RAMAIAH
Institute of Technology

M. S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)

Department of Computer Science and Engineering

Tutorial on Operating Systems Concepts

Chandrika Prasad
Vandana S Sardar
Dr. Mohan K S

Topics for the Discussion

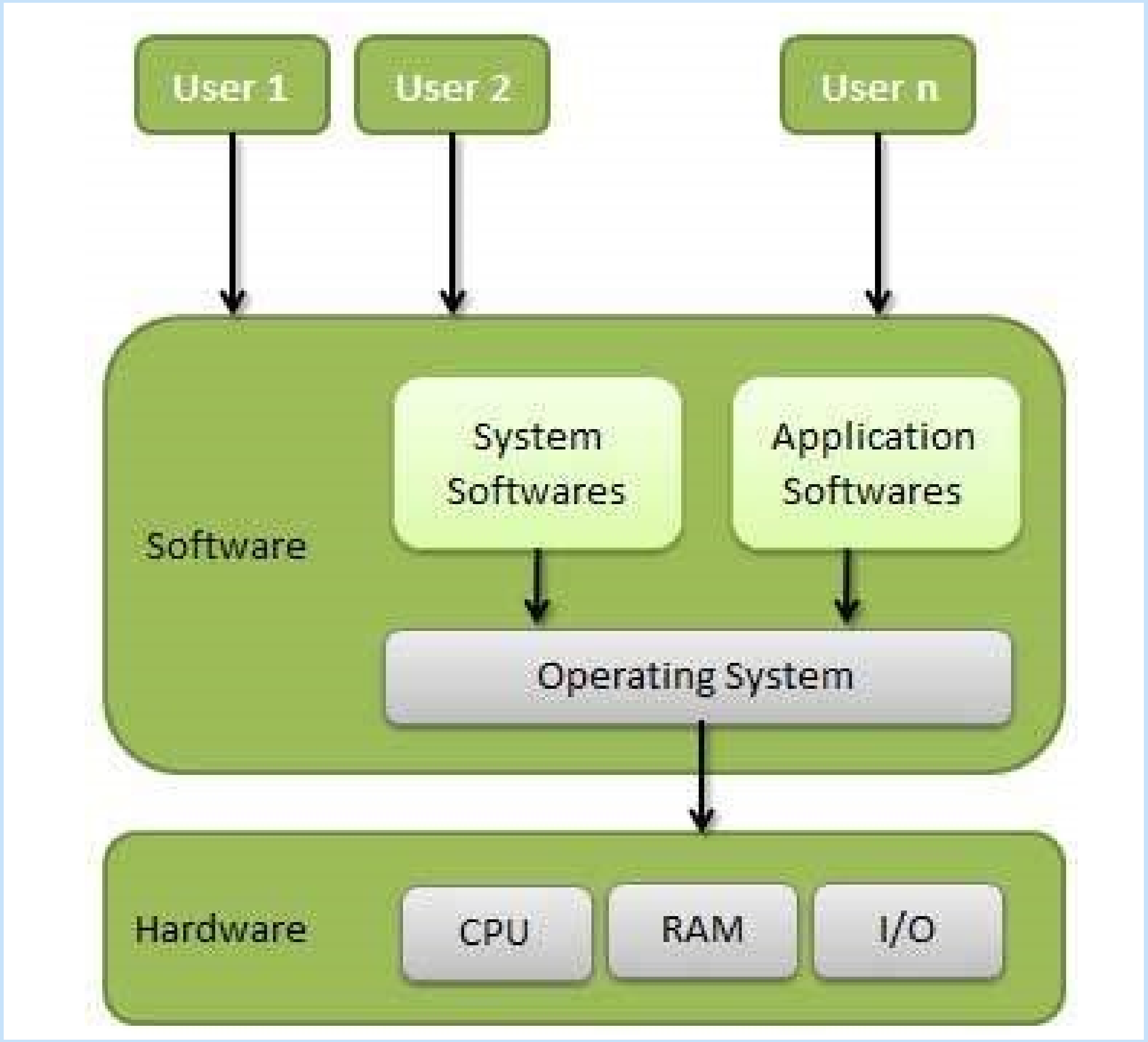
1. OS and its functions
2. Virtual Machine
3. Process Management
   - Scheduling algorithms
   - Synchronization
4. Deadlocks
5. Memory Management

OS and its functions

- An operating system (OS) is a collection of software that manages computer hardware resources and provides common services for computer programs.

- The operating system is a vital component of the system software in a computer system.

- An Operating System (OS) is an interface between a computer user and computer hardware.


User 1
User 2
User n
Software
System Softwares
Application Softwares
Operating System
Hardware
CPU
RAM
I/O

• Functions contd

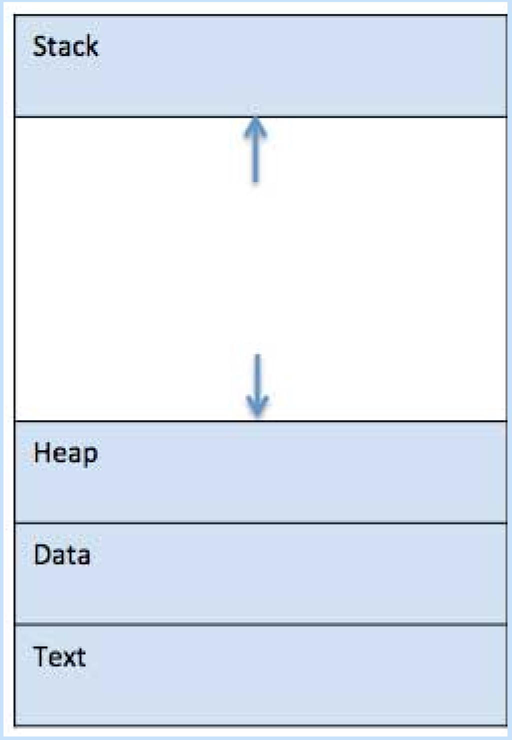
Following are some of important functions of an operating System.

