

Operating System Concepts

(CSc -514)-2008

Introduction

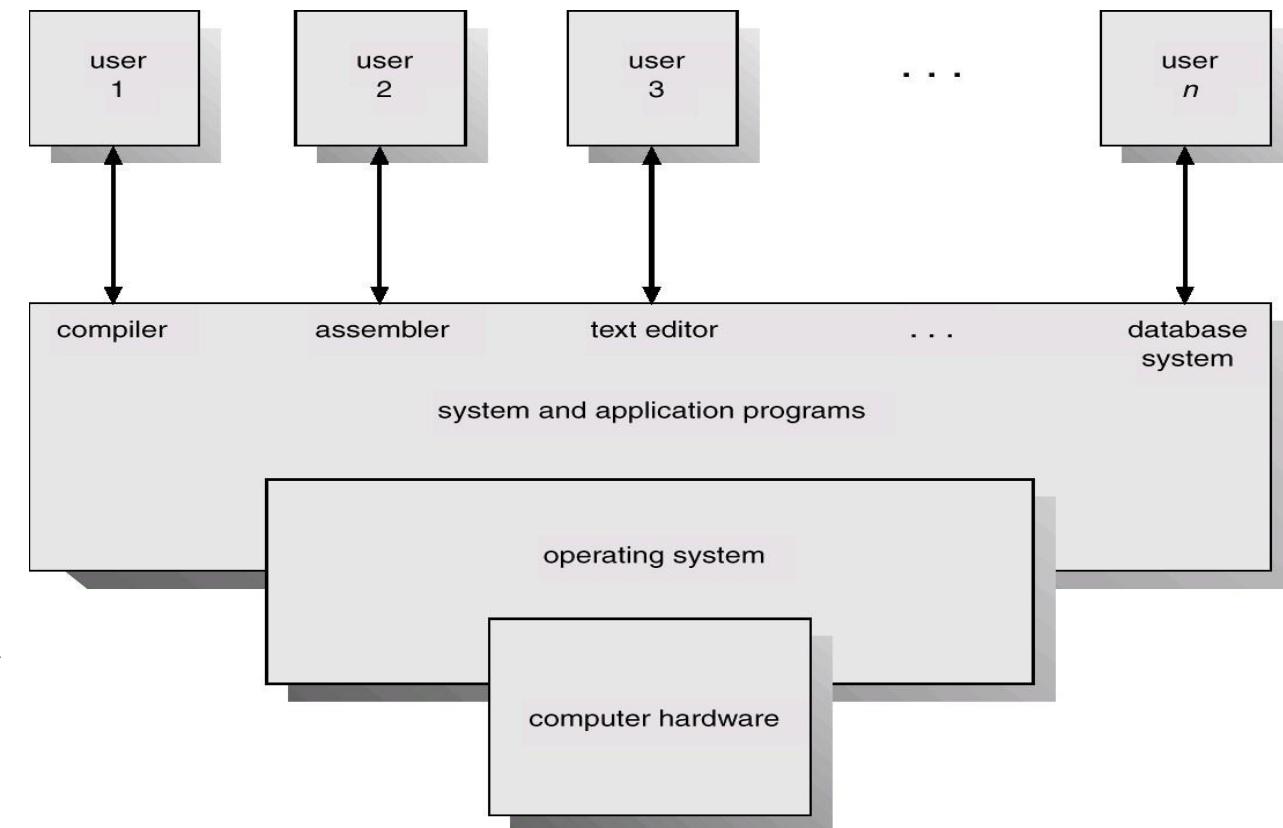
- Computer system
- What is an operating system?
- Variations of OS
- Computer System Operation
- Memory Hierarchy
- Hardware Protection
- Common OS Concepts
- System Calls

Reading:

Chapter 1 of Tanenbaum or chapter 1, 2 and 3 of silberschatz

Computer system

Computer system provide a capability for gathering data, performing computations, storing information, communicating with other computer system and generating output.



Computer system

1. **Hardware** – provides basic computing resources (CPU, memory, I/O devices).
2. **Operating system** – controls and coordinates the use of the hardware among the various application programs for the various users.
3. **Applications programs** – define the ways in which the system resources are used to solve the computing problems of the users (compilers, database systems, video games, business programs).
4. **Users** (people, machines, other computers).

What is an operating system?

An operating system (OS) is a collection of system programs that together control the operation of a computer system.

Operating system goals:

- Execute user programs and make solving user problems easier.
- Make the computer system convenient to use.
- Use the computer hardware in an efficient manner.

Provides an environment within which other programs can do useful work.

Two Functions of OS

- OS as an Extended Machine
- OS as a Resource Manager

OS as an Extended Machine

OS creates higher-level abstraction for programmer

Example: (Floppy disk I/O operation)

- disks contains a collection of named files

- each file must be open for READ/WRITE
 - after READ/WRITE complete close that file
 - no any detail to deal

OS shields the programmer from the disk hardware and presents a simple file oriented interface.

OS function is to present the user with the equivalent of an extended machine or virtual machine that is easier to program than the underlying hardware.

Challenges:

What are the right abstractions? How to achieve this?

OS as a Resource Manager

- What happens if three programs try to print their output on the same printer at the same time?
- What happens if two network users try to update a shared document at same time?

OS primary function is to manage all pieces of a complex system.

Advantages:

- Virtualizes resource so multiple users/applications can share
- Protect applications from one another
- Provide efficient and fair access to resources

Challenges:

- What mechanisms? What policies?

OS evolution

Computer Generation, Component and OS Types

1st (1945-55)

Vacuum Tubes

User Driven

2 nd (1955-65)	Transistor	Batch
3 rd (1965-80)	IC	Multiprogramming
4 th (1980-present)	PC	Client Server/Distributed

Batch System

During the 2nd generation, operator hired to run computer, the user prepared a job – which consisted of the program, the data, and some control information about the nature of the jobs – and submitted it to the computer operator. The operator perform the function of an OS.

Batch: Group of jobs submitted to machine together

-Operator collects jobs; orders efficiently; runs one at a time

Advantages:

- Amortize setup costs over many jobs
- Operator more skilled at loading tapes
- Keep machine busy while programmer thinks **Problem:**
- User must wait for results until batch collected and submitted

(If bug receive, memory and register dump; submit job again!)

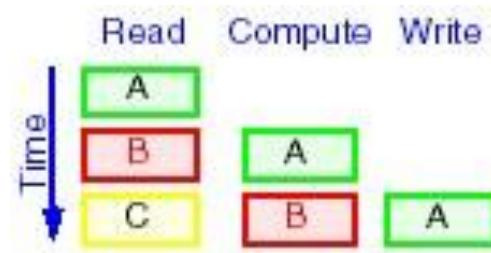
Spooling

(During 3rd generation)

Simultaneous Peripheral Operation On Line

Problem: Mechanical I/O devices much slower than CPU.

- Read 17 cards/sec vs. execute 1000s instructions/sec **Spooling:** Overlap I/O (e.g., card reading and line printing) with execution



Advantage: Improves throughput and utilization

Disadvantage: Single job must wait during I/O for data

New OS functionality

- Buffering, DMA, interrupt handling

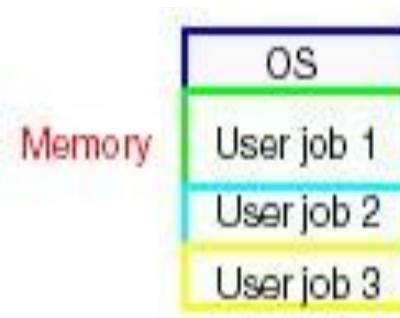
Multiprogramming

(During 3rd generation)

- Problem:**
- wait time still too long
 - long jobs delay everything

Spooling provides pool of ready jobs.

Multiple jobs in memory at the same time. Each memory space protected from each other. OS picks one, execute it for a while, stops (e.g. when programs reads for input or randomly), picks other to run.



Advantage: Improves throughput and utilization

Disadvantage: Still not interactive

New OS functionality

- Job scheduling policies
- Memory management and protection (virtual memory)

Timesharing

(During 3rd generation)

Problem:

- increasing number of users who want to interact with programs while they are running.
- humans' time is expensive don't want to wait.

The CPU executes multiple jobs by switching among them, but the switches occurs so frequently that the user can interact with each program while it is running.

Goal: Improve user's response time

Advantage:

- Users easily submit jobs and get immediate feedback

New OS functionality

- More complex job scheduling, memory management
- Concurrency control and synchronization

Personal Computers

Dedicated machine per user

CPU utilization is not a prime concern

Hence, some of the design decision made for advanced system may not be appropriate for these smaller system. But other design decisions, such as those for security, are appropriate because PCs can now be connected to other computer and users through the network and the web.

There are the different types of OS for PC such as Windows, MacOS, UNIX, Linux.

Conclusion: OS Functionality changes with hardware and users

Real-Time Systems

Time is a key parameter

Often used as a control device in a dedicated application such as controlling scientific experiments, medical imaging systems, industrial control systems, and some display systems.

Real-Time systems may be either *hard* or *soft* real-time.

Hard Real-Time:

-well defined fixed time constraints, processing must be done within the defined constraints, or the system will fail. ***Soft Real-Time:***

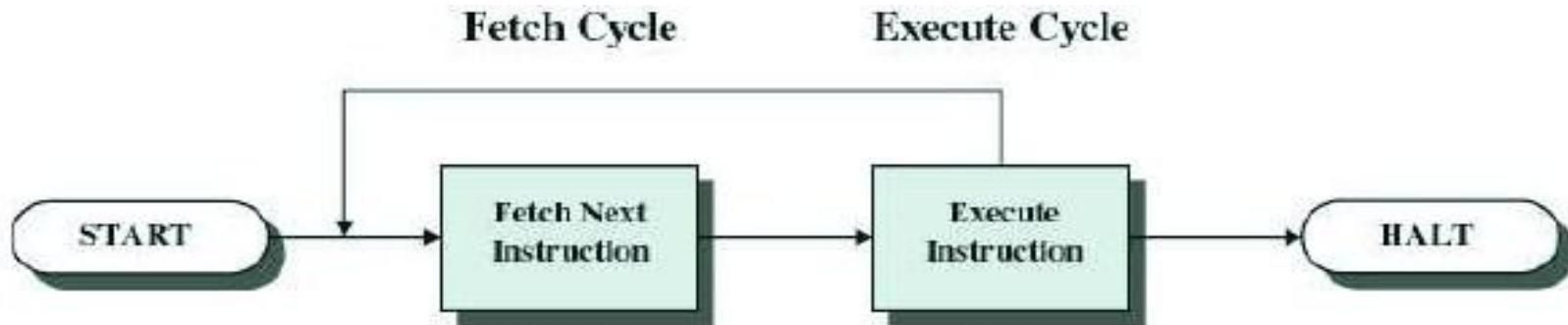
-less stringent timing constraints, and do not support the dead line scheduling.

Handheld Systems

- Personal Digital Assistants (PDAs)
- Cellular telephones
- Issues:
 - Limited memory
 - Slow processors
 - Small display screens.

Computer System Operation

CPU fetches the instructions from memory and executes them. The basic cycle of CPU is to fetch the first instruction from memory, decodes it to determine its type and operands, execute it, and then fetch, decode, and execute subsequent instructions.



Registers are used to hold variables and temporary results.

Program counter – contains the memory address of next instruction to be fetched.

Instruction register – holds the actual instruction.

The instruction is decoded to determine what action to be performed. The action is specified by instruction's **opcode** bits.

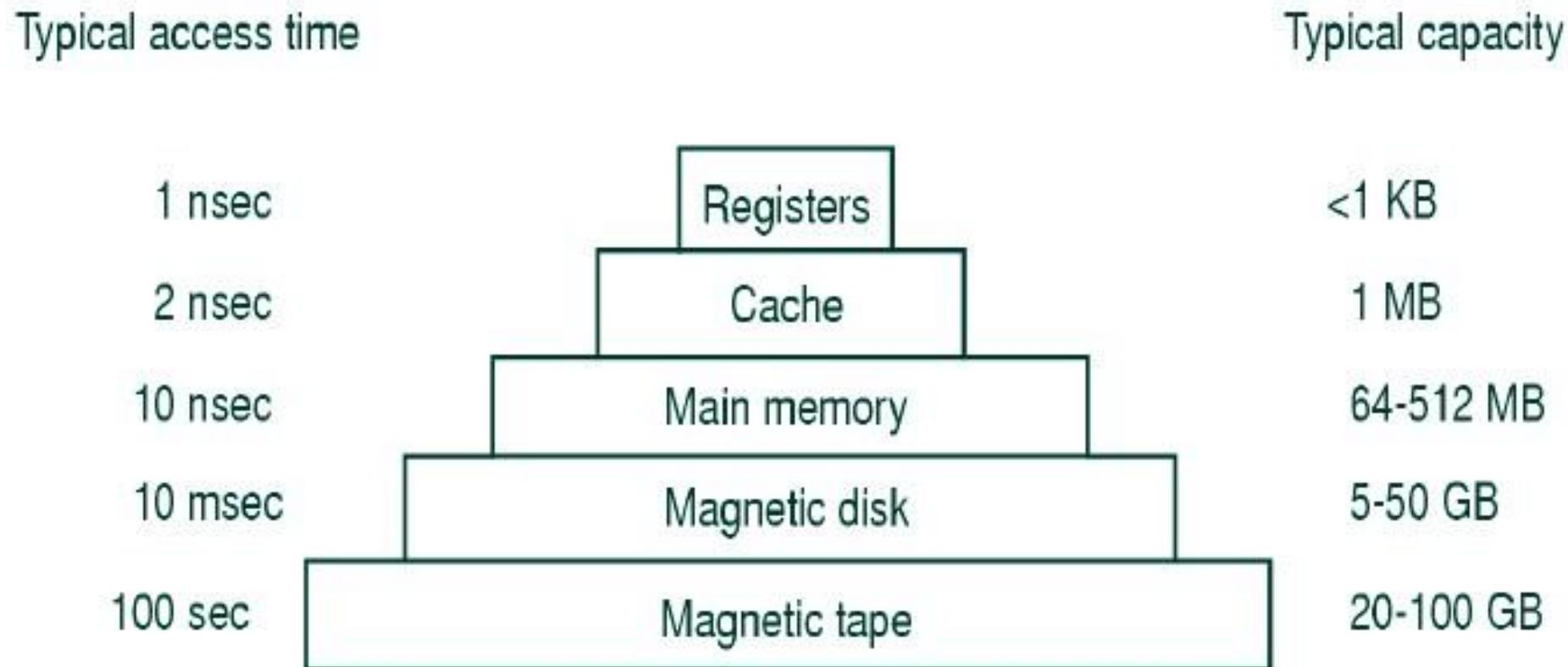
Computer System Operation

The PSW (Program Status Word) - contains the condition code bits, which are set by comparison instruction, the CPU priority, the modes (user or kernel), and various other control bits.

In timesharing system, OS stops a running program to start another one. When stopping a program it save all registers so they can be restored when the programs runs later.

Device controller informs CPU that it has finished its operation by causing an *interrupt* (Signal generated by hardware such as I/O), a *trap* is a software generated interrupt caused either by an error (e.g., division by 0, trying to excess the non existent I/O etc.) or by user (such as by calling System Calls). The CPU responds to the trap and interrupt by saving the current value of program counter.

Memory Structure



A typical memory hierarchy (based on year 2000)

Hardware Protection

Situation: *Multiprogramming put several programs in memory at the same time. When one program contain erroneous data it might modify the program or data of another program.(MS-DOS and MacOS).*

OS must insure that an incorrect program can not cause other programs to execute incorrectly, and must terminate the such erroneous program.

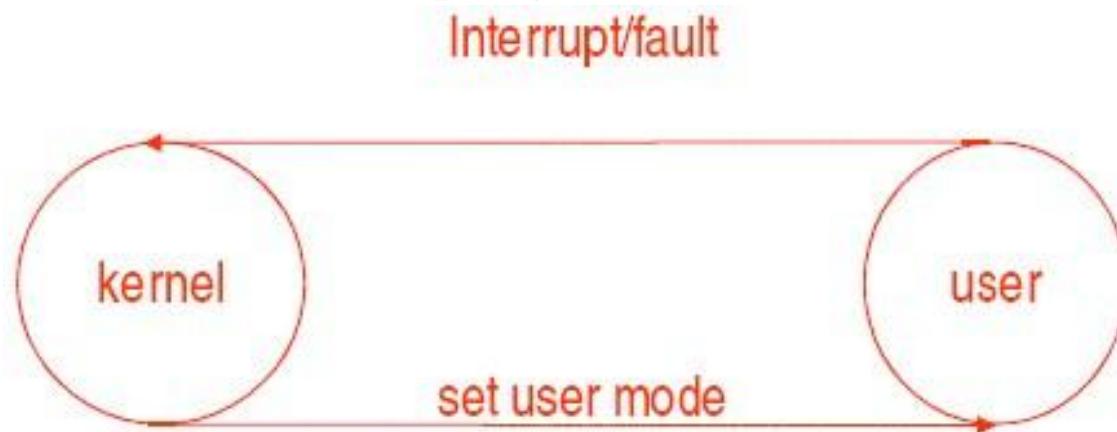
- Dual-Mode Operation
- I/O Protection
- Memory Protection
- CPU Protection

Dual-Mode Operation

- Provide hardware support to differentiate between at least two modes of operations.
 1. **User mode** . execution done on behalf of a user.
User programs runs in user mode, which permits only the subset of instructions to be executed and subset of features to be accessed.
 2. **Kernel mode** (or supervisor mode) . execution done on behalf of operating system.
OS runs in kernel mode, giving it access to the complete hardware.
- Mode bit is added to the hardware of the computer to indicate the current mode: kernel(0) and user(1).

Dual-Mode Operation

When an trap or interrupt occurs hardware switches from user mode to the kernel mode. Thus, whenever the OS gains control of computer, it is in kernel mode. The system always switches to user mode before passing control to the user program.



Privileged instructions can be issued only in kernel mode.

Advantages:

protect OS from errant users and errant user from one another.

I/O Protection

Situation: A user program may disrupt the normal operation of the system by issuing illegal I/O operation, by accessing memory locations within the OS itself.

- All I/O instructions must be privileged instructions. Thus the user can not issue the I/O instruction directly.
- Must ensure that a user program could never gain control of the computer in kernel mode (i.e., a user program that, as part of its execution, stores a new address in the interrupt vector).

Memory Protection

Problem:

- *How to separate each program's memory space ? - How to handle relocation ?*

- Must provide memory protection at least for the interrupt vector and the interrupt service routines.
- In order to have memory protection, add two registers that determine the range of legal addresses a program may access:

Base register - holds the smallest legal physical memory address.

Limit register - contains the size of the range

- Memory outside the defined range is protected.

CPU Protection

Situation: *To prevent a user program from getting stuck in an infinite loop or not calling system services, and never returning control to the OS.*

- **Timer** - interrupts computer after specified period to ensure operating system maintains control.
 - Timer is decremented every clock tick.
 - When timer reaches the value 0, an interrupt occurs.

If the timer interrupts, control transfers automatically to the OS, which may treat the interrupt as a fatal error or may give the program more time.

- Timer commonly used to implement time sharing.
- Time also used to compute the current time.
- Load-timer is a privileged instruction.

Common OS Concepts

- Process Management
- Memory Management
- File system Management
- Device Management

Process Management

A process is a program in execution. A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task.

- The operating system is responsible for the following activities in connection with process management.
 - Process creation and deletion.
 - Process suspension and resumption.
 - Provision of mechanisms for:
- process synchronization
- process communication
- deadlock handling

Memory Management

Memory is a large array of words or bytes, each with its own address. It is a repository of quickly accessible data shared by the CPU and I/O devices.

- The operating system is responsible for the following activities in connections with memory management:
 - Keep track of which parts of memory are currently being used and by whom.
 - Decide which processes to load when memory space becomes available.
 - Allocate and deallocate memory space as needed.

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.

- The operating system 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.

I/O Management

All computers have physical devices for acquiring input and producing output. The I/O system consists of:

- A memory management component that includes buffering, caching, and spooling.
- A general device-driver interface.
- Drivers for specific hardware devices.
- Disk management.

System calls

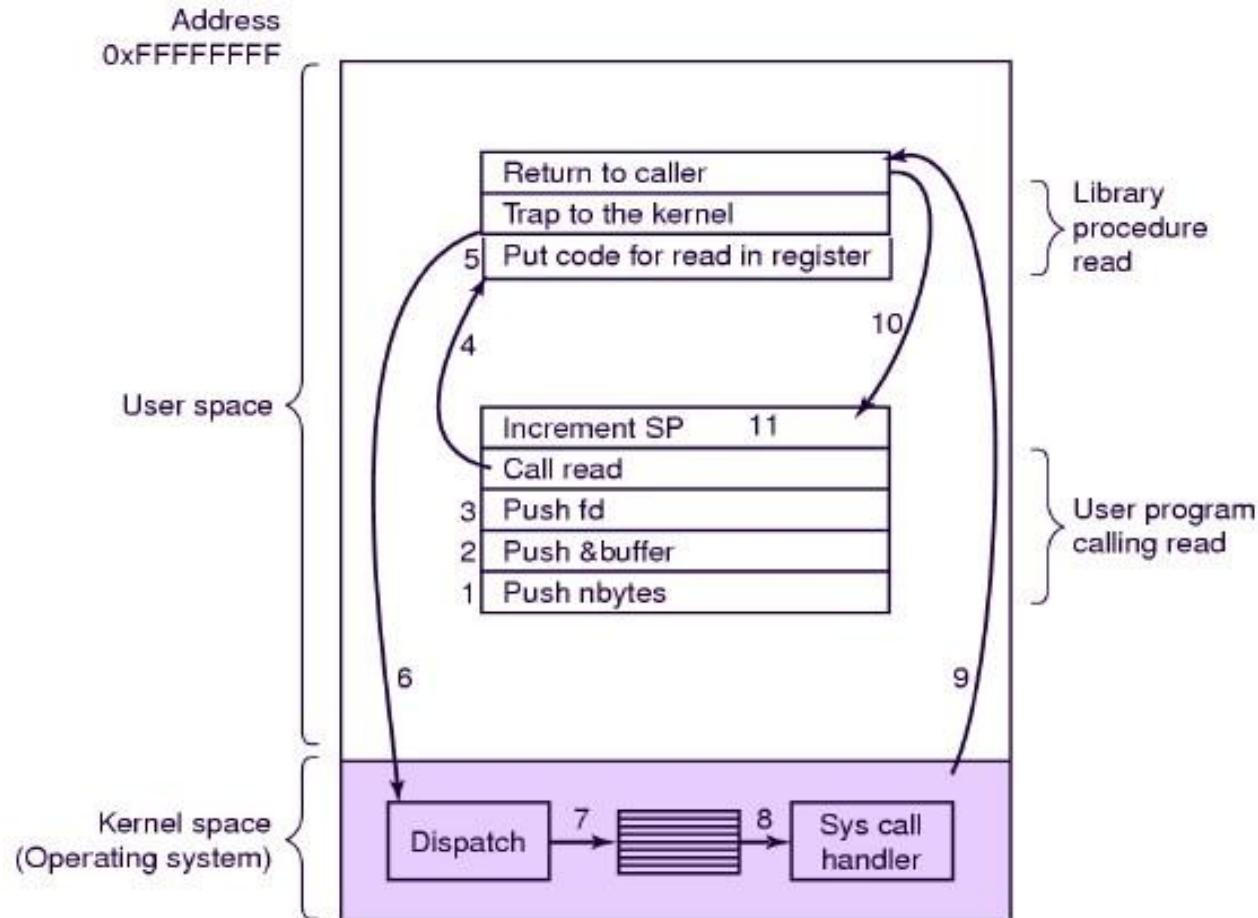
System calls provide the interface between a process and the operating system.

- Generally available as assembly-language instructions.
- Languages defined to replace assembly language for systems programming allow system calls to be made directly (e.g., C, C++)
- System calls performed in a series of steps. Most of the system calls are invoked as the following example system call: read

count = read(fd, buffer, nbytes)

- Push parameters into the stack (1-3)
- Calls library procedure (4)
- Pass parameters in registers.(5)
- Switch from user mode to kernel mode and start to execute.(6)
- Examines the system call number and then dispatches to the correct system call handler via a table of pointer.(7)
- Runs system call handlers (8).
- Once the system call handler completed its work, control return to the library procedure.(9)
- This procedure then return to the user program in the usual way. (10) – Increments SP before call to finish the job.

System calls



Steps in making the system call
read (fd, buffer, nbytes)

Types of System calls

Process management

Call	Description
pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Types of System calls

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Some Win32 API 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
Iseek	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

Home Works

HW #1:

1. Textbook: (Ch. 1) 1, 2, 3, 8, 9 & 14.
2. What does the CPU do when there are no programs to run?
3. What must user programs be prohibited from writing to the memory locations containing the interrupt vector?
4. Classify the following applications as batch-oriented or interactive.
 - a) Word processing
 - b) Generating monthly bank statements
 - c) Computing pi to a million decimal places
5. What is a purpose of system calls?

Processes

Reading: Section 2.1 of Textbook (Tanenbaum)

What is a Process?

The program in execution.

A program is an inanimate entity; only when a processor “breathes life” into it does it become the “active” entity, we call a process.

Programs and Processes

A process is different than a program.

Consider the following analogy

Scenario-1: A computer scientist is baking a birthday cake for his daughter

- Computer scientist - CPU
- Birthday cake recipe - program
- Ingredients - input data Activities: - processes
 - reading the recipe
 - fetching the ingredients
 - backing the cake

Programs and Processes

Scenario-2: Scientist's son comes running in crying, saying he has been stung by a bee.

Scientist records where he - the state of running process

was in the recipe saved

Reach first aid book and
materials - Another process fetched

Follow the first aid action - Processor switched for new
(high priority job) process

On completion of aid, cake - Completion of high priority
baking starts again from job & return back to the last one where it was left

A process is an activity of some kind, it has program, input, output and state.

Process Models

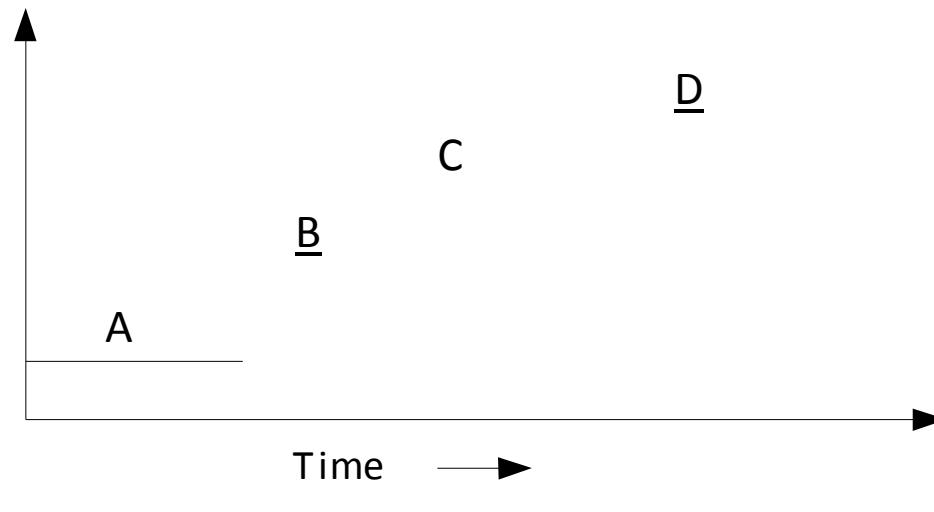
- Uniprogramming

- Multiprogramming

- Multiprocessing

Uniprogramming

Only one process at a time.



Examples: Older systems

Advantages: Easier for OS designer

Disadvantages: Not convenient for user and poor performance

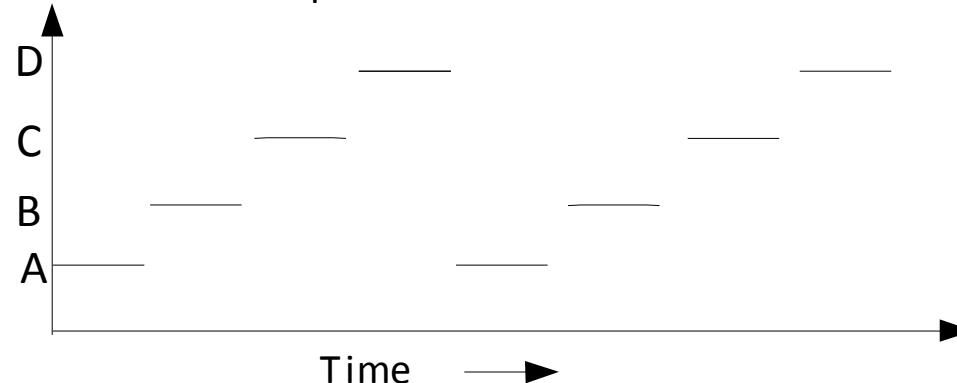
Multiprogramming

Multiple processes at a time.

OS requirements for multiprogramming:

Policy: to determine which process is to schedule.

Mechanism: to switch between the processes.



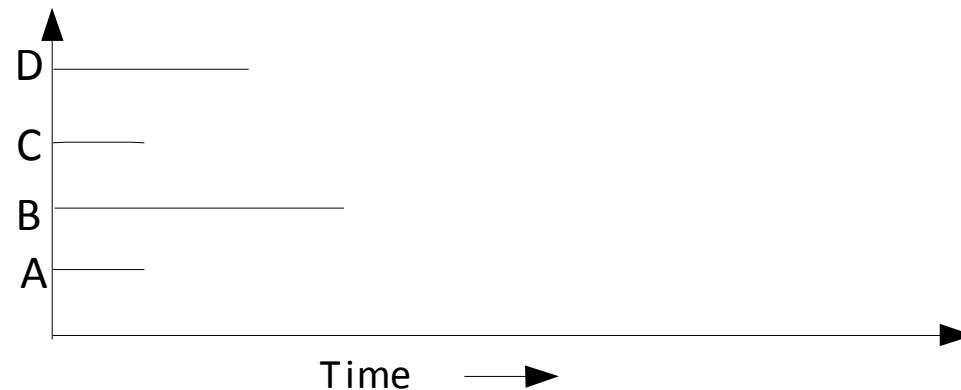
Examples: Unix, WindowsNT

Advantages: Better system performance and user convenience.

Disadvantages: Complexity in OS

Multiprocessing

System with multiple processors.



Process States

A process goes through a series of discrete process states. **Running state:**

- Process executing on CPU. - Only one process at a time.

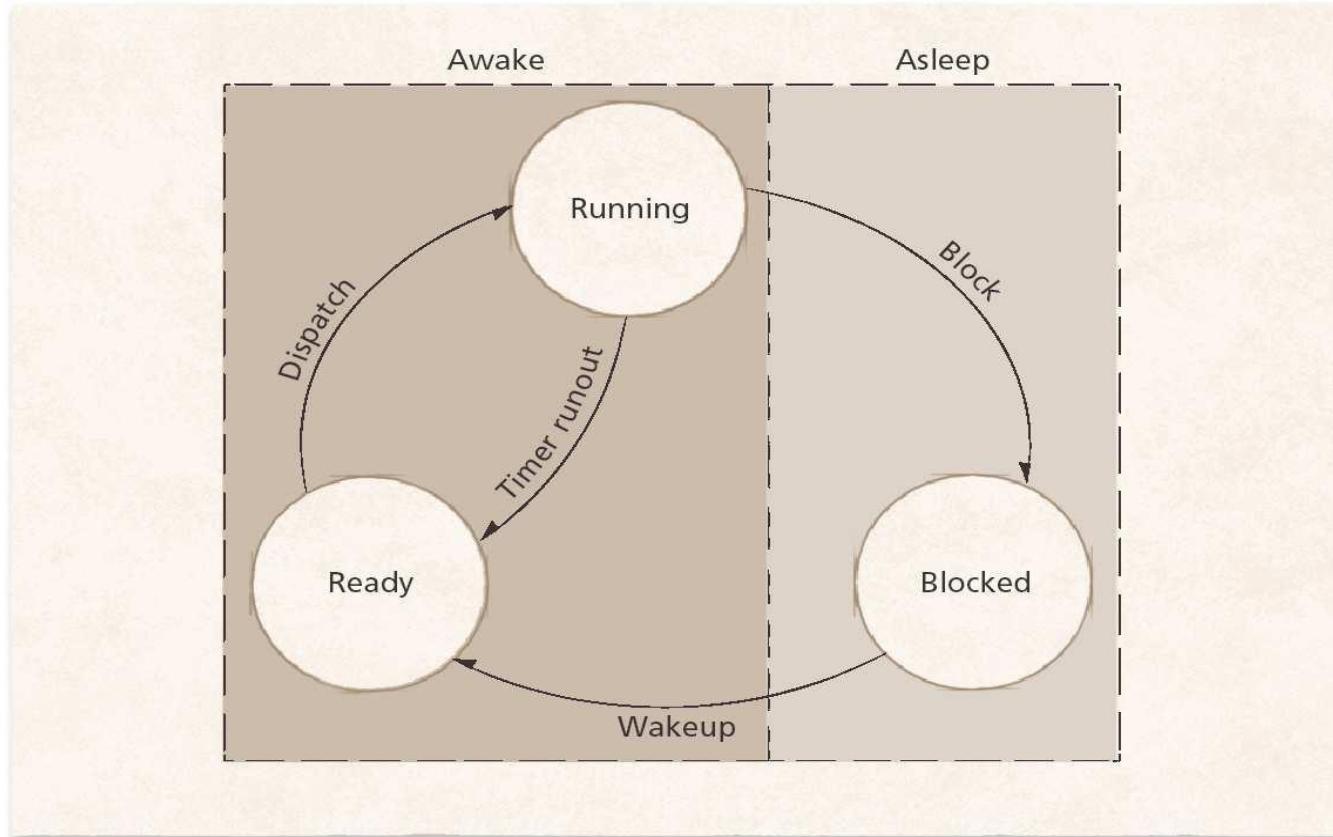
Ready state:

- Process that is not allowed to CPU but is ready to run.
- a list of processes ordered based on priority.

Blocked state:

- Process that is waiting for some event to happen (e. g. I/O completion events).
- a list of processes (no clear priority).

Process States



Process state transitions diagram

Process State Transitions

When a job is admitted to the system, a corresponding process is created and normally inserted at the back of the ready list.

When the CPU becomes available, the process is said to make a state transition from ready to running.

dispatch(processname): ready -> running.

To prevent any one process from monopolizing the CPU, OS specify a time period (quantum) for the process. When the quantum expire makes state transition running to ready. *timerrunout(processname): running -> ready.*

When the process requires an I/O operation before quantum expire, the process voluntarily relinquishes the CPU and changed to the blocked state. *block(processname): running -> block.*

When an I/O operation completes. The process make the transition from block state to ready state. *wakeup(processname): blocked -> ready.*

Process Control Block

Process must be saved when the process is switched from one state to another so that it can be restarted later as it had never been stopped.

The PCB is the data structure containing certain important information about the process -also called process table or processor descriptor.

Process state: running, ready, blocked.

Program counter: Address of next instruction for the process.

Registers: Stack pointer, accumulator, PSW etc.

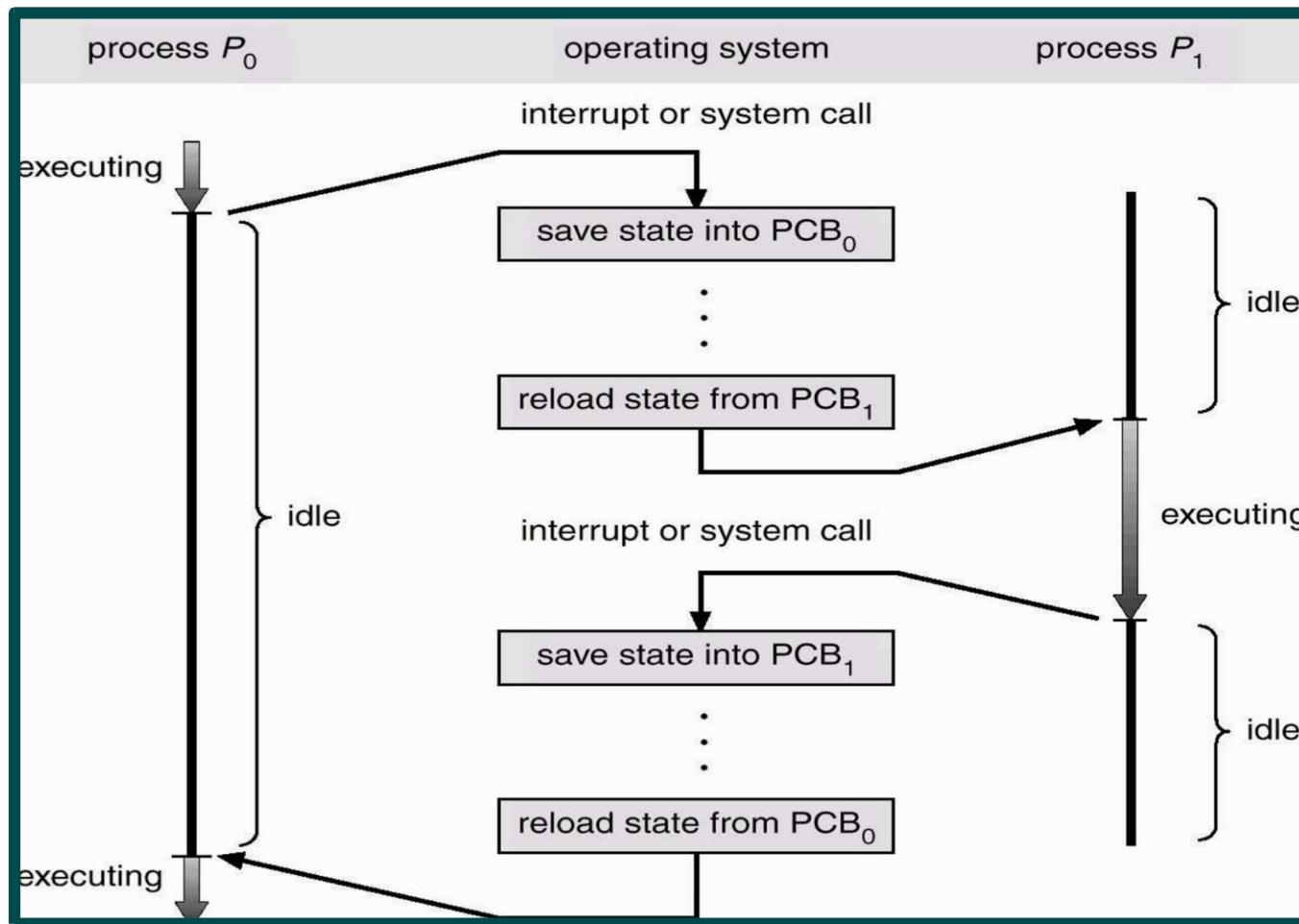
Scheduling information: Process priority, pointer to scheduling queue etc.

