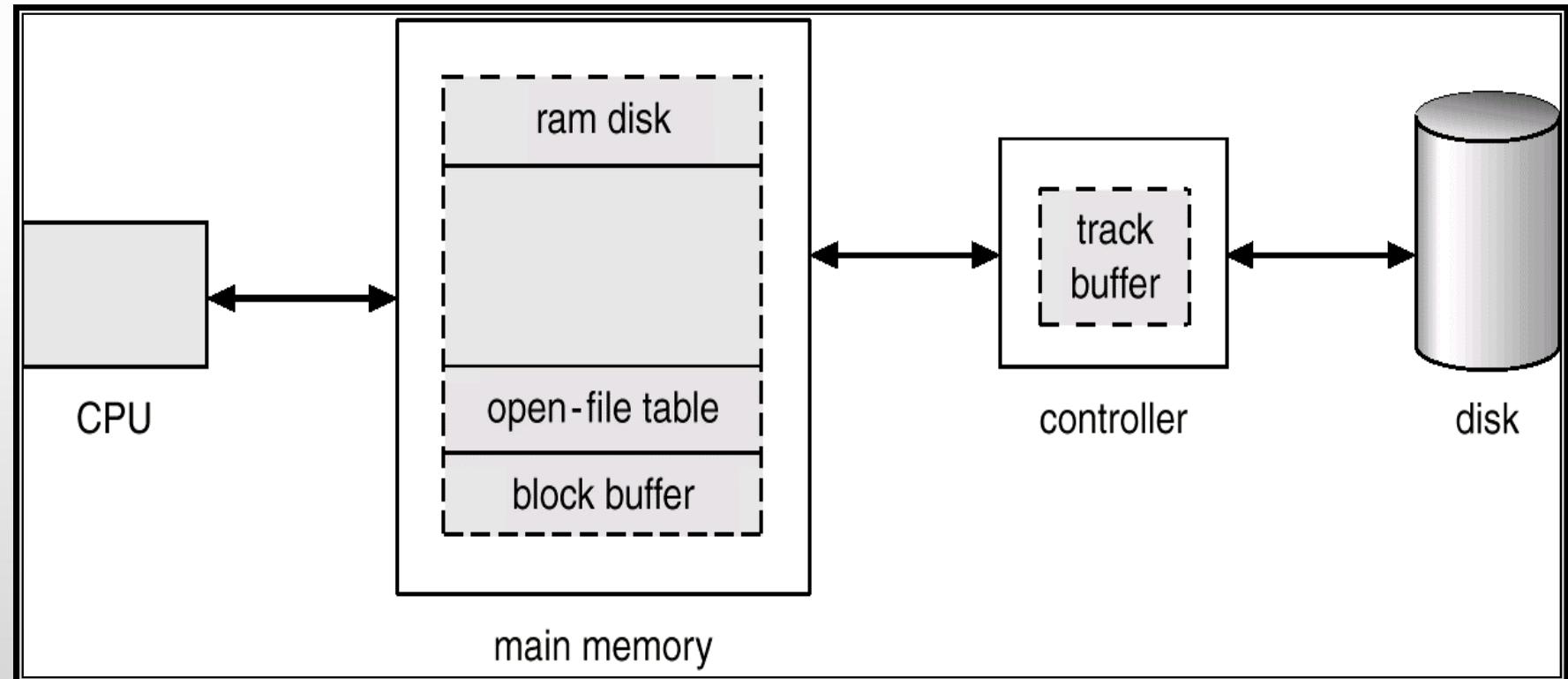




Efficiency and Performance

THE DISK CACHE MECHANISM

There are many places to store disk data so the system doesn't need to get it from the disk again and again.





Efficiency and Performance

THE DISK CACHE MECHANISM

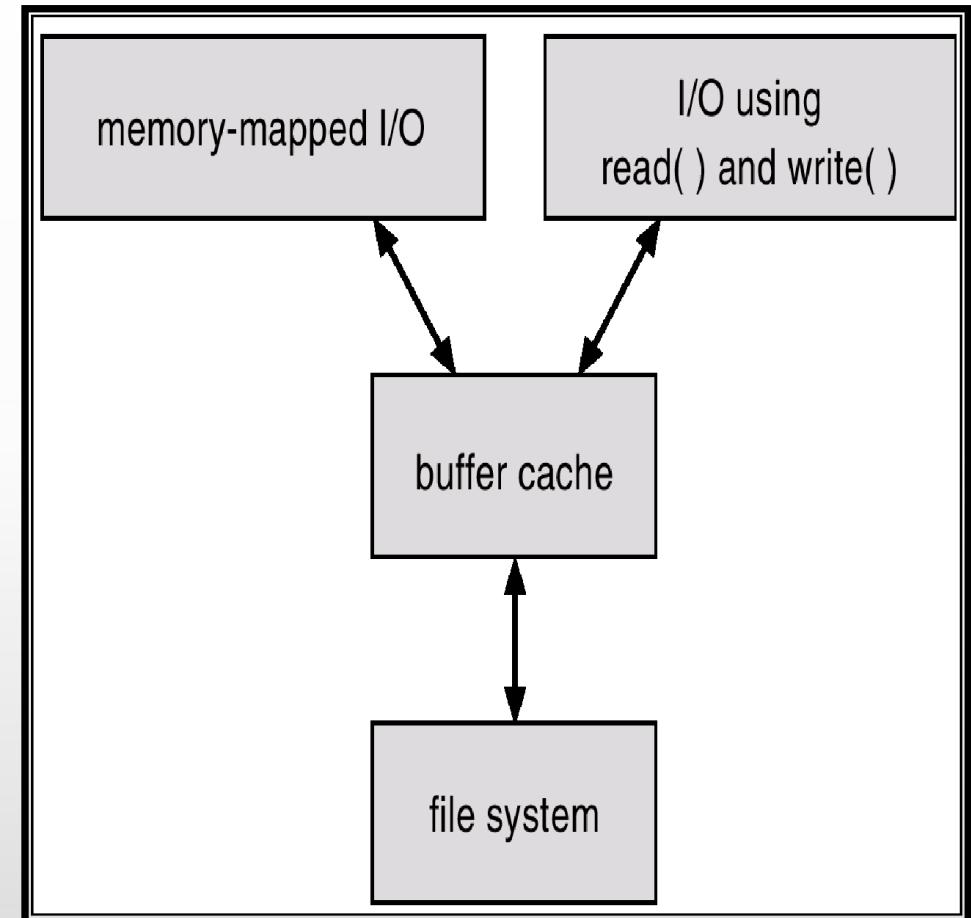
This is an essential part of any well-performing Operating System.

The goal is to ensure that the disk is accessed as seldom as possible.

Keep previously read data in memory so that it might be read again.

They also hold on to written data, hoping to aggregate several writes from a process.

Can also be “smart” and do things like read-ahead. Anticipate what will be needed.





Thank You Very Much

Benjamin Kommey

bkommey.coe@knust.edu.gh

nii_kommey@msn.com

050 770 32 86

Skype_id: calculus.affairs





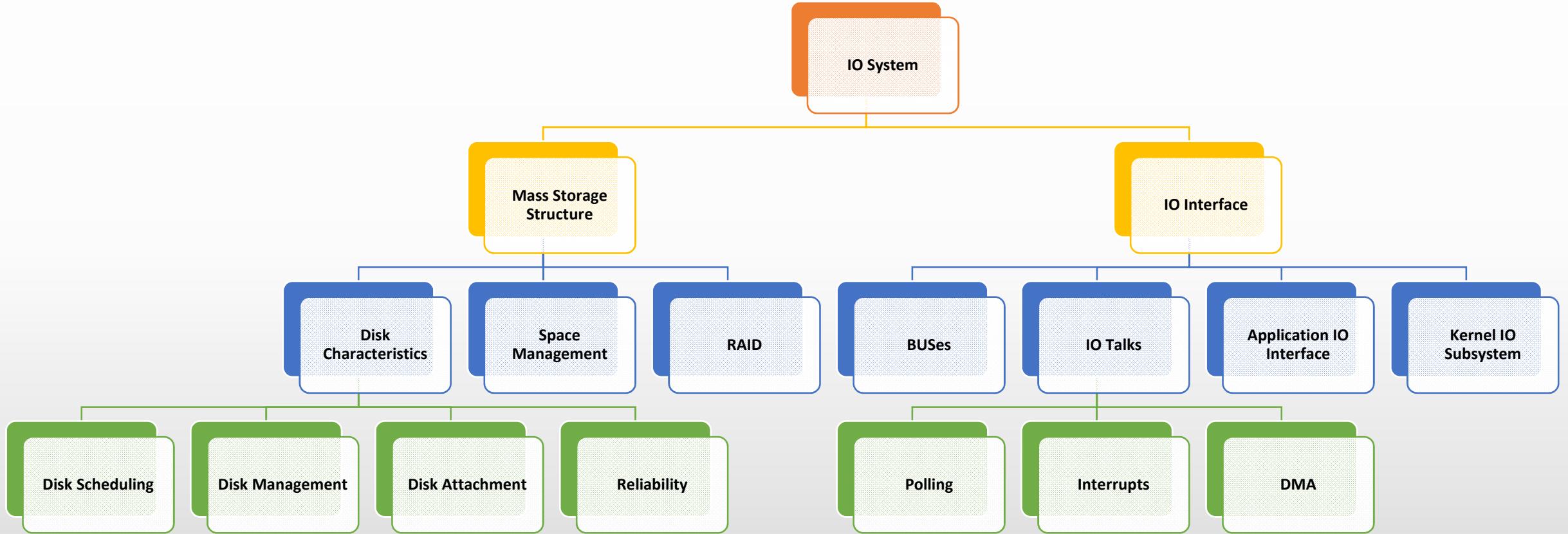
“The Internet is becoming the town square for the global village of tomorrow.”

Bill Gates





Overview





Mass Storage Structure

Disk Characteristics

A disk can be viewed as an array of blocks.

In fact, a file system will want to view it at that logical level.

However, there's a mapping scheme from logical block address B, to physical address (represented by a track / sector pair.)

The smallest storage allocation is a block - nothing smaller can be placed on the disk.

This results in unused space (internal fragmentation) on the disk, since quite often the data being placed on the disk doesn't need a whole block.



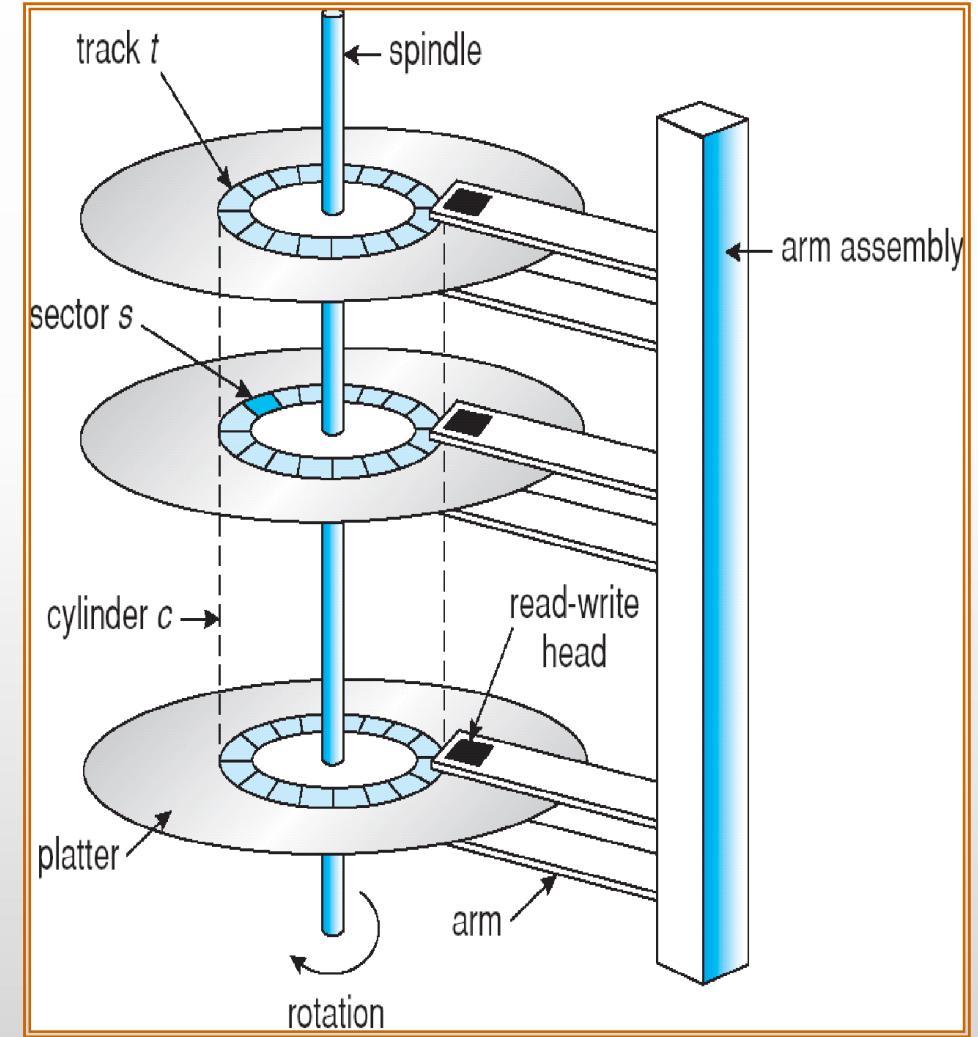


Disk Scheduling

The components making up disk service time include:

$$\text{Total time} = \text{setup} + \text{seek} + \text{rotation time} + \text{transfer} + \text{wrap-up}$$

The methods discussed below try to optimize seek time but make no attempt to account for the total time. The ideal method would optimize the total time and many controllers are now able to accomplish this.





