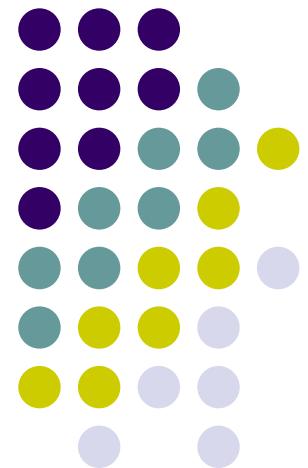


6TH SEM: Operating System

Muktikanta Sahu
Spring 2020



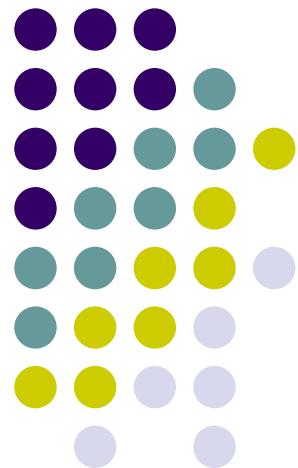


General Information

- Textbook:
 - Operating System Concepts, 8th Ed, by Silberschatz, Galvin, and Gagne
 - Materials from other books as and when needed

- Grading Policy
 - * Midsem – 30%
 - * Quiz and TA - 20%
 - * Endsem – 50%

Introduction





What is an Operating System?

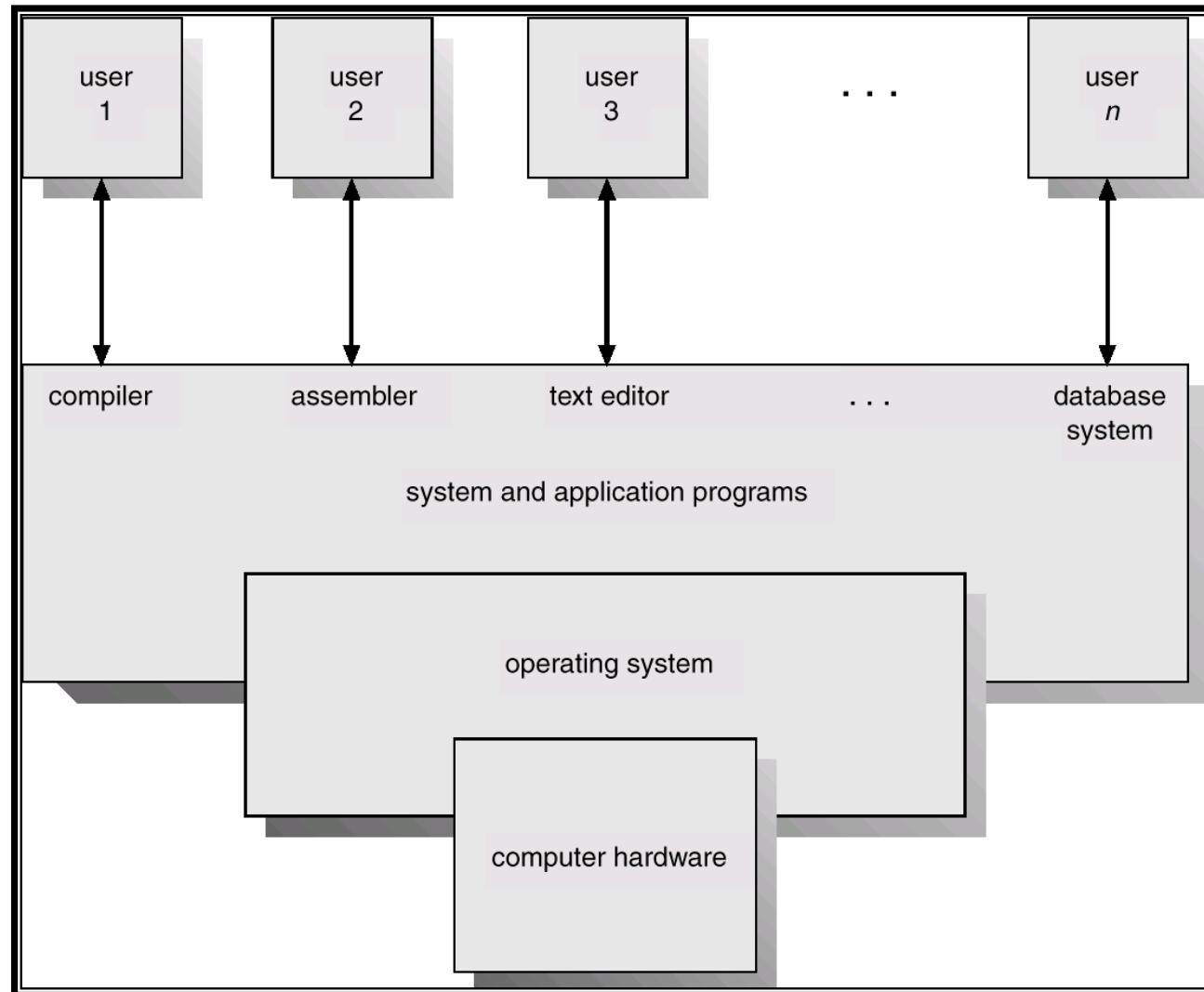
- User-centric definition
 - A program that acts as an intermediary between a user of a computer and the computer hardware
 - Defines an interface for the user to use services provided by the system
 - Provides a “view” of the system to the user
- System-centric definition
 - Resource allocator – manages and allocates resources
 - Control program – controls the execution of user programs and operations of I/O devices

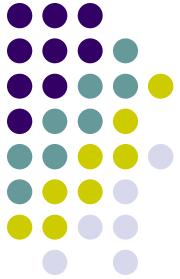


Computer System Components

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. Application programs – defines the ways in which the system resources are used to solve the computing problems of the users (compilers, databases, games, ...).
4. Users (people, machines, other computers).

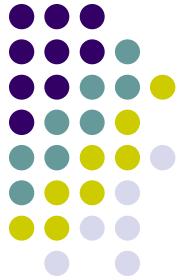
Abstract View of System Components



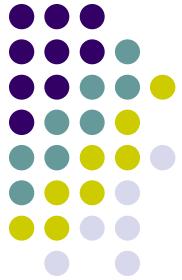


Types of Systems

- Batch Systems
 - Multiple jobs, but only one job in memory at one time and executed (till completion) before the next one starts
- Multiprogrammed Batch Systems
 - Multiple jobs in memory, CPU is multiplexed between them
 - CPU-bound vs I/O bound jobs
- Time-sharing Systems
 - Multiple jobs in memory and on disk, CPU is multiplexed among jobs in memory, jobs swapped between disk and memory
 - Allows interaction with users



- Personal Computers
 - Dedicated to a single user at one time
- Multiprocessing Systems
 - More than one CPU in a single machine to allocate jobs to
 - Symmetric Multiprocessing, NUMA machines ...
 - Multicore
- Other Parallel Systems, Distributed Systems, Clusters...
 - Different types of systems with multiple CPUs/Machines
- Real Time Systems
 - Systems to run jobs with time guarantees
- Other types possible depending on resources in the machine, types of jobs to be run...



- OS design depends on the type of system it is designed for
- Our primary focus in this course:
 - Uniprocessor, time-sharing systems running general purpose jobs from users
 - Effect of multicore/multiprocessors
- Will discuss some other topics at end



Resources Managed by OS

- Physical
 - CPU, Memory, Disk, I/O Devices like keyboard, monitor, printer
- Logical
 - Process, File, ...



Main functions of an OS

- Resource-Centric View
 - Process Management
 - Main Memory Management
 - File Management
 - I/O System Management
 - Secondary Storage Management
 - Security and Protection System
- User-centric view
 - System Calls
 - Command Interpreter (not strictly a part of an OS)



Process Management

• A process is a program in execution.

- Needs certain resources to accomplish its task
 - CPU time, memory, files, I/O devices...
- OS responsibilities
 - Process creation and deletion.
 - Process suspension and resumption.
 - Provide mechanisms for:
 - process synchronization
 - interprocess communication



Main-Memory Management

- OS responsibilities
 - Keeps track of which parts of memory are currently being used and by whom
 - Decides which processes to load when memory space becomes available
 - Allocates and deallocates memory space as needed



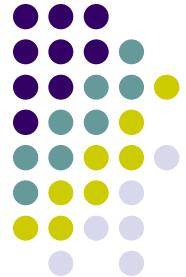
File Management

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



I/O System Management

- The I/O system consists of:
 - A buffer-caching system
 - Device driver interface
 - Drivers for specific hardware devices



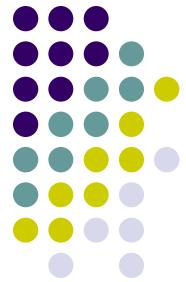
Secondary-Storage Management

- Most modern computer systems use disks as the principal on-line storage medium, for both programs and data.
- OS responsibilities
 - Free space management
 - Storage allocation
 - Disk scheduling



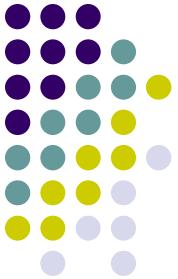
Security and Protection System

- *Protection* refers to a mechanism for controlling access by programs, processes, or users to both system and user resources.
- The protection mechanism must:
 - distinguish between authorized and unauthorized usage
 - specify the controls to be imposed
 - provide a means of enforcement



System Calls

- System calls provide the interface between a running program and the OS
 - Think of it as a set of functions available to the program to call (but somewhat different from normal functions, we will see why)
 - Generally available as assembly-language instructions.
 - Most common languages (e.g., C, C++) have APIs that call system calls underneath
- Passing parameters to system calls
 - Pass parameters in *registers*
 - Store the parameters in a table in memory, and the table address is passed as a parameter in a register
 - *Push* (store) the parameters onto the *stack* by the program, and *pop* off the stack by operating system



Command-Interpreter System

- Strictly not a part of OS, but always there
 - the *shell*
- Allows user to give commands to OS, interprets the commands and executes them
 - Calls appropriate functions/system calls

Dual Mode Operation

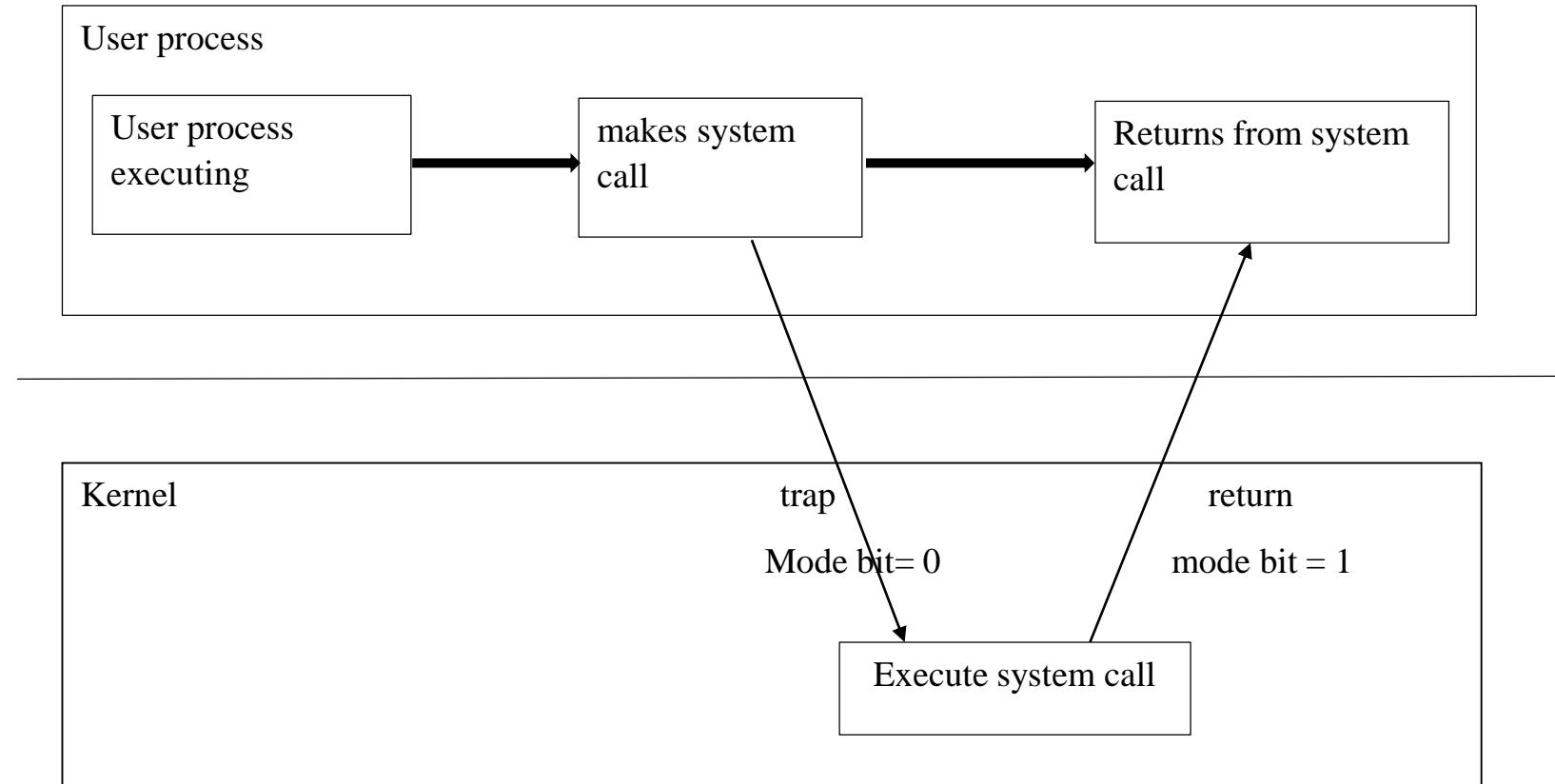
In order to ensure the proper execution of the OS, we must be able to distinguish between the execution of the OS code and user defined code. The approach taken by most computer systems is to provide hardware support that allows us to differentiate among modes of execution.

At least there should be two modes of operation:

- User mode
- Kernel mode (also called supervisor mode / system mode/ privileged mode)

A bit, called **mode bit**, is added to the hardware of the computer to indicate the current mode: **kernel (0) or user (1)**. With the mode bit, we are able to distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (**via a system call**), it must transit from user to kernel mode to fulfill the request.

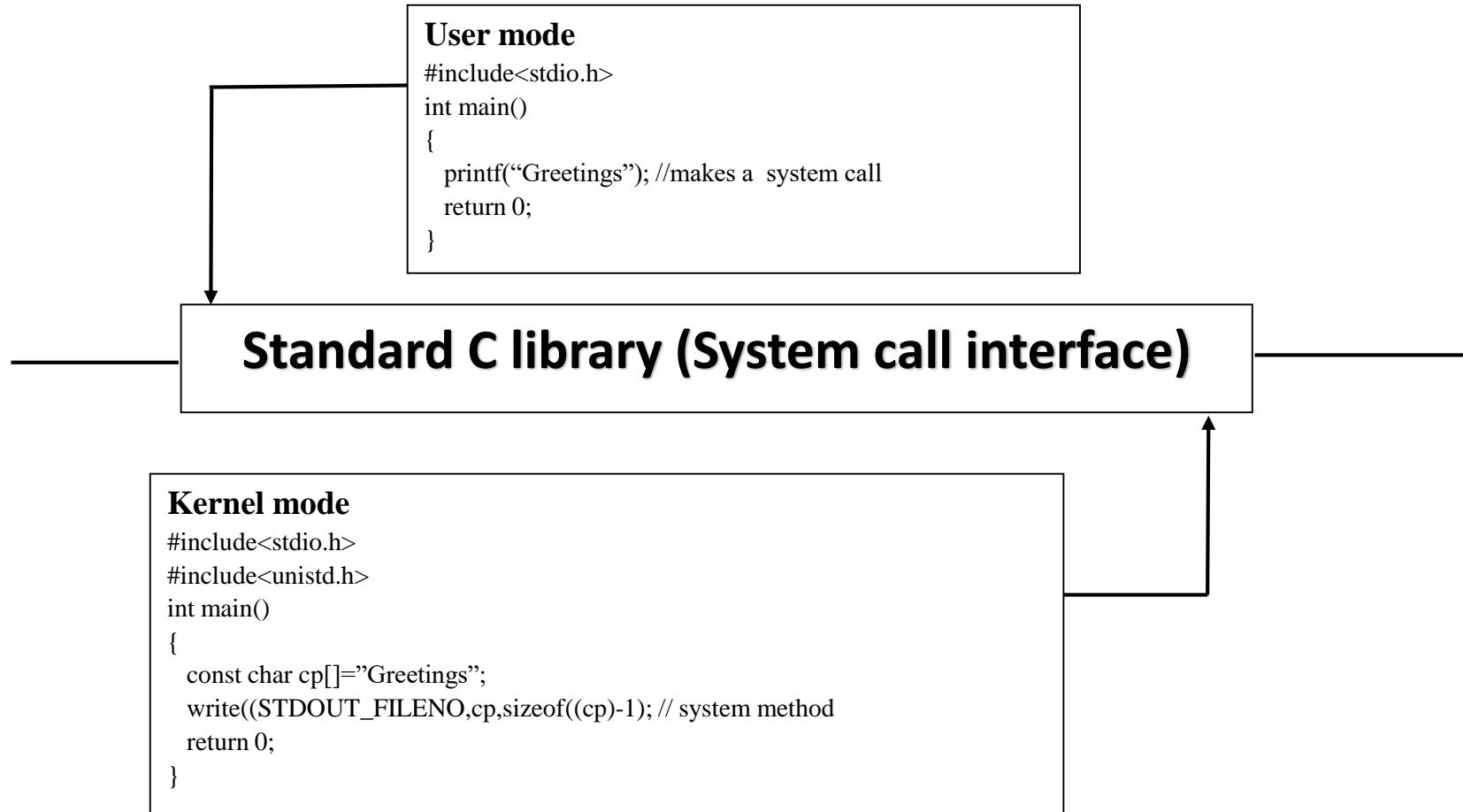
Transition from user to kernel mode



System Calls

- System calls provide the interface between a running program and the OS
 - * think of it as a set of functions available to the program to call (but somewhat different from normal functions, we will see why)
 - * Generally available as assembly-language instructions.
 - * Most common languages (e.g., C, C++) have APIs that call system calls underneath
- Passing parameters to system calls
 - * Pass parameters in *registers*
 - * Store the parameters in a table in memory, and the table address is passed as a parameter in a register
 - * Push (store) the parameters onto the *stack* by the program, and *pop* off the stack by operating system

Example



Example contd...

```
int main()
{
    int x;
    const char cp[]="Enter a value for x\n";
    write(STDOUT_FILENO,cp,sizeof(cp)-1);
    read(STDIN_FILENO,&x,sizeof(x));
    write(STDOUT_FILENO,&x,sizeof(x));
    return 0;
}
```

Types of System Calls

- Process Control
- File Manipulation
- Device Manipulation
- Information Maintenance
- Communications
- Protection

Examples of Windows System Calls:

CreateProcess()
ExitProcess()
WaitForSingleObject()

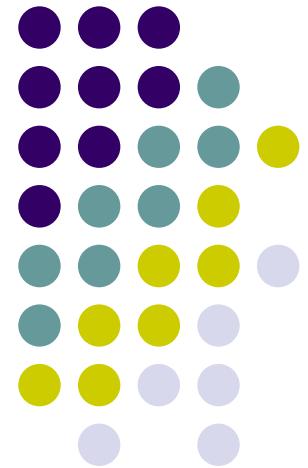
Exmaples of UNIX System Calls:

fork()
exit()
wait()

APIs for System Calls

- The API specify a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.
- Three of the most common APIs available to application programmers are:
 - * Win32 API for Windows systems
 - * POSIX APIs for POSIX based systems like UNIX, Linux and Mac OS
 - * Java APIs for Java virtual machines

Process Management





What is a Process?

- **Process** – an *instance* of a program in execution
 - Multiple instances of the same program are different processes
- A process has resources allocated to it by the OS during its execution
 - CPU time
 - Memory space for code, data, stack
 - Open files
 - Signals
 - Data structures to maintain different information about the process
 - ...
- **Each process is identified by a unique, positive integer id (process id)**



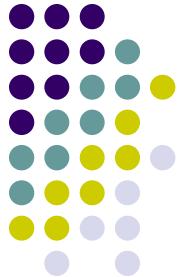
Process Control Block (PCB)

- The primary data structure maintained by the OS that contains information about a process
- One PCB per process
- OS maintains a list of PCB's for all processes



Typical Contents of PCB

- Process id, parent process id
- Process state
- CPU state: CPU register contents, PSW
- Priority and other scheduling info
- Pointers to different memory areas
- Open file information
- Signals and signal handler info
- Various accounting info like CPU time used etc.
- Many other OS-specific fields can be there
 - Linux PCB (*task_struct*) has 100+ fields

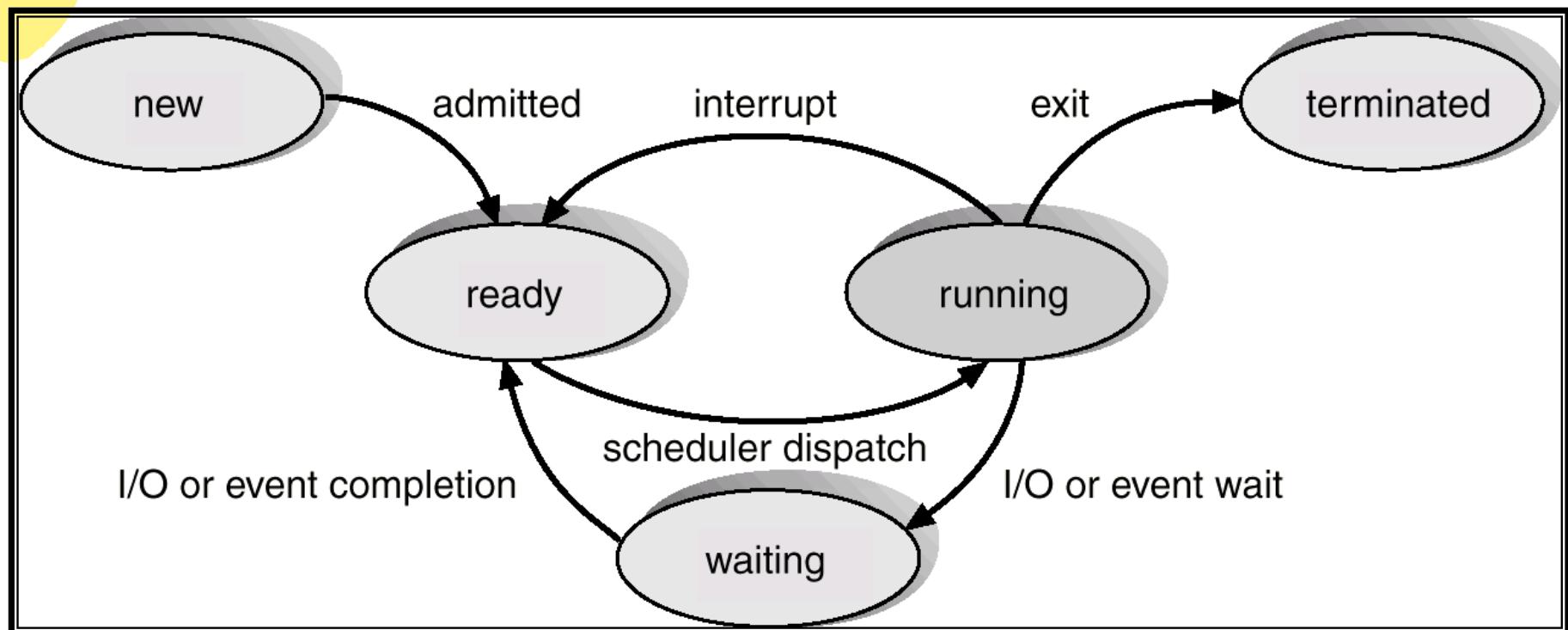


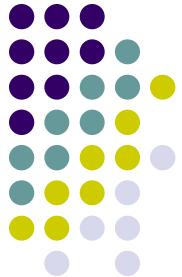
Process States (5-state model)

- As a process executes, it changes *state*
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event (needed for its progress) to occur
 - **ready**: The process is waiting to be assigned to a CPU
 - **terminated**: The process has finished execution



Process State Transitions





Main Operations on a Process

- **Process creation**
 - Data structures like PCB set up and initialized
 - Initial resources allocated and initialized if needed
 - Process added to ready queue (queue of processes ready to run)
- **Process scheduling**
 - CPU is allotted to the process, process runs
- **Process termination**
 - Process is removed
 - Resources are reclaimed
 - Some data may be passed to parent process (ex. exit status)
 - Parent process may be informed (ex. SIGCHLD signal in UNIX)



Process Creation

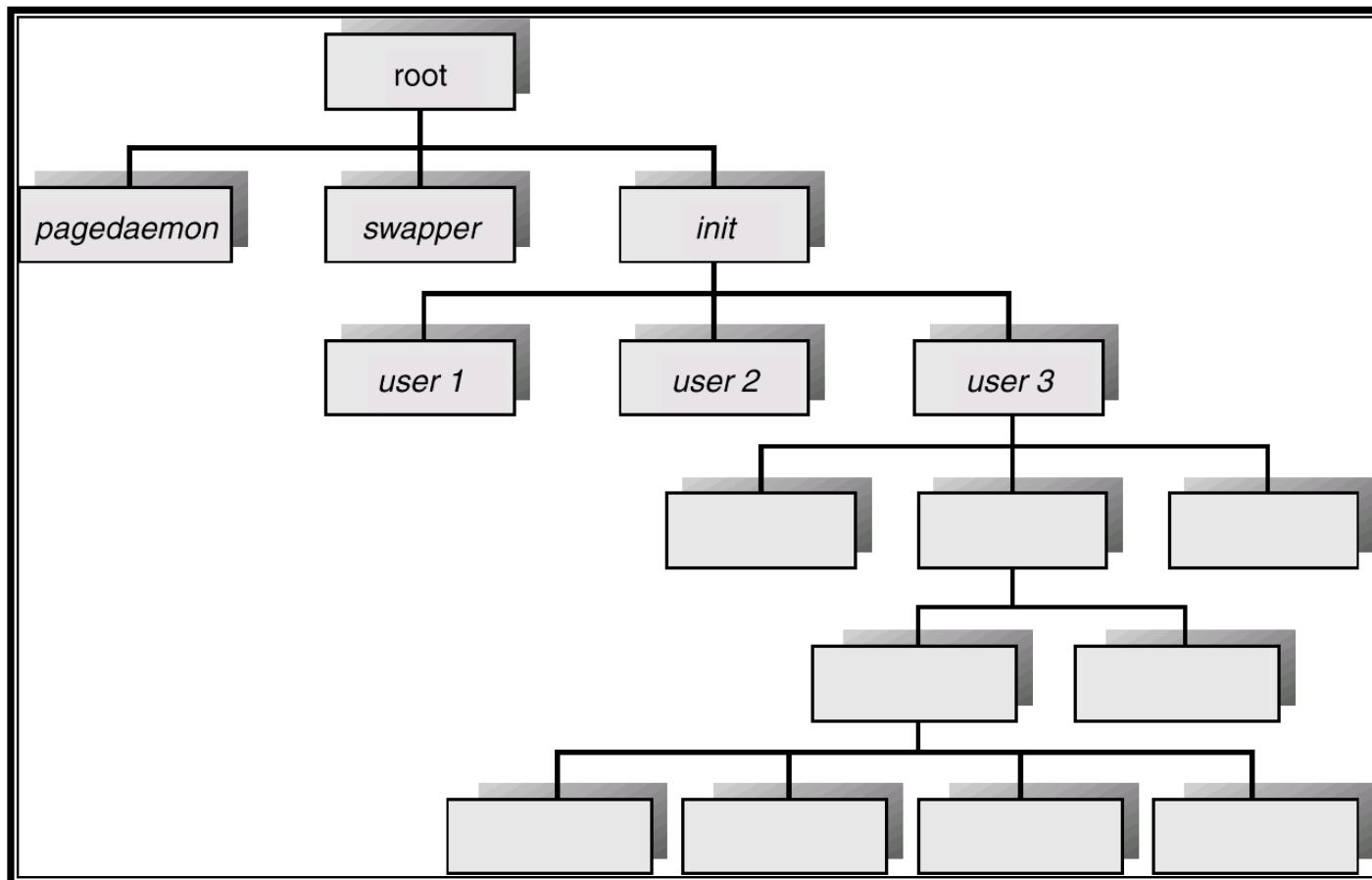
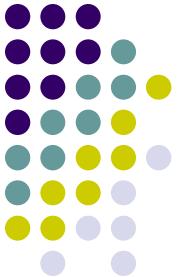
- A process can create another process
 - 💡 By making a system call (a function to invoke the service of the OS, ex. `fork()`)
 - 💡 **Parent** process: the process that invokes the call
 - 💡 **Child** process: the new process created
- The new process can in turn create other processes, forming a tree of processes
- The first process in the system is handcrafted
 - No system call, because the OS is still not running fully (not open for service)

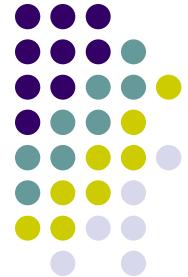


Process Creation (contd.)

- Resource sharing possibilities
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution possibilities
 - Parent and children execute concurrently
 - Parent waits until children terminate
- Memory address space possibilities
 - Address space of child duplicate of parent
 - Child has a new program loaded into it

Processes Tree on a UNIX System





Process Termination

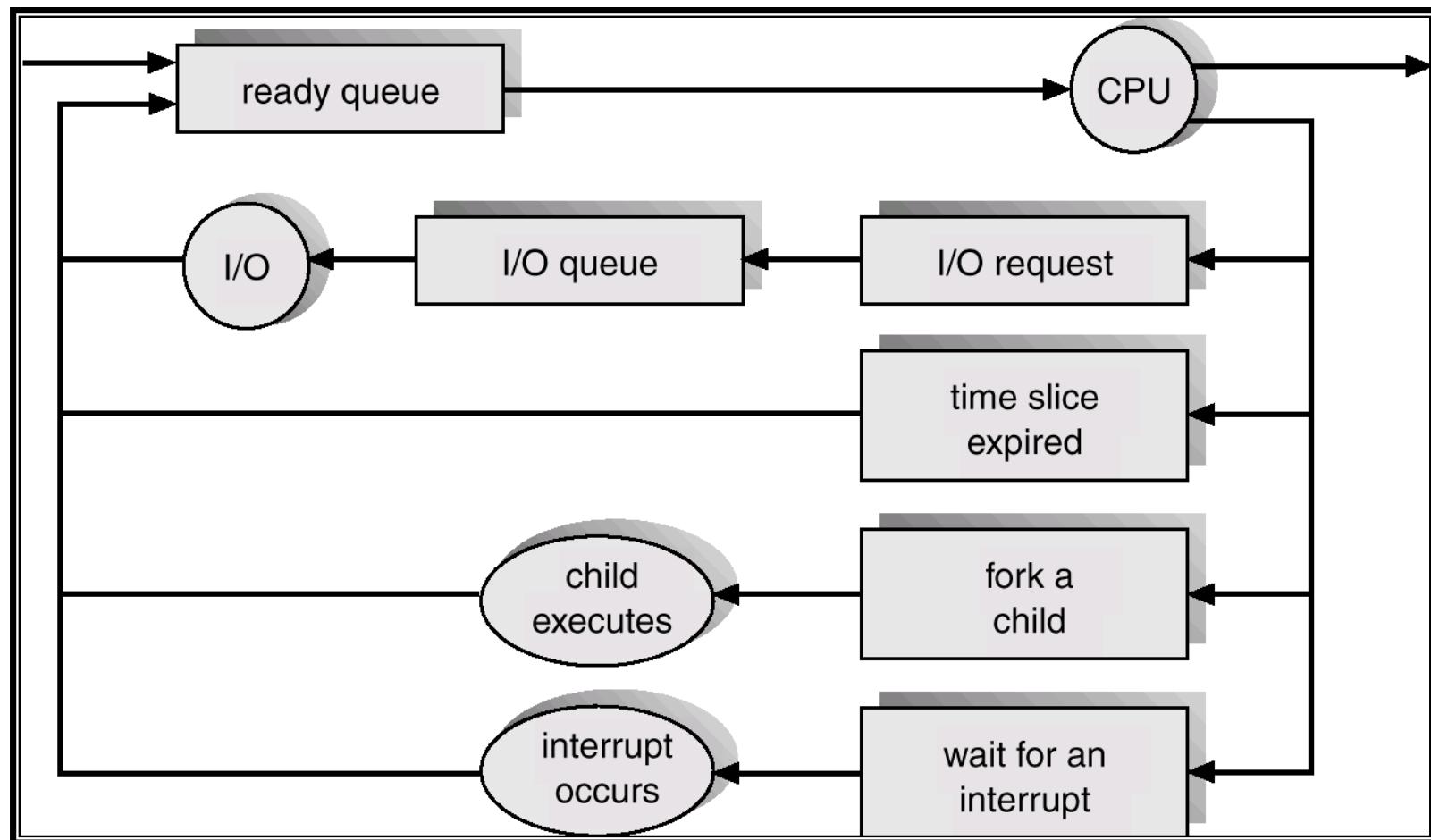
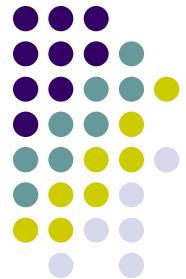
- Process executes last statement and asks the operating system to terminate it (ex. *exit/abort*)
- Process encounters a fatal error
 - Can be for many reasons like arithmetic exception etc.
- Parent may terminate execution of children processes (ex. *kill*). Some possible reasons
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
- Parent is exiting
 - Some operating systems may not allow child to continue if its parent terminates



Process Scheduling

- **Ready queue** – queue of all processes residing in main memory, ready and waiting to execute (links to PCBs)
- **Scheduler/Dispatcher** – picks up a process from ready queue according to some algorithm (**CPU Scheduling Policy**) and assigns it the CPU
- Selected process runs till
 - It needs to wait for some event to occur (ex. a disk read)
 - The CPU scheduling policy dictates that it be stopped
 - CPU time allotted to it expires (**timesharing systems**)
 - Arrival of a higher priority process
 - When it is ready to run again, it goes back to the ready queue
- Scheduler is invoked again to select the next process from the ready queue

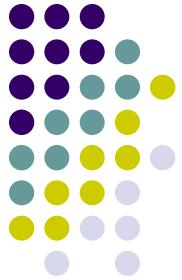
Representation of Process Scheduling





Schedulers

- Long-term scheduler (or job scheduler)
 - Selects which processes should be brought into the ready queue
 - Controls the *degree of multiprogramming* (no. of jobs in memory)
 - Invoked infrequently (seconds, minutes)
 - May not be present in an OS (ex. linux/windows does not have one)
- Short-term scheduler (or CPU scheduler)
 - Selects which process should be executed next and allocates CPU
 - Invoked very frequently (milliseconds), must be fast



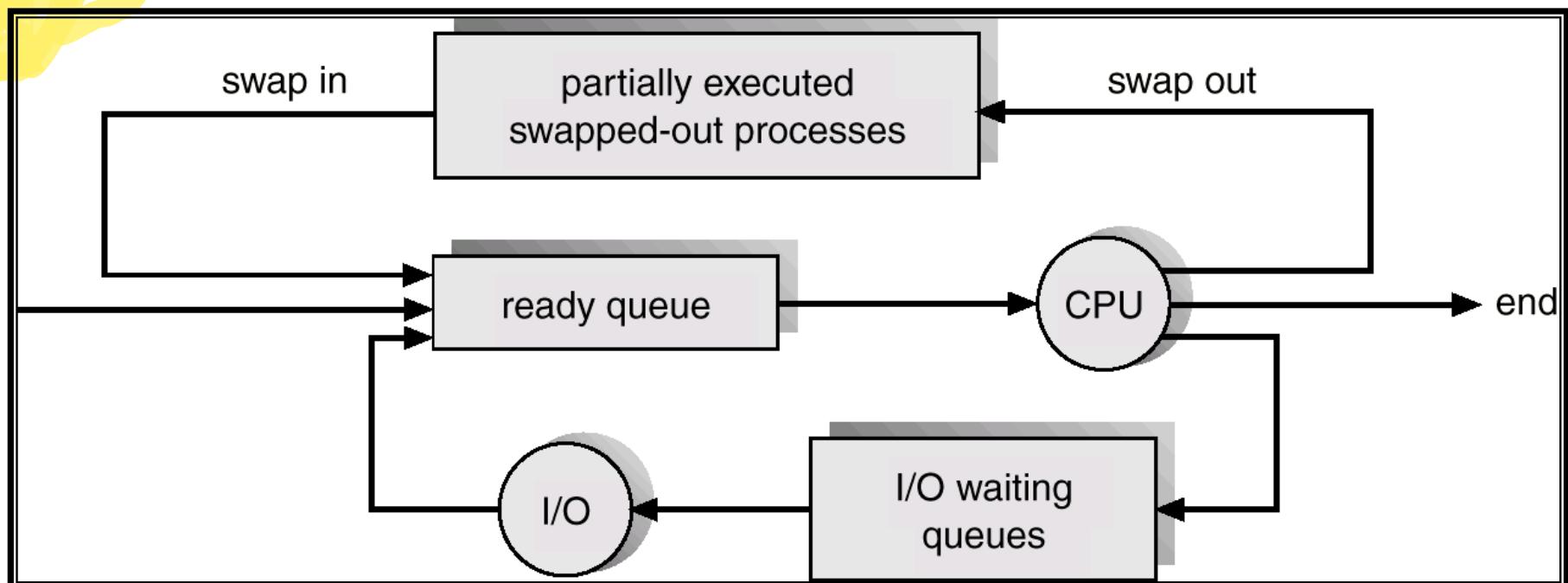
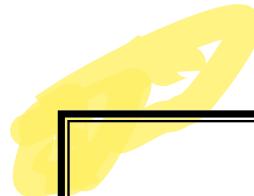
What if all processes do not fit in memory?