- Memory Management
- Processor Management
- Device Management
- File Management
- Security
- Control over system performance
- Job accounting
- Error detecting aids
- Coordination between other software and users

Process Management

- A process is basically a program in execution. The execution of a process must progress in a sequential fashion.
- A process is defined as an entity which represents the basic unit of work to be implemented in the system.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data. The following image shows a simplified layout of a process inside main memory



- The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.
- Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

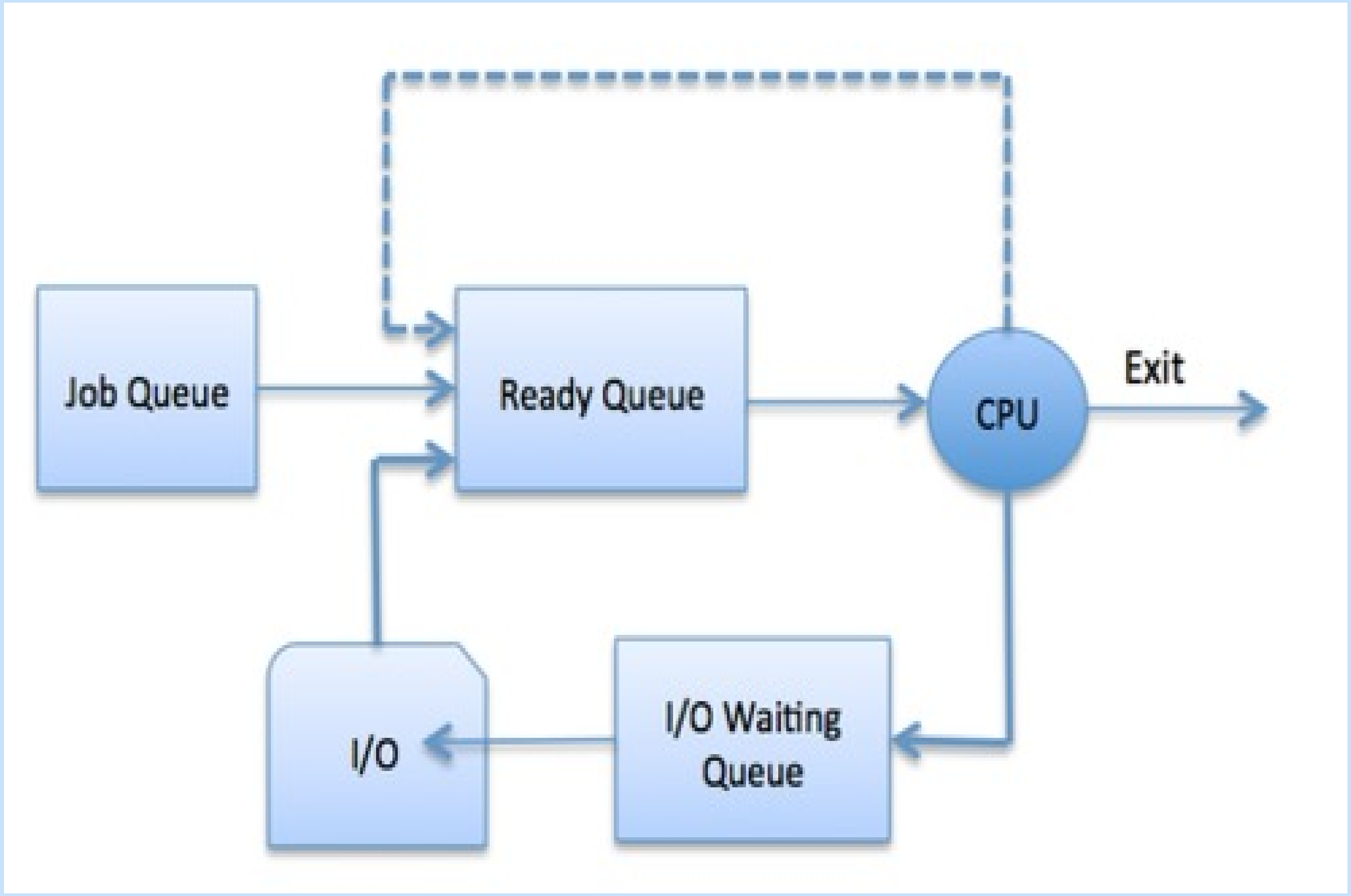
• Scheduling Queues

- The Operating System maintains the following important process scheduling queues −

Job queue − This queue keeps all the processes in the system.

Ready queue − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.

Device queues − The processes which are blocked due to unavailability of an I/O device constitute this queue.


Job Queue
Ready Queue
CPU
Exit
I/O
I/O Waiting Queue

- Schedulers

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run.

- Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

Scheduling algorithms

- A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms.
- These algorithms are either **non-preemptive or preemptive**.
- Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time.
- Preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

- There are six popular process scheduling algorithms which we are going to discuss in
- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

Questions on OS concepts and Process

1) What are the different operating systems?
2) What are the basic functions of an operating system?
3) What is user space and kernel space?
4) What are the different kinds of kernels?
5) What are different states of process?
6) What is context switching?
7) Is non-preemptive scheduling frequently used in a computer? Why?
8) What is a time-sharing system?
9) What are real-time systems?

11. Why is round robin algorithm considered better than first come first serve algorithm?

12. Which of the following is the number of processes that complete their execution per time unit?

a. Throughput   b. Turnaround time

c. Waiting time   d. Response time

13.What is the average waiting time for the following processes with non preemptive SJF (Shortes Job First).

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

a. 6.5

b. 6.75

c. 7.5

d 7.75

Which of the following is the amount of time to execute a particular process ?

a. Throughput

b. Turnaround time

c. Waiting time

d Response time

Round robin scheduling falls under the category of ____________

a) Non-preemptive scheduling

b) Preemptive scheduling

c) All of the mentioned

d) None of the mentioned

The portion of the process scheduler in an operating system that dispatches processes is concerned with ____________

a) assigning ready processes to CPU

b) assigning ready processes to waiting queue

c) assigning running processes to blocked queue

d) all of the mentioned

What is FIFO algorithm?
a) first executes the job that came in last in the queue
b) first executes the job that came in first in the queue
c) first executes the job that needs minimal processor
d) first executes the job that has maximum processor needs

There are 10 different processes running on a workstation. Idle processes are waiting for an input event in the input queue. Busy processes are scheduled with the Round-Robin time sharing method. Which out of the following quantum times is the best value for small response times, if the processes have a short runtime, e.g. less than 10ms?
a) tQ = 15ms
b) tQ = 40ms
c) tQ = 45ms
d) tQ = 50ms

Which of the following statements are true?