Disk Management

Disk formatting

- Creates a logical disk from the raw disk. Includes setting aside chunks of the disk for booting, bad blocks, etc.
Also provides information needed by the driver to understand its positioning

Boot block

- That location on the disk that is accessed when trying to boot the operating system. It's a well-known location that contains the code that understands how to get at the operating system - generally this code has a rudimentary knowledge of the file system

Bad blocks

- The driver knows how to compensate for a bad block on the disk. It does this by putting a pointer, at the location of the bad block, indicating where a good copy of the data can be found.

Swap Space Management

- The Operating System requires a contiguous space where it knows that disk blocks have been reserved for paging. This space is needed because a program can't be given unshared memory unless there's a backing store location for that memory





Disk Attachment

Host-attached storage

accessed through I/O ports talking to I/O busses

SCSI itself is a bus, up to 16 devices on one cable, **SCSI initiator** requests operation and **SCSI targets** perform tasks

Each target can have up to 8 **logical units** (disks attached to device controller)

Fibre Channel (FC) is high-speed serial architecture

Can be switched fabric with 24-bit address space – the basis of **storage area networks (SANs)** in which many hosts attach to many storage units

Serial ATA (SATA) – current standard for home & medium sized servers.

Advantage is low cost with “reasonable” performance.





Disk Attachment

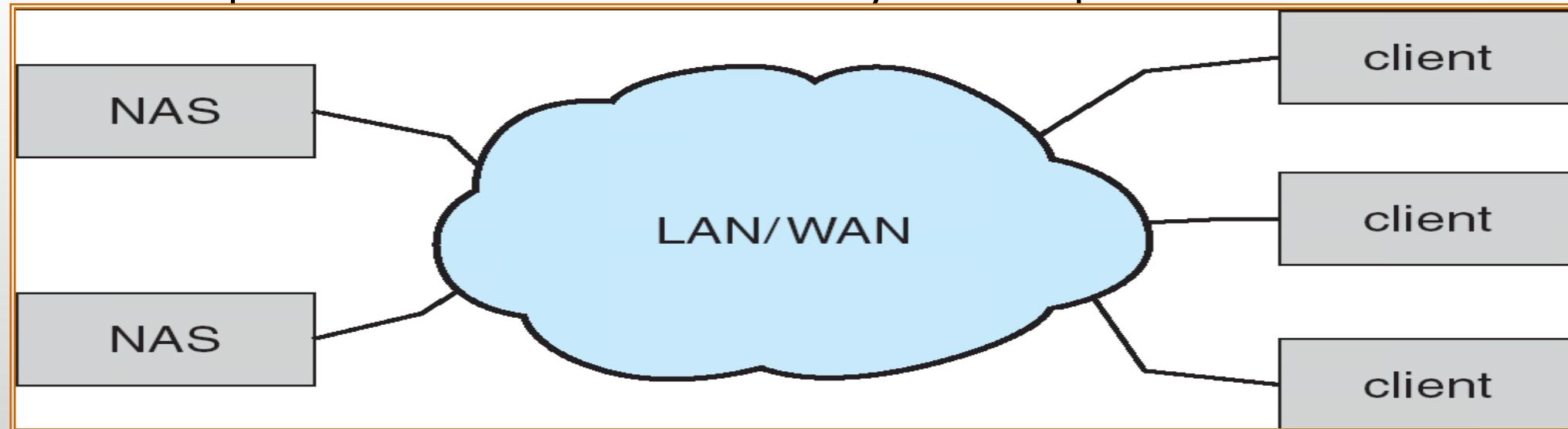
Network-attached storage

Network-attached storage (**NAS**) is storage made available over a network rather than over a local connection (such as a bus)

NFS and CIFS are common protocols

Implemented via remote procedure calls (RPCs) between host and storage

New iSCSI protocol uses IP network to carry the SCSI protocol



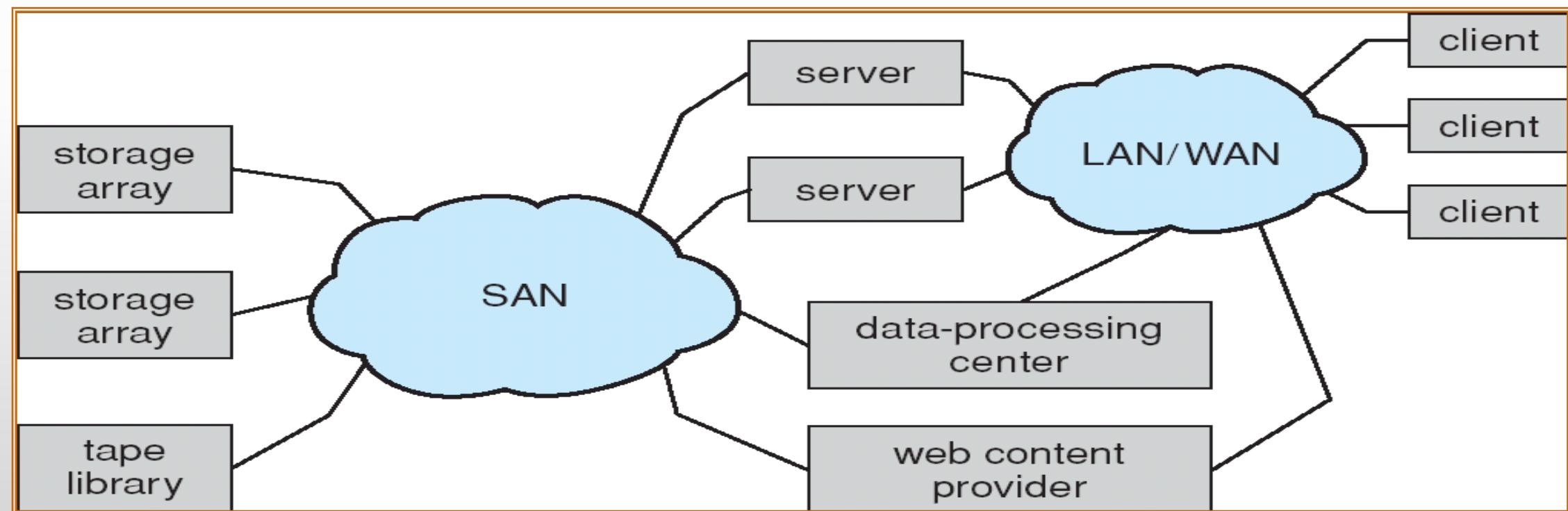


Disk Attachment

Storage-Area Network

Common in large storage environments (and becoming more common)

Multiple hosts attached to multiple storage arrays - flexible



Reliability

MIRRORING

One way to increase reliability is to "mirror" data on a disk. Every piece of data is maintained on two disks - disk drivers must be capable of getting data from either disk. Performance issues: a read is faster since data can be obtained from either disk - writes are slower since the data must be put on both disks.

RAID

Redundant Array of Inexpensive Disks: Rather than maintain two copies of the data, maintain one copy plus parity. For example, four disks contain data, and a fifth disk holds the parity of the XOR of the four data disks. Reads slower than mirroring, writes much slower. But RAID is considerably CHEAPER than mirroring.

DISK STRIPING

Disk tend to be accessed unevenly - programs ask for a number of blocks from the same file, for instance. Accesses can be distributed more evenly by spreading a file out over several disks. This works well with RAID. Thus block 0 is on disk 0, block 1 is on disk 1, block 4 is on disk 0.

Consider how to recover from a failure on these architectures.





Redundant Array Inexpensive Disk RAID

These are the various levels of RAID.

The reliability increases with higher levels.

In practice, only levels 0, 1, 5 and 10 are typically used.

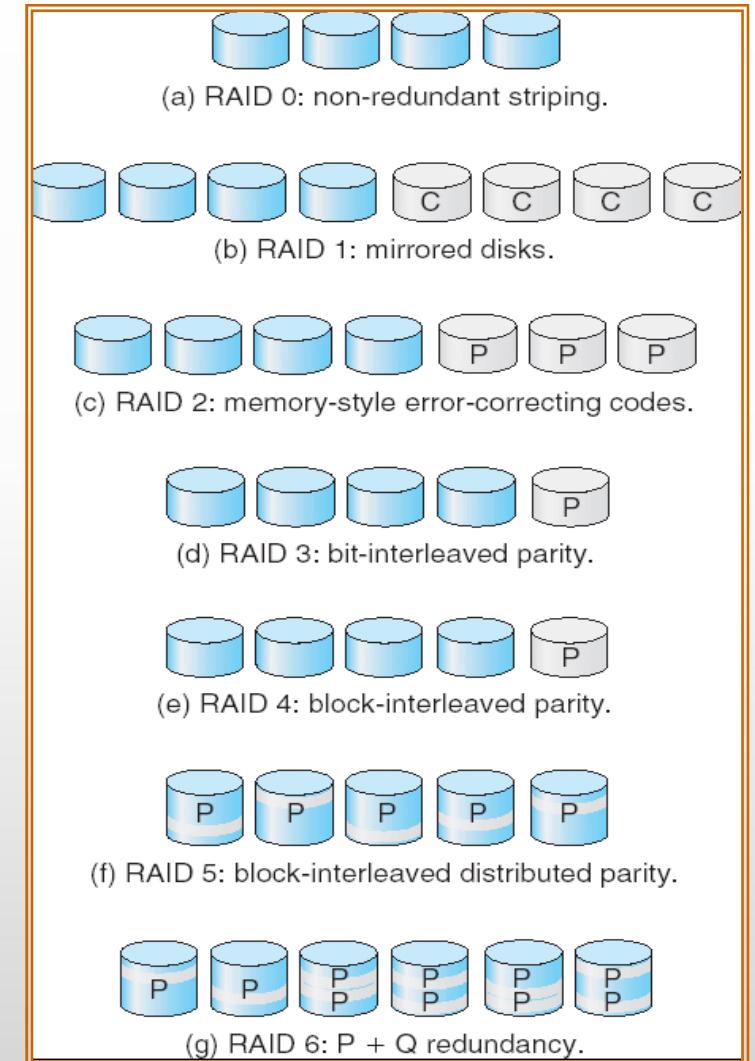
Several improvements in disk-use techniques involve the use of multiple disks working cooperatively.

Disk striping uses a group of disks as one storage unit.

RAID schemes improve performance and improve the reliability of the storage system by storing redundant data.

Mirroring or shadowing keeps duplicate of each disk.

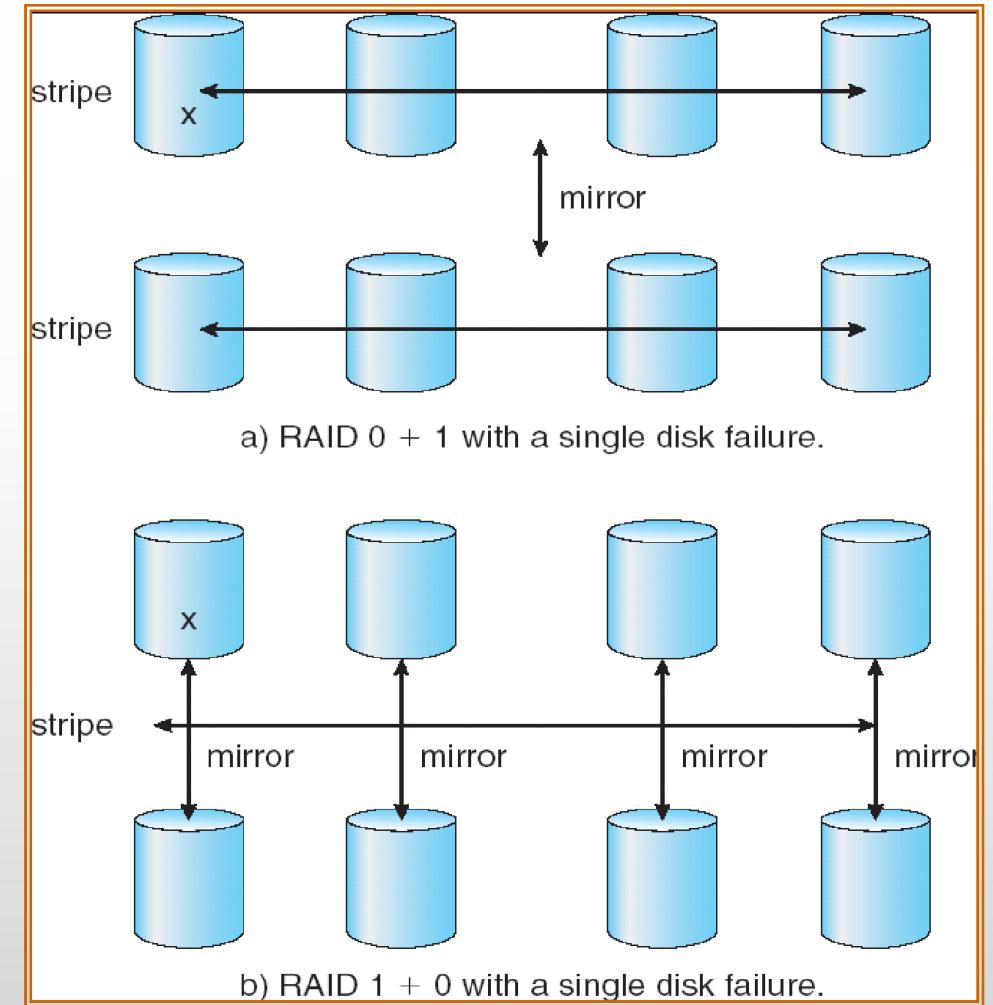
Block interleaved parity uses much less redundancy.





