UNIT 2: PROCESS MANAGEMENT

Processes - Process Concept - Process Scheduling - Operations on Processes - Inter-process Communication; CPU Scheduling - Scheduling criteria - Scheduling algorithms: Threads - Multithread Models – Threading issues; Process Synchronization - The Critical-Section problem - Synchronization hardware – Semaphores – Mutex - Classical problems of synchronization - Monitors; Deadlock - Methods for handling deadlocks, Deadlock prevention, Deadlock avoidance, Deadlock detection, Recovery from deadlock.

PART – A

CO2	Apply Process synchronization, process scheduling, and deadlocks concepts in the given scenario to solve the problems.	K3

1. What are the states of process? (CO2-K1)

1. New – The process is being created
2. Running – Instructions are being executed
3. Waiting – Process is waiting for some other event to occur
4. Ready – Process is waiting to be assigned to a processor.
5. Terminated – The process has finished execution

2. What are the types of schedulers? (CO2-K1)

1. **Long term scheduler (Job scheduler)** - Selects processes from job pool and load them into memory for execution.
2. **Short term scheduler (CPU scheduler)** - Selects processes from ready queue and allocates the CPU to one of them.
3. **Medium term scheduler** – Schedules the removal and reintroduction of processes (swapping) in memory.

3. What is meant by context switch? (CO2-K1)

Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as context switch.

4. What is a thread? What are it's elements? (CO2-K1)

A thread otherwise called a lightweight process (LWP) is a basic unit of CPU utilization, it comprises of a thread id, a program counter, a register set and a stack. It shares with other threads belonging to the same process its code section, data section, and operating system resources such as open files and signals.

5. What are safe state and an unsafe state? (CO2-K1)

A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock. A system is in safe state only if there exists a **safe sequence**. A sequence of processes <P1,P2,....Pn> is a safe sequence for the current allocation state if, for each Pi, the resource that Pi can still request can be satisfied by the current available resource plus the resource held by all the Pj, with j<i.

If no such sequence exists, then the system state is said to be unsafe.

6. What is cascading termination? (CO2-K1)

When a parent process is exiting, the operating system does not allow a child to continue if its parent terminates. So if a parent process terminates all its children must also be terminated. This is termed as cascading termination.

7. What are conditions under which a deadlock situation may arise? (CO2-K1)

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

a. Mutual exclusion
b. Hold and wait
c. No pre-emption
d. Circular wait

8. What is the use of inter process communication? (CO2-K1)

Inter process communication provides a mechanism to allow the co-operating process to communicate with each other and synchronies their actions without sharing the same address space. It is provided a message passing system.

9. Compare user threads and kernel threads. (CO2-K2)

User threads	Kernel threads
User threads are supported above the kernel and are implemented by a thread library at the user level.	Kernel threads are supported directly by the operating system.
Thread creation & scheduling are done in the user space, without kernel intervention.	Thread creation, scheduling and management are done by the operating system.

Therefore they are fast to create and manage blocking system call will cause the entire process to block Kernel threads	Therefore they are slower to create & manage compared to user threads. If the thread performs a blocking system call, the kernel can schedule another thread in the application for execution

10. Define thread cancellation & target thread. (CO2-K1)

The thread cancellation is the task of terminating a thread before it has completed. A thread that is to be cancelled is often referred to as the target thread. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be cancelled.

11. What is preemptive and nonpreemptive scheduling? (CO2-K1)

Nonpreemptive scheduling once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or switching to the waiting state.

Preemptive scheduling can preempt a process which is utilizing the CPU in between its execution and give the CPU to another process.

12. List out the different types of process scheduling algorithms.

1. First Come First Serve
2. Shortest Job First
3. Round Robin
4. Priority Scheduling

13. What is turnaround time? (CO2-K1)

Turnaround time is the interval from the time of submission to the time of completion of a process. It is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

14. What is semaphore? List it's types.(CO2-K1)

A semaphore is a **synchronization tool** which gives solution to critical section problem. A semaphore 'S' is an integer value that, apart from initialization, is accessed only through two standard atomic operations; **wait and signal**.

Semaphores can be used to deal with the n-process critical section problem.

It can be also used to solve various Synchronization problems.

```
wait (S)
{
while S≤ 0
;do no-op;
S--;
}

signal (S)
{
  S++;
}
```

1. **Counting semaphore** – Integer value can range over an unrestricted domain.
2. **Binary semaphore** – Integer value can range only between 0 and 1.

15. Define race condition. (CO2-K1)

When several process access and manipulate same data concurrently, then the outcome of the execution depends on particular order in which the access takes place is called race condition. To avoid race condition, only one process at a time can manipulate the shared variable.

PART – B

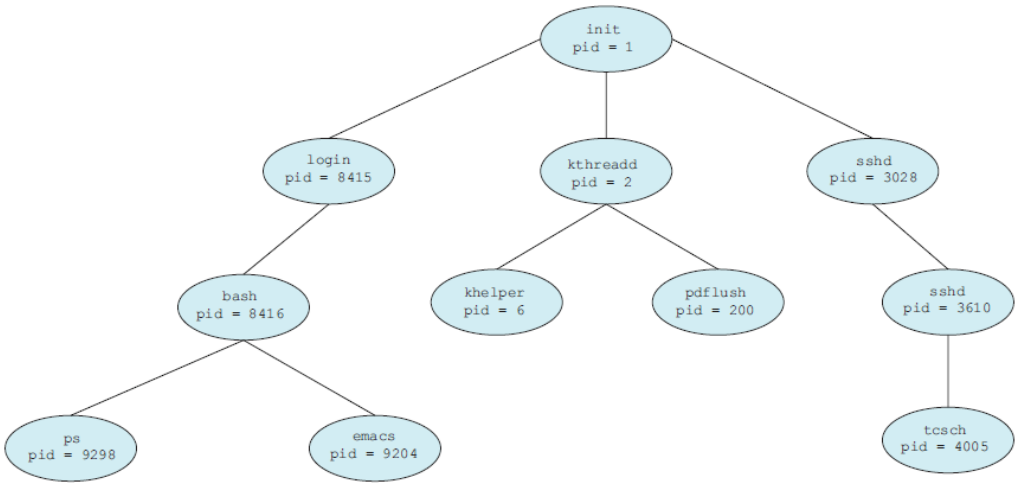
1. Discuss the different operations on Process. (CO2-K2)

- The processes in most systems can execute concurrently, and they may be created and deleted dynamically.
- Thus, these systems must provide a mechanism for process creation and termination.

Process Creation

- During the course of execution, a process may create several new processes.
- The creating process is called a **parent process**, and the new processes are called the **children** of that process.

- Each of these new processes may in turn create other processes, forming a tree of processes.
- Most operating systems (including UNIX, Linux, and Windows) identify processes according to a **unique process identifier (or pid),** which is typically an integer number.
- The pid provides a unique value for each process in the system, and it can be used as an **index** to access various attributes of a process within the kernel.
- Following Figure illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid.



A tree of processes on a typical Linux system.

- The init process (which always has a pid of 1) serves as the **root parent** process for all user processes.
- Once the system has booted, the init process can also create various user processes, such as a web or print server, an ssh server, and the like.
- In Figure two children of init—kthreadd and sshd is shown.
- The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel (in this situation, khelper and pdflush).
- The sshd process is responsible for managing clients that connect to the system by using ssh (secure shell).

- The login process is responsible for managing clients that directly log onto the system.
- Using the bash command-line interface, this user has created the process ps as well as the emacs editor.
- On UNIX and Linux systems, we can obtain a listing of processes by using

  ps –el
- This will list complete information for all processes currently active in the system.
- It is easy to construct a process tree by recursively tracing parent processes all the way to the 'init' process.
- In general, when a process creates a child process, that child process will need certain **resources (CPU time, memory, files, I/O devices)** to accomplish its task.
- A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.
- The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children.
- Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.
- In addition to supplying various physical and logical resources, the parent process may pass along initialization data (input) to the child process.
- For example, consider a process whose function is to display the contents of a file —say, image.jpg—on the screen of a terminal. When the process is created, it will get, as an input from its parent process, the name of the file image.jpg.
- Using that file name, it will open the file and write the contents out. It may also get the name of the output device. Alternatively, some operating systems pass resources to child processes.
- On such a system, the new process may get two open files, image.jpg and the terminal device, and may simply transfer the datum between the two.

- When a process creates a new process, two possibilities for execution exist:

  1. The parent continues to execute concurrently with its children.
  2. The parent waits until some or all of its children have terminated.