I. Shortest remaining time first scheduling may cause starvation
II. Preemptive scheduling may cause starvation
III. Round robin is better than FCFS in terms of response time

a) I only
b) I and III only
c) II and III only
d) I, II and III

Memory Management

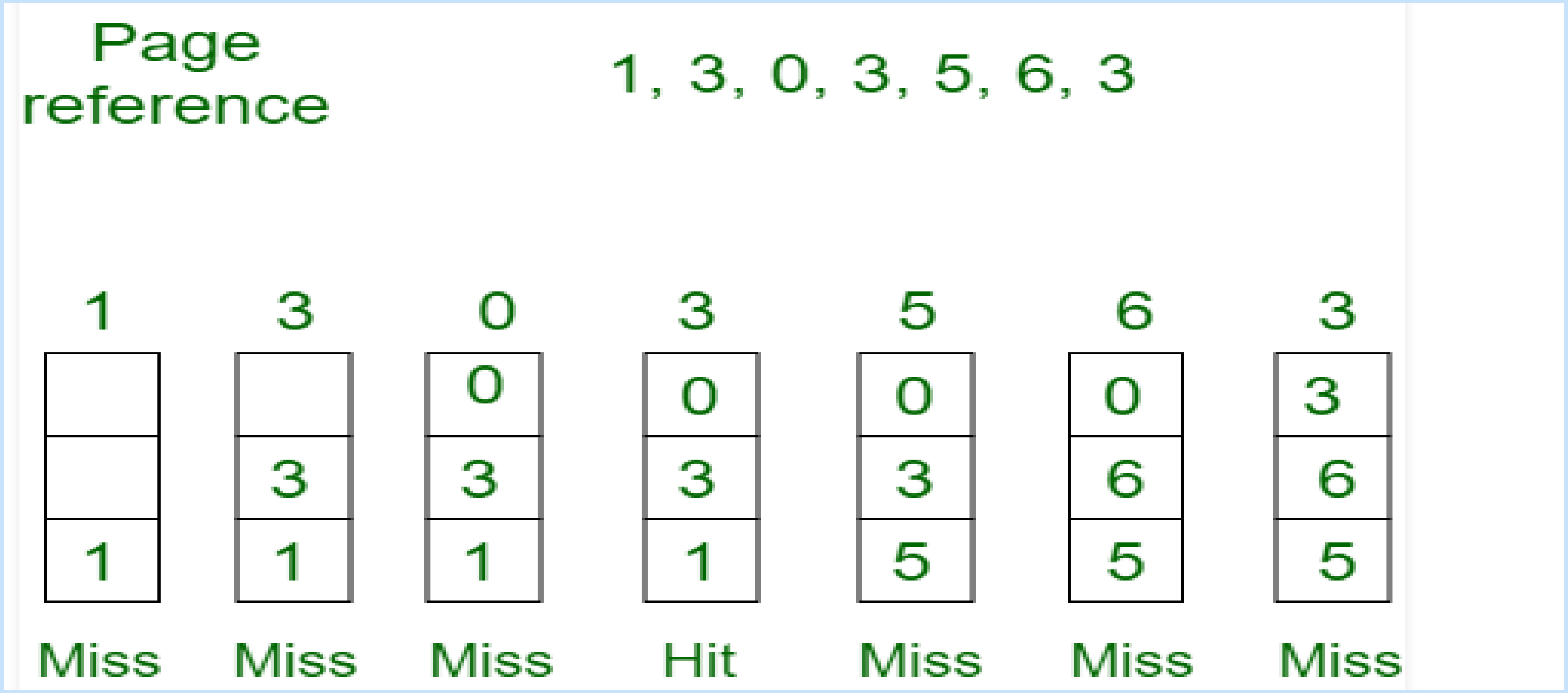
Virtual memory: An illusion created to the user that enough physical memory is available to execute user program
Paging: Dividing logical memory into partitions using constant size.

Segmentation: Dividing memory into partitions using random size

Page Replacement algorithms: used to decide which of the frame to be loaded into which of the page

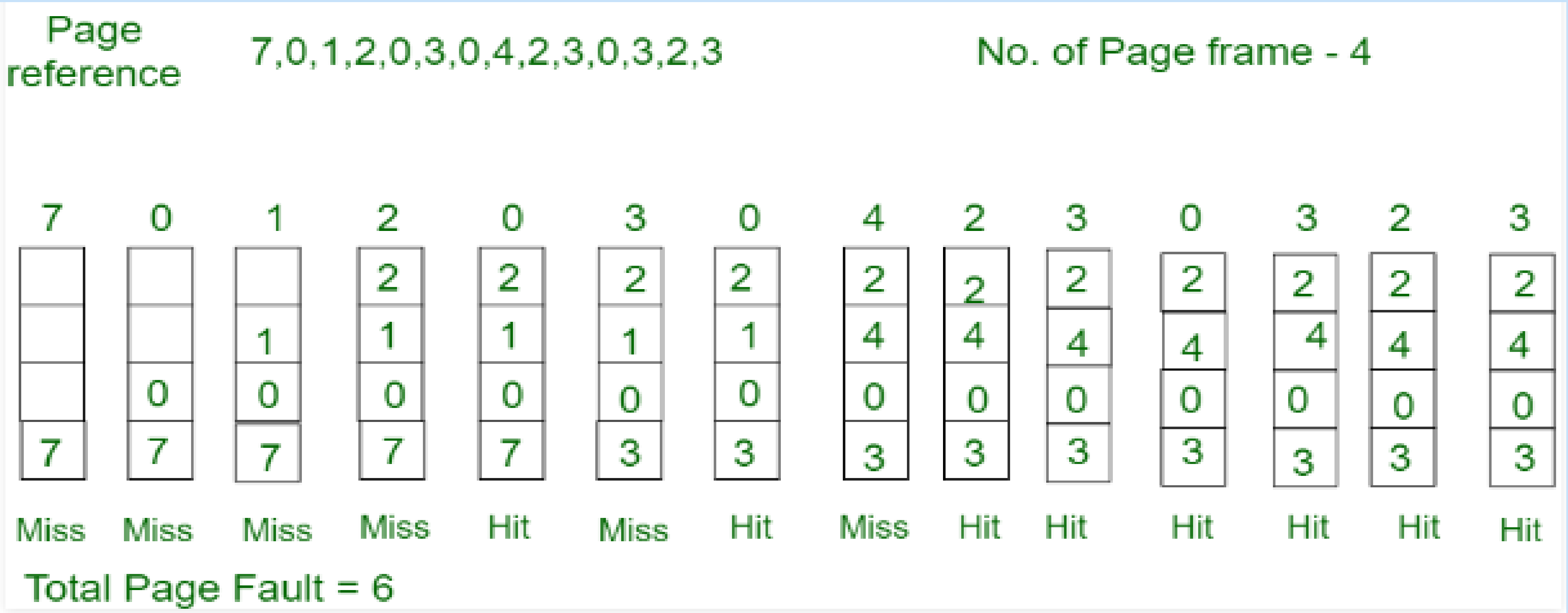
Page replacement algorithms

- First in-first out (FIFO): This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.