RAID

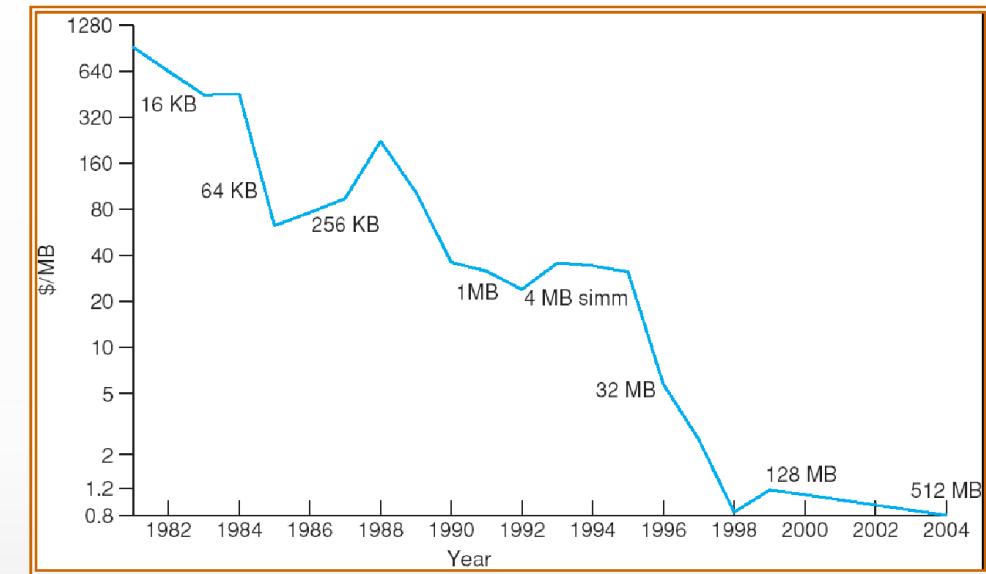
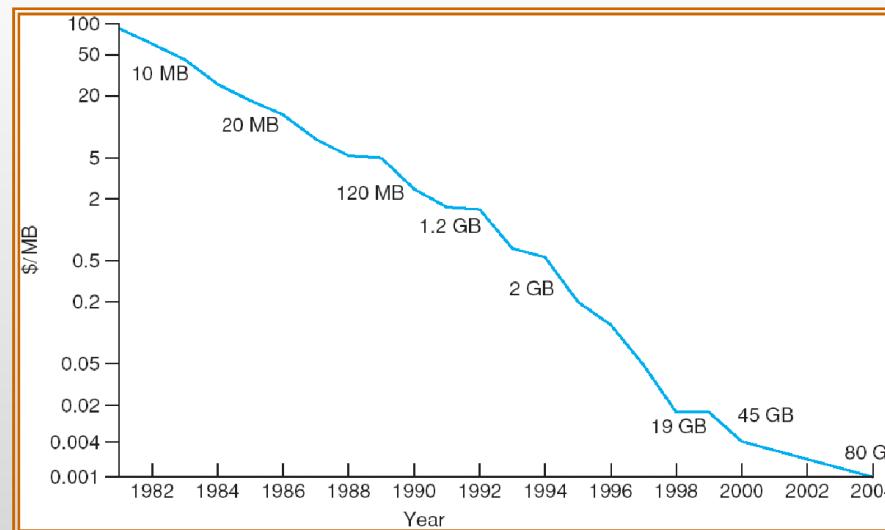
RAID 10 becoming more and more popular.





Storage Prices

Price per Megabyte of DRAM, From 1981 to 2004



Price per Megabyte of Hard Disk, From 1981 to 2004



IO Hardware

Incredible variety of I/O devices

Common concepts

Port

Bus (daisy chain or shared direct access)

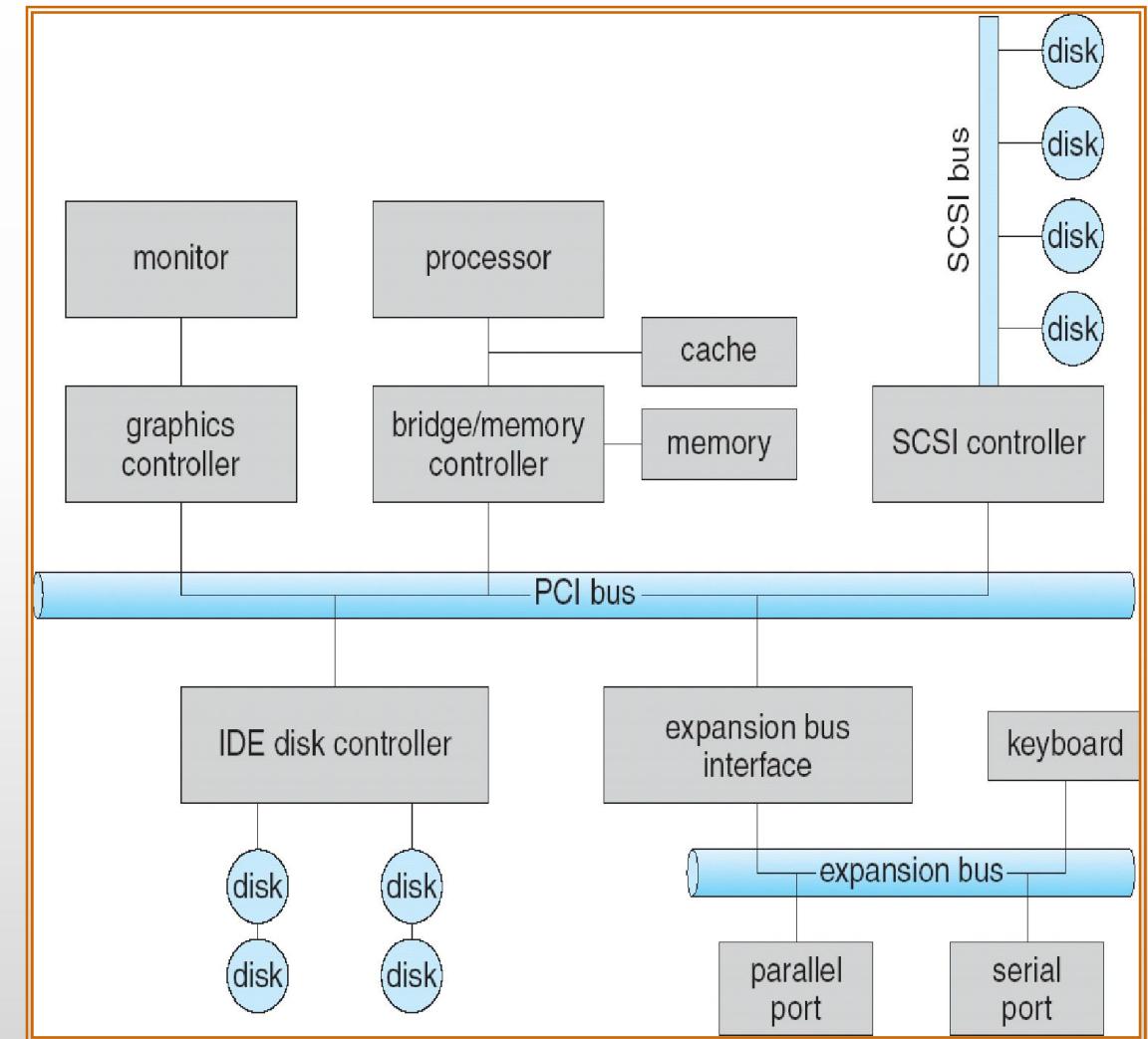
Controller (host adapter)

I/O instructions control devices

Devices have addresses, used by

Direct I/O instructions

Memory-mapped I/O



IO Hardware

Memory Mapped IO:

Works by associating a memory address with a device and a function on that device.

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)





Polling and Interrupts

CPU Interrupt request line triggered by I/O device

Interrupt handler receives interrupts

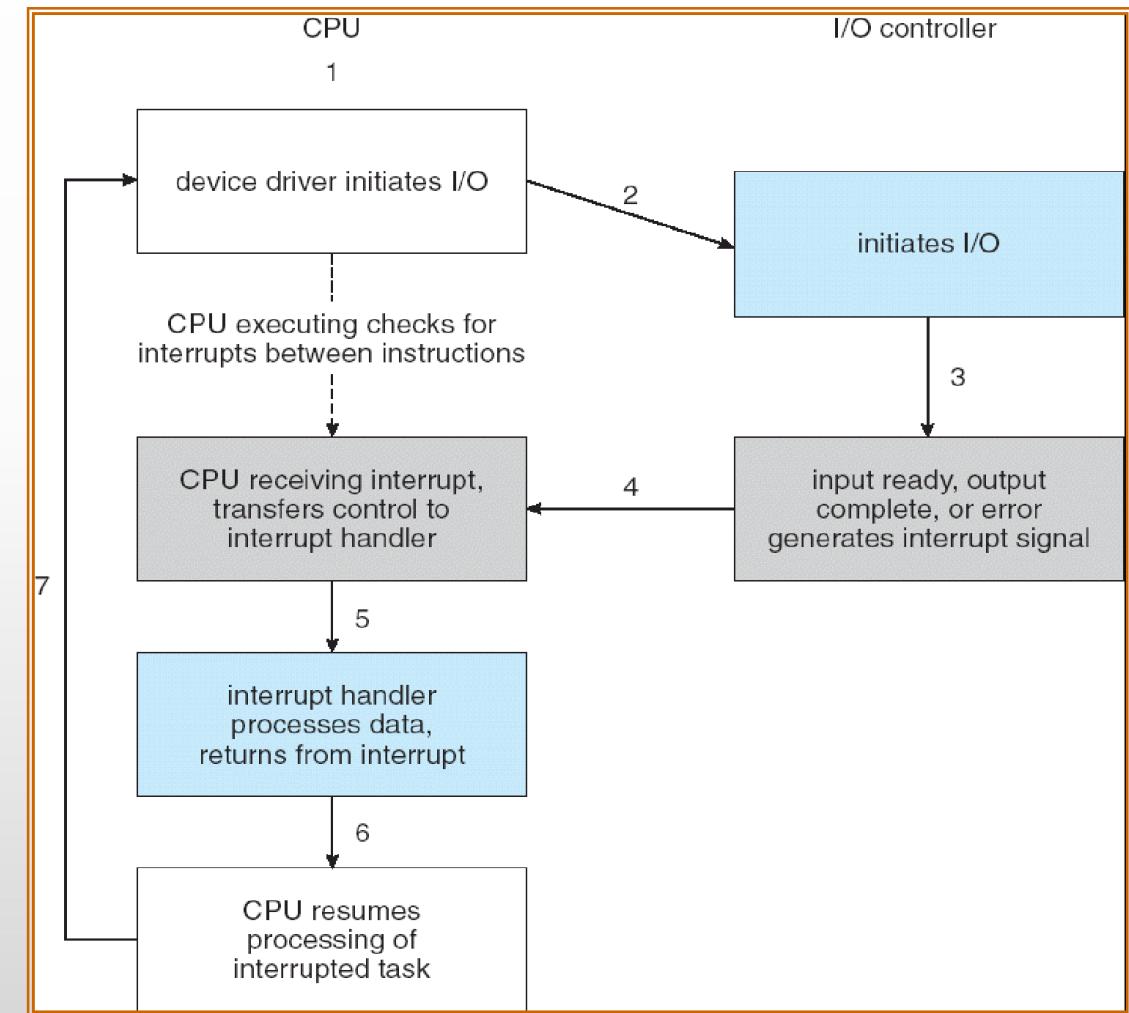
Maskable to ignore or delay some interrupts

Interrupt vector to dispatch interrupt to correct handler

Based on priority

Some unmaskable

Interrupt mechanism also used for exceptions.



Polling and Interrupts

When you get an interrupt, you need to be able to figure out the device that gave you the interrupt.

These are the interrupt vectors for an Intel Processor.

Notice that most of these are actually exceptions.

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19-31	(Intel reserved, do not use)
32-255	maskable interrupts

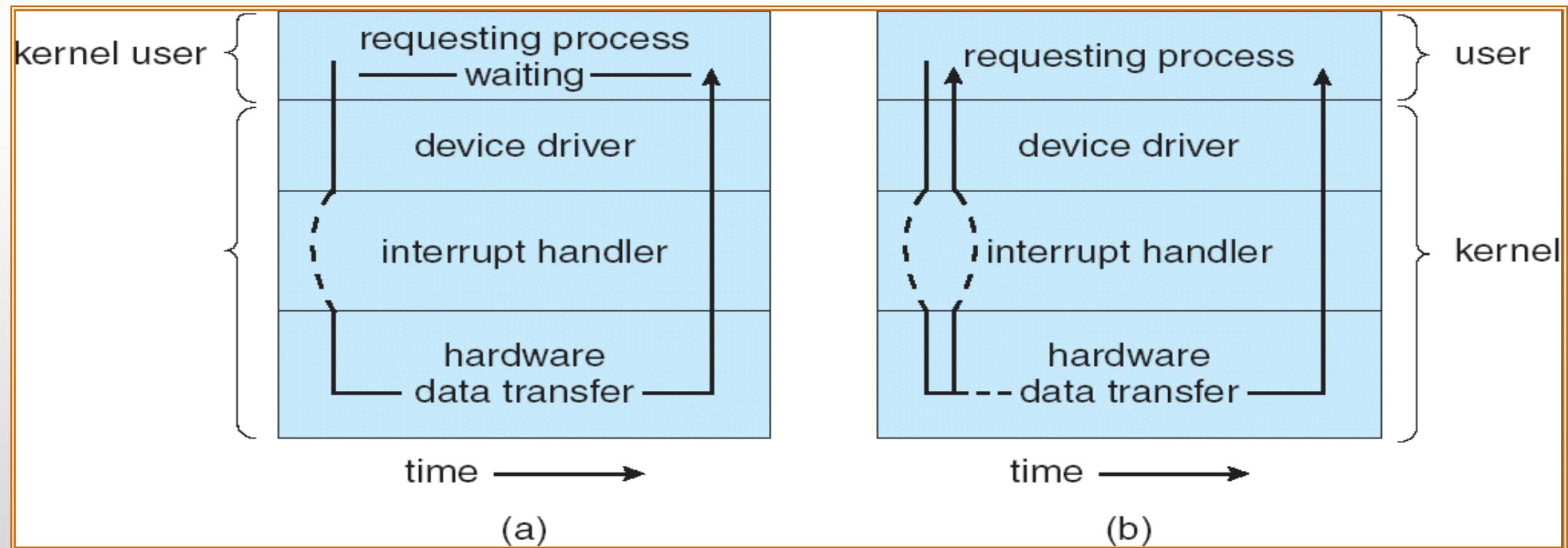




Synchronous or Asynchronous

Synchronous does the whole job – all at one time – data is obtained from the device by the processor.