- There are also two address-space possibilities for the new process:

  1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
  2. The child process has a new program loaded into it.

- In UNIX, each process is identified by its process identifier, which is a unique integer.

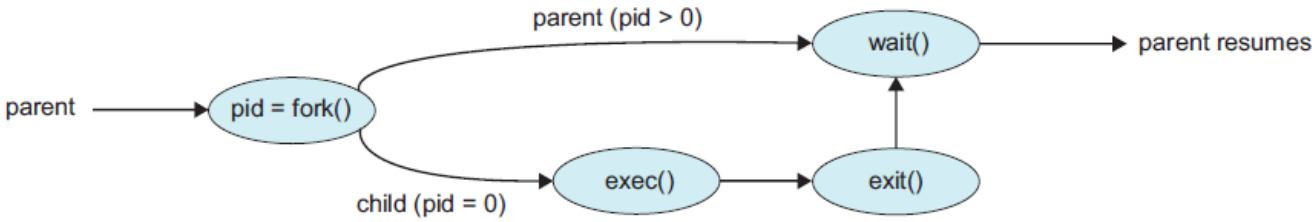
- A new process is created by the **fork()** system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.

- Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

- After a fork() system call, one of the two processes typically uses the **exec()** system call to replace the process's memory space with a new program.

- The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways.

  - The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a **wait()** system call to move itself off the ready queue until the termination of the child. Because the call to exec() overlays the process's address space with a new program, the call to exec() does not return control unless an error occurs.



Process creation using the `fork()` system call.

- Processes are created in the Windows API using the **CreateProcess()** function, which is similar to fork() in that a parent creates a new child process.
- However, whereas fork() has the child process inheriting the address space of its parent, CreateProcess() requires loading a specified program into the address space of the child process at process creation.
- Furthermore, whereas fork() is passed no parameters, CreateProcess() expects no fewer than ten parameters.

Process Termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the **exit()** system call.
- At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call).
- All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are **deallocated** by the operating system.
- Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, TerminateProcess() in Windows).
- Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs.
- A parent needs to know the identities of its children if it is to terminate them.
- Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

- A parent may terminate the execution of one of its children for a variety of reasons, such as these:

    1. The child has exceeded its usage of some of the resources that it has been allocated.
    2. The task assigned to the child is no longer required.
    3. The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

- Some systems do not allow a child to exist if its parent has terminated.

- In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

- In Linux and UNIX systems, can terminate a process by using the exit() system call, providing an exit status as a parameter:

```
exit(1);
```