Page reference
1, 3, 0, 3, 5, 6, 3
1
3
0
3
5
6
3
1
3
1
0
3
1
0
3
1
0
3
5
0
6
5
3
6
5
Miss
Miss
Miss
Hit
Miss
Miss
Miss

Optimal page replacement algorithm

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

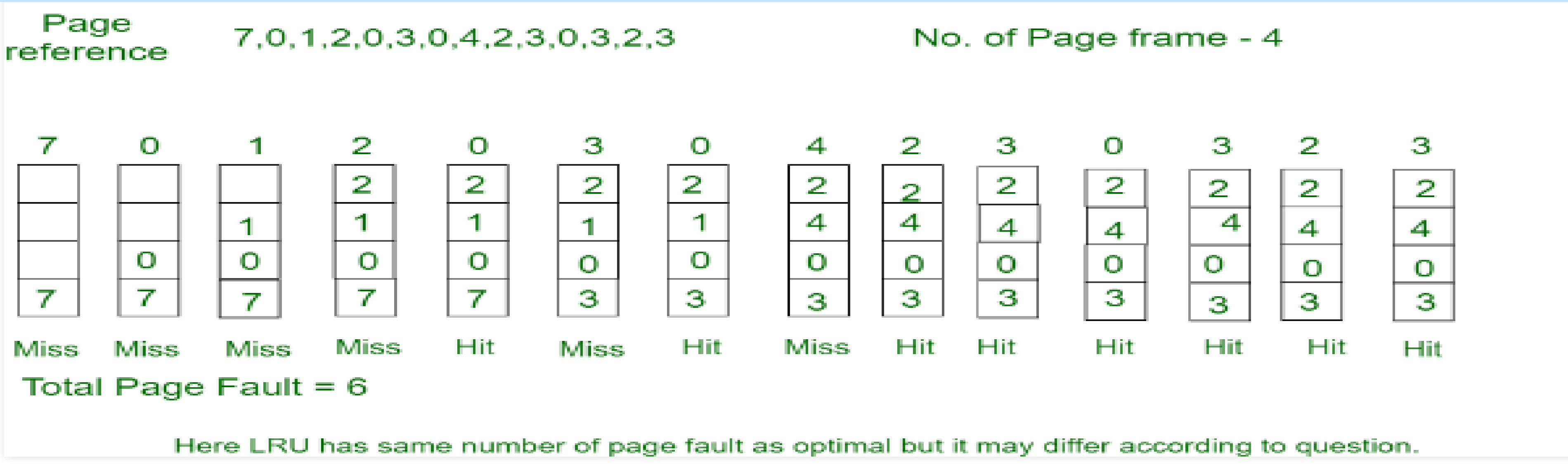


7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit	Hit	Hit	Hit

Total Page Fault = 6

Least recently used algorithm(LRU)

In this algorithm page will be replaced which is least recently used.



Total Page Fault = 6

Here LRU has same number of page fault as optimal but it may differ according to question.

Question and Answer 2

A memory buffer used to accommodate a speed differential is called ____________

a) stack pointer
b) cache
c) accumulator
d) disk buffer

CPU fetches the instruction from memory according to the value of ____________

a) program counter
b) status register
c) instruction register
d) program status word

Which one of the following is the address generated by CPU?
a) physical address
b) absolute address
c) logical address
d) none of the mentioned

Run time mapping from virtual to physical address is done by ______________
a) Memory management unit
b) CPU
c) PCI
d) None of the mentioned

The page table contains _____________

a) base address of each page in physical memory
b) page offset
c) page size
d) none of the mentioned

Memory management technique in which system stores and retrieves data from secondary storage for use in main memory is called?
a) fragmentation
b) paging
c) mapping
d) none of the mentioned

The address of a page table in memory is pointed by ____________
a) stack pointer
b) page table base register
c) page register
d) program counter

What is compaction?

a) a technique for overcoming internal fragmentation
b) a paging technique
c) a technique for overcoming external fragmentation
d) a technique for overcoming fatal error

Operating System maintains the page table for _____________

a) each process
b) each thread
c) each instruction
d) each address

Important Tips (Optional)

If the size of logical address space is 2 ^m, and a page size is 2 ^ n addressing units, then the high order ______ bits of a logical address designate the page number, and the _____ low order bits designate the page offset.

a) m, n
b) n, m
c) m – n, m
d) m – n, n

References

Consider a machine with 64 MB physical memory and a 32-bit virtual address space. If the page size is 4KB, what is the approximate size of the page table?

(A) 16 MB

(B) 8 MB

(C) 2 MB

(D) 24 MB

Ans: (C)

- Using above formula we can say that there will be 2^(32-12) = 2^20 entries in page table.
- Since memory is byte addressable. So we take that each page table entry is 16 bits i.e. 2 bytes long.
- so 2^20 * 2=2mb

Which of the following are the goals of the virtual memory
a)transparency
b)efficiency
c)protection
d) all of the above

Virtual memory is
a)Large secondary memory
b)Large main memory
c)Illusion of large main memory
d)None of the above

Which of the following statements is false?

a) Virtual memory implements the translation of a program's
address space into physical memory address space
b) Virtual memory allows each program to exceed the size of
the primary memory
c) Virtual memory increases the degree of multiprogramming
d) Virtual memory reduces the context switching overhead

The dirty bit in the page address is set to 1 when
(A)A page is removed from the main memory
(B)When a modification has been made to the loaded page
(C)When the page is no longer valid
(D)When the page is not available for use

The aim of creating page replacement algorithms is to _____________
a) replace pages faster
b) increase the page fault rate
c) decrease the page fault rate
d) to allocate multiple pages to processes

A FIFO replacement algorithm associates with each page the _______
a) time it was brought into memory
b) size of the page in memory
c) page after and before it
d) all of the mentioned

What is the Optimal page – replacement algorithm?
a) Replace the page that has not been used for a long time
b) Replace the page that has been used for a long time
c) Replace the page that will not be used for a long time
d) None of the mentioned

Optimal page – replacement algorithm is difficult to implement, because _____________
a) it requires a lot of information
b) it requires future knowledge of the reference string
c) it is too complex
d) it is extremely expensive

LRU page – replacement algorithm associates with each page the ______

a) time it was brought into memory

b) the time of that page's last use

c) page after and before it

d) all of the mentioned

For 3 page frames, the following is the reference string:

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

How many page faults does the LRU page replacement algorithm produce?

a) 10

b) 15

c) 11

d) 12

What are the two methods of the LRU page replacement policy that can be implemented in hardware?

a) Counters

b) RAM & Registers

c) Stack & Counters

d) Registers

Deadlocks

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 disk drives.
  - P_1 and P_2 each hold one disk drive and each needs another one.
- Example
  - semaphores *A* and *B*, initialized to 1

```
P0              P1
wait (A);       wait(B)
wait (B);       wait(A)
```

System Model

- Resource types R_1, R_2, . . ., R_m

CPU cycles, memory space, I/O devices

- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Deadlock Characterization

Deadlock can arise if 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:** 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 $\{P_0, P_1, \ldots, P_0\}$ 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