Asynchronous has the device and the processor acting in time independent of each other.



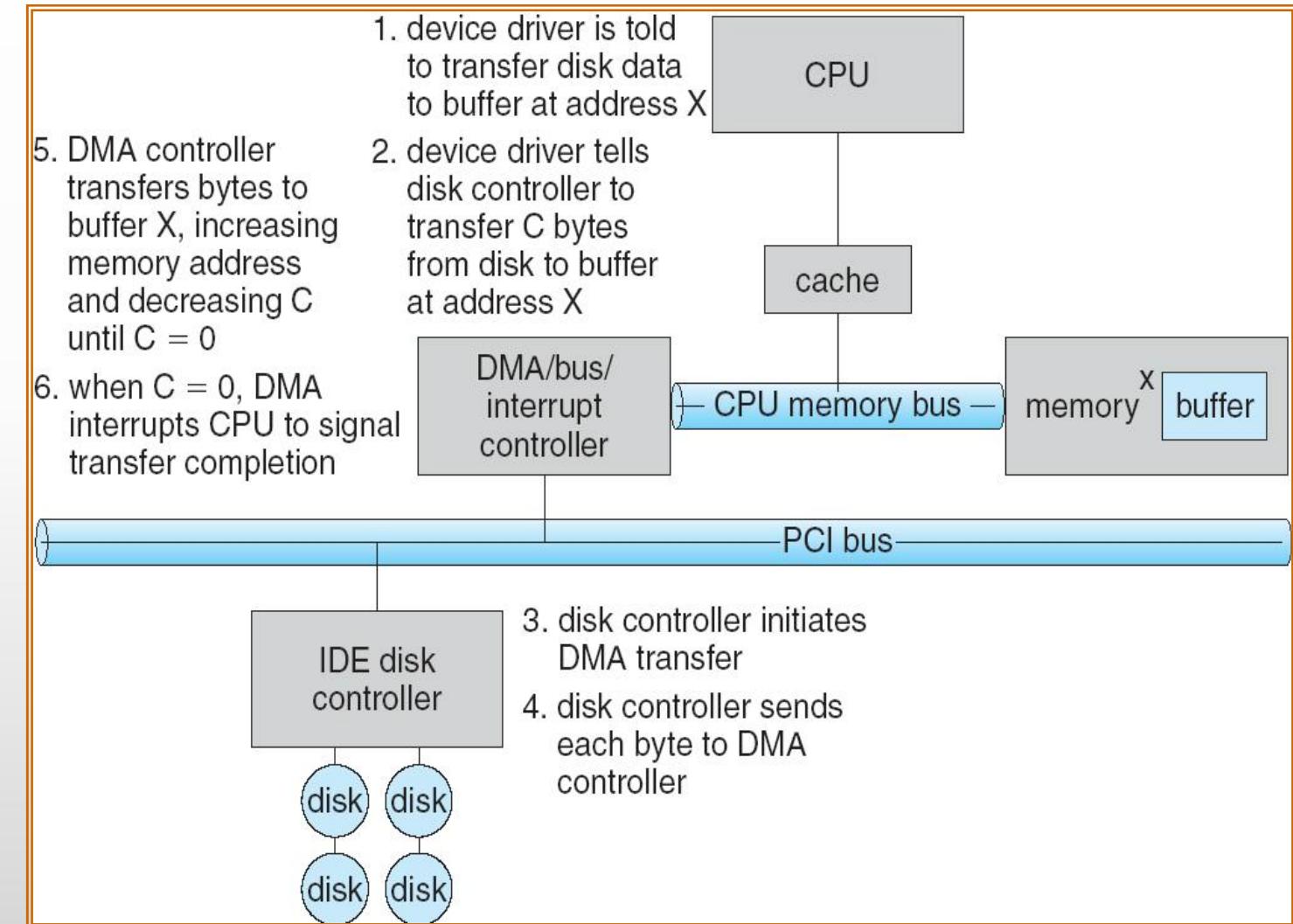


Used to avoid programmed I/O for large data movement

Requires DMA controller

Bypasses CPU to transfer data directly between I/O device and memory

DMA



Streams

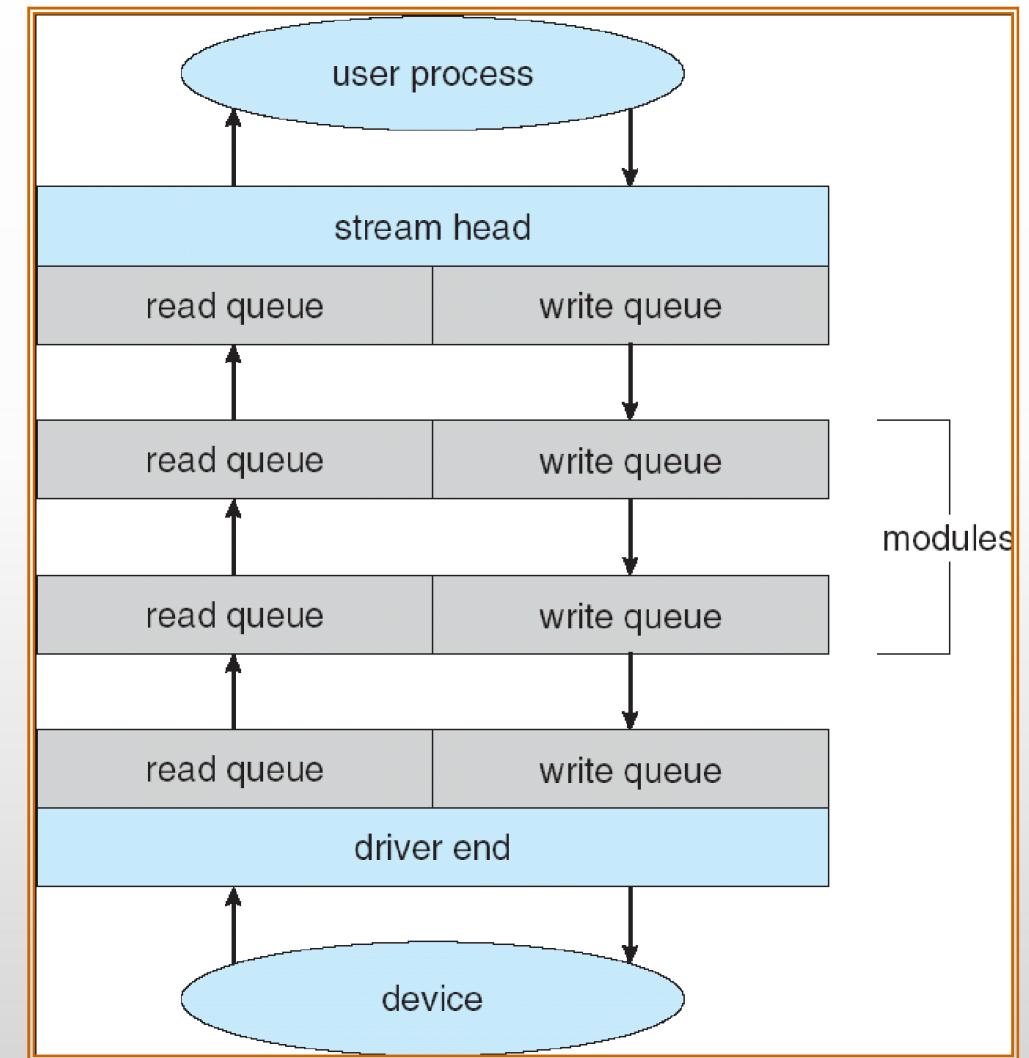
STREAM – a full-duplex communication channel between a user-level process and a device in Unix System V and beyond

A STREAM consists of:

- STREAM head interfaces
- driver end interfaces with the device
- zero or more STREAM modules between them.

Each module contains a **read queue** and a **write queue**

Message passing is used to communicate between queues



Interfaces

Block and Character Devices

Typical for disks – use `read()`, `write()`, `seek()` sequence.

Network Devices

Clocks and Timers

The OS uses an incredible number of clock calls – many events are timestamped within the OS

Blocking and Non-Blocking IO

Non- Blocking: Some devices are started by the OS, and then proceed on without further OS intervention. The delay timer in our project works this way.

Blocking: Any Read-Device will be blocking since the program can't proceed until it gets the information it wanted from the device.





Kernel IO Subsystem

Buffering

Used to interface between devices of different speeds (modem and disk for instance.)

Interface between operations having different data sizes.

(small network packets as part of a bigger transfer.)

Users often read or write small number of bytes – but the disk wants 4096 bytes.

The file system maintains this buffer.

Spooling

Kernel data structures – what needs to be maintained to

Support the device

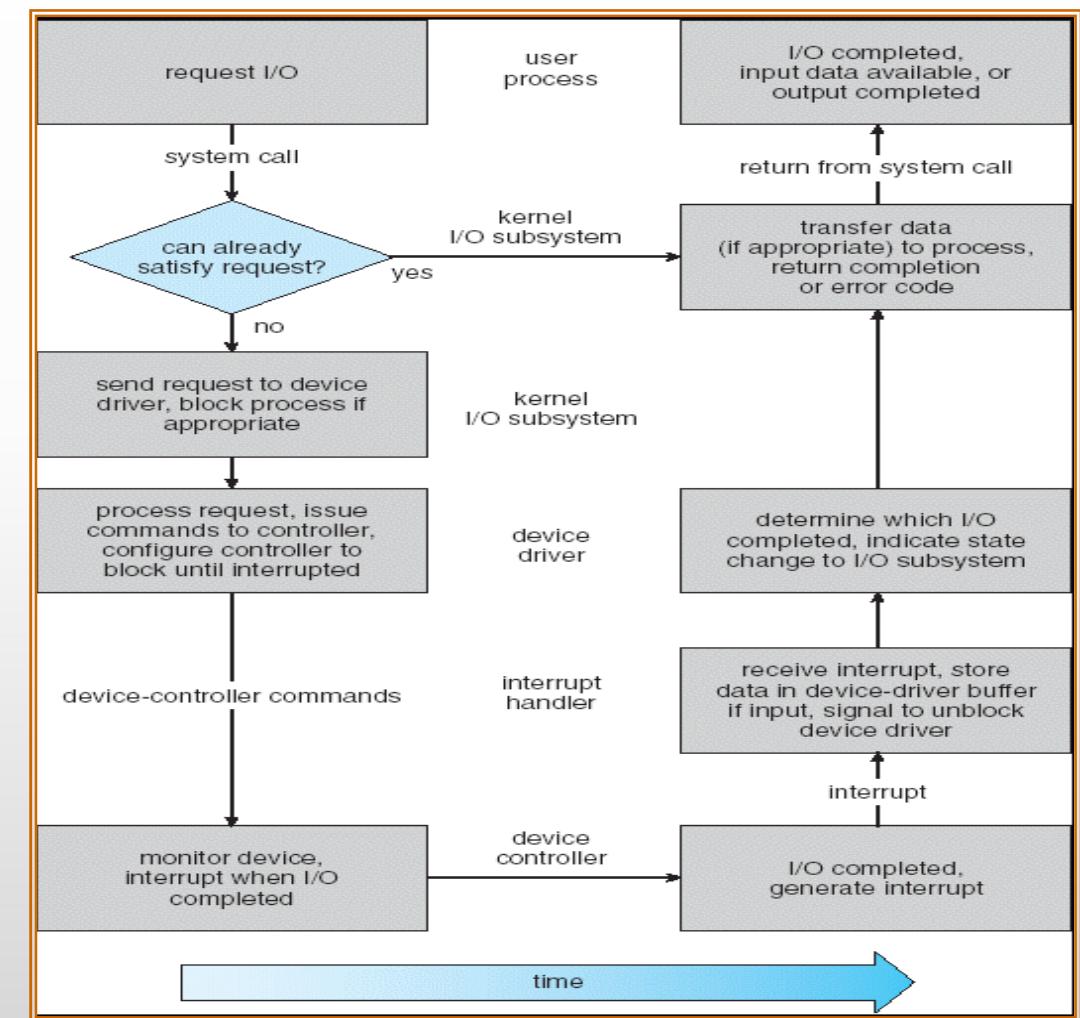
Support an instance of opening the device.