- In fact, under normal termination, exit() may be called either directly or indirectly.
- A parent process may wait for the termination of a child process by using the **wait()** system call. The wait() system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid t pid;
int status;
pid = wait(&status);
```

- When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status.

- A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie process**.

- All processes transition to this state when they terminate, but generally they exist as zombies only briefly. Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released.

- Now consider what would happen if a parent did not invoke wait() and instead terminated, thereby leaving its child processes as **orphans**.

- Linux and UNIX address this scenario by assigning the init process as the new parent to orphan processes.

- The init process periodically invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

2. Explain in detail the concepts of inter process communication. (CO2-K2)

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.

- A process is **independent** if it does not share data with any other processes executing in the system.

- A process is **cooperating** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

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

  Information sharing. Since several applications may be interested in the same piece of information (for instance, copying and pasting), we must provide an environment to allow concurrent access to such information.

  Computation speedup. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.

  Modularity. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

- Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data— that is, send data to and receive data from each other.

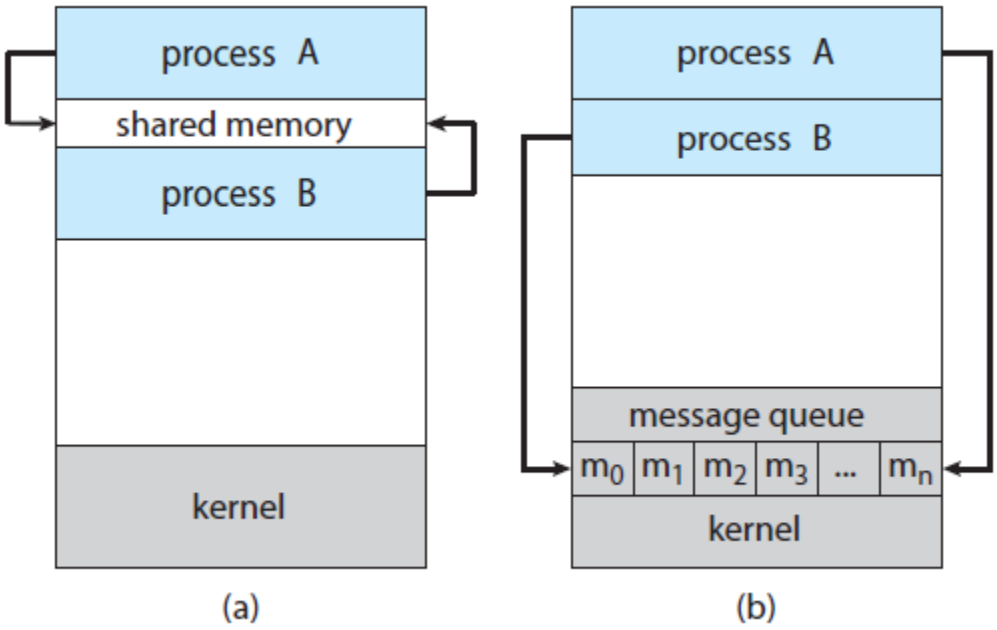
- There are two fundamental models of interprocess communication: **shared memory and message passing**.

- In the shared-memory model, a region of memory that is shared by the

cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.

- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

- The two communications models are contrasted in the below diagram.



Communications models. (a) Shared memory. (b) Message passing.

- Both of the models just mentioned are common in operating systems, and many systems implement both.

- Message passing is useful for exchanging **smaller amounts of data**, because no conflicts need be avoided. Message passing is also easier to implement in a distributed system than shared memory.

- Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.

- In shared-memory systems, system calls are required only to establish shared memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel

is required.

IPC in Shared-Memory Systems

- Inter process communication using shared memory requires communicating processes to establish a region of shared memory.
- Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space.
- Normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas.
- The form of the data and the location are determined by these processes and are not under the operating system's control.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.
- To illustrate the concept of cooperating processes, let's consider the **producer–consumer problem**, which is a common paradigm for cooperating processes.
- A producer process **produces information** that is **consumed by** a consumer process. For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader.
- One solution to the producer–consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have a **buffer of items** that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item. The producer and consumer must be **synchronized,** so that the consumer does not try to consume an item that has not yet been produced.
- Two types of buffers can be used.

- 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.

- 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.

IPC in Message-Passing Systems

- Message passing provides a mechanism to allow processes to **communicate and to synchronize** their actions without sharing the same address space.

- It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

- A message-passing facility provides at least two operations:

  send(message) and **receive(message)**

- Messages sent by a process can be either fixed or variable in size. If only fixed-sized messages can be sent, the system-level implementation is simple.

- Variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler.

- 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 a variety of ways.

- Several methods for logically implementing a link and the send()/receive() operations:

  - Direct or indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering

Naming:

- Processes that want to communicate must have a way to refer to each

other.

- They can use either direct or indirect communication.

- Under **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication.
- In this scheme, the send() and receive() primitives are defined as:

  • send(P, message)—Send a message to process P.
  • receive(Q, message)—Receive a message from process Q.

- A communication link in this scheme has the following properties:

  - A link is established automatically 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 exactly one link.

- This scheme exhibits **symmetry in addressing**; that is, both the sender process and the receiver process must name the other to communicate.

- A variant of this scheme employs **asymmetry in addressing**. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send() and receive() primitives are defined as follows:

  - send(P, message)—Send a message to process P.
  - receive(id, message)—Receive a message from any process. The variable id is set to the name of the process with which communication has taken place.

- With **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. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox.

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

  • send(A, message)—Send a message to mailbox A.

• receive(A, message)—Receive a message from mailbox A.

- In this scheme, a communication link has the following properties:
  - 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, a number of different links may exist, with each link corresponding to one mailbox.

Synchronization:

- Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive.

- Message passing may be either **blocking or nonblocking**.
- **Blocking** is considered **synchronous**
- **Non-blocking** is considered **asynchronous**
- **send** and **receive** primitives may be either blocking or non-blocking.

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

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, the link cannot have any messages waiting in it. In this case, 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 queue is not full when a new message is sent, the message is placed in the queue, and the sender can continue execution without waiting. The link's capacity is finite, however. 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.

3. Assume the following processes arrive for execution at the time t=0 and also mentioned with the length of the CPU-burst time given in milli seconds.

JOB (ms)	BURST TIME (ms)	PRIORITY
P1	24	2
P2	3	2
P3	3	4

I. Give a Gantt chart illustrating the execution of these processes using FCFS, SJF, Round Robin(quantum=2) and Priority
II. Calculate the average waiting time and average turn around time for each of the above scheduling algorithm.

(CO2-K3)

FCFS Algorithm

NOTE:

1. **In this problem arrival time to be considered as t=0.**
2. **For this algorithm, the processes will be allocated in the order in which they have arrived, that is the job which came first will be allocated first and so on.**

Gantt chart:

Average Waiting Time & Turn Around Time:

NOTE:

1. **Waiting time=Process Allocation Time – Process Arrival Time**
2. **In this problem, all the processes arrive at t=0. So this arrival time should be considered for the calculation of waiting time.**

Job	Waiting Time (ms) =Process Allocation Time – Process Arrival Time	Turnaround Time (ms) =Waiting Time + Burst Time
P1	0-0=0	0+24=24
P2	24-0=24	24+3=27
P3	27-0=27	27+3=30
TOTAL	51	81

For FCFS, Average Waiting Time = 51/3=17 ms

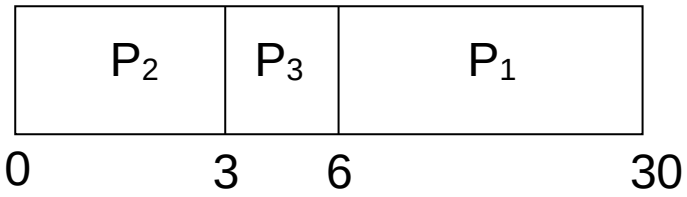
For FCFS, Average Turnaround Time = 81/3=27 ms

SJF (Non-Premptive) Algorithm

NOTE:

1. **In this problem arrival time for all process to be considered as 0.**
2. **For this algorithm, the processes with shortest burst time will be allocated first and so on. In this problem, two process P2 and P3 have shortest burst time. Considering P2 arrived before P3, P2 will be allocated first.**

Gantt chart:



Average Waiting Time & Turn Around Time:

NOTE:

1. **Waiting time=Process Allocation Time – Process Arrival Time**
2. **In this problem, all the processes arrive at t=0. So this arrival time should be considered for the calculation of waiting time.**