  $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by

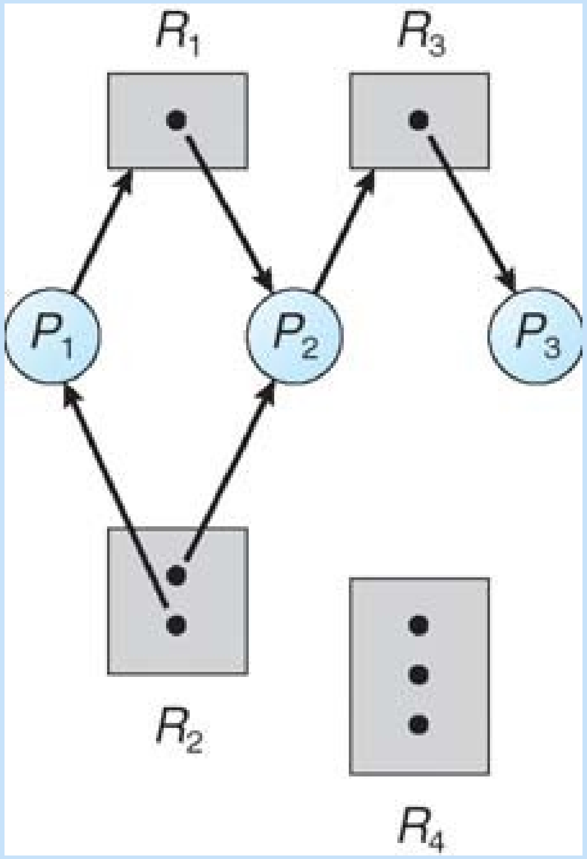
  P_n, and P_0 is waiting for a resource that is held by P_0.

Resource-Allocation Graph

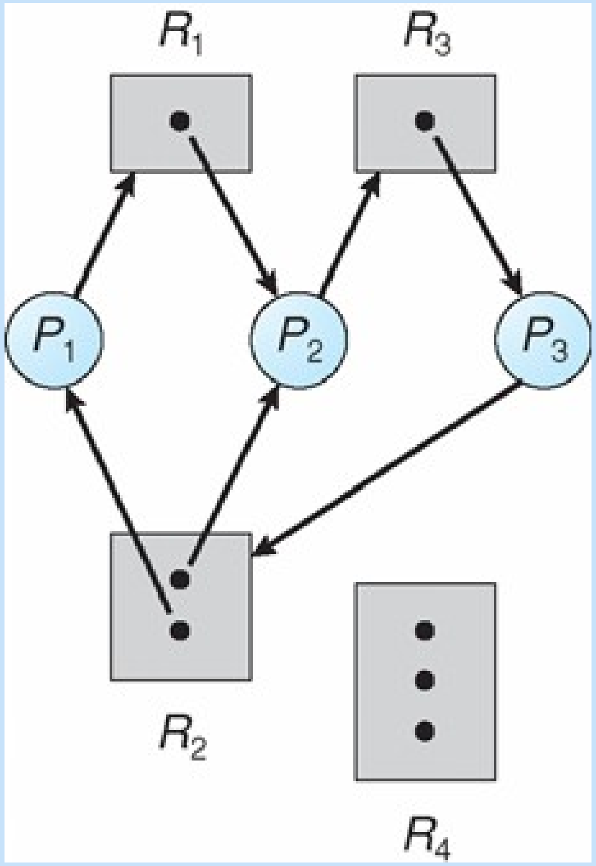
A set of vertices *V* and a set of edges *E*.

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

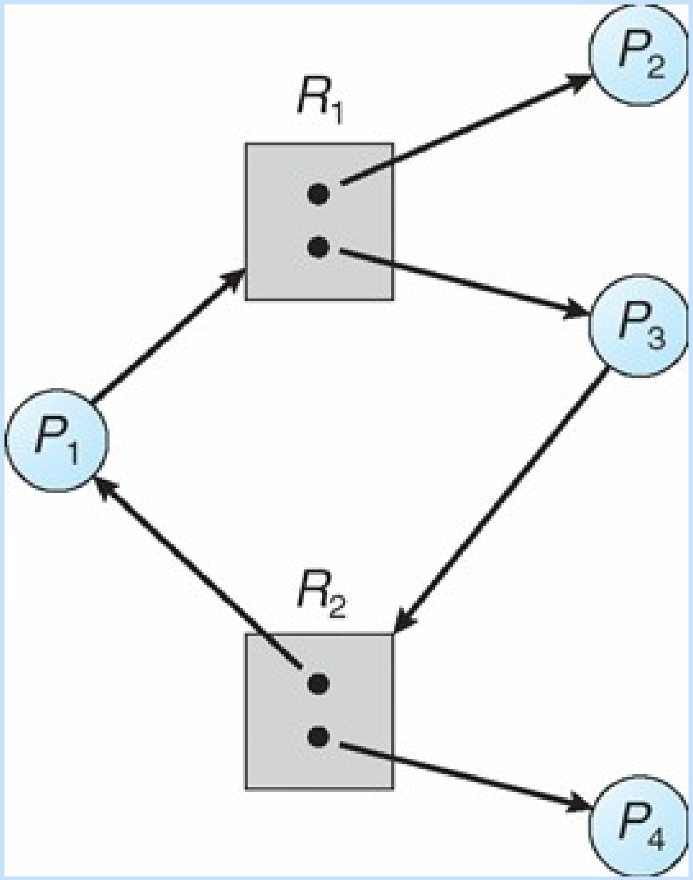
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Methods for Handling Deadlocks

- 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.

Deadlock Prevention

Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
    - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
    - Low resource utilization; starvation possible.

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

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.
- The 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