- Partially executed jobs in secondary memory (**swapped out**)
 - Copy the process image to some pre-designated area in the disk (swap out)
 - Bring in again later and add to ready queue later





Addition of Medium Term Scheduling





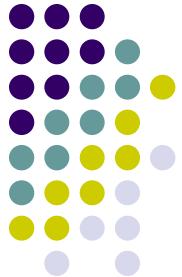
Other Questions

- How does the scheduler get scheduled? (Suppose we have only one CPU)
 - As part of execution of an ISR (ex. timer interrupt in a time-sharing system)
 - Called directly by an I/O routine/event handler after blocking the process making the I/O or event request
- What does it do with the running process?
 - Save its **context**
- How does it start the new process?
 - Load the saved context of the new process chosen to be run
 - Start the new process



Context of a Process

- Information that is required to be saved to be able to restart the process later from the same point
- Includes:
 - CPU state – all register contents, PSW
 - Program counter
 - Memory state – code, data
 - Stack
 - Open file information
 - Pending I/O and other event information



Context Switch



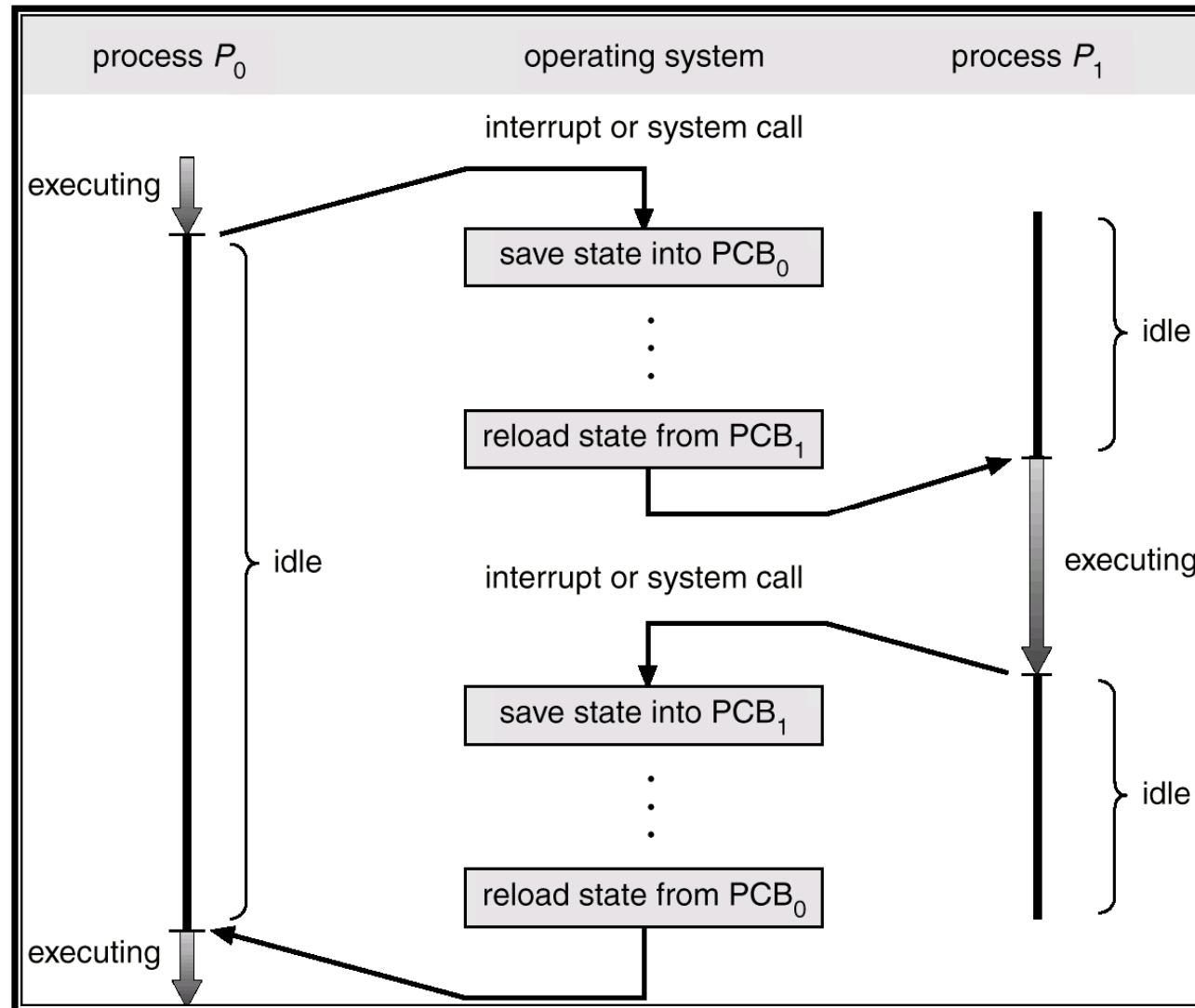
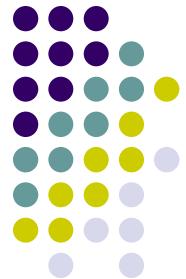
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support



Handling Interrupts

- H/w saves PC, PSW
- Jump to ISR
- ISR should first save the context of the process
- Execute the ISR
- Before leaving, ISR should restore the context of the process being executed
- Return from ISR restores the PC
- ISR may invoke the dispatcher, which may load the context of a new process, which runs when the interrupt returns instead of the original process interrupted

CPU Switch From Process to Process



Example: Timesharing Systems



- Each process has a time quantum T allotted to it
- Dispatcher starts process P_0 , loads an external counter (timer) with counts to count down from T to 0
- When the timer expires, the CPU is interrupted
- The ISR invokes the dispatcher
- The dispatcher saves the context of P_0
 - PCB of P_0 tells where to save
- The dispatcher selects P_1 from ready queue
 - The PCB of P_1 tells where the old state, if any, is saved
- The dispatcher loads the context of P_1
- The dispatcher reloads the counter (timer) with T
- The ISR returns, restarting P_1 (since P_1 's PC is now loaded as part of the new context loaded)
- P_1 starts running

Creating a child process

Parent waits for the child to terminate

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
    pid_t pid;
    printf("Main Program\n");
    pid=fork();
    if(pid<0)
    { printf("Error\n"); return 1; }
    else if(pid==0)
    { printf("This is child process and id=%d and my parent id=%d\n",getpid(),getppid()); }
    else
    {
        printf("I am parent and my id is=%d\n",getpid());
        wait(NULL);
    }
    return 0;
}
```

output

Main Program

I am parent and my id is=6910

This is child process and id=6911 and my parent id=6910

Creating a child process

Parent does not wait for the child to terminate

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
    pid_t pid;
    printf("Main Program\n");
    pid=fork();
    if(pid<0)
    { printf("Error\n"); return 1; }
    else if(pid==0)
    { printf("This is child process and id=%d and my parent id=%d\n",getpid(),getppid()); }
    else
    {
        printf("I am parent and my id is=%d \n",getpid());
        // wait(NULL);
    }
    return 0;
}
```

output

Main Program

I am parent and my id is=5515

This is child process and id=5516 and my parent id=1

Address space of child process same as that of parent

```
fork2.c
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
    pid_t pid;
    printf("Main Program\n");
    pid=fork();
    if(pid<0)
    {
        printf("Error\n");
        return 1;
    }
    else if(pid==0)
    {
        printf("This is child process\n");
        //execvp("/home/user3/myfile","myfile",NULL);
    }
    else
    {
        printf("The parent is waiting\n");
        wait(NULL);
        printf("Child has finished\n");
    }
    printf("Main program terminates\n");
    return 0;
}
```

output
Main Program
The parent is waiting
This is child process
Main program terminates
Child has finished
Main program terminates

Address space of child process is different from parent process

```
fork1.c
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
    pid_t pid;
    printf("Main Program\n");
    pid=fork();
    if(pid<0)
    {
        printf("Error\n");
        return 1;
    }
    else if(pid==0)
    {
        printf("This is child process\n");
        execp("/home/user22/myfile","myfile",NULL);
    }
    else
    {
        printf("The parent is waiting\n");
        wait(NULL);
        printf("Child has finished\n");
    }
    printf("Main program terminates\n");
    return 0;
}
```

```
myfile.c
#include<stdio.h>
int main()
{
    printf("hello\n");
    return 0;
}
```

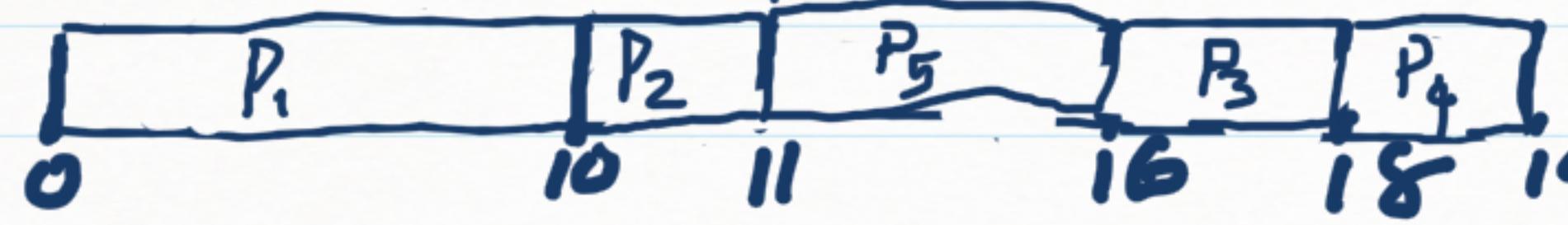
output
Main Program
The parent is waiting
This is child process
hello
Child has finished
Main program terminates

output
hello

Priority Based Scheduling Example

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>	<u>Arrival Time</u>
P ₁	10	3	0
P ₂	1	1	1
P ₃	2	4	2
P ₄	1	5	3
P ₅	5	2	4

Non preemptive



$$\begin{aligned}P_1 &= 0-0 = 0 \\P_2 &= 10-1 = 9 \\P_3 &= 16-2 = 14 \\P_4 &= 18-3 = 15 \\P_5 &= 11-4 = 7\end{aligned}$$

$$AT = 45/5 = 9$$

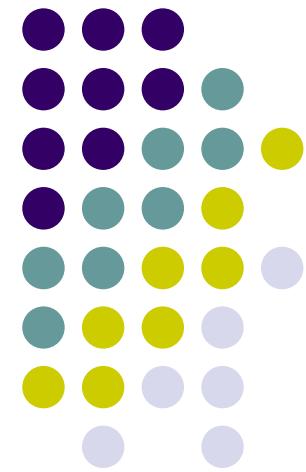
Preemptive



$$\begin{aligned}P_1 &= (0-0) + (2-1) + (9-4) = 6 \\P_2 &= 1-1 = 0 \\P_3 &= 16-2 = 14 \\P_4 &= 18-3 = 15 \\P_5 &= 4-4 = 0\end{aligned}$$

$$AWT = 35/5 = 7$$

CPU Scheduling





Types of jobs

- CPU-bound vs. I/O-bound
 - Maximum CPU utilization obtained with multiprogramming
- Batch, Interactive, real time
 - Different goals, affects scheduling policies



CPU Scheduler



- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
 - Switches from running to waiting state
 - Switches from running to ready state
 - Switches from waiting to ready
 - Terminates
- Scheduling under 1 and 4 is *nonpreemptive*.
- All other scheduling is *preemptive*.



Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU scheduler; this involves:
 - switching context
 - switching to user mode
 - jumping to the proper location in the user program to restart that program
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running.



Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)



Optimization Criteria

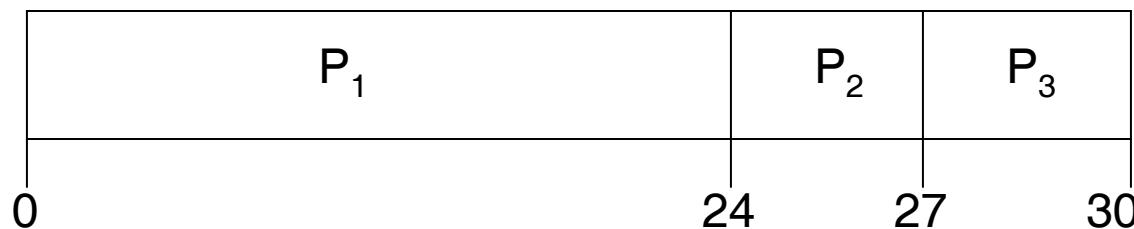
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First-Come, First-Served (FCFS) Scheduling



<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3 . The Gantt Chart for the schedule is:



- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: $(0 + 24 + 27)/3 = 17$

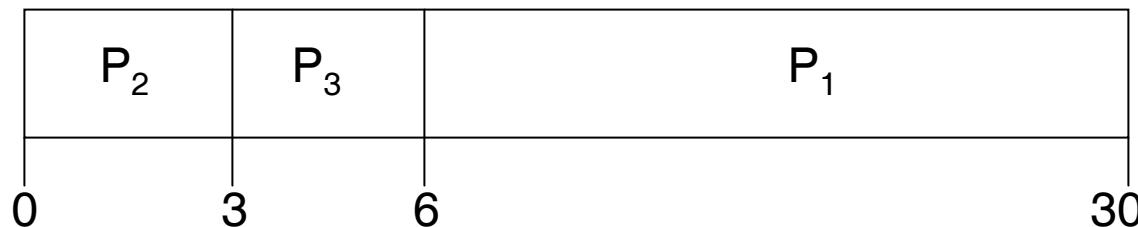


FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



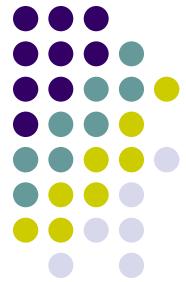
- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case.
- *Convoy effect*: short process behind long process

Shortest-Job-First (SJF) Scheduling



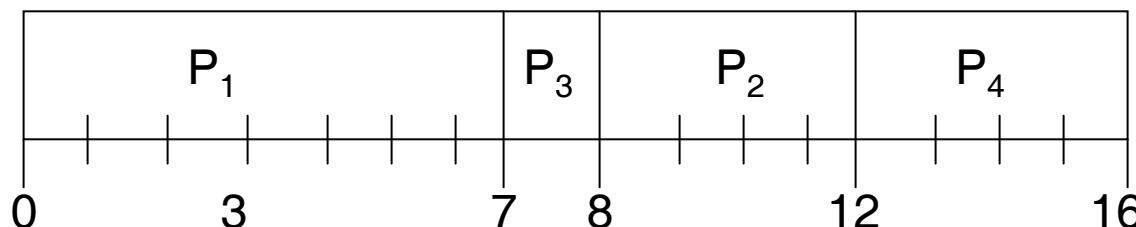
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
 - nonpreemptive – once CPU given to the process it cannot be preempted until it completes its CPU burst
 - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the **Shortest-Remaining-Time-First (SRTF)**
- **SJF is optimal** – gives minimum average waiting time for a given set of processes

Example of Non-Preemptive SJF



<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)



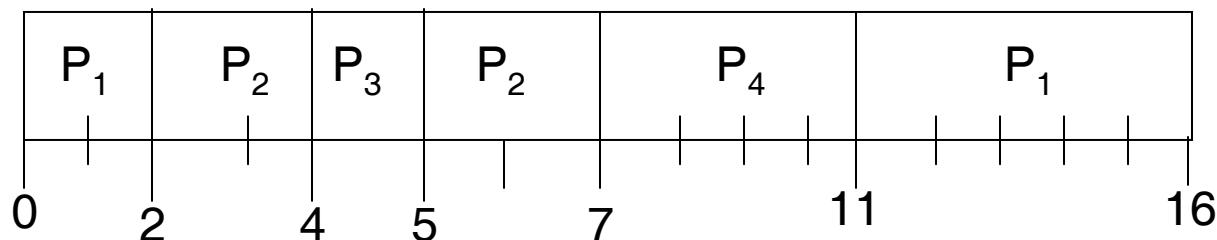
- Average waiting time = $(0 + 6 + 3 + 7)/4 = 4$



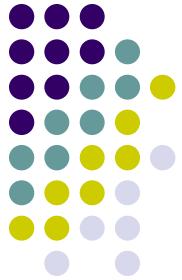
Example of Preemptive SJF

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



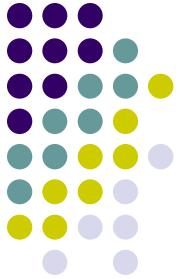
- Average waiting time = $(9 + 1 + 0 + 2)/4 = 3$



Determining Length of Next CPU Burst

- Can only estimate the length.
- Can be done by using the length of previous CPU bursts, using exponential averaging
 - $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n, \quad 0 \leq \alpha \leq 1$
 - t_n = actual length of n^{th} CPU burst
 - τ_n = predicted length of nth CPU burst

Properties of Exponential Averaging



- $\alpha = 0$
 - $\tau_{n+1} = \tau_n$
 - Recent history does not count
- $\alpha = 1$
 - $\tau_{n+1} = t_n$
 - Only the actual last CPU burst counts
- If we expand the formula, each successive term has less weight than its predecessor
 - Recent history has more weight than old history



Priority Scheduling

- A **priority** number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer \equiv highest priority).
 - Preemptive
 - Nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem \equiv **Starvation** – low priority processes may never execute
- Solution \equiv **Aging** – as time progresses increase the priority of the process



Round Robin (RR)

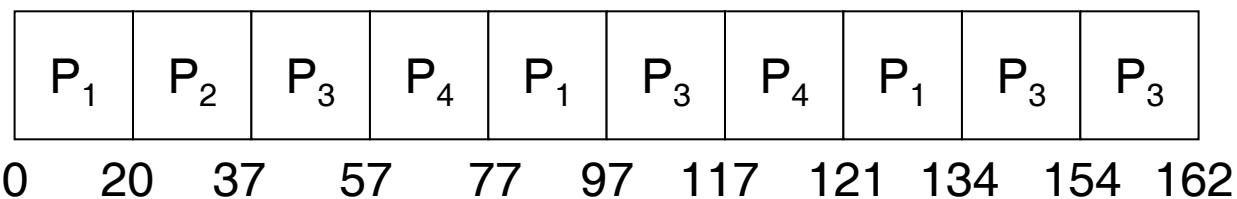
- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue
- If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units
- Performance
 - q large \Rightarrow FIFO
 - q small \Rightarrow q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 20



<u>Process</u>	<u>Burst Time</u>
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

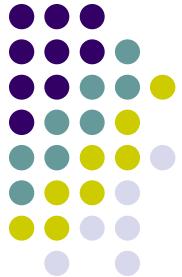


- Typically, higher average turnaround than SJF, but better *response*.



Multilevel Queue

- Ready queue is partitioned into separate queues: foreground (interactive) and background (batch)
- Each queue has its own scheduling algorithm,
 - foreground – RR
 - background – FCFS
- Scheduling must be done between the queues
 - Fixed priority scheduling (i.e., serve all from foreground then from background). Possibility of starvation.
 - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
 - 20% to background in FCFS



Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way.
- Multilevel-feedback-queue scheduler defined by the following parameters:
 - number of queues
 - scheduling algorithms for each queue
 - method used to determine when to upgrade a process
 - method used to determine when to demote a process
 - method used to determine which queue a process will enter when that process needs service

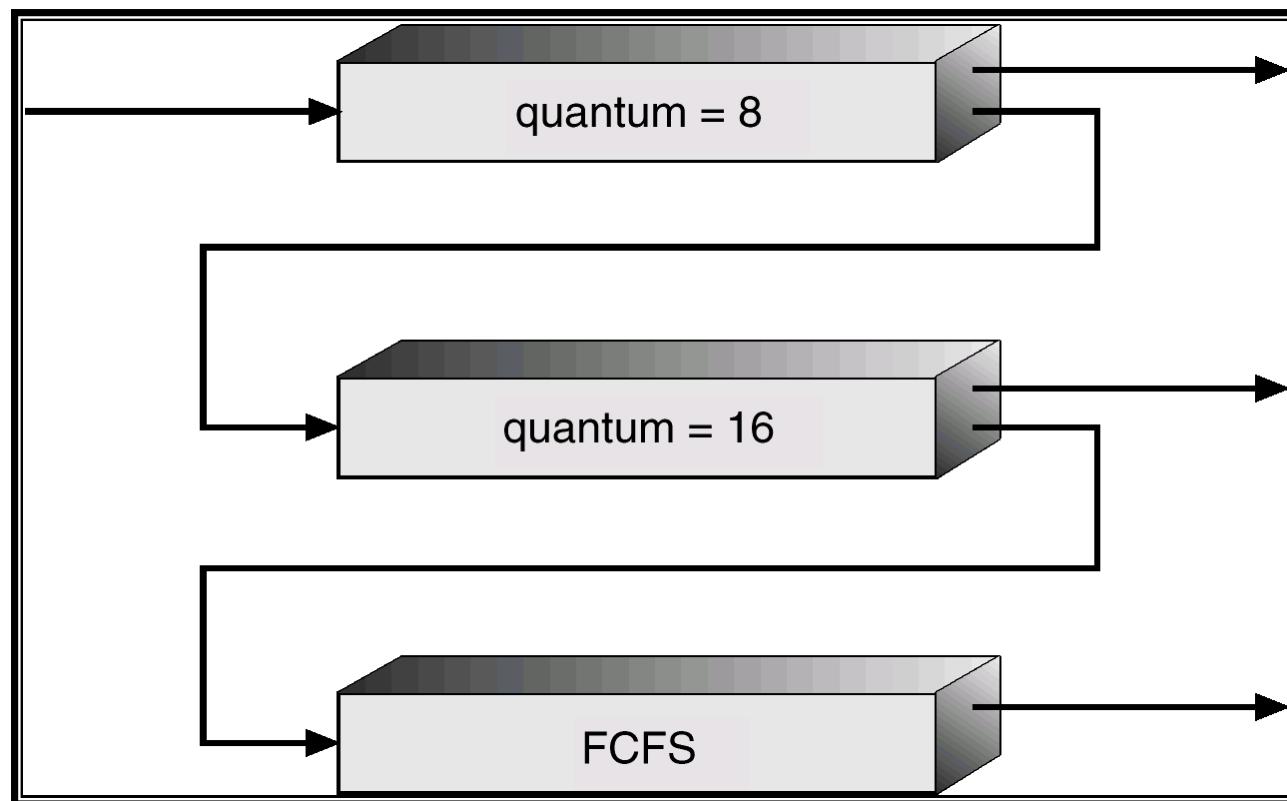
Example of Multilevel Feedback Queue



- Three queues:
 - Q_0 – time quantum 8 milliseconds
 - Q_1 – time quantum 16 milliseconds
 - Q_2 – FCFS
- Scheduling
 - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1 .
 - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2 .



Multilevel Feedback Queues



Inter Process Communication

- Types of Processes
- Why IPC?
- Models of IPC
- Shared Memory Systems
- Message Passing Systems
- Shared Memory Vs. Message Passing



Types of Processes

- Independent **Process**: A process is independent if it cannot affect or be affected by the other processes. Any process that does not share data with any other process is independent.
- Cooperating Process: A process is cooperating if it can **affect** or be affected by other processes executing in the system. Any process that shares data with other processes is a cooperating process.

Why IPC?

There are several reasons for providing an environment that allows cooperation:

- **Information Sharing:** Several users may be interested for same piece of information
- **Computation Speedup:** For faster execution a task may be broken into subtasks and run in parallel
- **Modularity:** A system may be constructed in modular fashion by dividing the system functions into separate processes
- **Convenience:** An individual user may work on many tasks at the same time

Types of IPC

Cooperating processes require that an Inter Process Communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental Models of inter process communication:

- Shared Memory

- Message Passing

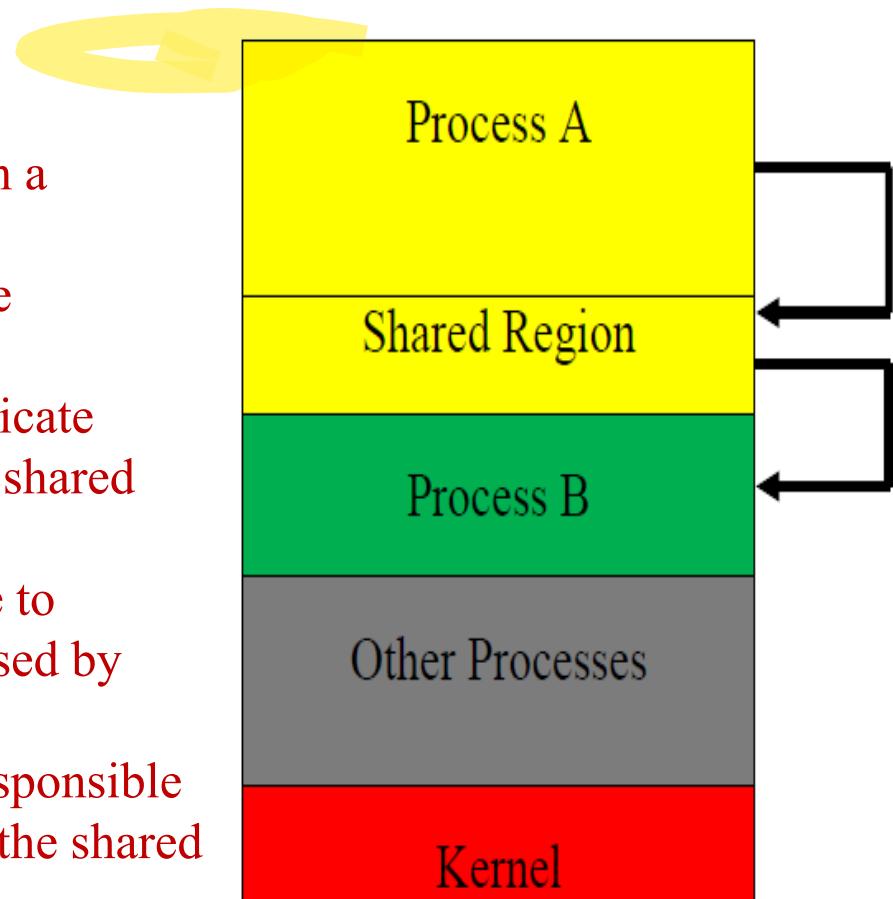
Shared Memory Systems

- Requirements
- Example
- Types of Buffers
- Programming Example

Shared Memory Systems

Requirements:

- Communicating processes to establish a region of shared memory
- Typically shared region belongs to the address space of the creating process
- Other processes that wish to communicate must attach their address space to the shared region
- Communicating processes must agree to remove the security restrictions imposed by the OS
- Communicating processes are also responsible for ensuring writing serializability in the shared region



Shared Memory Systems

Example: Producer-Consumer Problem

- 💡 A producer process produces information that is consumed by a consumer process. A compiler may produce assembly code which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.
- 💡 To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- 💡 The producer and consumer must be synchronized , so that the consumer does not try to consume an item that has not yet been produced.

Shared Memory Systems

Types of Buffers

- **Unbounded Buffers:** The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
- **Bounded Buffers:** The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Shared Memory Systems

Programming Example

shm_server.c

- Writes the characters from a to z in the shared memory
- Waits for the client program to read the characters
- Terminates when client modifies the first character to *

shm_client.c

- Reads the characters from a to z in the shared memory
- Terminates after modifying the first character to * in the shared memory

Message Passing Systems

- Requirements
- Types
- Direct Communication
- Indirect Communication
- Synchronous/Asynchronous
- Buffering

Message Passing Systems

- **Message passing** provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and particularly useful in a distributed environment.
- Requirements: A message passing facility provides at least two operations
 - • Send(message)
 - • Receive(message)

Message Passing Systems

- >If processes P and Q want to communicate, they must send messages to and receive messages from each other. A communication link must exist between them. This link can be implemented in the following ways:
 - Direct or Indirect Communication
 - Synchronous or Asynchronous Communication
 - Automatic or Explicit Buffering

Message Passing Systems

Direct Communication:

Each process that wants to communicate must explicitly name the recipient or sender of the communication. The send() and receive() primitives are defined as:

- Send(P, message) – send a message to P.
- Receive(Q, message) – receive a message from Q

Message Passing Systems

Direct Communication

Properties of a communication link:

- A link is established between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate
- A link is associated with exactly two processes
- Between each pair of processes, there exists only one link

Message Passing Systems

Direct Communication

Disadvantages:

- Limited modularity of the resulting process definitions
- Changing the identifier of a process may necessitate examining all other process definitions
- All references to the old identifier must be found and modified to the new identifier

Message Passing Systems

Indirect Communication

The messages are sent to and received from **mailboxes**, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification.

The send() and receive() primitives are defined as:

- Send(A, message) – send a message to mailbox A.
- Receive(A, message) – receive a message from mailbox A.

Message Passing Systems

Indirect Communication

Properties of a communication link:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

Message Passing Systems

Indirect Communication

Ownership of the mailbox:

- A mailbox may be owned by a process
 - The owner can only receive messages through this mailbox
 - The user can only send messages to this mailbox
 - When the owner(process) terminates mailbox disappears
- A mailbox may be owned by the OS: The OS must provide a mechanism that allows a process to do the following:
 - Create a new mailbox
 - Send and receive messages through the mail box
 - Delete a mailbox

Message Passing Systems

Synchronous/Asynchronous Communication

Message passing may be either blocking or non blocking also known as synchronous and asynchronous.

- Blocking send: The sending process is blocked until the message is received by the receiving process or the mailbox.
- Non blocking send: The sending process sends the message and resumes operation.
- Blocking receive: The receiver blocks until a message is available.
- Non blocking receiver: The receiver retrieves either a valid message or a null.

Message Passing Systems

Buffering: Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- Zero capacity: The queue has a maximum length of zero; thus a link cannot have any messages waiting in it. The sender must block until the recipient receives the message.
- Bounded capacity: The queue has finite length n ; thus, at most n messages can reside in it. If the link is full, the sender must block until space is available in the queue.
- Unbounded capacity: The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

Shared Memory Vs. Message Passing



<p>Easier to implement in case of an intra-computer system as the main memory is shared</p>	<p>Easier to implement in case of distributed (inter computer) systems</p>
<p>Shared memory is faster as system calls are required only to establish shared memory region.</p>	<p>Message passing is slower as these are implemented using system calls which requires the more time consuming task of kernel intervention</p>

Threads

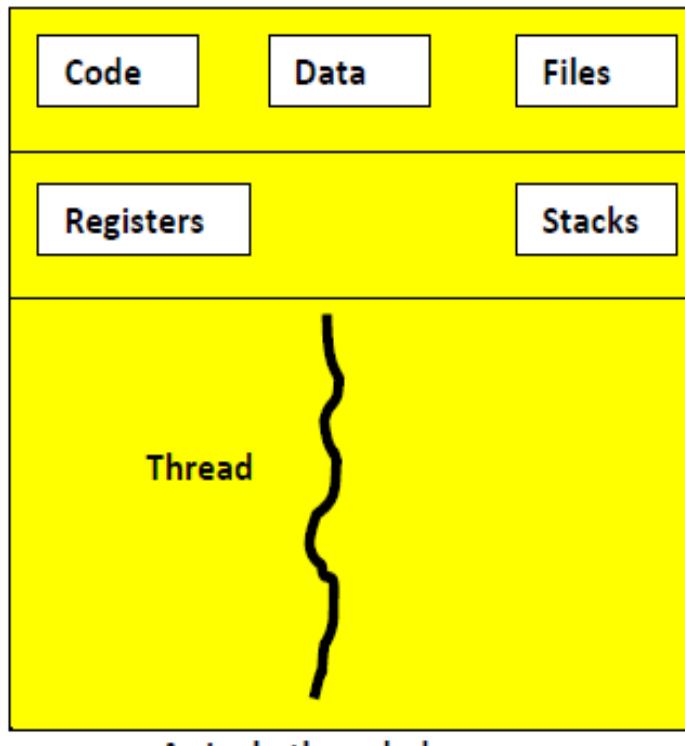
- Introduction
- Benefits
- Demerits
- Types of Threads
- Multithreading Models
- Programming Example

Introduction

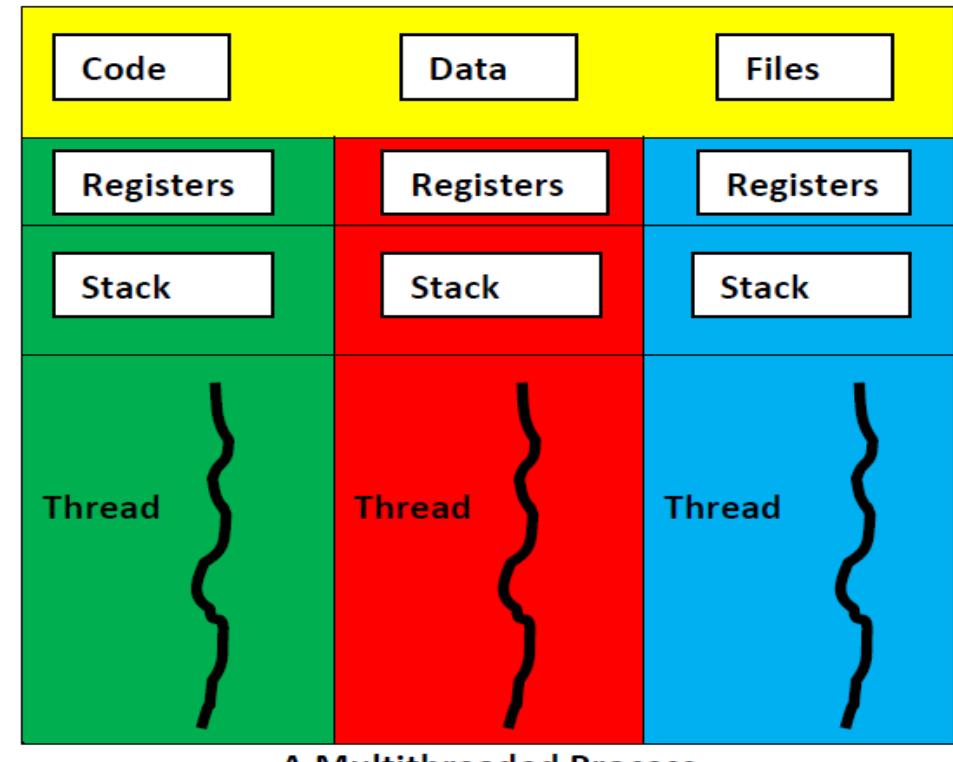
- A thread is a semi-process or a lightweight process, that has its own stack, and executes a given piece of code.
- Unlike a process, a thread normally shares its memory with other threads.
- A thread group is a set of threads all executing inside the same process sharing the same memory, global variables, heap and same set of file descriptors.
- A traditional(or heavyweight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

Example

- A web browser might have one thread for displaying images or text while another for retrieving data from the network.
- A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user.



1/25/2017



Muktikanta Sahu

3

Benefits

- **Responsiveness:** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation thereby increasing responsiveness to the user.
- **Resource Sharing:** Unlike processes, threads share memory and resources of the process to which they belong by default.
- **Economy:** As threads share the resources of the processes to which they belong it is more economical to create and context switch threads.
- **Scalability:** The benefits of multithreading can be greatly increased in a multiprocessor architecture where threads may be running in parallel on different processors.

Demerits

- Because threads in a group use the same memory space, if one of them corrupts the contents of its memory, other threads might suffer as well.
- Unlike processes, threads belonging to same process have to run on the same machine.

Types of Threads

- User Level Threads: User threads are supported above the kernel and managed without kernel support. All thread management is done by the application by using a thread library.
- Kernel Level Threads: Kernel threads are supported and managed directly by the operating system.

User Level Threads

- Kernel Activity for ULTs
 - The kernel is not aware of thread activity but it is still managing process activities.
 - When a thread makes a system call, the whole process blocks
 - Thread states are independent of process states
- Advantages
 - Thread switching does not involve the kernel. No mode switching
 - Scheduling can be application specific.
 - ULTs can run on any OS. Only need a thread library.
- Disadvantages
 - Most system calls are blocking and the kernel blocks processes.
 - The kernel can only assign processes to processors. Two threads within the same process cannot run simultaneously on two processors.

Kernel Level Threads

Advantages

- The kernel can simultaneously schedule many threads of the same process on many processors. Blocking is done on a thread level.
- Kernel routines can be multithreaded.

Disadvantages

- Thread switching within the same process involves the kernel. For example, if we have two mode switches per thread switch this results in a significant slow down.

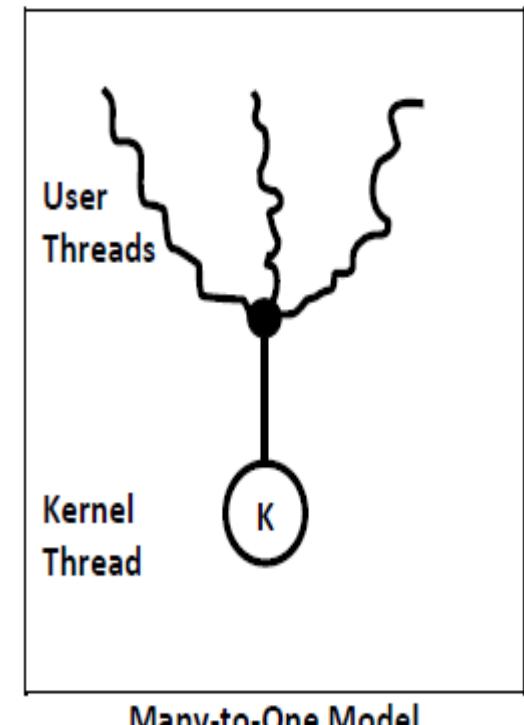
Multithreading Models

- Many-to-One Model
- One-to-One Model
- Many-to-Many Model

Multithreading Models

Many-to-One Model:

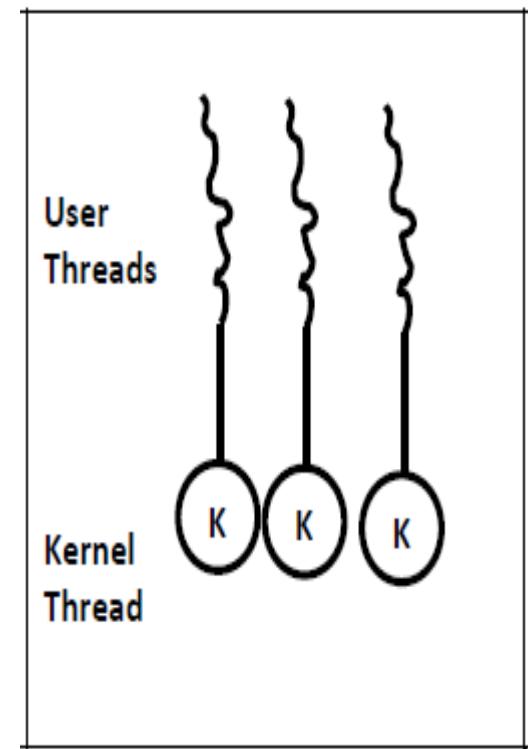
- This model maps many user level threads to one kernel thread.
- Thread management is done by the thread library in user space, so it is efficient.
- Entire process will block if a thread makes a blocking system call.
- Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.



Multithreading Models

One-to-One Model

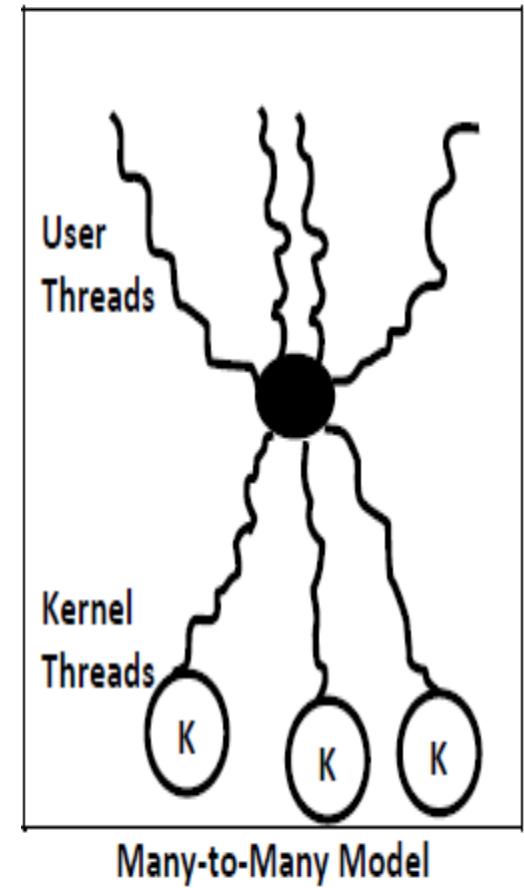
- This model maps each user thread to a kernel thread.
- Provides more concurrency than many to one model by allowing another thread to run when a thread makes a blocking system call.
- Allows multiple threads to run in parallel on multiprocessors.
- The only drawback is that creating user thread requires creating the corresponding kernel thread.



Multithreading Models

Many-to-Many Model

- This model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- This model overcomes the problems that occurred in many-to-one model and one-to-one model.
- Many user threads can be created and the corresponding kernel threads can run in parallel on a multiprocessor.
- When a thread performs a blocking system call, the kernel can schedule another thread for execution.