Memory-allocation: value of base and limit register, page table, segment table etc.

Accounting information: time limit, process numbers etc.

Status information: list of I/O devices, list of open files etc.

Process Control Block



Operations on Processes

The processes in the system can execute concurrently, and they must be created and deleted dynamically. OS provide the mechanism for process *creation* and *termination*.

Process Creation.

Process Termination.

Process Creation

There are four principal events that cause the process to be created:

- » System initialization.
- » Execution of a process creation system call.
- » User request to create a new process.
- » Initiation of a batch job.

A process may create several new processes during the course of execution. The creating process is called a parent process, whereas the new processes are called children of that process.

Process Creation

Two ways to create a new process

Build a new one from scratch

- Load specified code and data into memory.
- Create and initialize PCB.
- Put processes on the ready list.

Colon an existing one (e.g. Unix fork() syscall)

- Stop the current process and save its state.
- Make copy of code, data, stack, and PCB.
- Add new process PCB to ready list.

Unix Process Creation Ex.

```
#include<stdio.h>
int main(int argc, char *argv[])
{
    int pid; pid = fork(); /* create new
                           process */ if(pid < 0) {/* error
                           occurred */ fprintf(stderr, "fork
                           failed"); exit(-1);
                           } else if(pid == 0){ /* child process */
                           execlp("/bin/ls", "ls",
                           Null); }
                           else { /* parent process */ wait(Null);
                           printf("Child Complete"); exit(0);
                           }
                           return 0;
}
```

Process Termination

Process are terminated on the following conditions

1. Normal exit.
2. Error exit.
3. Fatal error.
4. Killed by another process.

Example:

In Unix the normal exit is done by calling a *exit* system call. The process return data (output) to its parent process via the *wait* system call. *kill* system call is used to kill other process.

Home Work

HW #2:

1. Q. No. 1, 2 & 3 from the Textbook (Tanenbaum).
2. What are disadvantages of too much multiprogramming?
3. List the definitions of process.
4. For each of the following transitions between the process states, indicate whether the transition is possible. If it is possible, give an example of one thing that would cause it.
 - a) Running -> Ready
 - b) Running -> Blocked
 - c) Blocked -> Running

Reading: Section 2.2 of Textbook (Tanenbaum)

Threads

Reading: Section 2.2 of the Textbook (Tanenbaum)

What is Thread?

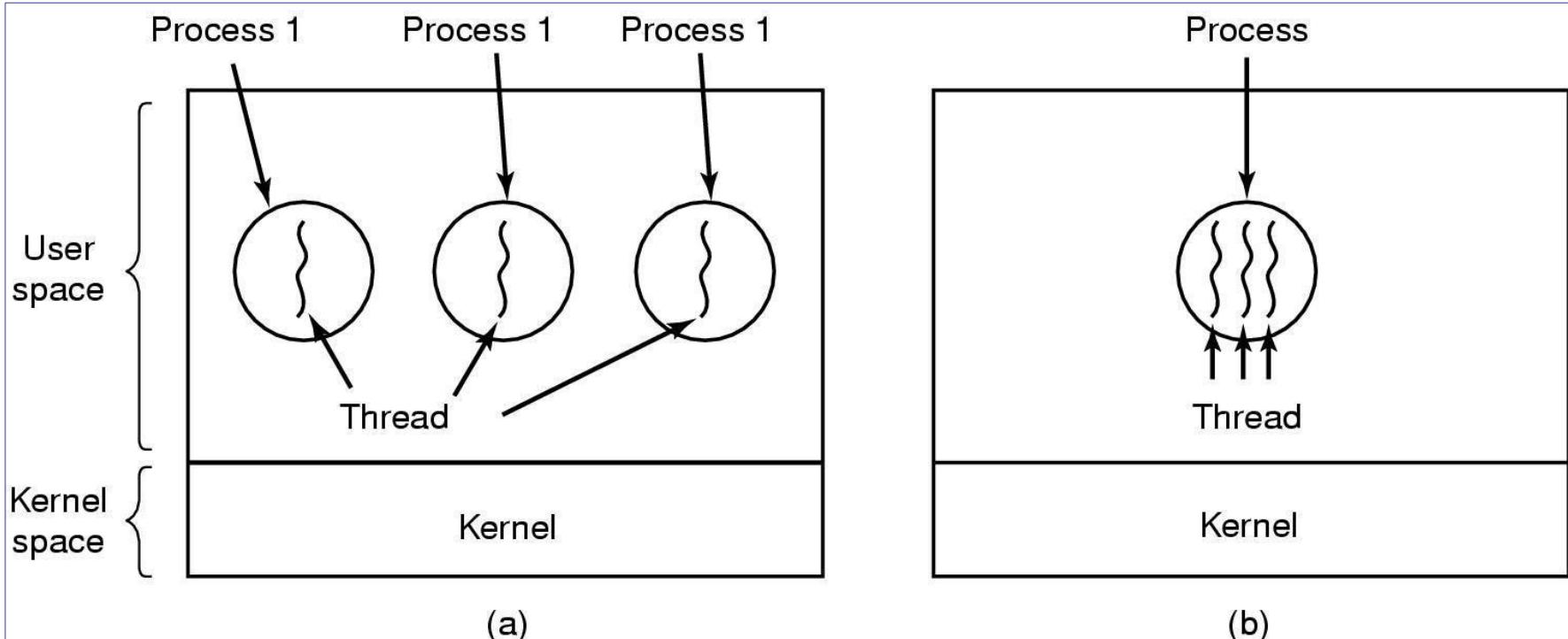
Threads, like process, are a mechanism to allow a program to do more than one thing at a time.

Conceptually, a thread (also called *lightweight process*) exists within a process (*heavyweight process*). Threads are a finer-grained unit of execution than processes

Threads

- Traditional threads has its own address space and a single thread of control.
- The term multithreading is used to describe the situation of allowing the multiple threads in same process.
- When multithreaded process is run on a single-CPU system, the threads take turns running as in the multiple processes.
- All threads share the same address space, global variables, set of open file, child processes, alarms and signals etc.

Threads



(a) Three process each with one thread (b) one process with three threads.

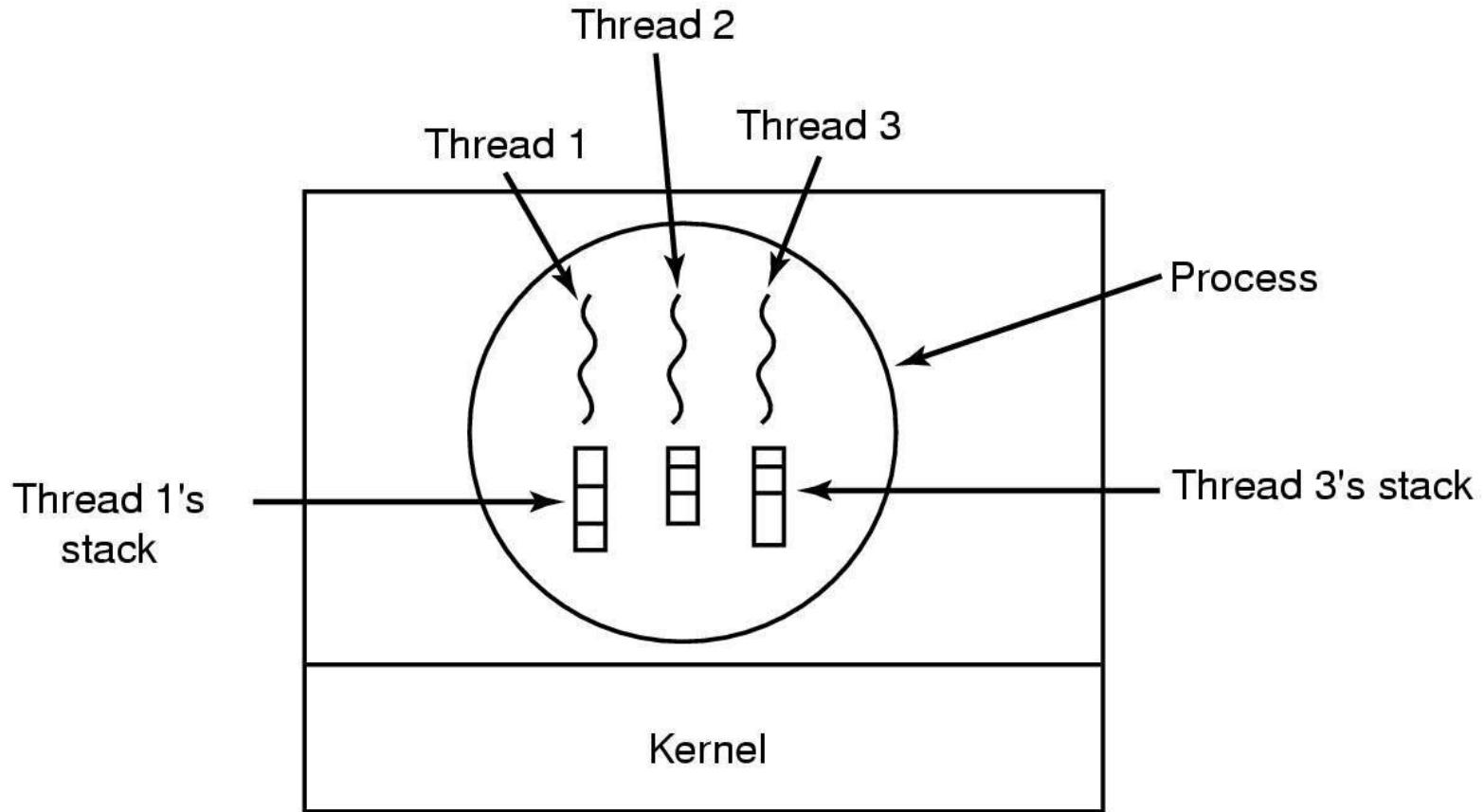
- *Figure (a)* organization is used when three processes are unrelated whereas *(b)* would be appropriate when the three threads are actually part of the same job and are actively and closely cooperating with each other.

Source: www.csitnepal.com

Threads

- Each thread maintain its own stack.

Threads



Example – Word processor

Threads

Needs to accept user input, display it on screen, spell check, auto save and grammar check.

- Implicit: Write code that reads user input, displays/formats it on screen, calls spell checked etc. while making sure that interactive response does not suffer.
- Threads: Use threads to perform each task and communicate using queues and shared data structures
- Processes: expensive to create and do not share data structures and so explicitly passed.

Others: Spreadsheet, Server for www, browser etc.

Advantages:

- Responsiveness.
- Resource sharing.
- Economy.

Problems: designing complexity.

Threads

In Unix system when fork create it copy all threads of parent to the child.

What happens if the thread of parent blocked ? When the line typed, do both thread get a copy of it?

Threads share the many data structure.

What happens if one thread closes a file while another is still reading from it?

Need complex scheduling operations

Users and Kernel Threads

User Threads:

Thread management done by user-level threads library.

- Implemented as a library
 - Library provides support for thread creation, scheduling and management with no support from the kernel.
 - Fast to create
 - If kernel is single threaded, blocking system calls will cause the entire process to block.
- Example:** POSIX Pthreads, Mach C-threads.

Kernel Threads:

Supported by the Kernel

- Kernel performs thread creation, scheduling and management in kernel space.
 - Slower to create and manage
 - 1. Blocking system calls are no problem
 - 2. Most OS's support these threads
- Examples:** WinX, Linux

Source: www.csitnepal.com

Home works

HW #3:

1. Q. 7, 8, 9 & 12 from the Textbook (Tanenbaum)
2. Dsribe how multithreading improve performance over a singled-threaded solution.
- 3.What are the two differences between the kernel level threads and user level threads? Which one has a better performance?
4. List the differences between processes and threads.
5. What resources are used when a thread is created? How do they differ from those used when a process is created?

Reading: Section 2.3 of Textbook (Tanenbaum)

InterProcess Communication

Reading: Section 2.3 from TextBook (Tanenbaum)

- How one process can pass the information to the another? –
- How to make sure two or more processes do not get into each other's way when engaging in critical activities?
- How to maintain the proper sequence when dependencies are presents?

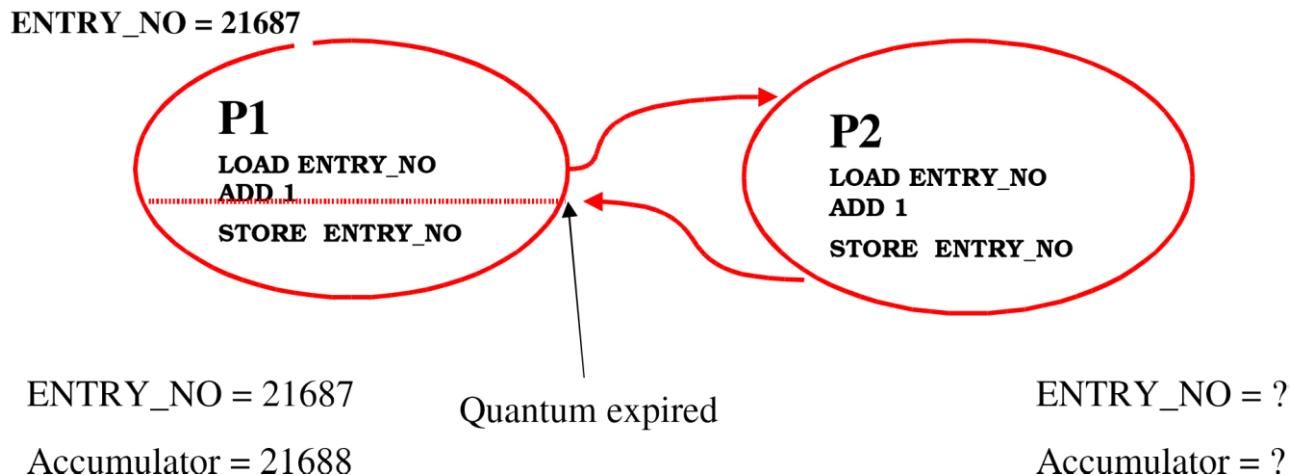
Thus, InterProcess Communication (IPC) and synchronization

IPC provide the mechanism to allow the processes to communicate and to synchronize their actions.

Race Conditions

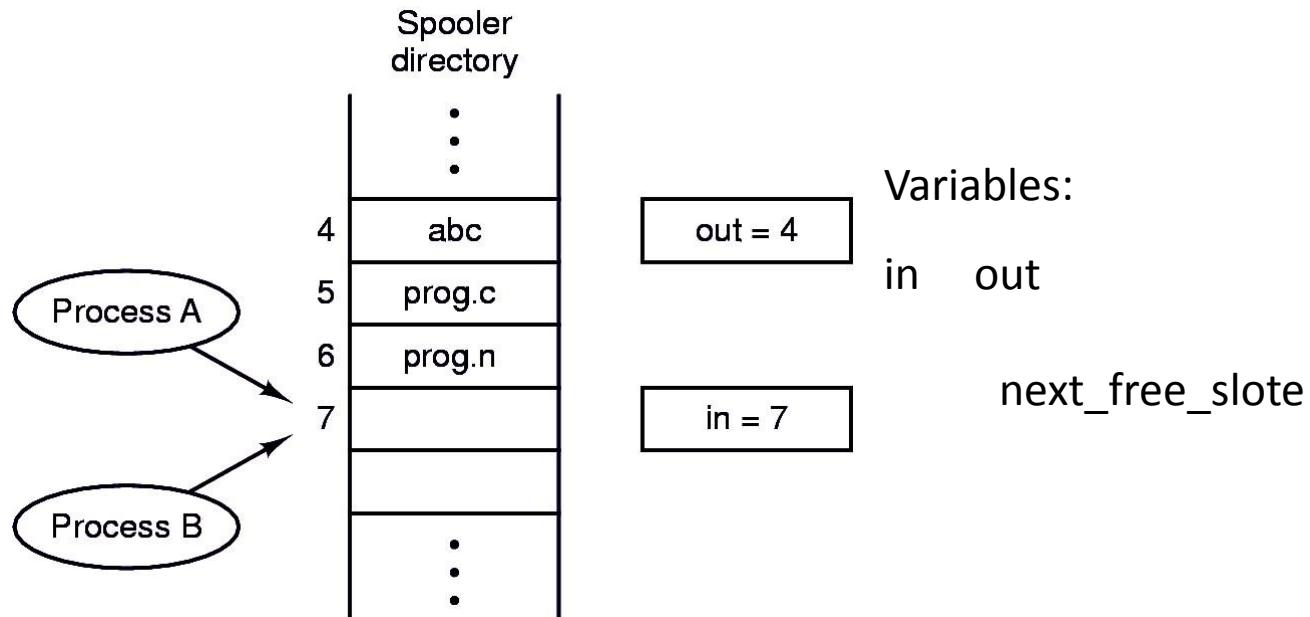
Scenario 1: Suppose two user working in a computer, there is a mechanism to monitor continuously the total number of lines each users have entered, each user entry is recorded globally, different process monitoring different users .

Each process : LOAD ENTRY_NO ADD 1
 STORE ENTRY_NO



Race Conditions

Scenario2: a print spooler: when a process wants to print a file, it enters the file name in a special spooler directory, another process, print daemon, periodically checks to see if there are any files to be printed, print them and removes their names from the directory.



Situation where two or more processes are reading or writing some shared data and the final result depends on who run precisely, are called race conditions

Mutual Exclusion

Possibilities of race:

- many concurrent process read the same data.
- one process reading and another process writing same data.
- two or more process writing same data.

Solution: prohibiting more than one process from reading writing the same data at the same time- **Mutual Exclusion**. Mutual Exclusion:

Some way of making sure that if one process is using a shared variables or files, the other process will be excluded from doing the same thing.

Critical Section

Problem: How to avoid race?

Fact: The part of the time, process is busy doing internal computations and other things that do not lead to the race condition.

Code executed by the process can be grouped into sections, some of which require access to shared resources, and other that do not. The section of the code that require access to shared resources are called critical section.

General structure of process P_i (other process P_j)

```
while(true){  
    entry_section  
    critical_section}
```

Critical Section

```
    exit_section  
    reminder_section  
}
```

- When a process is accessing a shared modifiable data, the process is said to be in critical section.
- All other processes (those access the same data) are excluded from their own critical region.
- All other processes may continue executing outside their CR.
- When a process leaves its critical section, then another processes waiting to enter its own CR should be allowed to proceed.

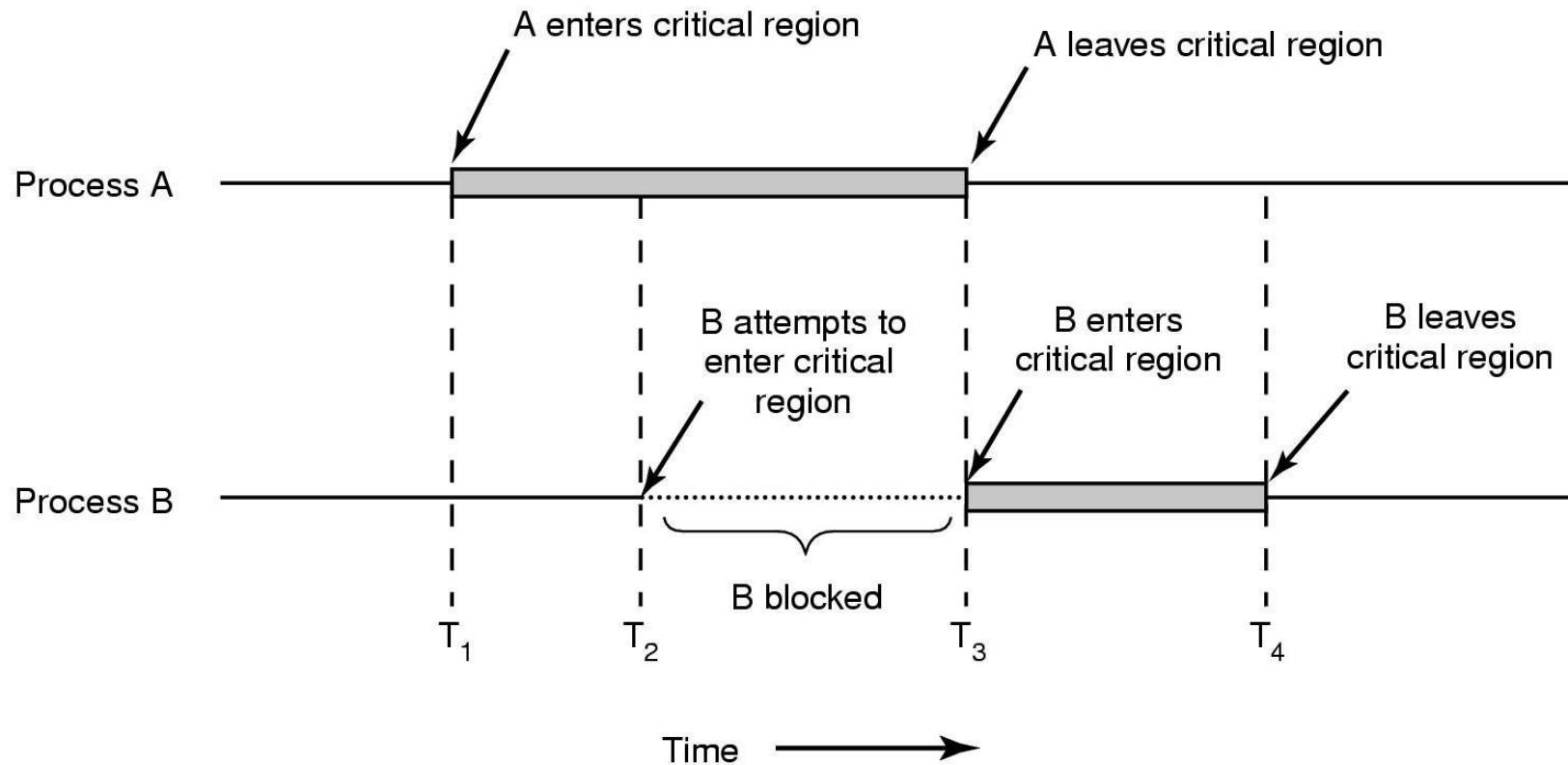
CR must satisfy the following conditions:

1. No two processes may be simultaneously inside their CRs (mutual exclusion).

Critical Section

2. No assumptions may be made about the speeds or number of CPUs.
3. No process running outside its CR may block other process.
4. No process should have to wait forever to enter its CR.

Critical Section



Mutual exclusion using critical regions

ME with Busy Waiting Interrupt Disabling

Each process disable all interrupts just after entering its CR and re-enable them just before leaving it.

No clock interrupt, no other interrupt, no CPU switching to other process until the process turn on the interrupt.

```
DisableInterrupt()  
// perform CR task  
EnableInterrupt()
```

Advantages:

Mutual exclusion can be achieved by implementing OS primitives to disable and enable interrupt.

Problems:

- allow the power of interrupt handling to the user.

- The chances of never turned on – is a disaster.
- it only works in single processor environment.

ME with Busy Waiting

Lock Variables

A single, shared (lock) variable, initially 0. When a process wants to enter its CR, it first test the lock. If the lock is 0, the process set it to 1 and enters the CR. If the lock is already 1, the process just waits until it becomes 0. Advantages: seems no problems. Problems:

problem like spooler directory; suppose that one process reads the lock and sees that it is 0, before it can set lock to 1, another process

ME with Busy Waiting

scheduled, enter the CR, set lock to 1 and can have two process at CR (violates mutual exclusion).

Strict Alternation

Processes share a common integer variable turn. If turn == i then process P_i is allowed to execute in its CR, if turn == j then process P_j is allowed to execute.

```
While (true){  While (true){  
    while(turn!=i); /*loop */    while(turn!=j); /*loop*/  
    critical_section();  critical_section();  
    turn = j;  turn = i;  
    noncritical_section();  noncritical_section();  
} }
```

Process P_i Process P_j

Advantages: Ensures that only one process at a time can be in its CR.

Problems: strict alternation of processes in the execution of the CR.

What happens if process i just finished CR and again need to enter CR and the process j is still busy at non-CR work? (violate condition 3)

ME with Busy Waiting Peterson's Algorithm

ME with Busy Waiting

```
#define FALSE 0
#define TRUE 1
#define N      2          /* number of processes */

int turn;                  /* whose turn is it? */
int interested[N];         /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;              /* number of the other process */

    other = 1 - process;    /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;         /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

ME with Busy Waiting Peterson's Algorithm

- Before entering its CR, each process call *enter_region* with its own process number, 0 or 1 as parameter.
- Call will cause to wait, if need be, until it is safe to enter.
- When leaving CR, the process calls *leave_region* to indicate that it is done and to allow other process to enter CR.

Advantages: Preserves all conditions.

Problems: difficult to program for n-processes system and less efficient.

ME with Busy Waiting

Think: *How it preserves all conditions?*

Hardware Solution - TSL

help from the hardware

- ▶ Test and Set Lock (TSL) instruction reads the contents of the memory word lock (shared variable) into the register and then stores nonzero value at the memory address lock.
- ▶ This automatically as in a un-interruptible time unit.
- ▶ The CPU executing TSL locks the memory bus to prohibit other CPUs from accessing memory until it is done.
- ▶ When lock is 0, any process may set it to 1 using the TSL instruction.

ME with Busy Waiting

- When it is done the process lock back to 0.

Hardware Solution - TSL

enter_region:

*TSL register, lock |copy lock to register and set lock to 1 CMP
register, #0 |was lock 0?*

*JNE enter_region |if it was non zero, lock was set, so loop
RET |return to caller; critical_region();*

leave_region:

*MOVE lock, #0 |store a 0 in lock
RET |return to caller noncritical_section();*

*Advantages: Preserves all condition, easier programming task and
improve system efficiency.*

Problems: difficulty in hardware design.

Busy Waiting Alternate?

Busy waiting:

When a process want to enter its CR, it checks to see if the entry is allowed, if it is not, the process just sits in a tight loop waiting until it is.

Waste of CPU time for NOTHING!

Possibility of *Sleep* and *Wakeup* pair instead of waiting.

Sleep causes to caller to block, until another process wakes it up.

Sleep and Wakeup Producer-Consumer Problem

- Two process share a common, fixed sized buffer.
- Suppose one process, producer, is generating information that the second process, consumer, is using.
- Their speed may be mismatched, if producer insert item rapidly, the buffer will full and go to the sleep until consumer consumes some item, while consumer consumes rapidly, the buffer will empty and go to sleep until producer put something in the buffer.

Producer-Consumer Problem

Sleep and Wakeup

```
#define N = 100 /* number of slots in the buffer*/
int count = 0; /* number of item in the buffer*/
void producer(void) void consumer(void)
{
    {
        int item; int item;
        while(TRUE){ /* repeat forever*/ while(TRUE){ /* repeat forever*/
            item = produce_item(); /*generate next item*/ if(count == 0)sleep(); /* if buffer is full go to sleep*/
            if(count == N) sleep(); /*if buffer is empty go to sleep*/
            item = remove_item(); /*take item out
            insert_item(item); /* put item in buffer */ of buffer*/ count = count +1;
            /*increment count */ count = count -1; /*decrement count*/ if (count == 1)
            wakeup(consumer); if (count == N-1)wakeup(producer);
            /* was buffer empty*/ /*was buffer full ?*/
        } consume_item(); /*print item*/
    }
}
```

Producer-Consumer Problem

Problem:

Sleep and Wakeup

-leads to race as in spooler directory.

-What happen if

When the buffer is empty, the consumer just reads count and quantum is expired, the producer inserts an item in the buffer, increments count and wake up consumer. The consumer not yet asleep, so the wakeup signal is lost, the consumer has the count value 0 from the last read so go to the sleep. Producer keeps on producing and fill the buffer and go to sleep, both will sleep forever.

Think: If we were able to save the wakeup signal that was lost.....

Semaphores

E. W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups, called a semaphore.

It could have the value 0, indicating no wakeups were saved, or some positive value if one or more wakeups were pending.

Operations: *Down* and *Up* (originally he proposed *P* and *V* in Dutch and sometimes known as wait and signal)

Down: checks if the value greater than 0 yes- decrement
the value (i.e. Uses one stored wakeup) and continues.

 No- process is put to sleep without completing down.

 - Checking value, changing it, and possibly going to sleep, is all done as single action.

Up: increments the value; if one or more processes were sleeping, unable to complete earlier down operation, one of them is chosen and is allowed to complete its down.

Semaphores

```
typedef int semaphore S;  
void down(S)  
{ if(S> 0) S--;  
else sleep();  
}  
void up(S)  
{if(one or more processes are sleeping on S)  
one of these process is proceed;  
else S++;  
} while(TRUE){  
down(mutex)  
critical_region();  
up(mutex);  
noncritical_region();  
}
```

Semaphores

Producer-Consumer using Semaphore

```
#define N 100 /*number of slots in buffer*/ typedef int semaphore;  
/*defining semaphore*/ semaphore mutex = 1; /* control access  
to the CR */ semaphore empty = N /*counts empty buffer slots*/  
semaphore full = 0; /*counts full buffer slots*/  
  
void producer(void)  
{  
    int item;  
    while(TRUE){ /*repeat forever */  
        item = produce_item(); /*generate something */  
        down(empty); /*decrement empty count*/ down(mutex);  
        /*enter CR */  
        insert_item(); /* put new item in buffer*/ up(mutex); /* leave  
CR*/
```

```
        up(full); /*increment count of full slots*/ }  
    }
```

Semaphores

Producer-Consumer using Semaphore

```
void consumer(void)  
{  
    int item;  
    while(TRUE){ /*repeat forever*/ down(full); /*decrement full  
    count */ down(mutex); /*enter CR*/  
        item = remove_item(); /*take item from buffer*/  
        up(mutex); /*leave CR*/  
        up(empty); /*increment count of empty slots*/ consume_item();  
        /*print item*/ }  
}
```

Use of Semaphore

1. *To deal with n-process critical-section problem.*

The n processes share a semaphore, (e. g. mutex) initialized to 1.

2. *To solve the various synchronizations problems.*

For example two concurrently running processes: P1 with statement S1 and P2 with statement S2, suppose, it required that S2 must be executed after S1 has completed. This problem can be implemented by using a common semaphore, synch, initialized to 0.

```
P1: S1;  
    up(synch);  
P2: down(synch);  
    S2;
```

Criticality Using Semaphores

All process share a common semaphore variable mutex, initialize to 1. Each process must execute down(mutex) before entering CR, and up(mutex) afterward. *What happens when this sequence is not observed?*

1. When a process interchange the order of *down* and *up* operation: causes the multiple processes in CR simultaneously.
2. When a process replace *up* by *down*: causes the dead lock.
3. When a process omits *down* or *up* or both: violated mutual exclusion and deadlock occurs.

A subtle error is capable to bring whole system grinding halt!!

Monitors

Higher level synchronization primitive.

A monitor is a programming language construct that guarantees appropriate access to the CR. It is a collection of procedures, variables , and data structures that are all grouped together in a special kinds of module or package.

Processes that wish to access the shared data, do through the execution of monitor functions.

Only one process can be active in a monitor at any instant.

Compiler level management.

Monitor monitor_name

```
{  
    shared variable declarations;  
    procedure p1(){ ..... }  
    procedure p2(){ .... }  
    .....  
    procedure pn(){ ... } {initialization  
    code;}  
}
```

Producer Consumer Using Monitors

Monitor ProducerConsumer

```
{Void producer(){ int  
count;while(TRUE){  
    condition full, empty;    item =  
ProcedureConsumer.insert(item);  
insert_item(item);}  
    count++;void consumer(){  
if (count ==1) signal(empty)  
}    item =  
remove(){consume_item();  
if(count ==0) wait(empty);  
remove_item(); }    count--;  
if(count ==N-1) signal(full);  
}  
count =0;  
};
```

```
produce_item(); void insert(int item){  
if (count == N ) wait(full);    }
```

```
while(TRUE){  
ProduceConsumer.remove(); void
```

Problems with

Monitors

1. Lack of implementation in most commonly used programming languages.

How to implement in C?

2. Both semaphore and monitors are used to hold mutual exclusion in multiple CPUs that all have a common memory, but in distributed system consisting multiple CPUs with its own private memory, connected by LAN, none of these primitives are applicable.

Semaphores are too low level and monitors are not usable except in a few programming language.

Message Passing

With the trend of distributed operating system, many OS are used to communicate through Internet, intranet, remote data processing etc.

Interprocess communication based on two primitives: send and receive.

send(destination, &message);

receive(source, &message);

The send and receive calls are normally implemented as operating system calls accessible from many programming language environments.

Producer-Consumer with Message Passing

```
#define N 100          /*number of slots in the buffer*/ void  
producer(void)
```