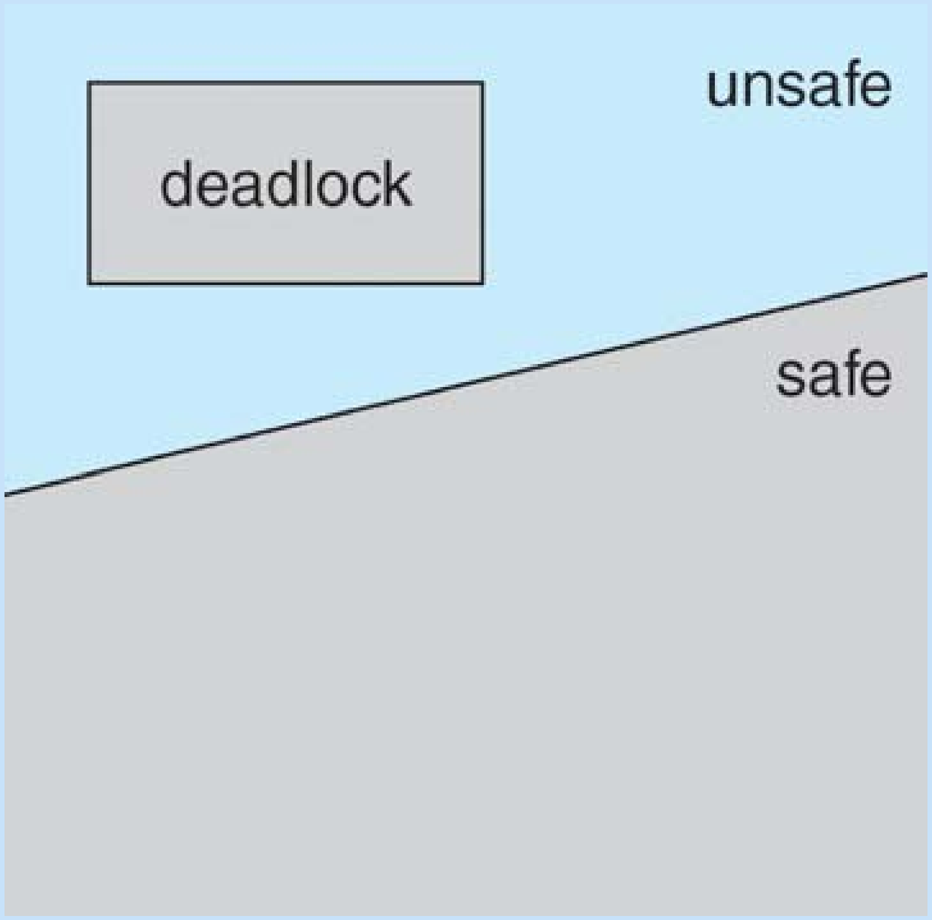
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

- System is in safe state if there exists a sequence <P_1, P_2, …, P_n> of ALL the processes is the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks.
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

Safe, Unsafe , Deadlock State



Avoidance algorithms

- Single instance of a resource type. Use a resource-allocation graph
- Multiple instances of a resource type. Use the banker’s algorithm

Banker’s Algorithm

- Multiple instances.
- 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.

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- ***Available**:* Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available.
- ***Max**:* n x m matrix. If $Max\ [i,j] = k$, then process P_i may request at most k instances of resource type R_j.
- ***Allocation**:* n x m matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j.
- ***Need**:* n x m matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task.

$Need\ [i,j] = Max[i,j] – Allocation\ [i,j]$.

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

   Work = *Available*

   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1.

2. Find and *i* such that both:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$

   If no such *i* exists, go to step 4.

3. $Work = Work + Allocation_i$ *Finish*[*i*] = *true*

   go to step 2.

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state.

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:
   $Available = Available - Request$; $Allocation_i = Allocation_i +$

   $Request_i$; $Need_i = Need_i - Request_i$;

   - *If safe* $\Rightarrow$ *the resources are allocated to Pi.*
   - *If unsafe* $\Rightarrow$ *Pi must wait, and the old resource-allocation state is restored*

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true.

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement.
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Several Instances of a Resource Type

- ***Available****:* A vector of length *m* indicates the number of available resources of each type.

- ***Allocation****:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

- ***Request****:* An *n* x *m* matrix indicates the current request of each process. If *Request* [i_j] = *k*, then process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:

   (a) *Work = Available*

   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then
 Finish[i] = false;otherwise, *Finish*[i] = *true*.

2. Find an index *i* such that both:

   (b) *Finish*[*i*] == *false*

   (c) $Request_i \leq Work$

 If no such *i* exists, go to step 4.

Detection Algorithm (Cont.)

3. *Work* = *Work* + *Allocation*$_i$ *Finish*[*i*] = *true*
 go to step 2.

4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

Algorithm requires an order of O(*m* x *n*$^{2)}$ operations to detect whether the system is in deadlocked state.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances).
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 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	

- Sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in *Finish*[*i*] = true for all *i*.

Example (Cont.)

- P_2 requests an additional instance of type C.

	Request *A B C*
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
  - 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?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

- A system has 6 identical resources and N processes competing for them. Each process can request atmost 2 resources. Which one of the following values of N could lead to a deadlock?

A :1

B : 2

C : 3

D : 4

Consider the following policies for preventing deadlock in a system with mutually exclusive resources.

I. Processes should acquire all their resources at the beginning of execution. If any resource is not available, all resources acquired so far are released

II. The resources are numbered uniquely, and processes are allowed to request for resources only in increasing resource numbers

III. The resources are numbered uniquely, and processes are allowed to request for resources only in decreasing resource numbers

IV. The resources are numbered uniquely. A process is allowed to request only for a resource with resource number larger than its currently held resources

Which of the above policies can be used for preventing deadlock?

A : Any one of I and III but not II or IV

B : Any one of I, III, and IV but not II

C : Any one of II and III but not I or IV

D : Any one of I, II,III and IV

Consider a system having “n” resources of same type. These resources are shared by 3 processes, A, B, C, These have peak demands of 3, 4, and 6 respectively. For what value of “n” deadlock won’t occur.

A : 15

B: 9

C: 10

D: 11

- A system has 3 processes sharing 4 resources. If each process needs a maximum of 2 units, then

A: Deadlock can never occur

B: Deadlock may occur

C: Deadlock has to occur

D: None of these

Concurrency

Introduction

- New abstraction for a single running process: **thread**

- **Multi-threaded program has more than** one point of execution.

- Each thread is very much like a separate process Except for a difference:
- They *share the same address space and thus can access the same data.*