Thread Libraries

- POSIX thread for UNIX like systems
- Win32 API threads for Windows systems
- Java threads for Java virtual machines
- Multithreading through OpenMP API
- Multithreading through MPI API
- Multithreading through CUDA API

Programming Example

OpenMP threading

```
#include<iostream>
#include<cstdlib>
#include<omp.h>
using namespace std;
int main()
{
    int i=0, end=64;
    #pragma omp parallel for num_threads(4)
    for(i=0;i<end;i++)
    {
        int ID=omp_get_thread_num();
        #pragma omp critical
        {
            cout<<"I am thread: "<<ID<<"\t printing i="<<i<<endl;
        }
    }
    return 0;
}
```

Compilation command

```
g++ -fopenmp -o filename filename.cpp
```

Programming Example

POSIX threading

```
#include<stdio.h>
#include<pthread.h>
void* do_loop(void* data)
{int i; // counter to print data
 int j; //counter for delay
 int me=*((int*)data); // thread ID
 for(i=0;i<10;i++)
 {
    for(j=0;j<5000000;j++); // delay loop
    printf("%d Got %d\n",me,i);
 }
 pthread_exit(NULL); // terminate thread
}
int main()
{
    int thr_id; // thread ID for newly created thread
    pthread_t p_thread; // thread structure
    int a=1; // thread 1 ID
    int b=2; // thread 2 ID
    // create a new thread that will execute do loop
    thr_id(pthread_create(&p_thread, NULL, do_loop, (void*)&a));
    // run do_loop in main thread as well
    do_loop((void*)&b);
    return 0;
}
```

Compilation command

```
gcc -o filename filename.c -lpthread
```

Programming Example

MPI threads

mpi_thread.cpp

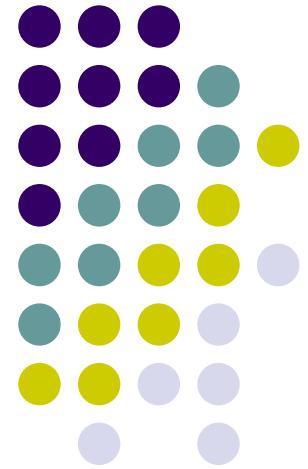
Compilation command

mpic++ –fopenmp –o filename filename.cpp

Execute command

mpirun –np 4 ./filename

Process Coordination





Why is it needed?

- Processes may need to share data
 - More than one process reading/writing the same data (a shared file, a database record,...)
 - Output of one process being used by another
 - Needs mechanisms to pass data between processes
- Ordering executions of multiple processes may be needed to ensure correctness
 - Process X should not do something before process Y does something etc.
 - Need mechanisms to pass control signals between processes

Interprocess Communication (IPC)

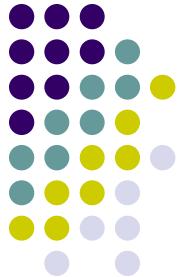


- Mechanism for processes P and Q to communicate and to synchronize their actions
 - Establish a communication link
- Fundamental types of communication links
 - Shared memory
 - P writes into a shared location, Q reads from it and vice-versa
 - Message passing
 - P and Q exchange messages
- We will focus on shared memory, will discuss issues with message passing later



Implementation Questions

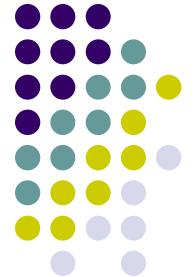
- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?



Producer Consumer Problem

- Paradigm for cooperating processes
- *producer* process produces information that is consumed by a *consumer* process.
 - *unbounded-buffer* places no practical limit on the size of the buffer.
 - *bounded-buffer* assumes that there is a fixed buffer size.
- Basic synchronization requirement
 - Producer should not write into a full buffer
 - Consumer should not read from an empty buffer
 - All data written by the producer must be read exactly once by the consumer

Bounded-Buffer – Shared-Memory Solution



- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- We will see how to create such shared memory between processes in the lab



Bounded-Buffer: Producer Process

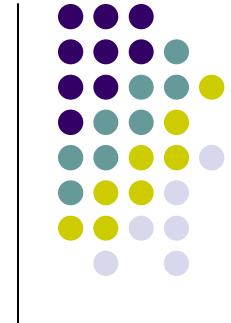
item nextProduced;

```
while (1) {
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
```

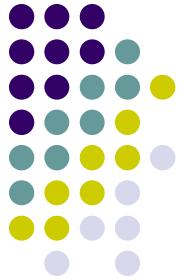


Bounded-Buffer: Consumer Process

```
item nextConsumed;  
  
while (1) {  
    while (in == out)  
        ; /* do nothing */  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
}
```



- The solution allows at most $n - 1$ items in buffer (of size n) at the same time. A solution, where all n items are used is not simple
- Suppose we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer



Shared data

```
#define B_SIZE 10
typedef struct {
    ...
} item;
item buffer[B_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Will this work?

Producer process

```
item nextProduced;
while (1) {
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer process

```
item nextConsumed;
while (1) {
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```



The Problem with this solution

- The statement “*counter++*” may be implemented in machine language as:

register1 = counter

register1 = register1 + 1

counter = register1

- The statement “*counter--*” may be implemented as:

register2 = counter

register2 = register2 – 1

counter = register2



- If both the producer and consumer attempt to update *counter* concurrently, the assembly language statements may get interleaved.
- Interleaving depends upon how the producer and consumer processes are scheduled.



An Illustration

- Assume *counter* is initially 5. One interleaving of statements is:

producer: *register1* = *counter* (*register1* = 5)

producer: *register1* = *register1* + 1 (*register1* = 6)

consumer: *register2* = *counter* (*register2* = 5)

consumer: *register2* = *register2* – 1 (*register2* = 4)

producer: *counter* = *register1* (*counter* = 6)

consumer: *counter* = *register2* (*counter* = 4)

- The value of *counter* may be either 4 or 6, where the correct result should be 5.



Race Condition



- A scenario in which the final output is dependent on the relative speed of the processes
 - Example: The final value of the shared data *counter* depends upon which process finishes last
- Race conditions must be prevented
 - Concurrent processes must be **synchronized**
 - Final output should be what is specified by the program, and should not change due to relative speeds of the processes



Atomic Operation

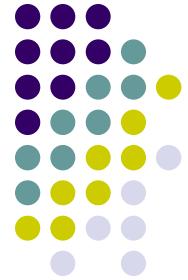


- An operation that is either executed fully without interruption, or not executed at all
 - The “operation” can be a group of instructions
 - Ex. the instructions for *counter++* and *counter--*
 - Note that the producer-consumer problem’s solution works if *counter++* and *counter--* are made atomic
 - In practice, the process may be interrupted in the middle of an atomic operation, but the atomicity should ensure that no process uses the effect of the partially executed operation until it is completed



The Critical Section Problem

- n processes, all competing to use some shared data (in general, use some shared **resource**)
- Each process has **a section of code**, called **critical section**, in which a shared data is accessed.
- **Problem** – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section, irrespective of the relative speeds of the processes
- Also known as the **Mutual Exclusion Problem** as it requires that access to the critical section is mutually exclusive



Requirements for Solution to the Critical-Section Problem

Mutual Exclusion: If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

Progress: If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

Bounded Waiting/No Starvation: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.



Entry and Exit Sections

- **Entry section:** a piece of code executed by a process just before entering a critical section
- **Exit section:** a piece of code executed by a process just after leaving a critical section
- General structure of a process P_i

entry section

critical section

exit section

*remainder section /*remaining code */*

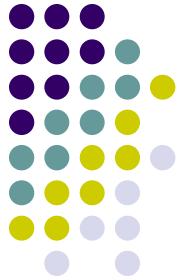
- Solutions vary depending on how these sections are written



Petersen's Solution

- Only 2 processes, P_0 and P_1
- Processes share some common variables to synchronize their actions
 - *int turn = 0*
 - $turn = i \Rightarrow P_i$'s turn to enter its critical section
 - *boolean flag[2]*
 - initially $flag [0] = flag [1] = false$
 - $flag [i] = true \Rightarrow P_i$ ready to enter its critical section

int j = 1 - i j is not shared



- Process P_i

```
do {
```

```
    flag [i]:= true;
```

```
    turn = j;
```

```
    while (flag [j] and turn == j) ;
```

```
        critical section
```

```
        flag [i] = false;
```

```
        remainder section
```

```
} while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes
 - Can be extended to n processes by pairwise mutual exclusion – too costly

Problem with S/W based solution

- Software based solutions such as Peterson's solution for critical section are not guaranteed to work on modern computer architecture. Why? Atomic LOAD, STORE!!!
- Any solution to the critical section problem requires a simple tool – **a lock**.
- Race conditions are prevented by **requiring that critical regions be protected** by locks.
- A process must acquire a lock before entering a critical region; it releases the lock when it exits the critical section.

Disabling Interrupts

- Critical section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified.
- Currently running code would execute without preemption.
- Too inefficient on multiprocessor systems as OS using this not broadly scalable.

Why?

As disabling interrupts can be time consuming as the message is passed to all the processors.



Hardware Instruction Based Solutions

- Some architectures provide special instructions that can be used for synchronization
- **TestAndSet**: Test and modify the content of a word **atomically**

```
boolean TestAndSet (boolean &target) {  
    boolean rv = target;  
    target = true;  
    return rv;  
}
```



- *Swap*: *Atomically* swap two variables.

```
void Swap(boolean &a, boolean &b) {  
    boolean temp = a;  
    a = b;  
    b = temp;  
}
```

Mutual Exclusion with Test-and-Set



- Shared data:

boolean lock = false;

- Process P_i

```
do {  
    while (TestAndSet(lock)) ;  
    critical section  
    lock = false;  
    remainder section  
}while(TRUE);
```



Mutual Exclusion with Swap

- Shared data (initialized to *false*):
boolean lock;
boolean waiting[n];
- Process P_i
do {
key = true;
while (key == true)
 Swap(lock,key);
 critical section
lock = false;
 remainder section
}while(TRUE);

Problem with TestAndSet()/Swap()

Although these satisfy the mutual exclusion requirement, they do not satisfy the bounded-waiting.

Modified Version

```
do{
    waiting[i]=TRUE;
    key=TRUE;
    while(waiting[i]&&key)
        key=TestAndSet(&lock);
    waiting[i]=FALSE;
    // critical section
    j=(i+1)%n;
    while((j!=i)&&!waiting[j])
        j=(j+1)%n;
    if(j==i)
        lock=FALSE;
    else
        waiting[j]=FALSE;
    // REMAINDER SECTION
}while(TRUE);
```

Shared variables are:

boolean waiting[n];
boolean lock;

Initialized to FALSE

Satisfies all the three requirements

Drawback:

For hardware designers, implementing atomic TestAndSet() instructions on multiprocessors is not a trivial task.

Semaphores

- Much easier to implement
- Semaphore S – an integer variable
- Two standard operations modify S:

 wait()

 signal()

- These two operations are atomic
- wait() operation was originally termed P(from the Dutch *proberen*, “*to test*”)
- signal() was originally called V(from *verhogen*, “*to increment*”)

Semaphore Operations

wait(S)

{

 while(S<=0); // No operation, called busy waiting, spin lock

 S--;

}

signal(S)

{

 S++;

}

Where S is an integer.

All the modifications to the integer value of the semaphore are **atomic**. That is, when one process modifies the semaphore value, no other process can simultaneously modify the same semaphore.

Types of Semaphores

- 
- Counting semaphore – can be any integer
 - Binary semaphore – can be 0 or 1, also known as **mutex locks**, as they are locks providing mutual exclusion.

Semaphore usage

Mutual Exclusion

We can use binary semaphores to deal with the critical section problem for multiple processes.

Then “n” processes share a semaphore mutex, initialized to 1.

Each process P_i organized as shown below:



```
while(TRUE)
{
    wait(mutex);
    // critical section
    signal(mutex)
    // remainder section
}
```

Semaphore usage

Process Synchronization

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running process: P_1 with a statement S_1 and P_2 with a statement S_2 . Suppose we want that S_2 can be executed only after S_1 has completed. We can implement this scheme readily by letting P_1 and P_2 share a common semaphore **synch**, initialized to 0, and by inserting the statements

```
S1;  
signal(synch); } P1  
  
wait(synch); } P2  
S2;
```

Because **synch** is initialized to 0, P_2 will execute S_2 only after P_1 has invoked **signal(synch)**, which is after statement S_1 has been executed.

Semaphore usage

Control Access To Resource With Finite Number of Instances

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource perform a wait() operation on that semaphore(thereby decrementing the count). When a process releases a resource, it performs a signal() operation(incrementing the count). When the count for semaphore goes to 0, all resources are being used. After that, processes that wish to use resources will block until the count becomes greater than 0.

Disadvantage

- The main disadvantage of the semaphore definition given here is that it requires **busy waiting**.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- This continual looping is clearly a problem in a real world multiprogramming system, where a single CPU is shared among many processes.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of semaphore is also called a **spinlock** because the process spins while waiting for the lock.

Semaphore without busy waiting

- To overcome busy waiting, we modify the definition of semaphore, wait() and signal() to **block and resume** processes to increase CPU utilization.
- Rather than engaging in busy waiting, a process can block itself. The block operation places a process into a **waiting queue** associated with the semaphore, and the state of the process is **switched to waiting state**. Then the control is transferred to the CPU scheduler, which selects another process to execute.
- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is **restarted** by a **wakeup()** operation, which changes the process from **waiting state to ready state**.

New semaphore definition

```
typedef struct  
{  
    int value;  
    struct process *list;  
}semaphore;
```

Each semaphore has an integer value and a queue of processes.

1

The **wait()** semaphore operation can now be defined as:

```
wait()(semaphore *S)  
{  
    S->value--;  
    if(S->value<0)  
    {  
        add this process to S->list;  
        block();  
    }  
}
```

2

The **signal()** semaphore operation can now be defined as:

```
signal(semaphore *S)  
{  
    S->value++;  
    if(S->value<=0)  
    {  
        remove process P from S->list;  
        wakeup(P);  
    }  
}
```

3

Note that in this implementation, semaphore values may be negative. If a semaphore value is **negative**, its magnitude is the number of **processes waiting** on that semaphore.

4

Semaphore list(queue), S->list can be implemented using any queuing strategy(e.g., FIFO, LIFO, Priority).

5

The above implementation satisfies mutual exclusion and progress. **Bounded waiting** is dependent on the queueing strategy of the semaphore list.

Pitfalls

Use carefully to avoid

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended

Example of Deadlock

Let S and Q be two semaphores initialized to 1

P_0
wait(S);
wait(Q);

.

.

.

signal(S);
signal(Q);

P_1
wait(Q);
wait(S);

.

.

.

signal(Q);
signal(S);

Example of Starvation

Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO order.

Precautions while using Semaphores

Incorrect use of semaphores can result in serious errors that are difficult to detect.

Example 1

Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex)  
.....  
critical section  
.....  
wait(mutex);
```

This may result in several processes running in their critical section simultaneously.

Example 2

Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes

```
wait(mutex);  
.....  
critical section  
.....  
wait(mutex);
```

In this case, a deadlock will occur.

Example 3

Suppose that a process omits the wait(mutex), or signal(mutex), or both. In this case either mutual exclusion is violated or a deadlock will occur.

Deadlocks & Deadlock Prevention

IIIT BHUBANESWAR

Muktikanta Sahu

Department of Computer Science & Engineering

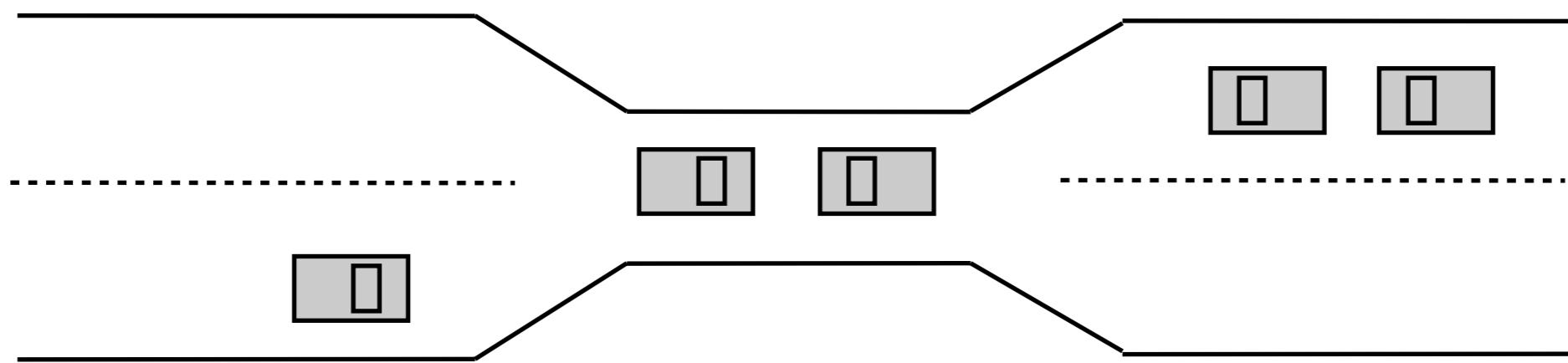
The Deadlock Problem

- **Deadlock - A condition that arises when two or more processes are waiting indefinitely for an event that can only be caused by one of the waiting processes**

- The events that the processes are waiting for will never happen :-(
- Other processes in the system that need the resources that are currently locked will never get access to them
- **Operating systems do not typically provide deadlock-prevention facilities**
 - It is up to the programmer to design deadlock-free programs
- **Deadlock will become an even greater problem with the trend toward more processor cores and more threads**

Bridge Crossing Example

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



Deadlock Characterization

• Deadlock can arise if the four conditions hold simultaneously

- Mutual exclusion - only one process at a time can use a resource
- Hold and wait - a process holding at least one resource is waiting to acquire additional resources held by other processes
- No preemption of resources - a resource can be released only voluntarily by the process holding it, after that process has completed its task
- Circular wait - there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0

Resource-Allocation Graph

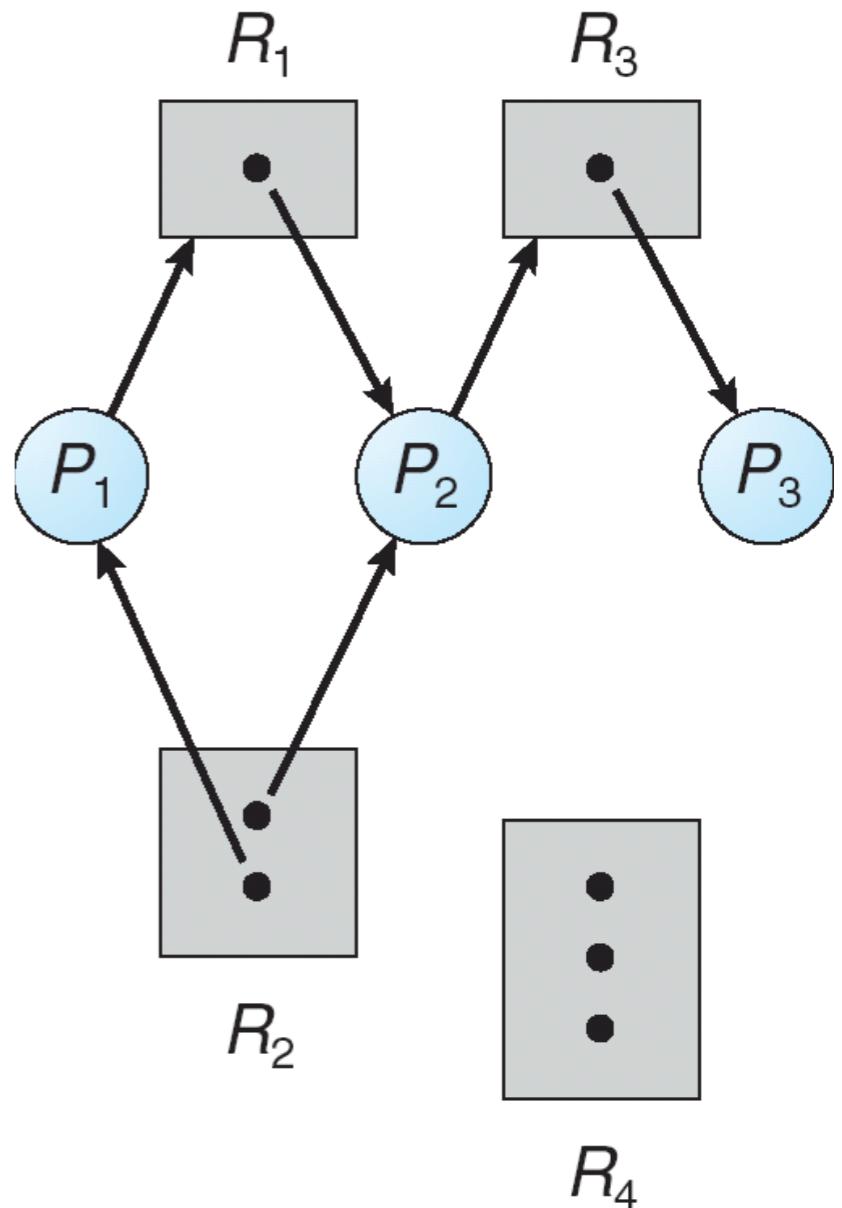
• Deadlocks can be described precisely using system resource-allocation graphs

• A resource-allocation graph consists of a set of vertices V and a set of edges E

- V is partitioned into two sets:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- There are also two different types of edges:
 - Request edge – directed edge $P_i \rightarrow R_j$, a process P_i is requesting a resource of type R_j
 - Assignment edge – directed edge $R_j \rightarrow P_i$, a resource of type R_j has been assigned to a process P_i

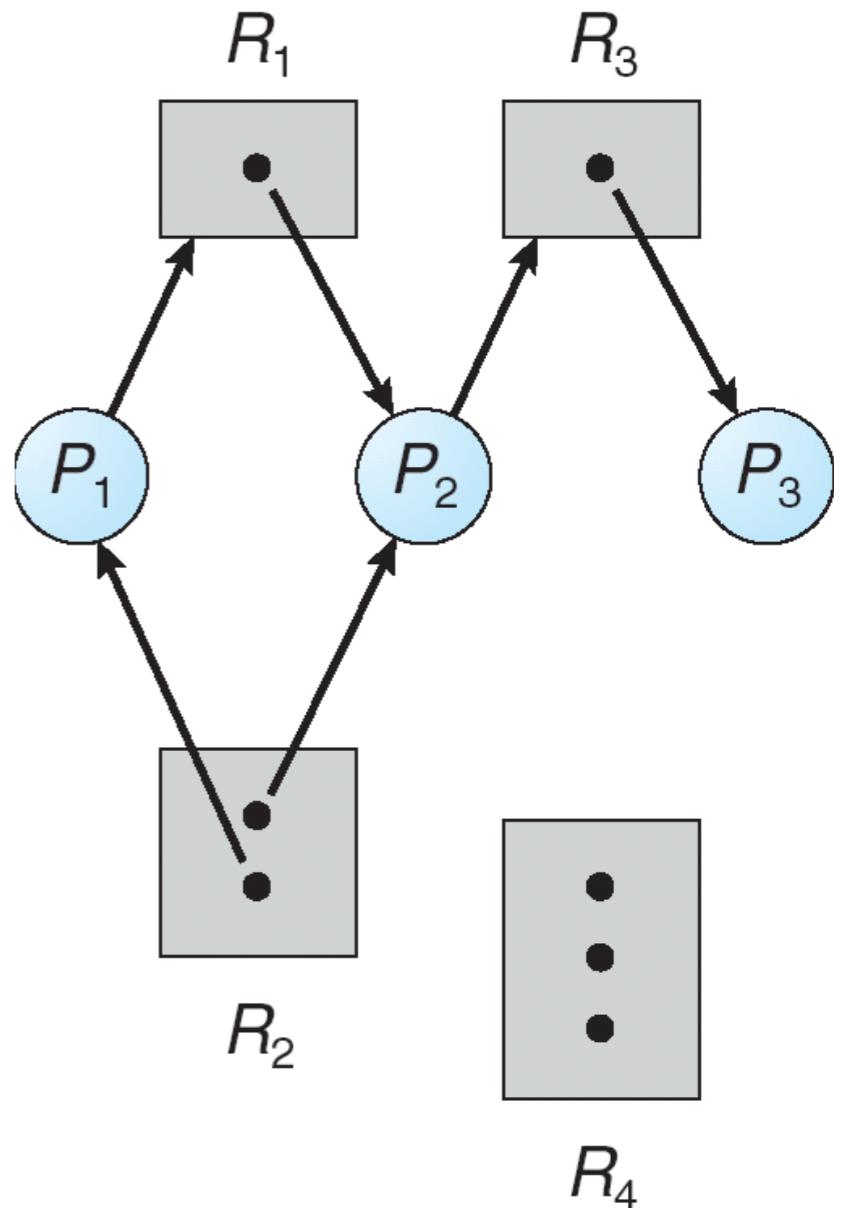
Resource-Allocation Graph (Cont.)

- A process is represented as a circular vertex in the graph
- A resource type is represented as a square vertex in the graph
 - Because there may be multiple instances of some resources (i.e. two disks), each instance of a resource is represented using a dot
- A directed edge from a process P_i to a resource R_j is a request from P_i for a resource of type R_j
- A directed edge from a resource instance to a process P_i indicates that the resource has been allocated to P_i



Resource-Allocation Graph (Cont.)

- If the graph contains no cycles, then no process in the system is deadlocked
- If the graph does contain cycles, then deadlock may exist, but is not guaranteed to exist
 - If a resource type has exactly one instance then a cycle implies that a deadlock has occurred
 - If a resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred



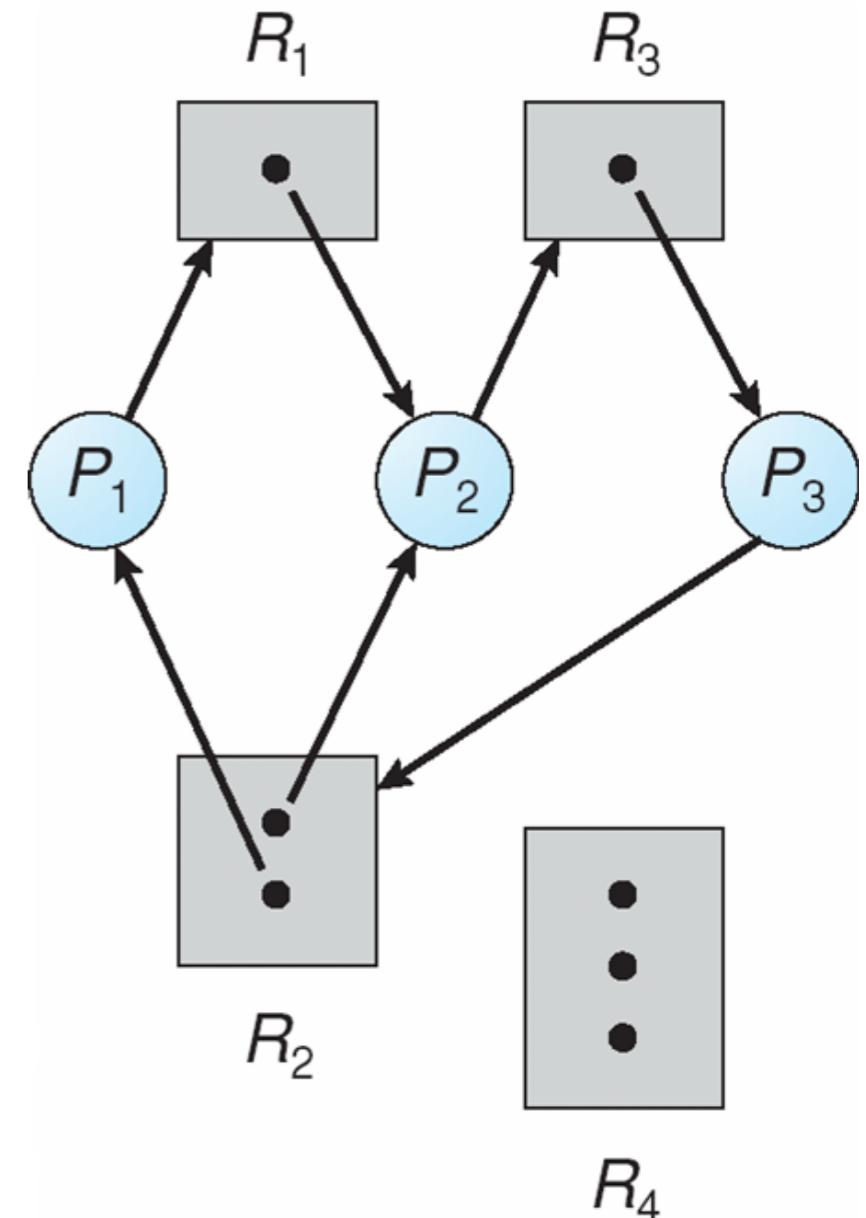
Resource Allocation Graph With A Deadlock

- Two cycles exist in the graph to the right

- $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

- Processes P_1 , P_2 and P_3 are deadlocked

- P_2 is waiting for R_3 (held by P_3)
- P_3 is waiting for R_2 (held by P_1 and P_2)
- P_1 is waiting for R_1 (held by P_2)



Graph With a Cycle But No Deadlock

- A cycle exists in the graph to the right

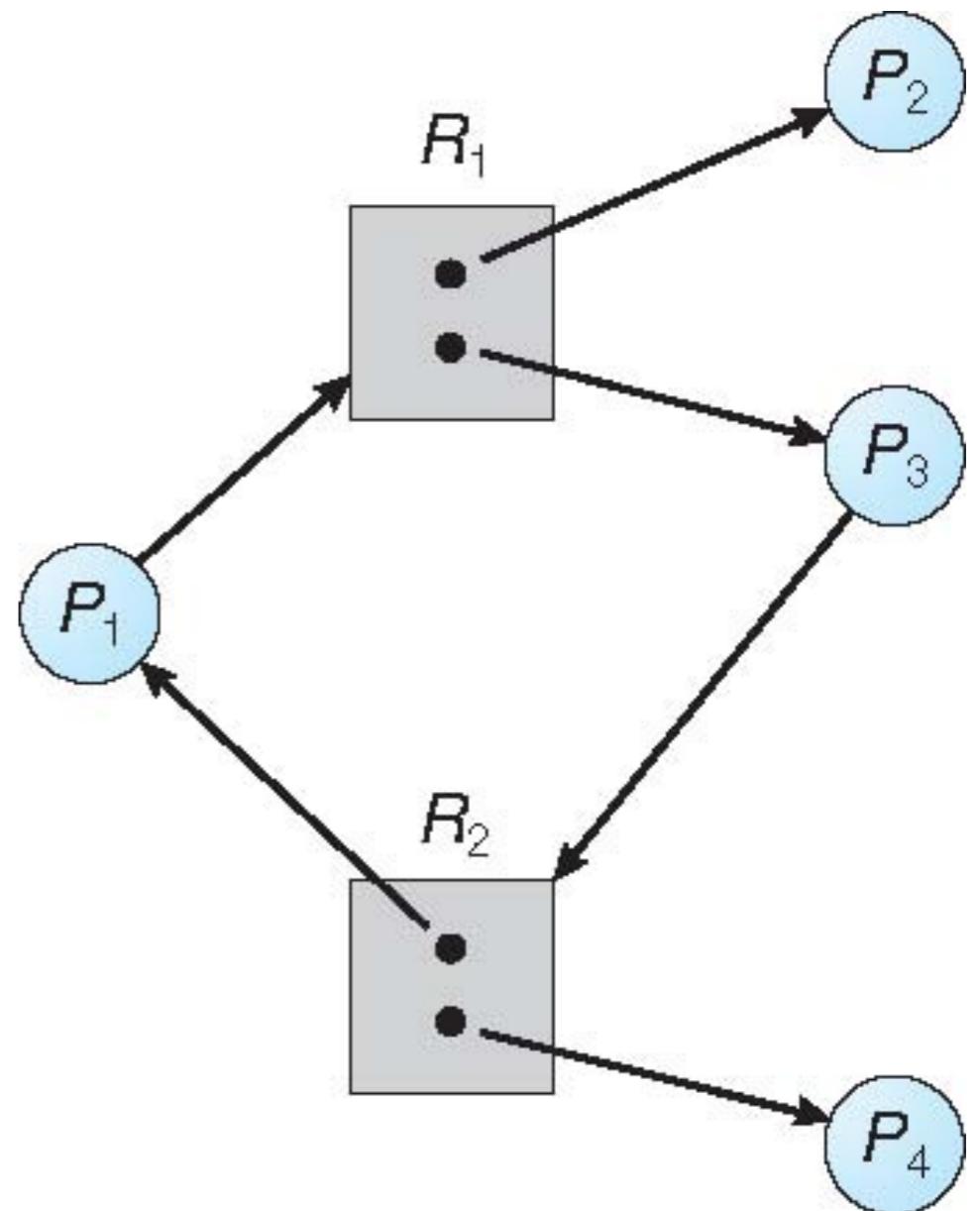
- $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

- Processes P_3 is waiting for resource R_2

- P_4 should eventually release R_2

- Process P_1 is waiting for resource R_1

- P_2 should eventually release R_1



Resource Allocation Graph Recap

- **If graph contains no cycles then no deadlock possible**
- **If graph contains a cycle then,**
 - If only one instance per resource type, then deadlock exists
 - If several instances per resource type, then deadlock is possible

Methods for Handling Deadlocks

- **Possible for OS to deal with deadlock in one of three ways:**
 - Ensure that the system will never enter a deadlock state
 - Allow the system to enter a deadlock state and then recover
 - Ignore the problem and pretend that deadlocks never occur in the system
(used by most operating systems, including UNIX)
 - Requires that application developers write programs that avoid deadlock

Methods for Handling Deadlocks

- To ensure that deadlocks never occur, a system can use either deadlock-prevention or deadlock-avoidance
- Deadlock Prevention - ensure that at least one of the four necessary conditions for deadlock cannot hold
- Deadlock Avoidance - requires that the operating system be given comprehensive information about which resources a process will request during its lifetime
 - Operating system can then make intelligent decisions about when a process should be allocated a resource

Methods for Handling Deadlocks

- If a system does not use either deadlock-prevention or deadlock-avoidance, a deadlock situation may eventually arise
 - Need some way to detect these situations -- deadlock detection
 - Need some way to recover from these situations -- deadlock recovery
- If a system does not use deadlock-detection/deadlock-avoidance, and does not use deadlock-detection/deadlock-recovery, then the system performance may eventually deteriorate
 - Reboot

Deadlock Prevention

Muktikanta Sahu

Department of Computer Science



International Institute of Information Technology Bhubaneswar
muktikanta@iiit-bh.ac.in

February 28, 2019

Deadlock Prevention

Overview

- Mutual Exclusion
- Hold and wait
- No preemption
- Circular wait

Deadlock Prevention

Do not allow one of the four conditions to occur.

Mutual Exclusion

- Automatically holds for printers and other non-sharable resources.
- Shared entities such as read-only files don't need mutual exclusion.
- Prevention not possible, since some resources are intrinsically non-sharable.

Deadlock Prevention

Do not allow one of the four conditions to occur.

Hold and wait

- Collect all resources before execution.
- A particular resource can only be requested when no others are being held.
- Results in low resource utilization. Starvation possible.

Do not allow one of the four conditions to occur.

No preemption

- If a process holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently held are released.
- Preempted resources are added to the list of resources for which the process is running.
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- This protocol is often applied to resources whose states can be saved and restored later, such as CPU registers and memory space. It cannot be generally applied to such resources as printers and tape drives.

Deadlock Prevention

Do not allow one of the four conditions to occur.

Circular wait

Impose total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Let $R = \{R_1, R_2, \dots, R_m\}$ be the set of resource types. Each resource type is assigned with a unique integer.

We define a **one-to-one** function $F : R \rightarrow N$, where N is the set of natural numbers. For example, if the set of resource type R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

$$F(\text{tapedrive}) = 1$$

$$F(\text{diskdrive}) = 5$$

$$F(\text{printer}) = 12$$

Deadlock Prevention

Do not allow one of the four conditions to occur.

Circular wait

We now consider the following protocols to prevent deadlocks:

- Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type- say, R_i . After that, the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$.
- Alternatively, we can require that a process requesting an instance of resource type R_j must have released any resource R_i such that $F(R_i) \geq F(R_j)$.

If these two protocols are used, then the circular wait condition cannot hold.

Deadlock Prevention

Do not allow one of the four conditions to occur.

Circular wait

Proof by contradiction:

Let the set of processes involved in circular wait be $\{P_0, P_1, \dots, P_n\}$, where P_i is waiting for resource R_i , which is held by process P_{i+1} . (Modulo arithmetic is used on the indexes, so that P_n is waiting for a resource R_n held by P_0).

Since, $R_i \rightarrow P_{i+1} \rightarrow R_{i+1}$

$\implies F(R_i) < F(R_{i+1})$ for all i

$\implies F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$

\implies By transitivity $F(R_0) < F(R_0)$, which is impossible.

\implies Therefore, there can be no circular wait.

Possible side effects of preventing deadlocks are, low device utilization and reduced system throughput.

Thank you

Deadlock Avoidance

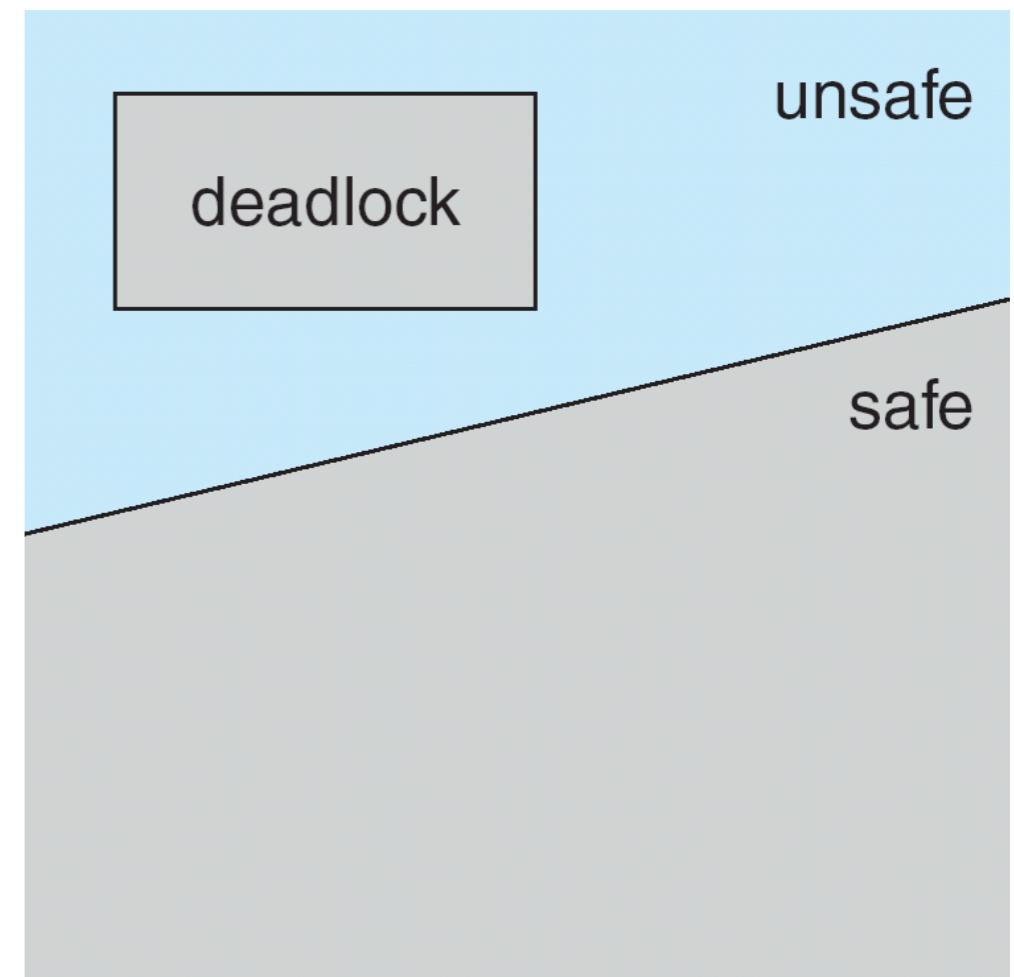
- **Requires that the system has some additional *a priori* information available**
 - Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need
 - A deadlock-avoidance algorithm dynamically examines the resource-allocation *state* to ensure that there can never be a circular-wait condition
 - Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
-  • In a safe state if the system can allocate resources to each process (up to its maximum) in some order and still avoid deadlock
- System is in safe state if there exists a sequence $\langle P_1, P_2, \dots, P_n \rangle$ of ALL the processes in the systems such that for each P_i , the resources that P_i can still request can be satisfied by currently available resources plus resources held by all the P_j , with $j < i$
 - That is:
 - If resources needed by P_i are not immediately available, then P_i can wait until all P_j have finished
 - When P_j is finished, P_i can obtain the resources it needs, execute, return allocated resources, and terminate
 - When P_i terminates, P_{i+1} can obtain the resources it needs, and so on

Safe, Unsafe, Deadlock State

- If a system is in safe state \Rightarrow no deadlocks
- If a system is in unsafe state \Rightarrow possibility of deadlock
- To avoid deadlock, ensure that a system will never enter an unsafe state



Avoidance Algorithms

- **Two main avoidance algorithms ... choice depends on how many instances of available resource types exist**



- Single instance of a resource type
 - Use a resource-allocation graph



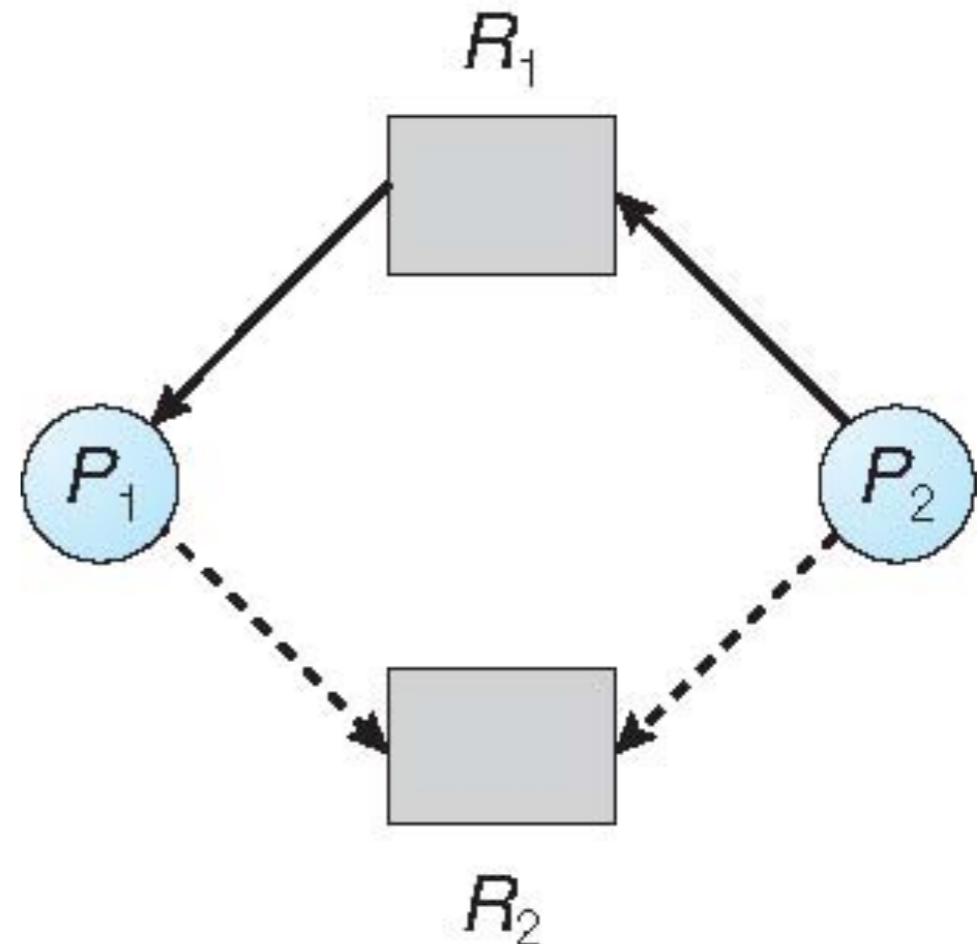
- Multiple instances of a resource type
 - Use the Banker's Algorithm

Resource-Allocation Graphs for Deadlock Avoidance

- Introduce a new type of edge to the resource-allocation graph, the claim edge
 - A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request a resource R_j
 - A claim edge is represented as dashed lines in resource-allocation graph
 - Resources must be claimed a priori in the system, so all claim edges are known
- A claim edge converts to request edge when a process actually requests a resource
- A request edge is converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, an assignment edge converts back to a claim edge

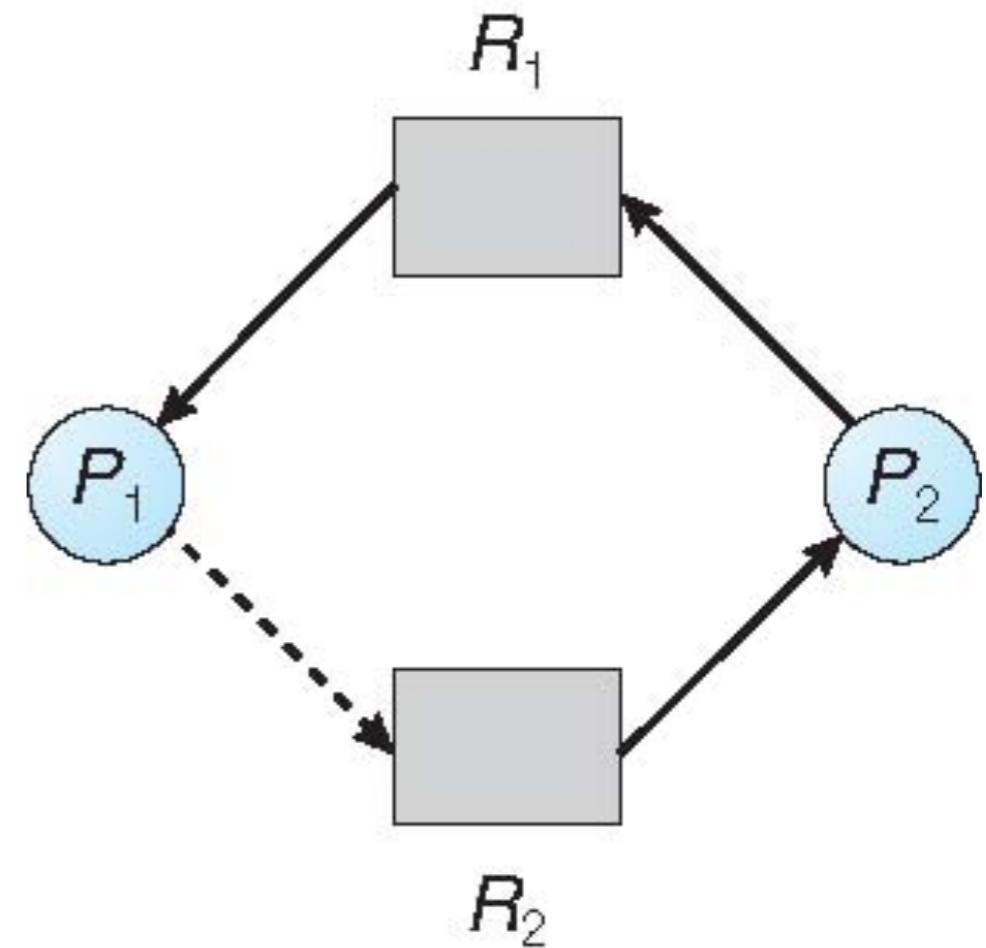
Resource-Allocation Graph

- In this graph, both processes P_1 and P_2 may eventually want to request resource R_2
 - Claim edges exists, because the system knows in advance that P_1 and P_2 may both want to request R_2
- Currently, there are no cycles in the graph so the system is in a safe state
- If process P_2 were to request resource R_2 and be granted that resource then ...



Unsafe State In Resource-Allocation Graph

- If process P_2 were to request resource R_2 and be granted that resource then a cycle is formed in the resource-allocation graph
- The system is not yet deadlocked, but is in an unsafe state
- If process P_1 were to request resource R_2 , then the system would be in a deadlocked state
- To prevent an unsafe state and the possibility of deadlock, P_2 should not be allocated resource R_2 when it makes the request, P_2 must wait for P_1 to release R_1 so that no cycles are created



Banker's Algorithm



• Use for deadlock avoidance when multiple instances of resources exist

- Each process must *a priori* claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time
- Banker's algorithm needs to know three things to work:
 - How much of each resource each process could possibly request
 - How much of each resource each process is currently holding
 - How much of each resource the system currently has available

Data Structures for the Banker's Algorithm

- Let $n = \text{number of processes}$, and $m = \text{number of resource types}$
 - *Available* -- Vector of length m that indicates the number of available resources of each type. If $\text{Available}[j] = k$, there are k instances of resource type R_j available
 - *Max* -- $n \times m$ matrix that defines the maximum demand of each process on each resource type. If $\text{Max}[i,j] = k$, then process P_i may request at most k instances of resource type R_j
 - *Allocation* -- $n \times m$ matrix that defines the number of resources of each type currently allocated to each process. If $\text{Allocation}[i,j] = k$ then P_i is currently allocated k instances of R_j
 - *Need* -- $n \times m$ matrix that indicates the remaining resource needs of each process. If $\text{Need}[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$

Banker's Algorithm -- Safety Procedure

- Used to determine whether or not a system is in a safe state
- Local Data Structures
 - *Finish* -- a vector[1 .. n] , initialize as *false* for each process P_i
- The procedure to check for a safe state:

(1) Find a process P_i such that $Finish[i] = \text{false}$ and $Need_i \leq Available$

If P_i exists do the following:

$Available = Available + Allocation_i$

$Finish[i] = \text{true}$

Go back to Step #1

Let X and Y be vectors of length n. We say that

- $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all $i=1, 2, \dots, n$

- $Y < X$ if $Y \leq X$ and $Y \neq X$

(2) If no such process P_i exists

If $Finish[i] = \text{true}$ for all processes P_i for $i=1 \dots n$, then the system is in a safe state

Otherwise, processes with a $Finish[i]$ value of *false* are in an unsafe state and can potentially deadlock

Example of Banker's Algorithm-Check Safe State

- Consider a system with
 - Five processes $P_0 - P_4$
 - Three resource types, A (10 instances), B (5 instances), and C (7 instances)
- At some time t_0 , the system looks like the following:

	<u>Allocation</u>			<u>Max</u>		
	A	B	C	A	B	C
P_0	0	1	0	7	5	3
P_1	2	0	0	3	2	2
P_2	3	0	2	9	0	2
P_3	2	1	1	2	2	2
P_4	0	0	2	4	3	3

<u>Available</u>		
A	B	C
3	3	2

- Is this a safe state?

Example of Banker's Algorithm-Check Safe State (Cont.)

- The $n \times m$ matrix **Need** is defined as
 - $\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$
- The sequence $\langle P_1, P_3, P_0, P_2, P_4 \rangle$ is a safe sequence (there are other safe sequences as well)

	<u>Allocation</u>			<u>Max</u>			<u>Need</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3	3	3	2
P_1	2	0	0	3	2	2	1	2	2			
P_2	3	0	2	9	0	2	6	0	0			
P_3	2	1	1	2	2	2	0	1	1			
P_4	0	0	2	4	3	3	4	3	1			

Banker's Algorithm -- Resource-Request

- Used to determine whether requests for resources can be safely granted
- Local Data Structure
 - Request_i -- a vector[1 .. m], is a request vector for process P_i , the number of additional resources, of each type 1.. m , that process P_i is requesting
- The procedure to determine if resource can be safely granted
 - (1) If ($\text{Request}_i > \text{Need}_i$) then raise an error -- process P_i is trying to get more resources than it initially declared it would need
 - (2) If ($\text{Request}_i > \text{Available}_i$) then process P_i must wait because there are not currently sufficient resources available to satisfy the request
 - (3) Pretend to allocate the requested resources to process P_i
 $\text{Available} = \text{Available} - \text{Request}_i$
 $\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$
 $\text{Need}_i = \text{Need}_i - \text{Request}_i$
 - (4) Check the safety of the state that would result from step #3
If step#3 produces a safe state, then the resources are allocated for real
If step#3 produces an unsafe state, then don't allocate the resources to process P_i , the process must wait

Example of Banker's Algorithm - P_1 Request

- Since the system is in a safe state, use the Resource Request procedure to determine if the following request can be granted
 - P_1 requests one instance of resource A, and two instances of resource C
The request vector looks like --> $\text{Request}_1 = (1, 0, 2)$
 - First note if the resources are available to fulfill this request: $\text{Request}_1 \leq \text{Available}$

	<u>Allocation</u>			<u>Max</u>			<u>Need</u>			<u>Available</u>
	A	B	C	A	B	C	A	B	C	
P_0	0	1	0	7	5	3	7	4	3	
P_1	2	0	0	3	2	2	1	2	2	
P_2	3	0	2	9	0	2	6	0	0	
P_3	2	1	1	2	2	2	0	1	1	
P_4	0	0	2	4	3	3	4	3	1	

$(1, 0, 2) \leq (3, 3, 2)$

Example of Banker's Algorithm - P_1 Request (Cont.)

- After determining that the resources are actually available, pretend to allocate them to process P_1
- Update the **Allocation**, **Need** and **Available** values
- Check to see if this new state is safe, if yes, then allocate resources for real

	<u>Allocation</u>			<u>Max</u>			<u>Need</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3	3	3	2
P_1	2	0	0	3	2	2	1	2	2	2	3	0
P_1	3	0	2	3	2	2	0	2	0	3	3	2
P_2	3	0	2	9	0	2	6	0	0	2	3	0
P_3	2	1	1	2	2	2	0	1	1	1	0	1
P_4	0	0	2	4	3	3	4	3	1	0	0	1

- Is this a safe state?

The sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ is a safe sequence

Example of Banker's Algorithm - P_1 Request (Cont.)

- What if P_4 requests $(3, 3, 0)$?

Request cannot be granted, not enough resources available

- What if P_0 requests $(0,2,0)$?

Request cannot be granted, system ends up in an unsafe state

	<u>Allocation</u>			<u>Max</u>			<u>Need</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3	2	3	0
P_1	3	0	2	3	2	2	0	2	0			
P_2	3	0	2	9	0	2	6	0	0			
P_3	2	1	1	2	2	2	0	1	1			
P_4	0	0	2	4	3	3	4	3	1			

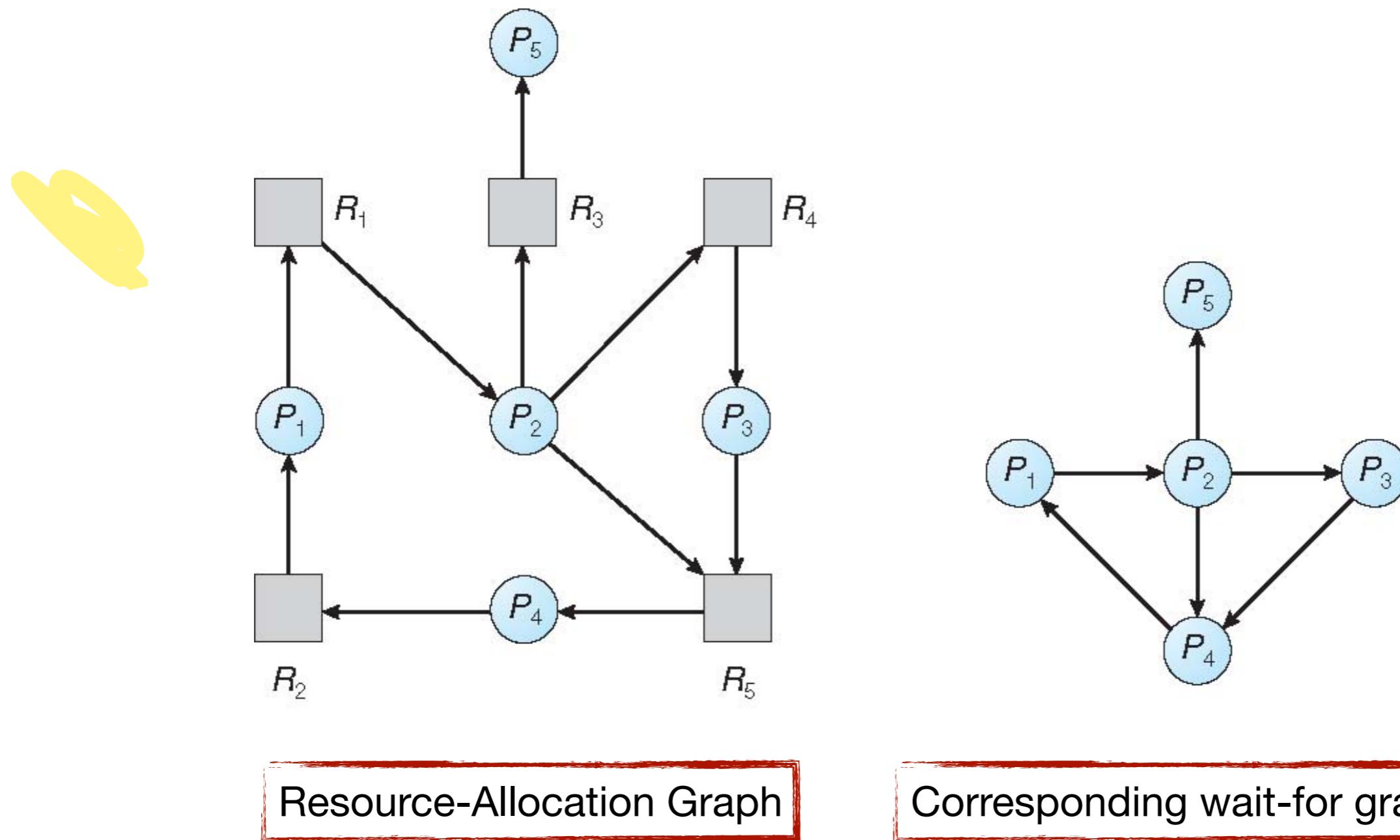
Deadlock Detection

- In systems that don't use deadlock-prevention or deadlock-avoidance, deadlock may occur
- Such a system may provide:
 - A deadlock detection algorithm that determines if a deadlock has occurred in the system
 - A deadlock recovery algorithm to recover from deadlock if it has occurred
- As with deadlock avoidance, systems with multiple instances of a single resource type must be considered differently than systems with only single instance resources

Single Instance of Each Resource Type

- If all resources have only a single instance, use a wait-for graph for deadlock detection
 - All Nodes are processes
 - An edge $P_i \rightarrow P_j$ exists if P_i is waiting for P_j to release a resource
- Deadlock exists in the system if and only if the wait-for graph contains a cycle
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Several Instances of a Resource Type

- The wait-for graph cannot be used in systems with multiple instances of each resource type
- The deadlock detection algorithm for several instance contains similar data structures to the Banker's Algorithm
- Let n = number of processes, and m = number of resource types
 - *Available* -- Vector of length m that indicates the number of available resource of each type. If $\text{available}[j] = k$, there are k instances of resource type R_j available
 - *Allocation* -- $n \times m$ matrix that defines the number of resources of each type currently allocated to each process. If $\text{Allocation}[i,j] = k$ then P_i is currently allocated k instances of R_j
 - *Request* -- $n \times m$ matrix that indicates the current request of each process. If $\text{Request}[i,j] = k$ then process P_i is requesting k more instances of resource type R_j

Deadlock Detection Algorithm

- Used to determine whether or not a system is in a deadlocked state (very similar to Banker's Algorithm safety procedure)
- Local Data Structures
 - *Finish* -- a vector[1 .. n] , initialize as *false* for each process P_i where $Allocation_i \neq 0$, *true* otherwise
- The procedure to check for a deadlocked state:
 - (1) Find a process P_i such that $Finish[i] = \text{false}$ and $Request_i \leq Available$
If P_i exists do the following:
 $Available = Available + Allocation_i$
 $Finish[i] = \text{true}$
Go back to Step #1
 - (2) If no such process P_i exists
If $Finish[i] = \text{false}$ for all processes P_i for $i=1 .. n$, then P_i is in a deadlocked state

Example of Deadlock Detection Algorithm

- Consider a system with
 - Five processes $P_0 - P_4$
 - Three resource types, A (7 instances), B (2 instances), and C (6 instances)
- At some time T_0 , the system looks like the following:

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>
	A	B	C	A	B	C	
P_0	0	1	0	0	0	0	
P_1	2	0	0	2	0	2	
P_2	3	0	3	0	0	0	
P_3	2	1	1	1	0	0	
P_4	0	0	2	0	0	2	

- Is the system deadlocked?

The sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = \text{true}$ for all processes

Example of Deadlock Detection Algorithm (Cont.)

- Continuing the previous example, suppose that process P_2 now requests one additional resource of type C
- At some time T_0 , the system looks like the following:

	<u>Allocation</u>			<u>Request</u>		
	A	B	C	A	B	C
P_0	0	1	0	0	0	0
P_1	2	0	0	2	0	2
P_2	3	0	3	0	0	1
P_3	2	1	1	1	0	0
P_4	0	0	2	0	0	2

	<u>Available</u>		
	A	B	C
	0	0	0

- Is the system deadlocked?

Can only reclaim resources held by P_0 . Insufficient resources to handle the requests from other processes. **Deadlock exists** consisting of processes P_1 , P_2 , P_3 , and P_4

Deadlock Detection Algorithm Usage

- **When, and how often, to invoke the detection algorithm depends on:**
 - How often a deadlock is likely to occur?
 - How many processes will need to be rolled back?
- **If the detection algorithm is invoked every time a request for allocation cannot be granted:**
 - The algorithm can detect specifically which processes caused the deadlock
 - There is a lot of computational overhead when running the algorithm so frequently
- **If the detection algorithm is invoked less frequently (e.g. once per hour):**
 - It is difficult to determine which processes caused the deadlock
 - The computational overhead can be reduced

Deadlock Recovery

- **Once deadlock has been detected in a system there are several options for dealing with deadlock**
 - Notify the system user and let them deal with it
 - Have the system terminate one or more processes
 - Have the system preempt resources from one or more of the deadlocked processes

Deadlock Recovery: Process Termination

- When terminating processes to recover from deadlock, resources are reclaimed from any terminated process
 - Terminating a process means that work done by that process may be lost
- Two options:
 - Abort all deadlocked processes
 - Potential to lose a lot of work / computation time
 - Abort one process at a time until the deadlock cycle is eliminated
 - Lots of overhead since the deadlock-detection algorithm must be run after each process is terminated

Deadlock Recovery: Process Termination (Cont.)

- **If terminating processes, in what order should processes be terminated?**
- **Consider:**
 - Priority of the process
 - How long process has computed, and how much longer to completion
 - Resources the process has used
 - Resources process needs to complete
 - How many processes will need to be terminated
 - Is process interactive or batch?

Deadlock Recovery: Resource Preemption

- Continually preempt the resources from processes and give those resources to other processes until the system is no longer deadlocked
- Must consider the following:
 - How should victims be selected? And how can cost be minimized?
 - How should a preempted process get rolled back?
 - Would like to return the process to some safe state without having to completely restart the process. To do so requires some fancy bookkeeping by the system to know where the most recent safe state was prior to the deadlock.
 - How to ensure that starvation does not occur?
 - Don't want resources to always get preempted from the same process

OPERATING SYSTEMS

MEMORY MANAGEMENT



OPERATING SYSTEM

Memory Management

What Is In This Chapter?

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

MEMORY MANAGEMENT

Definitions

- The concept of a logical *address space* that is bound to a separate *physical address space* is central to proper memory management.
 - **Logical address** – generated by the CPU; also referred to as *virtual address*
 - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

MEMORY MANAGEMENT

Definitions

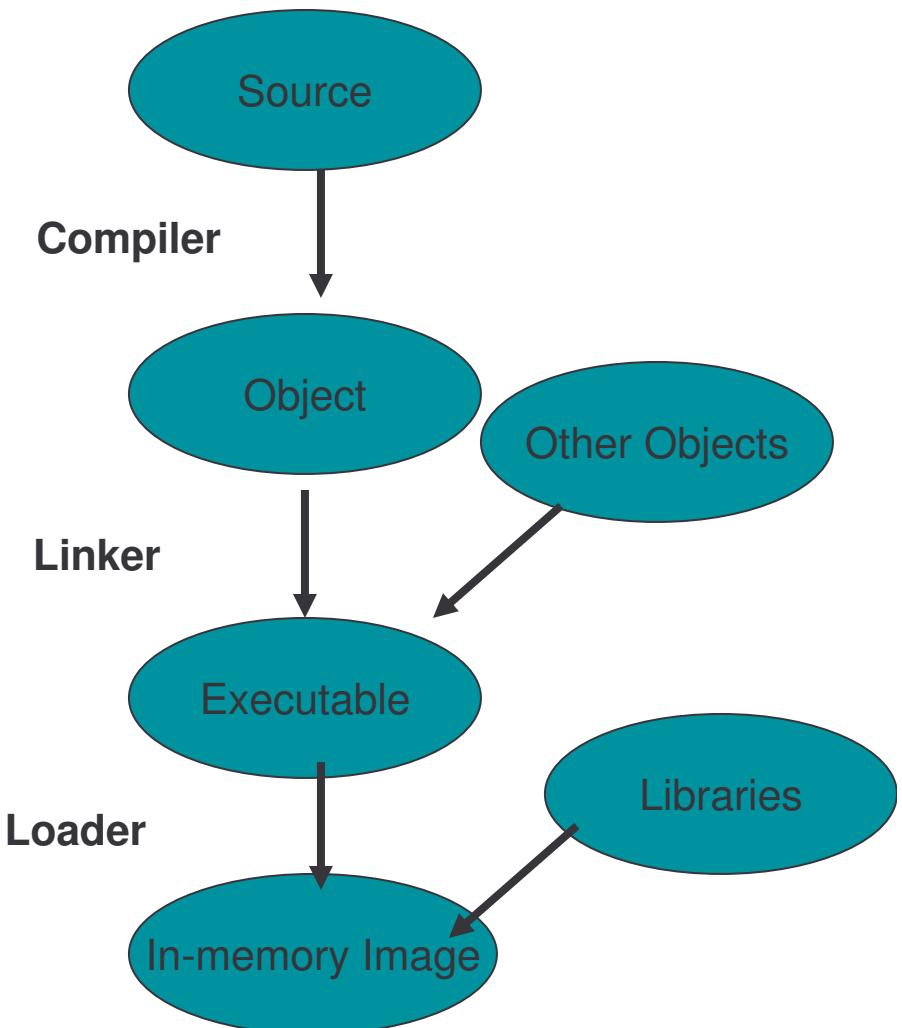
- Relocatable** Means that the program image can reside anywhere in physical memory.
- Binding** Programs need real memory in which to reside. When is the location of that real memory determined?
- This is called **mapping** logical to physical addresses.
 - This binding can be done at compile/link time. Converts symbolic to relocatable. Data used within compiled source is offset within object module.
- Compiler:** If it's known where the program will reside, then absolute code is generated. Otherwise compiler produces relocatable code.
- Loader:** Binds relocatable to physical. Can find best physical location.
- Execution:** The code can be moved around during execution. Means flexible virtual mapping.

MEMORY MANAGEMENT

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

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

Binding Logical To Physical



MEMORY MANAGEMENT

More Definitions

Dynamic loading

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

Dynamic Linking

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

Memory Management

Performs the above operations. Usually requires hardware support.

Operating System

Muktikanta Sahu

Department of Computer Science



IIIT Bhubaneswar
Imagine, Innovate, Inspire
(A University established by Government of Odisha)

International Institute of Information Technology Bhubaneswar
muktikanta@iiit-bh.ac.in

February 27, 2020

Create modules to be added to a static library

Create a header file named as calculate.h and add the following function declarations.

Listing 1: calculate.h

```
1     int add( int a, int b);
2     int sub( int a, int b);
```

Create definition files for add() and sub().

Listing 2: add.cpp

```
1     int add( int a, int b)
2     {return a+b;} // Save it as add.cpp
```

Listing 3: sub.cpp

```
1     int sub( int a, int b)
2     {return a-b;} // Save it as sub.cpp
```

Create object files of the definition files

Generate object modules for the library

Listing 4: Create add.o

```
1 g++ -c -o add.o add.cpp
```

Listing 5: Create sub.o

```
1 g++ -c -o sub.o sub.cpp
```

Create a static linked library

Listing 6: Create staticlib.a

```
1      ar ru staticlib.a add.o sub.o
```

Create a driver program

Create a driver program named as staticmain.cpp to be linked with the static library.

Listing 7: Create staticmain.cpp

```
1 #include<iostream>
2 #include"calculate.h"
3 int main()
4 {
5     int x,y;
6     std :: cout<<" Enter value for x"<<std :: endl;
7     std :: cin>>x;
8     std :: cout<<" Enter value for y"<<std :: endl;
9     std :: cin>>y;
10    std :: cout<<"Sum="<<add(x,y)<<std :: endl;
11    std :: cout<<" Difference="<<sub(x,y)<<std :: endl;
12    return 0;
13 }
```

Static linking of the driver program with the library

Now, do the static linking of the driver program with the library.

Listing 8: Static linking

```
1      g++ -o staticmain staticmain.cpp staticlib.a
2      staticmain // To run the driver program
```

Create modules to be added to a dynamic or shared library

Create a header file named as calculate.h and add the following function declarations.

Listing 9: calculate.h

```
1 int add( int a , int b );
2 int sub( int a , int b );
```

Create definition files for add() and sub().

Listing 10: add.cpp

```
1 int add( int a , int b )
2 { return a+b; } // Save it as add.cpp
```

Listing 11: sub.cpp

```
1 int sub( int a , int b )
2 { return a-b; } // Save it as sub.cpp
```

Create object files of the definition files

Generate object modules for the library

Listing 12: Create add.o

```
1 g++ -c -o add.o add.cpp
```

Listing 13: Create sub.o

```
1 g++ -c -o sub.o sub.cpp
```

Create a shared or dynamic library

Listing 14: Create dynamiclib.so

```
1 g++ -fPIC -shared -o dynamiclib.so add.o sub.o
```

Create a driver program

Create a driver program named as staticmain.cpp to be linked with the static library.

Listing 15: Create dynamicmain.cpp

```
1 #include<iostream>
2 #include"calculate.h"
3 int main()
4 {
5     int x,y;
6     std :: cout<<" Enter value for x"<<std :: endl;
7     std :: cin>>x;
8     std :: cout<<" Enter value for y"<<std :: endl;
9     std :: cin>>y;
10    std :: cout<<"Sum="<<add(x,y)<<std :: endl;
11    std :: cout<<" Difference="<<sub(x,y)<<std :: endl;
12    return 0;
13 }
```

Dynamic linking of the driver program with the library

Now, do the dynamic linking of the driver program with the library.

Listing 16: Dynamic linking