Job	Waiting Time (ms) =Process Allocation Time – Process Arrival Time	Turnaround Time (ms) =Waiting Time + Burst Time
P1	6-0=6	6+24=30
P2	0-0=0	0+3=3
P3	3-0=3	3+3=6
TOTAL	9	39

For SJF, Average Waiting Time = 9/3=3 ms

For SJF, Average Turnaround Time = 39/3=13 ms

Priority Scheduling (Non Preemptive)

NOTE:

1. **In this problem arrival time for all process to be considered as 0.**
2. **For this algorithm, the processes with highest priority will be allocated first. (If not mentioned in the question paper, considers the lowest value as highest priority.)**
3. **Process P1 and P2 have the same higher priority. Considering P2 arrived first, P1 will be allotted first.**

Gantt chart:

Average Waiting Time & Turn Around Time:

NOTE:

1. **Waiting time=Process Allocation Time – Process Arrival Time**
2. **In this problem, all the processes arrive at t=0. So this arrival time should be considered for the calculation of waiting time.**

Job	Waiting Time (ms) =Process Allocation Time – Process Arrival Time	Turnaround Time (ms) =Waiting Time + Burst Time
P1	0-0=0	0+24=24
P2	24-0=24	24+3=27
P3	27-0=27	27+3=30
TOTAL	51	81

For Priority Scheduling, Average Waiting Time = 51/3=17 ms

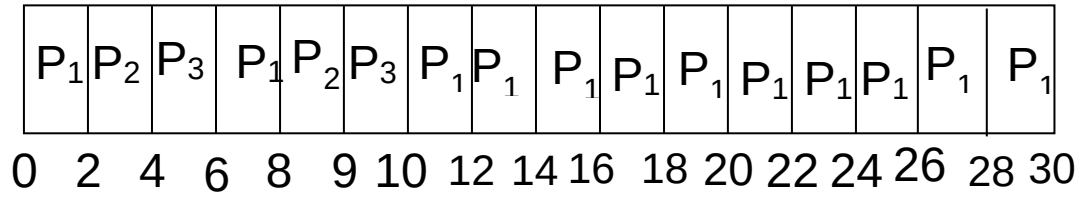
For Priority Scheduling Turnaround Time = 81/3=27 ms

Round Robin Algorithm

NOTE:

1. **In this problem, all processes are arrived in time t=0.**
2. **For this algorithm, the processes will be preempted and allocated based on the given quantum value. (Given quantum is 2 ms).**
3. **Process P1 will be allotted first and executed till the quantum value expires.**
4. **Once P1 finishes its quantum value it is preempted and the chance will be given to the next process P2. P2 will execute for the given quantum and preempted for giving chance to the next process P3. Like this, the allotment continued till all the processes complete its execution.**

Gantt chart:



Average Waiting Time & Turn Around Time:

NOTE:

1. **Waiting time=Process Allocation Time – Process Arrival Time**
2. **In this problem, all the processes arrive at t=0. So this arrival time should be considered for the calculation of waiting time.**

Job	Waiting Time (ms) =Process Allocation Time – Process Arrival Time	Turnaround Time (ms) =Waiting Time + Burst Time
P1	(10-8)+ (6-2) + (0-0) = 6	6+24=30
P2	(8-4) + (2-0)=6	6+3=9
P3	(9-6)+(4—0)=7	7+3=10
TOTAL	19	49

For Round Robin, Average Waiting Time = 19/3=6.3 ms

For Round RobinTurnaround Time = 49/3=16.3 ms

4. Assume the following processes arrive for execution at the time indicated and also mention with the length of the CPU-burst time given in milli seconds.

JOB	BURST TIME (ms)	PRIORITY	ARRIVAL TIME (ms)
P1	6	2	0
P2	2	2	1
P3	3	4	1
P4	1	1	2

P5	2	3	2

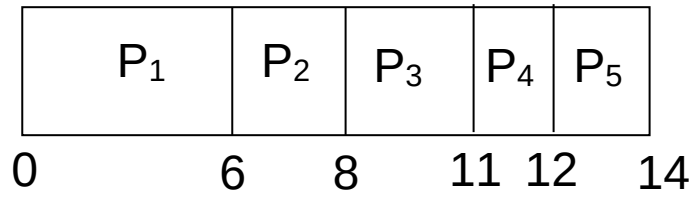
I. Give a Gantt chart illustrating the execution of these processes using FCFS, SJF, SRTF, Round Robin(quantum=1) and Priority (Preemptive and Non Preemptive)
II. Calculate the average waiting time and average turn around time for each of the above scheduling algorithm. **(CO2-K3)**

FCFS Algorithm

NOTE:

1. **In this problem arrival time for all process to be considered. Note here all processes are not arrived in time t=0.**
2. **For this algorithm, the processes will be allocated in the order in which they have arrived, that is the job which came first will be allocated first and so on.**

Gantt chart:



Average Waiting Time & Turn Around Time:

NOTE:

1. **Waiting time=Process Allocation Time – Process Arrival Time**
2. **In this problem, the process P1 arrives at t=0. Other processes P2 & P3 arrived at 1ms, and P4&P5 arrived at 2ms. So this arrival time should be considered for the calculation of waiting time.**

Job	Waiting Time (ms) =Process Allocation Time – Process Arrival Time	Turnaround Time (ms) =Waiting Time + Burst Time
P1	0-0=0	0+6=6
P2	6-1=5	5+2=7
P3	8-1=7	7+3=10
P4	11-2=9	9+1=10
P5	12-2=10	10+2=12
TOTAL	31	45

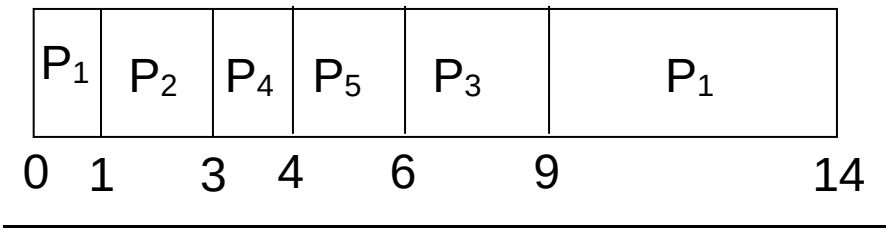
For FCFS, Average Waiting Time = 31/5=6.2 ms

For FCFS, Average Turnaround Time = 45/5=9 ms

SJF (Premptive) Algorithm) / SRTF (Shortest Remaining Time First)

NOTE:

1. **In this problem arrival time for all process to be considered. Note here all processes are not arrived in time t=0.**
2. **For this algorithm, the processes with shortest remaining burst time will be allocated first.**
3. **At t=0, only one process is available that is P1, that will be allotted first and executed till the next process arrives.**
4. **At t=1, next process P2 with burst time 2 ms and P3 with burst time 3 come and their burst time is compared with the remaining burst time of P1 (6-1=5ms). As P2 has less burst time, P1 preempted and P2 permitted to execute till the next process arrives.**
5. **At t=2, next process P4 with burst time 1ms and P5 with burst time 2ms come and their burst time is compared with the remaining burst time of P1 and P2. Here P1 has 5ms and P2 has 1ms (2-1). Now, as both P2 and P4 have same burst time, FCFS will be used and P2 (as it came first) permitted to execute till the next process arrives.**
6. **Here in this problem no other process comes after t=2, that means all the processes are arrived. So the selection should be made among the available jobs based on the remaining burst time.**
7. **Here P2 completed its entire burst time. Selection will be among P1=5ms, P3=3ms, P4=1ms and P5=2ms. The next allocation sequence will be P4, P5, P3 and P1.**



Average Waiting Time & Turn Around Time:

NOTE:

1. **Waiting time=Process Allocation Time – Process Arrival Time**
2. **In this problem, the process P1 arrives at t=0. Other processes P2 & P3 arrived at 1ms, and P4&P5 arrived at 2ms. So this arrival time should be considered for the calculation of waiting time.**

Job	Waiting Time (ms) =Process Allocation Time – Process Arrival Time	Turnaround Time (ms) =Waiting Time + Burst Time
P1	(9-1)+(0-0)=8	8+6=14
P2	1-1=0	0+2=2
P3	6-1=5	5+3=8
P4	3-2=1	1+1=2
P5	4-2=2	2+2=4
TOTAL	16	30

For SJF(Preemptive)/ SRTF, Average Waiting Time = 16/5=3.2 ms

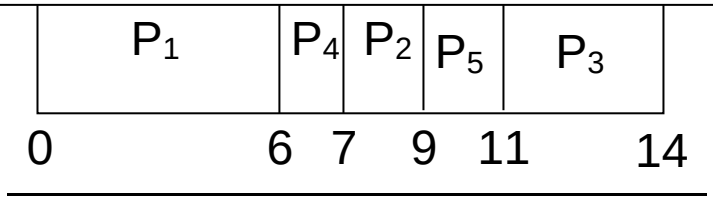
For SJF(Preemptive)/ SRTF, Average Turnaround Time = 30/5=6 ms

SJF (Non-Premptive) Algorithm

NOTE:

1. **In this problem arrival time for all process to be considered. Note here all processes are not arrived in time t=0.**
2. **For this algorithm, the processes with shortest burst time will be allocated first.**
3. **At t=0, only one process is available that is P1, that will be allotted first and executed till it finishes its execution as it is non-preemptive.**
4. **Once P1 finishes its completion the selection will be made among the remaining processes P2=2ms, P3=3ms, P4=1ms and P5=2ms.**
5. **The allocation will be based on the size of the process, that is the shortest job (P4) will be allotted next . After its completion, the comparison will be made between P2 & P5 since both are having equal burst time. Now by using FCFS, P2(as it came first) will be allotted next. Then P5 and P3 will be allotted.**

G



Average Waiting Time & Turn Around Time:

NOTE:

1. **Waiting time=Process Allocation Time – Process Arrival Time**
2. **In this problem, the process P1 arrives at t=0. Other processes P2 & P3 arrived at 1ms, and P4&P5 arrived at 2ms. So this arrival time should be considered for the calculation of waiting time.**

Job	Waiting Time (ms) =Process Allocation Time – Process Arrival Time	Turnaround Time (ms) =Waiting Time + Burst Time
P1	0-0=0	0+6=6
P2	7-1=6	6+2=8
P3	11-1=10	10+3=13
P4	6-2=4	4+1=5
P5	9-2=7	7+2=9
TOTAL	27	41

For SJF(Non-Preemptive), Average Waiting Time = 27/5=5.4ms

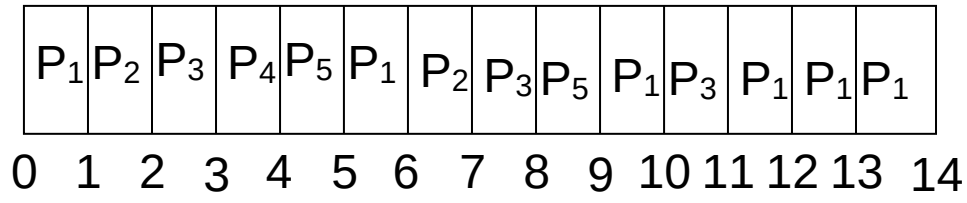
For SJF(Non-Preemptive), Average Turnaround Time = 41/5=8.2 ms

Round Robin Algorithm

NOTE:

1. **In this problem, arrival time for all process to be considered. Note here all processes are not arrived in time t=0.**
2. **For this algorithm, the processes will be preempted and allocated based on the given quantum value. (Given quantum is 1).**
3. **At t=0, only one process is available that is P1, that will be allotted first and executed till the quantum value expires.**
4. **Once P1 finishes its quantum value it is preempted and the chance will be given to the next arrived process P2. P2 will execute for the given quantum and preempted**

Gantt chart:



Average Waiting Time & Turn Around Time:

NOTE:

1. **Waiting time=Process Allocation Time – Process Arrival Tim**
2. **In this problem, the process P1 arrives at t=0. Other processes P2 & P3 arrived at 1ms, and P4&P5 arrived at 2ms. So this arrival time should be considered for the calculation of waiting time.**

Job	Waiting Time (ms) =Process Allocation Time – Process Arrival Time	Turnaround Time (ms) =Waiting Time + Burst Time
P1	(11-10)+(9-6)+(5-1)+(0-0)=8	8+6=14
P2	(6-2)+(1-1)=4	4+2=6
P3	(10-8)+(7-3)+(2-1)=7	7+3=10
P4	3-2=1	1+1=2
P5	(8-5)+(4-2)=5	5+2=7
TOTAL	25	39

For Round Robin Scheduling, Average Waiting Time = 25/5=5ms

For Round Robin Scheduling, Average Turnaround Time = 39/5=7.8 ms

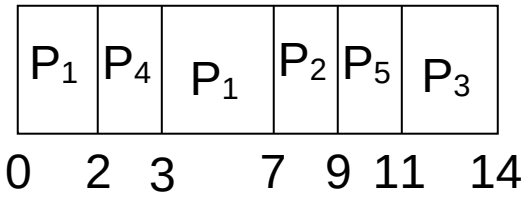
Priority Scheduling (Preemptive)

NOTE:

1. **In this problem arrival time for all process to be considered. Note here all processes are not arrived in time t=0.**