```
{ int item;
    message m;          /*message buffer*/ while (TRUE){
    item = produce_item(); /*generate something */
    receive(consumer, &m); /*wait for an empty to arrive*/
    build_message(&m, item); /*construct a message to send*/
    send(consumer, &m); }

void consumer(void)
{
    int item;
    message m;
    for(i = 0; i<N; i++) send(producer, &m); /*send N empties*/ while(TRUE){
        receive(producer, &m);           /* get message containing item*/
        item = extract_item(&m);        /* extract item from message*/
        send(producer, &m);            /* send back empty reply*/
        consume_item(item);           /*do something with item*/ }}
```

Message Passing

No shared memory.

Messages sent but not yet received are buffered automatically by OS, it can save N messages.

The total number of messages in the system remains constant, so they can be stored in given amount of memory known in advance.

Implementing Message Passing:

Direct addressing: provide ID of destination.

Indirect addressing: Send to a mailbox.

Mail box: It is a message queue that can share by multiple senders and receivers, senders sends the message to the mailbox while the receiver pick ups the message from the mailbox.

Classical IPC Problems

The Dining Philosophers Problem

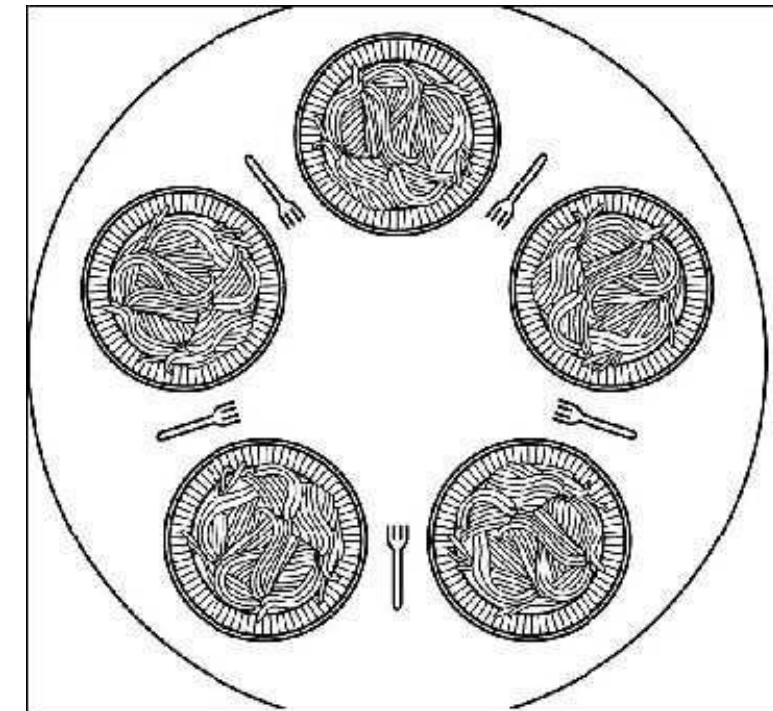
Scenario: consider the five philosophers are seated in a common round table for their lunch, each philosopher has a plate of spaghetti and there is a fork between two plates, a philosopher needs two forks to eat it. They alternates thinking and eating.

What is the solution (program) for each philosopher that does what is supposed to do and never got stuck?

Solution

Attempt 1: When the philosopher is hungry it picks up a fork and wait for another fork, when get, it eats for a while and put both forks back to the table.

Problem: What happen, if all five philosophers take their left fork simultaneously?



Classical IPC Problems The Dining Philosophers Problem

Attempt 2: After taking the fork, checks for right fork. If it is not available, the philosopher puts down the left one, waits for some time, and then repeats the whole process.

Problem: What happen, if all five philosophers take their left fork simultaneously?

Attempt 3: Using semaphore, before starting to acquire a fork he would do a down on mutex, after replacing the forks, he would do up on mutex.

Problem: adequate but not perfect: only one philosopher can be eating at any instant.

Attempt 4: Using semaphore for each philosopher, a philosopher move only in eating state if neither neighbor is eating. -*perfect solution.*

Read:

Readers and writer, sleeping barber and Cigarette-smokers.

Home Works

HW #4

1. Textbook (Tanenbaum): 18, 19, 20, 21, 22, 23.
2. What is the meaning of busy waiting? What other kinds of waiting are in OS? Compare each types on their applicability and relative merits.
3. Show the Peterson's algorithm preserve mutual exclusion, indefinite postponement and dead lock.
4. Compare the use of monitor and semaphore operations.

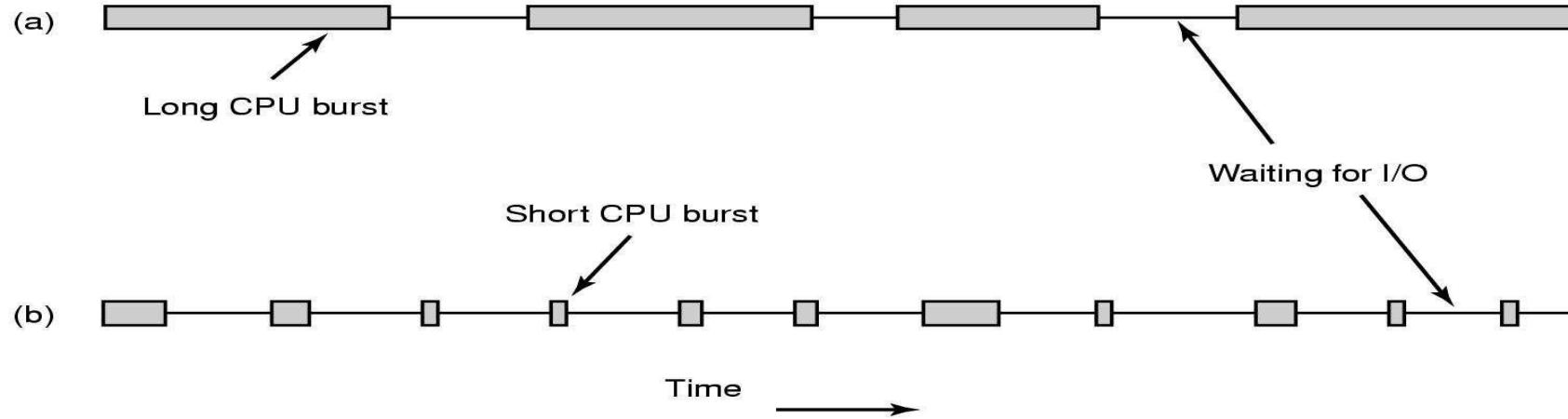
Reading: Section 2.5 of Textbook (Tanenbaum)

Scheduling

Which process is given control of the CPU and how long?

By switching the processor among the processes, the OS can make the computer more productive.

Scheduling CPU-I/O Burst



Process execution consists of a cycle of CPU execution and I/O wait.

Process execution begins with a CPU burst that is followed by I/O burst, then another CPU burst, then another I/O burst.....

CPU-bound: Processes that use CPU until the quantum expire.

I/O-bound: Processes that use CPU briefly and generate I/O request.

CPU-bound processes have a long CPU-burst while I/O-bound processes have short CPU burst.

Key idea: when I/O bound process wants to run, it should get a chance quickly.

Scheduling When to Schedule

1. When a new process is created.
2. When a process terminates.
3. When a process blocks on I/O, on semaphore, waiting for child termination etc.

4. When an I/O interrupt occurs.
5. When quantum expires.

Preemptive vs. Nonpreemptive Scheduling

Nonpreemptive:

- Once a process has been given the CPU, it runs until blocks for I/O or termination.
- Treatment of all processes is fair.
- Response times are more predictable.
- Useful in real-time system.

Shorts jobs are made to wait by longer jobs - no priority Preemptive:

- Processes are allowed to run for a maximum of some fixed time.
- Useful in systems in which high-priority processes requires rapid attention.
- In timesharing systems, preemptive scheduling is important in guaranteeing acceptable response times.
- High overhead.

Role of Dispatcher vs. Scheduler

Dispatcher

- Low level mechanism.
- **Responsibility:** Context switch
- Save execution state of old process in PCB.
- Load execution state of new process from PCB to registers.
- Change the scheduling state of the process (running, ready, blocked)
- Switch from kernel to user mode.

Scheduler

- Higher-level policy.
- **Responsibility:** Which process to run next.

Scheduling Criteria

The scheduler is to identify the process whose selection will results the best possible system performance.

Criteria for comparing scheduling algorithms:

CPU Utilization Balance Utilization

Throughput Turnaround Time Waiting Time

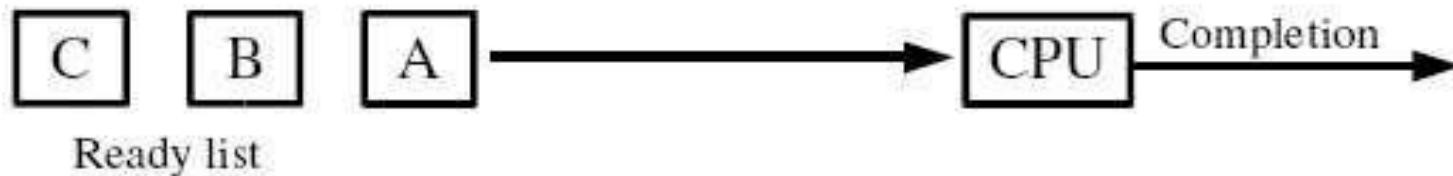
Response Time

Predictability Fairness Priorities

The scheduling policy determine the important of each criteria.
The scheduling algorithms should designed to optimizes maximum possible criteria.

Scheduling Algorithms

First-Come First-Serve (FCFS)



Processes are scheduled in the order they are received.

Once the process has the CPU, it runs to completion -Nonpreemptive.

Easily implemented, by managing a simple queue or by storing time
the process was received.

Fair to all processes. **Problems:**

- No guarantee of good response time.

- Large average waiting time.

- Not applicable for interactive system.

Scheduling Algorithms

Shortest Job First (SJF)

The processing times are known in advanced.

SJF selects the process with shortest expected processing time. In case of the tie FCFS scheduling is used.

The decision policies are based on the CPU burst time. **Advantages:**
Reduces the average waiting time over FCFS.
Favors shorts jobs at the cost of long jobs.

Problems:

Estimation of run time to completion. Accuracy?
Not applicable in timesharing system.

Scheduling Algorithms

SJF -Performance

Scenario: Consider the following set of processes that arrive at time 0, with length of CPU-burst time in milliseconds.

Processes	Burst time
P1	24
P2	3
P3	3

if the processes arrive in the order P1, P2, P3 and are served in FCFS order,



The average waiting time is $(0 + 24 + 27)/3 = 17$.

if the processes are served in SJF



The average waiting time is $(6 + 0 + 3)/3 = 3$.

Scheduling Algorithms

Shortest-Remaining-Time-First (SRTF)

Preemptive version of SJF.

Any time a new process enters the pool of processes to be scheduled, the scheduler compares the expected value for its remaining processing time with that of the process currently scheduled. If the new process's time is less, the currently scheduled process is preempted.

Merits:

Low average waiting time than SJF.

Useful in timesharing. **Demerits:**

Very high overhead than SJF.

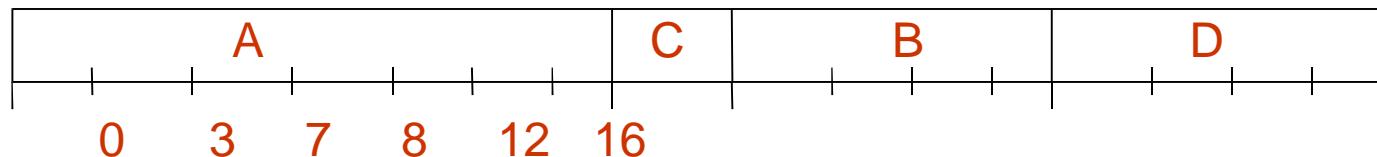
Requires additional computation.

Favors short jobs, longs jobs can be victims of starvation.

Concept Scheduling Algorithms SRTF-Performance

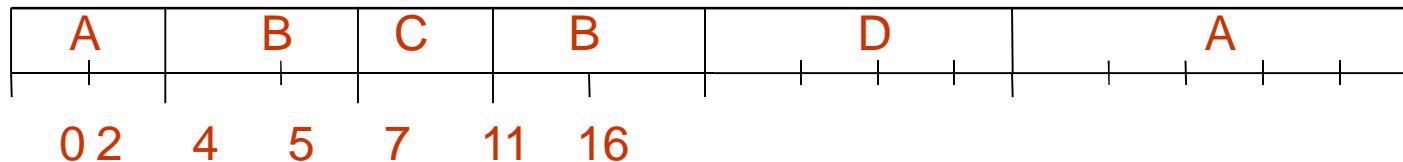
Scenario: Consider the following four processes with the length of CPU-burst time given in milliseconds:

Processes	Arrival Time	Burst Time
A	0.0	7
B	2.0	4
C	4.0	1
D	5.0	4 SJF:



$$\text{Average waiting time} = (0 + 6 + 3 + 7)/4 = 4$$

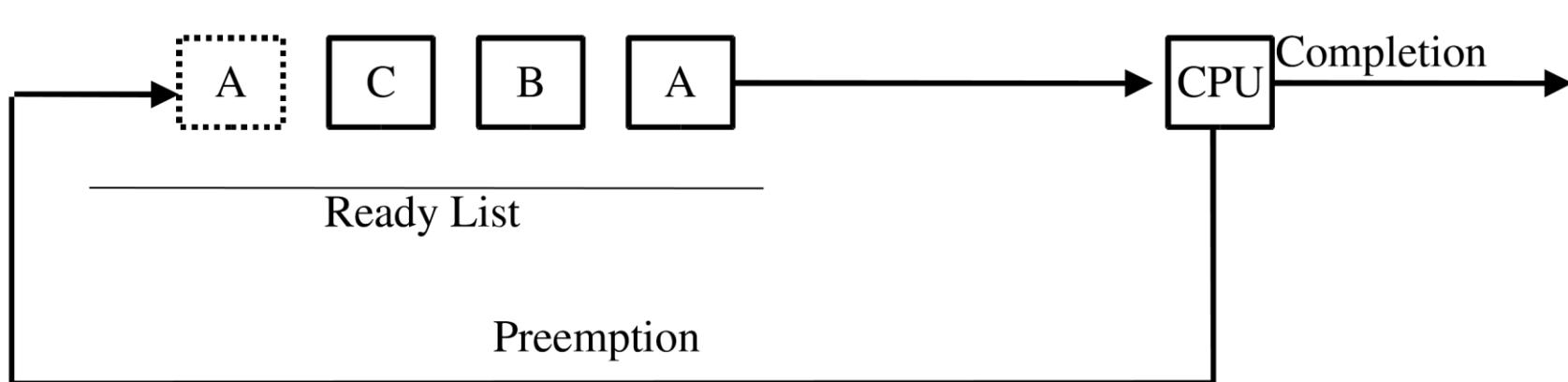
SRTF:



$$\text{Average waiting time} = (9 + 1 + 0 + 2)/4 = 3$$

Scheduling Algorithms

Round-Robin (RR)



Preemptive FCFS.

Each process is assigned a time interval (quantum), after the specified quantum, the running process is preempted and a new process is allowed to run.

Preempted process is placed at the back of the ready list.

Advantages:

Fair allocation of CPU across the process.

Used in timesharing system.

Low average waiting time when process lengths very widely.

Scheduling Algorithms

RR-Performance

- Poor average waiting time when process lengths are identical.

Imagine 10 processes each requiring 10 msec burst time and 1msec quantum is assigned.

RR: All complete after about 100 times.

FCFS is better! (About 20% time wastages in context-switching).

- Performance depends on quantum size. **Quantum size:**

If the quantum is very large, each process is given as much time as needs for completion; RR degenerate to FCFS policy.

If quantum is very small, system busy at just switching from one process to another process, the overhead of context-switching causes the system efficiency degrading.

Optimal quantum size?

Key idea: 80% of the CPU bursts should be shorter than the quantum.

20-50 msec reasonable for many general processes.

Scheduling Algorithms

Example of RR with Quantum = 20

Process	Burst Time
A	53
B	17
C	68
D	24

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better *response*.

Scheduling Algorithms

Priority

Each process is assigned a priority value, and runnable process with the highest priority is allowed to run.

FCFS or RR can be used in case of tie.

To prevent high-priority process from running indefinitely, the scheduler may decrease the priority of the currently running process at each clock tick.

Assigning Priority Static:

Some processes have higher priority than others.

Problem: Starvation.

Dynamic:

Priority chosen by system.

Decrease priority of CPU-bound processes.

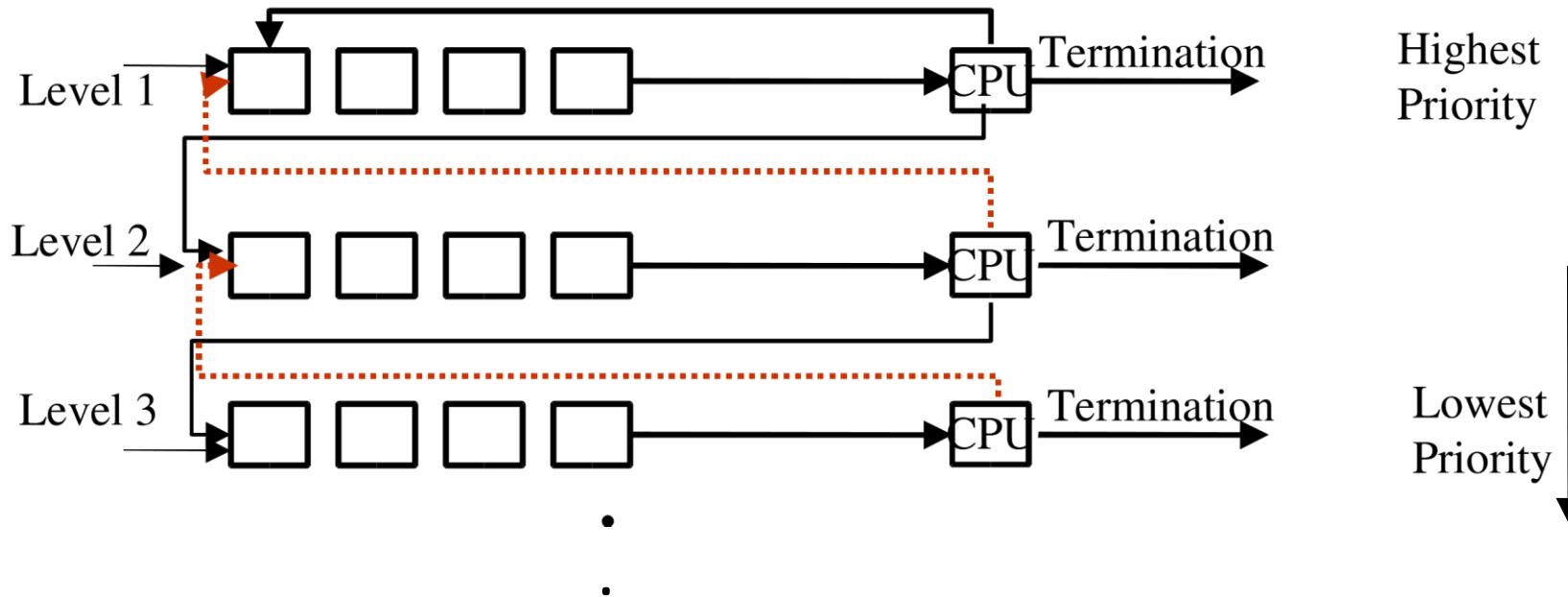
Increase priority of I/O-bound processes.

Many different policies possible.....

E. g.: priority = (time waiting + processing time)/processing time.

Scheduling Algorithms

Multilevel-Feedback-Queues (MFQ)



MFQ implements multilevel queues having different priority to each level (here lower level higher priority), and allows a process to move between the queues. If the process use too much CPU time, it will be moved to a lower-priority queue. Each lower-priority queue larger the quantum size. This leaves the I/O-bound and interactive processes in the high priority queue.

Scheduling Algorithms

MFQ-Example

Consider a MFQ scheduler with three queues numbered 1 to 3, with quantum size 8, 16 and 32 msec respectively.

The scheduler execute all process in queue 1, only when queue 1 is empty it execute process in queue 2 and process in queue 3 will execute only if queue 1 and queue 2 are empty.

A process first enters in queue 1 and execute for 8 msec. If it does not finish, it moves to the tail of queue 2.

If queue 1 is empty the processes of queue 2 start to execute in FCFS manner with 16 msec quantum. If it still does not complete, it is preempted and move to the queue 3.

If the process blocks before using its entire quantum, it is moved to the next higher level queue.

Home Works

HW#5

1. Textbook (Tanenbaum) 35, 36, 37, 38, 39 & 40.
2. For the processes listed in following table, draw a Gantt chart illustrating their execution using:
 - (a) First-Come-First-Serve.
 - (b) Short-Job-First.
 - (c) Shortest-Remaining-Time-Next.
 - (d) Round-Robin (quantum = 2).
 - (e) Round-Robin (quantum = 1).

Processes	Arrival Time	Burst Time
A	0.00	4
B	2.01	7
C	3.01	2
D	3.02	2

Scheduling Algorithms

- i) What is the turnaround time?
- ii) What is average waiting?

Deadlock

A process in a multiprogramming system is said to be in dead lock if it is waiting for a particular event that will never occur.

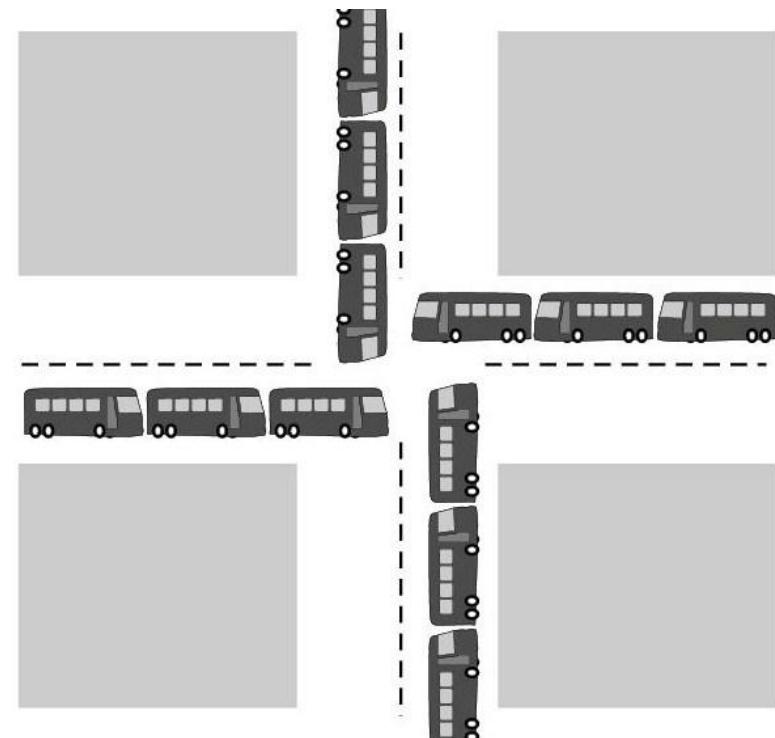
Deadlock Example

- All automobiles trying to cross.

- Traffic Completely stopped.
- Not possible without backing some.

A Traffic Deadlock

Resource Deadlock



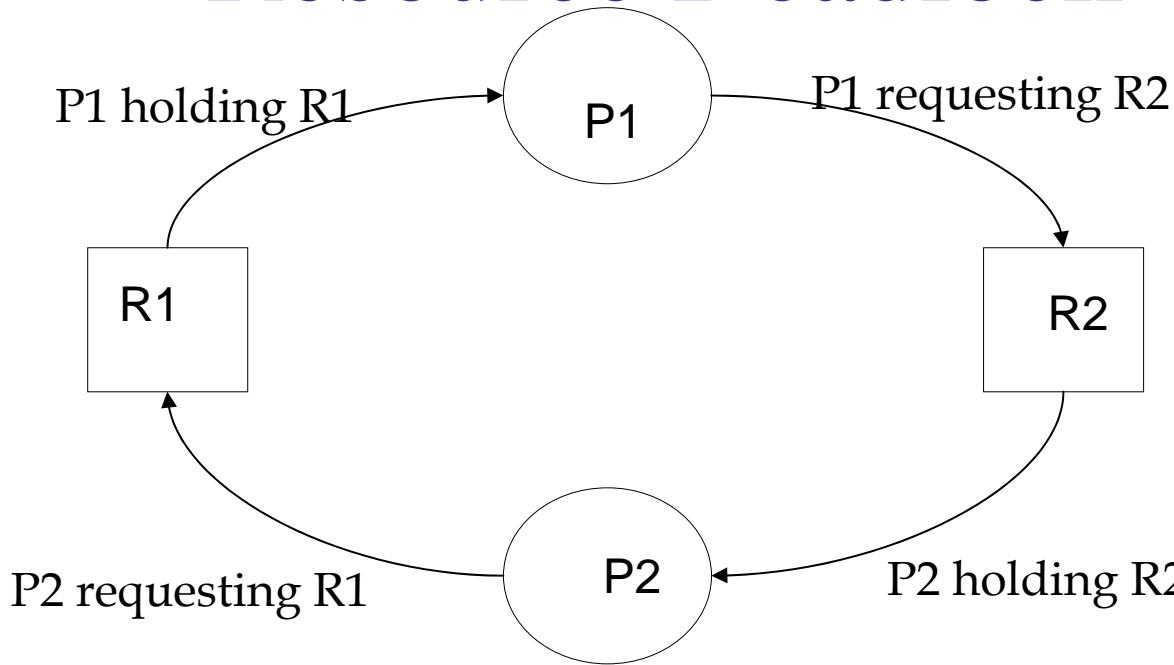
A process request a resource before using it, and release after using it.

1. Request the resource.
2. Use the resource.
3. Release the resource.

If the resource is not available when it is requested, the requesting process is forced to wait.

Most deadlocks in OS developed because of the normal contention for dedicated resources.

Resource Deadlock



- Process P1 holds resource R1 and needs resource R2 to continue; Process P2 holds resource R2 and needs resource R1 to continue – deadlock.

Key: Circular wait - deadlock

Conditions for Deadlock

1. *Mutual Exclusion*: Process claims exclusive control of resources they require.
2. *Hold and Wait*: Processes hold resources already allocated to them while waiting for additional resources.
3. *No preemption*: Resources previously granted can not be forcibly taken away from the process.
4. *Circular wait*: Each process holds one or more resources that are requested by next process in the chain.

A deadlock situation can arise if all four conditions hold simultaneously in the system.

Handling Deadlocks

- Deadlock handling strategies:
- We can use a protocol to *prevent* or *avoid* deadlocks, ensuring that the system never enter a deadlock state.
- We can allow the system to enter a deadlock state, *detect* it, and *recover*.

- We can ignore the problem all together, and pretended that deadlock never occur in the system.

Deadlock prevention

Fact: *If any one of the four necessary conditions is denied, a deadlock can not occur.*

Denying Mutual Exclusion:

- Sharable resources do not require mutually exclusive access such as read only shared file.

- Problem: Some resources are strictly nonsharable, mutually exclusive control required.

We can not prevent deadlock by denying the mutual exclusion

Deadlock prevention

Denying Hold and Wait:

Resources grant on *all or none* basis;

If all resources needed for processing are available then granted and allowed to process.

If complete set of resources is not available, the process must wait set available.

While waiting, the process should not hold any resources.

Problem:

Low resource utilization.

Starvation is possible.

Deadlock prevention

Denying No-preemption:

When a process holding resources is denied a request for additional resources, that process must release its held resources and if necessary, request them again together with additional resources.

Problem:

When process releases resources the process may loose all its works to that point .

Indefinite postponement or starvation.

Deadlock prevention

Denying Circular Wait:

All resources are uniquely numbered, and processes must request resources in linear ascending order.

The only ascending order prevents the circular.

Problem:

Difficult to maintain the resource order; dynamic update in addition of new resources. Indefinite postponement or starvation.

Deadlock Avoidance

Avoiding deadlock by careful resource allocation.

Decide whether granting a resource is safe or not, and only make the allocation when it is safe.

Need extra information in advanced, maximum number of resources of each type that a process may need.

The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

Has Max

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

Has Max

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

Has Max

	Has	Max
A	3	9
B	0	-
C	2	7

Free: 5

Has Max

	Has	Max
A	3	9
B	0	-
C	7	7

Free: 0

Has Max

	Has	Max
A	3	9
B	0	-
C	0	-

Free: 7

Safe and Unsafe States

A state is said to be safe if it is not deadlocked and there is a some scheduling order in which every process can run to completion.

(a)

(b)

(c)

(d)

(e)

State in (a) is safe

Deadlock Avoidance

Deadlock Avoidance

Safe and Unsafe States

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

(d)

State in (b) is not safe.

In a safe state, system can guarantee that all processes will finish – no deadlock occur; from an unsafe state, no such guarantee can be given – deadlock may occur.

Banker's Algorithm

Operating System Concepts Process Management

Models on the way of banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.

When a process request the set of resources, the system determine whether the allocation of these resources will left the system in safe state, if it will the resources are allocated, otherwise process must wait until some other process release enough resources.

Data structures: Available, Max, Allocation & Need.
need = max – allocation.

Deadlock Avoidance Banker's Algorithm

```
finishi= False;  
  
for each process if (needi <= available &&  
finishi = false)  
    continue;  
  
available = available + allocation;  
  
finishi = True;
```

```
for each process if (finishi = True) system  
is in safe state.  
  
if (requesti <= needi)  
  
    if (requesti <= available) { available =  
        available- requesti; allocationi =  
        allocationi + requesti; needi =  
        needi-requesti;  
  
    }  
  
else wait;
```

Deadlock Avoidance

else

error;

Banker's Algorithm

allo. max allo. max allo. max

Deadlock Avoidance

A	0	6
B	0	5
C	0	4
D	0	7

Available = 10

A	1	6
B	1	5
C	2	4
D	4	7

Available = 2

A	1	6
B	2	5
C	2	4
D	4	7

Available = 1

Which one is unsafe?

Deadlock Avoidance Problem with Banker's Algorithm

Algorithms requires fixed number of resources, some processes dynamically changes the number of resources.

Algorithms requires the number of resources in advanced, it is very difficult to predict the resources in advanced.

Deadlock Avoidance

Algorithms predict all process returns within finite time, but the system does not guarantee it.

Deadlock Detection and Recovery

Instead of trying to prevent or avoid deadlock, system allow to deadlock to happen, and recover from deadlock when it does occur.

Hence, the mechanism for deadlock detection and recovery from deadlock required.

Deadlock Detection

Consider the following scenario:

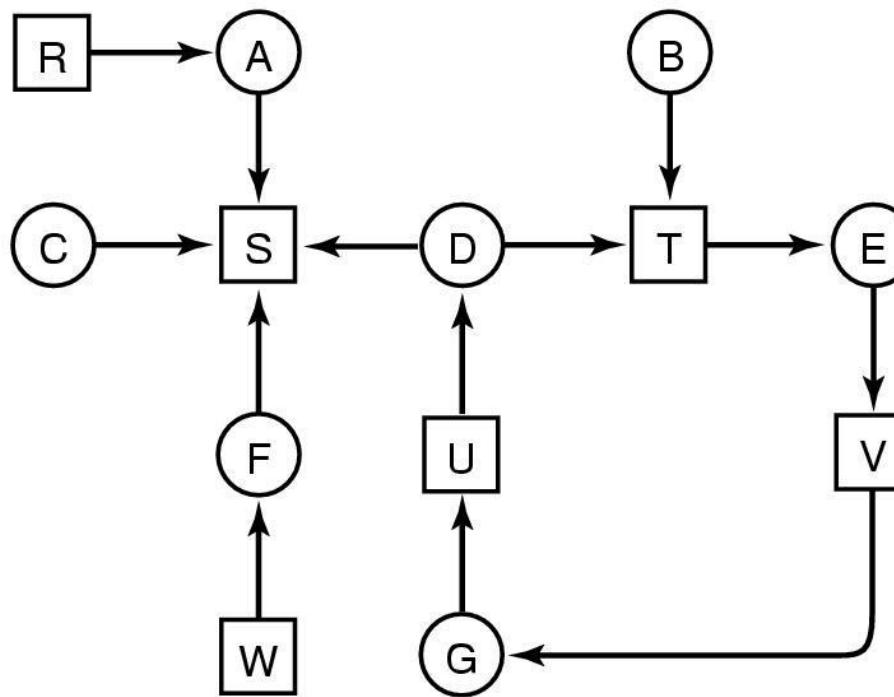
a system with 7 processes (A – G) , and 6 resources (R – W) are in following state.

1. Process A holds R and wants S.
2. Process B holds nothing but wants T.
3. Process C holds nothing but wants S.
4. Process D holds U and wants S and T.
5. Process E holds T and wants V.

6. Process F holds W and wants S. 7 . Process G holds V and wants U.

Is there any deadlock situation?

Deadlock Detection



Resource graph

How to detect the cycle in directed graph?

Deadlock Detection Algorithm

List L;

Boolean cycle = False;

for each node

L = Φ /* initially empty list*/ add node to L; for each
node in L for each arc if (arc[i] = true)

 add corresponding node to L;

 if (it has already in list)

 cycle = true;

 print all nodes between these nodes;

 exit;