```
1      g++ -o dynamicmain dynamicmain.cpp dynamiclib.s
2      dynamicmain // To run the driver program
```

Static Linking vs Dynamic Linking

Static Linking	Dynamic Linking
Static linking is the process of copying all library modules used in the program into the final executable image. This is performed by the linker and it is done as the last step of the compilation process. The linker combines library routines with the program code in order to resolve external references, and to generate an executable image suitable for loading into memory. When the program is loaded, the operating system places into memory a single file that contains the executable code and data. This statically linked file includes both the calling program and the called program.	In dynamic linking the names of the external libraries (shared libraries) are placed in the final executable file while the actual linking takes place at run time when both executable file and libraries are placed in the memory. Dynamic linking lets several programs use a single copy of an executable module.
Static linking is performed by programs called linkers as the last step in compiling a program. Linkers are also called link editors.	Dynamic linking is performed at run time by the operating system.
Statically linked files are significantly larger in size because external programs are built into the executable files.	In dynamic linking only one copy of shared library is kept in memory. This significantly reduces the size of executable programs, thereby saving memory and disk space.
In static linking if any of the external programs has changed then they have to be recompiled and re-linked again else the changes won't reflect in existing executable file.	In dynamic linking this is not the case and individual shared modules can be updated and recompiled. This is one of the greatest advantages dynamic linking offers.
Statically linked program takes constant load time every time it is loaded into the memory for execution.	In dynamic linking load time might be reduced if the shared library code is already present in memory.
Programs that use statically-linked libraries are usually faster than those that use shared libraries.	Programs that use shared libraries are usually slower than those that use statically-linked libraries.
In statically-linked programs, all code is contained in a single executable module. Therefore, they never run into compatibility issues.	Dynamically linked programs are dependent on having a compatible library. If a library is changed (for example, a new compiler release may change a library), applications might have to be reworked to be made compatible with the new version of the library. If a library is removed from the system, programs using that library will no longer work.

MEMORY MANAGEMENT

SINGLE PARTITION ALLOCATION

BARE MACHINE:

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

DEFINITION OF PARTITIONS:

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

MEMORY MANAGEMENT

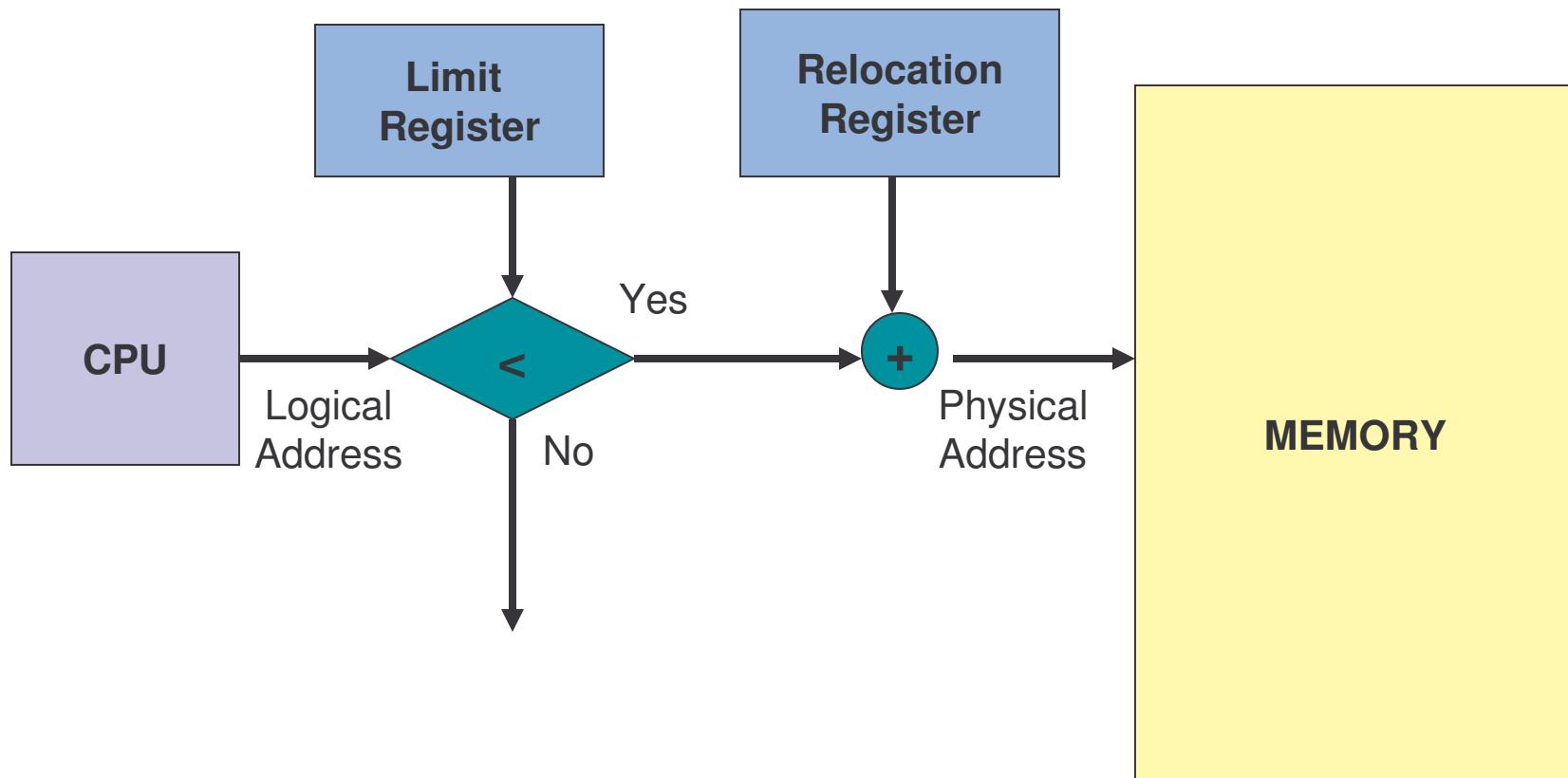
SINGLE PARTITION ALLOCATION

RESIDENT MONITOR:

- Primitive Operating System.
- Usually in low memory where interrupt vectors are placed.
- Must check each memory reference against fence (fixed or variable) in hardware or register. If user generated address < fence, then illegal.
- User program starts at fence -> fixed for duration of execution. Then user code has fence address built in. But only works for static-sized monitor.
- If monitor can change in size, start user at high end and move back, OR use fence as base register that requires address binding at execution time. Add base register to every generated user address.
- Isolate user from physical address space using logical address space.
- Concept of "mapping addresses" shown on next slide.

MEMORY MANAGEMENT

SINGLE PARTITION ALLOCATION



MEMORY MANAGEMENT

JOB SCHEDULING

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

CONTIGUOUS ALLOCATION



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

Fundamental Memory Management Problem

- How do we manage applications whose size may be larger than the size of memory available?
 - Partition in blocks and load as necessary
- How do we share memory resources among different processes?
- Achieved by partitioning memory
 - Look at several schemes

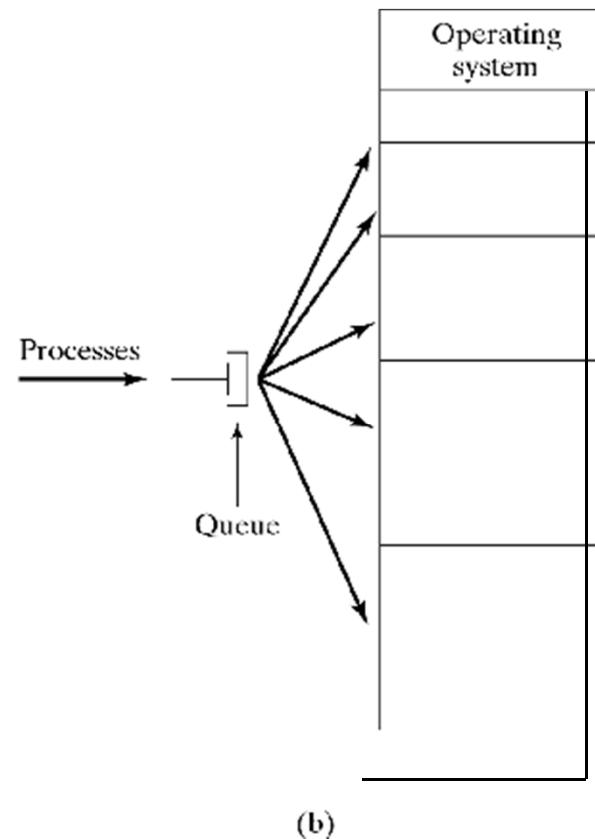
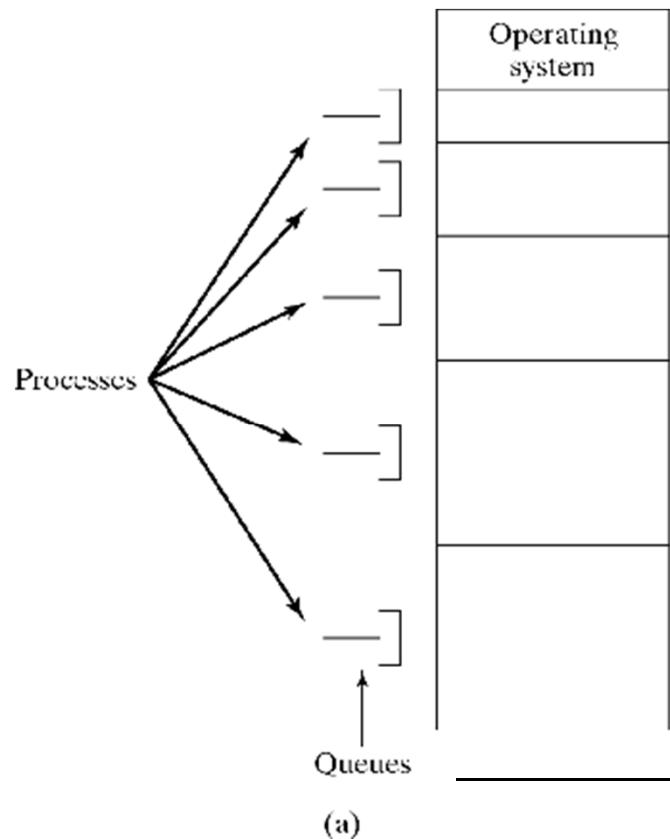
Memory Partitioning Schemes

- Memory sharing schemes:
 - Single-program systems: 2 partitions (OS/user)
 - Multi-programmed:
 - Divide memory into partitions of different sizes
- Fixed partitions: size of partition determined at the time of OS initialization and cannot be changed
- Limitations of fixed partitions
 - Program size limited to largest partition
 - Internal fragmentation (unused space within partitions)
- How to assign processes to partitions
 - FIFO for each partition: Some partitions may be unused
 - Single FIFO: More complex, but more flexible
 - Choose the one that fits the best

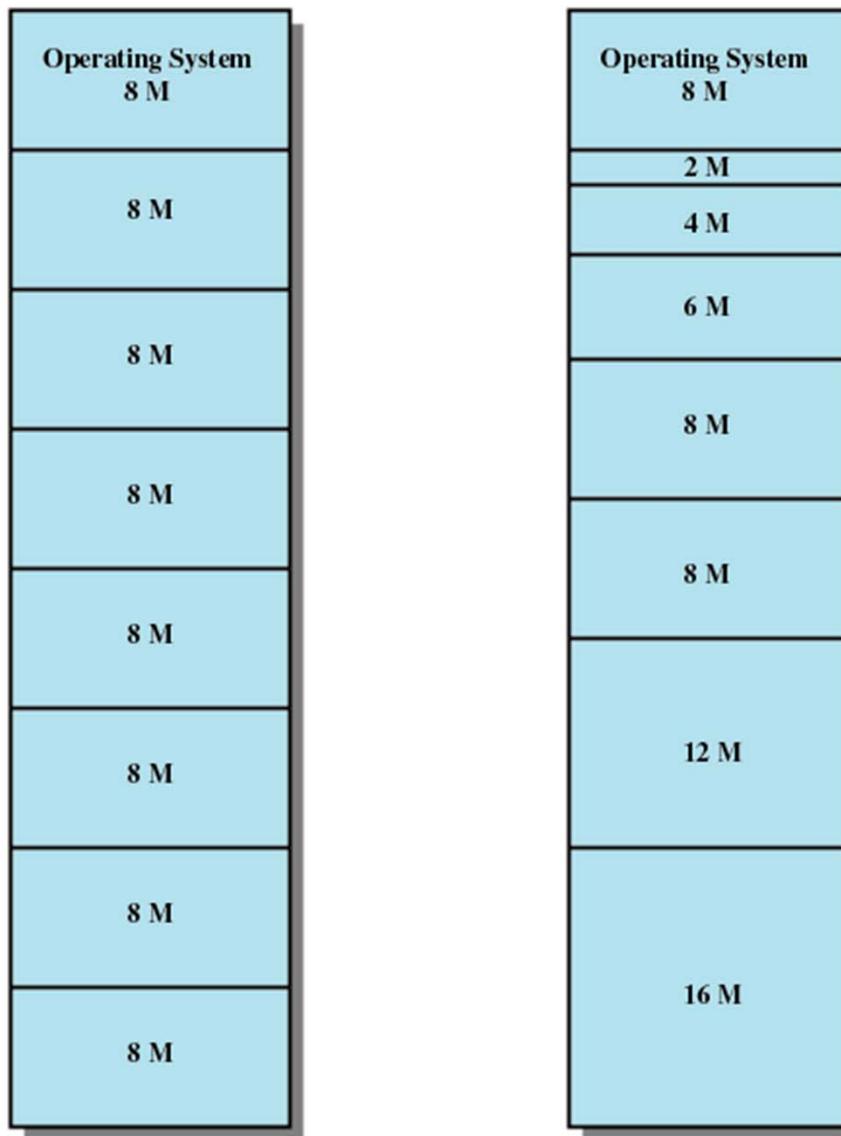
Fixed Partitioning

Fixed partitions:

1 queue per partition *vs* 1 queue for all partitions



Example: Fixed Partitioning



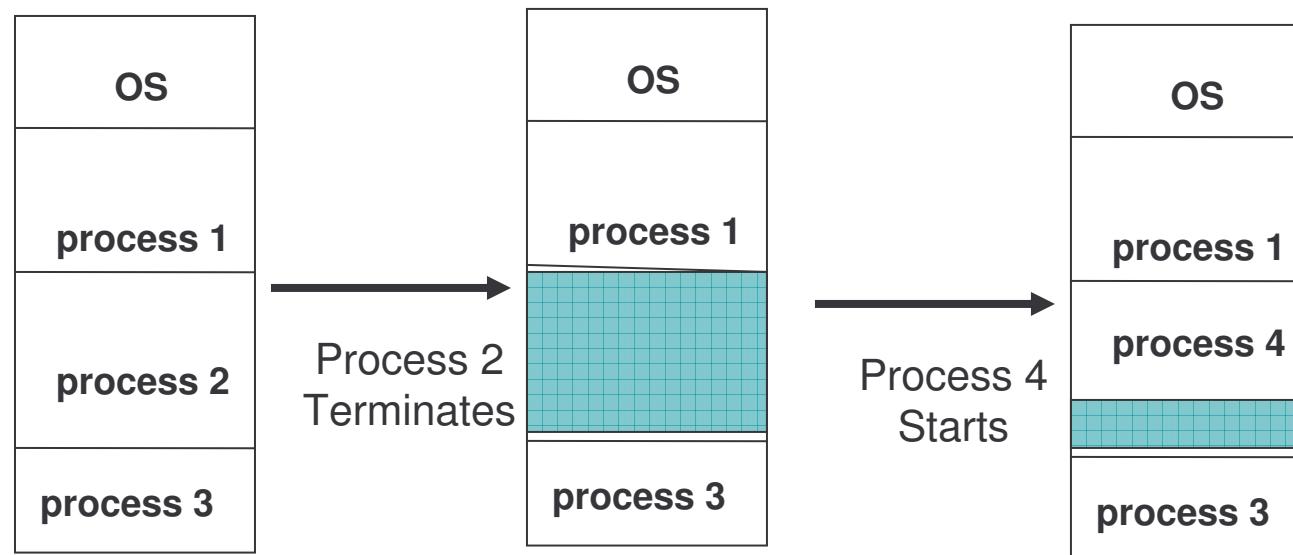
MEMORY MANAGEMENT

DYNAMIC STORAGE

CONTIGUOUS ALLOCATION

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

ALLOCATION PRODUCES HOLES



MEMORY MANAGEMENT

CONTIGUOUS ALLOCATION

HOW DO YOU ALLOCATE MEMORY TO NEW PROCESSES?

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

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

Worst fit - allocate largest hole.

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

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

The Buddy System

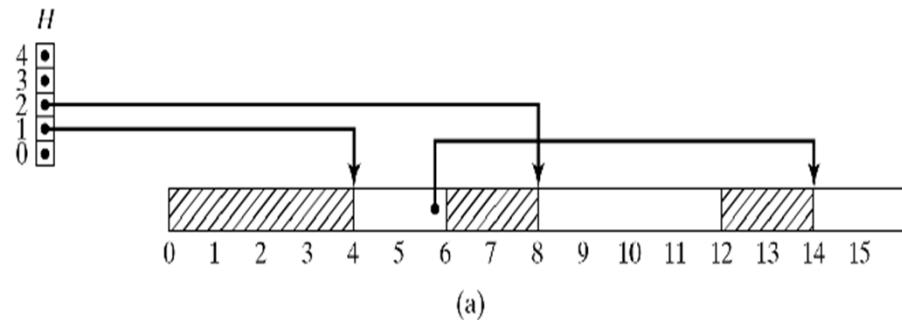
- Compromise between fixed and variable partitions
- Fixed number of possible hole sizes; typically, 2^i
 - Each hole can be divided (equally) into 2 *buddies*.
 - Track holes by size on separate lists
- When n bytes requested, find smallest i so that $n \leq 2^i$
 - If hole of this size available, allocate it;
otherwise, consider larger holes.
Recursively split each hole into two buddies
until smallest adequate hole is created
.Allocate it and place other holes on appropriate lists
- On release, recursively coalesce buddies
 - Buddy searching for coalescing can be inefficient

The Buddy System

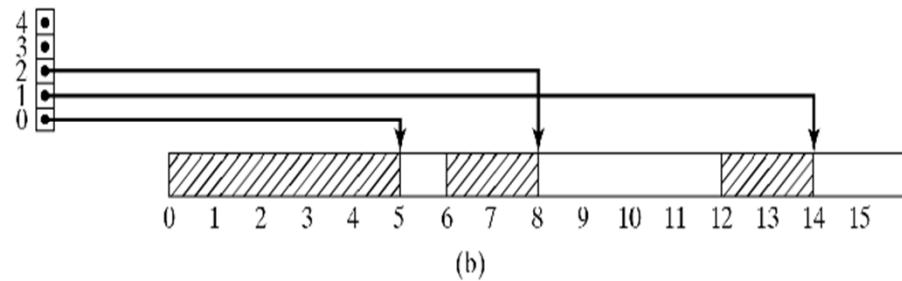
- Assume: Memory of 16 allocation units

Sizes: 1, 2, 4, 8, 16

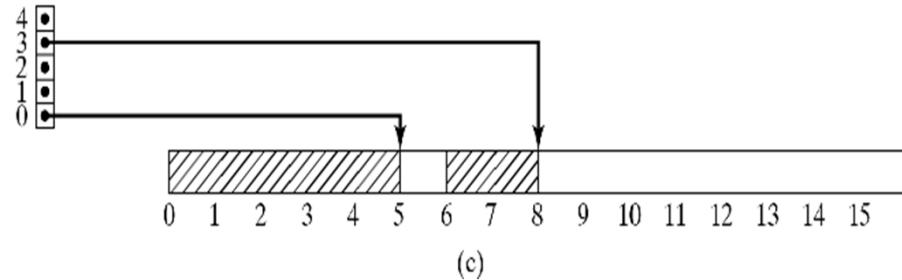
- a) 3 blocks allocated
& 3 holes left



- b) Block of size 1 allocated



- c) Block 12-13 released



MEMORY MANAGEMENT

LONG TERM SCHEDULING

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

- wait for memory
- skip to next job and see if it fits.

What are the pros and cons of each of these?

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

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

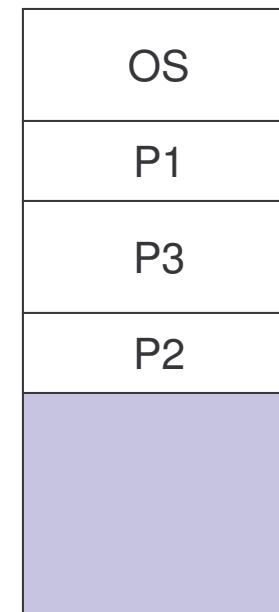
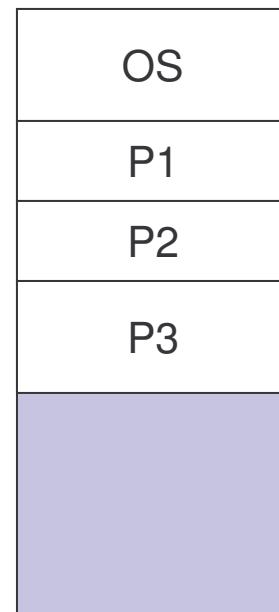
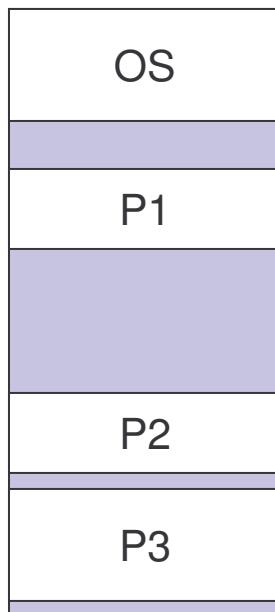
MEMORY MANAGEMENT COMPACTION

Trying to move free memory to one large block.

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

There are many ways to move programs in memory.

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



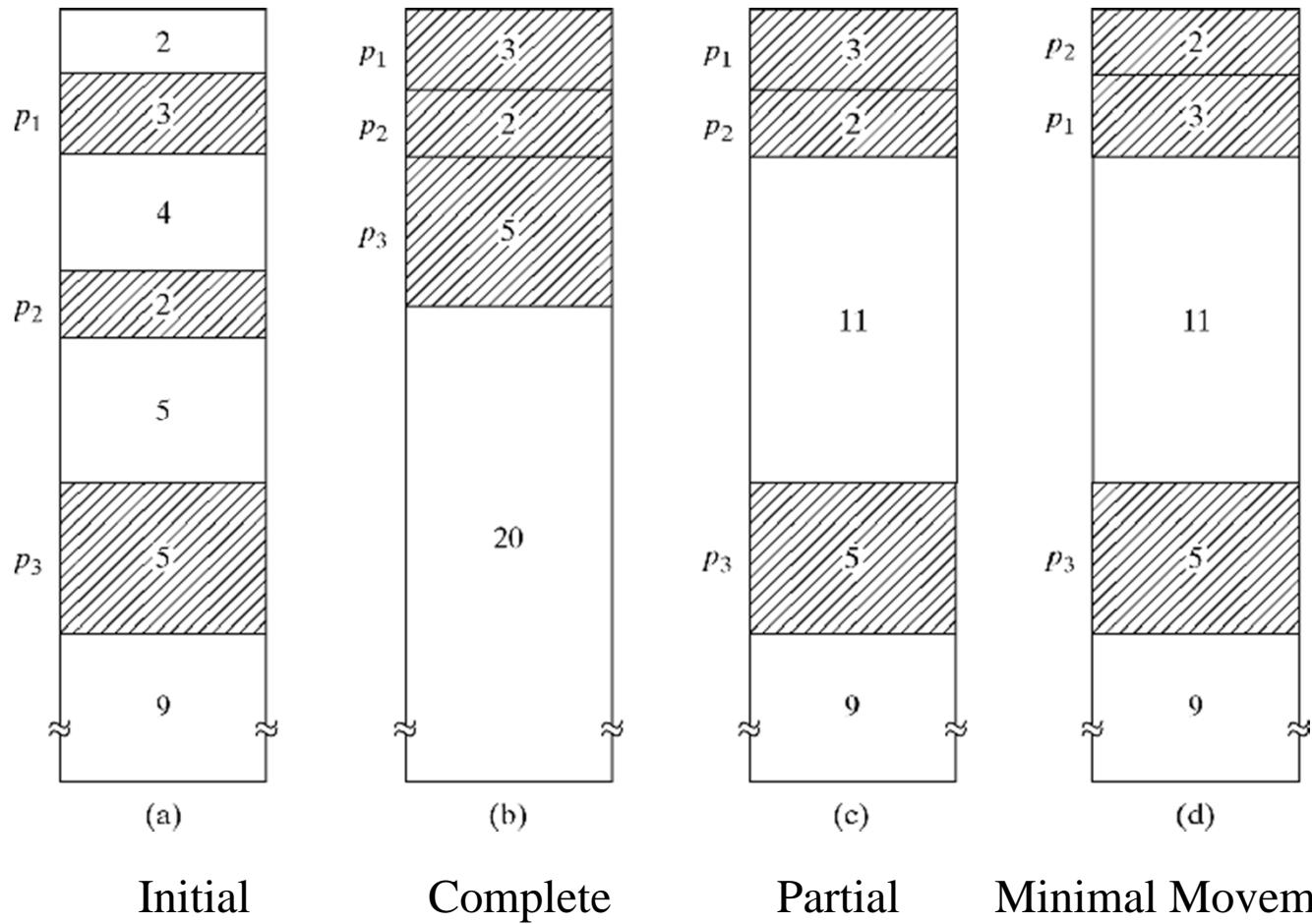
8: Memory Management

Dealing with Insufficient Memory

- Memory compaction
 - How much and what to move?
- Swapping
 - Temporarily move process to disk
 - Requires dynamic relocation
- Overlays
 - Allow programs large than physical memory
 - Programs loaded as needed according to calling structure.

Dealing with Insufficient Memory

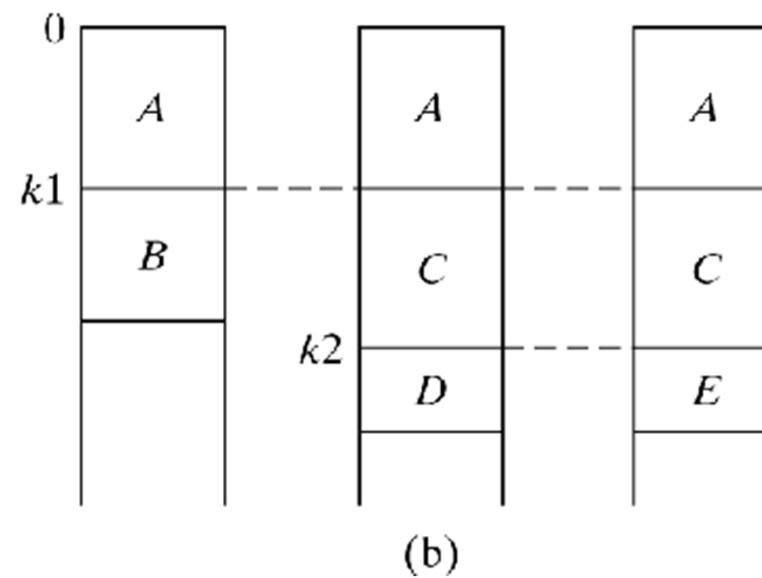
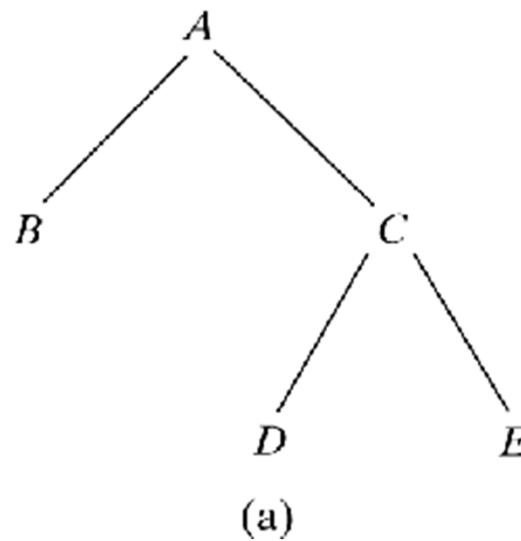
Memory compaction



Dealing with Insufficient Memory

Overlays

- Allow programs large than physical memory
- Programs loaded as needed according to calling structure



Example:

Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB and 600 KB (in order), how the first-fit, best-fit , and worst-fit algorithm would place processes of 212 KB, 417 KB, 112 KB and 426 KB (in-order)? Which algorithm makes the most efficient use of memory?

Answer:

First-fit	Best-fit	Worst-fit
212 KB → 500 KB A hole of 288 KB would be created	212 KB → 300 KB A hole of 88 KB would be created	212 KB → 600 KB A hole of 388 KB would be created
417 KB → 600 KB	417 KB → 500 KB A hole of 83 KB would be created	417 KB → 500 KB A hole of 83 KB would be created
112 KB → New hole of 288 KB A whole of 176 KB would be created	112 KB → 200 KB	112 KB → New hole of 388 KB A new hole of 276 KB would be created
426 KB must wait	426 KB → 600 KB A new hole of 174 KB would be created	426 KB must wait

MEMORY MANAGEMENT

PAGING

New Concept!!



- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever that memory is available and the program needs it.
- Divide **physical** memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes).
- Divide **logical** memory into blocks of same size called **pages**.
- Keep track of all free frames.
- To run a program of size n pages, need to find n free frames and load program.
- Set up a page table to translate logical to physical addresses.
- Internal fragmentation.

MEMORY MANAGEMENT

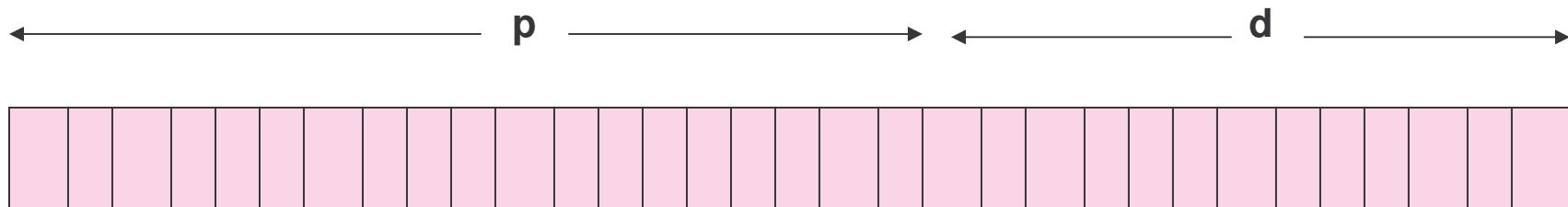
PAGING

Address Translation Scheme

Address generated by the CPU is divided into:

- *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory.
- *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit.

4096 bytes = 2^{12} – it requires 12 bits to contain the Page offset



MEMORY MANAGEMENT

PAGING

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

Frames = physical blocks

Pages = logical blocks

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

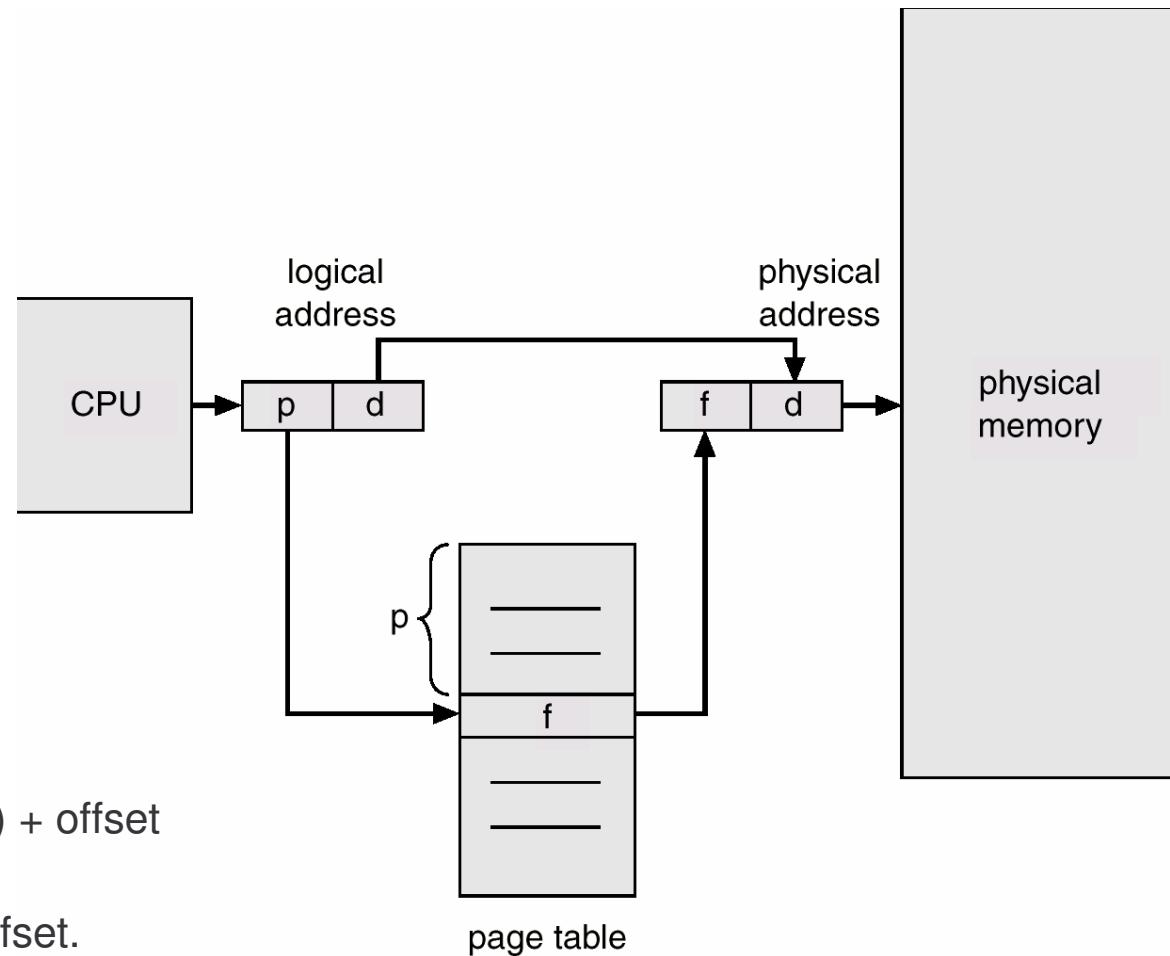
HARDWARE

An address is determined by:

page number (index into table) + offset

---> mapping into --->

base address (from table) + offset.



MEMORY MANAGEMENT

PAGING

Paging Example - 32-byte memory with 4-byte pages

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

Logical Memory

0	5
1	6
2	1
3	2

Page Table

Physical Memory

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

8: Memory Management