2. **For this algorithm, the processes with highest priority will be allocated first. (If not mentioned in the question paper, considers the lowest value as highest priority.)**
3. **At t=0, only one process is available that is P1, that will be allotted first and executed till the next process arrives.**
4. **At t=1, next process P2 with priority 2 and P3 with priority 4 has come and their priorities are compared with the priority of P1 (2). Here P1& P2 have same priorities, and by FCFS P1 (as it came first) permitted to continue the execution till the next process arrives.**
5. **At t=2, next process P4 with priority 1 and P5 with priority 3 has come and their priorities are compared with the priorities of P1, P2 & P3. Here P4 has highest priority, P1 preempted from its execution and P4 will be permitted execute till the next process arrives.**
6. **Here in this problem no other process comes after t=2, that means all the processes are arrived. So the selection should be made among the available jobs based on their priorities**
7. **Here P4 completed its entire burst time. Selection will be among P1=2, P2=2, P3=4 and P5=3. The next allocation sequence will be P1(as it came first), P2, P5 and P3.**

G

By considering the lowest value represents the highest priority



Average Waiting Time & Turn Around Time:

NOTE:

1. **Waiting time=Process Allocation Time – Process Arrival Time**
2. **In this problem, the process P1 arrives at t=0. Other processes P2 & P3 arrived at 1ms, and P4&P5 arrived at 2ms. So this arrival time should be considered for the calculation of waiting time.**

Job	Waiting Time (ms) =Process Allocation Time – Process Arrival Time	Turnaround Time (ms) =Waiting Time + Burst Time
P1	(3-2)+(0-0)=1	1+6=7
P2	(7-1)=6	6+2=8
P3	(11-1)=10	10+3=13
P4	2-2=0	0+1=1
P5	9-2=7	7+2=9
TOTAL	24	38

For Priority Scheduling(Preemptive), Average Waiting Time = 24/5=4.8ms

For Priority Scheduling(Preemptive), Average Turnaround Time = 38/5=7.6 ms

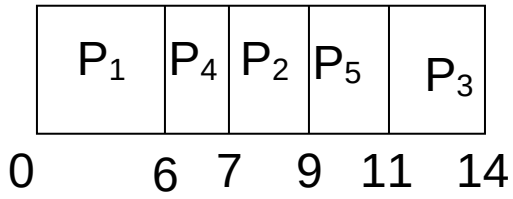
Priority Scheduling (Non Preemptive)

NOTE:

1. **In this problem arrival time for all process to be considered. Note here all processes are not arrived in time t=0.**
2. **For this algorithm, the processes with highest priority will be allocated first. (If not mentioned in the question paper, considers the lowest value as highest priority.)**
3. **At t=0, only one process is available that is P1, that will be allotted first and executed till it finishes its execution as it is non-preemptive.**
4. **Once P1 finishes its completion the selection will be made among the remaining processes P2=2, P3=4, P4=1 and P5=3.**
5. **The allocation will be based on the priority of the process, that is the highest priority process (P4) will be allotted next . After its completion, the comparison will be made between remaining processes and the allocation will be made based on the priority.**
6. **The next allocation sequence is P2,P5.P3.**

G

B



Average Waiting Time & Turn Around Time:

NOTE:

1. **Waiting time=Process Allocation Time – Process Arrival Tim**
2. **In this problem, the process P1 arrives at t=0. Other processes P2 & P3 arrived at 1ms, and P4&P5 arrived at 2ms. So this arrival time should be considered for the calculation of waiting time.**

Job	Waiting Time (ms) =Process Allocation Time – Process Arrival Time	Turnaround Time (ms) =Waiting Time + Burst Time
P1	(0-0)=0	0+6=6
P2	(7-1)=6	6+2=8
P3	(11-1)=10	10+3=13
P4	6-2=4	4+1=5
P5	9-2=7	7+2=9
TOTAL	27	41

For Priority Scheduling(Non-Preemptive), Average Waiting Time = 27/5=5.4ms

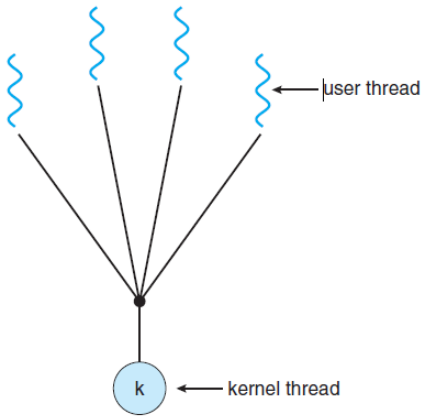
For Priority Scheduling (Non Preemptive), Average Turnaround Time = 41/5=8.2 ms

5. Explain about Multithreading Models (CO2-K2)

- Support for threads may be provided either at the user level, for user threads, or by the kernel, for kernel threads.
- User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.
- Virtually all contemporary operating systems—including Windows, Linux, Mac OS X, and Solaris— support kernel threads.
- Ultimately, a relationship must exist between user threads and kernel threads. In this section, we look at three common ways of establishing such a relationship: the many-to-one model, the one-to-one model, and the many-to many model.

Many-to-One Model

- The many-to-one 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.
- However, the entire process will block if a thread makes a blocking system call.
- Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.
- Green threads—a thread library available for Solaris systems and adopted in early versions of Java—used the many-to-one model.
- However, very few systems continue to use the model because of its inability to take advantage of multiple processing cores.

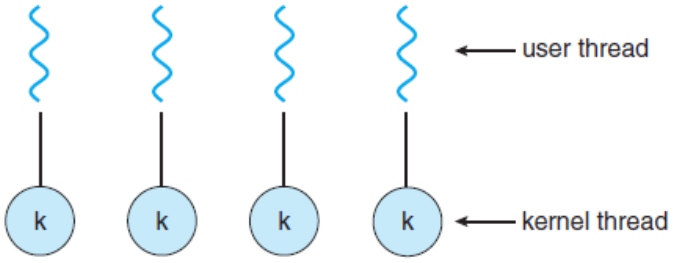


Many-to-one model.

One-to-One Model

- The one-to-one model maps each user thread to a kernel thread.
- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- It also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
- Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.

- Linux, along with the family of Windows operating systems, implement the one-to-one model.

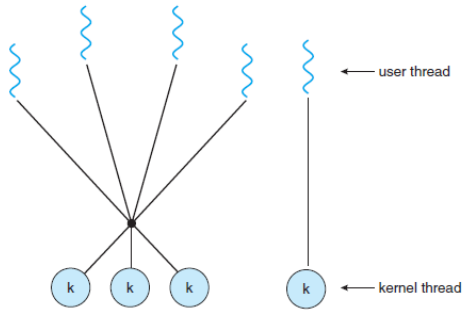


One-to-one model.

Many-to-Many Model

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a single processor).
- Let's consider the effect of this design on concurrency. Whereas the many to- one model allows the developer to create as many user threads as the developer wishes, it does not result in true concurrency, because the kernel can schedule only one thread at a time.
- The one-to-one model allows greater concurrency, but the developer has to be careful not to create too many threads within an application (and in some instances may be limited in the number of threads she can create).
- The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.
- One variation on the many-to-many model still multiplexes many user level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the **two-level model**.
- The Solaris operating system supported the two-level model in versions older than

Solaris 9. However, beginning with Solaris 9, this system uses the one-to-one model.



Two-level model.

6. Explain in detail about the threading issues. (CO2-K2)

- Semantics of fork() and exec() system calls.
- Thread cancellation.
- Signal handling
- Thread pools
- Thread specific data

The fork() and exec() system calls:

- In multithreaded program the semantics of the *fork* and *exec* system calls change.

- Does *fork()* duplicate only the calling thread or all threads?
  - Some UNIX systems have chosen to have two versions of *fork*, one that duplicated all threads and another that duplicates only the thread that invoked the *fork* system call.

- If the *exec* is called immediately after forking, then duplicating all threads in unnecessary, as the program specified in the parameters to *exec* will replace the process

Cancellation:

- Terminating a thread before it has finished
- Two general approaches:

- **Asynchronous cancellation** terminates the target thread immediately
- **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Signal handling:

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- Options:
  1. Deliver the signal to the thread to which the signal applies
  2. Deliver the signal to every thread in the process
  3. Deliver the signal to certain threads in the process
  4. Assign a specific thread to receive all signals for the process

Thread pools:

- Create a number of threads in a pool where they await work
- Advantages:
  1. Usually slightly faster to service a request with an existing thread than create a new thread
  2. Allows the number of threads in the application(s) to be bound to the size of the pool

Thread specific data:

- Allows each thread to have its own copy of data
- Useful when there is no control over the thread creation process (i.e., when using a thread pool)

7. What is critical section problem and explain the algorithms for two process solutions and multiple process solutions? (CO2-K2)

- *n* processes all competing to use some shared data
- Each process has a code segment, called ***critical section*****,** in which the shared data is accessed.

- The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.
- The ***critical-section problem*** is to design a protocol that the processes can use to cooperate

Solution to Critical-Section Problem:

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

2. **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.

3. **Bounded Waiting**. 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.

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the *n* processes.

Initial Attempts to Solve Problem:

- Each process must request permission to enter its critical section.
- The section of code implementing this request is the **entry section**.
- The critical section may be followed by an **exit section**.
- The remaining code is the **remainder section**.

- Only 2 processes, *P*0 and *P*1
- General structure of process *Pi* (other process *Pj*)