```
else if (no such arc) remove  
    node from L;  
    backtrack;
```

Recovery from Deadlock

What do next if the deadlock detection algorithm succeed? – recover from deadlock.

By Resource Preemption

Preempt some resources temporarily from a processes and give these resources to other processes until the deadlock cycle is broken.

Problem:

Depends on the resources.

Need extra manipulation to suspend the process.

Difficult and sometime impossible.

Recovery from Deadlock

By Process Termination

Eliminating deadlock by killing one or more process in cycle or process not in cycle.

Problem:

If the process was in the midst of updating file, terminating it will leave file in incorrect state.

Key idea: we should terminate those processes the termination of which incur the minimum cost.

Ostrich Algorithm

Fact: there is no good way of dealing with deadlock.

Ignore the problem altogether.

For most operating systems, deadlock is a rare occurrence;

So the problem of deadlock is ignored, like an ostrich sticking its head in the sand and hoping the problem will go away.

Home Works

HW #6

1. Q. 1, 2, 3, 4, 6, 12, 14, 15, 20 & 22 from Textbook (Tanenbaum)
2. Distinguish indefinite postponement and Deadlock.
2. State the conditions necessary for deadlock to exist. Give reason, why all conditions are necessary.
3. A system has two processes and three identical resources. Each process needs a maximum of two resources. Is deadlock possible? Explain your answer.

4. Show that four necessary conditions hold in traffic deadlock example. State a simple rule that will avoid deadlock in traffic system.

Virtual Memory

Reading: Section 4.3 of Textbook (Tanenbaum)

Virtual memory is a concept that is associated with ability to address a memory space much larger than that the available physical memory.

The basic idea behind the virtual memory is that the combined size of the program, data, and stack may exceed the amount of physical memory available for it. The OS keeps those part of the program currently in use in main memory, and the rest on the disk.

Virtual Memory

Virtual storage is not a new concept, this concept was devised by Fotheringham, 1961 and used in Atlas computer system.

But the common use in OS is the recent concept, all microprocessor now support virtual memory.

Virtual memory can be implemented by two most commonly used methods : *Paging* and *Segmentation or mix of both.*

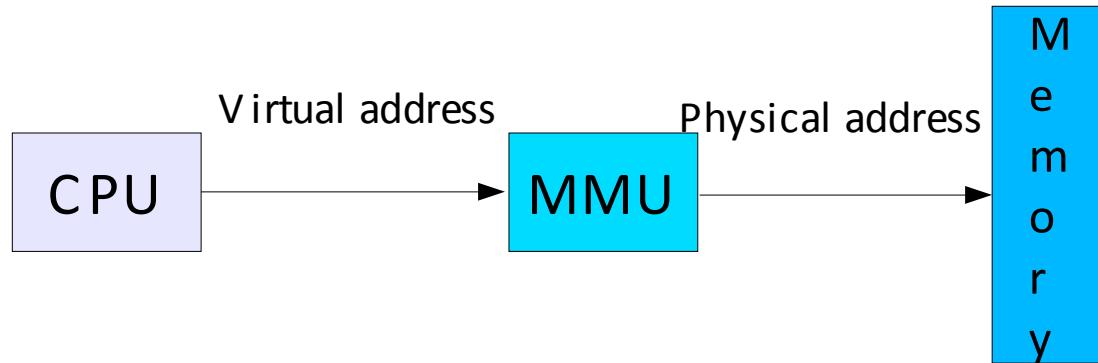
Virtual Memory

Virtual address space vs. Physical address space

The set of all virtual (logical) addresses generated by a program is a *virtual address space*; the set of all physical addresses corresponding to these virtual addresses is a *physical address space*.

MMU

The run time mapping from virtual address to physical address is done by hardware devices called *memory-management-unit (MMU)*.



Paging

The virtual address space (process) is divided up into fixed sized blocks called pages and the corresponding same size block in main memory is called frames.

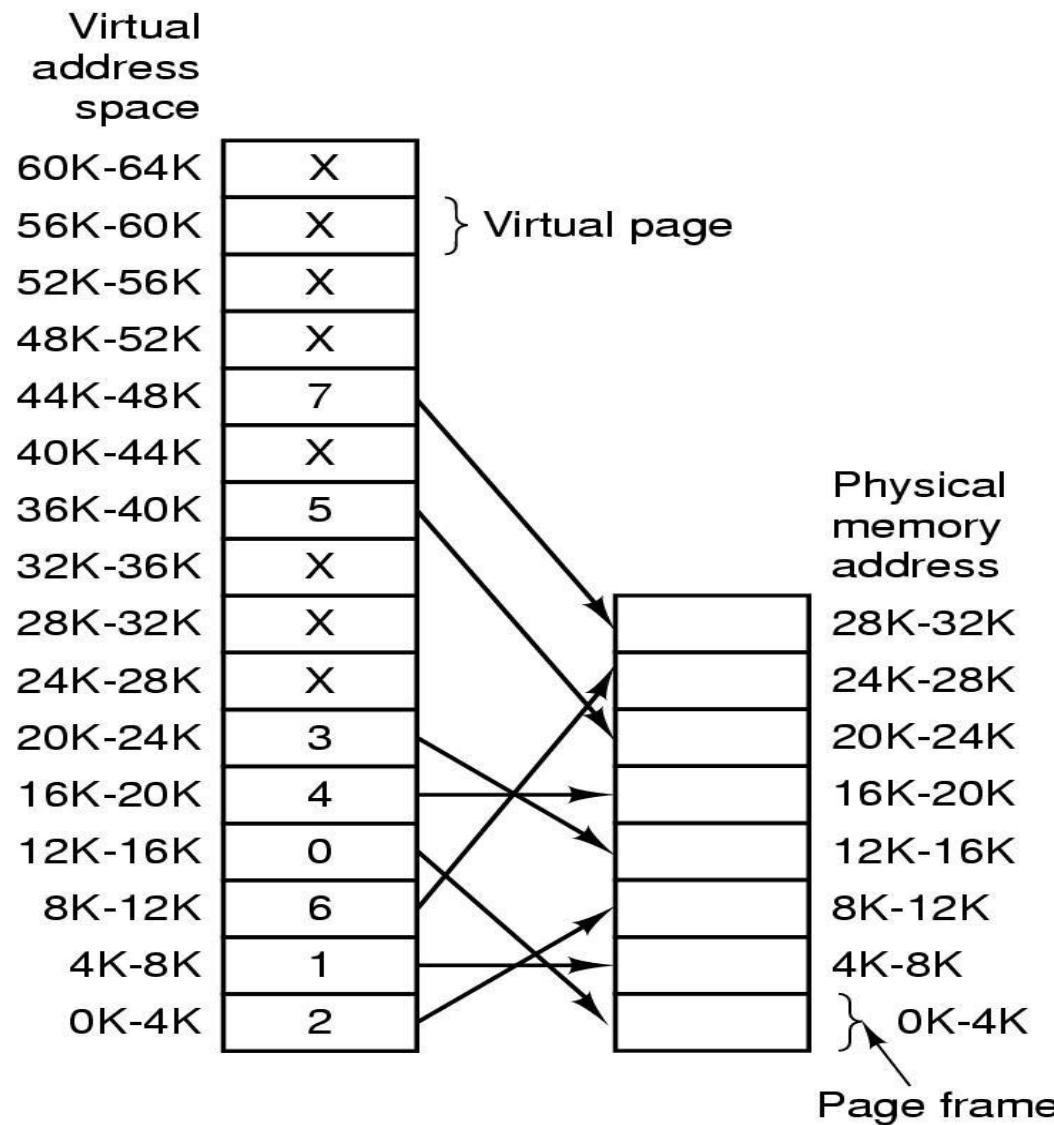
When a process is to be executed, its pages are loaded into any available memory frames from the backing store.

The size of the pages is determined by the hardware, normally from 512 bytes to 64KB (in power of 2).

Paging permits the physical address space of process to be noncontiguous.

Traditionally, support for paging has been handled by hardware, but the recent design have implemented by closely integrating the hardware and OS.

Paging



This example shows 64KB program can run in 32KB physical memory. The complete copy is stored in the disk and the pieces can be brought into memory as needed.

Paging

With 64KB of virtual address space and 32KB of physical memory and 4KB page size, we get 16 virtual pages and 8 frames.

What happen in following instruction?

MOV REG, 0

This virtual address, 0, is sent to the MMU. The MMU sees that this virtual address falls in page 0 (0-4095), which is mapping to frame 2 (8192 -12287). Thus the address 0 is transformed to 81912 and output address is 8192.

Address Translation

Address generated by CPU is divided into:

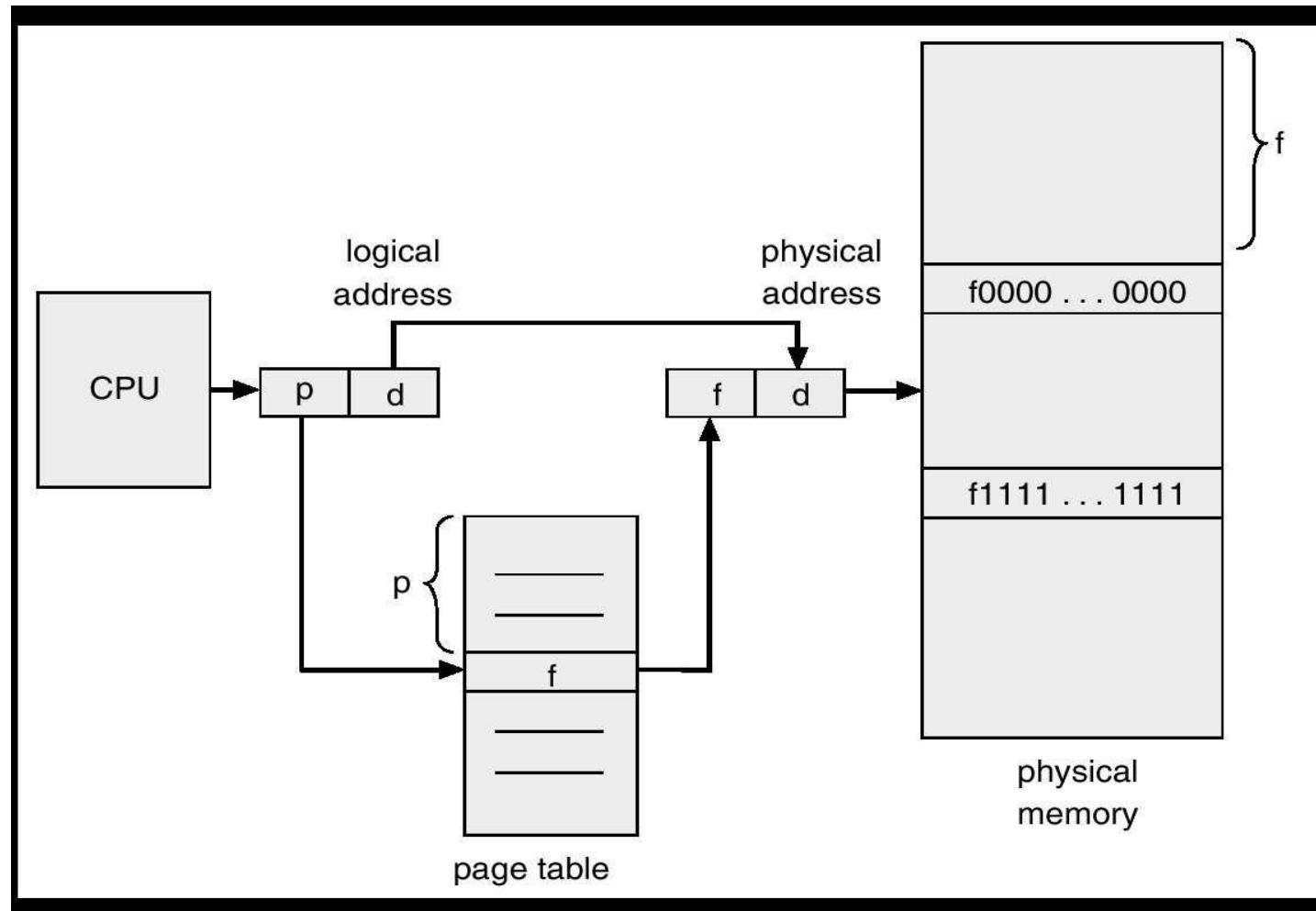
Page number (p) – used as an index into a *page table* which contains base address of each page in physical memory.

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

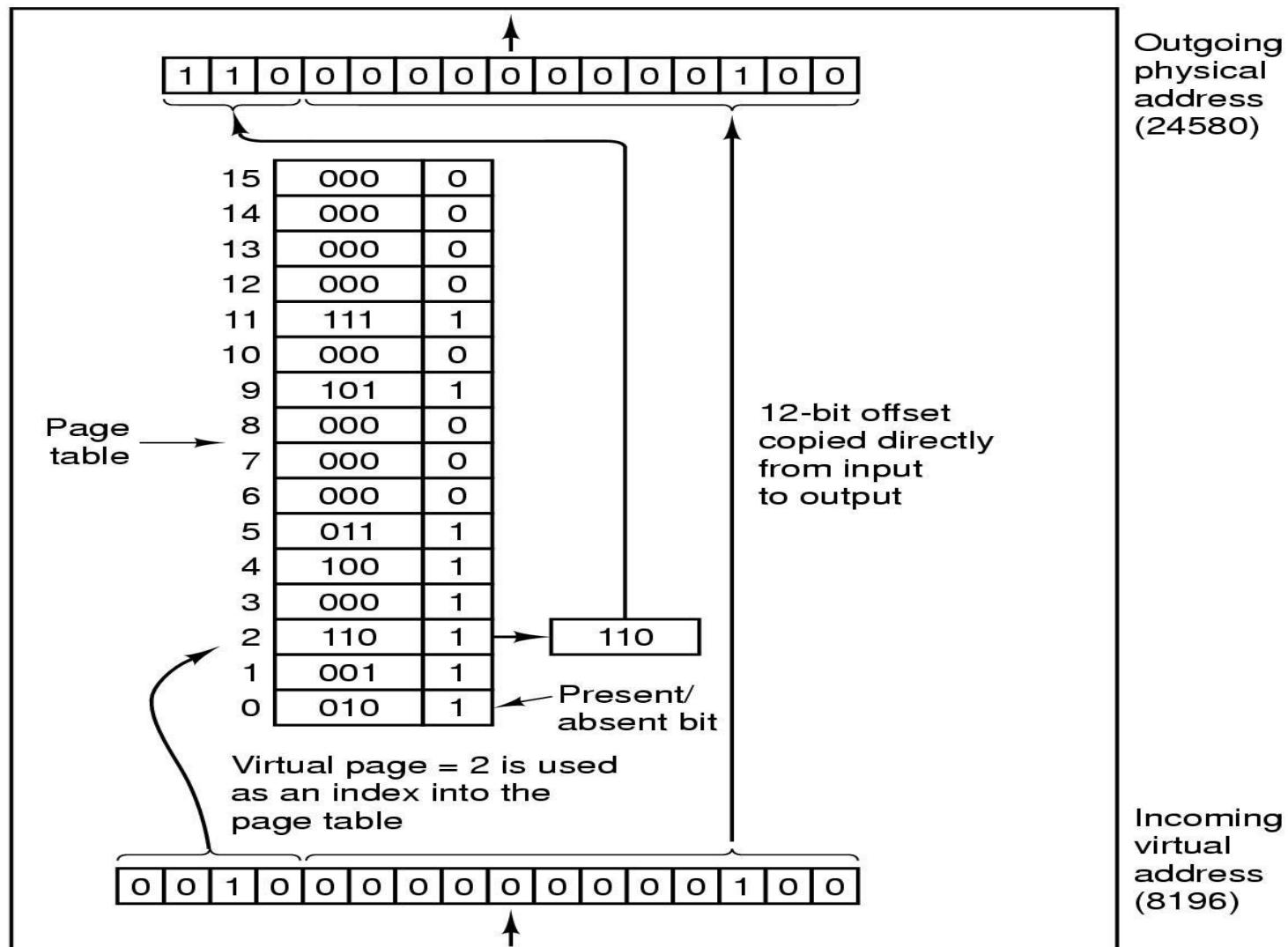
If the size of the logical address space is 2^m and page size is 2^n , then the high-order $m-n$ bits of logical address designate page number, and n lower order bits designate the page offset.

Present/absent bit keeps the track of which pages are physically present in memory.

Address Translation



Address Translation Example



Page Tables

For each process, page table stores the number of frame, allocated for each page.

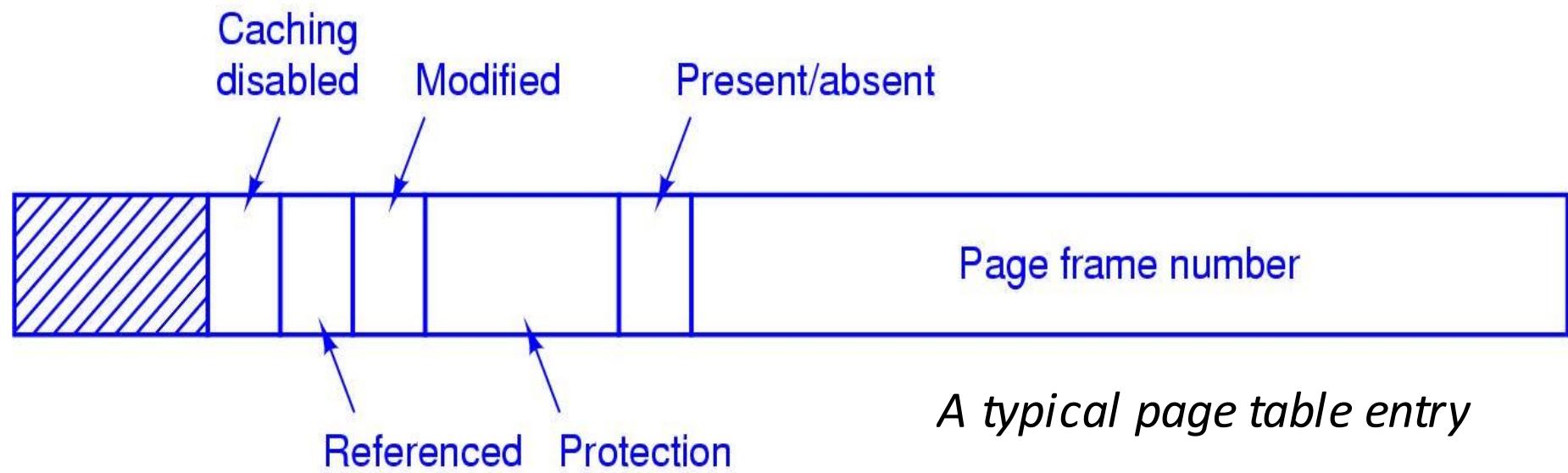
The purpose of the page table is to map virtual pages into pages frames. This function of page table can be represented in mathematical notation as:

$$\text{page_frame} = \text{page_table}(\text{page_number})$$

The virtual page number is used as an index into the page table to find the corresponding page frame.

Page Table Structure

The exact layout of page table entry is highly machine dependent, but more common structure for 32-bit system is as:



Frame number: The goal is to locate this value.

Page Table Structure

Present/absent bit: If present/absent bit is present, the virtual addresses is mapped to the corresponding physical address. If present/absent is absent the trap is occur called page fault.

Protection bit: Tells what kinds of access are permitted read, write or read only.

Modified bit (dirty bit): Identifies the changed status of the page since last access; if it is modified then it must be rewritten back to the disk.

Referenced bit: set whenever a page is referenced; used in page replacement.

Caching disabled: used for that system where the mapping into device register rather than memory.

Page Tables Issues

Size of page table.

Most modern computers support a large virtual-address space (2^{32} to 2^{64}). If the page size is 4KB, a 32-bit address space has 1 million pages. With 1 million pages, the page table must have 1 million entries.

think about 64-bit address space???

Efficiency of mapping.

If a particular instruction is being mapped, the table lookup time should be very small than its total mapping time to avoid becoming CPU idle.

What would be the performance, if such a large table have to load at every mapping.

How to handle these issues?

Multilevel Page Tables

To get around the problem of having to store huge page tables in memory all the time, many computers use the multilevel page table in which the page table itself is also paged.

Pentium II -2 level, 32-bit Motorola -4 level, 32 -bit SPARC-3 level etc.

For 64-bit architectures, multilevel page table are generally considered inappropriate. For example, the 64-bit UltraSPARC would require seven level of paging – increase accessing complexity.

Multilevel Page Tables

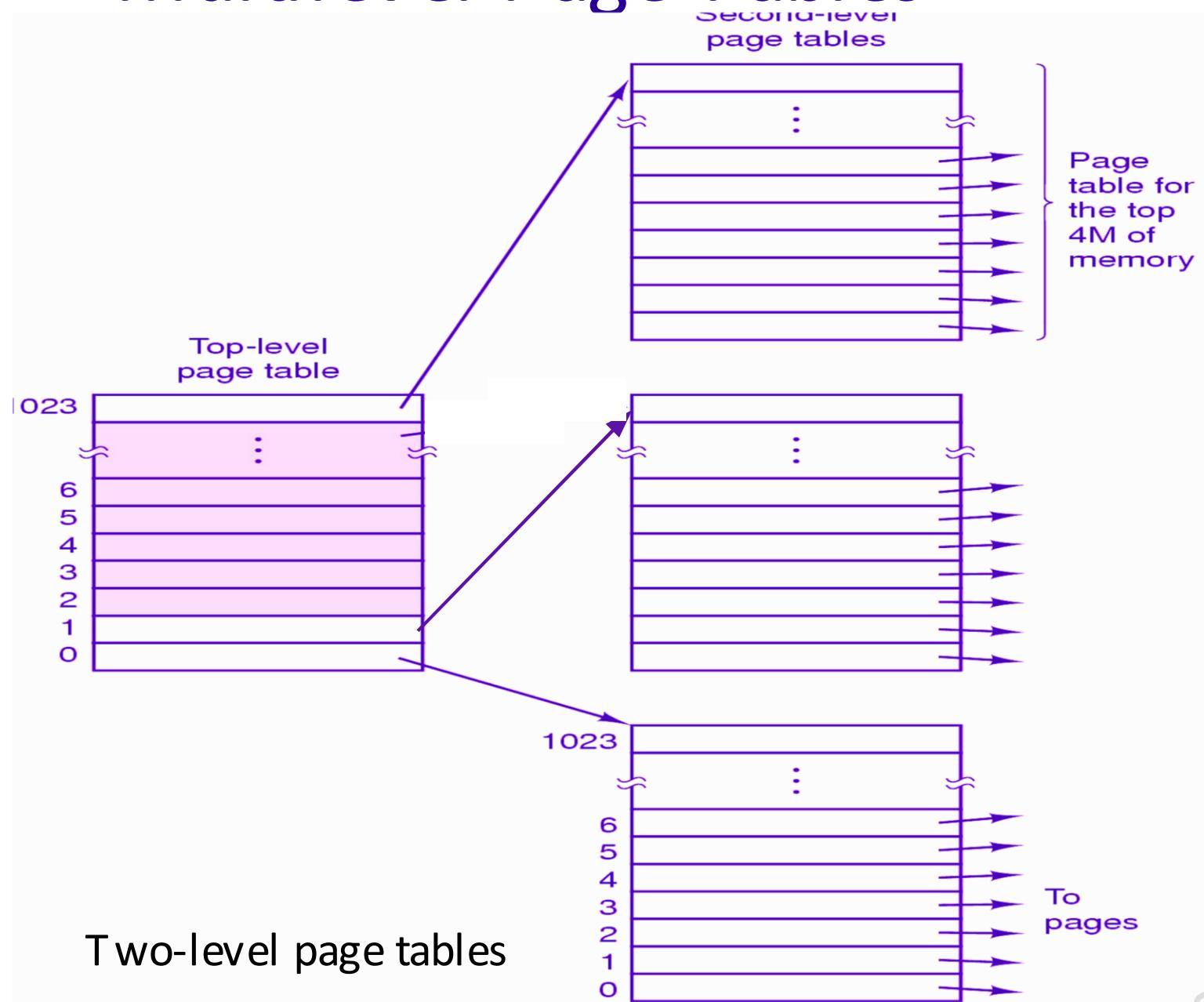
Example: Two-Level Page Tables

A 32-bit virtual address space with a page size of 4 KB, the virtual address space is partitioned into a 10-bit PT1 field, a 10-bit PT2 field, and a 12-bit offset field.

Bits	10	10	12
	PT1	PT2	Offset

The top level have 1024 entries, corresponding to PT1. At mapping, it first extracts the PT1 and uses this value as an index into the top level page table. Each of these entries have again 1024 entries, the resulting address of top-level yields the address or page frame number of second-level page table.

Multilevel Page Tables



Hashed Page Tables

A common approach for handling address space larger than 32-bit.

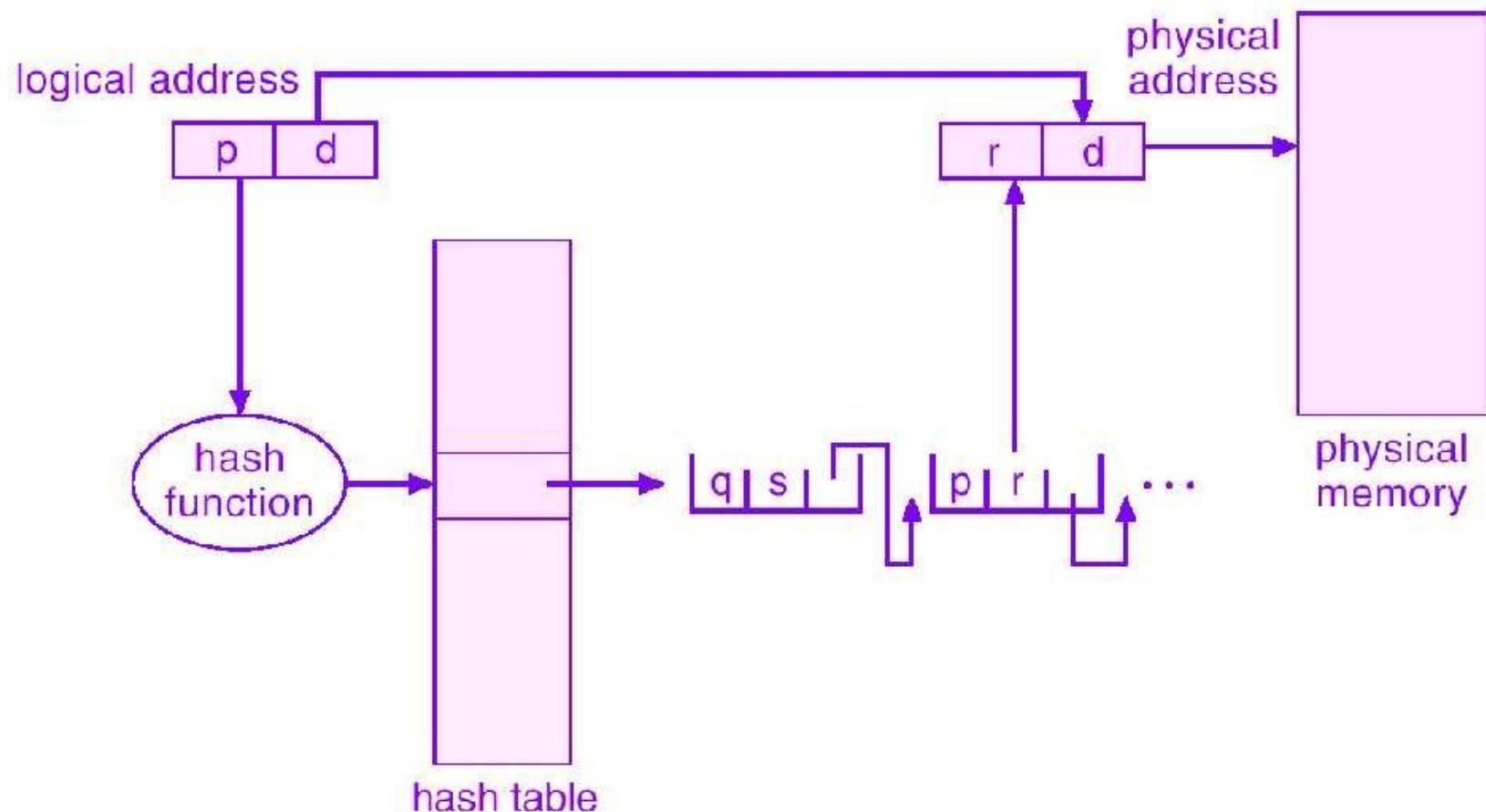
The hash value is the virtual-page number.

Each entry in the hash table contains a linked list of elements that hash to the same location.

Each element consists of three fields: virtual-page-number, value of mapped page frame, and a pointer to the next element.

The virtual address is hashed into the hash table, if there is match the corresponding page frame is used, if not, subsequent entries in the linked list are searched.

Hashed Page Tables



Hashed Page Table

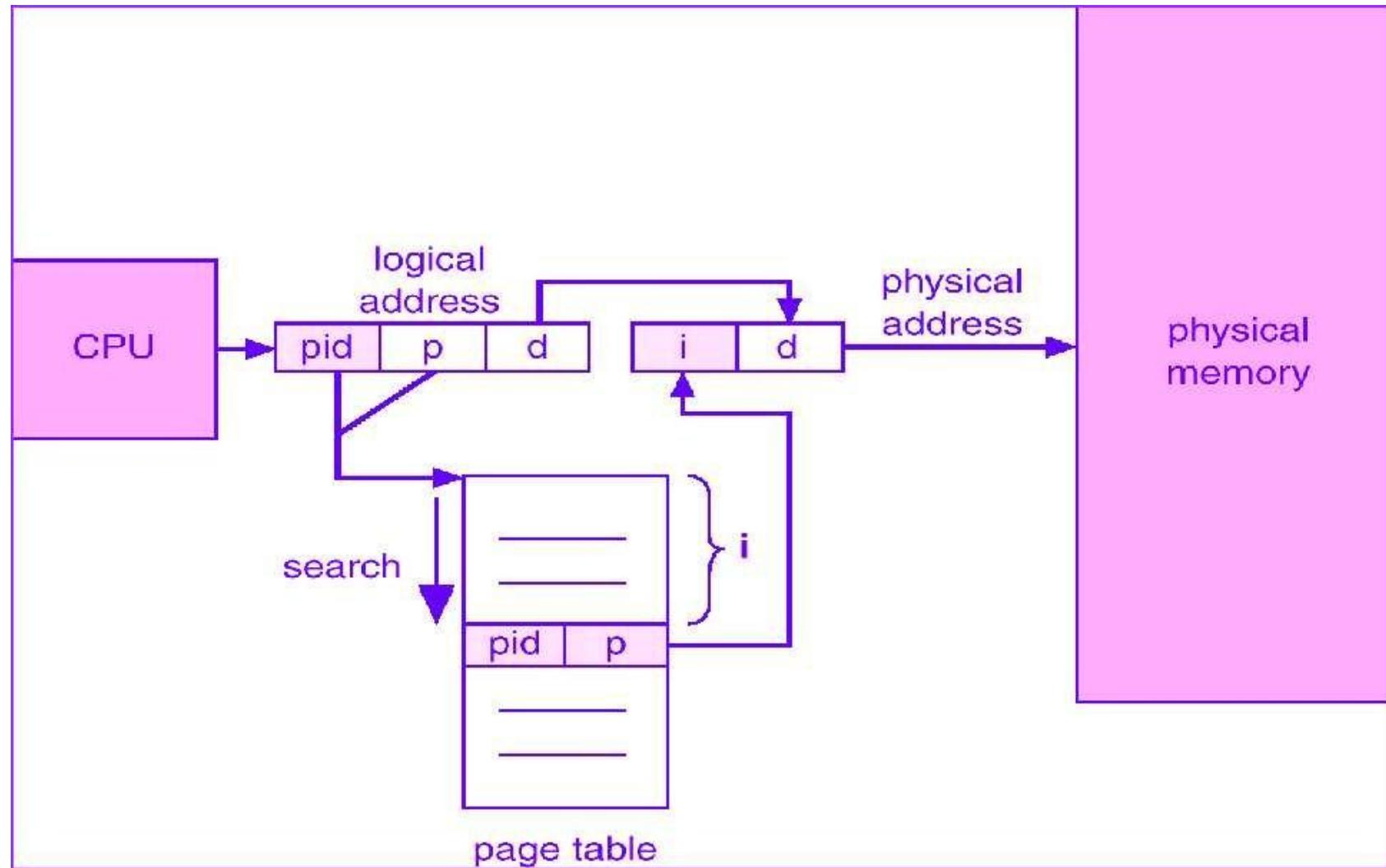
Inverted Page Tables

A common approach for handling address space larger than 32-bit. One entry per page frame, rather than one entry per page in earlier tables. (Ex: 256MB RAM with 4KB page requires only 65,536 entries in table)

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

Virtual address consists three fields: [process-id, page-number, offset]. The inverted page table entry is determined by [process-id, page-number]. The page table is search for the match, say at entry i the match is found, then the physical address [i, offset] is generated.

Inverted Page Tables



Inverted Page Tables

Advantage: Decreases the memory size to store the page table.

Problem: It must search entire table in every memory reference, not just on page faults.

Searching a 64K table in every reference is not a way to make system fast! Solutions:

Use of Hashed tables.

Use of TLBs.

Translation Lookaside Buffers (TLBs)

How to speed up address translation?

Locality: Most processes use large number of reference to small number of pages.

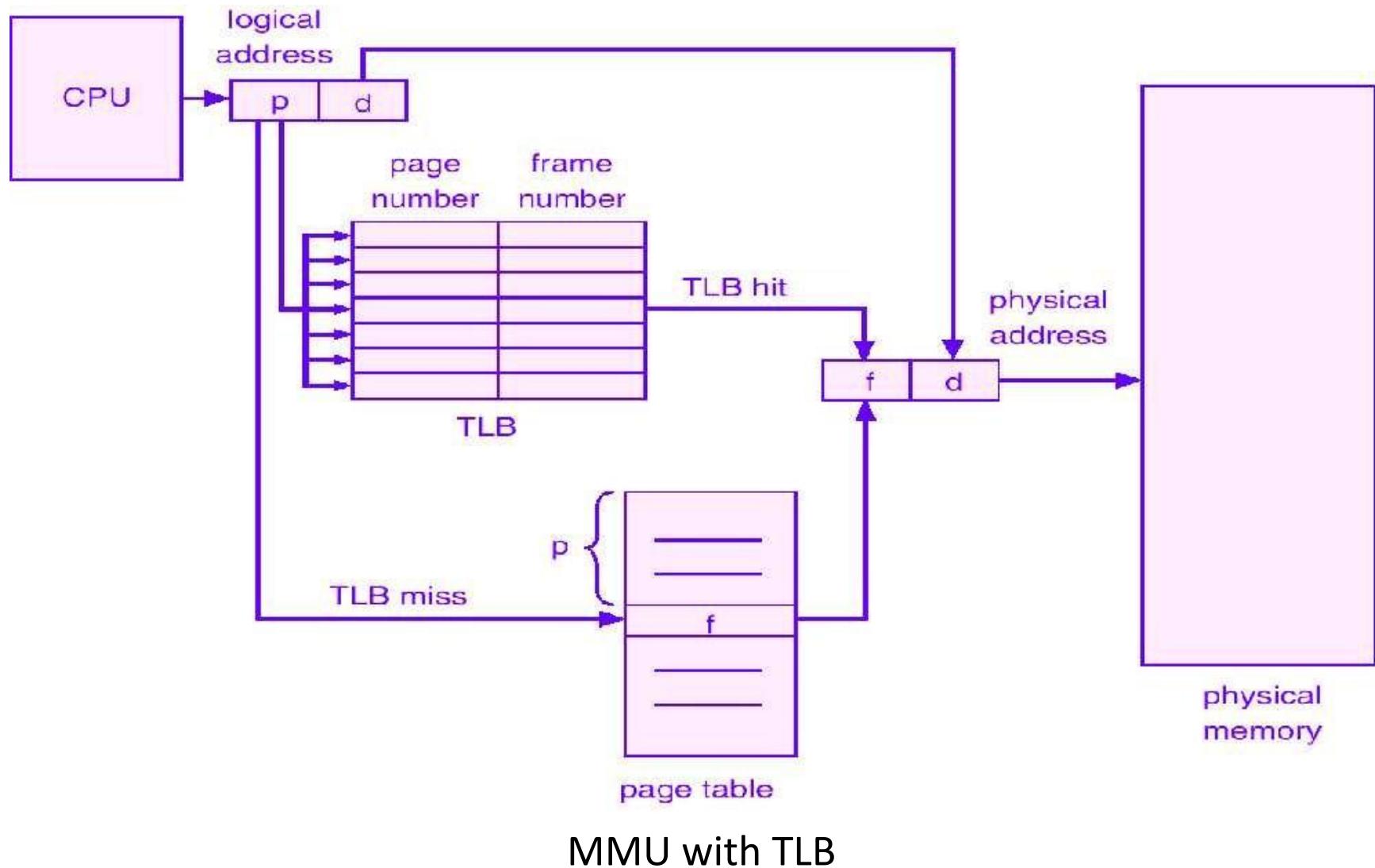
The small, fast, lookup hardware cache, called TLB is used to overcome this problem, by mapping logical address to physical address without page tables.

The TLB is associative, high-speed, memory; each entry consists information about virtual page-number and physical page frame number.

Typical sizes: only 64 to 1024 entries.

Key of improvement: Parallel search for all entries.

Translation Lookaside Buffers (TLBs)



MMU with TLB

Translation Lookaside Buffers (TLBs)

When logical address is generated by CPU, its page number is presented to the TLB; if page number is found (*TLB hit*), its frame number is immediately available, the whole task would be very fast because it compare all TLB entries simultaneously.

If the page number is not in TLB (*TLB miss*), a memory reference to the page table must be made. It then replace one entry of TLB with the page table entry just referenced. Thus in next time, for that page, TLB hit will found.

Advantages/Disadvantages of Paging

Advantages:

Fast to allocate and free:

Alloc: keep free list of free pages, grab first page in the list.

Free: Add pages to free list.

Easy to swap-out memory to disk.

Frame size matches disk page size.

Swap-out only necessary pages.

Easy to swap-in back from disk.

Disadvantages:

Additional memory reference.

Page table are kept in memory.

Internal fragmentation: process size does not match allocation size.

Home Works

HW #8

1. 6,7, 8, 10, 11, 12, 13,17 & 20 from text book (Tanenbaum),
Ch. 4.

2. Why are page sizes always a power of 2?
3. On a simple paging system with 2^{24} bytes of physical memory, 256 pages of logical address space, and a page size of 2^{10} bytes, how many bits are in a logical address?
4. Describe, how TLB increase performance in paging.

Read:

Page Replacement (Section 4.4 of Tanenbaum)

Page Replacement

Reading: Section 4.4 of Tanenbaum or 10.4 of Siberschatz

When a page fault occurs, the OS has to choose a page to remove from memory to make the room for the page that has to be brought in.

Which one page to be removed ?

What happen if the page that required next, is removed?

Principle of Optimality: *To obtain optimal performance the page to replace is one that will not be used for the furthest time in the future.*

Optimal Page Replacement (OPR)

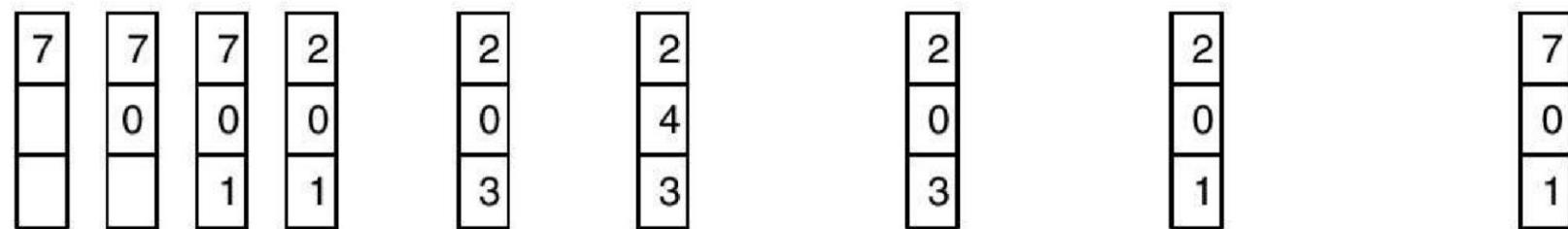
Replace the page that will not be used for the longest period of time.

Each page can be labeled with number of instructions that will be executed before that page is first referenced.

Ex: For 3 - page frames and 8 pages system the optimal page replacement is as:

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

By Bishnu Gautam

2

Optimal Page Replacement

3

Advantages:

An optimal page-replacement algorithm; it guarantees the lowest possible page fault rate.

Problems:

Unrealizable, at the time of the page fault, the OS has no way of knowing when each of the pages will be referenced next.

This is not used in practical system.

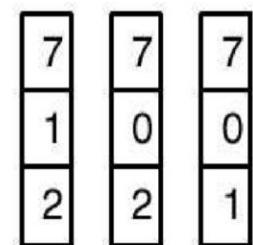
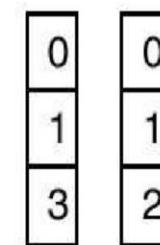
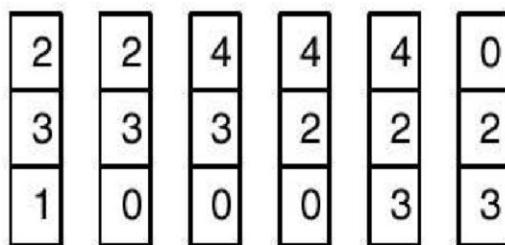
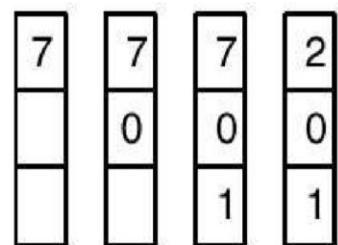
First-In-First-Out (FIFO)

This associates with each page the time when that page was brought into the memory. The page with highest time is chosen to replace.

This can also be implemented by using queue of all pages in memory.

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

By Bishnu Gautam

4

First-In-First-Out (FIFO)

5

Advantages:

Easy to understand and program.

Distributes fair chance to all.

Problems:

FIFO is likely to replace heavily (or constantly) used pages and they are still needed for further processing.

Second Chance

FIFO to avoid the mis-replacing of heavily used pages.

The reference bit is also checked and if it is present then its entry is updated as it has been just arrived into the system.

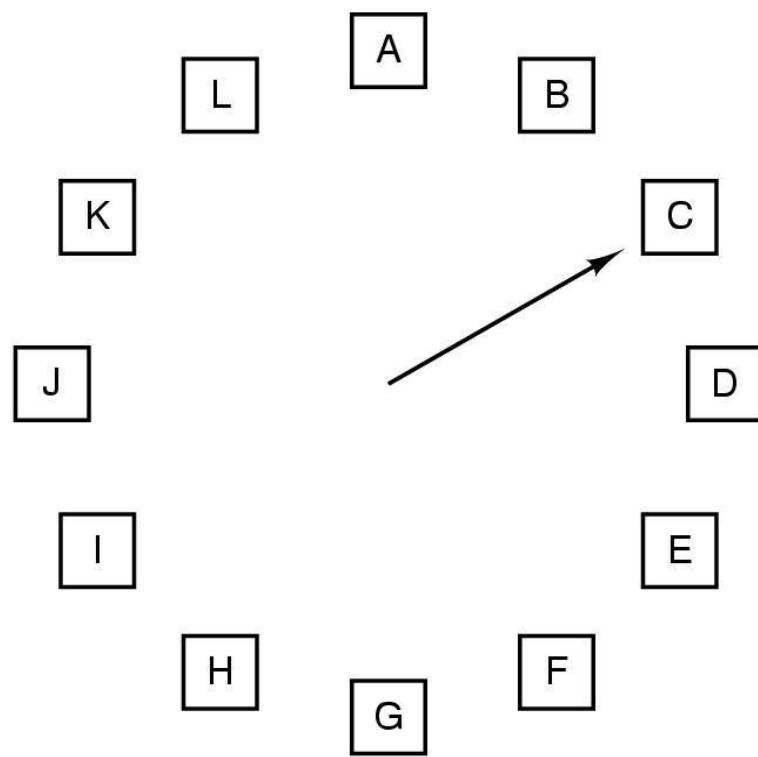
When a page gets a second chance, its reference bit is cleared and its arrival time is reset to the current time. **Advantages:**

Big improvement over FIFO.

Problems:

If all the pages have been referenced, second chance degenerates into pure FIFO.

Clock Page Replacement



When a page fault occurs,
the page the hand is
pointing to is inspected.
The action taken depends
on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

Arrange the pages in a circular list instead of a linear list.
Differ from second chance only in implementation.
Advantages: More efficient than second chance.

Least Recently Used (LRU)

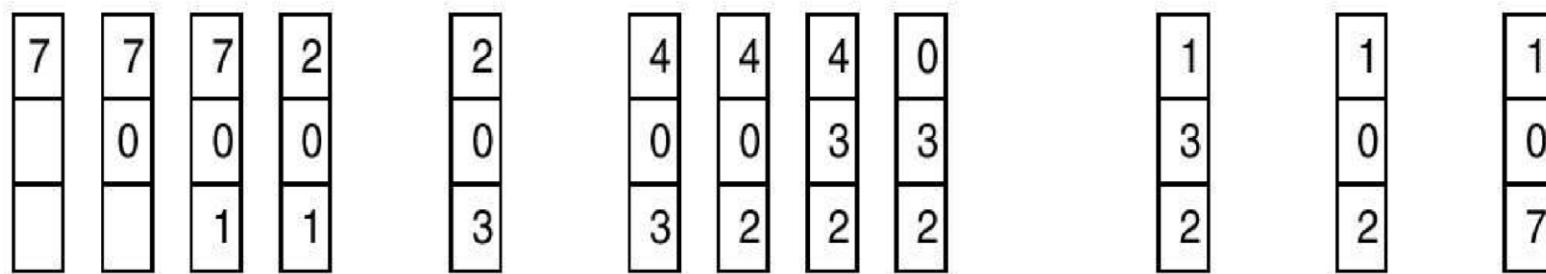
Recent past is a good indicator of the near future.

When a page fault occurs, throw out the page that has been unused for longest time.

It maintains a linked list of all pages in memory with the most recently used page at the front and least recently used page at the rear. The list must be updated on every memory reference.

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

By Bishnu Gautam

Least Recently Used (LRU)

Advantages:

Excellent, efficient is close to the optimal algorithm. Problems:

Difficult to implement exactly.

How to find good heuristic?

If it is implemented as linked list, updating list in every reference is not a way making system fast!

The Alternate implementation is by hardware primitives, it requires a time-of-use field in page table and a logical clock or counter in the CPU.

Least Frequently Used (LFU)

One approximation to LRU, software implementation.

LFU requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.

It requires a software counter associated with each page. When page fault occurs the page with lowest counter is chosen for replacement.

Problem: likely to replace highly active pages.

This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

Not Recently Used (NRU)

LRU approximation by enhancing second chance.

Pages not recently used are not likely to be used in near future and they must be replaced with incoming pages.

To keep useful statistics about which pages are being used and which pages are not, most computers have two status bits associated with each page – referenced and modified. These bits must be updated on every memory reference. When a page fault occurs, the OS inspects all the pages and divides them into four categories based on the current values of their referenced and modified bits.

Not Recently Used (NRU)

Class 0: not referenced, not modified.

Class 1: not referenced, modified.

Class 2: referenced, not modified.

Class 3: referenced, modified.

Pages in the lowest numbered class should be replaced first, and those in the highest numbered groups should be replaced last.

Pages within the same class are randomly selected.

Advantage: Easy to understand and efficient to implement.

Problem: Class 1 unrealistic.

Working Set (WS) Page Replacement

In multiprogramming, processes are frequently move to disk to let other process have a turn at the CPU.

What to do when a process just swapped out and another process has to load?

The set of pages that a process is currently using is called its working set.

If the entire working set is in memory, the process will run without causing many faults until it moves into another execution. Otherwise, excessive page fault might occur called *thrashing*.

Many paging systems try to keep track of each process' working set and make sure that it in memory before the process runs - *working set model or prepaging*.

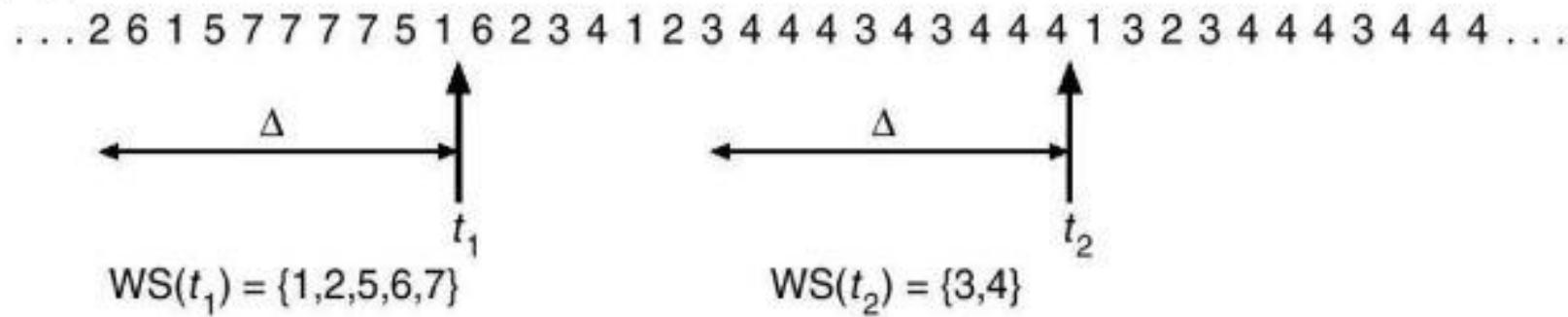
Working Set (WS) Page Replacement

The working set of pages of process, $ws(t, \Delta)$ at time t , is the set of pages referenced by the processes in time interval $t-k$ to t .

The variable Δ is called working-set-window, the size of Δ is central issue in this model.

Ex: Working-set-model with $\Delta = 10$.

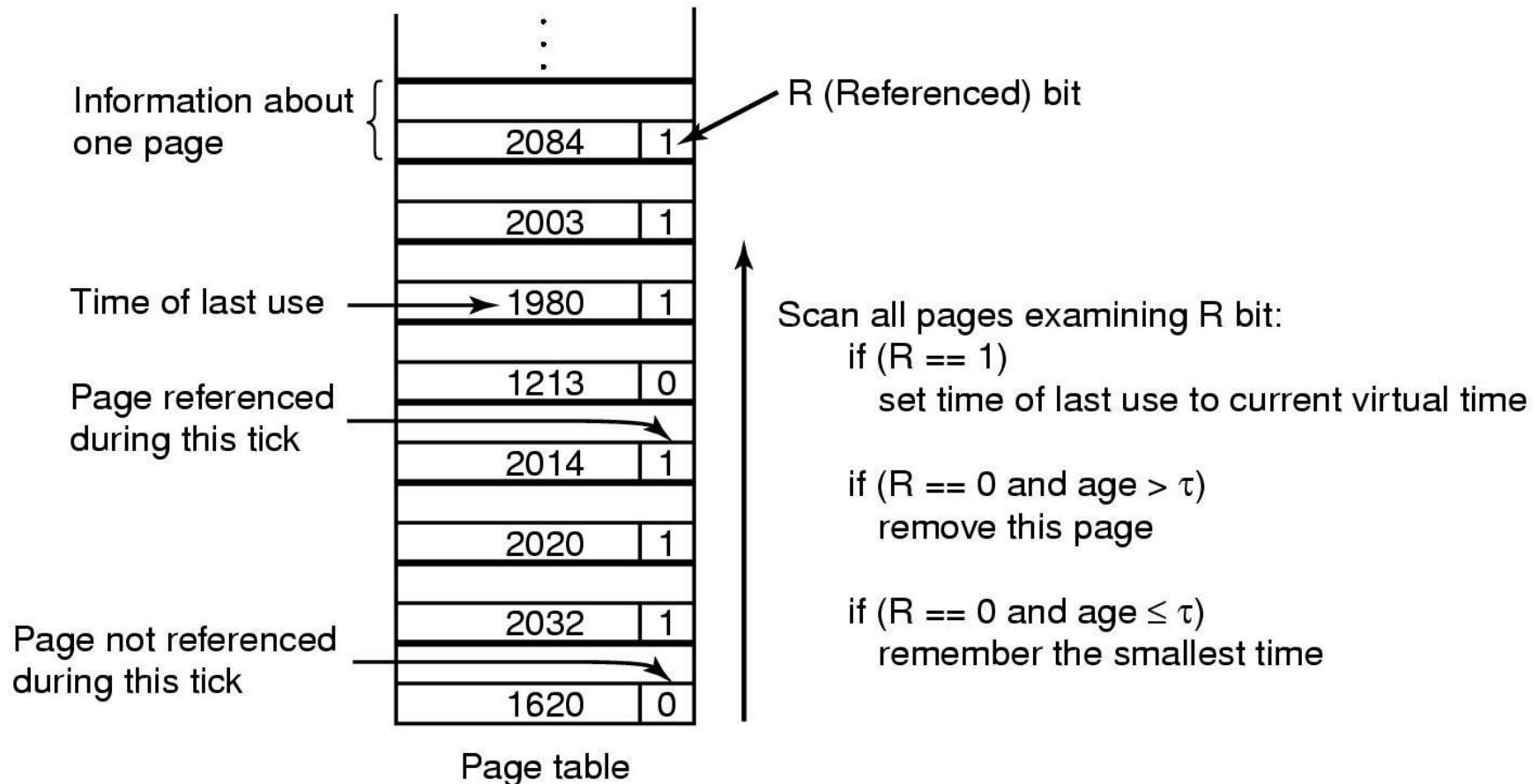
page reference table



Working Set (WS) Page Replacement

2204

Current virtual time



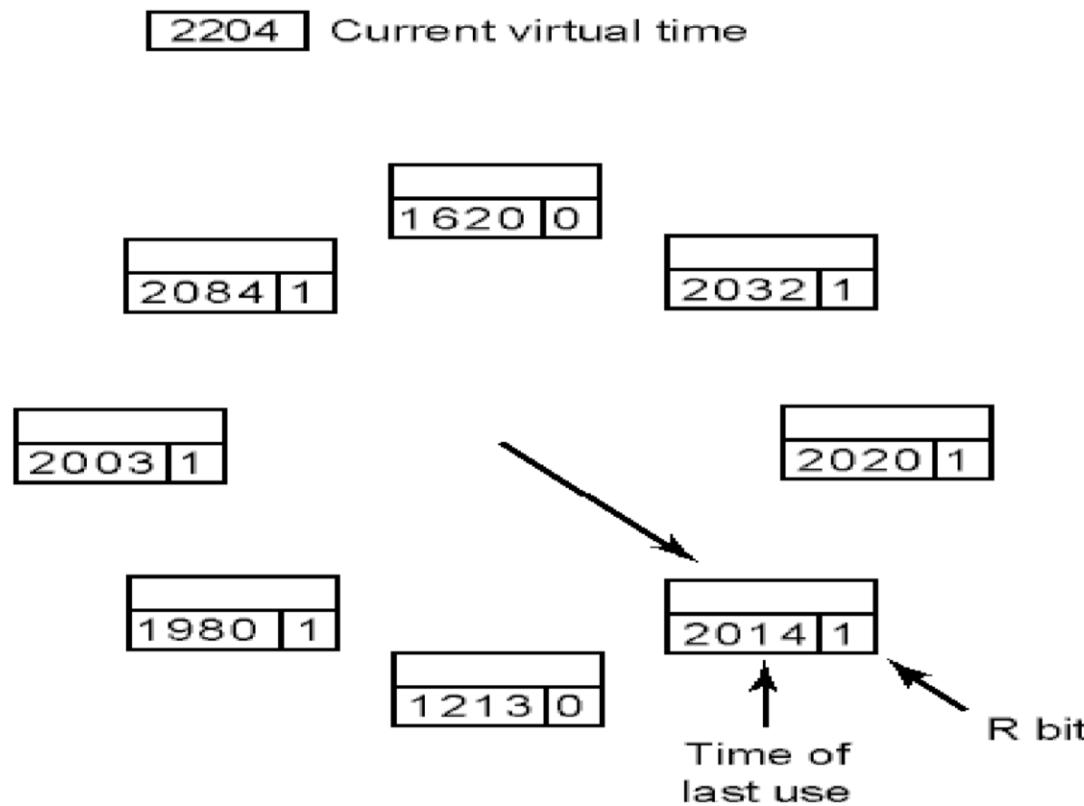
Source: www.csitnepal.com

Page replacement with working set model
Efficient but expensive to implement.

WSClock Page Replacement

Improved WS. Implement as clock with working set.

Each entry contain the time of last use field and reference bit.



Good efficient and widely used algorithm.

By Bishnu Gautam

Home Works HW#9:

1. 23, 24, 25, 26, 27 & 29 from your Textbook(Tanenbaum) Ch. 4.
2. Under what circumstances do page fault occur? Describe the action taken by the OS when a page fault occurs.
3. Given references to the following pages by a program,
1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6.
How many page faults will occur if the program has three page frames available to it and uses:
 - a) FIFO replacement?
 - b) LRU replacement?
 - c) Optimal replacement?

Source: www.csitnepal.com

(Remember that all frames are initially empty)

Read: Segmentation, Section 4.8 (Tanenbaum).

Segmentation

What happens if program increase their size in their execution?

How to manage expanding and contracting tables?

How to protect only data from the program?

How to share data to other program or functions?

The general solution of these issues is to provide the machine with many completely independent address spaces, called segments.

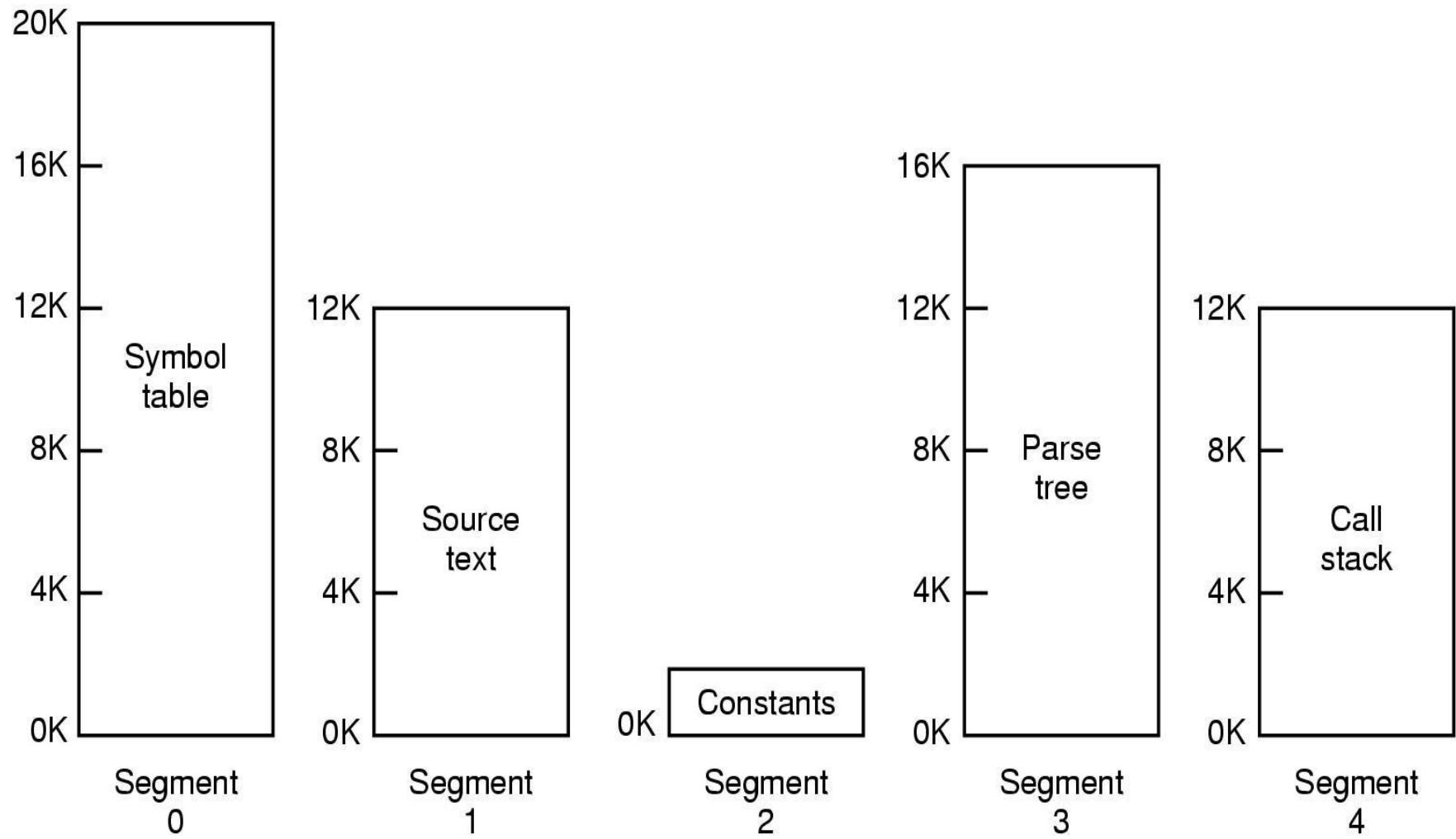
Segmentation

Memory management that support variable partitioning and mechanisms with freedom of contiguous memory requirement restriction.

The independent block of the program is a segment such as: main program, procedures, functions, methods, objects, local variables, global variables, common blocks, stacks, symbol table, arrays.

The responsibility for dividing the program into segments lies with user (or compiler).

Segmentation



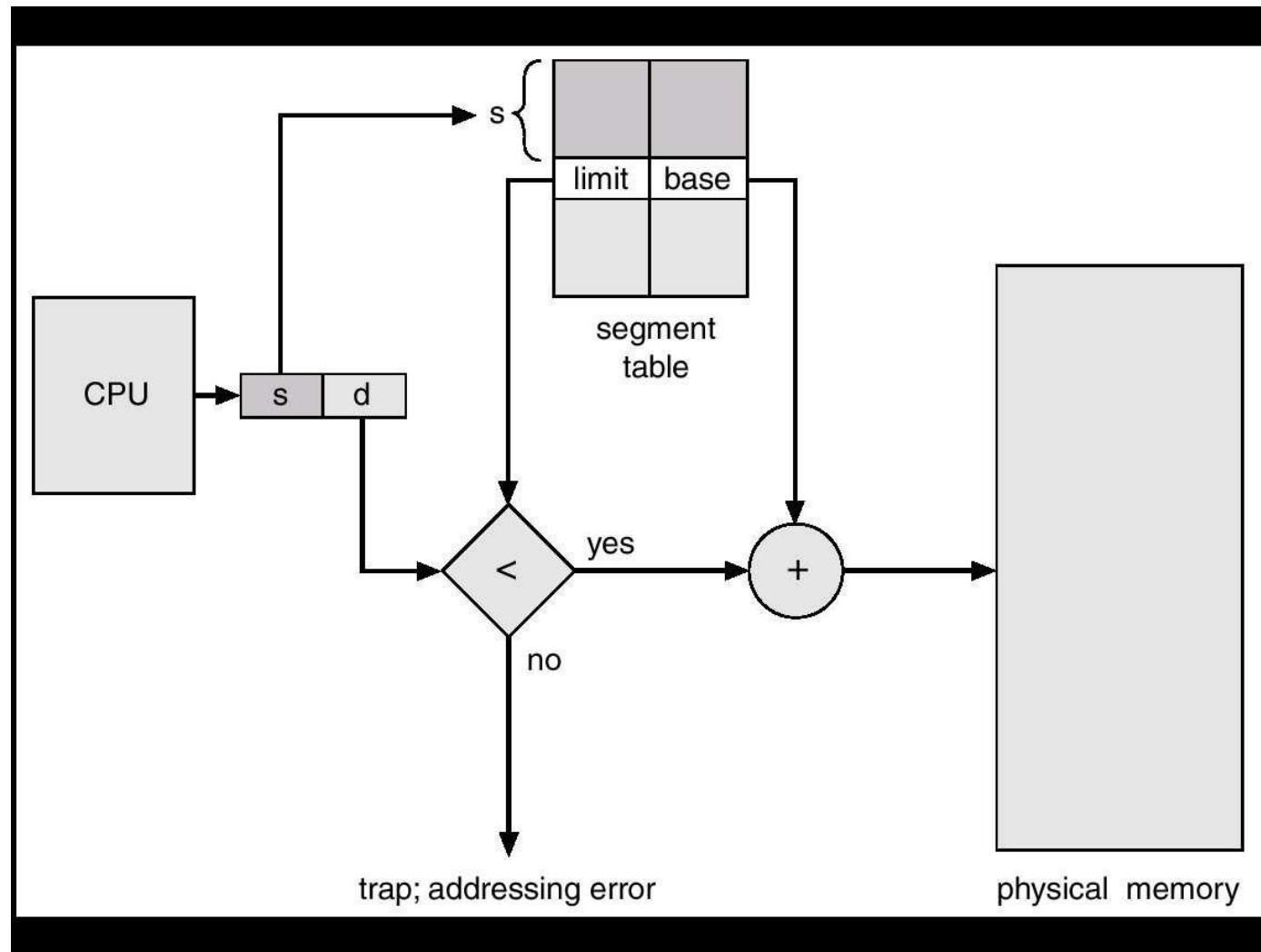
Segmentation

Different segments have its own name and size. The different segment can grow or shrink independently, with out effecting the others; so the size of segment changed during execution.

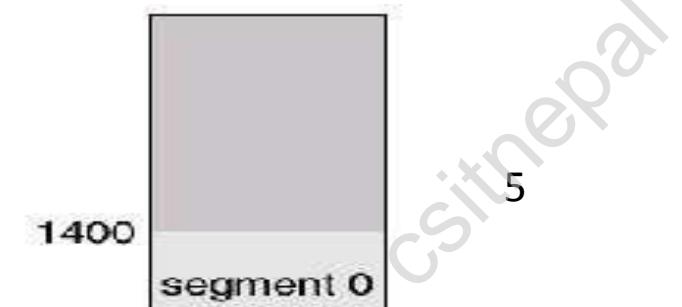
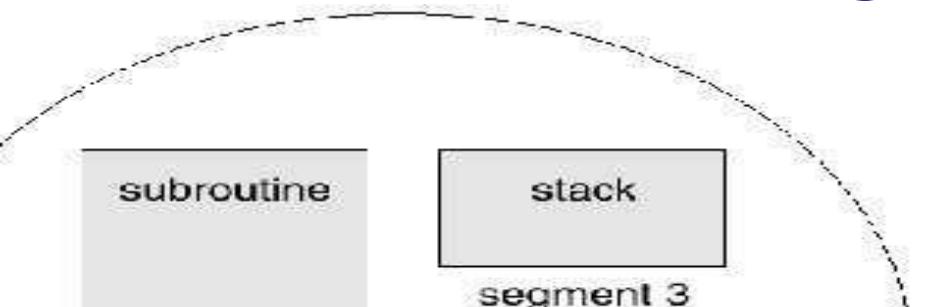
For the simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by segment name. Thus the logical address consist: *segment number* and *offset*.

The segment table (like page table but each entry consist limit and base register value) is used to map the logical address to physical address.

Segmentation



Segmentation



The segment number used as index into the segment table.

The offset d of the logical address must be between 0 and the segment limit. If not , trap occur, if it is legal it is added to the segment base to produce the address in the physical memory.

The segmentation scheme causes fragmentation, this can be handle by same technique of variable partition memory management.

Paging vs. Segmentation

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Segmentation with Paging

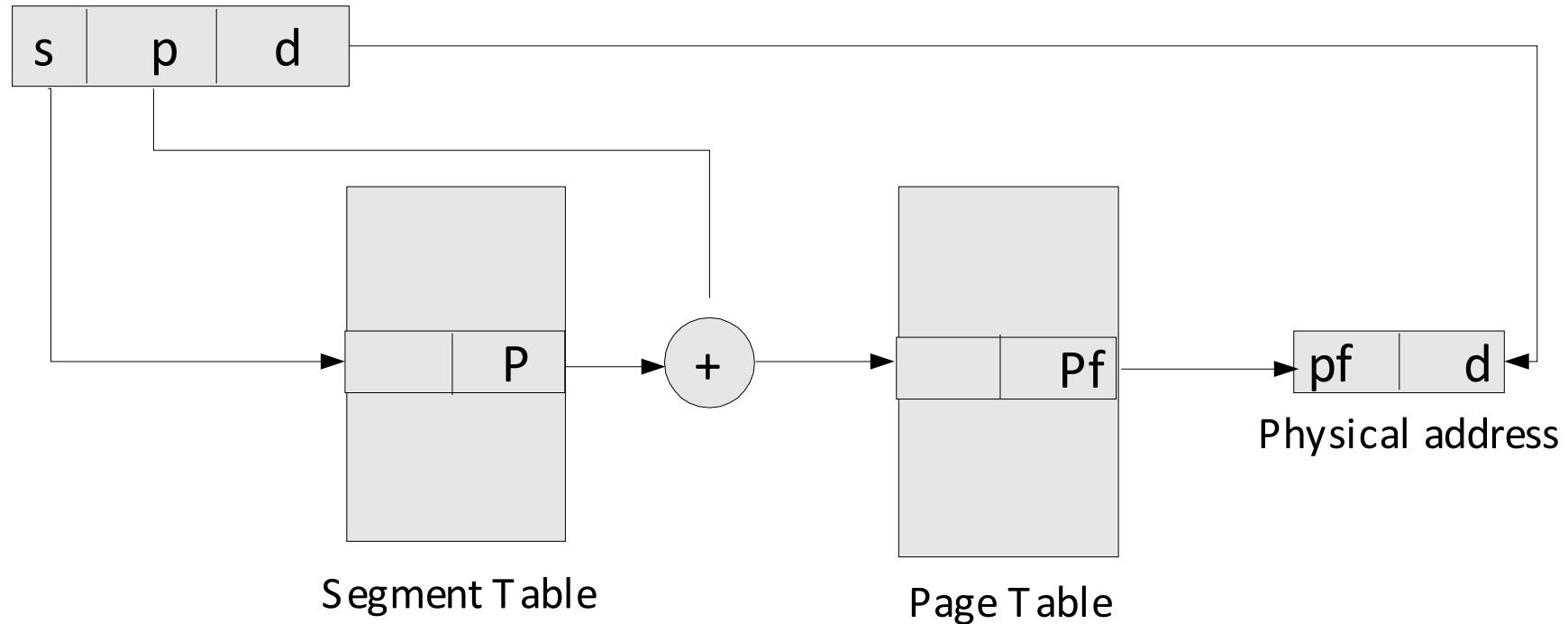
What happen when segment are larger than main memory?

Segmentation can be combined with paging to provide the efficiency of paging with the protection and sharing capabilities of segmentation.

As with simple segmentation, the logical address specifies the segment number and the offset within the segment. When paging is added, the segment offset is further divided into a page number and page offset. The segment table entry contains the address of the segment's page table.

Segmentation with Paging

Logical address



Segmentation with Paging

Examples:

The Intel Pentium:

The Intel Pentium 80386 and later architecture uses segmentation with paging memory management. The maximum number of segments per process is 16K, and each segment can be large as 4 GB. The page size is 4K. It use two-level paging scheme.

Multics:

It has 256K independent segments, and each up to 64K. The page size is 1K or small.

Home Works

HW#10:

1. Q. 36, & 37 from Textbook Tanenbaum.
2. Why are segmentation and paging sometimes combined into one scheme?

3. Consider the following segment table: segment base size

0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical address for the following logical address?

- a) 0430
- b) 110
- c) 2500
- d) 3400
- e) 4112

4. Distinguish the paging and segmentation.

Memory Management

Reading:Section 4.1 & 4.2 from Textbook(Tanenbaum)

The entire program and data of a process must be in main memory for the process to execute.

How to keep the track of processes currently being executed?

Which processes to load when memory space is available?

How to load the processes that are larger than main memory?

How do processes share the main memory?

OS component that is responsible for handling these issues is a *memory manager*.

By

Memory Management Issues

- Uniprogrammnig or Multiprogramming system.
- Absolute or relocatable partitions.
- Multiple fixed or multiple variable partitions.
- Partition in rigid manner or dynamic.
- Program run from specific partition or anywhere.
- Job placing at contiguous block or any available slot.

Memory Management Requirements

Relocation

Programmer does not know where the program will be placed in memory when it is executed
While the program is executing, it may be swapped to disk and returned to main memory at a different location (relocated)

Memory addresses must be translated in the code to actual physical memory address.

Hardwares used - Base (relocatable) register, limit register. By

Memory Management Requirements

Protection

Prevent processes from interfering with the OS or other processes.
Processes should not be able to reference memory locations in another process without permission.
Impossible to check absolute addresses in programs since the program could be relocated.
Often integrated with relocation.

Sharing

Allow several processes to access the same portion of memory (allow to share data).

Better to allow each process (person) access to the same copy of the program rather than have their own separate copy.

Logical vs. Physical Memory Address

Logical Address: The address generated by CPU.
Also known as virtual address.

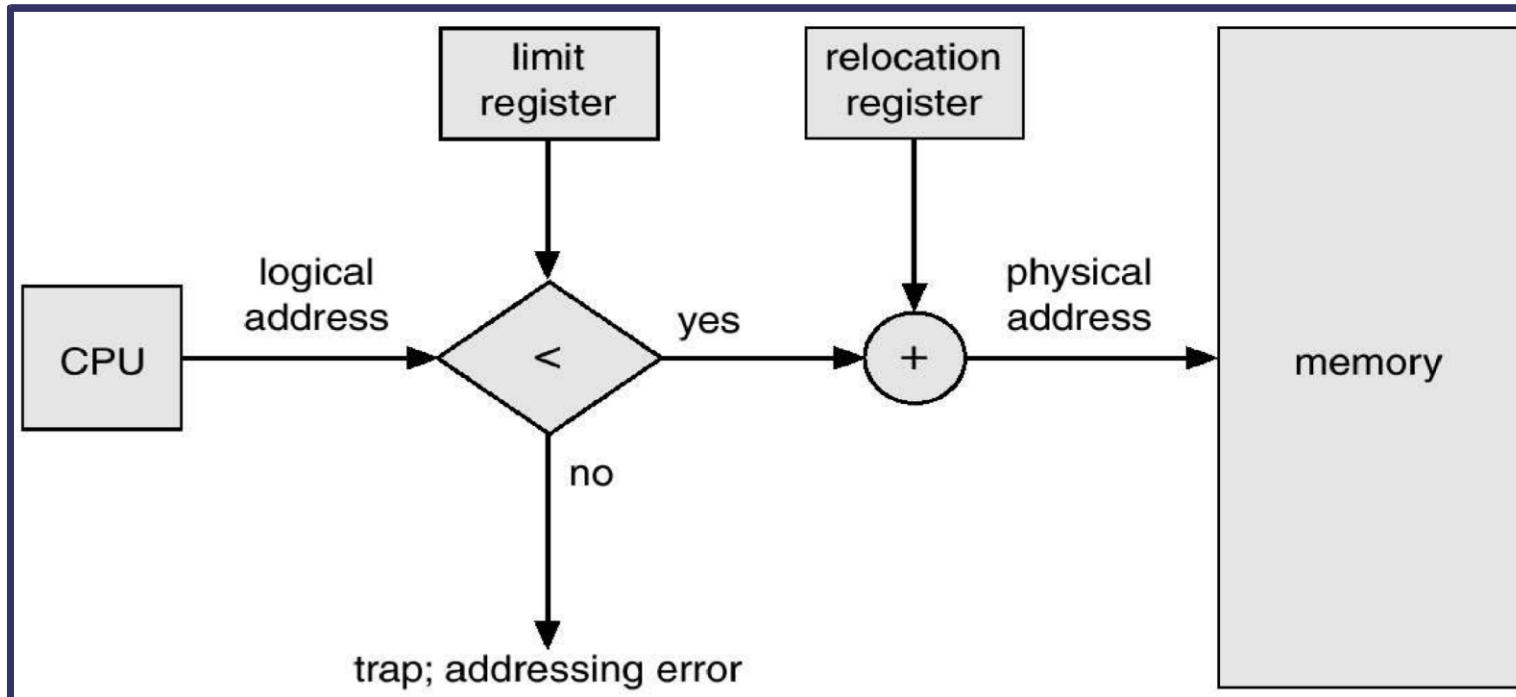
Physical Address: Actual address loaded into the
memory address register.

Mapping from logical address to physical address is
relocation.

Two registers: *Base* and *Limit* are used in mapping.
This mechanism also used in memory protection –
memory outside the range are protected.

if Base = 1500 physical address =
1500 + 300 = 1800.

Logical to Physical Address



Base register (relocation): Holds smallest legal physical memory address.

Limit register: contain the size of range.

Example:

logical address = 300

By Bishnu

Uniprogramming Model

2^n

Highest memory holds OS

Process runs from 0 up to OS

Run one program at a process. Memory partitions by allocated to OS and the other to executing process.

OS
User programs

0

time with single segment per space is divided into two convention; one partition is

Uniprogramming Model

As soon as the user types the command, the OS copies the requested program from disk to memory and execute it.
Used in early Dos system.

Disadvantages:

Only one process can run at a time.

Processes can destroy OS (an erroneous process can crash the entire system).

Multiprogramming Model

Fixed Partition Multiprogramming

Multiple Partitions are created to allow multiple user processes to resident in memory simultaneously.

The partitions are fixed (possibly unequal), can be done using *base* and *limit* register.

Partition table stores the partition information for a process. When job arrives, it can be put into the input queue for the smallest partition large enough to hold it. Since the partition are fixed, any space in partition not used by a process is lost. Used by OS/360 on large mainframes.

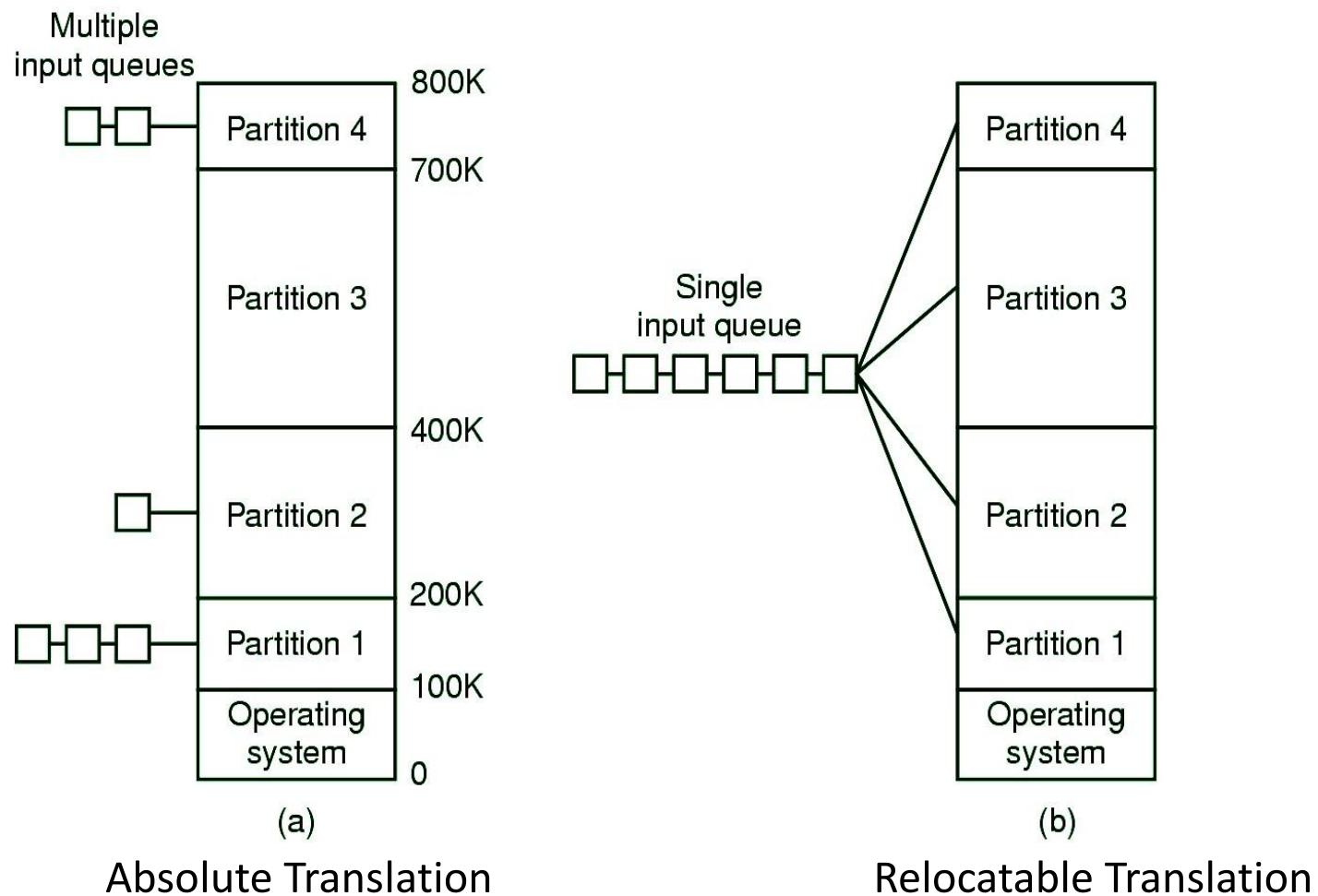
Implemented in two ways :

Dedicate partitions for each process (Absolute translation).

Maintaining a single queue (Relocatable translation).

Multiprogramming Model

Fixed Partition Multiprogramming



Multiprogramming Model Fixed Partition Multiprogramming

Dedicating Partitions

Separate input queue is maintained for each partition.

Processes are translated with absolute assemblers and compilers to run only in specific partition.

Problems:

if a process is ready to run and its partition is occupied then that process has to wait even if other partitions are available – *wastage of storage*.

Multiprogramming Model

Fixed Partition Multiprogramming

Maintaining Single Queue

Maintains a single queue for all processes.