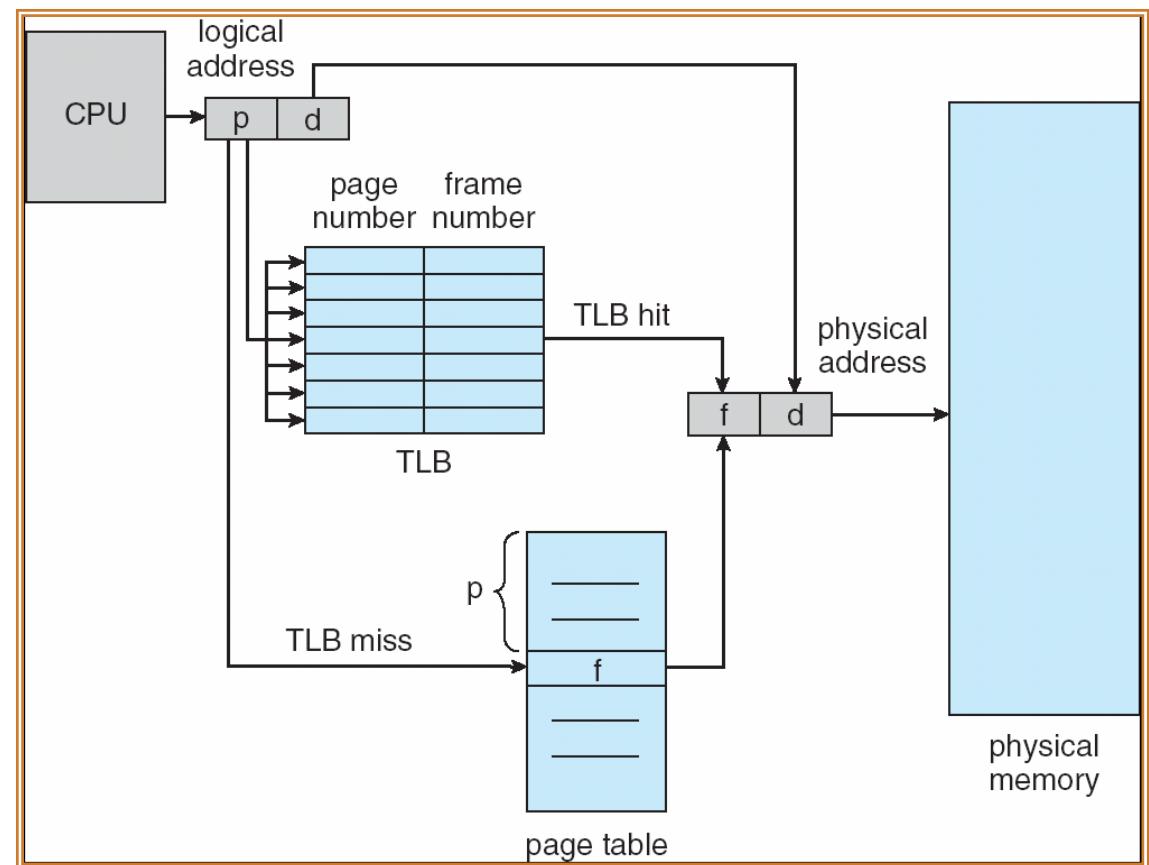
MEMORY MANAGEMENT

PAGING

- A 32 bit machine can address 4 gigabytes which is 4 million pages (at 1024 bytes/page). WHO says how big a page is, anyway?
- Could use dedicated registers (OK only with small tables.)
- Could use a register pointing to table in memory (slow access.)
- Cache or associative memory
- (TLB = Translation Lookaside Buffer):
- simultaneous search is fast and uses only a few registers.

IMPLEMENTATION OF THE PAGE TABLE

TLB = Translation Lookaside Buffer



MEMORY MANAGEMENT

PAGING

IMPLEMENTATION OF THE PAGE TABLE

Issues include:

key and value

hit rate 90 - 98% with 100 registers

add entry if not found

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

Relevant times:

2 nanoseconds to search associative memory – the TLB.

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

Calculate time of access:

hit = 1 search + 1 memory reference

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

MEMORY MANAGEMENT

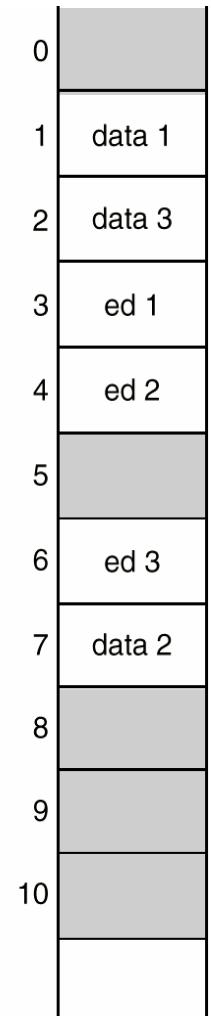
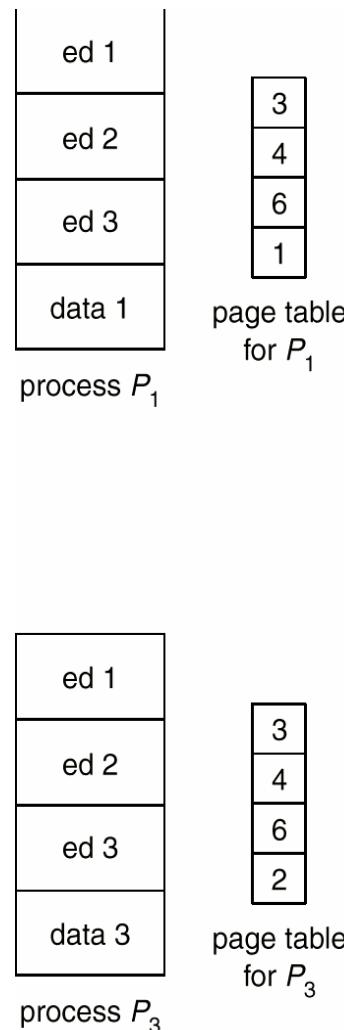
PAGING

SHARED PAGES

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

Useful for common code - must be write protected. (NO writeable data mixed with code.)

Extremely useful for read/write communication between processes.

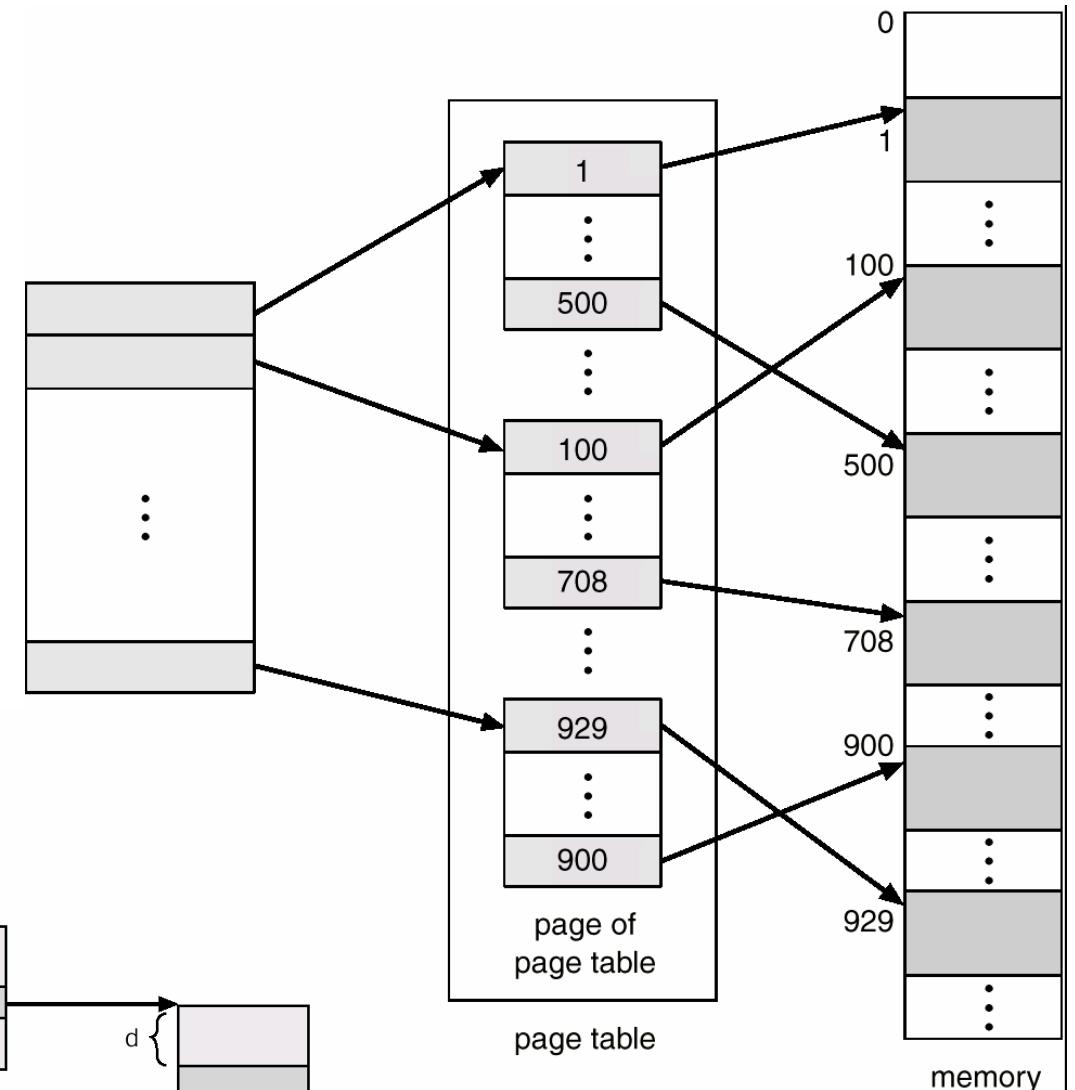
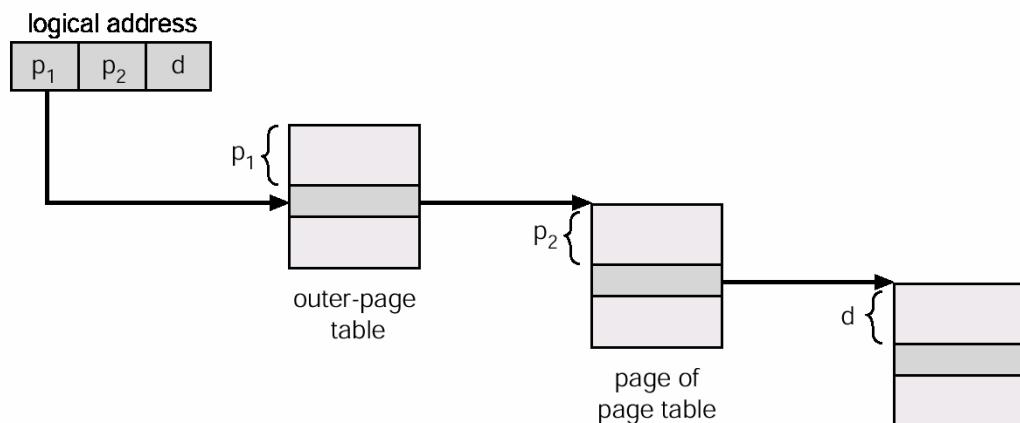


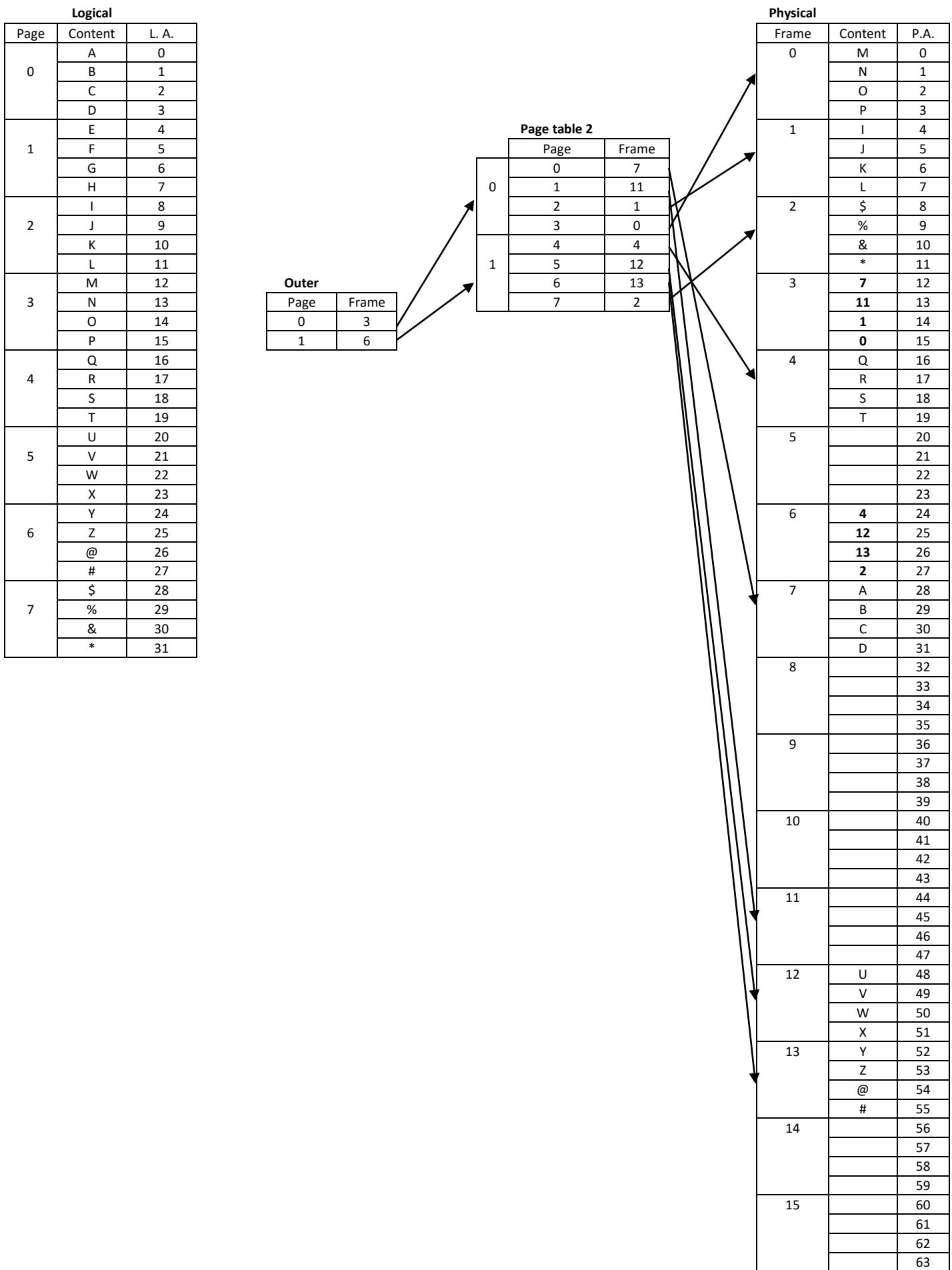
MEMORY MANAGEMENT

PAGING

MULTILEVEL PAGE TABLE

A means of using page tables for large address spaces.





MEMORY MANAGEMENT

PAGING

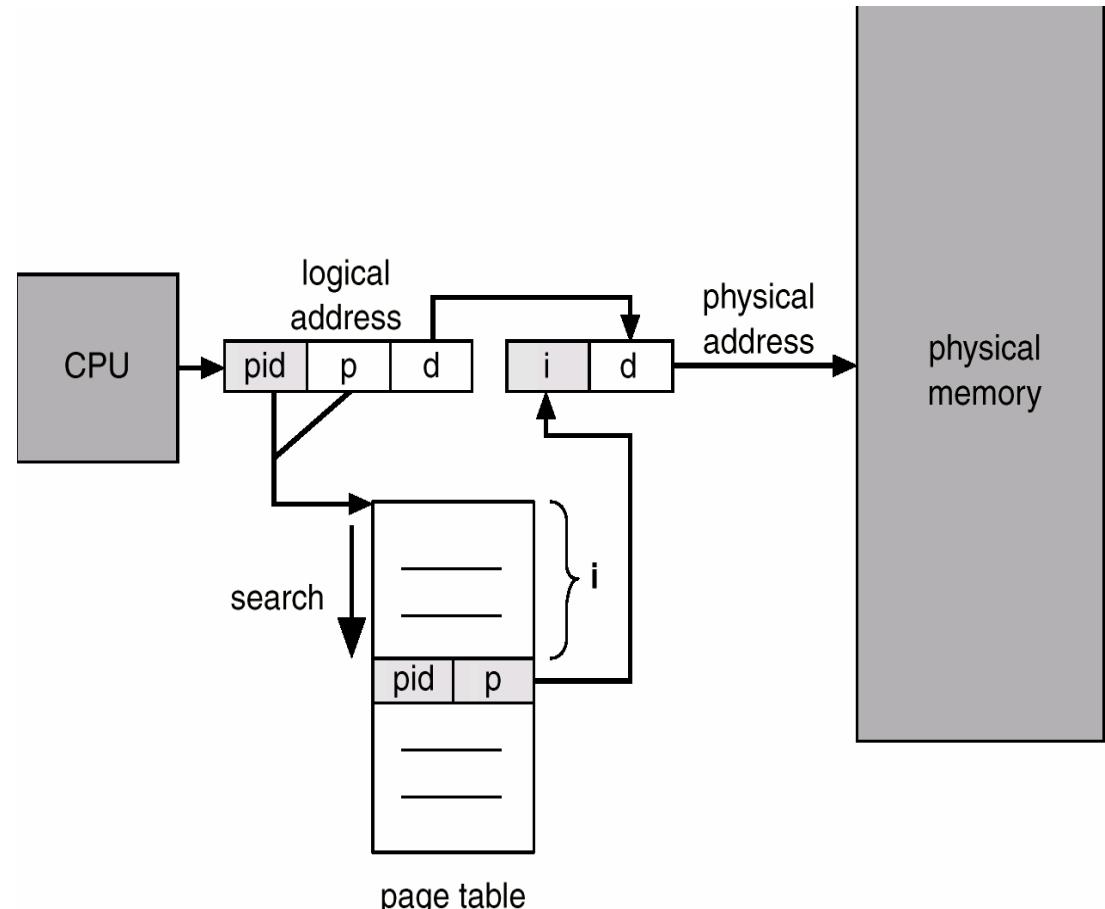
INVERTED PAGE TABLE:

One entry for each real page of memory.

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

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

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





Example:

A process of size 2 GB with:

Page size = 512 Bytes

Size of page table entry = 4 Bytes, then

Calculate the size of the page table.

Calculate the size of the inverted page table if the main memory size is 128 MB and each entry in the inverted page table is of size 4 Bytes.

Answer:

$$\text{Number of pages in the process} = 2 \text{ GB} / 512 \text{ Bytes} = 2^{31}/2^9 = 2^{22}$$

Thus, 2^{22} entries would be there in a normal page table.

$$\text{Page table size} = 2^{22} * 2^2 = 2^{24} \text{ Bytes}$$

$$\text{Number of entries in an inverted page table} = 128 \text{ MB} / 512 \text{ Bytes} = 2^{27} / 2^9 = 2^{18}$$

$$\text{Inverted page table size} = 2^{18} * 2^2 = 2^{20} \text{ Bytes}$$

MEMORY MANAGEMENT

Segmentation

USER'S VIEW OF MEMORY

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

Typical segments include

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

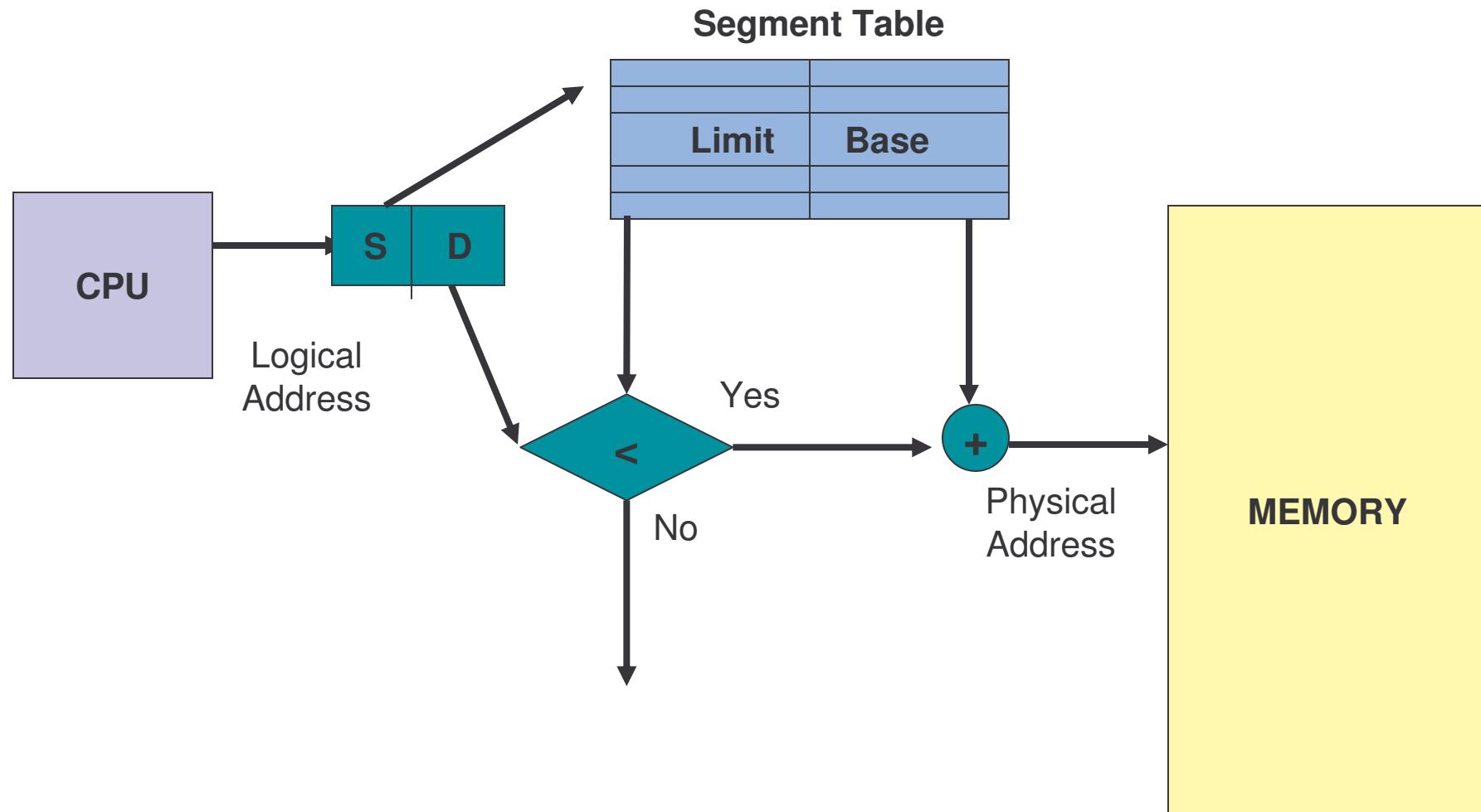
Logical address = segment name (number) + offset

Memory is addressed by both segment and offset.

MEMORY MANAGEMENT

Segmentation

HARDWARE -- Must map a dyad (segment / offset) into one-dimensional address.

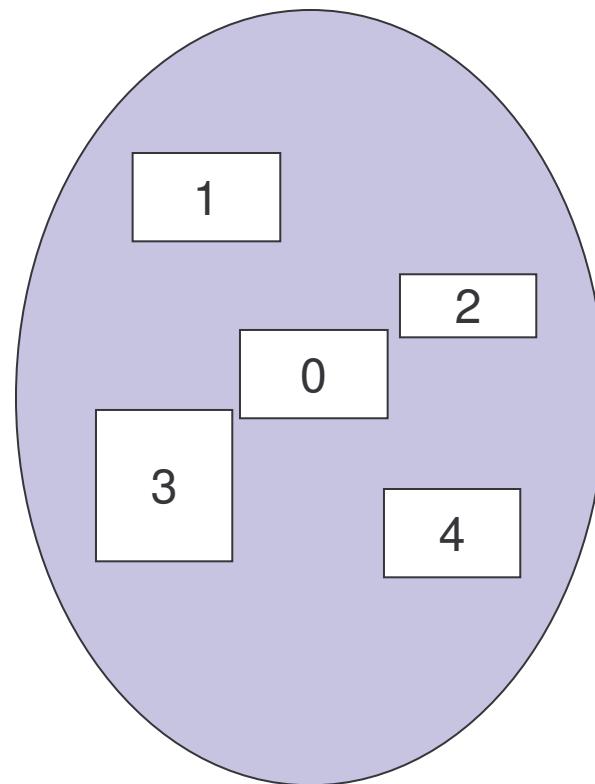


MEMORY MANAGEMENT

Segmentation

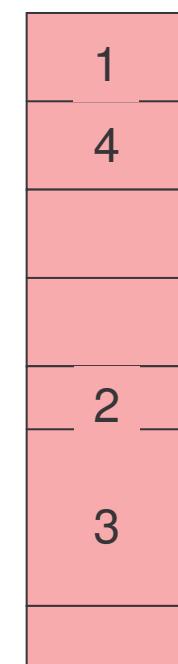
HARDWARE

base / limit pairs in a segment table.



Logical Address Space

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



Physical Memory

MEMORY MANAGEMENT

PROTECTION AND SHARING

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

Can do bounds checking on arrays

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

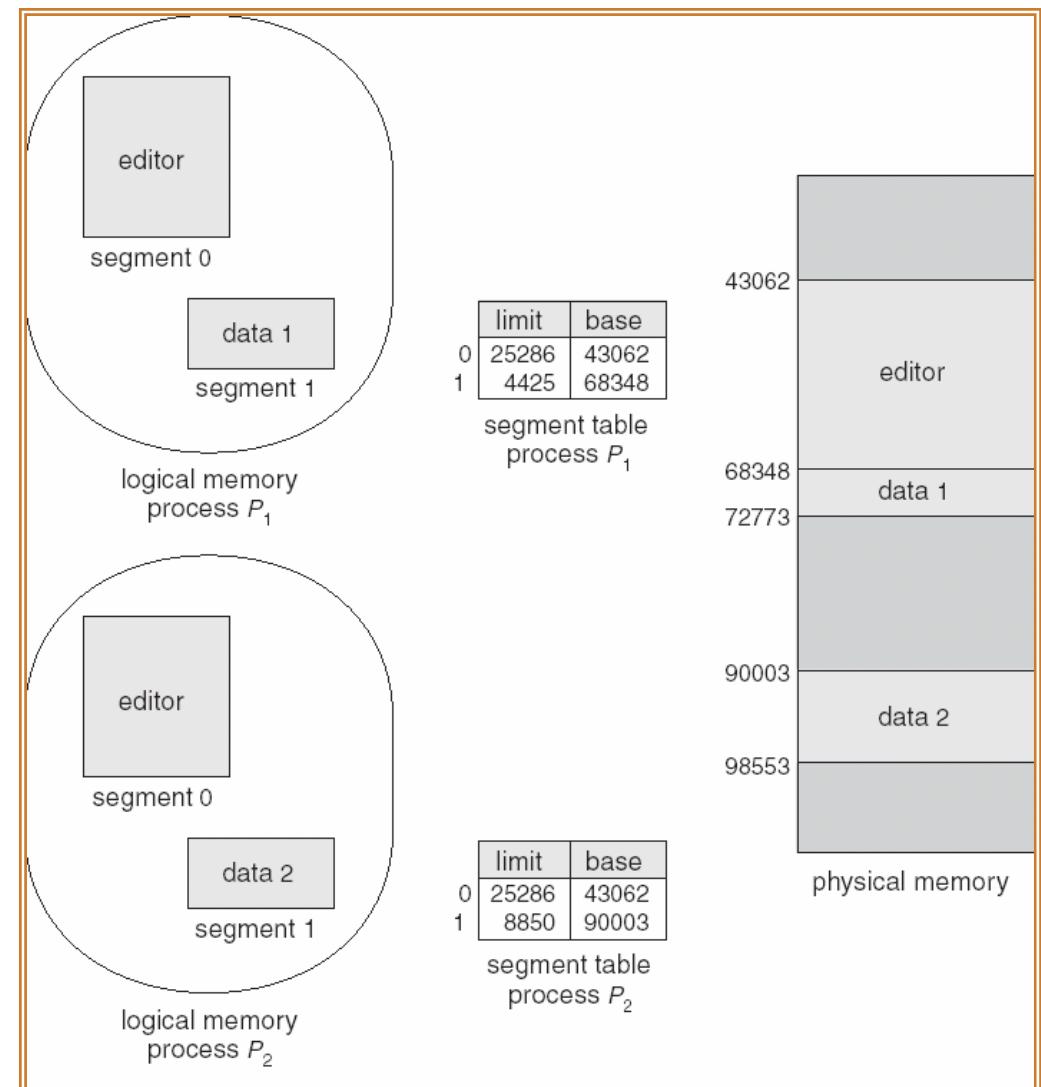
Can share some code but not all - for instance a common library of subroutines.

FRAGMENTATION

Uses variable allocation since segment lengths vary.

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

Segmentation



Consider the following segment table-

Segment No.	Base	Length
0	1219	700
1	2300	14
2	90	100
3	1327	580
4	1952	96

Which of the following logical address will produce trap addressing error?

Calculate the physical address if no trap is produced.

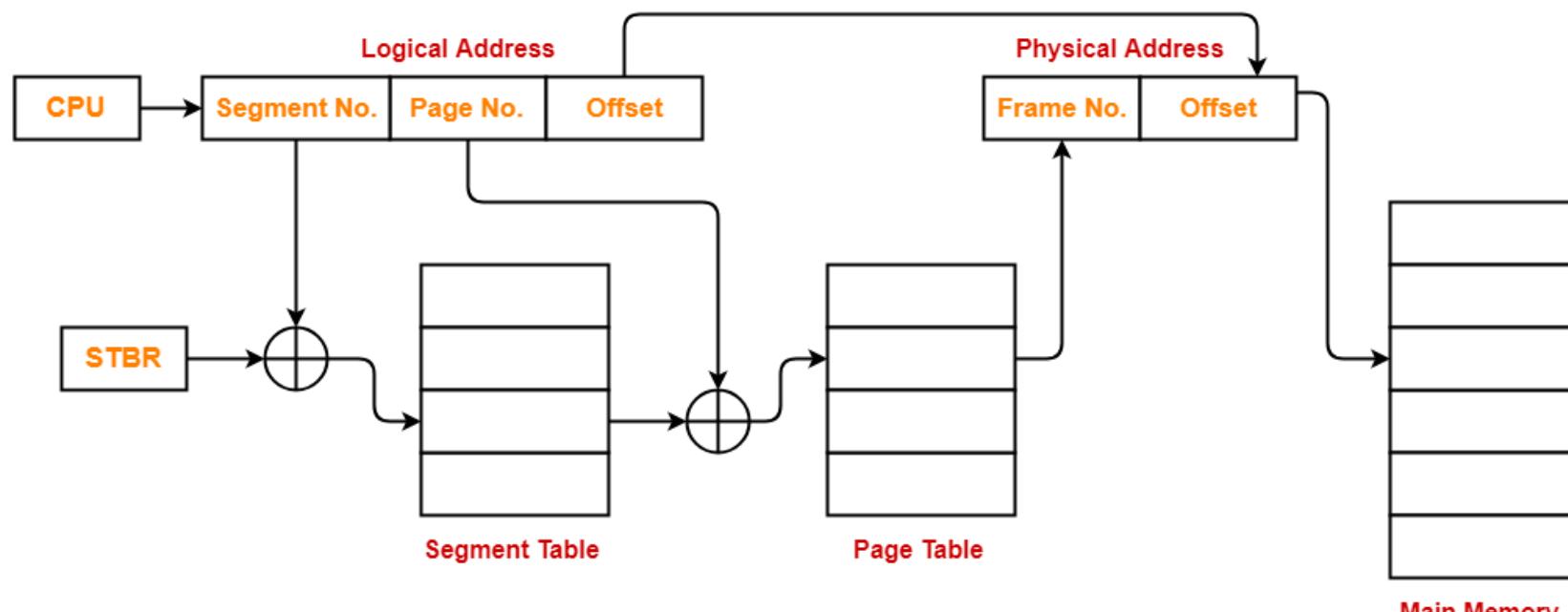
1. 0, 430
2. 1, 11
3. 2, 100
4. 3, 425
5. 4, 95

Answer:

1. Offset = 430 < 700 => No trap. Physical Address = 1219(Base) + 430(offset) = 1649
2. Offset = 1 < 11 => No trap. Physical Address = 2300(Base) + 11(offset) = 2311
3. Offset = 100 $\not\prec$ 100 => Trap error
4. Offset = 425 < 580 => No trap. Physical Address = 1327(Base) + 425(offset) = 1752
5. Offset = 95 < 96 => No trap. Physical Address = 1952(Base) + 95(offset) = 2047

Paged Segmentation

- Process is first divided into segments and then each segment is divided into pages.
- These pages are then stored in the frames of main memory.
- A page table exists for each segment that keeps track of the frames storing the pages of that segment.
- Each page table occupies one frame in the main memory.
- Number of entries in the page table of a segment = Number of pages that segment is divided.
- A segment table exists that keeps track of the frames storing the page tables of segments.
- Number of entries in the segment table of a process = Number of segments that process is divided.
- The base address of the segment table is stored in the segment table base register.



Translating Logical Address into Physical Address

Translating Logical Address to Physical Address

CPU generates a logical address consisting of three parts: < Segment Number, Page Number, Page Offset> or <s, p, o>

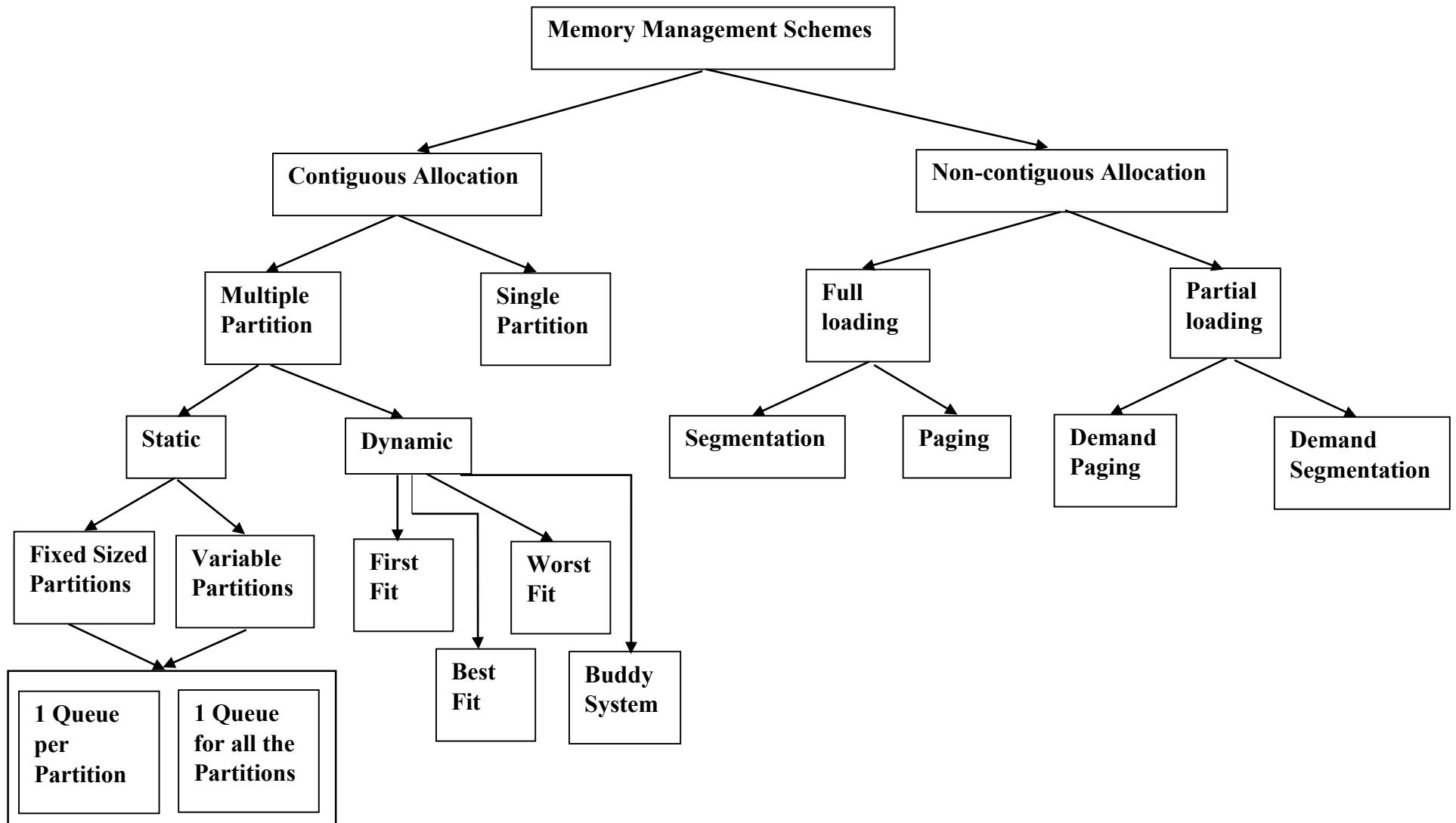
1. Segment Number specifies the specific segment from which CPU wants to reads the data.
2. Page Number specifies the specific page of that segment from which CPU wants to read the data.
3. Page Offset specifies the specific word on that page that CPU wants to read.
4. For the generated segment number, corresponding entry is located in the segment table.
5. Segment table provides the frame number of the frame storing the page table of the referred segment.
6. The frame containing the page table is located.
7. For the generated page number, corresponding entry is located in the page table.
8. Page table provides the frame number of the frame storing the required page of the referred segment.
9. The frame containing the required page is located.
10. The frame number combined with the page offset forms the required physical address.
11. For the generated page offset, corresponding word is located in the page and read.

MEMORY MANAGEMENT

WRAPUP

We've looked at how to do paging - associating logical with physical memory.

This subject is at the very heart of what every operating system must do today.



Consider the following segment table-

Segment No.	Base	Length
0	1219	700
1	2300	14
2	90	100
3	1327	580
4	1952	96

Which of the following logical address will produce trap addressing error?

Calculate the physical address if no trap is produced.

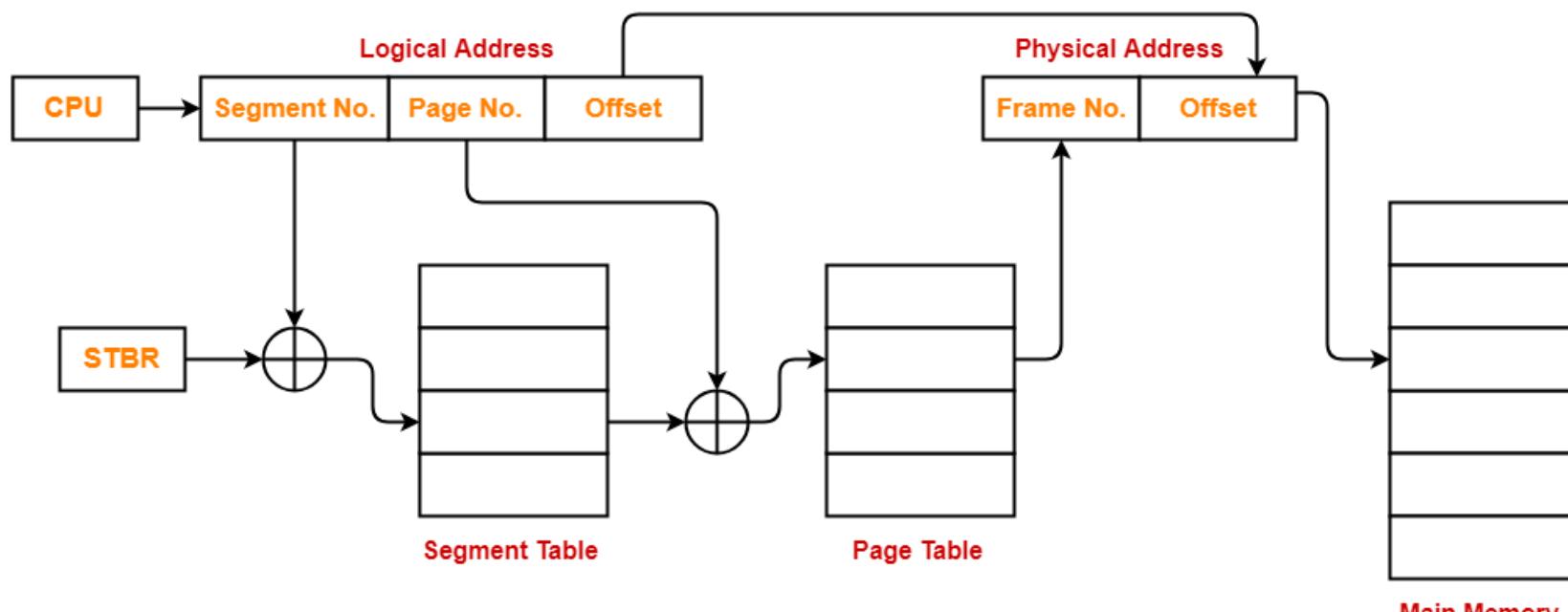
1. 0, 430
2. 1, 11
3. 2, 100
4. 3, 425
5. 4, 95

Answer:

1. Offset = 430 < 700 => No trap. Physical Address = 1219(Base) + 430(offset) = 1649
2. Offset = 1 < 11 => No trap. Physical Address = 2300(Base) + 11(offset) = 2311
3. Offset = 100 $\not\prec$ 100 => Trap error
4. Offset = 425 < 580 => No trap. Physical Address = 1327(Base) + 425(offset) = 1752
5. Offset = 95 < 96 => No trap. Physical Address = 1952(Base) + 95(offset) = 2047

Paged Segmentation

- Process is first divided into segments and then each segment is divided into pages.
- These pages are then stored in the frames of main memory.
- A page table exists for each segment that keeps track of the frames storing the pages of that segment.
- Each page table occupies one frame in the main memory.
- Number of entries in the page table of a segment = Number of pages that segment is divided.
- A segment table exists that keeps track of the frames storing the page tables of segments.
- Number of entries in the segment table of a process = Number of segments that process is divided.
- The base address of the segment table is stored in the segment table base register.



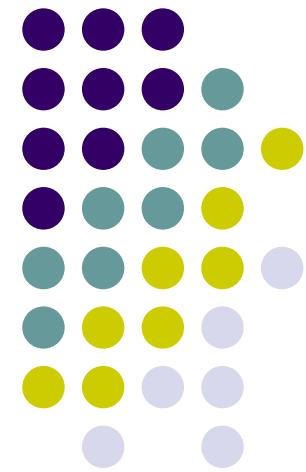
Translating Logical Address into Physical Address

Translating Logical Address to Physical Address

CPU generates a logical address consisting of three parts: < Segment Number, Page Number, Page Offset> or <s, p, o>

1. Segment Number specifies the specific segment from which CPU wants to reads the data.
2. Page Number specifies the specific page of that segment from which CPU wants to read the data.
3. Page Offset specifies the specific word on that page that CPU wants to read.
4. For the generated segment number, corresponding entry is located in the segment table.
5. Segment table provides the frame number of the frame storing the page table of the referred segment.
6. The frame containing the page table is located.
7. For the generated page number, corresponding entry is located in the page table.
8. Page table provides the frame number of the frame storing the required page of the referred segment.
9. The frame containing the required page is located.
10. The frame number combined with the page offset forms the required physical address.
11. For the generated page offset, corresponding word is located in the page and read.

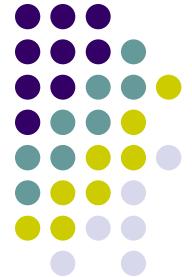
Virtual Memory



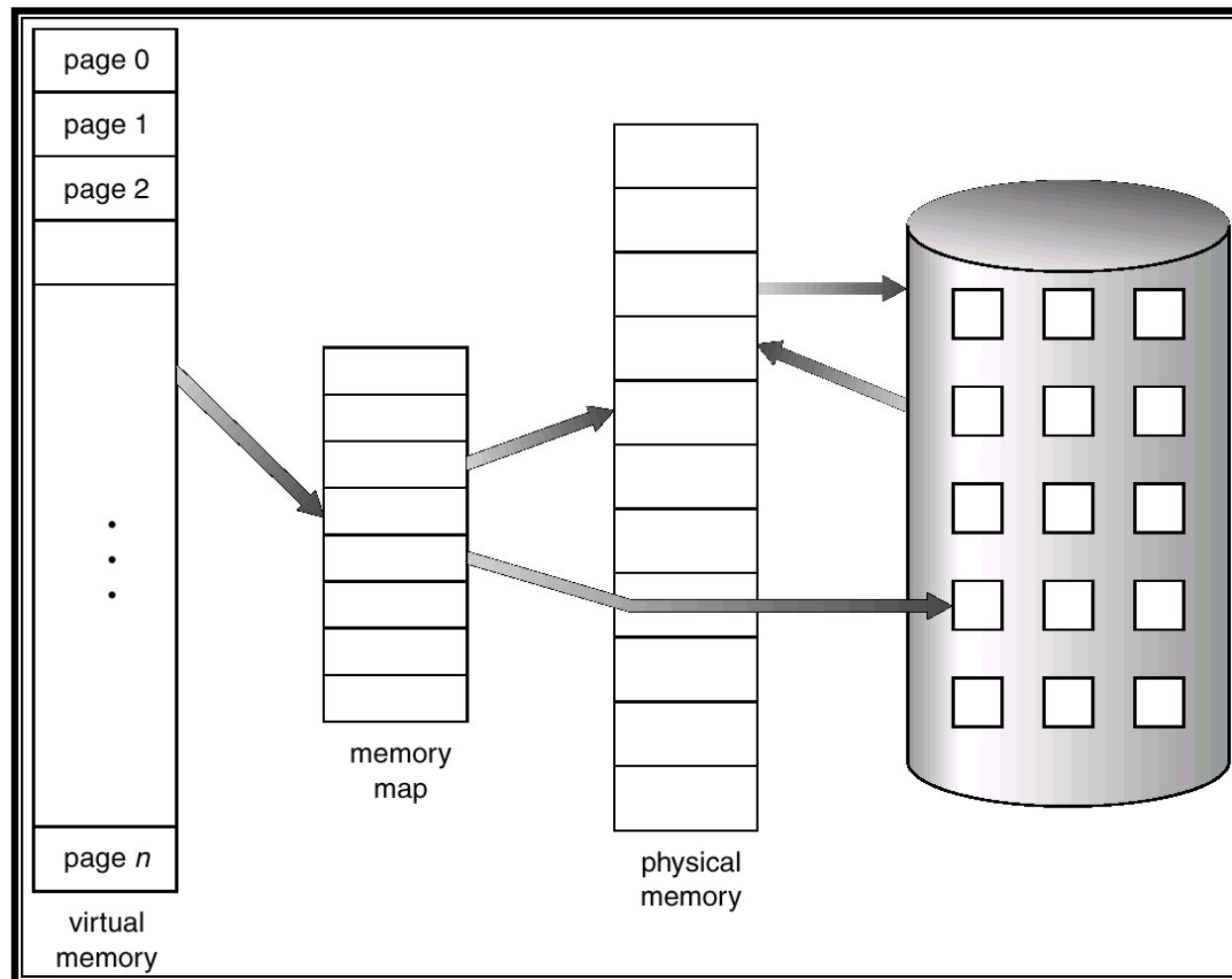


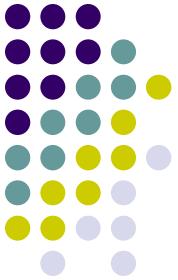
Basic Concept

- Usually, only part of the program needs to be in memory for execution
- Allow logical address space to be larger than physical memory size
- Bring only what is needed in memory when it is needed
- Virtual memory implementation
 - Demand paging
 - Demand segmentation



Virtual Memory That is Larger Than Physical Memory

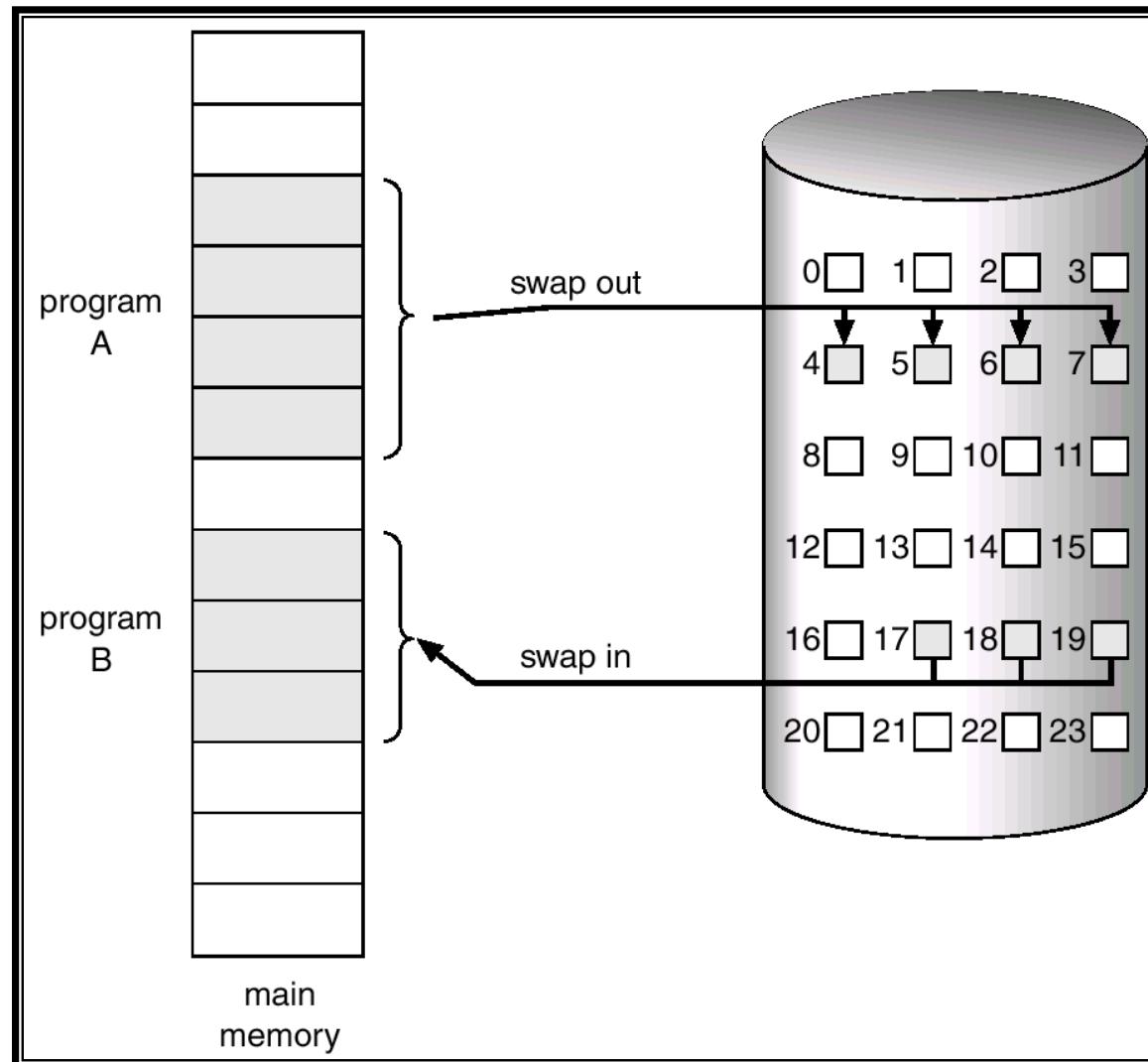




Demand Paging

- Bring a page into memory only when it is needed (**on demand**)
 - Less I/O needed to start a process
 - Less memory needed
 - Faster response
 - More users
- Page is needed \Rightarrow reference to it
 - invalid reference \Rightarrow abort
 - not-in-memory \Rightarrow bring to memory

Transfer of a Paged Memory to Contiguous Disk Space





Some questions

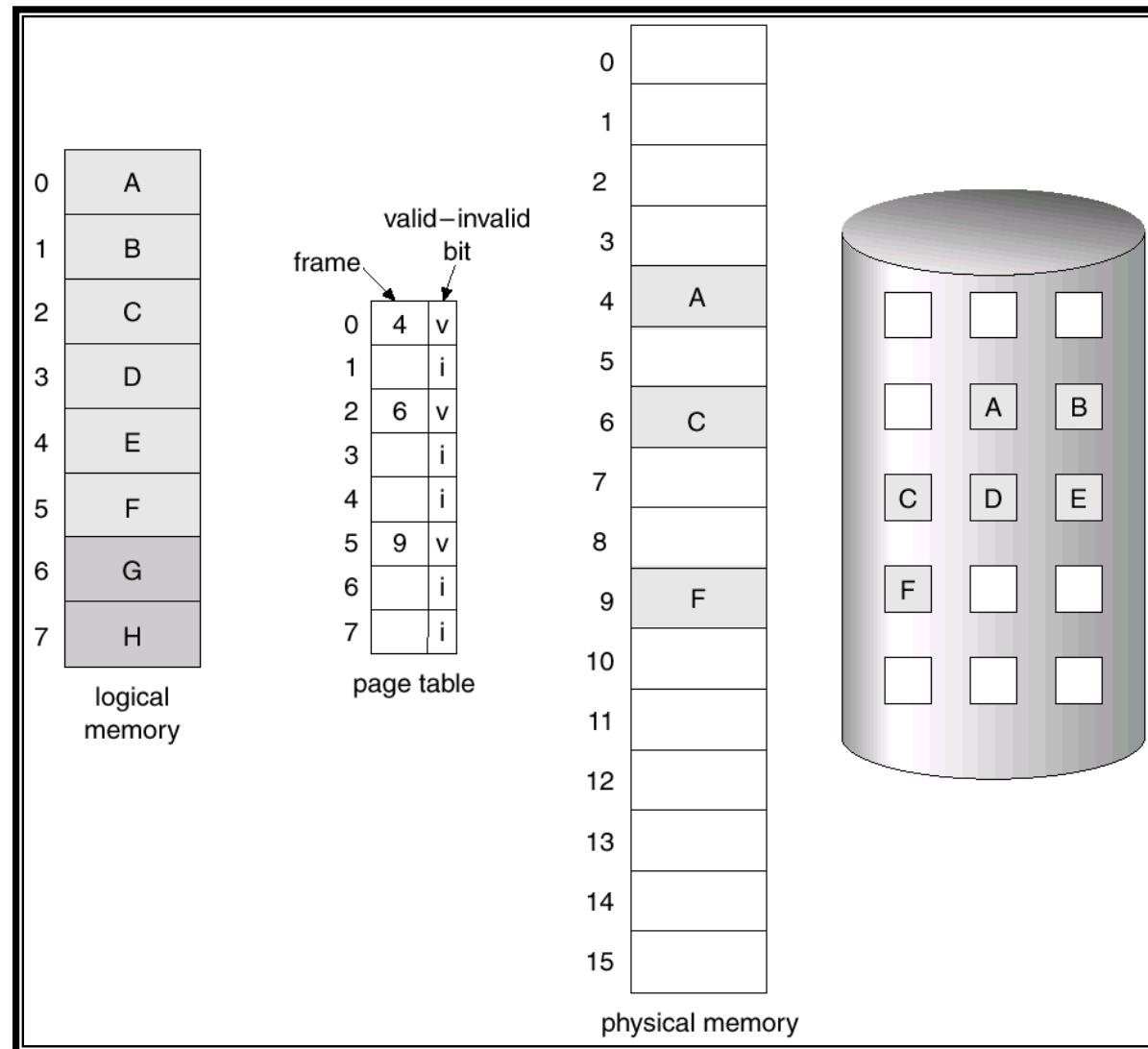
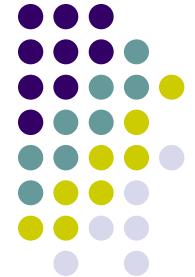
- How to know if a page is in memory?
 - Valid bit
- If not present, what happens during reference to a logical address in that page?