```
do {
        entry section
                critical section
        exit section
                reminder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.

Algorithm 1:

- Shared variables:
  - **int turn**
  - initially **turn = 0**
  - **turn - i** $\Rightarrow$ *Pi* can enter its critical section

- Process *Pi*

```
do {
        while (turn != i) ;
                critical section
        turn = j;
                reminder section
} while (1);
```

Our first approach is to let the processes share a common integer variable turn initialized to 0 (or 1). If turn == i, then process Pi is allowed to execute in its critical section. The structure of process Pi is shown.

This solution ensures that only one process at a time can be in its critical section. However, it **does not satisfy the progress requirement**, since it requires strict alternation of processes in the execution of the critical section. For example, if turn == 0 and P_1 is ready to enter its critical section, P_1 cannot do so, even though Po may be in its remainder section.

Algorithm 2:

The problem with algorithm 1 is that it does not retain sufficient information about the state of each process; it remembers only which process is allowed to enter its critical section. To remedy this problem, we can replace the variable t u r n with the following array:

```
boolean flag [2] ;
```

The elements of the array are initialized to false. If flag [i] is true , this value indicates that P is

ready to enter the critical section. The structure of process Pi is shown below.

In this algorithm, process Pi first sets f l a g [il to be t r u e , signaling that it is ready to enter its critical section. Then, Pi checks to verify that process Pj is not also ready to enter its critical_section.

- Shared variables
  - **boolean flag[2];**
  - initially **flag [0] = flag [1] = false.**
  - **flag [i] = true** ⇒ *Pi* ready to enter its critical section

- Process *Pi*

```
do {

        flag[i]=true;
        while (flag[j]) ;

                critical section

        flag [i] = false;

                remainder section

} while (1);
```

The structure of process Pi in algorithm 2. not also ready to enter its critical section. If Pi were ready, then Pi would wait until Pihad indicated that it no longer needed to be in the critical section (that is, until flag [j] was false) . At this point, Pi would enter the critical section. On exiting the critical section, Pi would set flag [i] to be false, allowing the other process (if it is waiting) to enter its critical section.

In this solution, the mutual-exclusion requirement is satisfied. Unfortunately, the progress requirement is not met. To illustrate this problem, we consider the following execution sequence:

To: Po sets flag [0] = t r u e

T_I: P_1sets flag [1] = t r u e

Now Po and P_1 are looping forever in their respective while statements. This algorithm is crucially dependent on the exact timing of the two processes. The sequence could have been derived in an environment where there are several processors executing concurrently, or where an interrupt (such as a timer interrupt) occurs immediately after step To is executed, and the CPU is switched from one process to another.

Note that switching the order of the instructions for setting flag [i], and testing the value of a flag[j], will not solve our problem. Rather, we will have a situation where it is possible for both processes to be in the critical section at the same time, violating the mutual-exclusion requirement.

Algorithm 3:

By combining the key ideas of algorithm 1 and algorithm 2, we obtain a correct solution to the critical-section problem, where all three requirements are met. The processes share two variables:

boolean f l a g [2];

int turn;

Initially flag [0] = flag [1] = false , and the value of turn is immaterial (but is either 0 or 1).

To enter the critical section, process Pi first sets flag [i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved,

2. The progress requirement is satisfied,

3. The bounded-waiting requirement is met.

To prove property 1, we note that each P, enters its critical section only if either flag [j] == false or turn == i.

The structure of process Pi is shown below

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

remainder section

} **while (1);**

- To prove properties 2 and 3, we note that a process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn == j; this loop is the only one.

- Meets all three requirements; solves the critical-section problem for two processes.

Critical section for multiple processes - Bakery Algorithm:

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.

- If processes *Pi* and *Pj* receive the same number, if *i* < *j*, then *Pi* is served first; else *Pj* is served first.

- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

- Notation <≡ lexicographical order (ticket #, process id #)
  - (*a,b*) < *c,d*) if *a* < *c* or if *a* = *c* and *b* < *d*
  - max (*a0*,…, *an*-1) is a number, *k*, such that *k* ≥ *a*i for *i* - 0, …, *n* – 1

- Shared data

  boolean choosing[n];

  int number[n];

  Data structures are initialized to **false** and **0** respectively

```
do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
```

```
        while (choosing[j]) ;
        while ((number[j] != 0) && (number[j,j] < number[i,i])) ;
}
        critical section
number[i] = 0;
        remainder section
} while (1);
```

8. Explain about Semaphores. (CO2-K2)

The solutions to the critical-section problem presented are not easy to generalize to more complex problems. To overcome this difficulty, use a **synchronization tool** called a **semaphore**. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait and signal** . These operations were originally termed P (for wait) and V (for signal)

The classical definition of **wait** in pseudocode is

```
wait(S) {
  while (S < 0)
    ;/ / no-op
s--;
        }
```

The classical definitions of **signal** in pseudocode is

```
Signal(S)
        {
        S++;
        }
```

Usage

Operating systems often distinguish between **counting and binary semaphores**. The value of a counting semaphore can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

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 performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0.

In process P1, we insert the statements

```
S1;
signal(synch);
```

In process *P*2, we insert the statements

```
wait(synch);
S2;
```

Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed.

IMPLEMENTATION

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 the waiting state to the ready state. The process is then placed in the ready queue.
To implement semaphores under this definition, we define a semaphore as follows:

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

```
} semaphore;
```

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.

Now, the wait() semaphore operation can be defined as

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

and the signal() semaphore operation can be defined as

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

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

Note that in this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait() operation.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue. In general, however, the list can use any queueing strategy. Correct usage of semaphores does not depend on a particular queueing strategy for the semaphore lists.

It is critical that semaphore operations be executed atomically. We must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time. This is a critical-section problem; and in a single-processor environment, we can solve it by simply inhibiting interrupts during the time the wait() and signal() operations are executing.

This scheme works in a single-processor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are reenabled and the scheduler can regain control.

In a multiprocessor environment, interrupts must be disabled on every processor. Otherwise, instructions from different processes (running on different processors) may be interleaved in some arbitrary way. Disabling interrupts on every processor can be a difficult task and furthermore can seriously diminish performance. Therefore, SMP systems must provide alternative locking techniques— such as compare and swap() or spinlocks—to ensure that wait() and signal() are performed atomically.

It is important to admit that we have not completely eliminated busy waiting with this definition of the wait() and signal() operations. Rather, we have moved busy waiting from the entry section to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the wait() and signal() operations, and these sections are short (if properly coded, they should be no more than about ten instructions). Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time. An entirely different situation exists with application programs whose critical sections may be long (minutes or even hours) or may almost always be occupied. In such cases, busy waiting is extremely inefficient.

Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be **deadlocked**.

To illustrate this, consider a system consisting of two processes, *P*0 and *P*1, each accessing two semaphores, S and Q, set to the value 1:

```
P0              P1
wait(S);        wait(Q);
wait(Q);        wait(S);
. .
. .
. .
signal(S);      signal(Q);
signal(Q);      signal(S);
```

Suppose that *P*0 executes wait(S) and then *P*1 executes wait(Q).When *P*0 executes wait(Q), it must wait until *P*1 executes signal(Q). Similarly, when *P*1 executes wait(S), it must wait until *P*0 executes signal(S).
Since these signal() operations cannot be executed, *P*0 and *P*1 are deadlocked. We say that a set of processes is in a deadlocked state when every process in the set is waiting

for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. Another problem related to deadlocks is **indefinite blocking** or **starvation**, a situation inwhich processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

9. Explain in detail the classic problems of synchronization. (CO2-K2)

Bounded-Buffer Problem:

It is commonly used to illustrate the power of synchronization primitives. Assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers, respectively. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The code for the producer process is shown, the code for the consumer process is also shown. Note the symmetry between the producer and the consumer. Interpret this code as the producer producing full buffers for the consumer, or as the consumer producing empty buffers for the producer.

In our problem, the producer and consumer processes share the following

data structures:

int n;

semaphore mutex = 1;

semaphore empty = n;

Bounded-Buffer Problem Producer Process:

do {

…

produce an item in **nextp**

…

```
            wait(empty);
            wait(mutex);
                    …
            add nextp to buffer
                    …
            signal(mutex);
            signal(full);
        } while (1);
```

Bounded-Buffer Problem Consumer Process:

```
    do {
            wait(full)
            wait(mutex);
                    …
            remove an item from buffer to nextc
                    …
            signal(mutex);
            signal(empty);
                    …
            consume the item in nextc
                    …
        } while (1);
```

Readers-Writers Problem:

A data object (such as a file or record) is to be shared among several concurrent processes. Some of these processes may want only to read the content of the shared

object, whereas others may want to update (that is, to read and write) the shared object. We distinguish between these two types of processes by referring to those processes that are interested in only reading as readers, and to the rest as writers. Obviously, if two readers access the shared data object simultaneously, no adverse effects will result.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared object. This synchronization problem is referred to as the readers-writers problem.