When a partitions becomes free, the processes closest to the front of queue that fits in it could be loaded into the empty partition and run. Not to waste the large partition for small process, another strategy to search the whole input queue whenever a partition becomes free and pick the largest process that fits.

Problems:

Eliminate the absolute problems but implementation is complex.

Wastage of storage when many processes are small.

Read yourself: Section 4.1.3 and 4.1.4 (Tanenbaum)

Variable Partition Multiprogramming

How to allocate the space for a process as much as they need?

-variable partition multiprogramming.

When processes arrive, they are given as much storage as they need.

When processes finish, they leave holes in main memory; OS fills

these holes with another processes from the input queue of processes.

Advantages:

Processes get the exact size partition of their request for memory – *no waste of memory.*

Problems:

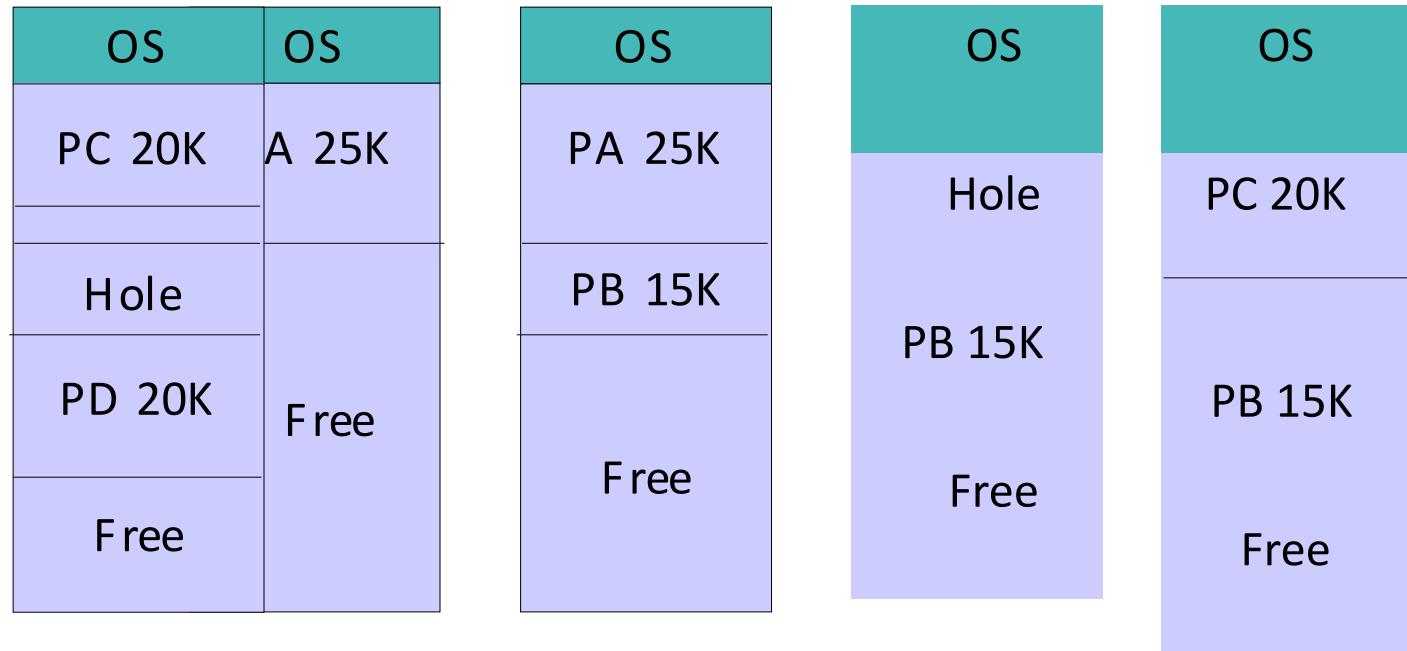
What will happen when we leave hear ???

When holes given to other process they may again partitioned, the remaining holes gets smaller eventually becoming too small to hold new processes – waste of memory occur.

Variable Partition Multiprogramming

Input queue

PA 25K PB 15K PC 20 K PD 20 K



Memory allocation and holes

Solutions???

Variable Partition Multiprogramming

Memory Compaction

By moving processes in memory, the memory holes can be collected into a single section of unallocated space – memory compaction.

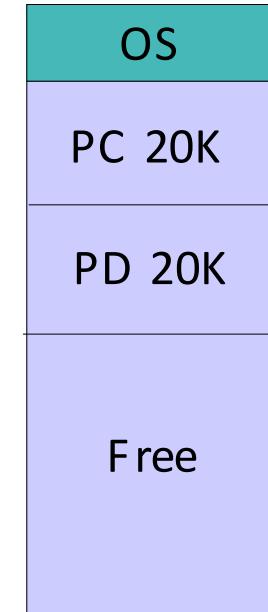
Problems:

Not possible in absolute translation.

If it possible, it is highly expensive – it requires a lots of CPU time;

Eg: On a 256-MB machine that can copy 4 bytes in 40 nsec, it takes about 2.7 sec to compact all memory.

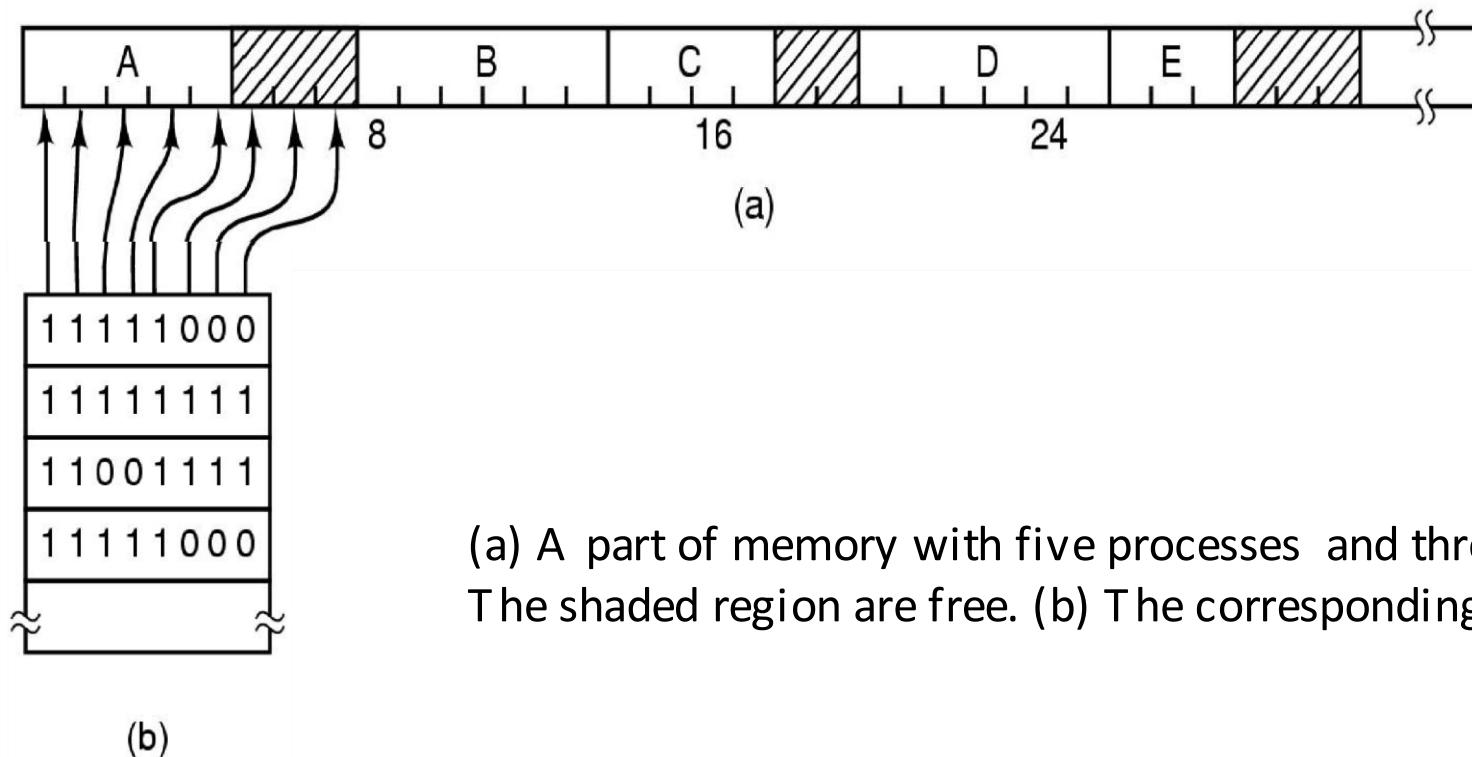
It stops every thing when compaction. Memory compaction
This technique is not used.



Variable Partition Multiprogramming Bitmap Management

Memory is divided up into allocations units and each bit in the map

indicates whether or not the corresponding unit is allocated.



Variable Partition Multiprogramming

Bitmap Management

Each allocation unit is a bit in the bit map, 0 if the unit is free, and 1 if it is occupied (or vice versa).

When a process come into the memory, it search required numbers of consecutive 0 bits in the map.

The size of the allocation unit is an important design issue; Increasing the size of allocation decrease the size of bitmap, but increase the amount of memory wasted when the process size is not a multiple of size of the allocation unit. **Advantages:**

Provides a simple way to keep track of memory words.

Problems:

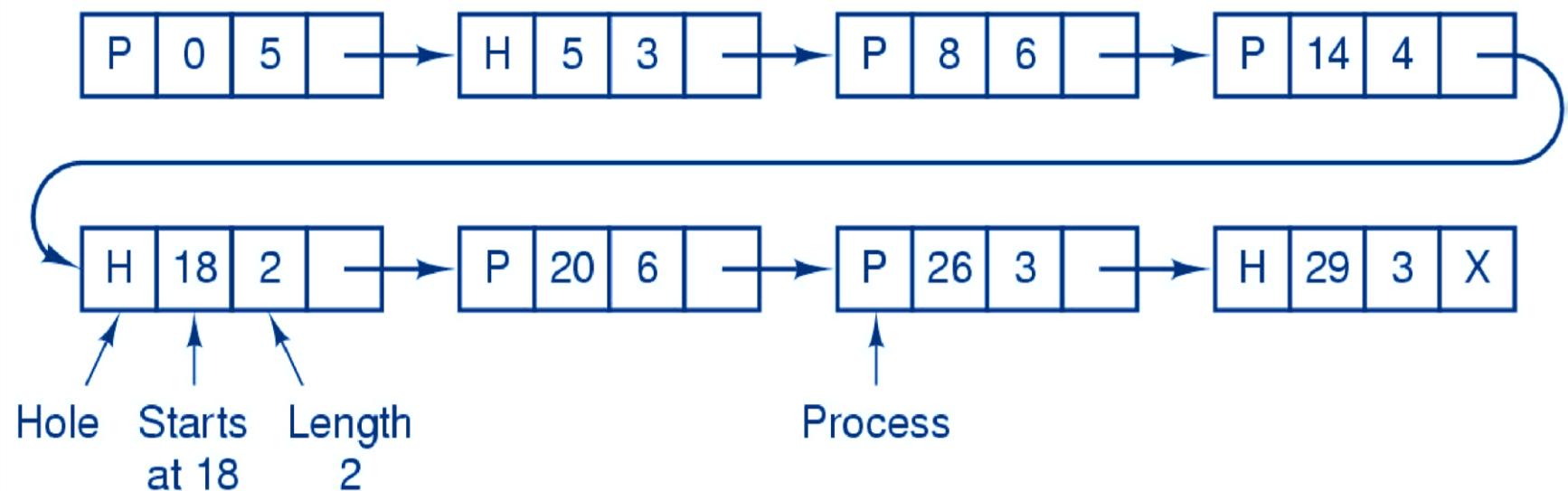
Searching a bitmap for a run of given length is a slow operation.

Variable Partition Multiprogramming

Linked List Management

Maintains a linked list of allocated and free memory segments, where a segment is either a process or a hole between the two processes.

Each segment contains an entry indicating the segment size and a pointer to the next segment in the list.



Variable Partition Multiprogramming

Linked List Management

H represents hole and P represents process and the segment list is kept sorted by address. Sorting has an advantage that when a process terminates, updating the list is straightforward. It may be implemented as doubly-linked list to make a more convenient search.

When a process terminates, if any neighbors is already hole, merge the holes – called coalescing.



Problem : Still some form of fragmentation

Variable Partition Multiprogramming

Partition Selection Algorithms

Situation: Multiple memory holes are large enough to contain a process, OS must select which hole the process will be loaded into.

First Fit

Allocate the first hole that is big enough. It stop the searching as soon as it find a free hole that is large enough. The hole is then broken up into two pieces, one for the process and one for unused memory.

Advantage: It is a fast algorithm because it search as little as possible.

Problem: not good in terms of storage utilization.

Variable Partition Multiprogramming

Partition Selection Algorithms

Best Fit

Allocate the smallest hole that is big enough. It search the entire list, and takes the smallest hole that is big enough. Rather than breaking up a big hole that might be needed later, it finds a hole that is close to the actual size.

Advantage: More storage utilization than first fit but not always.

Problem: Slower than first fit because it requires to search whole list at every time.

Creates tiny hole that may not be used.

Variable Partition Multiprogramming Partition Selection Algorithms

Worst Fit

Allocate the largest hole. It search the entire list, and takes the largest hole. Rather than creating tiny hole it produces the largest leftover hole, which may be more useful.

Advantage: Some time it has more storage utilization than first fit and best fit.

Problem: not good for both performance and utilization.

Swapping

Till now: *User process remains in main memory until completion and contiguous allocation.*

In timesharing system, the situation may be different, some times there is no enough memory to hold all processes.

How to handle this problem? – *swapping.*

Access process must be kept on the disk and brought it into memory and run dynamically.

A process, however, can be swapped temporarily out of the memory to disk, and then brought back into memory for continued execution.

In timesharing system processes will be swapped in and out many times before its completion.

In some swapping systems, one processes occupies the main storage at once; when process can no longer to continue it relinquishes both the storage and CPU.

Overlays

In the past when the programs were too big to fit in available main memory, the solution adopted was to split the program into pieces called overlays.

Overlays 1 would start running first; when it was done, it would call next overlay 2 and so on.

The overlays are kept on disk and swapped in and out by OS.

Problem:

The job of splitting program into pieces (making overlays) had to be done by programmer; time consuming and boring for programmer.

Solution: Virtual memory

Home Works

HW #7

1. Problems 1, 3, 4 and 5 (Textbook, Tanenbaum, Ch. 4).
2. How fragmentation occur? Discuss the techniques that manage the fragmentation.

3. A program is created by assuming it will be loaded at the address 100, so the program refers address as: 135, 160, 164, 220, 224. But it loaded at the location 500, the will be physical address of the program. 4. Given a memory partitions of 100KB, 500KB, 200KB, 300KB, & 600KB (in order), how would each of the first fit, best fit and worst fit algorithms places processes of 212KB, 417KB, 112KB, & 426KB (in order)? Which algorithm makes the most efficient use of memory?

Be ready for virtual memory.

Disk Management

Reading : Section 5.4 of Text Tananebaum or Ch 14 From Text Silberschatz

Disk is an I/O devices that is common to every computer.

Disk structure

Disk Scheduling

RAID

Disk Formatting & Error Handling RAM

Disks

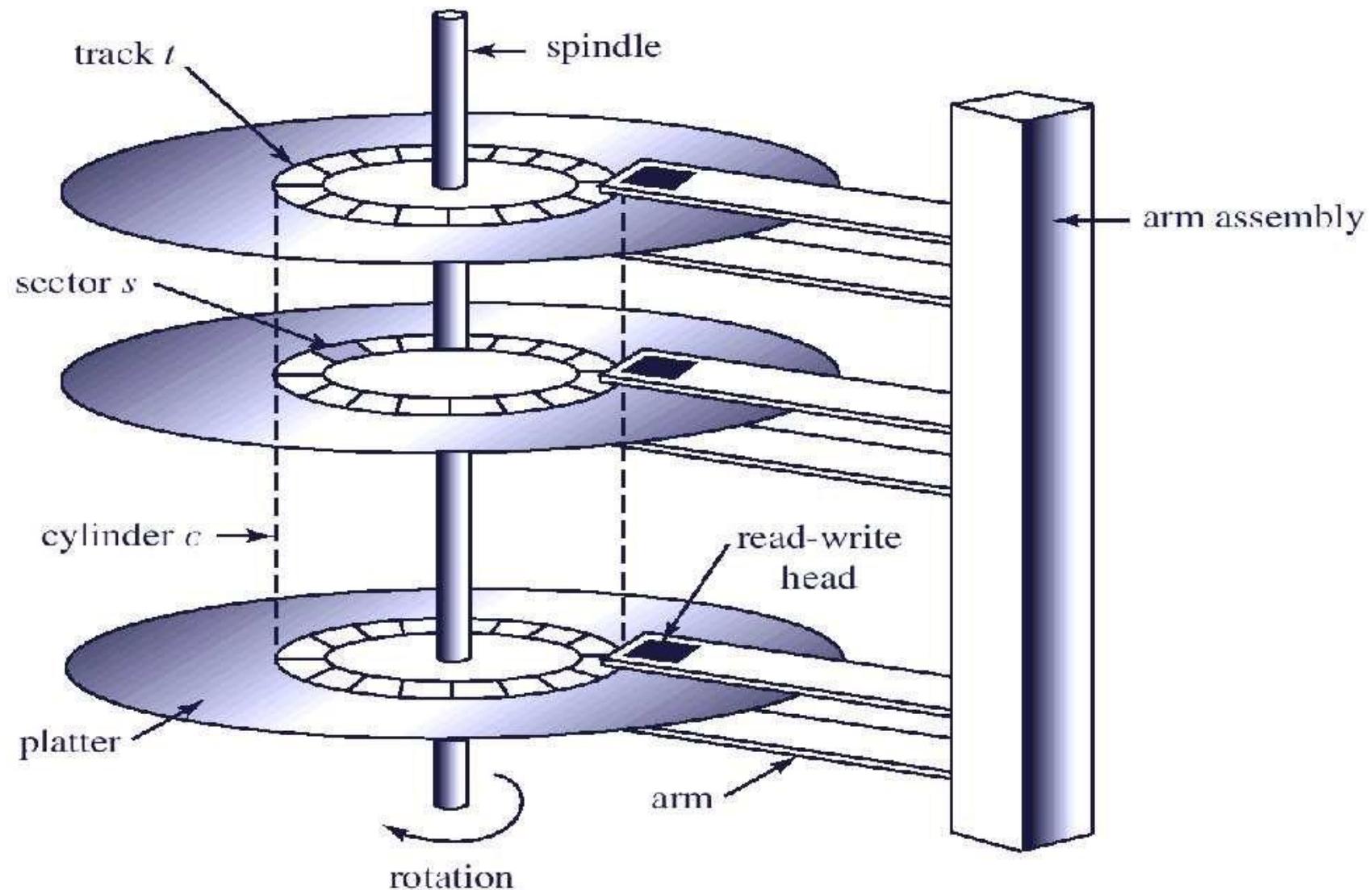
Disk Structure

Disks comes in many sizes and speeds, and information may be stored optically or magnetically; however, all disks share a number of important features.

For Example: floppy disks, hard disks, CD-ROMs and DVDs. Disk surface is divided into number of logical block called *sectors* and *tracks*.

The term *cylinder* refers to all the tracks at particular head position in hard disk.

Hard-Disk Structure



Disk Operations

Latency Time: The time taken to rotate from its current position to a position adjacent to the readwrite head.

Seek: The processes of moving the arm assembly to new cylinder.

To access a particular record, first the arm assembly must be moved to the appropriate cylinder, and then rotate the disk until it is immediately under the read-write head. The time taken to access the whole record is called *transmission time*.

Disk Scheduling

OS is responsible to use the hardware efficiently – for the disk drive this means fast seek, latency and transmission time.

For most disks, the *seek time* dominates the other two times, so reducing the *mean seek time* can improve system performance substantially.

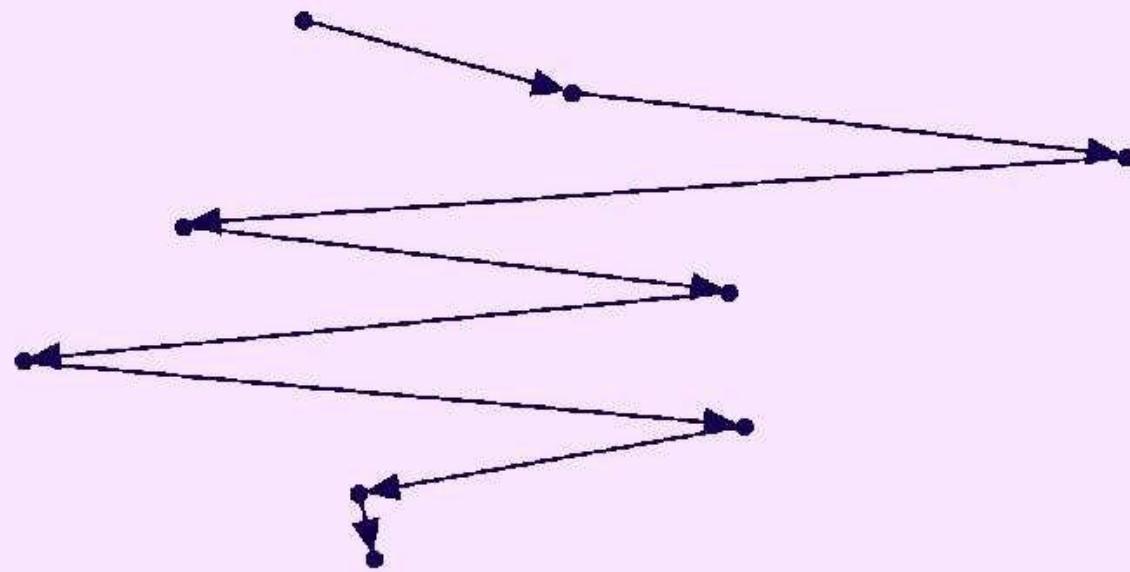
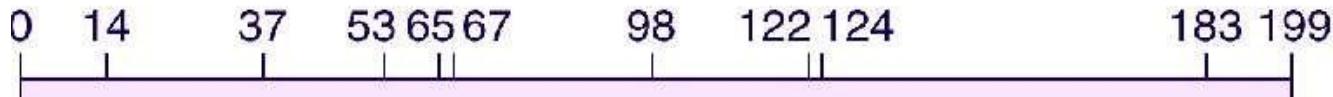
First-Come First-Served (FCFS)

The first request to arrive is the first one serviced.

Example:

Total Head
Movement =
640 cylinders

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53



Advantages:

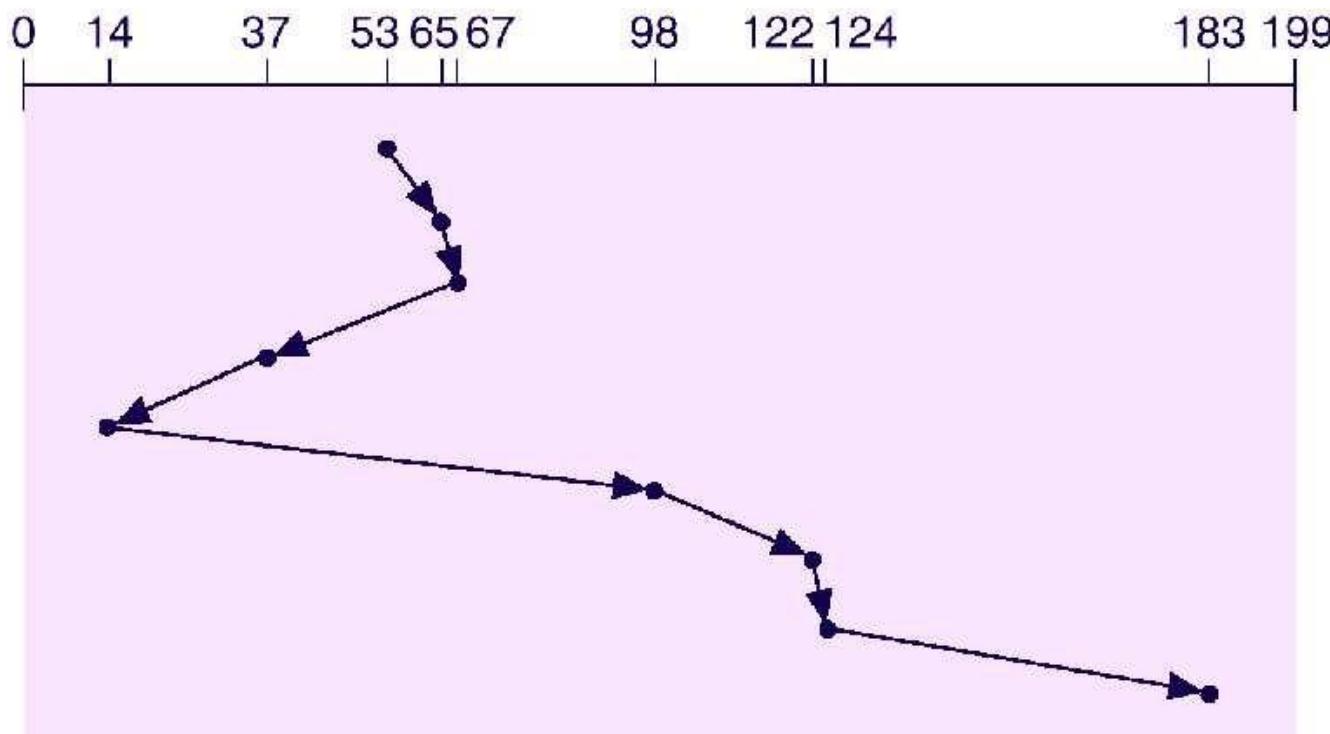
Simple and
Fair.

Problems: Does not provide fastest service.

Shortest-Seek-Time-First (SSTF)

*Selects the request with the minimum seek time
from the current head position.*

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53



Total Head Movement = 236 cylinders

Shortest-
Seek-
Time-
First

(SSTF)

Advantages:

Gives a substantial improvement in performance.

Problems:

SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.

Not optimal.

Used in batch system where throughput is the major consideration but unacceptable in interactive system.

SCAN

The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the

head movement is reversed and servicing continues.

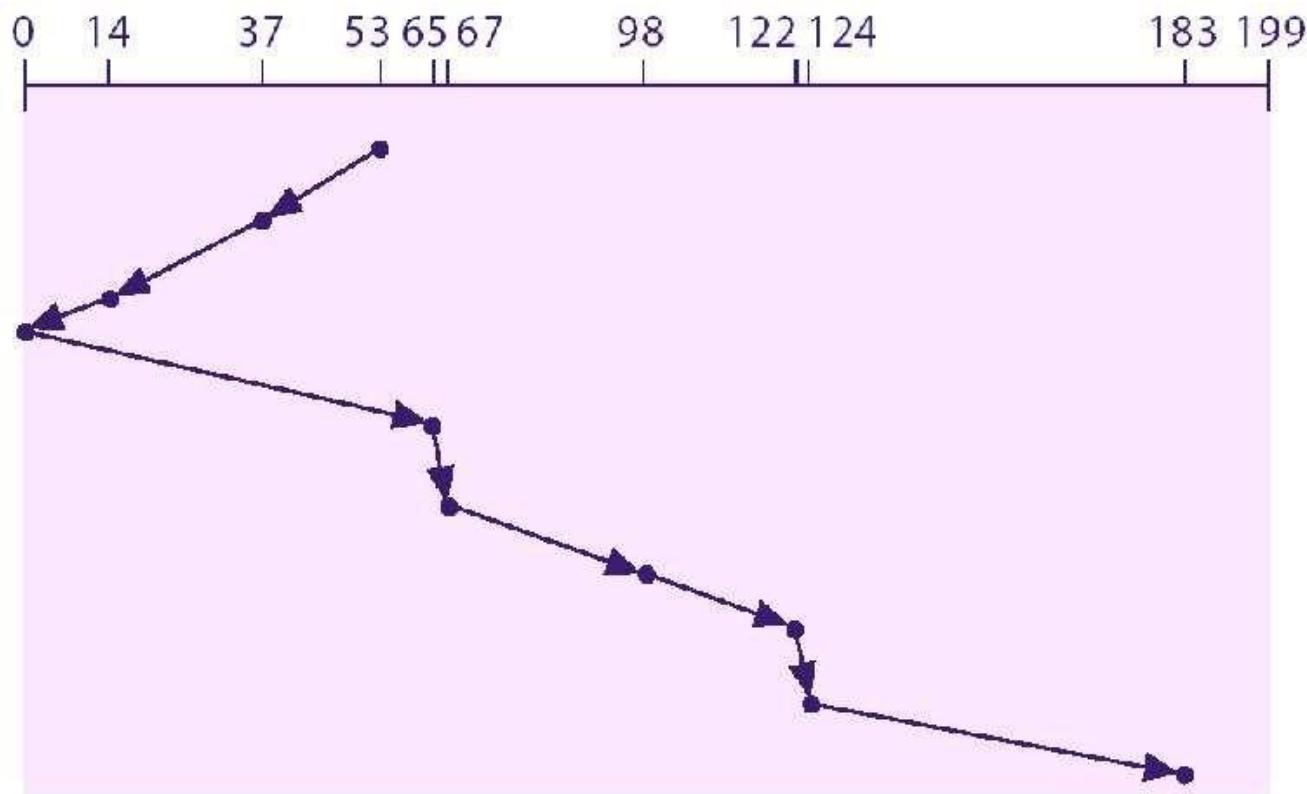
Sometimes called the *elevator algorithm*.

Advantages: Decreases variances in seek and improve response time.

Problem: Starvation is possible if there are repeated request in current track.

SCAN

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53



Total Head Movement = 208 cylinders

C-SCAN

When a uniform distribution of request for cylinders, only the few requests are in extreme cylinders, since these cylinders have recently been serviced.

Why not go to the next extreme?

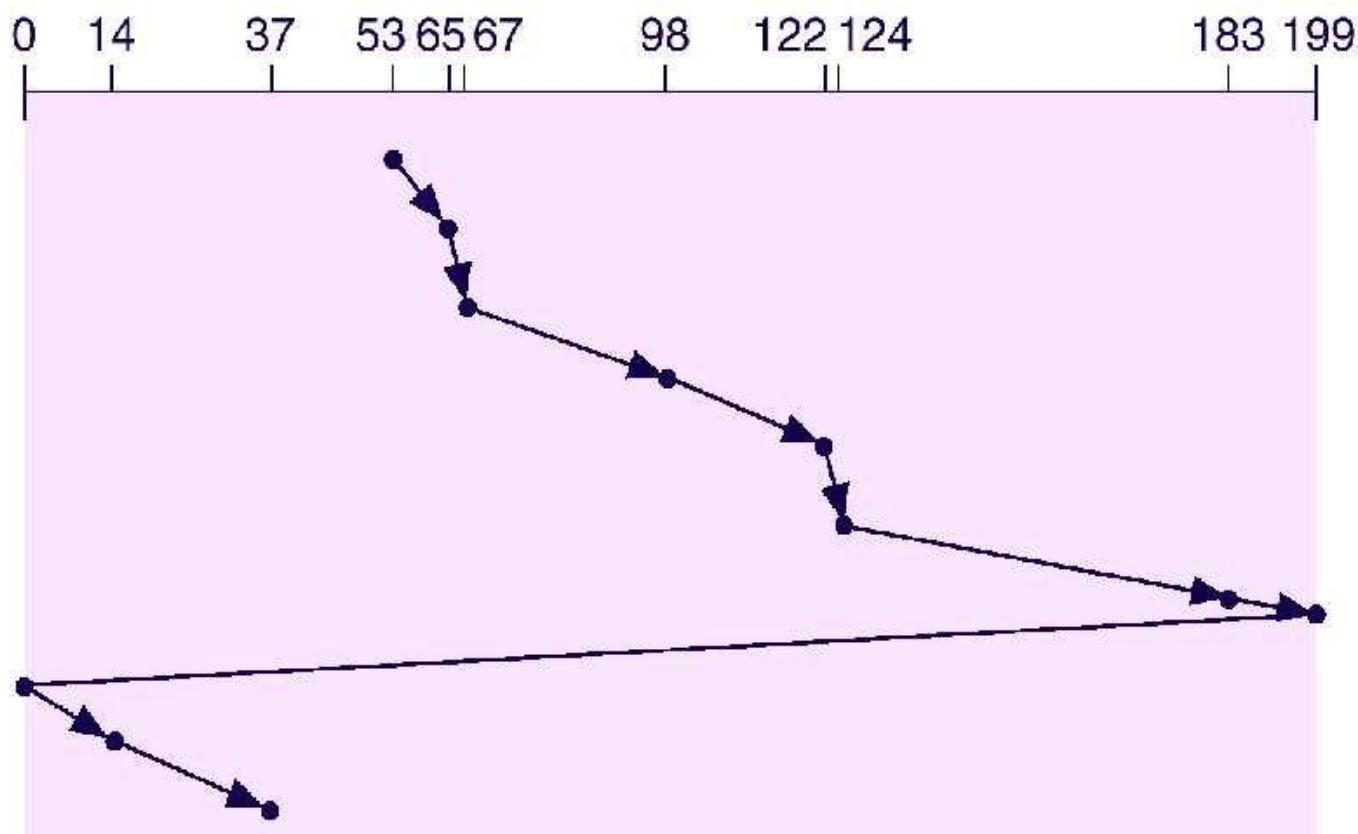
Circular SCAN is a variant of SCAN designed to provide a more uniform wait time.

The head moves from one end of the disk to the other. Servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.

Treats the cylinders as a circular list that wraps around from the last cylinder to the first one.

C-SCAN

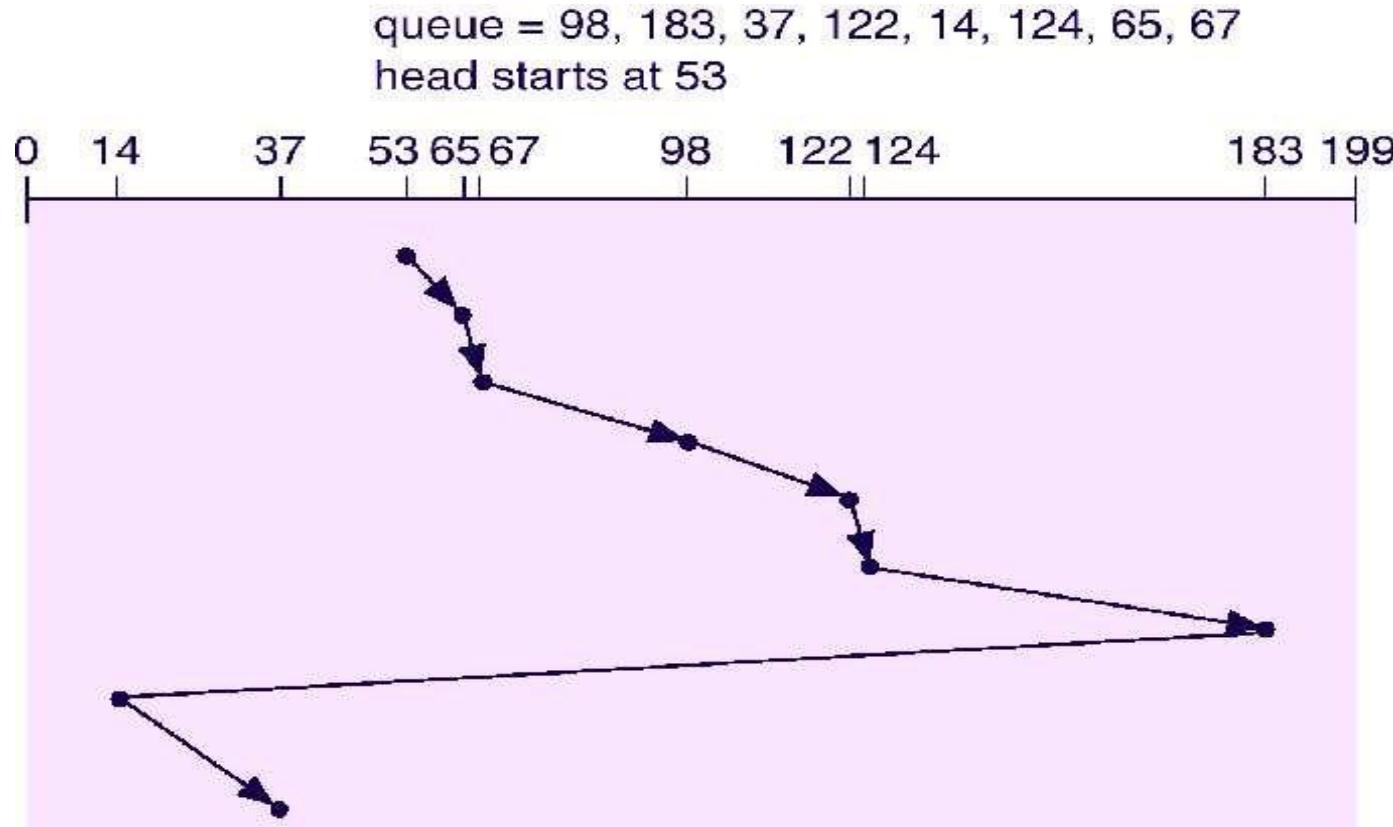
queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53



C-LOOK

Version of C-SCAN

Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.



RAID

Redundant Array of Inexpensive (Independent) Disks.

Issues: Disk performance, Amount of storage required & Reliability

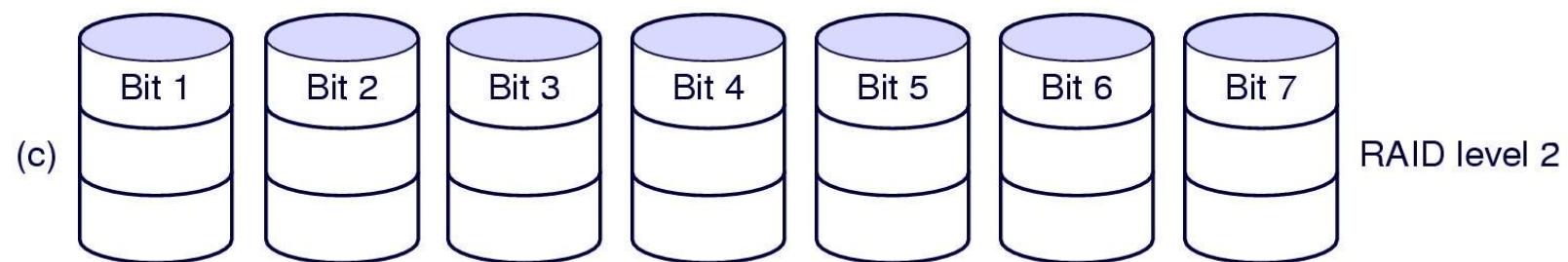
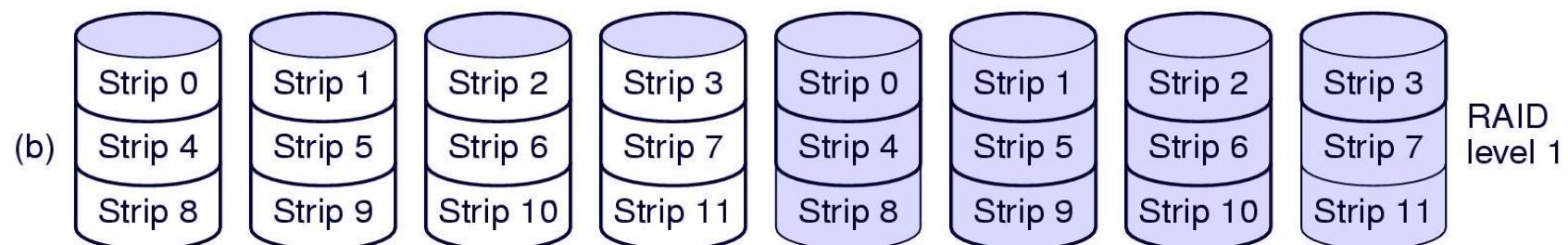
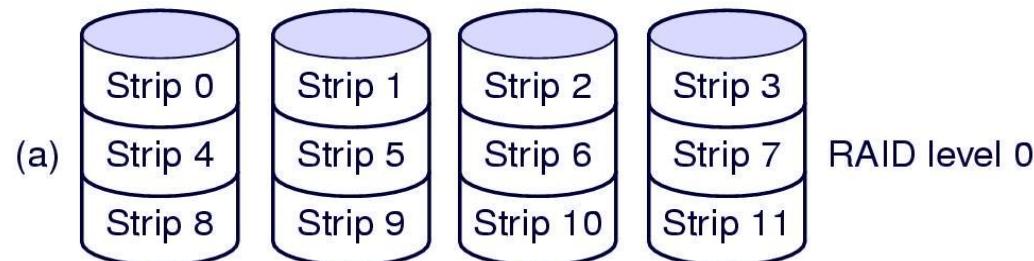
A *technique of organizing multiple disks to address above issues is RAID.*

RAID allows more than one disk to be used for a given operation, and allows continued operation and even automatic recovery in the face of disk failure.

Implemented in hardware or in OS.

RAID Levels

There are six types of organizations called RAID Levels.



RAID Levels

RAID Level 0:

RAID level 0 creates one large virtual disk from a number of smaller disks.

Storage is grouped into logical units called strips with the size of a strip being some multiple of sector size.

The virtual storage is sequence of strips interleaved among the disks in the array.

Advantages: Can create large disk; Performance benefit can be achieved.

Disadvantages: Reliability decrease.

RAID Levels

RAID Level 1:

Stores duplicate copy of each strip, with each copy on a different disk.

Advantages: Excellent reliability; if drive crashes, the copy is used. Read performance can be achieved.

Disadvantages: Write Performance is no better than in single drive.

RAID Levels

RAID Level 2:

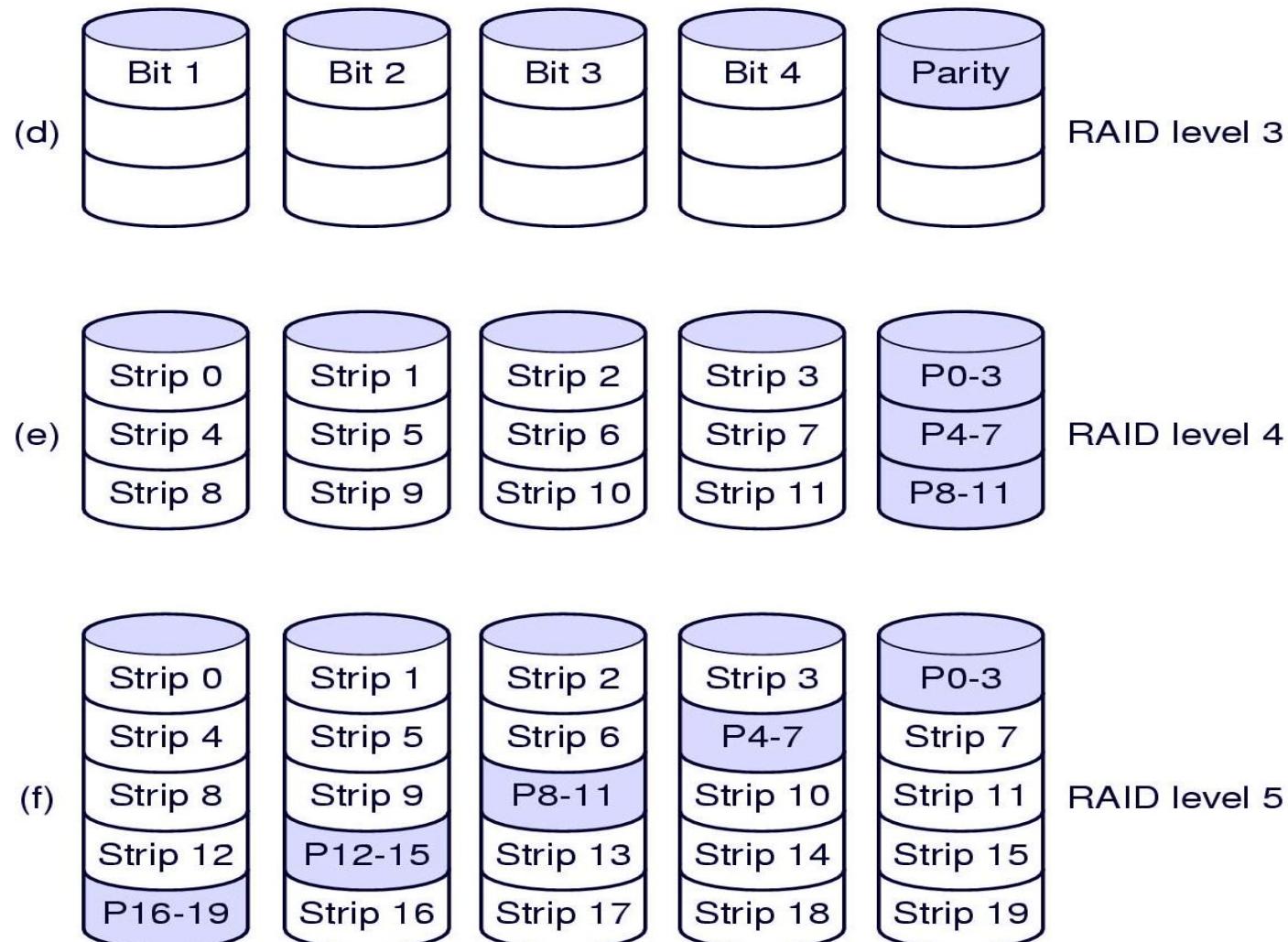
An error-correcting code is used for corresponding bits on each data disks.

Error-correcting scheme store two or more extra bits, and can reconstruct the data if a single bit get damaged.

For Example, the first bit of each byte is stored in disk 1, second bit in disk 2, and until eight bit in disk 8, and error correcting bits are stored in further disks. If one of the disk fail, the remaining bits of the byte and associated error-correction bits can be read from other disks and be used to reconstruct the damage data. Advantages: Total parallelism.

Disadvantages: Requires substantial number of drives.

RAID Levels



RAID Levels

RAID Level 3

Simplified version of Level 2.

A single parity bit is used instead of error-correcting code, hence required just one extra disk. If any disk in the array fails, its data can be determined from the data on the remaining disks.

It is as good as Level 2 but is less expensive in the number of extra disks.

RAID Levels

RAID Level 4

It uses block-level striping, as in Level 0, and in addition keeps a parity block on separate disk for corresponding blocks from other disks.

If one of the disks fails, the parity block can be used with the corresponding blocks from other disks to restore the blocks of the fail disks.

The transfer rate for large read as well as large write is high since reads and writes in parallel but small read and write can not be in parallel.