Threads

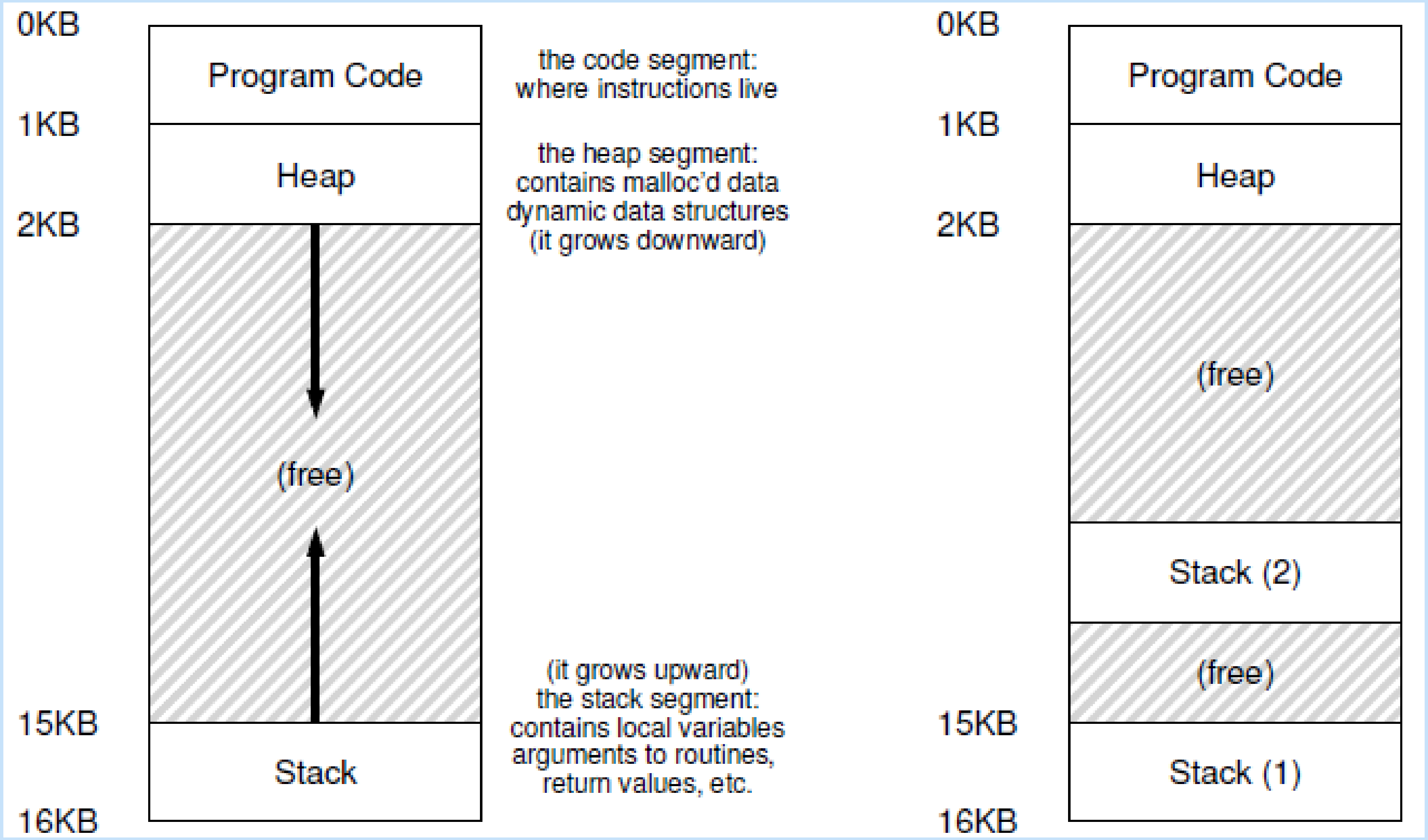
- The state of a single thread:
  - Program counter (PC).
  - Own private set of registers it uses for computation

- If there are two threads that are running on a single processor, when switching from running one (T1) to running the other (T2), a **context switch must take place.**

Process Vs Thread

- Process control blocks (PCBs) Vs thread control blocks (TCBs)
- Address space remains the same (i.e., there is no need to switch which page table we are using).
- For a process, there is a single stack, usually residing at the bottom of the address space.
- In case of a thread, one stack per thread.

Single-Threaded And Multi-Threaded Address Spaces


0KB
Program Code
the code segment: where instructions live
1KB
Heap
the heap segment: contains malloc'd data dynamic data structures (it grows downward)
2KB
(free)
(it grows upward) the stack segment: contains local variables arguments to routines, return values, etc.
15KB
Stack
16KB
0KB
Program Code
1KB
Heap
2KB
(free)
Stack (2)
(free)
15KB
Stack (1)
16KB

Why Use Threads?

- Parallelism

  Ex. Addition of arrays

- To avoid blocking program progress due to slow I/O

  Ex. Different types of I/O

- Threading enables **overlap of** I/O with other activities *within a single program*

Critical Section

- Multiple threads executing common code can result in a race condition, we call this code a **critical section.**
- **A critical section is a piece of** code that accesses a shared variable (or more generally, a shared resource)

  and must not be concurrently executed by more than one threads.
- **Mutual exclusion.** This property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

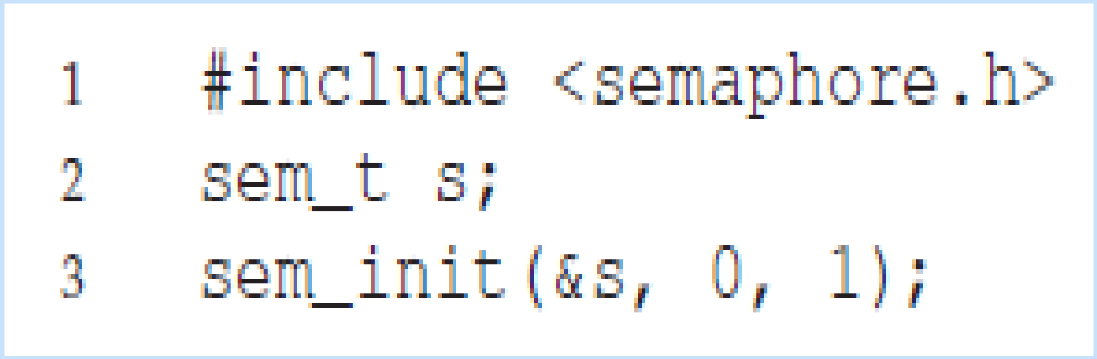
Synchronization Primitives

- To be able to build multi-threaded code.
- Accesses critical sections in a synchronized and controlled manner.
- Reliably produces the correct result despite the challenging nature of concurrent execution.

Semaphores

Semaphores

- Definition: A semaphore is an object with an integer value that we can manipulate with two routines.
- In the POSIX standard, these routines are sem_wait() and sem_post().
- Initialization:



```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

Semaphores: Definitions of wait and post

```
int sem_wait(sem_t *s) {
    decrement the value of semaphore s by one
    wait if value of semaphore s is negative
}