- Shared data

  semaphore mutex, wrt;

  Initially

  mutex = 1, wrt = 1, readcount = 0

Readers-Writers Problem Writer Process:

```
wait(wrt);
        …
    writing is performed
        …
signal(wrt);
```

Readers-Writers Problem Reader Process:

```
wait(mutex);
readcount++;
if (readcount == 1)
        wait(rt);
signal(mutex);
```

…

reading is performed

…

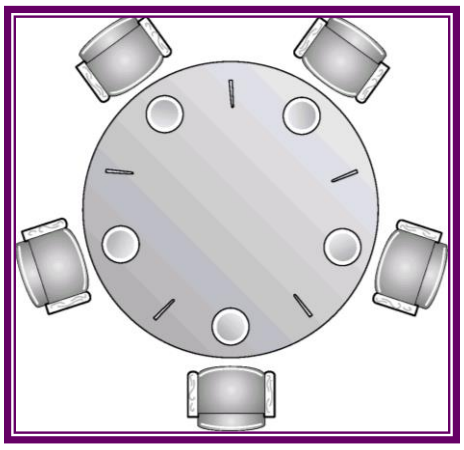
```
wait(mutex);
readcount--;
if (readcount == 0)
        signal(wrt);
signal(mutex):
```

Dining-Philosophers Problem:

Consider five philosophers who spend their lives thinking and eating. The philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (shown in Figure given below) When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks

- Shared data
  semaphore chopstick[5];

  Initially all values are 1



The situation of dining philosophers

The situation of the dining philosophers is blocking her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

The dining-philosophers problem is considered a classic synchronization problem, neither because of its practical importance nor because computer scientists dislike philosophers, but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock and starvation-free manner.

One simple solution is to represent each chopstick by a semaphore. A philosopher tries to grab the chopstick by executing a wait operation on that semaphore; she releases her chopsticks by executing the signal operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick [5];

where all the elements of chopstick are initialized to 1. The structure of philosopher i is shown below.

- Philosopher *i*:

```
do {
        wait(chopstick[i])
        wait(chopstick[(i+1) % 5])
                …
        eat
                …
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
                …
        think
                …
} while (1);
```

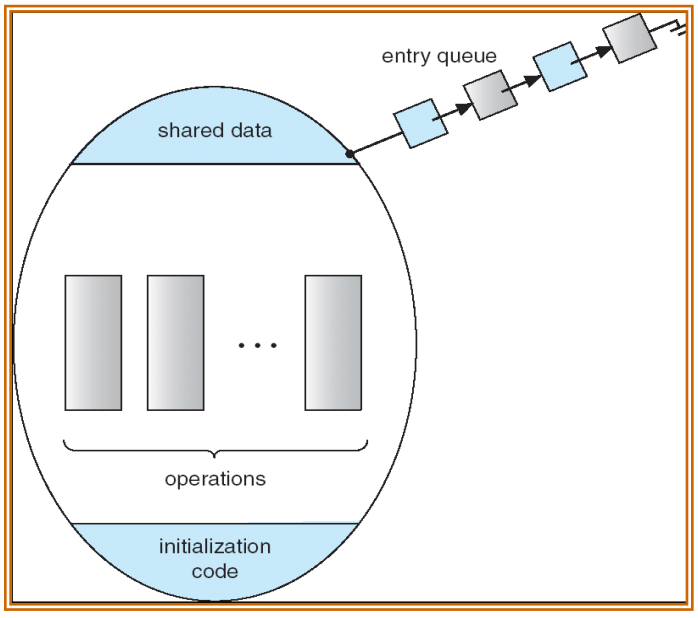
10. Explain about Monitor. (CO2-K2)

MONITORS: A monitor is characterized by a set of programmer-defined operators. The representation of a monitor type consists of declarations of variables whose values define the state of an instance of the type, as well as the bodies of procedures or functions that implement the operations on the type.

Syntax:

```
monitor monitor-name
{
        // shared variable declarations
        procedure P1 (…) { …. }
                …
        procedure Pn (…) {……}
    Initialization code ( ….) { … }
                …
        }
}
```

The monitor construct ensures that only one process at a time can be active within the monitor.



Schematic view of a monitor

A programmer who needs to write her own tailor-made synchronization scheme can define one or more variables of type condition:

Condition x,y;

The operations that can be performed on a condition variable are wait and signal.

The operation, x.wait() means that the process invoking this operation is suspended until another process invokes

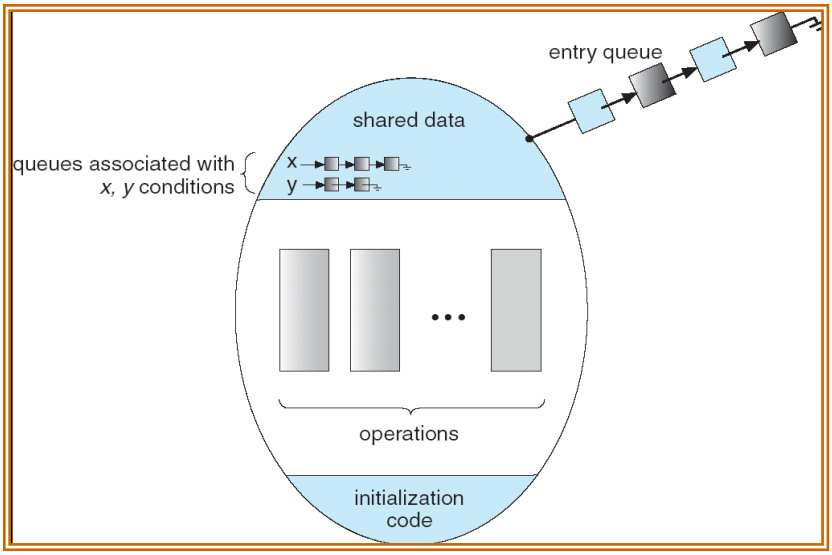
x.signal();

The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

Consider x.signal() operation is invoked by a process P, there is a suspended process Q associated with condition x. If the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q will be active simultaneously within the monitor.

Two possibilities exists:

1. P either waits until Q leaves the monitor, or waits for another condition.
2. Q either waits until P leaves the monitor, or waits for another condition.



Monitor with condition variable

11. Explain in detail the approaches for deadlock prevention and avoidance. (CO2-K2)

- 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 is a set of methods for ensuring that at least one the necessary conditions cannot hold.
- Deadlock avoidance, the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime.

Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, it can prevent the occurrence of a deadlock.

- **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.

- **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.
    - In the next approach, if the requested resource is not available, and if they may allocated to some other process that is waiting for additional resources, such resources will be preempted

- **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 safe sequence of all processes.
- Sequence <*P*1, *P*2, …, *Pn*> is safe if for each *P*i, the resources that *Pi* can still request can be satisfied by currently available resources + resources held by all the *Pj*, with *j<i*.
    - If Pi resource needs are not immediately available, then Pi can wait until all Pj have finished.
    - When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate.
    - When Pi terminates, Pi+1 can obtain its needed resources, and so on.

Basic Facts

- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensures that a system will never enter an unsafe state.

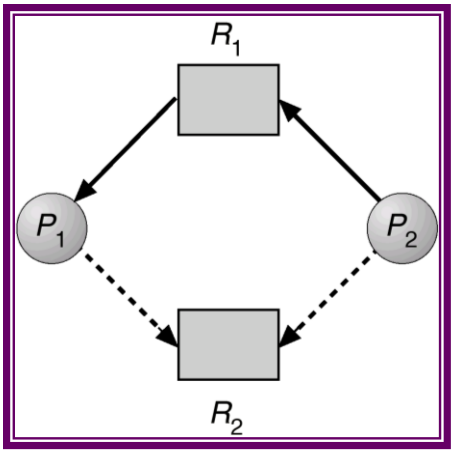
Resource-Allocation Graph Algorithm

- In addition to the request and assignment edges, we introduce a new type of edge, called a claim edge. A claim edge $P_i \rightarrow R_j$ indicates that process Pi may request resource R_j at some time in the future.
- This edge resembles a request edge in direction, but is represented by a dashed line. When process P_i requests resource R_j, the claim edge $P_i \rightarrow R_j$ is converted to a request edge. Similarly, when a resource R_j is released by Pi, assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_i$.
- Suppose that process P_i requests resource R_j. The request can be granted only

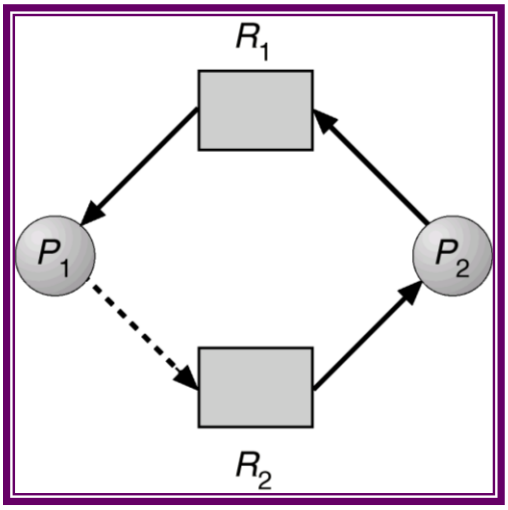
if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource-allocation graph.