 - Page fault
- If not present, where in disk is it?
- If a new page is brought to memory, where should it be placed?
 - Any free page frame
- What if there is no free page frame?
 - Page replacement policies
- Is it always necessary to copy a page back to disk on replacement?
 - Dirty/Modified bit

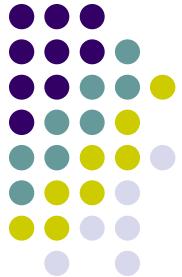


Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
(1 \Rightarrow in-memory, 0 \Rightarrow not-in-memory)
- Initially valid–invalid bit is set to 0 on all entries
- Set to 1 when memory is allocated for a page
(page brought into memory)
- Address translation steps:
 - Use page no. in address to index into page table
 - Check valid bit
 - If set, get page frame start address, add offset to get memory address, access memory
 - If not set, page fault

Page Table When Some Pages Are Not in Main Memory



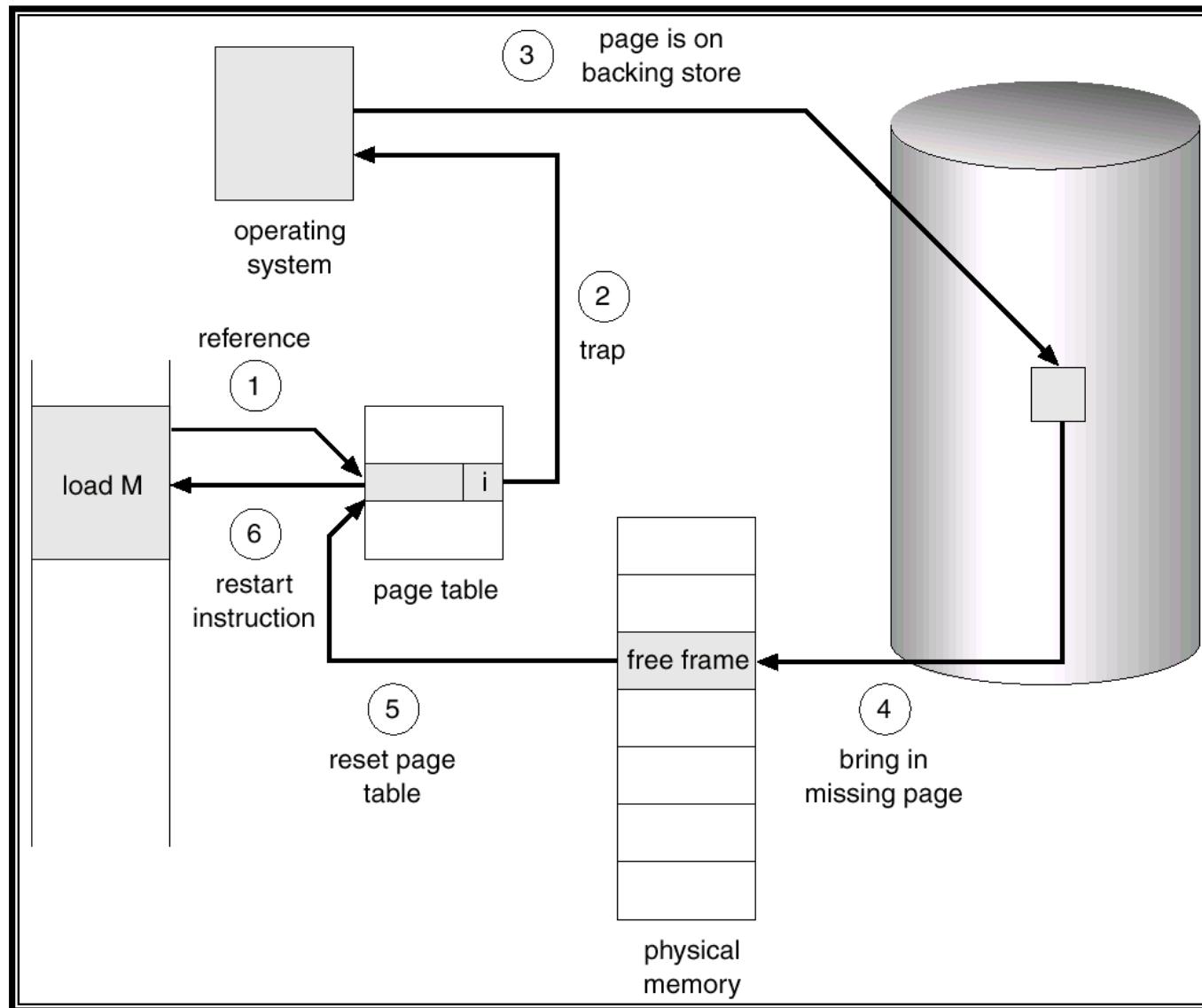


Page Fault

- If there is ever a reference to a page, first reference will trap to OS \Rightarrow page fault
- OS looks at another table to decide:
 - Invalid reference \Rightarrow abort
 - Just not in memory, get from disk
- Get empty frame
- Get disk address of page (in PTE)
- Swap page into frame from disk (context switch for I/O wait)
- Modify PTE entry for page with frame no., set valid bit = 1.
- Restart instruction: Least Recently Used
 - block move
 - auto increment/decrement location



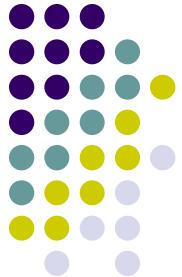
Steps in Handling a Page Fault



Performance of Demand Paging



- Page Fault Rate $0 \leq p \leq 1.0$
 - if $p = 0$ no page faults
 - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)
$$\text{EAT} = (1 - p) \times \text{memory access} + p (\text{page fault overhead} + [\text{swap page out}] + \text{swap page in} + \text{restart overhead})$$



Demand Paging Example

- Memory access time = 150 ns
- 50% of the time the page that is being replaced has been modified and therefore needs to be swapped out.
- Swap Page Time = 10 msec = 10^7 ns
$$EAT = (1 - p) \times 150 + p (1.5 \times 10^7)$$
$$\approx 150 + p (1.5 \times 10^7) \text{ ns}$$
- EAT = 200 ns will need $p = 0.0000033$!
- EAT = 165 ns (10% loss) will need $p = 0.000001$, or 1 fault in 1000000 accesses



Page in Disk

- Swap space: part of disk divided in page sized slots
 - First slot has swap space management info such as free slots etc.
- Pages can be swapped out from memory to a free page slot
 - Address of page <swap partition no., page slot no.>
 - Address stored in PTE
- Read-only pages can be read from the file system directly using memory-mapped files



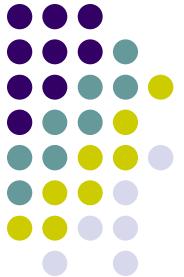
What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
 - algorithm
 - performance – want an algorithm which will result in minimum number of page faults.
- Same page may be brought into memory several times

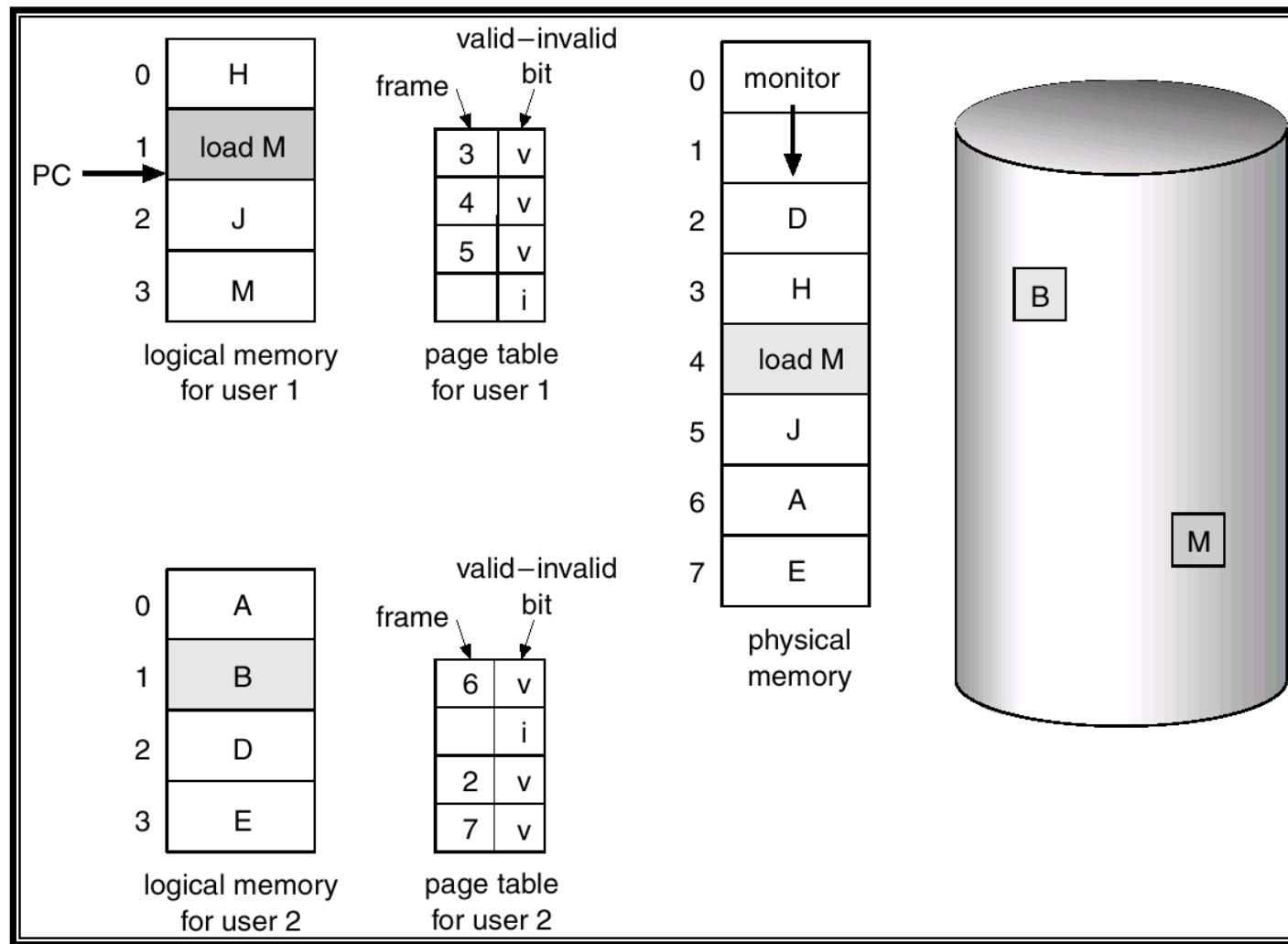


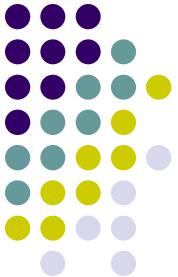
Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use *modified (dirty) bit* to reduce overhead of page transfers
 - Set to 0 when a page is brought into memory
 - Set to 1 when some location in a page is changed
 - Copy page to disk only if bit set to 1
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory



Need For Page Replacement



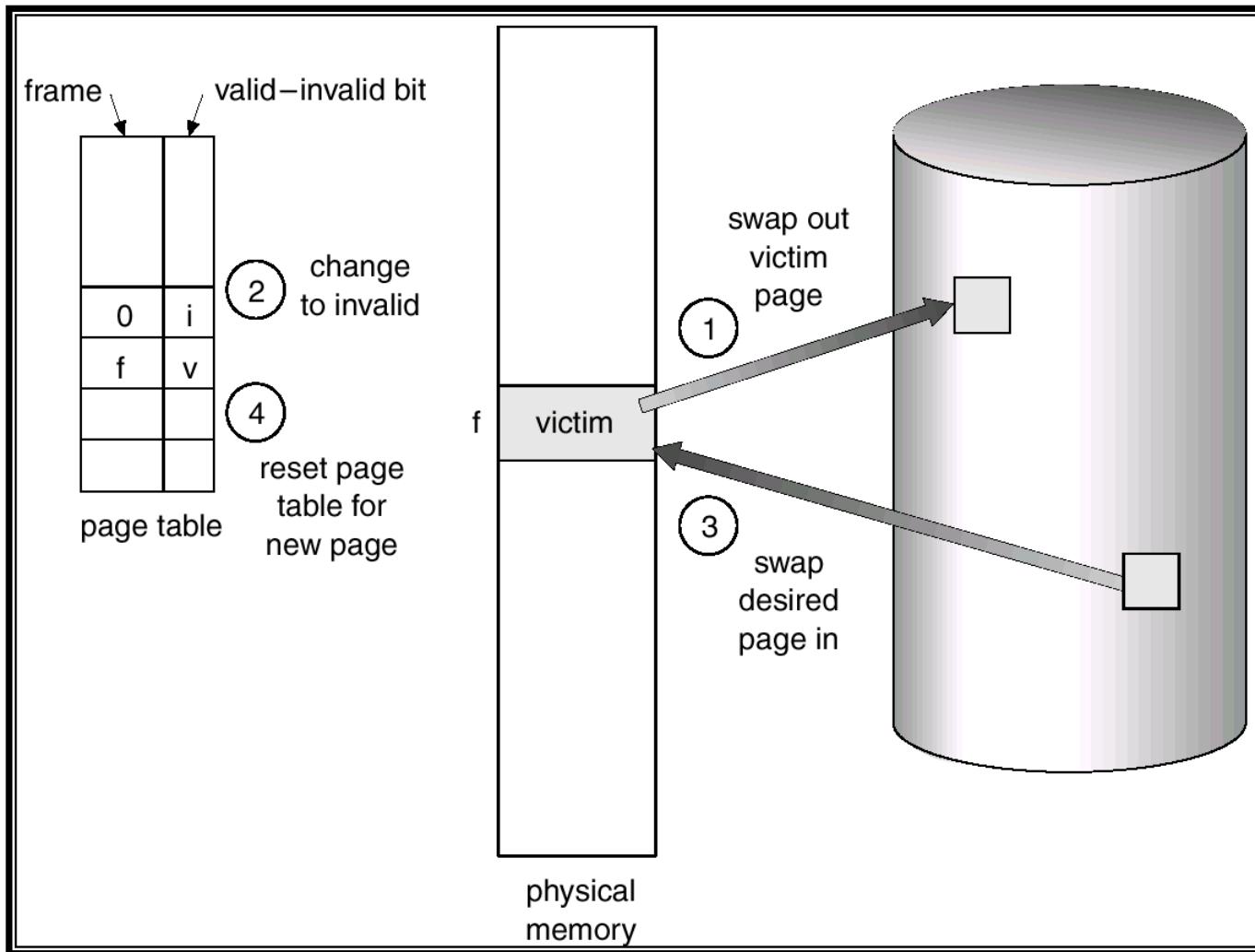


Basic Page Replacement

- Find the location of the desired page on disk
- Find a free frame
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to select a *victim* frame
- Copy the to-be replaced frame to disk if needed
- Read the desired page into the (newly) free frame. Update the page and frame tables.
- Restart the process



Page Replacement

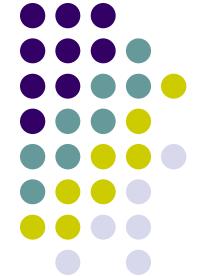




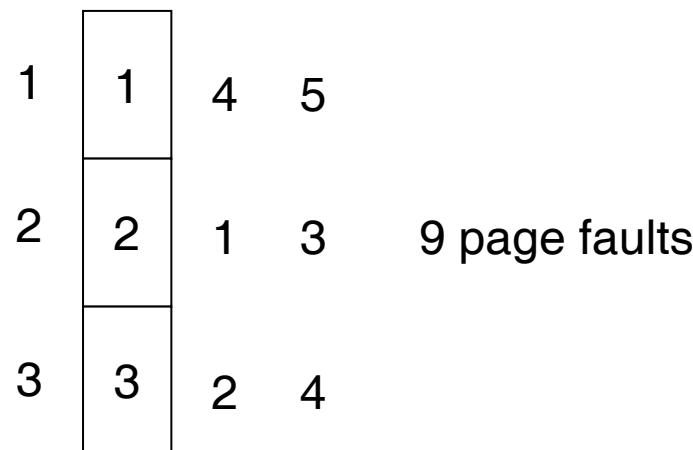
Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string of pages referenced) and computing the number of page faults on that string

First-In-First-Out (FIFO) Algorithm



- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)





FIFO Page Replacement

reference string

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

page frames



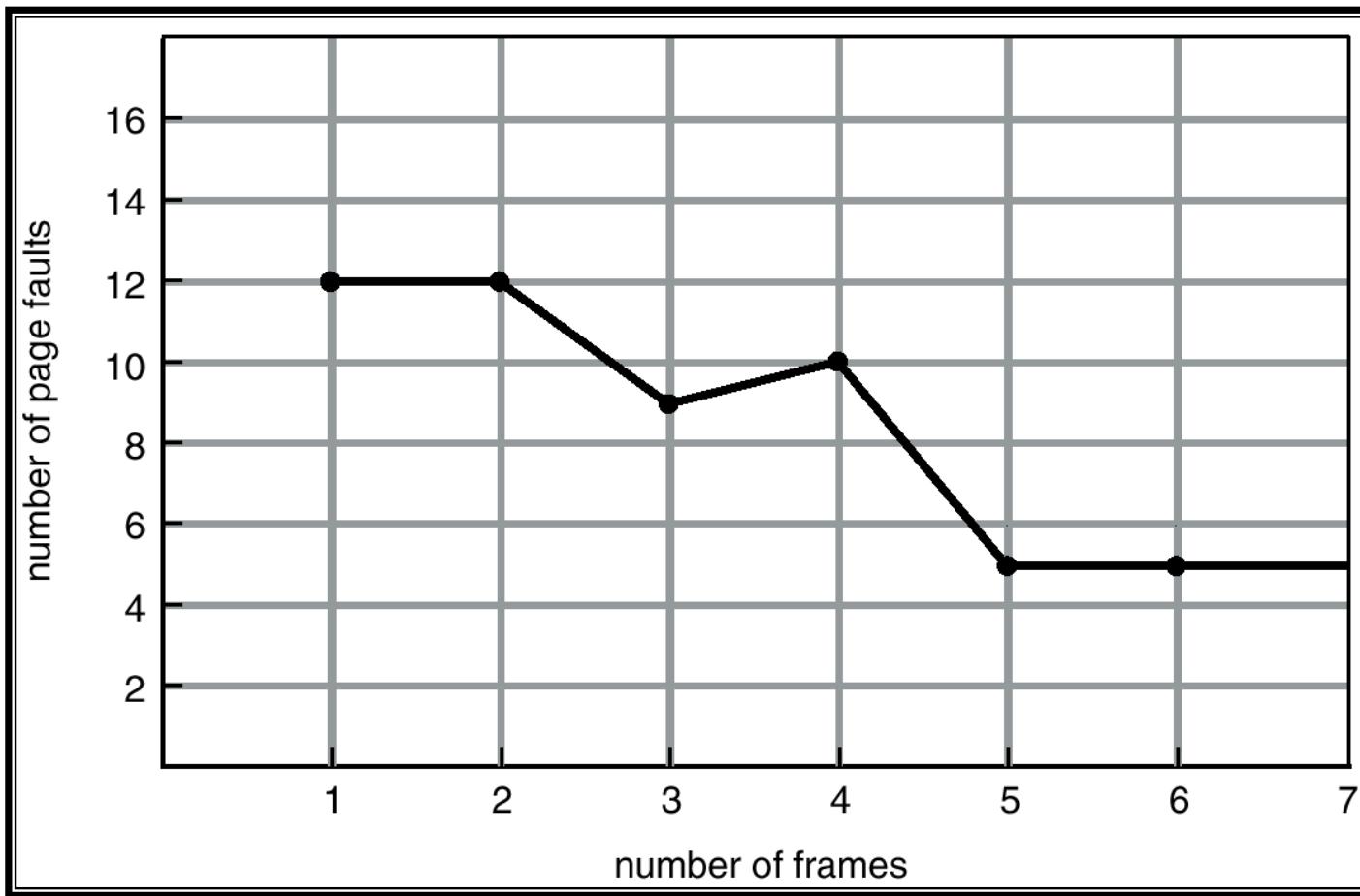
Belady's Anomaly

- The number of page faults may increase with increase in number of page frames for FIFO
 - Counter-intuitive
- Consider 4 page frames

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



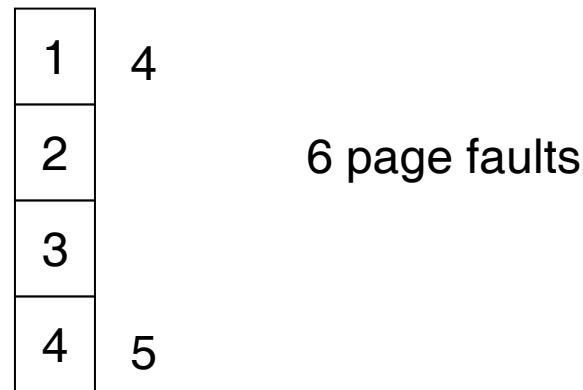
Belady's Anomaly



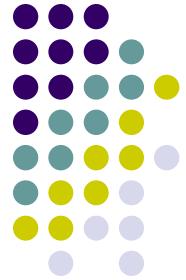


Optimal Algorithm

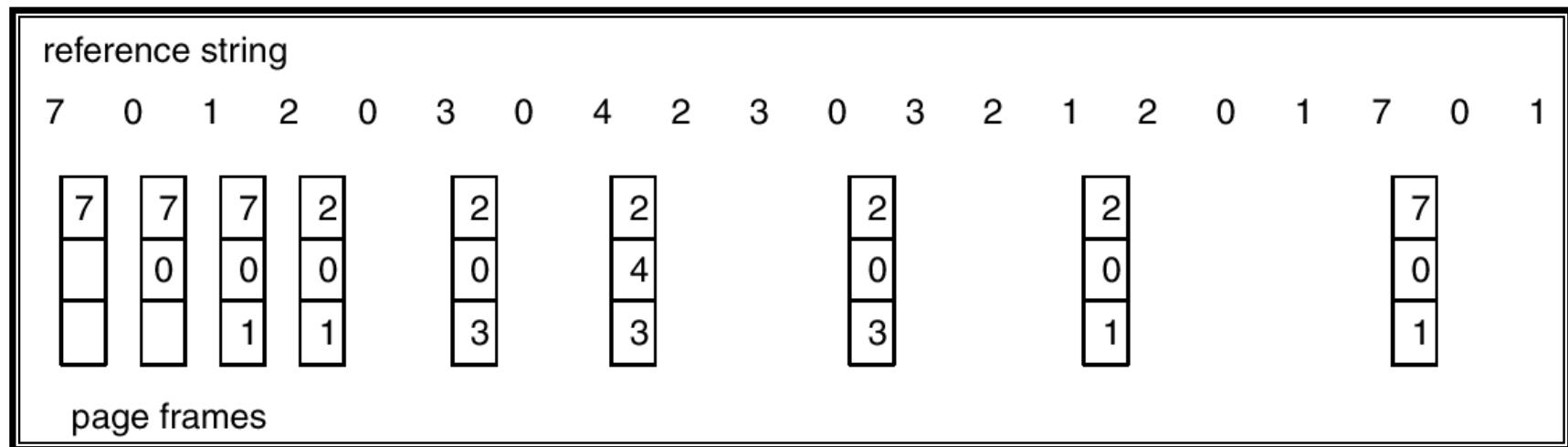
- Replace page that will not be used for longest period of time.
- 4 frames example: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



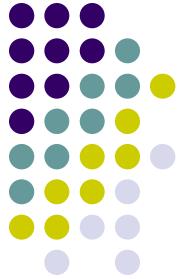
- How do you know this?
- Used for measuring how well your algorithm performs



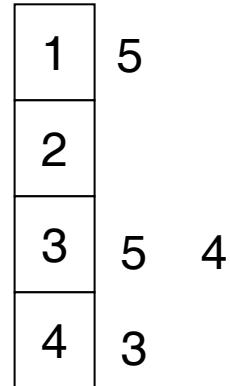
Optimal Page Replacement



Least Recently Used (LRU) Algorithm



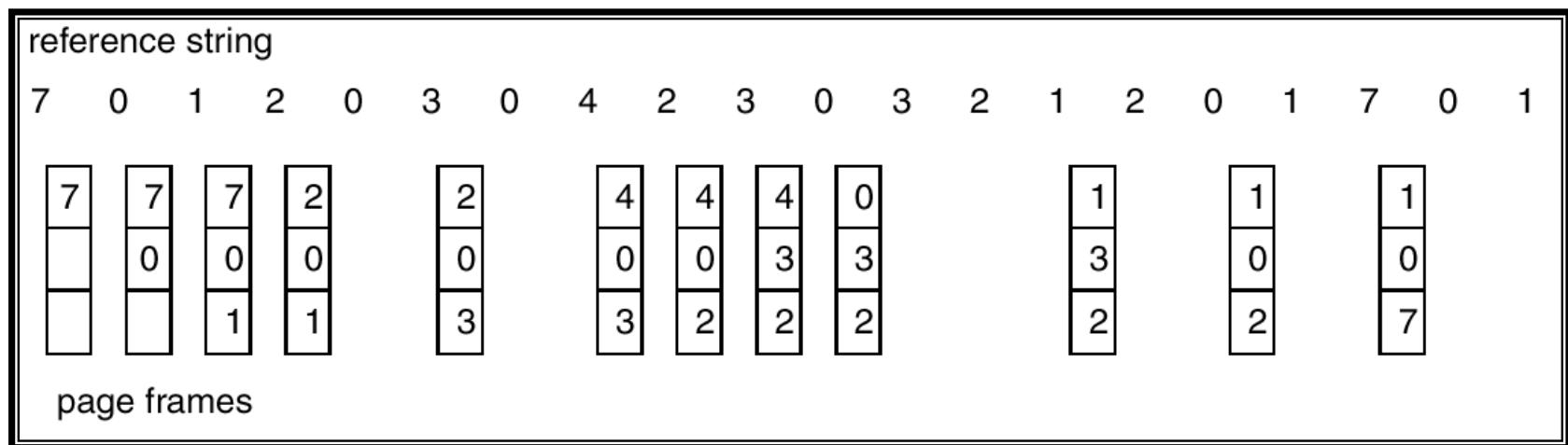
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

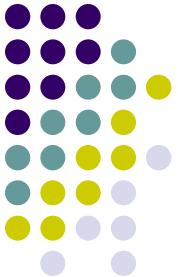


- Counter implementation
 - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
 - When a page needs to be changed, look at the counters to determine which are to change



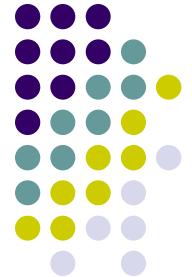
LRU Page Replacement



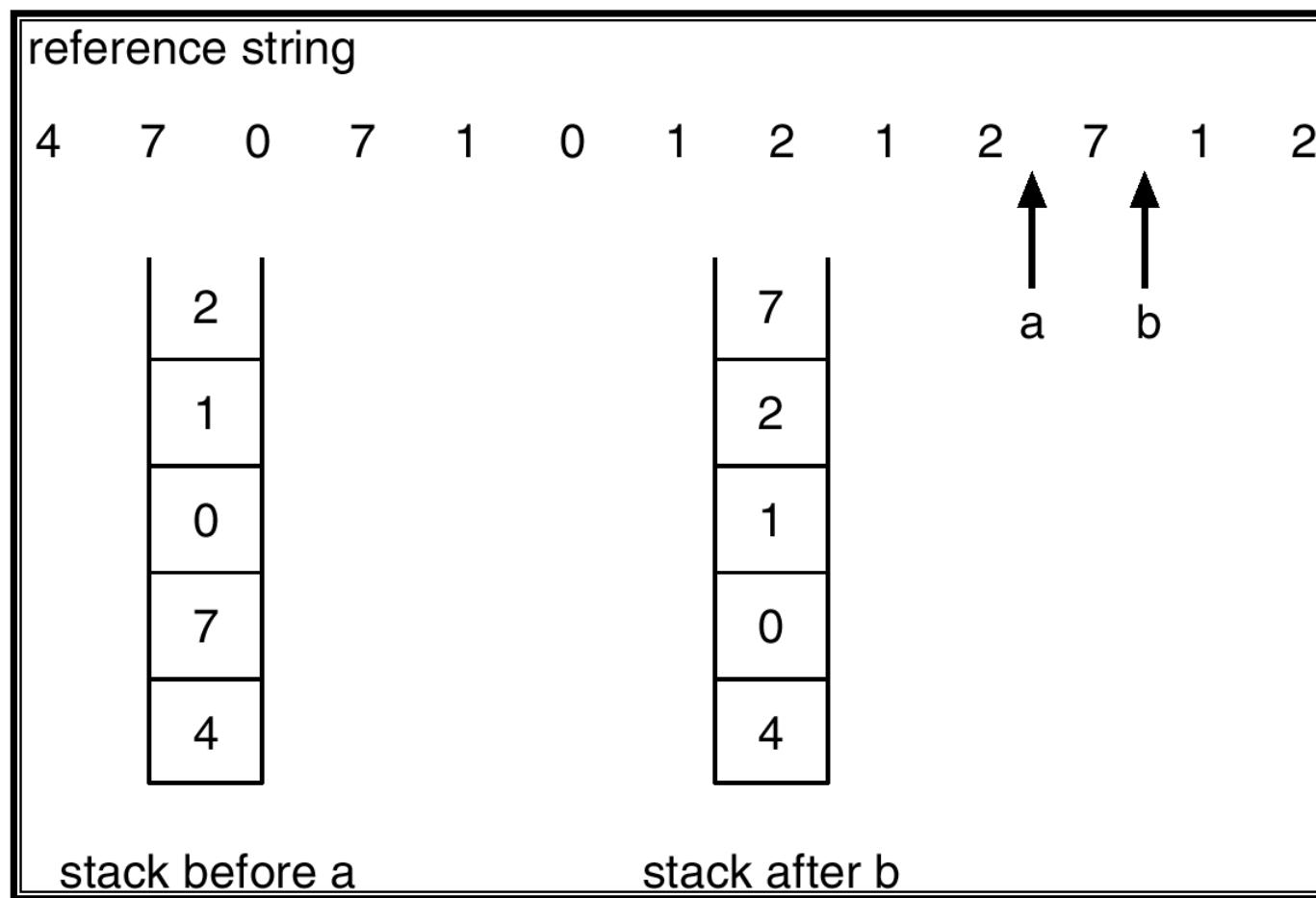


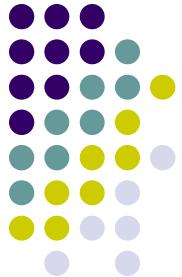
LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a double link form:
 - Page referenced:
 - move it to the top
 - requires 6 pointers to be changed
 - No search for replacement



Use Of A Stack to Record The Most Recent Page References





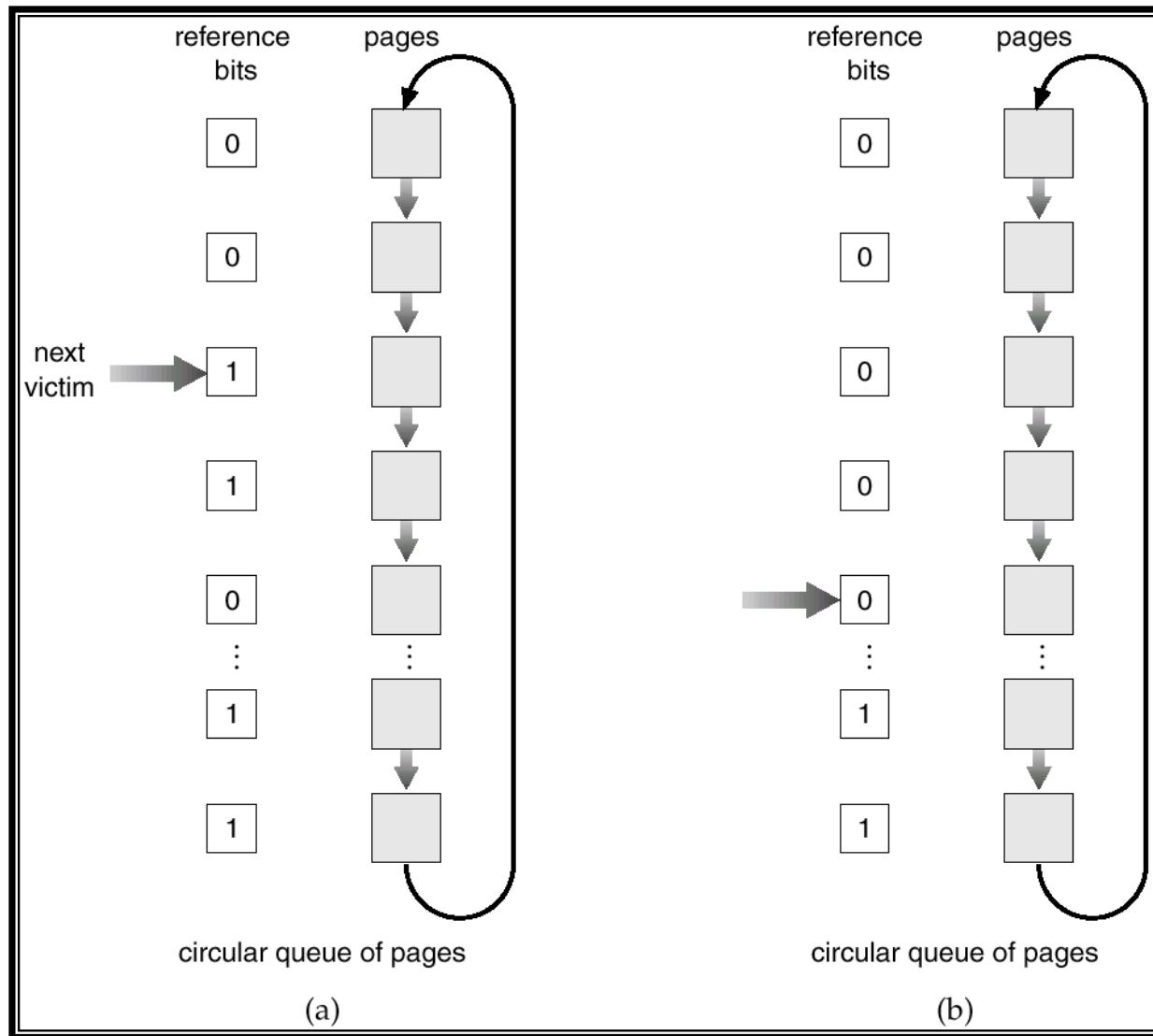
LRU Approximation Algorithms

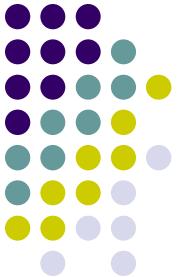
- Reference bit
 - 1 bit per page frame, initially = 0
 - When page frame is referenced, bit set to 1
 - Periodically reset to 0
 - Replace the one which is 0 (if one exists). We do not know the order, however
- Additional Reference Bits algorithm
 - K bits marked 1 to K from MSB
 - ith bit indicates if page accessed in the ith most recent interval
 - At every interval (timer interrupt), right shift the bits (LSB drops off), shift reference bit to MSB, and reset reference bit to 0
 - To replace, find the frame with the smallest value of the K bits



- Second chance
 - Need reference bit
 - Clock replacement
 - If page to be replaced (in clock order) has reference bit = 1, then:
 - set reference bit 0
 - leave page in memory
 - replace next page (in clock order), subject to same rules

Second-Chance (clock) Page-Replacement Algorithm





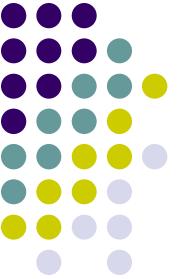
Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- LFU Algorithm: replaces page with smallest count
- MFU Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used



Allocation of Frames

- Each process needs some minimum number of pages
- No process should use up nearly all page frames
- Two major allocation schemes
 - fixed allocation
 - priority allocation



Fixed Allocation

- Equal allocation – e.g., if 100 frames and 5 processes, give each 20 pages.
- Proportional allocation – Allocate according to the size of process



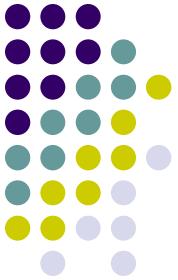
Priority Allocation

- Use a proportional allocation scheme using priorities rather than size.
- If process P_i generates a page fault,
 - select for replacement one of its frames
 - select for replacement a frame from a process with lower priority number



Global vs. Local Allocation

- Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another.
- Local replacement – each process selects from only its own set of allocated frames.



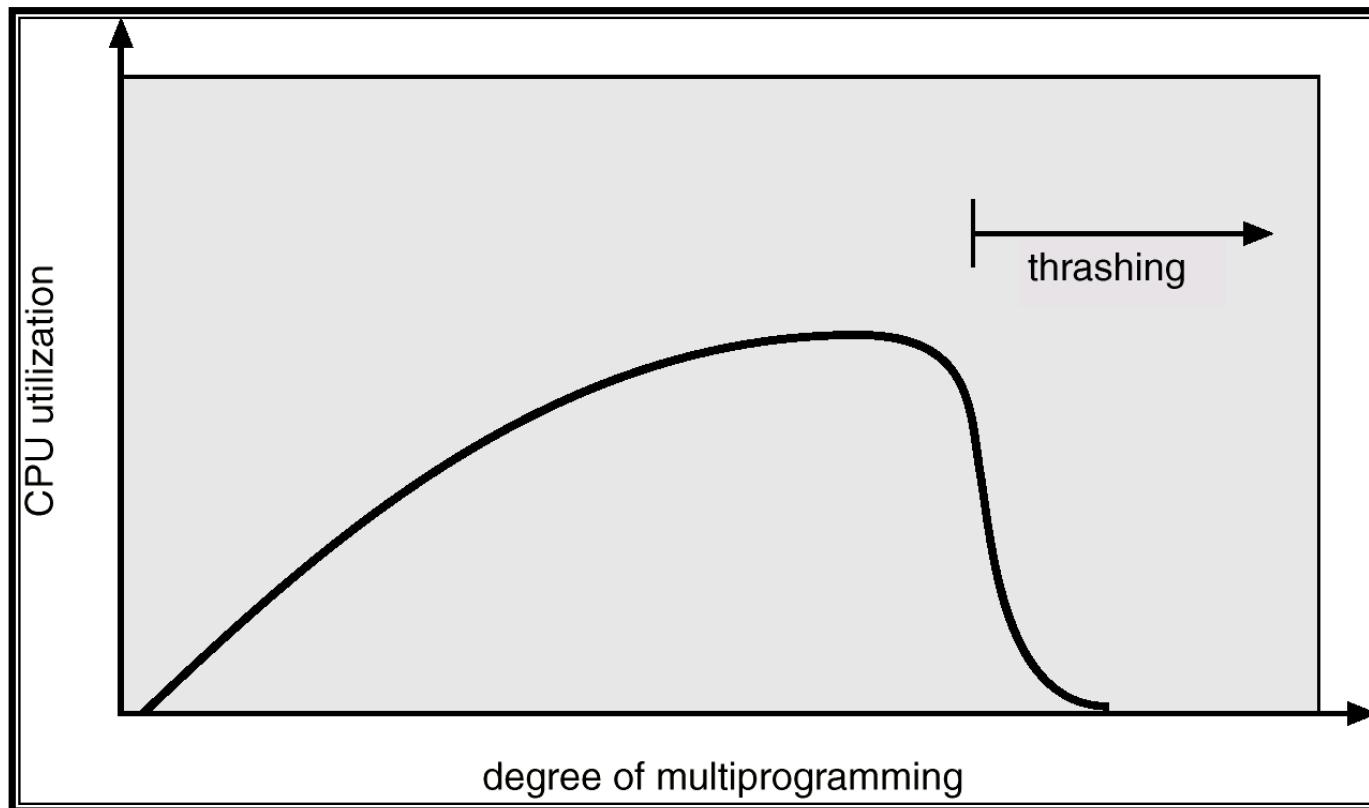
Why does paging work?

- Locality of reference
 - Processes tend to access locations which are close to each other ([spatial locality](#)) or which are accessed in the recent past ([temporal locality](#))
- Locality of reference implies once a set of page is brought in for a process, less chance of page faults by the process for some time
- Also, TLB hit ratio will be high
- [Working set](#) – the set of pages currently needed by a process
- Working set of a process changes over time
 - But remains same for some time due to locality of reference



Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
 - low CPU utilization.
 - operating system thinks that it needs to increase the degree of multiprogramming.
 - another process added to the system
 - Adds to the problem as even less page frames are available for each process
- **Thrashing** ≡ a process is busy swapping pages in and out

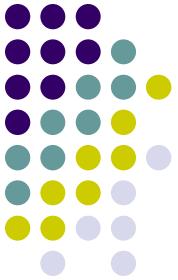


- Thrashing occurs when
$$\sum \text{working set of all processes} > \text{total memory size}$$

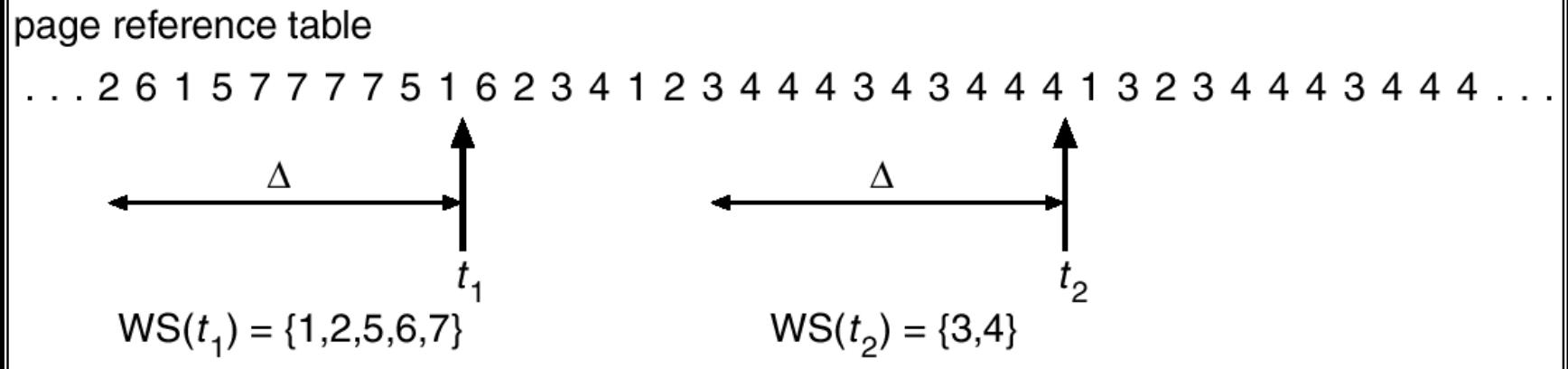


Working-Set Model

- $\Delta \equiv$ working-set window \equiv a fixed number of page references
Example: 10,000 instruction
- WSS_i (working set of Process P_i) =
total number of pages referenced in the most recent Δ
(varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum WSS_i \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend one of the processes.



Working-set model



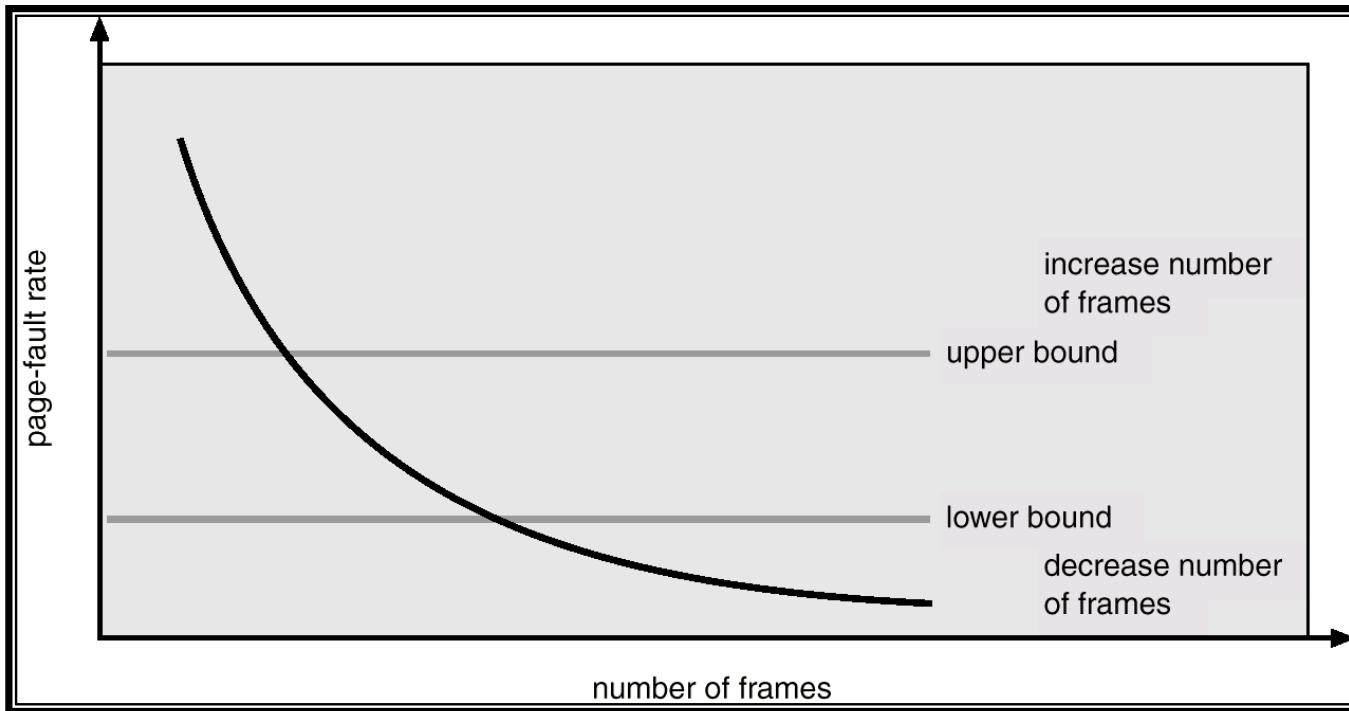
Keeping Track of the Working Set



- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
 - Timer interrupts after every 5000 time units
 - Keep in memory 2 bits for each page
 - Whenever a timer interrupts copy and sets the values of all reference bits to 0
 - If one of the bits in memory = 1 \Rightarrow page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units



Page-Fault Frequency Scheme



- Establish “acceptable” page-fault rate.
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame



What should the page size be?

- Reduce internal fragmentation
 - Smaller is better
- Reduce table size
 - Larger is better
- Reduce I/O overhead
 - Smaller is better
- Capture locality
 - Larger is better
- Need to be chosen judiciously



Other Considerations

- Prepaging
 - Bring in pages not referenced yet
- TLB Reach
 - The amount of memory accessible from the TLB.
 - $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
 - Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of page faults



Other Considerations (Cont.)

- Program structure