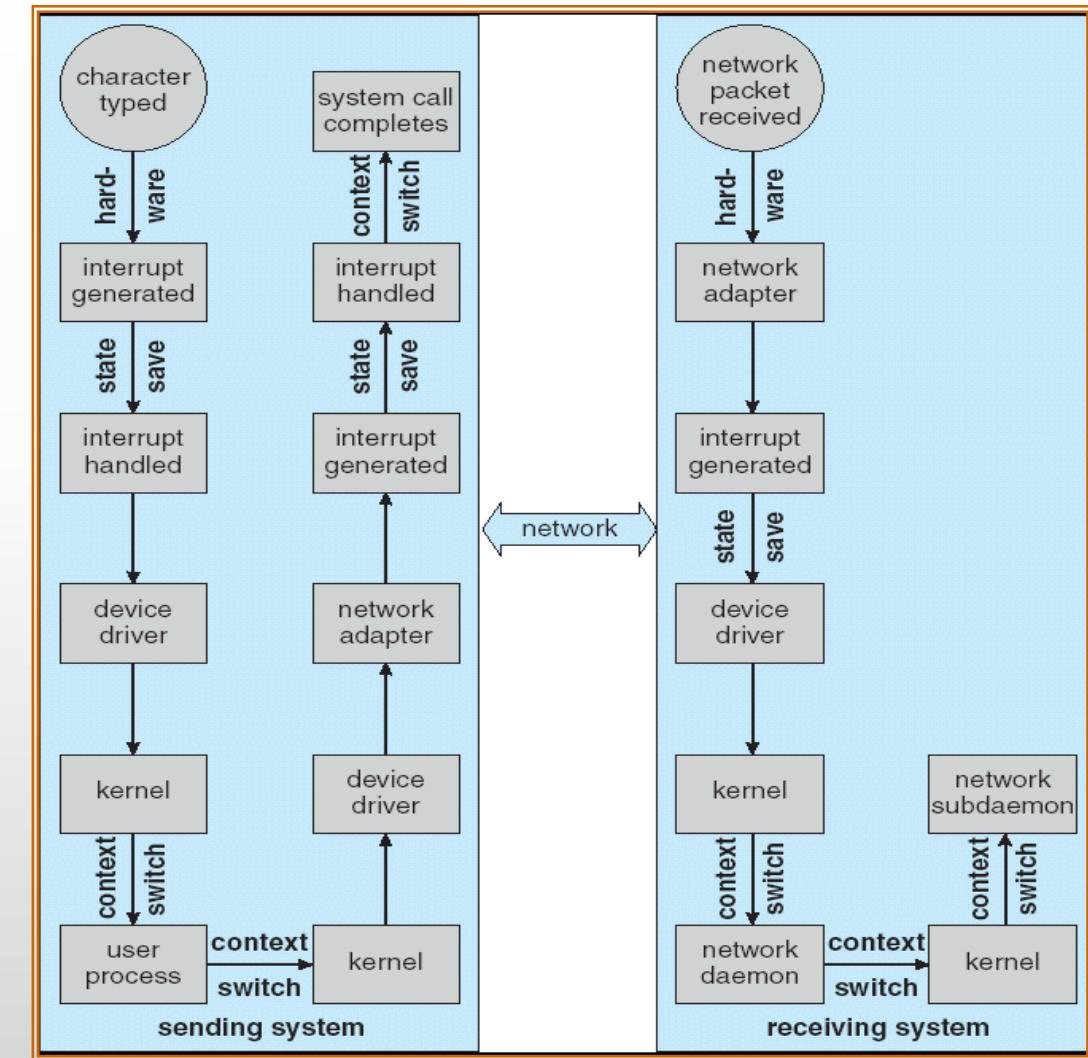
Kernel IO Subsystem

The steps in an IO request.



Performance

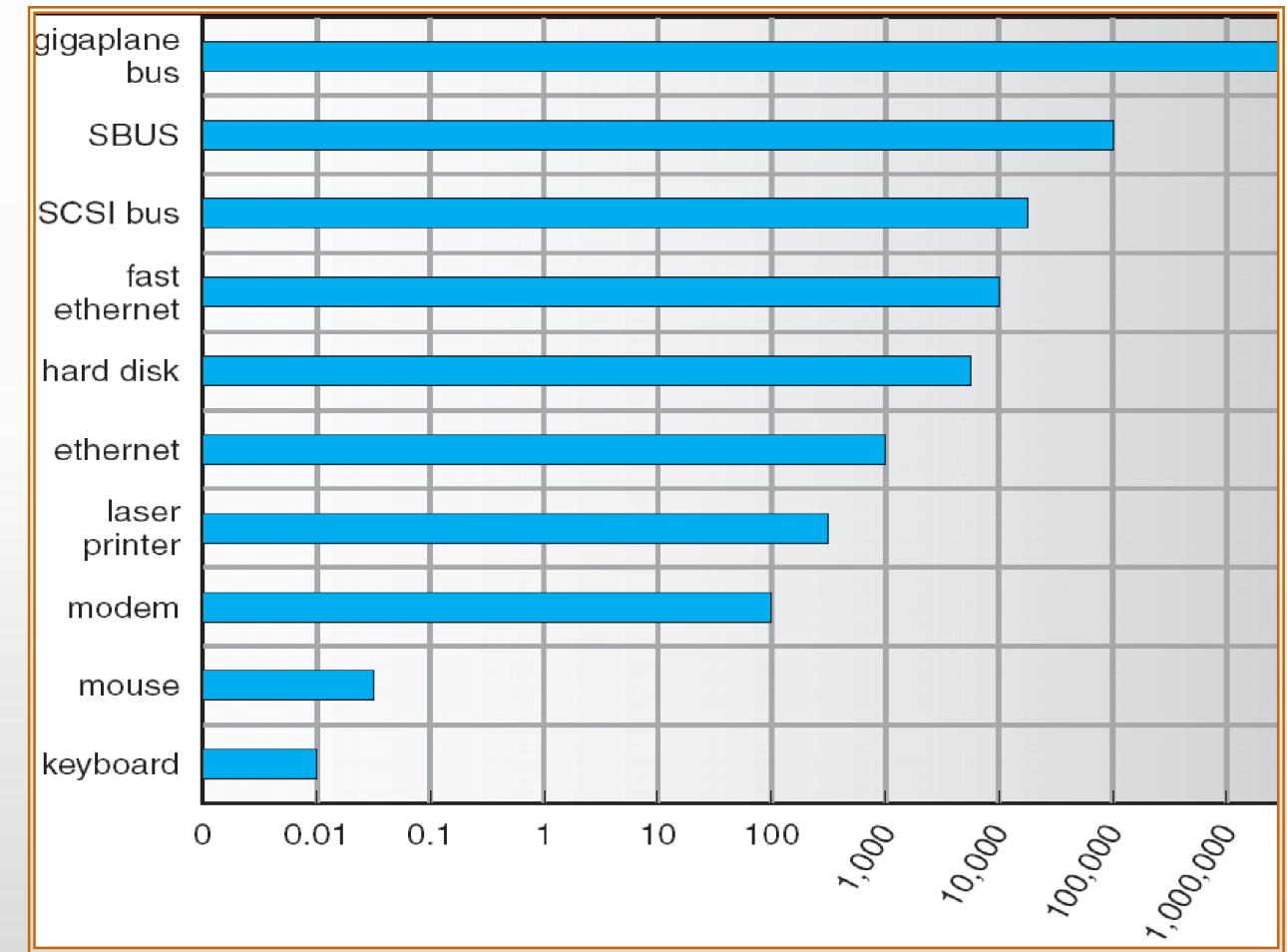
The steps required to handle a single keystroke across the network.





Performance

Throughput for various devices.





Thank You Very Much

Benjamin Kommey

bkommey.coe@knust.edu.gh

nii_kommey@msn.com

050 770 32 86

Skype_id: calculus.affairs





“The first rule of any technology used in a business is that automation applied to an efficient operation will magnify the efficiency.

The second is that automation applied to an inefficient operation will magnify the inefficiency.”

Bill Gates





Overview

System Characteristics

Features of Real-Time Systems

Implementing Real-Time Operating Systems

Real-Time CPU Scheduling

VxWorks 5.x





Objectives

- To explain the timing requirements of real-time systems
- To distinguish between hard and soft real-time systems
- To discuss the defining characteristics of real-time systems
- To describe scheduling algorithms for hard real-time systems





Overview of Real-Time Systems

A **real-time system** requires that results be produced within a specified deadline period.

An **embedded system** is a computing device that is part of a larger system (I.e. automobile, airliner.)

A **safety-critical system** is a real-time system with catastrophic results in case of failure.

A hard real-time system guarantees that real-time tasks be completed within their required deadlines.

A **soft real-time system** provides priority of real-time tasks over non real-time tasks





System Characteristics

Single purpose

Small size

Inexpensively mass-produced

Specific timing requirements





System-on-a-Chip

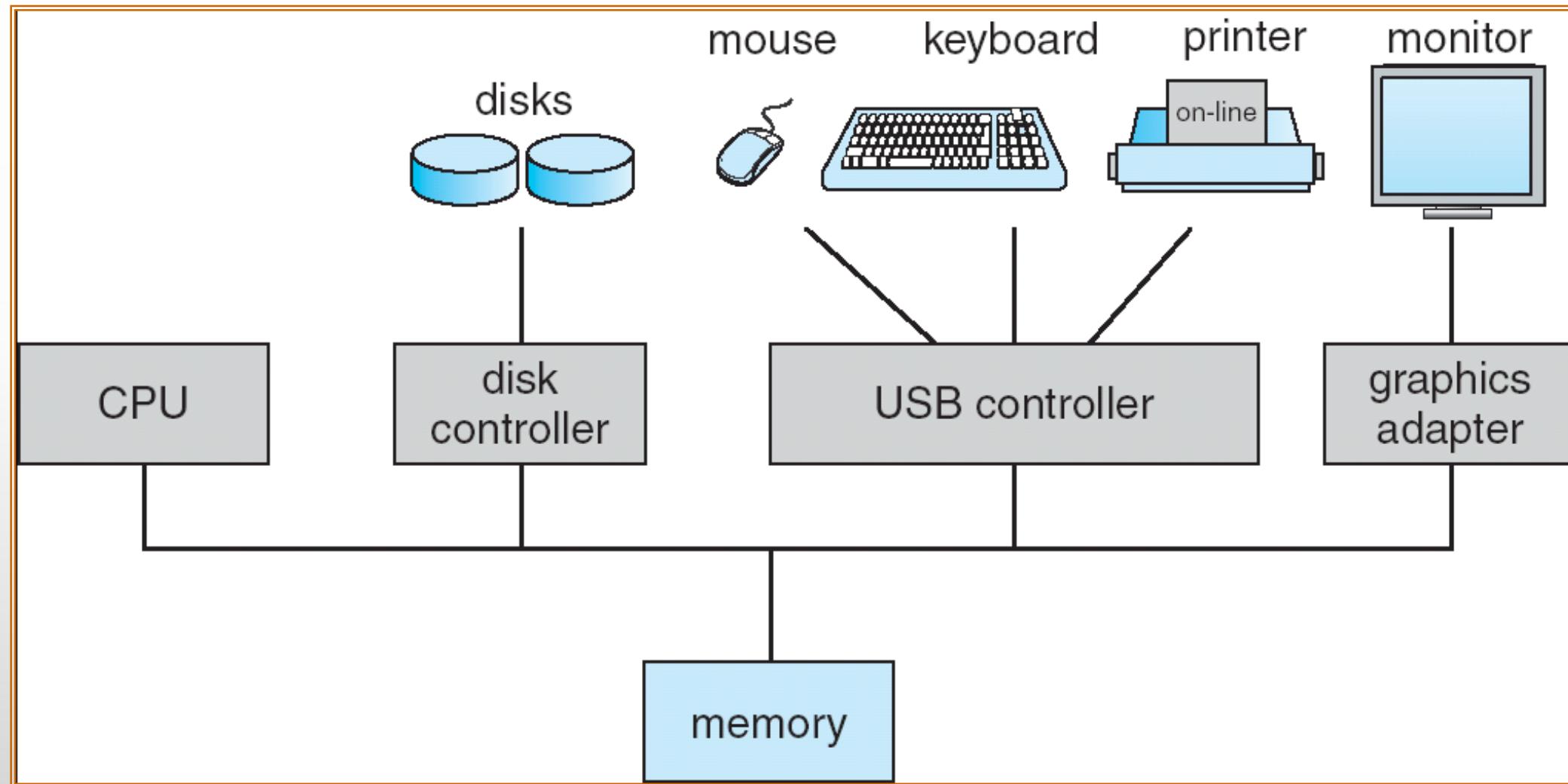
Many real-time systems are designed using system-on-a-chip (SOC) strategy.

SOC allows the CPU, memory, memory-management unit, and attached peripheral ports (I.e. USB) to be contained in a single integrated circuit





Bus-Oriented System





Features of Real-Time Kernels

Most real-time systems do not provide the features found in a standard desktop system.

Reasons include

Real-time systems are typically single-purpose.

Real-time systems often do not require interfacing with a user.

Features found in a desktop PC require more substantial hardware than what is typically available in a real-time system





Virtual Memory in Real-Time Systems

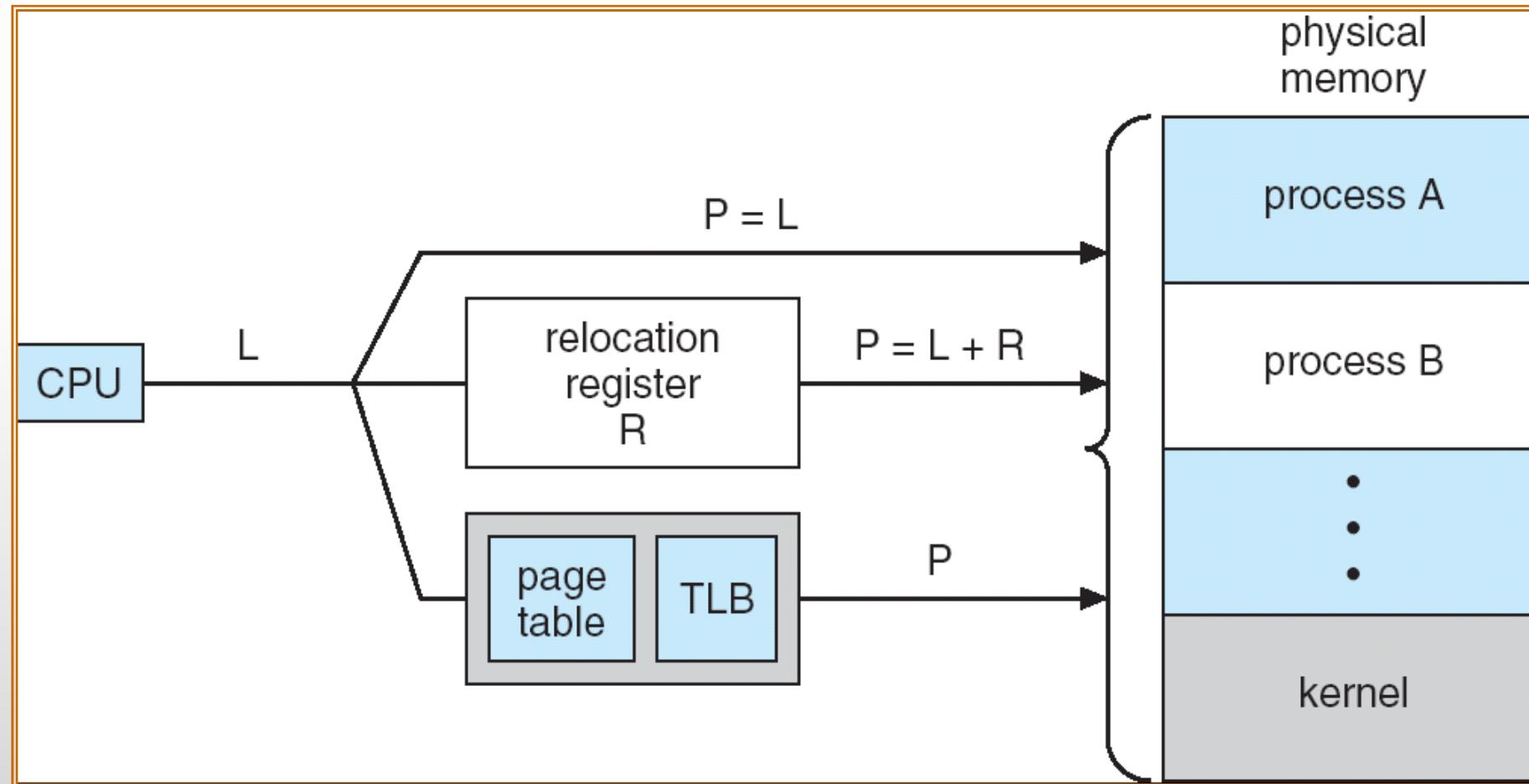
Address translation may occur via:

- (1) **Real-addressing mode** where programs generate actual addresses.
- (2) **Relocation register mode.**
- (3) Implementing full **virtual memory**





Address Translation





Implementing Real-Time Operating Systems

In general, real-time operating systems must provide:

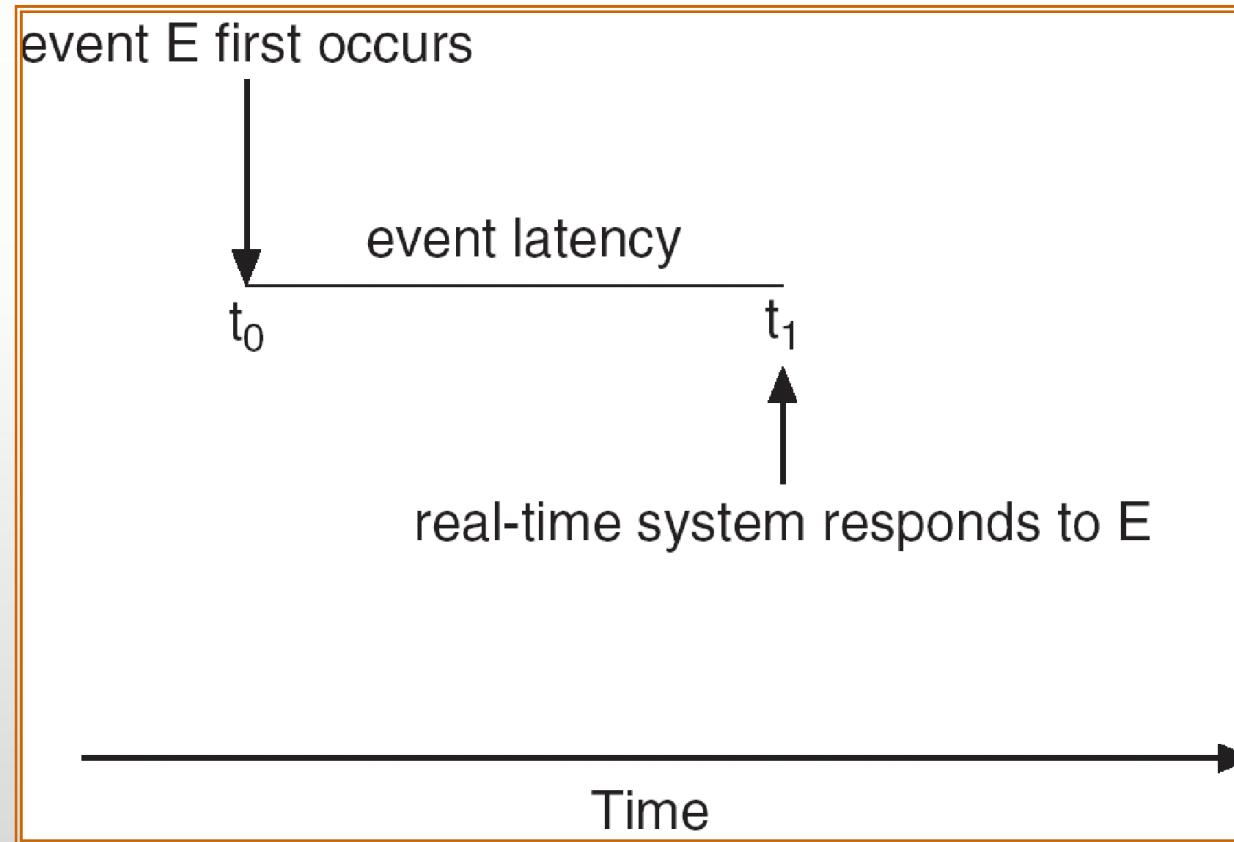
- (1) Preemptive, priority-based scheduling
- (2) Preemptive kernels
- (3) Latency must be minimized





Minimizing Latency

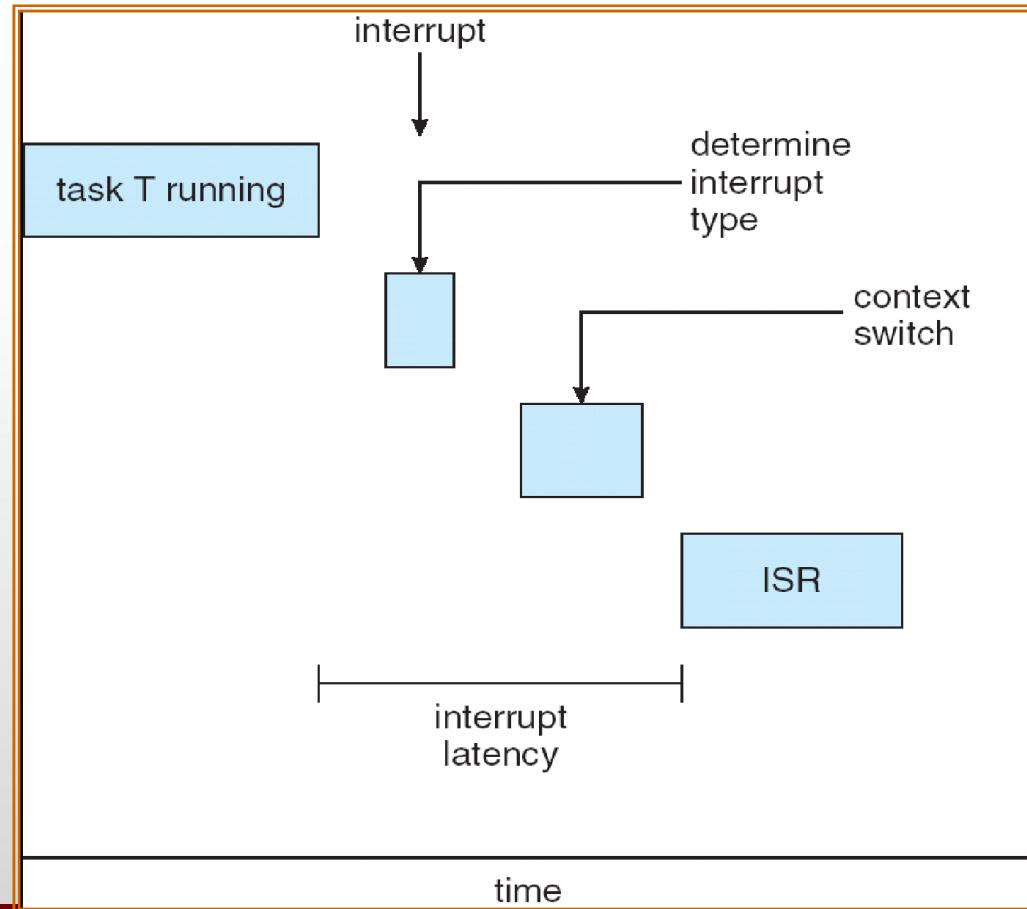
Event latency is the amount of time from when an event occurs to when it is serviced.





Interrupt Latency

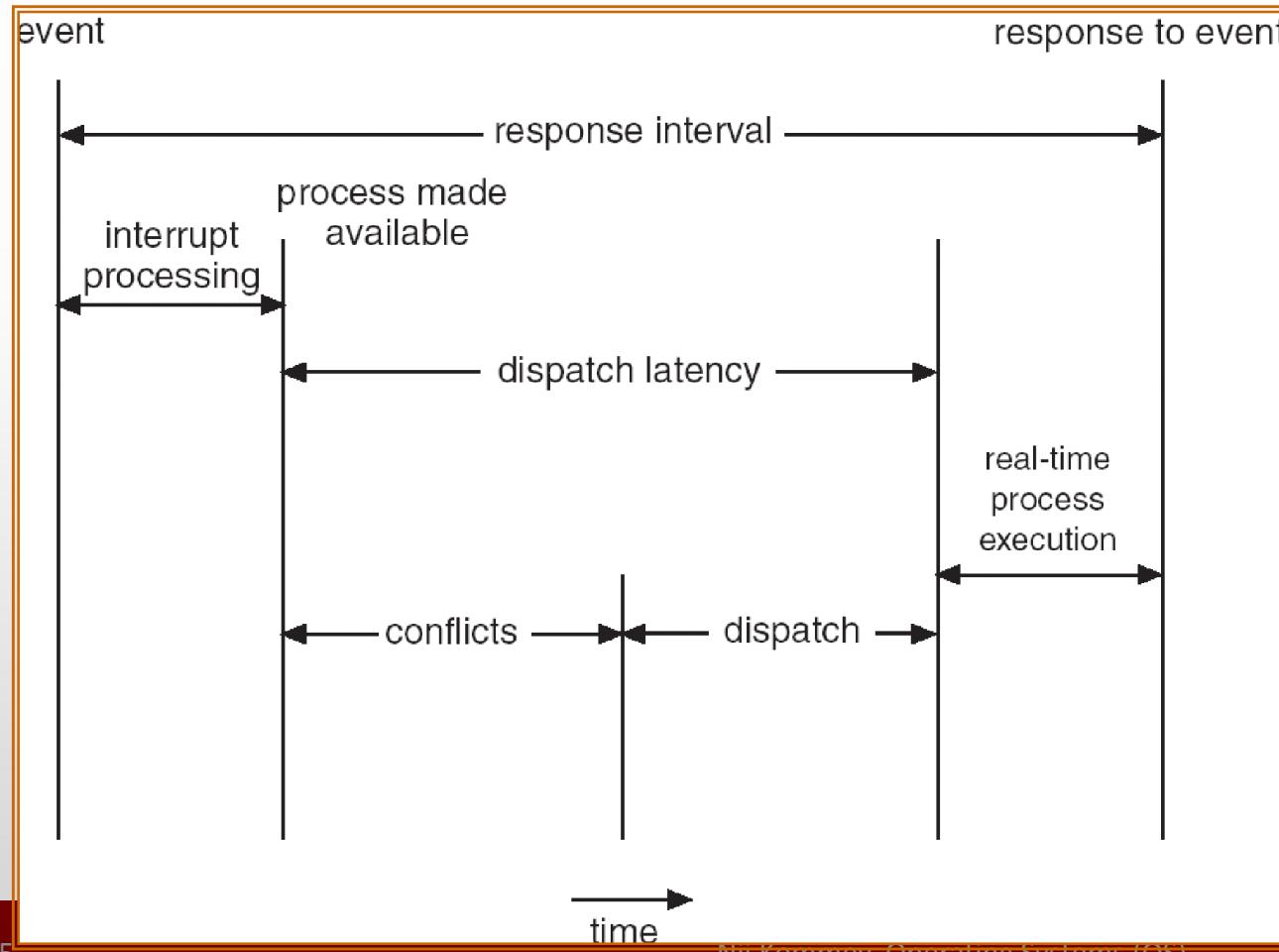
Interrupt latency is the period of time from when an interrupt arrives at the CPU to when it is serviced.





Dispatch Latency

Dispatch latency is the amount of time required for the scheduler to stop one process and start another.