- An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of processes in the system.

Resource-Allocation Graph For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph



Banker's Algorithm

- Suitable for resources with multiple instances.
- Each process must a priori claim maximum use. This number may not exceed the total number of resources in the system

- 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 *Rj* available.
- *Max: n x m* matrix. If *Max* [*i,j*] = *k*, then process *Pi* may request at most *k* instances of resource type *Rj*.
- *Allocation: n* x *m* matrix. If Allocation[*i,j*] = *k* then *Pi* is currently allocated *k* instances of *Rj.*
- *Need: n* x *m* matrix. If *Need*[*i,j*] = *k*, then *Pi* may need *k* more instances of *Rj* 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* = 1,2,3, …, *n.*

2. Find and *i* such that both:

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

   (b) *Needi* ≤ *Work*

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

3. *Work* = *Work* + *Allocationi*
   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:

Request = request vector for process *Pi*. If *Requesti* [*j*] = *k* then process *Pi* wants *k* instances of resource type *Rj*.

1. If *Requesti* ≤ *Needi* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If *Requesti* ≤ *Available*, go to step 3. Otherwise *Pi* must wait, since resources are not available.
1. Pretend to allocate requested resources to *Pi* by modifying the state as follows:

   Available = Available -Requesti;

   Allocationi = Allocationi + Requesti;

   Needi = Needi – Requesti;;

   - *If safe ⇒ the resources are allocated to Pi.*
   - *If unsafe ⇒ P*i *must wait, and the old resource-allocation state is restored*

12. Consider the following snapshot of a system:

Process	Allocation	Max	Available
	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	1 5 2 0
P1	1 0 0 0	1 7 5 0	
P2	1 3 5 4	2 3 5 6	
P3	0 6 3 2	0 6 5 2	
P4	0 0 1 4	0 6 5 6	

Answer the following questions applying the banker's algorithm*

a. What is the content of the matrix *Need*? Is the system in a safe state?

b. If a request from process P1 arrives for (0, 4, 2, 0), can the request be granted immediately?

(CO2-K3)

(a)Need Matrix

Need matrix can be computed by Max-Allocation

Process	Need
	A B C D
P0	0 0 0 0
P1	0 7 5 0
P2	1 0 0 2
P3	0 0 2 0
P4	0 6 4 2

According to Banker's safety algorithm,

Step 1:

work =available

work=(1,5,2,0)

finish[i]=false.

Step 2:

Check, Need≤ Work and Finish[i]=false.

For P0,

(0,0,0,0) ≤(1,5,2,0) ⇒True. Both conditions are satisfied and proceed to step 3.

Step 3:

Work=work+allocation

New work(available) = (1,5,2,0)+(0,0,1,2) = (1,5,3,2)

Finish[P0]=True

Next P1 will be considered

Finish[P1]=False

(0,**7**,5,0) ≤(1,**5**,3,2) ⇒ False, since the need of reference B (7) is greater that the work (5)

So Finish[P1]=false

Next P2 will be considered

Finish[P2]=False

(**1**,0,0,**2**) ≤(**1**,5,3,**2**) ⇒True.

Both conditions are satisfied and proceed to step 3.

Step 3:

Work=work+allocation

New work(available) = (1,5,3,2)+(1,3,5,4) = (2,8,8,6)

Finish[P2]=True

Next P3 will be considered

finish[P3]=false

(0,0,2,0) ≤(2,8,8,6) ⇒True. Both conditions are satisfied.

New Work(available)=(2,8,8,6)+(0,6,3,2)=(2,14,11,8)

Finish[P3]=True

Now P4 will be considered

finish[P4]=false

(0,6,4,2) ≤(2,14,11,8) ⇒True. Both conditions are satisfied

New work(available)=(2,14,11,8)+(0,0,1,4)=(2,14,12,12)

Finish[P4]=True

Now finish[P1] is false

Now P1 will be considered

(0,7,5,0) ≤(2,14,12,12) ⇒ True. Both conditions are satisfied

New work(available)= (2,14,12,12)+(1,0,0,0)=(3,14,12,12)

Finish[P1]=True

Now the finish[i] of all the process are true and the sequence is

<P0,P2,P3,P4,P1> which is the safe sequence.

(b) **Resource Request Algorithm**

Step 1

If *Requesti* ≤ *Needi* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

P1(0, 4, 2, 0) ≤(0,7,5,0) ⇒True

Step 2

If *Requesti* ≤ *Available*, go to step 3. Otherwise *Pi* must wait, since resources are not available.

P1(0, 4, 2, 0) ≤(1,5,2,0) ⇒True

Step 3

Available = Available -Request$_i$;

Allocation$_i$ = Allocationi + Request$_i$;

Need$_i$ = Need$_i$ – Request$_i$

Process	Allocation	Max	Available
	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	**1 1 0 0**
P1	**1 4 2 0**	1 7 5 0	
P2	1 3 5 4	2 3 5 6	
P3	0 6 3 2	0 6 5 2	

P4 0 0 1 4 0 6 5 6

Process	Need
	A B C D
P0	0 0 0 0
P1	**0 3 3 0**
P2	1 0 0 2
P3	0 0 2 0
P4	0 6 4 2

For P0

(0,0,0,0) ≤(1,1,0,0) ⇒True.

New work(available) = (1,1,0,0)+(0,0,1,2) = (1,1,1,2)

Finish[P0]=True

Next P1 will be considered

(0,3,3,0) ≤(1,**1**,1,2) ⇒ False

So Finish[P1]=false

Next P2 will be considered

(**1**,0,0,**2**) ≤(**1**,1,1,**2**) ⇒True.

New work(available) = (1,1,1,2)+(1,3,5,4) = (2,4,6,6)

Finish[P2]=True

Next P3 will be considered

(0,0,2,0) ≤(2,4,6,6) ⇒True.

New Work(available)=(2,4,6,6)+(0,6,3,2)=(2,10,9,8)

Finish[P3]=True

Now P4 will be considered

(0,6,4,2) ≤(2,10,9,8) ⇒True.

New work(available)=(2,10,9,8)+(0,0,1,4)=(2,10,10,12)

Finish[P4]=True

Now P1 will be considered

(0,3,3,0) ≤(2,10,10,12) ⇒ True.

New work(available)= (2,10,10,12)+(1,4,2,0)=(3,14,12,12)

Finish[P1]=True

Now the finish[i] of all the process are true and the sequence is

<P0,P2,P3,P4,P1> which is the safe sequence.

Since there exists a safe sequence, the request can be granted immediately.

13. Consider the following snapshot of a system with resource type A has 10 instances, resource type B has 5 instances and resource type C has 7 instances.

Process	Allocation	Max
	A B C	A B C
P0	0 1 0	7 4 3
P1	3 0 2	5 2 3
P2	3 0 1	6 0 3
P3	2 1 1	4 1 1
P4	0 0 2	4 3 2

Answer the following questions applying the banker's algorithm:

a. What is the content of the matrix *Need*? Is the system in a safe state? **(CO2-K3)**

Solution:

In this problem the Available vector is to be computed as follows.

The resource type A has 10 instances and 8 references were allocated. (For P1-3,P2-3,and for P3-2 references). So the Available for A will be 10-8=2

The resource type B has 5 instances and 2 were allocated.(For P0-1and P3-1 reference). So the available for B will be 5-2=3.

The resource type C has 7 instances and 6 were allocated. (For P1-2,P2-1,P3-1 and P4-2). So the available for C will be 7-6=1.

So the available will become 2,3,1

Then the problem can be solved as like the previous problem.