```
int A[][] = new int[1024][1024];
```

Each row is stored in one page

Program 1

```
for (j = 0; j < A.length; j++)
    for (i = 0; i < A.length; i++)
        A[i,j] = 0;
```

1024 x 1024 page faults!!

Program 2

```
for (i = 0; i < A.length; i++)
    for (j = 0; j < A.length; j++)
        A[i,j] = 0;
```

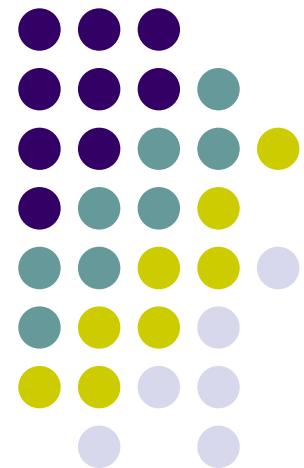
1024 page faults



Other Considerations (Cont.)

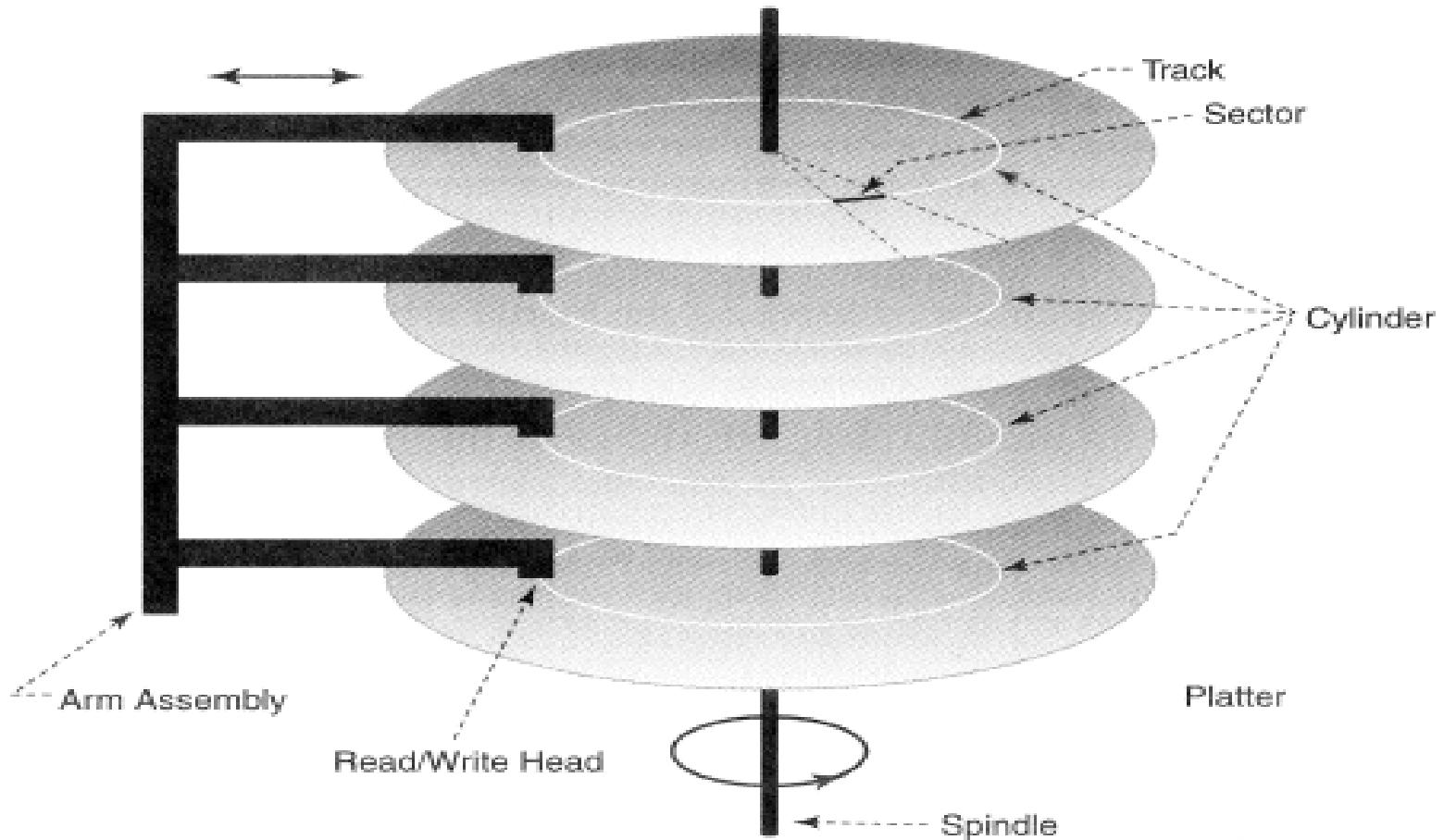
- I/O Interlock – Pages must sometimes be locked into memory
 - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
 - Some OS pages need to be in memory all the time
 - Use a lock bit to indicate if the page is locked and cannot be replaced

Disk Management





Physical Disk Structure





Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the logical block is the smallest unit of transfer
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially
 - Sector 0 is the first sector of the first track (top platter) on the outermost cylinder
 - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost



Disk Access Time

- Two major components
 - *Seek time* is the time for the disk to move the heads to the cylinder containing the desired sector
 - Typically 5-10 milliseconds
 - *Rotational latency* is the additional time waiting for the disk to rotate the desired sector to the disk head
 - Typically, 2-4 milliseconds
- One minor component
 - *Read/write time or transfer time* – actual time to transfer a block, less than a millisecond



Disk Scheduling

- Should ensure a fast access time and disk bandwidth
- Fast access
 - Minimize total seek time of a group of requests
 - If requests are for different cylinders, average rotation latency has to be incurred for each anyway, so minimizing it is not the primary goal (though some scheduling possible if multiple requests for same cylinder is there)
- Seek time \approx seek distance
- Main goal : reduce total seek distance for a group of requests
- Auxiliary goal: fairness in waiting times for the requests
- Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer



Disk Scheduling (Cont.)

- Several algorithms exist to schedule the servicing of disk I/O requests.
- We illustrate them with a request queue (0-199).

98, 183, 37, 122, 14, 124, 65, 67

Head pointer 53



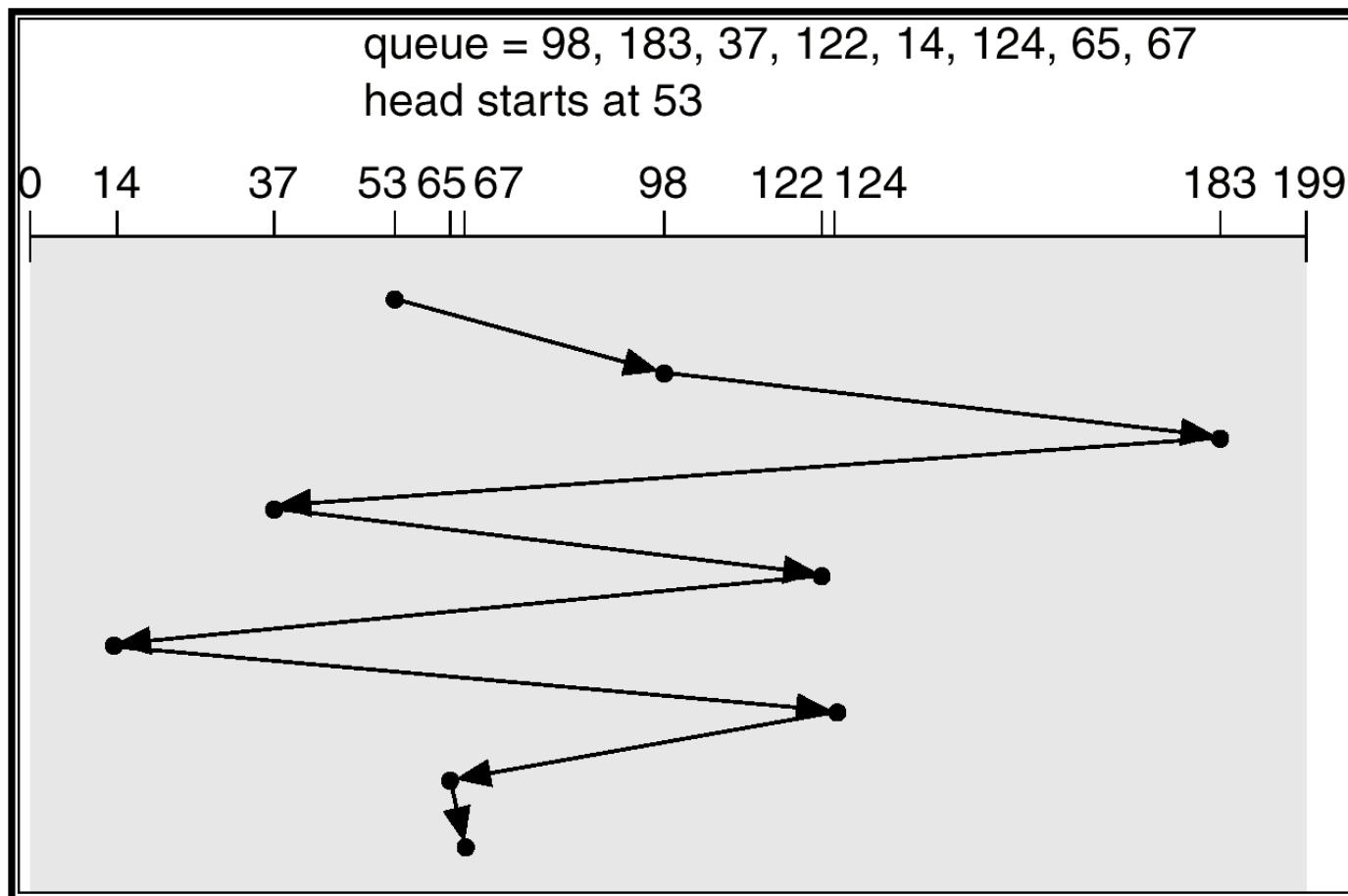
FCFS

- Service requests in the order they come
- Fair to all requests
- Can cause very large total seek time over all requests if the load is moderate to high



FCFS

Illustration shows total head movement of 640 cylinders.



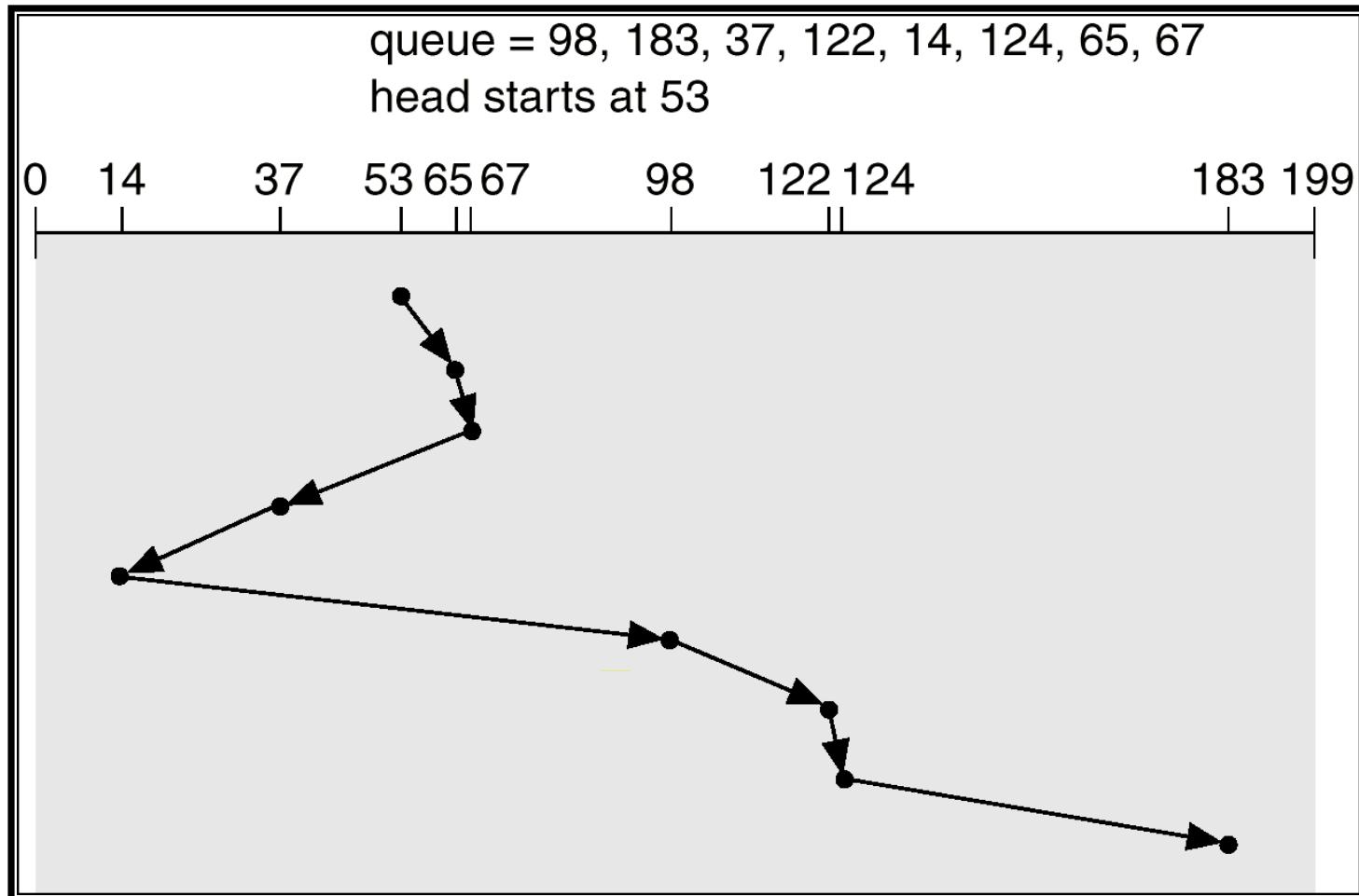


SSTF

- Selects the request with the minimum seek time from the current head position
- SSTF scheduling is a form of SJF scheduling
 - May cause starvation of some requests like SJF
 - But not optimal, unlike SJF
- Minimizes seek time, but not fair
- May work well if the load is not high



SSTF (Cont.)





SCAN

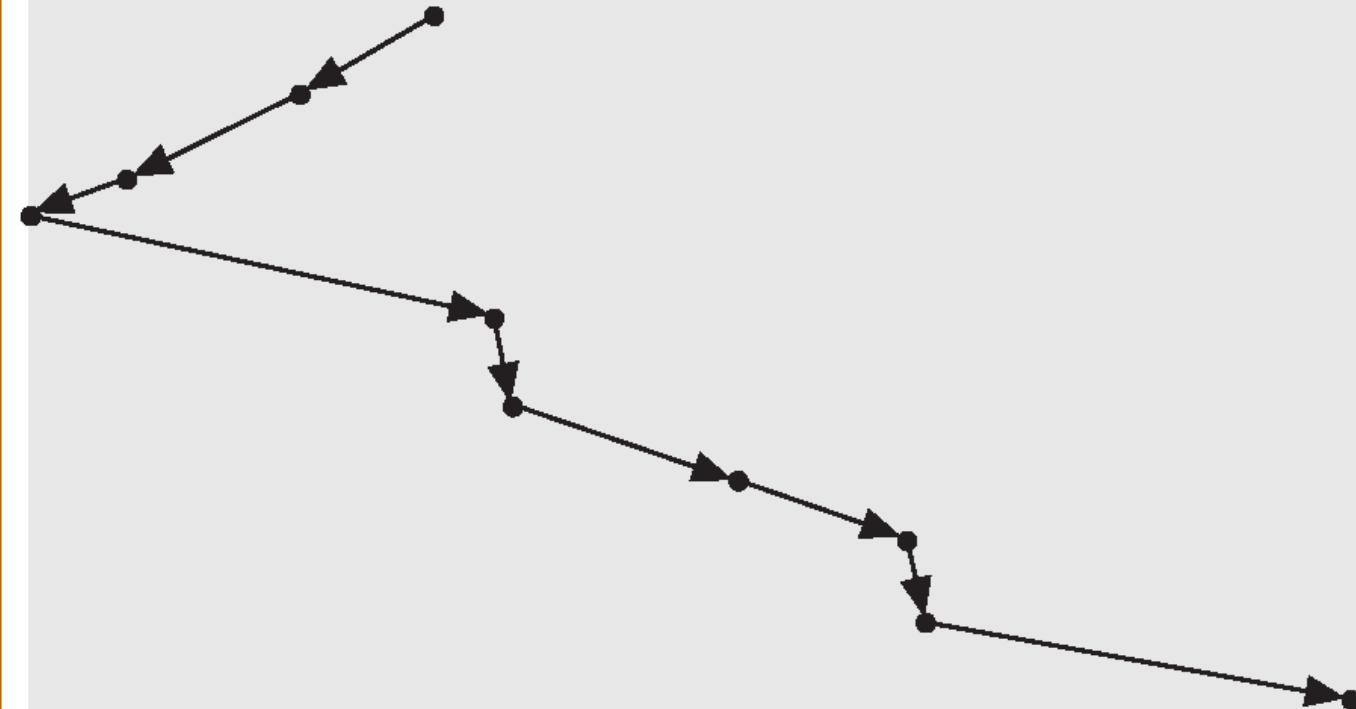
- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues
- Sometimes called the *elevator algorithm*



SCAN (Cont.)

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

0 14 37 53 65 67 98 122 124 183 199



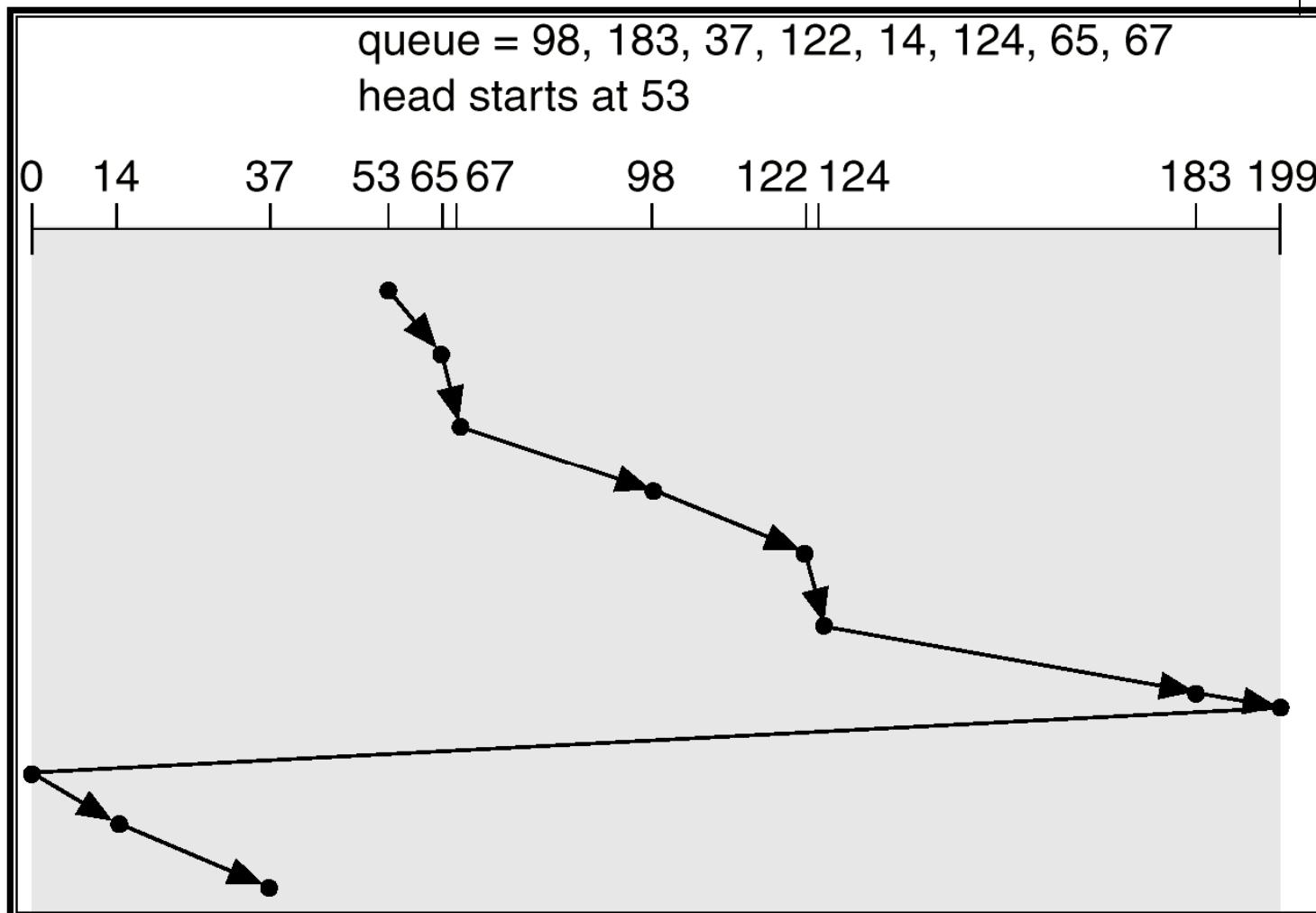


C-SCAN

- Provides a more uniform wait time than SCAN
- 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 (Cont.)



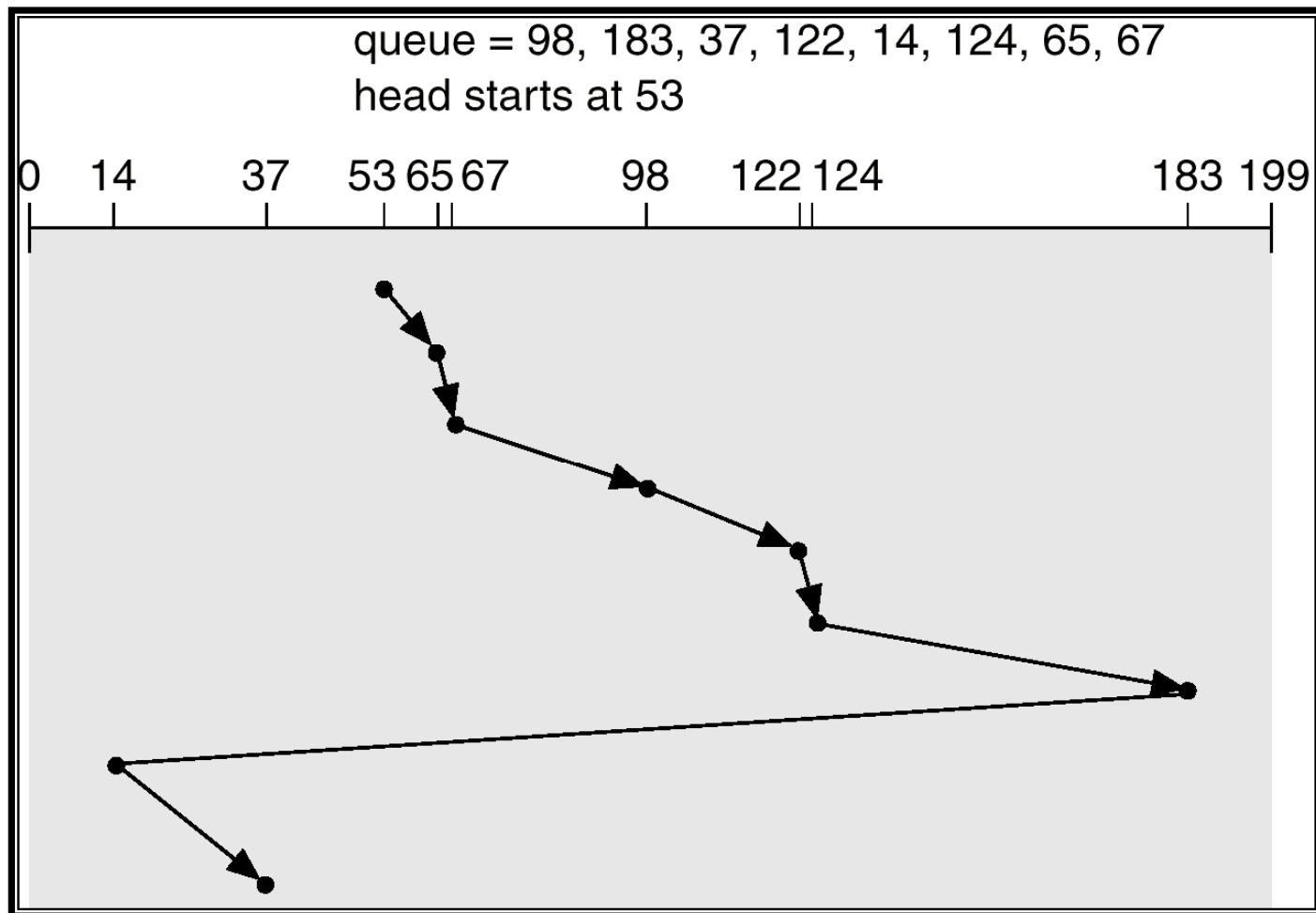


C-LOOK

- 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.



C-LOOK (Cont.)



Selecting a Disk-Scheduling Algorithm



- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk
- Performance depends on the number and types of requests
- Requests for disk service can be influenced by the file-allocation method
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary
- Either SSTF or C-LOOK is a reasonable choice for the default algorithm (depending on load)



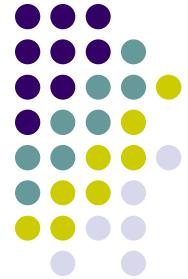
Disk Management

- *Low-level formatting*, or *physical formatting* — Dividing a disk into sectors that the disk controller can read and write.
- To use a disk to hold files, the operating system still needs to record its own data structures on the disk
 - *Partition* the disk into one or more groups of cylinders
 - *Logical formatting* or “making a file system”
- Boot block initializes system
 - The bootstrap is stored in ROM
 - *Bootstrap loader* program
- Methods such as *sector sparing* used to handle bad blocks



Operating System Issues

- Major OS jobs are to manage physical devices and to present a virtual machine abstraction to applications
- For hard disks, the OS provides two abstraction:
 - Raw device – an array of data blocks.
 - File system – the OS queues and schedules the interleaved requests from several applications.



Application Interface

- Most OSs handle removable disks almost exactly like fixed disks — a new cartridge is formatted and an empty file system is generated on the disk.
- Tapes are presented as a raw storage medium, i.e., and application does not open a file on the tape, it opens the whole tape drive as a raw device
- Usually the tape drive is reserved for the exclusive use of that application
- Since the OS does not provide file system services, the application must decide how to use the array of blocks
- Since every application makes up its own rules for how to organize a tape, a tape full of data can generally only be used by the program that created it

Numerical on Disk Management

Muktikanta Sahu

Department of Computer Science



IIIT Bhubaneswar
Imagine, Innovate, Inspire
(A University established by Government of Odisha)

International Institute of Information Technology Bhubaneswar
muktikanta@iiit-bh.ac.in

April 2, 2019



Question-1

A program of size 64KB is stored on disk which supports an average seek time of 30 ms and rotation time of 20 ms. Page size is 4 MB and track size is 32 MB; if the pages of the program are contiguously placed on Disk and the data transfer time is 10 ms, then what is the total time required to load the program from Disk in ms?

Answer

One track size = 32 MB

Number of tracks required to store the data = $64\text{MB}/32\text{MB} = 2 \text{ Track}$.

Seek time: time required to move the R/W Header to desired track = 30 ms

Rotational Latency: placing the data under R/W header = 20ms

Now Total time taken for the first track = Seek Time + Rotation latency + Transfer time = $30 \text{ ms} + 20 \text{ ms} + 10 \text{ ms} = 60 \text{ ms}$

After reading the complete one track, again there will be a need to place R/W header at the track in which next 32MB data present. So for this, it will again perform seek operation and find the location from where the next data is present.

Total time taken to transfer the second track

= Seek Time + Rotation latency + Transfer time

= $30 \text{ ms} + 20 \text{ ms} + 10 \text{ ms}$

= 60 ms

Total time = $60 + 60 = 120 \text{ ms}$

Question-2

Consider a disk pack with the following specifications- 16 surfaces, 128 tracks per surface, 256 sectors per track and 512 bytes per sector. Answer the following questions:

Question - 2(i)

What is the capacity of disk pack?

Answer - 2(i)

Given-

- Number of surfaces = 16
- Number of tracks per surface = 128
- Number of sectors per track = 256
- Number of bytes per sector = 512 bytes

Capacity of disk pack

= Total number of surfaces x Number of tracks per surface x Number of sectors per track x Number of bytes per sector

$$= 16 \times 128 \times 256 \times 512 \text{ bytes}$$

$$= 2^{28} \text{ bytes}$$

$$= 256 \text{ MB}$$

Question - 2(ii)

What is the number of bits required to address the sector?

Answer - 2(ii)

Total number of sectors

= Total number of surfaces x Number of tracks per surface x Number of sectors per track

= $16 \times 128 \times 256$ sectors

= 2^{19} sectors

Thus, Number of bits required to address the sector = 19 bits

Question - 2(iii)

If the format overhead is 32 bytes per sector, what is the formatted disk space?

Answer - 2(iii)

Formatting overhead

= Total number of sectors \times overhead per sector

= $2^{19} \times 32$ bytes

= $2^{19} \times 2^5$ bytes

= 2^{24} bytes

= 16 MB

Now, Formatted disk space = Total disk space – Formatting overhead

= 256 MB – 16 MB

= 240 MB

Question - 2(iv)

If the format overhead is 64 bytes per sector, how much amount of memory is lost due to

Answer - 2(iv)

Amount of memory lost due to formatting

= Formatting overhead

= Total number of sectors x Overhead per sector

= $2^{19} \times 64$ bytes

= $2^{19} \times 2^6$ bytes

= 2^{25} bytes

= 32 MB

Question - 2(v)

If the diameter of innermost track is 21 cm, what is the maximum recording density?

Answer - 2(v)

Storage capacity of a track

= Number of sectors per track x Number of bytes per sector

= 256×512 bytes

= $2^8 \times 2^9$ bytes

= 2^{17} bytes

= 128 KB

Circumference of innermost track

= $2 \times \pi \times \text{radius}$

= $\pi \times \text{diameter}$

= 3.14×21 cm

= 65.94 cm

Now, Maximum recording density

= Recording density of innermost track

= Capacity of a track / Circumference of innermost track

= 128 KB / 65.94 cm

= 1.94 KB/cm

Question - 2(vi)

If the diameter of innermost track is 21 cm with 2 KB/cm, what is the capacity of one track?

Answer - 2(vi)

Circumference of innermost track

$$= 2 \times \pi \times \text{radius}$$

$$= \pi \times \text{diameter}$$

$$= 3.14 \times 21 \text{ cm}$$

$$= 65.94 \text{ cm}$$

Capacity of a track

= Storage density of the innermost track \times Circumference of the innermost track

$$= 2 \text{ KB/cm} \times 65.94 \text{ cm}$$

$$= 131.88 \text{ KB}$$

$$\cong 132 \text{ KB}$$

Question - 2(vii)

If the disk is rotating at 3600 RPM, what is the data transfer rate?

Answer - 2(vii)

Number of rotations in one second

$$= (3600 / 60) \text{ rotations/sec}$$

$$= 60 \text{ rotations/sec}$$

Now, Data transfer rate

= Number of heads x Capacity of one track x Number of rotations in one second

$$= 16 \times (256 \times 512 \text{ bytes}) \times 60$$

$$= 2^4 \times 2^8 \times 2^9 \times 60 \text{ bytes/sec}$$

$$= 60 \times 2^{21} \text{ bytes/sec}$$

$$= 120 \text{ MBps}$$

Thank you