Problem: Parity bottleneck

RAID Levels

RAID Level 5

Similar to level 4 but parity information is distributed in all disks.

For each block one of the disk stores parity and other stores data.

For example, with an array of five disks, the parity for nth blocks is stored in disks $(n \bmod 5) + 1$; the nth block of the other four disks stores actual data for that block.

Disk Formatting

Before a disk can store data, it must be divided into sectors that the disk controller can read and write, called low-level formatting.

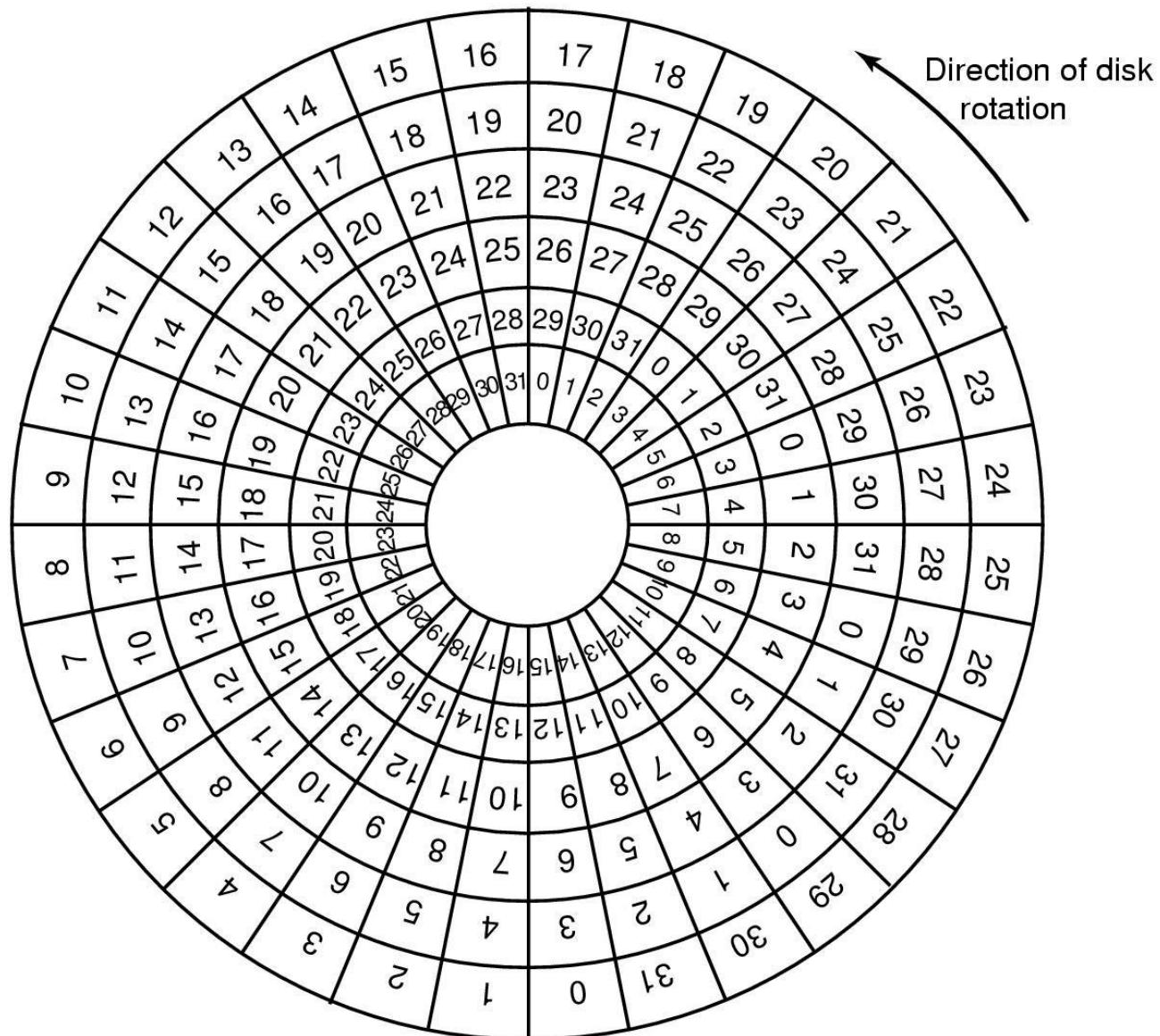
The sector typically consists of preamble, data and ECC. The preamble contains the cylinder and sector number and the

ECC contains redundant information that can be used to recover from read error.

The size depends upon the manufacturer, depending on reliability.

Preamble	Data	ECC
----------	------	-----

Disk Formatting



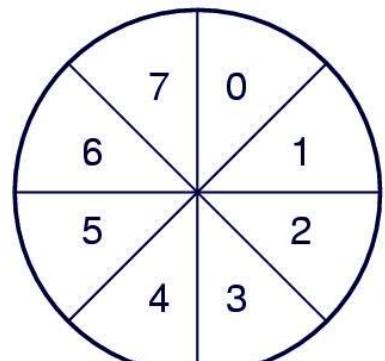
Disk Formatting

If disk I/O operations are limited to transferring a single sector at a time, it reads the first sector from the disk and doing the ECC calculation, and transfers to main memory, during this time the next sector will fly by the head.

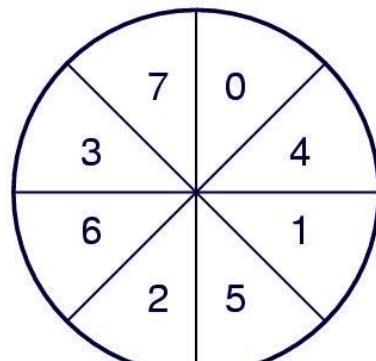
When transferring completes the controller will have to wait almost an entire rotation for the second sector to come around again.

This problem can be eliminated by numbering the sectors in an interleaved fashion when formating the disk. According to the copying rate, interleaving may be of single or double.

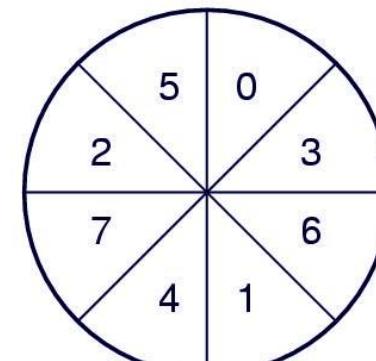
Disk Formatting



(a)



(b)



(c)

- a) No interleaving
- b) Single interleaving
- c) Double interleaving

Error Handling Bad Blocks

Most frequently, one or more sectors becomes defective or most disks even come from factory with bad blocks. Depending on the disk and controller in use, these blocks handled in variety of ways.

1. Bad blocks are handled manually- For example run MS-DOS chkdsk command to find bad block, and format command to create new block – data resided on bad blocks usually are lost.
2. Using bad block recovery- The controller maintains the list of bad blocks on the disk and for each bad block, one of the spares is substituted.

RAM Disks

RAM Disk is virtual block device created from main memory.
Commands to read or write disks blocks are implemented by RAM disk driver.
It completely eliminates seek and rotational delays suffered in disk devices.

RAM disks are particularly useful for storing files that are frequently accessed or temporary.

RAM disks are especially used in high performance applications. Some OS define the RAM disks at boot time, other dynamically.

Disadvantages: cost and volatility.

The volatility is solve by providing battery backups.

Home Works

HW #13:

1. 14, 15, 17, 19, & 24 from Text Book (Tanenbaum) Ch. 5.
2. Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of the pending requests, in FIFO order, is 86, 1470, 913, 1774, 948, 1502, 1022, 1750, 130.

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?

- a) FCFS b) SSTF c) SCAN d) C-SCAN e) C-LOOK
3. A disk has 8 sectors per track and spins at 600 rpm. It takes the controller 10ms from the end of one I/O operation before it can issue a subsequent one. How long does it take to read all 8 sectors using the following interleaving system?
- a) No interleaving b) Single interleaving c) Double interleaving

Files

Reading: Section 6.1-6.3 of Tanenbaum & Ch. 11& 12 from Silberschatz.

How to store the large amount of data into the computer?

What happens, when process terminates or killed using some data?

How to assign the same data to the multiple processes?

The solution to all these problems is to store information on disks or on other external media called files.

Files

A file is named collection of related information normally resides on a secondary storage device such as disk or tape.

Commonly, files represent programs (both source and object forms) and data; data files may be numeric, alphanumeric, or binary.

Information stored in files must be persistent,- not be effected by power failures and system reboot.

The files are managed by OS.

The part of OS that is responsible to manage files is known as the file system.

Files System Issues

How to *create* files?
How they *named*?
How they are *structured*?
What *operation* are allowed on files?
How to *protect* them?
How they *accessed* or *used*?
How to *implement*?

File Naming

When a process creates a file, it gives the file name; while process terminates, the file continue to exist and can be accessed by other processes.

A file is named, for the convenience of its human users, and it is referred to by its name. A name is string of characters.

The string may be of digits or special characters (eg. 2, !, % etc). Some system differentiate between the upper and lower case character, whereas other system consider the equivalent (like Unix and MS-DOS).

File Naming

Normally the string of max 8 characters are legal file name (e.g., in DOS), but many recent system support as long as 255 characters (eg. Windows 2000).

Many OSs support two-part file names; separated by period; the part following the period is called the file extension and usually

indicates something about the file (e.g., file.c – C source file). But in some system it may have two or more extension such as in Unix proc.c.Z - C source file compressed using Ziv-Lempel algorithm.

In some system (e.g., Unix), file extension are just conventions; in other system it requires (e.g., C compiler must requires .c source file).

File Structure

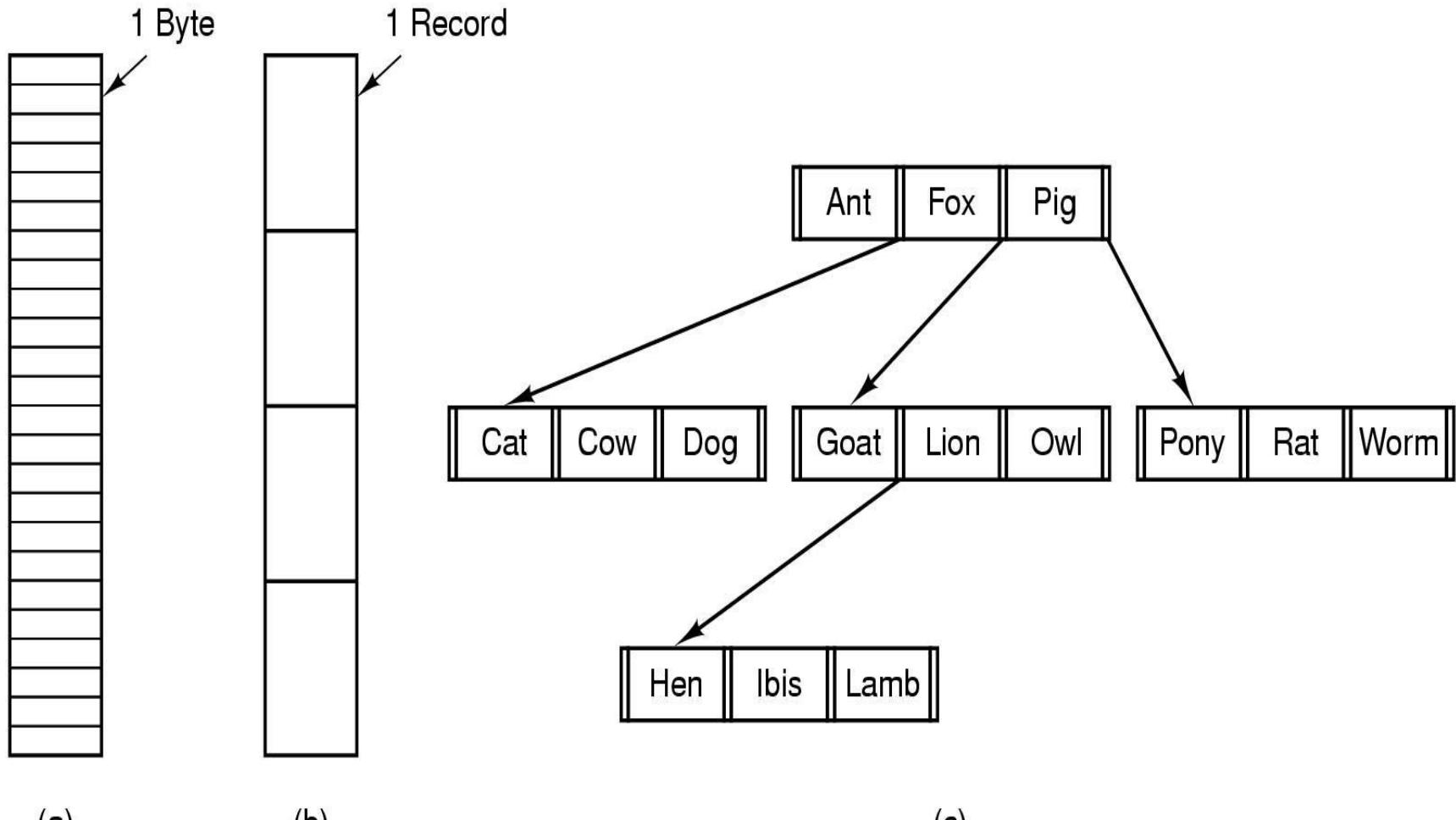
Files must have structure that is understood by OS.
Files can be structured in several ways. The most common structures are:

Unstructured

Record Structured

Tree Structured

File Structure



(a)

(b)

(c)

a) Unstructured b) Record structured c) Tree structured

File Structure

Unstructured:

Consist of unstructured sequence of bytes or words. OS does not know or care what is in the file. Any meaning must be imposed by user level programs.

Provides maximum flexibility; user can put anything they want and name them anyway that is convenient.

Both Unix and Windows use these approaches.

File Structure Record

Structured:

A file is a sequence of fixed-length records, each with some internal structure.

Each read operation returns one records, and write operation overwrites or append one record.

Many old mainframe systems use this structure.

Tree Structured:

File consists of tree of records, not necessarily all the same length.

Each containing a key field in a fixed position in the record, sorted on the key to allow the rapid searching.

The operation is to get the record with the specific key.

Used in large mainframe for commercial data processing.

File Types

Many OS supports several types of files

Regular files: contains user information, are generally ASCII or binary.

Directories: system files for maintaining the structure of file system.

Character Special files: related to I/O and used to model serial I/O devices such as terminals, printers, and networks.

Block special files: used to model disks.

File Types

ASCII files:

Consists of line of text.

Each line is terminated either by carriage return character or by line feed character or both.

They can be displayed and printed as is and can be edited with ordinary text editor.

Binary files:

Consists of sequence of byte only.

They have some internal structure known to programs that use them (e.g., executable or archive files).

Many OS use extension to identify the file types; but Unix like OS use a magic number to identify file types.

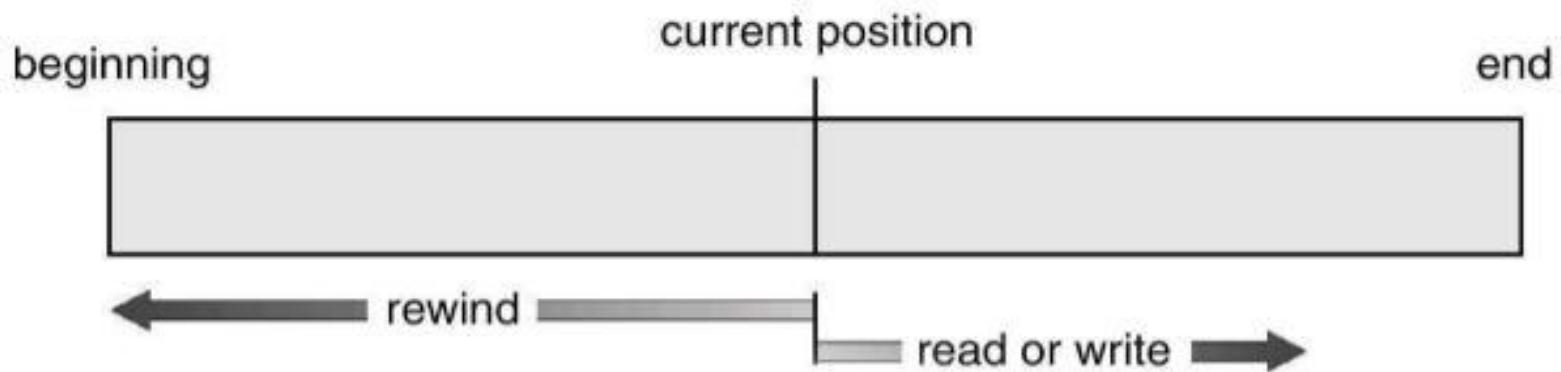
Access Methods Access

Sequential and Direct access

Sequential Access:

The simplest excess method; Information in the file is processed in order, one record after the other.

Convenient when the storage medium is magnetic tap.
Used in many early systems.



Methods

Direct Access:

Files whose bytes or records can be read in any order.
Based on disk model of file, since disks allow random access to any block.

Used for immediate access to large amounts of information.

When a query concerning a particular subject arrives, we compute which block contain the answer, and then read that block directly to provide desired information.

Operations: *read n*, *write n* (n is block number) or *seek* – to set current position. File can be access sequentially from the current position.

File Attributes

In addition to name and data, all other information about file is termed as file attributes.

The file attributes may vary from system to system.
Some common attributes are listed here.

File Attributes

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

File Operations

OS provides system calls to perform operations on files. Some common calls are:

Create: If disk space is available it creates new file without data.

Delete: Deletes files to free up disk space.

Open: Before using a file, a process must open it.

Close: When all access are finished, the file should be closed to free up the internal table space.

Read: Reads data from file.

File Operations

Append: Adds data at the end of the file.

Seek: Repositions the file pointer to a specific place in the file.

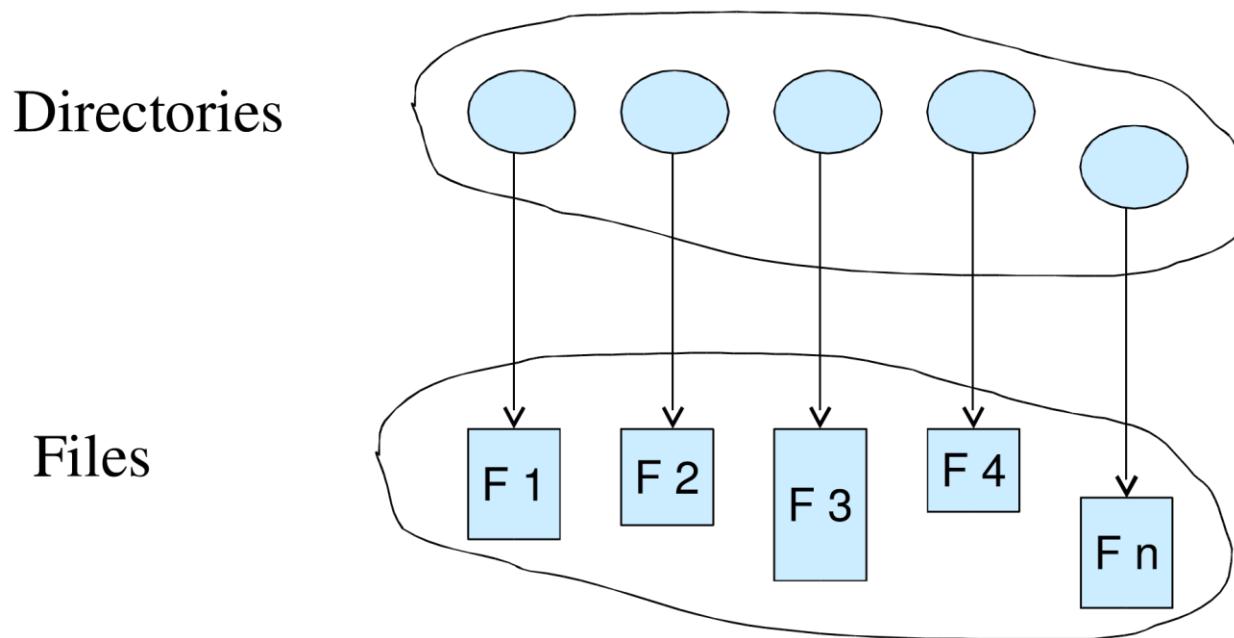
Get attributes: Returns file attributes for processing.

Set attributes: To set the user settable attributes when file changed.

Rename: Rename file.

Directory Structure

A directory is a node containing informations about files.



Directories can have different structures

Directory Structure

Single-Level-Directory:

All files are contained in the same directory.
Easy to support and understand; but difficult to manage large amount of files and to manage different users.

Two-Level-Directory:

Separate directory for each user.
Used on a multiuser computer and on a simple network computers.
It has problem when users want to cooperate on some task and to access one another's files. It also cause problem when a single user has large number of files.

Directory Structure

Hierarchical-Directory:

Generalization of two-level-structure to a tree of arbitrary height.

This allow the user to create their own subdirectories and to organize their files accordingly.

To allow to share the directory for different user acyclic-graph-structure is used.

Nearly all modern file systems are organized in this manner.

Path Names

Absolute and Relative path names.

Absolute Path Name:

Path name starting from root directory to the file.

e.g. In Unix: /usr/user1/bin/lab2.

Path separated by / in Unix and \ in windows.

Relative Path Name:

Concept of working directory.

A user can designate one directory as the current working directory, in which case all path names not beginning at the root directory are taken relative to the working directory.

E.g., bin/lab2 is enough to locate same file if current working directory is /usr/user1.

File-System Implementation

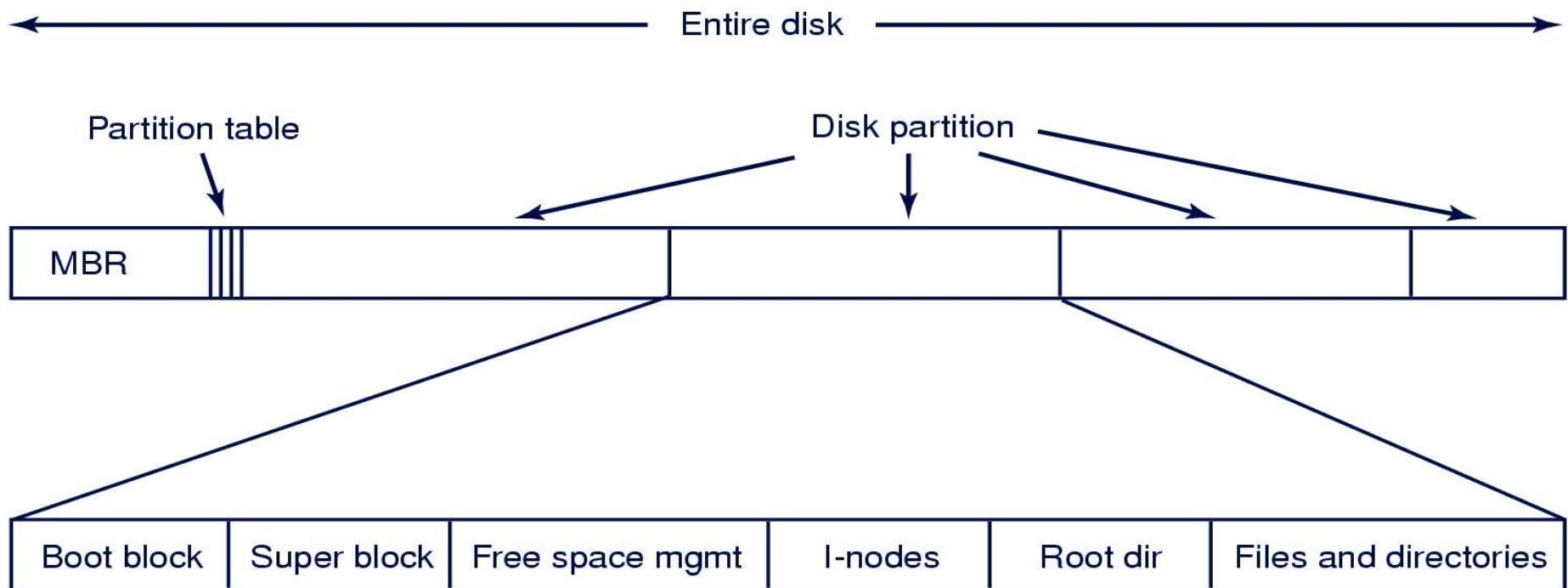
How files and directories are stored?

How disk space is managed?

How to make every thing work efficiently and
reliably?

File-System Layout

Disks are divided up into one or more partitions,
with independent file system on each partition.



Allocation Methods

Contiguous Allocation

Each file occupy a set of contiguous block on the disk.

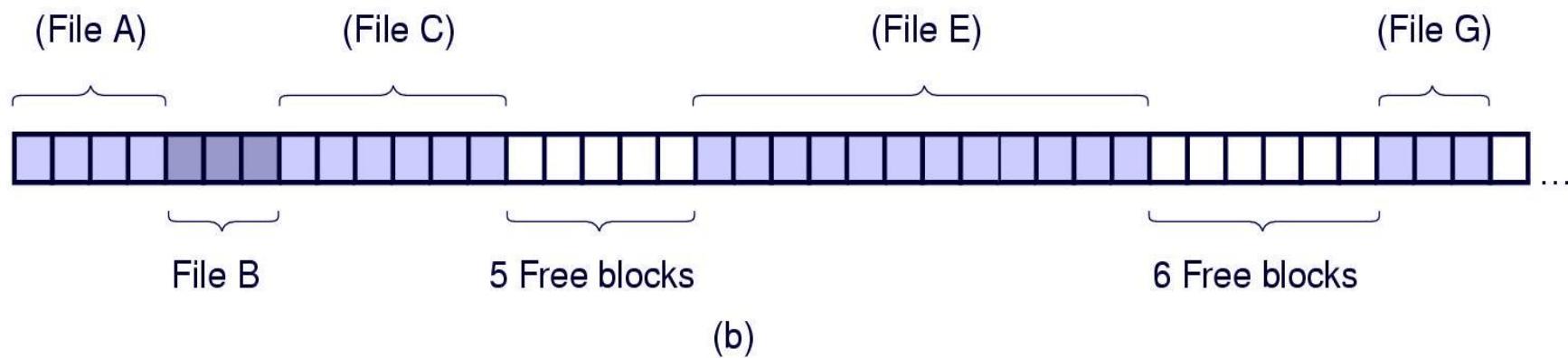
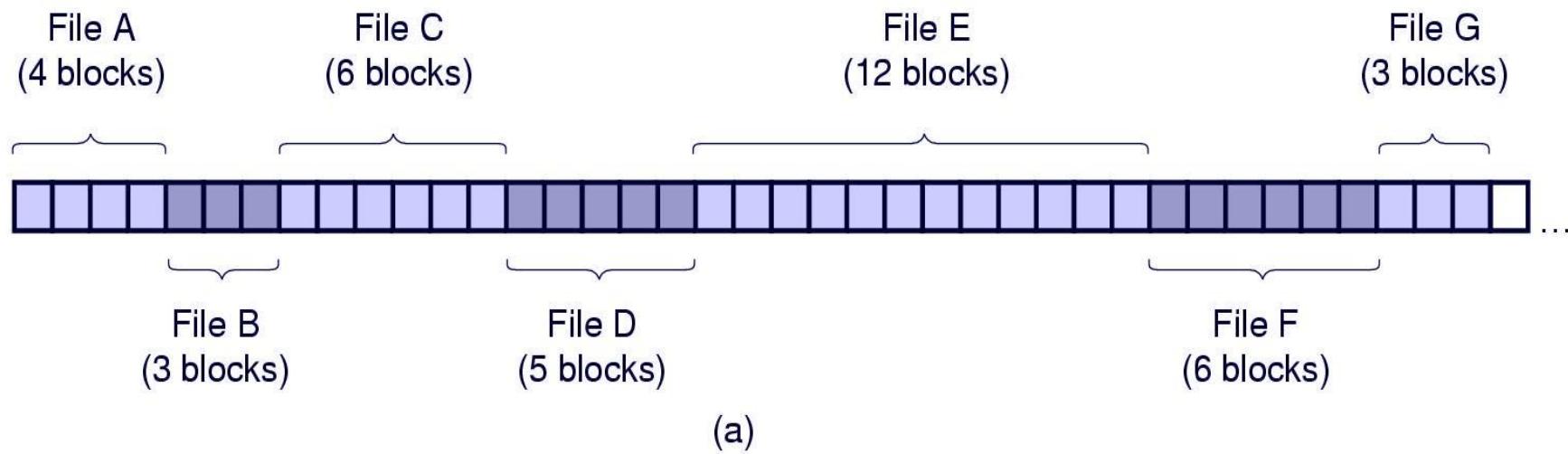
Disk addresses define a linear ordering on the disk.
File is defined by the disk address and length in block units.

With 2-KB blocks, a 50-KB file would be allocated 25 consecutive blocks.

Both sequential and direct access can be supported by contiguous allocation.

Allocation Methods

Contiguous Allocation



Allocation Methods

Contiguous Allocation Advantages:

Simple to implement; accessing a file that has been allocated contiguously is easy.

High performance; the entire file can be read in single operation i.e. decrease the seek time.

Problems:

fragmentation: when files are allocated and deleted the free disk space is broken into holes.

Dynamic-storage-allocation problem: searching of right holes.

Required pre-information of file size.

Due to its good performance it used in many system; it is widely used in CD-ROM file system.

Allocation Methods

Linked Allocation

Each file is a linked list of disk blocks; the disk block may be scattered anywhere on the disk.

Each block contain the pointer to the next block of same file.

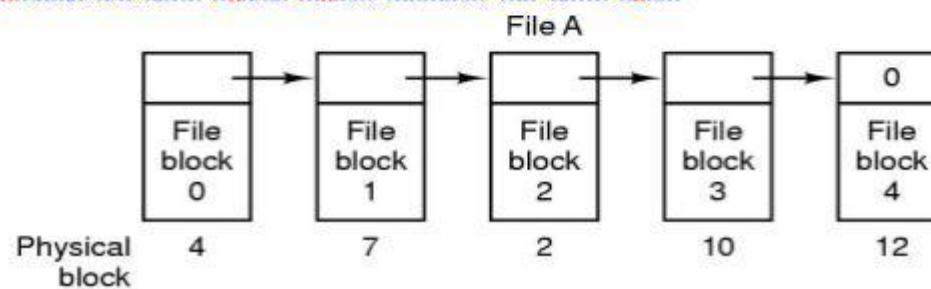
To create the new file, we simply create a new entry in the

directory; with linked allocation, each directory entry has a

Each block contain the pointer to the next block of same file.

To create the new file, we simply create a new entry in the directory; with linked allocation, each directory entry has a

pointer to the first disk block of the file.



Allocation Methods

Linked Allocation

Problems:

It solves all problems of contiguous allocation but it can be used only for sequential access file; random access is extremely slow.

Each block access required disk seek.

It also required space for pointer.

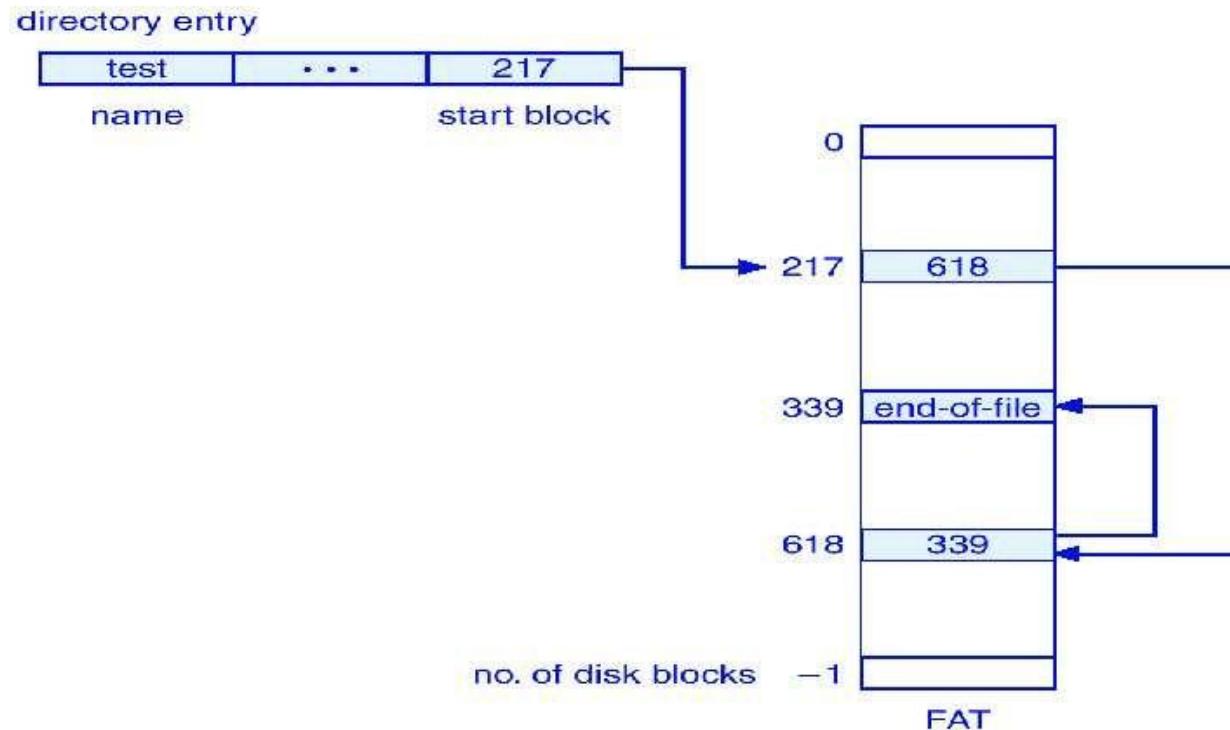
Solution: Using File Allocation Table (FAT).

The table has one entry for each disk block containing next block number for the file. This resides at the beginning of each disk partition.

Allocation Methods

Linked Allocation using FAT

The FAT is used as is a linked list. The directory entry contains the block number of the first block of the file. The FAT is looked to find next block until a special end-of-file value is reached.



Allocation Methods

Linked Allocation using FAT Advantages:

The entire block is available for data.

Result the significant number of disk seek; random access time is improved.

Problems:

The entire table must be in memory all the time to make it work.

With 20GB disk and 1KB block size, the table needs 20 millions entries. Suppose each entry requires 3 bytes. Thus the table will take up 60MB of main memory all the time.

Allocation Methods

Index Allocation

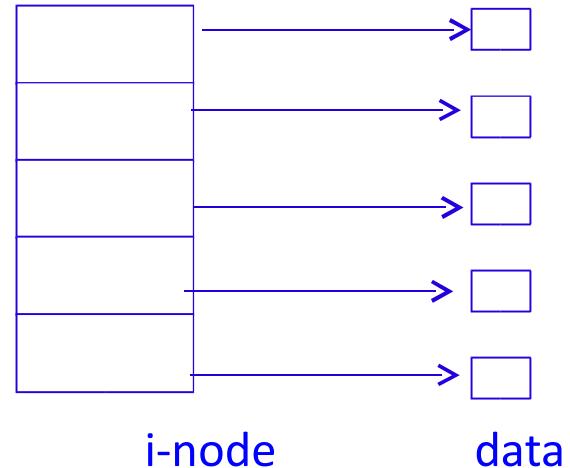
To keep the track of which blocks belongs to which file, each file has data structure (i-node) that list the attributes and disk address of the disk block associate with the file.

Each i-node are stored in a disk block, if a disk block is not sufficient to hold i-node it can be multileveled.

Independent to disk size.

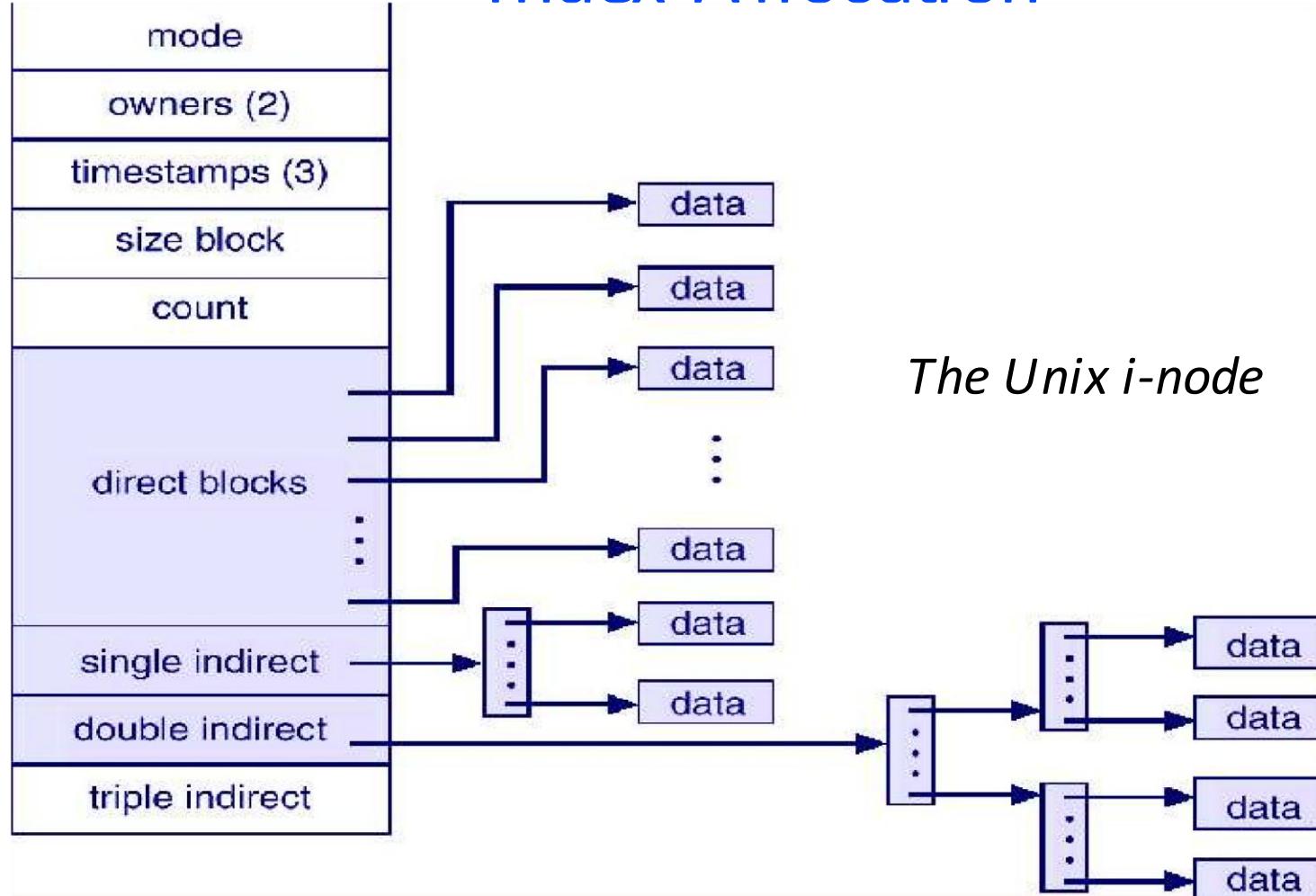
If i-node occupies n bytes for each file and k files are opened, the

total memory by i-nodes is kn bytes.



Allocation Methods

Index Allocation



Advantages: Far smaller than space occupied by FAT.

Problems: This can suffer from performance.

Directory Implementation

The directory entry provides the information needed to find the disk block of the file. The file attributes are stored in the directory. [Linear List](#)

Use the linear list of the file names with pointer to the data blocks.

It requires linear search to find a particular entry.

Advantages: Simple to implement.

Problem: linear search to find the file.

Directory Implementation Hash Table

It consist linear list with hash table. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list.

If that key is already in use, a linked list is constructed.

Advantages: greatly decrease the file search time.

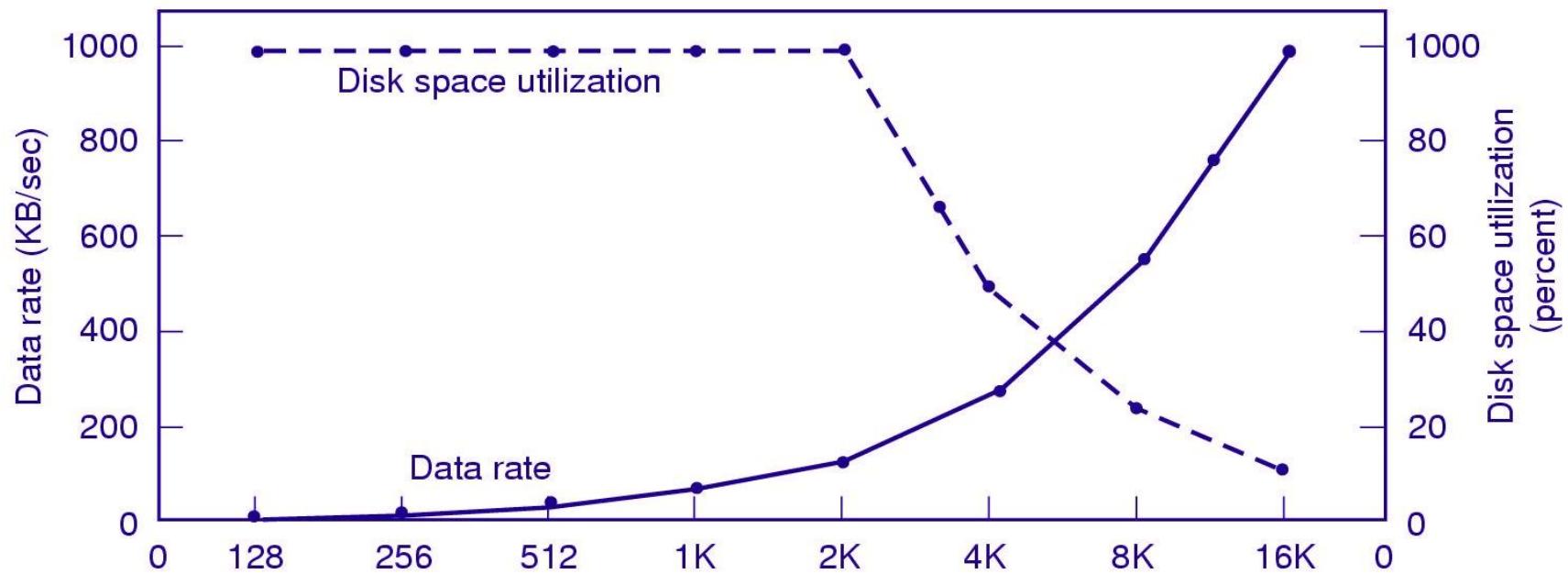
Problem: It greatly fixed size and dependence of the hash function on that size.

Block Size

Nearly all file systems chop files up into fixed-size block.

Using a large size result the wastage of disk space.

Using a small size increase the seek and rotational delay- low data access rate.



Dark line (left hand scale) gives data rate of a disk; Dotted line (right hand scale) gives disk space utilization. All files are 2KB

Free-Space Management

To keep track of free blocks, system maintains the freespace-list.
The free-space-list records all free blocks- those not allocated to some file or directory.

To create a file, system search the free-space-list for required amount of space, and allocate that space to the new file and removed from free-space-list.

When a file is deleted, its disk space is added to the freespace-list.

The free-space-list can be implemented in two ways:

Bitmap

Linked list

Free-Space Management

Bitmap

Each block is represented by a bit. If the block is free, the bit is 1; if the block is allocated the bit is 0.

A disk with n blocks requires a bitmap with n bits.

Ex: Consider a disk where blocks 2,3,4,5,8,9,10,11,12,13,17,18,.....

are free, and rest are allocated. The free space bit map would be

0011110011111100010.....

Advantages: Simple and efficient in finding first free block, or n consecutive free blocks.

Problems: Inefficient unless the entire bitmap is kept in main memory.

Keeping in main memory is possible only for small disk, when disk is large the bitmap would be large.

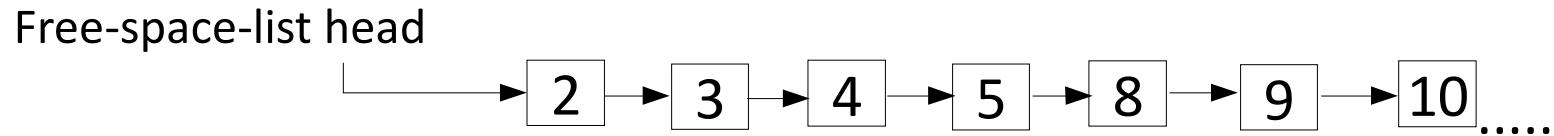
Many system use bitmap method.

Free-Space Management Linked List

Link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.

The first block contains a pointer to the next free block.

The previous example can be represented in linked as



Advantages: Only one block is kept in memory.

Problems: Not efficient; to traverse list, it must read each block.

Home Works

HW #11

1. 6, 7, 10, 15, 20, 22, 36 and 42 from your Textbook (Tanenbaum) Ch. 6.
2. How many bits would be needed to store the free-space-list under the following conditions if a bitmap were used to implement?
 - a) 500, 000 blocks total and 200, 000 free blocks.
 - b) 1,000,000 blocks total and 0 free blocks.

Also find how much space is required if it need to be stored in memory?

I/O Management

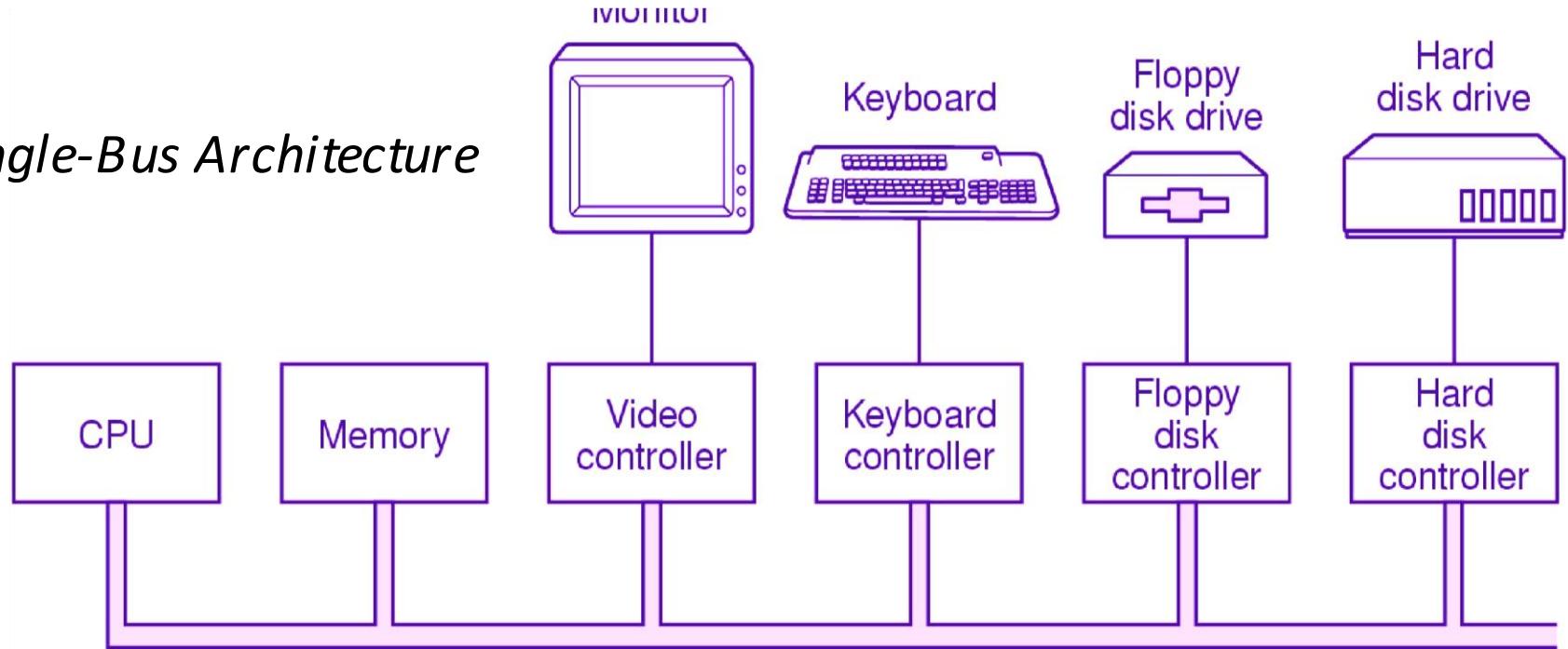
All computers have physical devices for acquiring input
and producing output.

*OS is responsible to manage and control all the
I/OOperations and I/O devices.*

Hardware Organization

The I/O devices, memory, and the CPU communicate with each other by way of one or more communication buses.

Single-Bus Architecture



Hardware Organization I/O Devices

The I/O units which consist mechanical components are called I/O devices such as hard-disk drive, printer etc.

There are two types of devices: *block* and *character* devices.

Block devices: Stores information in fixed-sized blocks, each one with its own address. Read or write is possible independent to other blocks-direct access.

Example: *disk*.

Character devices: Delivers or accepts a stream of characters without regard to any block structure. It is not addressable such as *printer*.

Some devices are neither block nor character such as clock.

Hardware Organization Device Controllers

A controller is a collection of electronics that can operate a bus or a device.

On PC, it often takes the form of printed circuit card that can be inserted into an expansion slot.

A single controller can handle multiple devices; some devices have their own built-in controller.

The controller has one or more registers for data and signals. The processor communicates with the controller by reading and writing bit patterns in these registers.

When transferring a disk block of size 512 bytes, the block first assembled bit by bit in a buffer inside the controller. After its checksum has been verified and the block declared to be error free, it can than be copied to main memory.

Hardware Organization

Memory-Mapped I/O

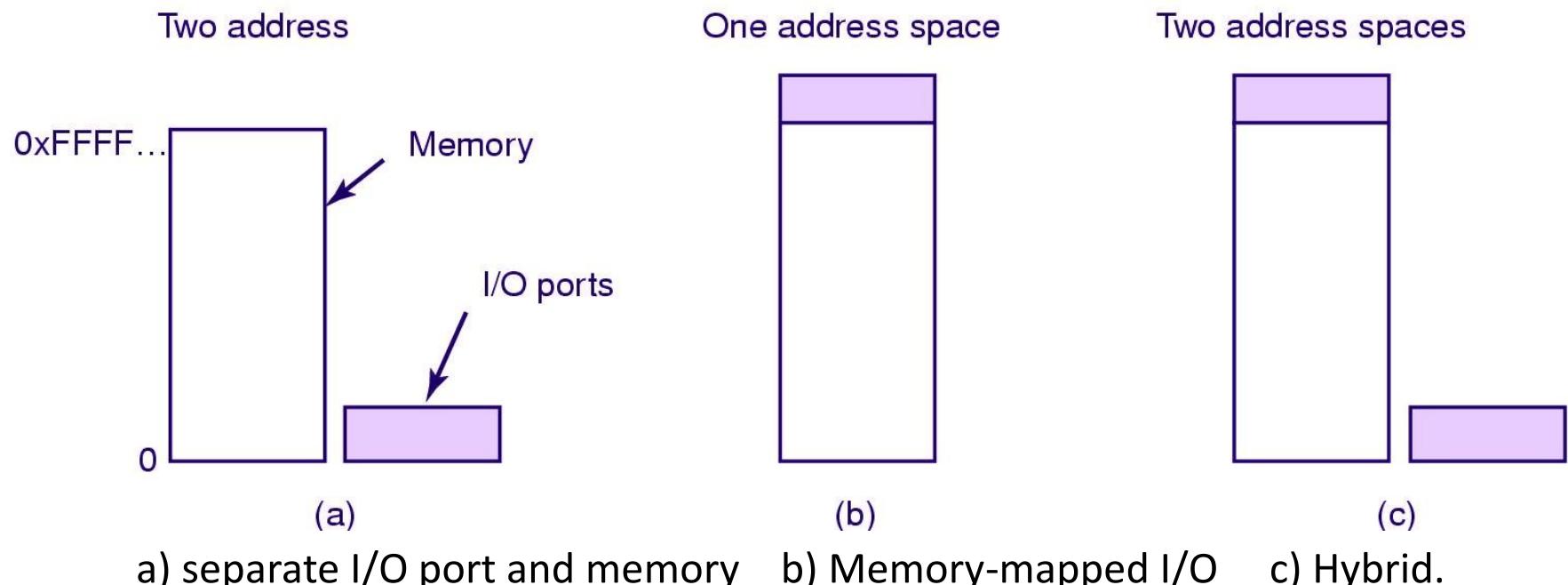
Device controller have their own register and buffer for communicating with the CPU, by writing and reading these register OS perform the I/O operation.

The device control registers are mapped into memory space, called memory-mapped I/O.

Usually, the assigned addresses are at the top of the address space. Some system (e.g. Pentium) use both techniques(hybrid).

The address 640K to 1M being reserved for device data buffers in addition to I/O ports 0 through 64K.

Hardware Organization Memory-Mapped I/O



When a CPU wants to read a word, either from memory or from I/O port, it puts the address needs on the bus' address line and then asserts the read signals on a bus' control line. A second signal is used to tell whether I/O or memory space is needed.

Hardware Organization Memory-Mapped I/O

Advantages:

Can be implemented in high-level languages such as C.

No separate protection mechanism is needed.

Every instruction that can reference the memory can also reference the control registers (in hybrid).

Disadvantages:

Adds extra complexity to both hardware and OS.

All memory modules and all I/O devices must examine all memory references to see which one to respond to.

Hardware Organization Direct-Memory-Access (DMA)

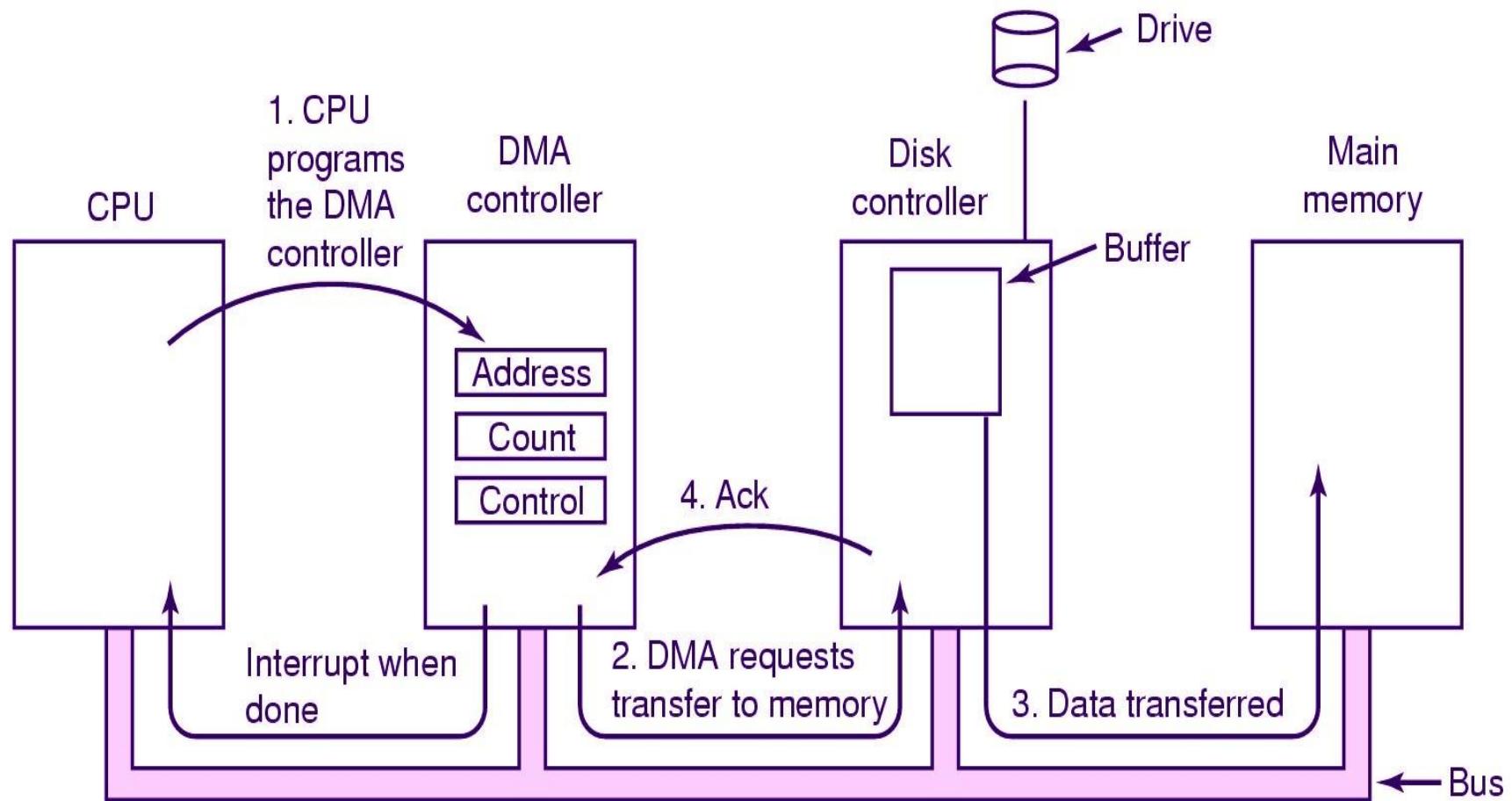
What happens when CPU directly reads data one byte at a time from an I/O controller?

Wastes the CPU's time (busy wait) - The solution is DMA.

Many computers avoid burdening the CPU by offloading some of its work to special propose processor called DMA controller. It consists memory address register, a byte count register, and one or more control registers.

More commonly, a single DMA controller is available (in-built in main-board) for regulating transfer to multiple devices but some systems may have integrated with disk controllers.

Hardware Organization Direct-Memory-Access (DMA)



Steps in DMA Transfer

Hardware Organization

Direct-Memory-Access (DMA) How it works?

1. The CPU programs the DMA controller by its registers so it knows what to transfer where.

It also issue a command to disk controller telling it to read data from disk to its internal buffer and verify the checksum. When valid data are in disk's controller buffer, DMA can begin.

2. The DMA controller initiates the transfer by issuing a read request over the bus to disk controller.
3. Data transferred from disk controller to memory.
4. When transferred completed, the disk controller sends an acknowledgment signal to DMA controller. The DMA controller then increments the memory address to use and decrement the byte count. This continues until the byte count greater than 0.
5. When transfer completed the DMA controller interrupt the CPU.

I/O Handling

Polling

Processing is interrupted at brief intervals to allow the CPU to check back to I/O to see if the I/O operation has completed.

Well manner way of getting attention.

Advantages: Simple

Problems: May bear long wait.

Can be a high overhead operation-inefficient.

Used in embedded system, where CPU has nothing else to do.

I/O Handling Interrupt

The hardware mechanism that enables a device to notify the CPU is called an interrupt.

Interrupt forced to stop CPU what it is doing and start doing something else.

Advantages: Improves efficiency.

Problem: *Because of the selfish nature, the first interrupt may not be served if the similar second happened before the time needed to serve the first.*

Example

Scenario: a cook is cooking something in modern microwave oven equipped kitchen.

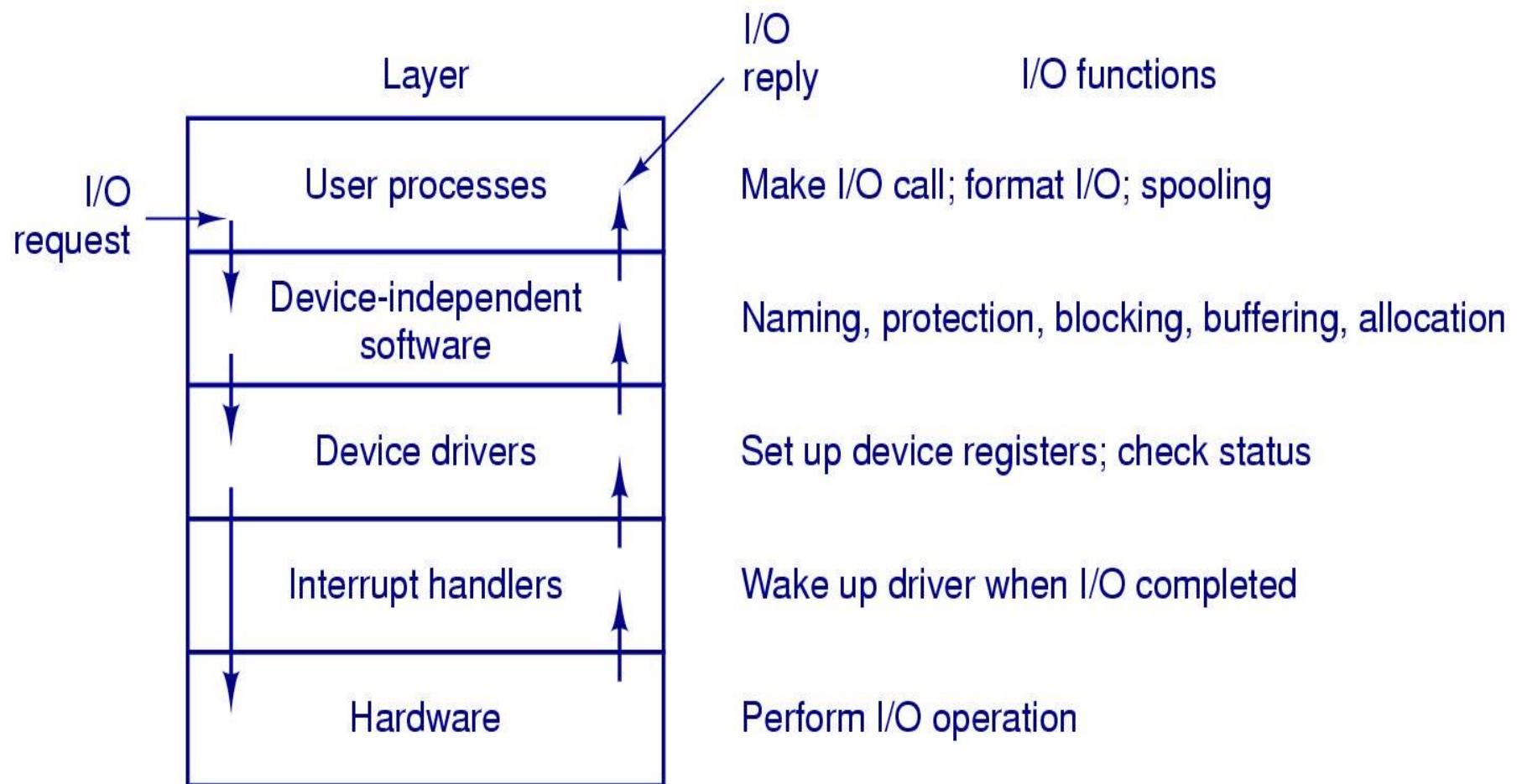
CASE A: The cook may regularly peek through the oven's glass door and watch as roast cooks under cook and watch; this kind of regular monitoring is *polling*.

CASE B: The cook may set a timer to expire after an appropriate number of minutes; the buzzer sounding after this interval is an *interrupt*.

I/O Software Issues

Device Independence.
Uniform Naming.
Error Handling.
Synchronous vs. Asynchronous transfer.
Buffering.

Layer Structure



Interrupt Handlers

Block the driver until the I/O has completed and the interrupt occur.

The interrupt-handler determines the cause of the interrupt and performs the necessary processing.

1. Save any registers (including the PSW) that have not already been saved by the interrupt hardware.
2. Set up a stack for interrupt service procedure.
3. Ack interrupt controller.
4. Copy registers from where they were saved (stack) to the process table.
5. Run the interrupt service procedure.
6. Set up the MMU context for the process to run next.
7. Load new process' registers, including PSW.
8. Start running the new process.

Devices Drivers

Encapsulate detail of devices.

Each I/O device attached to a computer needs some device-specific code for controlling it, called device driver, is generally written by the device's manufacturer and delivered along with the device. Each device driver normally handles one device type, or at most one class of closely related devices.

In some systems, the OS is a single binary program that contains all of the drivers that it will need compiled into it (e.g., UNIX).

Devices Drivers

Functions:

Accept read and write requests from the deviceindependent software above it.

Initialize the devices if necessary.

Manage power requirement and log events.

It checks the status of devices- in use or free. Decides which command to issue if there is command queue.

Device-Independent OS software

The devices independent-software is to perform the I/O functions that are common to all devices and

to provide a uniform interface to the user-level-software.

Some common functions are:

Uniform Interfacing for devices drivers-naming and protection.

Buffering.

Error reporting.

Allocating and releasing dedicated devices.

Providing a device-independent block size.

User Processes

System calls, including the I/O system calls are normally made by the user processes. User

processes put their parameters in the appropriate place for the system calls, or other procedures that actually do real work.

Home Works

HW #12

1. 4, 5, 8, 9, 10 & 13 from Textbook (Tanenbaum) Ch. 5.
2. How does DMA increase system concurrency? How does it complicate the hardware design?
3. Which one suited, polling/interrupt, for the following types of system? Give reason.
 - a) A system dedicated to controlling a single I/O devices.

- b) A personal computer running a single-tasking OS.
- c) A work station running as heavily used web server.