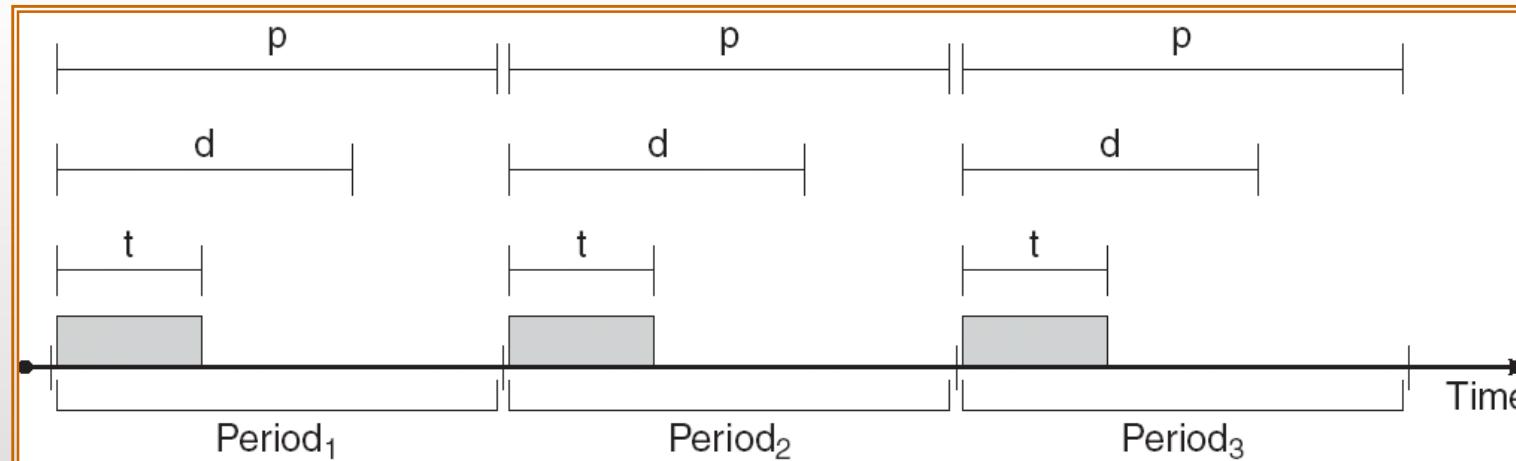
Real-Time CPU Scheduling

Periodic processes require the CPU at specified intervals
(periods)

p is the duration of the period

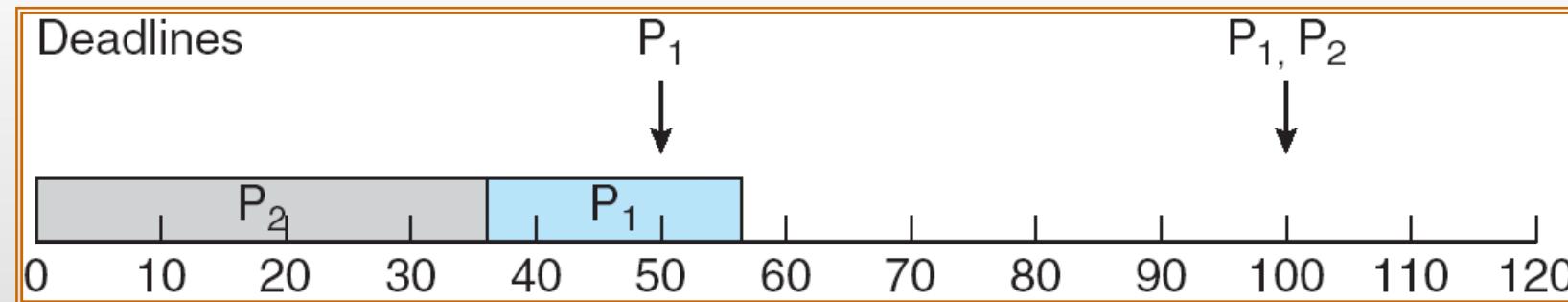
d is the deadline by when the process must be serviced

t is the processing time





Scheduling of tasks when P_2 has a higher priority than P_1





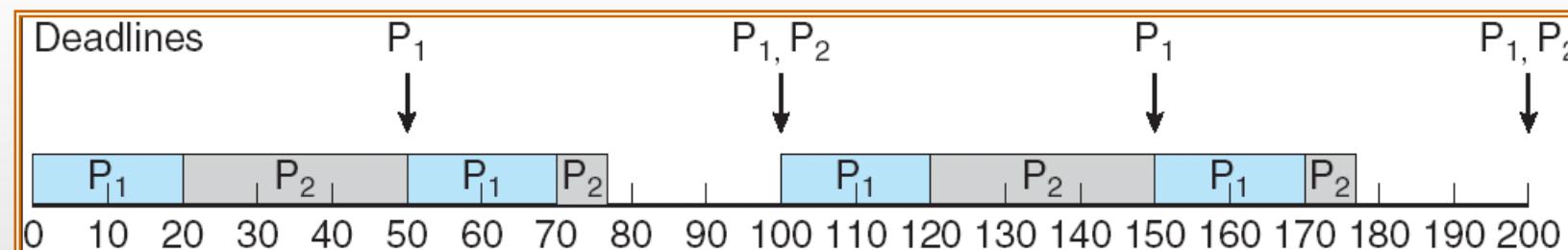
Rate Montonic Scheduling

A priority is assigned based on the inverse of its period

Shorter periods = higher priority;

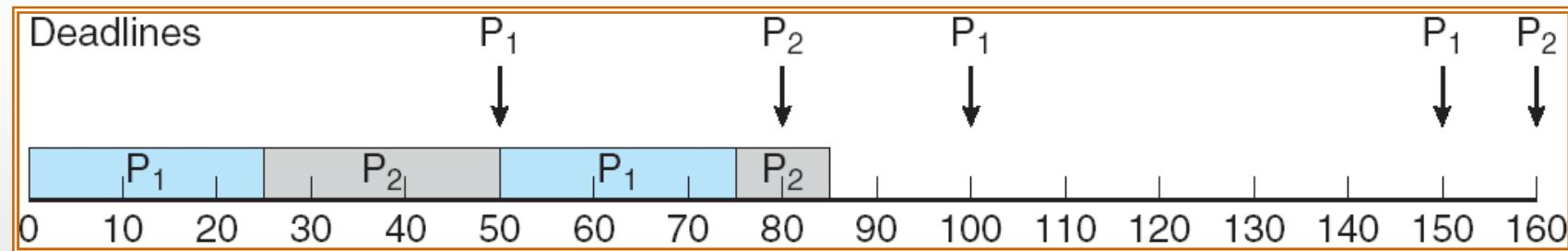
Longer periods = lower priority

P_1 is assigned a higher priority than P_2 .





Missed Deadlines with Rate Monotonic Scheduling



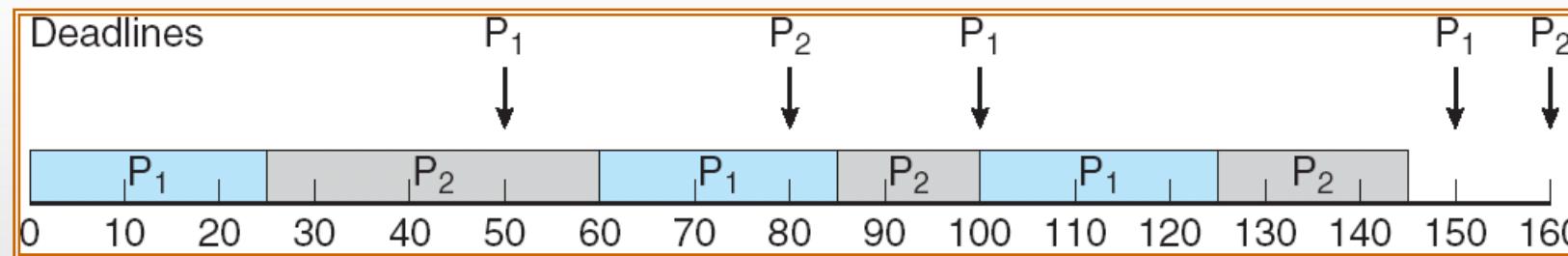


Earliest Deadline First Scheduling

Priorities are assigned according to deadlines:

the earlier the deadline, the higher the priority;

the later the deadline, the lower the priority.





Proportional Share Scheduling

T shares are allocated among all processes in the system.

An application receives N shares where $N < T$.

This ensures each application will receive N / T of the total processor time.





Pthread Scheduling

The Pthread API provides functions for managing real-time threads.

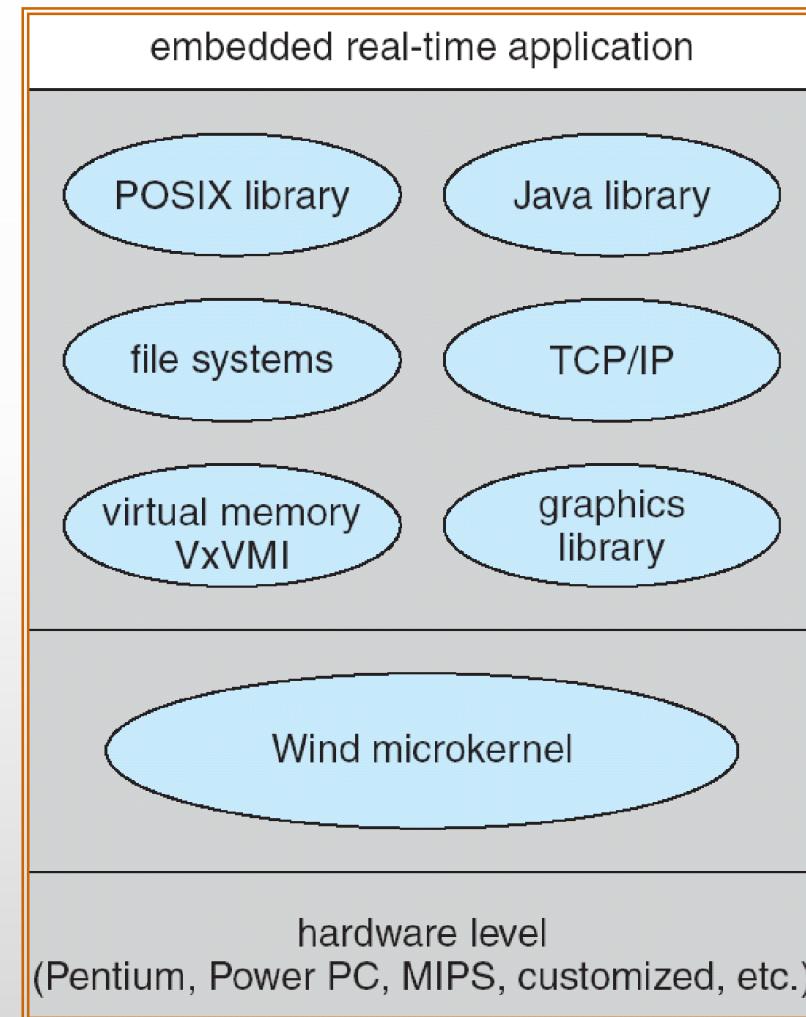
Pthreads defines two scheduling classes for real-time threads:

- (1) SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue.
There is no time-slicing for threads of equal priority.
- (2) SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority.





VxWorks 5.0





Wind Microkernel

The Wind microkernel provides support for the following:

- (1) Processes and threads;
- (2) preemptive and non-preemptive round-robin scheduling;
- (3) manages interrupts (with bounded interrupt and dispatch latency times);
- (4) shared memory and message passing interprocess communication facilities.





Thank You Very Much

Benjamin Kommey

bkommey.coe@knust.edu.gh

nii_kommey@msn.com

050 770 32 86

Skype_id: calculus.affairs





“ Most people overestimate what they can do in one year and underestimate what they can do in ten years. ”

Bill Gates





Revisions

What is an Operating System?

An OS is a program that acts an intermediary between the user of a computer and computer hardware.

Major cost of general purpose computing is software.

OS simplifies and manages the complexity of running application programs efficiently.





Operating System Views

Resource allocator

to allocate resources (software and hardware) of the computer system and manage them efficiently.

Control program

Controls execution of user programs and operation of I/O devices.

Kernel

The program that executes forever (everything else is an application with respect to the kernel).





Operating System Spectrum

Monitors and Small Kernels

Batch Systems

Polling vs. interrupt

Multiprogramming

Timesharing Systems

concept of time slice

Parallel and Distributed Systems

symmetric vs. asymmetric multiprocessing

Real-time systems

Hard vs. soft real time





Computer System Structures

Computer System Operation

I/O Structure

Storage Structure

 Storage Hierarchy

Hardware Protection

General System Architecture

System Calls and System Programs

Command Interpreter





Operating System Services

Services that provide user-interfaces to OS

Program execution - load program into memory and run it

I/O Operations - since users cannot execute I/O operations directly

File System Manipulation - read, write, create, delete files

Communications - interprocess and intersystem

Error Detection - in hardware, I/O devices, user programs

Services for providing efficient system operation

Resource Allocation - for simultaneously executing jobs

Accounting - for account billing and usage statistics

Protection - ensure access to system resources is controlled





Process Management

Process - fundamental concept in OS

Process is a program in execution.

Process needs resources - CPU time, memory, files/data and I/O devices.

OS is responsible for the following process management activities.

Process creation and deletion

Process suspension and resumption

Process synchronization and interprocess communication

Process interactions - deadlock detection, avoidance and correction





Process Concept

An operating system executes a variety of programs

batch systems - jobs

time-shared systems - user programs or tasks

job and program used interchangeably

Process - a program in execution

process execution proceeds in a sequential fashion

A process contains

program counter, stack and data section

Process States

e.g. new, running, ready, waiting, terminated.





Process Control Block PCB

Contains information associated with each process

Process State - e.g. new, ready, running etc.

Program Counter - address of next instruction to be executed

CPU registers - general purpose registers, stack pointer etc.

CPU scheduling information - process priority, pointer

Memory Management information - base/limit information

Accounting information - time limits, process number

I/O Status information - list of I/O devices allocated



Schedulers

Long-term scheduler (or job scheduler) -

selects which processes should be brought into the ready queue.
invoked very infrequently (seconds, minutes); may be slow.
controls the degree of multiprogramming

Short term scheduler (or CPU scheduler) -

selects which process should execute next and allocates CPU.
invoked very frequently (milliseconds) - must be very fast

Medium Term Scheduler

swaps out process temporarily
balances load for better throughput



Process Creation

Processes are created and deleted dynamically

Process which creates another process is called a *parent* process; the created process is called a *child* process.

Result is a tree of processes

e.g. UNIX - processes have dependencies and form a hierarchy.

Resources required when creating process

CPU time, files, memory, I/O devices etc





Process Termination

Process executes last statement and asks the operating system to delete it (*exit*).

- Output data from child to parent (via wait).

- Process' resources are de-allocated by operating system.

Parent may terminate execution of child processes.

- Child has exceeded allocated resources.

- Task assigned to child is no longer required.

- Parent is exiting

- OS does not allow child to continue if parent terminates

- Cascading termination





Producer-Consumer Problem

Paradigm for cooperating processes;
producer process produces information that is consumed by a consumer process.

We need buffer of items that can be filled by producer and emptied by consumer.

Unbounded-buffer places no practical limit on the size of the buffer. Consumer may wait, producer never waits.

Bounded-buffer assumes that there is a fixed buffer size.

Consumer waits for new item, producer waits if buffer is full.