int sem_post(sem_t *s) {
    increment the value of semaphore s by one
    if there are one or more threads waiting, wake one
}
```

Semaphores

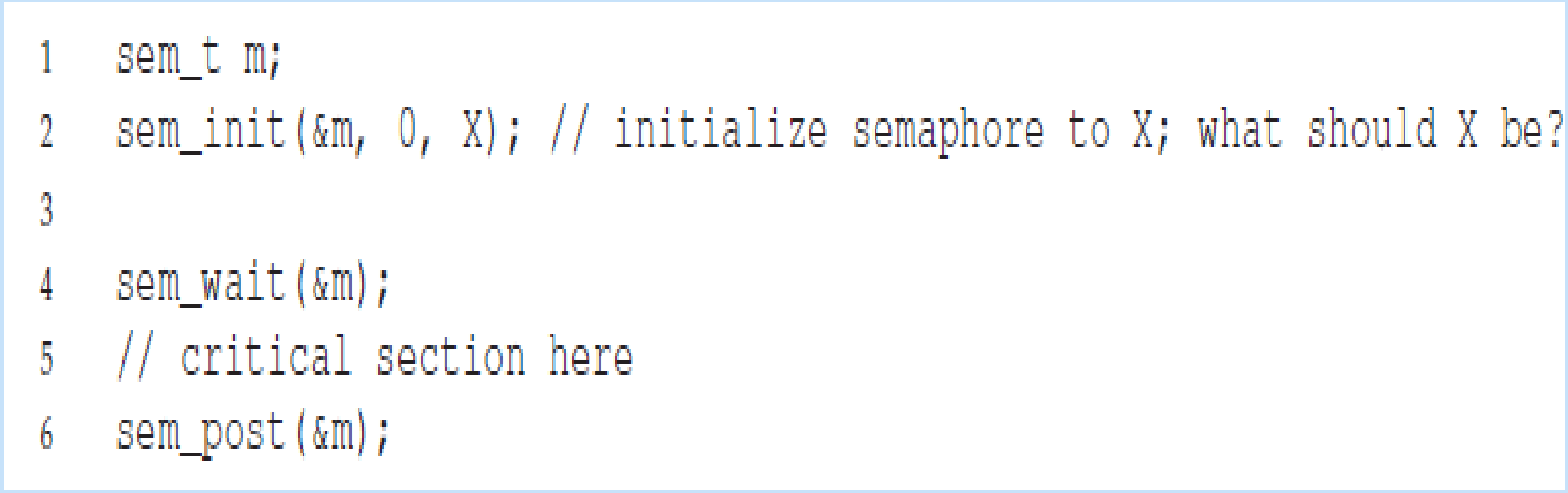
- Few salient aspects of interfaces:
- sem wait() will either return right away (because the value of the semaphore was one or higher when we called sem wait()), or it will cause the caller to suspend execution waiting for a subsequent post.
- Multiple calling threads may call into sem wait(), and thus all be queued waiting to be woken.

Semaphores

- sem_post() does not wait for some particular condition to hold like sem_wait() does. Rather, it simply increments the value of the semaphore and then, if there is a thread waiting to be woken, wakes one of them up.

- Value of the semaphore, when negative, is equal to the number of waiting threads.

A Binary Semaphore(Lock)



```
sem_t m;
sem_init(&m, 0, X); // initialize semaphore to X; what should X be?

sem_wait(&m);
// critical section here
sem_post(&m);
```

The Producer/Consumer (Bounded Buffer) Problem

- Imagine one or more producer threads and one or more consumer threads.

- Producers generate data items and place them in a buffer.

- Consumers grab said items from the buffer and consume them in some way.

The Producer/Consumer (Bounded Buffer) Problem

- First Attempt:
- Two semaphores: empty and full.

- Producer first waits for a buffer to become empty

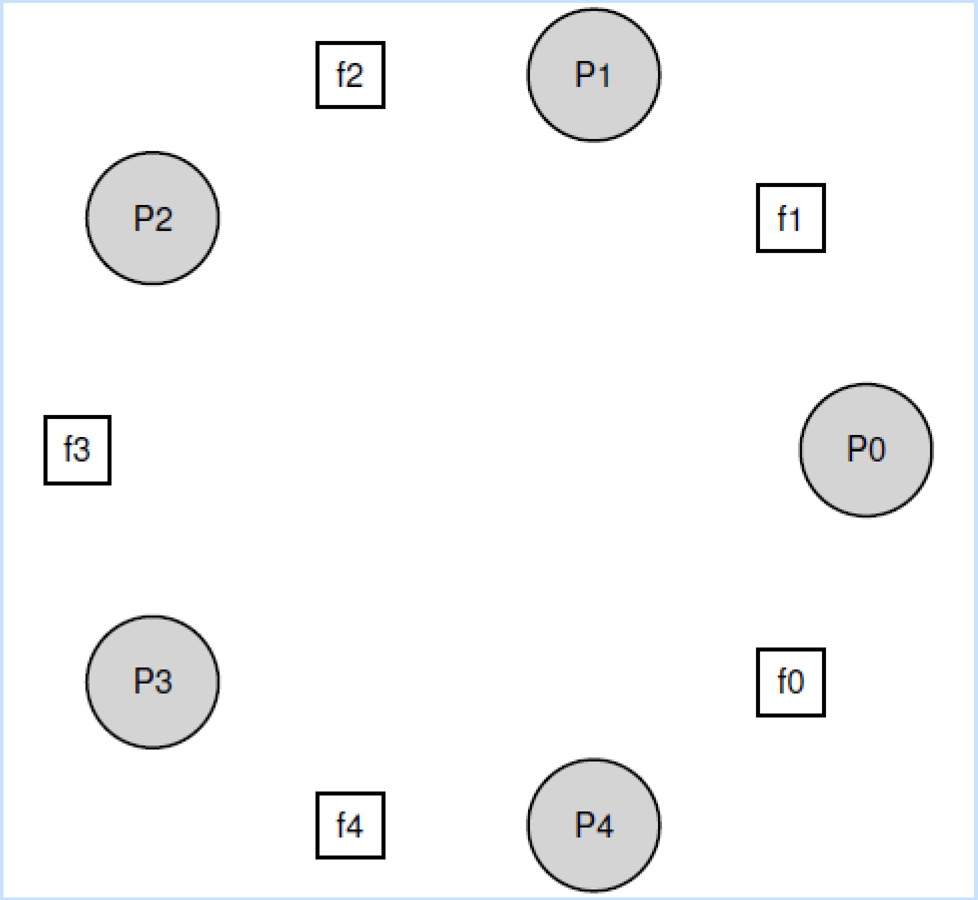
  in order to put data into it

- Consumer waits for a buffer to become filled before using it.
- Let us first imagine that MAX=1 (one item)

Reader-Writer Locks

- Different data structure accesses might require different kinds of locking.
- For example, imagine a number of concurrent list operations, including inserts and simple lookups.
- While inserts change the state of the list (and thus a traditional critical section makes sense), lookups simply *read the data structure; as long as we can* guarantee that no insert is on-going, we can allow many lookups to proceed concurrently.
- The special type of lock we will now develop to support this type of operation is known as a **reader-writer lock**

The Dining Philosophers



- A critical section is a program segment
  (a) which should run in a certain specified amount of time
  (b) which avoids deadlocks
  (c) where shared resources are accessed
  (d) which must be enclosed by a pair of semaphore operations, P and V

- A solution to the Dining Philosophers Problem which avoids deadlock is
  (a) ensure that all philosophers pick up the left fork before the right fork
  (b) ensure that all philosophers pick up the right fork before the left fork
  (c) ensure that one particular philosopher picks up the left fork before the right fork, and that all other philosophers pick up the right fork before the left fork
  (d) None of the above

- A counting semaphore was initialized to 10. Then 6 P (wait) operations and 4V (signal) operations were completed on this semaphore. The resulting value of the semaphore is
- (a) 0 (b) 8 (c) 10 (d) 12