Producer and Consumer must synchronize



Threads

Processes do not share resources well

high context switching overhead

A thread (or lightweight process)

basic unit of CPU utilization; it consists of:

program counter, register set and stack space

A thread shares the following with peer threads:

code section, data section and OS resources (open files, signals)

Collectively called a task.

Heavyweight process is a task with one thread.

Thread support in modern systems - e.g. Solaris 2.





Interprocess Communication (IPC)

Mechanism for processes to communicate and synchronize their actions.

Via shared memory

Via Messaging system - processes communicate without resorting to shared variables.

Messaging system and shared memory not mutually exclusive -
can be used simultaneously within a single OS or a single process.

IPC facility provides two operations.

send(*message*) - message size can be fixed or variable

receive(*message*)

Direct vs. Indirect communication





CPU Scheduling

Scheduling Objectives

Levels of Scheduling

Scheduling Criteria

Scheduling Algorithms

Multiple Processor Scheduling

Real-time Scheduling

Algorithm Evaluation





Scheduling Policies

FCFS (First Come First Serve)

Process that requests the CPU *FIRST* is allocated the CPU *FIRST*.

SJF (Shortest Job First)

Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

Priority

A priority value (integer) is associated with each process. CPU allocated to process with highest priority.

Round Robin

Each process gets a small unit of CPU time

MultiLevel

ready queue partitioned into separate queues

Variation: Multilevel Feedback queues.





Process Synchronization

The Critical Section Problem

Synchronization Hardware

Semaphores

Classical Problems of Synchronization

Critical Regions

Monitors





The Critical Section Problem

Requirements

Mutual Exclusion

Progress

Bounded Waiting

Solution to the 2 process critical section problem

Bakery Algorithm

Solution to the n process critical section problem

Before entering its critical section, process receives a number. Holder of the smallest number enters critical section



Synchronization Hardware

Test and modify the content of a word atomically - **Test-and-set instruction**

```
function Test-and-Set (var target: boolean): boolean;  
begin  
    Test-and-Set := target;  
    target := true;  
end;
```

Mutual exclusion using test and set.

Bounded waiting mutual exclusion using test and set.

“SWAP” instruction





Mutual Exclusion with Test-and-Set

Shared data: var lock: boolean (initially false)

Process Pi

repeat

while *Test-and-Set (lock)* **do** *no-op*;

 critical section

lock := false;

 remainder section

until *false*;



Semaphore

Semaphore S - integer variable

used to represent number of abstract resources.

Binary vs. counting semaphores.

Can only be accessed via two indivisible (atomic) operations

wait (S): **while** $S \leq 0$ **do** no-op

$S := S - 1;$

signal (S): $S := S + 1;$

P or wait used to acquire a resource, decrements count

V or signal releases a resource and increments count

If P is performed on a count ≤ 0 , process must wait for V or the release of a resource.

Block/resume implementation of semaphores





Classical Problems of Synchronization

Bounded Buffer Problem

Readers and Writers Problem

Dining-Philosophers Problem





Readers-Writers Problem

Shared Data

```
var mutex, wrt: semaphore (=1);  
readcount: integer (= 0);
```

Writer Process

```
wait(wrt);  
...  
writing is performed  
...  
signal(wrt);
```

Reader process

```
wait(mutex);  
readcount := readcount +1;  
if readcount = 1 then wait(wrt);  
signal(mutex);  
...  
reading is performed  
...  
wait(mutex);  
readcount := readcount - 1;  
if readcount = 0 then signal(wrt);  
signal(mutex);
```





Critical Regions

High-level synchronization construct

A shared variable v of type T is declared as:

```
var v: shared T
```

Variable v is accessed only inside statement

```
region v when B do S
```

where B is a boolean expression.

While statement S is being executed, no other process can access variable v



Monitors

High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
type monitor-name = monitor
    variable declarations
    procedure entry P1 (...);
        begin ... end;
    procedure entry P2 (...);
        begin ... end;
    :
    :
procedure entry Pn(...);
    begin ... end;
begin
    initialization code
end.
```





Deadlocks

System Model

Resource allocation graph, claim graph (for avoidance)

Deadlock Characterization

Conditions for deadlock - mutual exclusion, hold and wait, no preemption, circular wait.

Methods for handling deadlocks

Deadlock Prevention

Deadlock Avoidance

Deadlock Detection

Recovery from Deadlock

Combined Approach to Deadlock Handling





Deadlock Prevention

If any one of the conditions for deadlock (with reusable resources) is denied, deadlock is impossible.

Restrain ways in which requests can be made

Mutual Exclusion - cannot deny (important)

Hold and Wait - guarantee that when a process requests a resource, it does not hold other resources.

No Preemption

If a process that is holding some resources requests another resource that cannot be immediately allocated to it, the process releases the resources currently being held.

Circular Wait

Impose a total ordering of all resource types





Deadlock Avoidance

Requires that the system has some additional apriori information available.

Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.

Computation of Safe State

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

Sequence $\langle P_1, P_2, \dots, P_n \rangle$ is safe, if for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by P_j with $j < i$.

Safe state - no deadlocks, unsafe state - possibility of deadlocks

Avoidance - system will never reach unsafe state





Algorithms for Deadlock Avoidance

Resource allocation graph algorithm
only one instance of each resource type

Banker's algorithm

Used for multiple instances of each resource type.

Data structures required

Available, Max, Allocation, Need

Safety algorithm

resource request algorithm for a process





Deadlock Detection

Allow system to enter deadlock state

Detection Algorithm

Single instance of each resource type

use wait-for graph

Multiple instances of each resource type

variation of banker's algorithm

Recovery Scheme

Process Termination

Resource Preemption





Memory Management

Main Memory is an array of addressable words or bytes that is quickly accessible.

Main Memory is volatile.

OS is responsible for:

- Allocate and deallocate memory to processes.

- Managing multiple processes within memory - keep track of which parts of memory are used by which processes. Manage the sharing of memory between processes.

- Determining which processes to load when memory becomes available.



Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages.

Compile time, Load time, Execution time

Other techniques for better memory utilization

Dynamic Loading - Routine is not loaded until it is called.

Dynamic Linking - Linking postponed until execution time

Overlays - Keep in memory only those instructions and data that are needed at any given time

Swapping - A process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution

MMU - Memory Management Unit

Hardware device that maps virtual to physical address





Contiguous Allocation

Divides Main memory usually into two partitions

Resident Operating System, usually held in low memory with interrupt vector and User processes held in high memory.

Single partition allocation

Relocation register scheme used to protect user processes from each other, and from changing OS code and data

Multiple partition allocation

holes of various sizes are scattered throughout memory. When a process arrives, it is allocated memory from a hole large enough to accommodate it.

Variation: Fixed partition allocation





Dynamic Storage Allocation Problem

How to satisfy a request of size n from a list of free holes.

First-fit

Best-fit

Worst-fit

Fragmentation

External fragmentation

total memory space exists to satisfy a request, but it is not contiguous.

Internal fragmentation

allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

Reduce external fragmentation by compaction



Paging

Logical address space of a process can be non-contiguous;
process is allocated physical memory wherever the latter is available.

Divide physical memory into fixed size blocks called *frames*
size is power of 2, 512 bytes - 8K

Divide logical memory into same size blocks called *pages*.
Keep track of all free frames.

To run a program of size n pages, find n free frames and load program.

Set up a page table to translate logical to physical addresses.

Note:: Internal Fragmentation possible!!





Page Table Implementation

Page table is kept in main memory

Page-table base register (PTBR) points to the page table.

Page-table length register (PTLR) indicates the size of page table.

Every data/instruction access requires 2 memory accesses.

One for page table, one for data/instruction

Two-memory access problem solved by use of special fast-lookup hardware cache (i.e. cache page table in registers)

associative registers or translation look-aside buffers (TLBs)



Paging Methods

Multilevel Paging

Each level is a separate table in memory
converting a logical address to a physical one may take 4 or more memory
accesses.

Caching can help performance remain reasonable.

Inverted Page Tables

One entry for each real page of memory. Entry consists of virtual address of
page in real memory with information about process that owns page.

Shared Pages

Code and data can be shared among processes. Reentrant (non self-modifying) code can be shared. Map them into pages with common page frame mappings





Segmentation

Memory Management Scheme that supports user view of memory.

A program is a collection of segments.

A segment is a logical unit such as

- main program, procedure, function

- local variables, global variables, common block

- stack, symbol table, arrays

Protect each entity independently

Allow each segment to grow independently

Share each segment independently





Segmented Paged Memory

Segment-table entry contains not the base address of the segment, but the base address of a page table for this segment.

Overcomes external fragmentation problem of segmented memory.

Paging also makes allocation simpler; time to search for a suitable segment (using best-fit etc.) reduced.

Introduces some internal fragmentation and table space overhead.

Multics - single level page table

IBM OS/2 - OS on top of Intel 386

uses a two level paging scheme





Virtual Memory

Virtual Memory

Separation of user logical memory from physical memory.

Only *PART* of the program needs to be in memory for execution.

Logical address space can therefore be much larger than physical address space.

Need to allow pages to be swapped in and out.

Virtual Memory can be implemented via

Paging

Segmentation



Demand Paging

Bring a page into memory only when it is needed.

- Less I/O needed

- Less Memory needed

- Faster response

- More users

The first reference to a page will trap to OS with a page fault.

OS looks at another table to decide

- Invalid reference - abort

- Just not in memory





Page Replacement

Prevent over-allocation of memory by modifying page fault service routine to include page replacement.

Use modify(dirty) bit to reduce overhead of page transfers
- only modified pages are written to disk.

Page replacement

large virtual memory can be provided on a smaller physical memory.





Page Replacement Strategies

The Principle of Optimality

Replace the page that will not be used again the farthest time into the future.

Random Page Replacement

Choose a page randomly

FIFO - First in First Out

Replace the page that has been in memory the longest.

LRU - Least Recently Used

Replace the page that has not been used for the longest time.

LRU Approximation Algorithms - reference bit, second-chance etc.

LFU - Least Frequently Used

Replace the page that is used least often.

NUR - Not Used Recently

An approximation to LRU

Working Set

Keep in memory those pages that the process is actively using





Allocation of Frames

Single user case is simple

User is allocated any free frame

Problem: Demand paging + multiprogramming

Each process needs minimum number of pages based on instruction set architecture.

Two major allocation schemes:

Fixed allocation - (1) equal allocation (2) Proportional allocation.

Priority allocation - May want to give high priority process more memory than low priority process





Thrashing

If a process does not have enough pages, the page-fault rate is very high. This leads to:

low CPU utilization.

OS thinks that it needs to increase the degree of multiprogramming

Another process is added to the system.

System throughput plunges...

Thrashing

A process is busy swapping pages in and out.

In other words, a process is spending more time paging than executing.





Working Set Model

$\Delta \equiv$ working-set window

a fixed number of page references, e.g. 10,000 instructions

WSS_j (working set size of process P_j) - total number of pages referenced in the most recent Δ (varies in time)

If Δ too small, will not encompass entire locality.

If Δ too large, will encompass several localities.

If $\Delta = \infty$, will encompass entire program.

$D = \sum WSS_j \equiv$ total demand frames

If $D > m$ (number of available frames) \Rightarrow thrashing

Policy: If $D > m$, then suspend one of the processes





File System Management

File is a collection of related information defined by creator - represents programs and data.

OS is responsible for

File creation and deletion

Directory creation and deletion

Supporting primitives for file/directory manipulation.

Mapping files to disks (secondary storage).

Backup files on archival media (tapes).





File Concept

Contiguous logical address space

OS abstracts from the physical properties of its storage device to define a logical storage unit called file. OS maps files to physical devices.

Types

Data, Program, Documents

File Attributes

Name, type, location, size, protection etc.

File Operations

Create, read, write, reposition, delete etc





Directory Structure

Number of files on a system can be extensive

Hold information about files within partitions called directories.

Device Directory: A collection of nodes containing information about all files on a partition. Both the directory structure and files reside on disk. Backups of these two structures are kept on tapes.

Operations on a directory

create a file, delete a file, search for a file, list directory etc





Logical Directory Organization

Goals - Efficiency, Naming, grouping

Single Level Directories

Single level for all users, naming and grouping problem

Two Level Directories

first level - user directories, second level - user files

Tree Structured Directories

arbitrary depth of directories, leaf nodes are files

Acyclic Graph Directories

allows sharing, implementation by links or shared files

General Graph Directories

allow cycles - must be careful during traversal and deletion





File Protection - Access Lists and Groups

Associate each file/directory with access list

Problem - length of access list..

Solution - condensed version of list

Mode of access: read, write, execute

Three classes of users

owner access - user who created the file

groups access - set of users who are sharing the file and need similar access

public access - all other users

In UNIX, 3 fields of length 3 bits are used.

Fields are user, group, others(u,g,o),

Bits are read, write, execute (r,w,x).

E.g. chmod go+rwx file , chmod 761 game





File-System Implementation

File System Structure

File System resides on secondary storage (disks). To improve I/O efficiency, I/O transfers between memory and disk are performed in blocks.

Read/Write/Modify/Access each block on disk.

File System Mounting - File System must be mounted before it can be available to process on the system. The OS is given the name of the device and the mount point.

Allocation Methods

Free-Space Management

Directory Implementation

Efficiency and Performance, Recovery





Allocation of Disk Space

Low level access methods depend upon the disk allocation scheme used to store file data

Contiguous Allocation

Each file occupies a set of contiguous blocks on the disk. Dynamic storage allocation problem. Files cannot grow.

Linked List Allocation

Each file is a linked list of disk blocks. Blocks may be scattered anywhere on the disk. Not suited for random access.

Variation - FILE ALLOCATION TABLE (FAT) mechanisms

Indexed Allocation

Brings all pointers together into the index block. Need index table. Can link blocks of indexes to form multilevel indexes





Thank You Very Much

Benjamin Kommey

bkommey.coe@knust.edu.gh

nii_kommey@msn.com

050 770 32 86

Skype_id: calculus.affairs

