UNIT – 2

Basic Concepts of CPU Scheduling

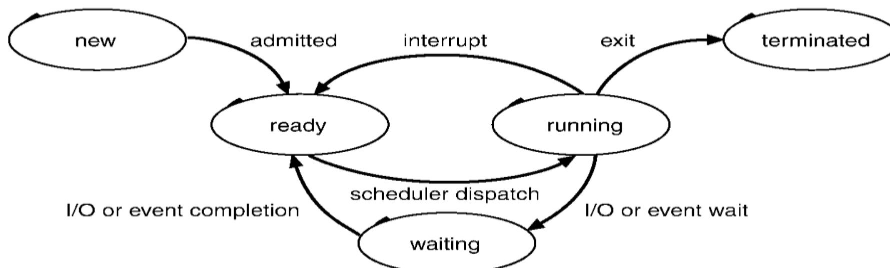
UNIT -02/Lecture 01

Process Concept
An operating system executes a variety of programs:
**Batch system – jobs
**Time-shared systems – user programs or tasks
**Textbook uses the terms job and process almost interchangeably.
**Process – a program in execution; process execution must progress in sequential fashion.

A process includes:
**program counter
**stack
**data section

Process State
**As a process executes, it changes state
****new**: The process is being created.
****running**: Instructions are being executed.
****waiting**: The process is waiting for some event to occur.
****ready**: The process is waiting to be assigned to a process.
****terminated**: The process has finished execution.

Process State



Process Control Block (PCB)

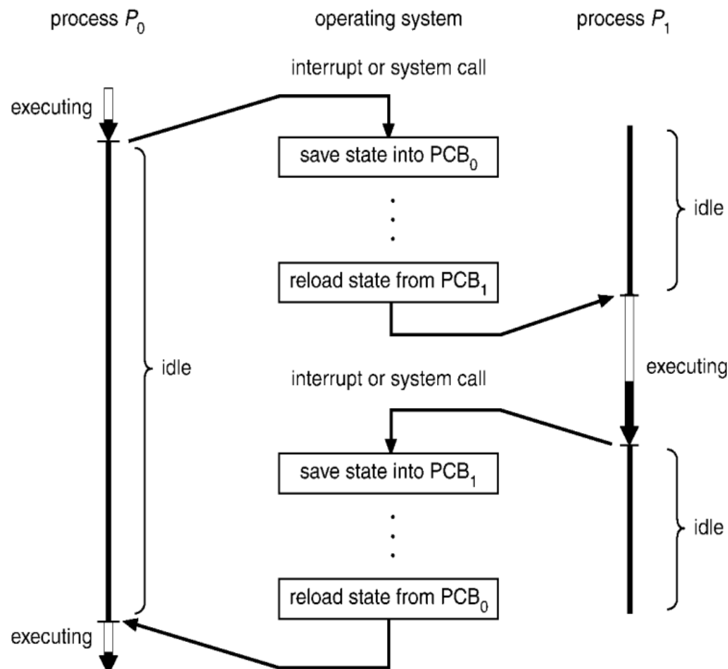
pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
⋮	

Information associated with each process.
** Process state
** Program counter
** CPU registers
** CPU scheduling information
** Memory-management information

we dont take any liability for the notes correctness.	http://www.rgpvonline.com

** Accounting information
** I/O status information

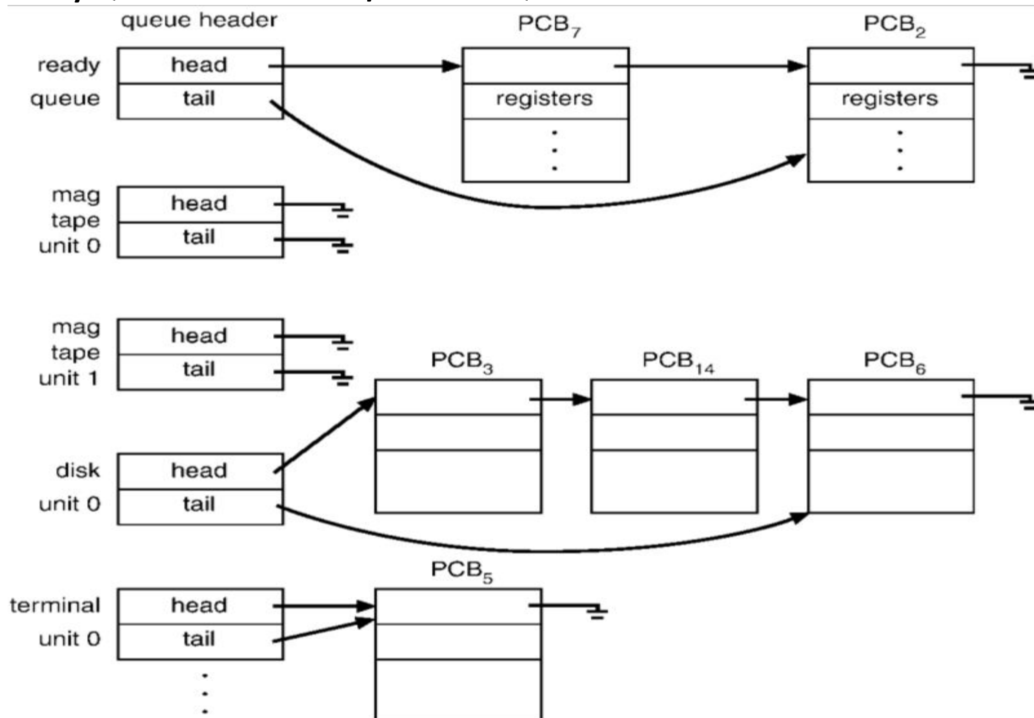
CPU Switch From Process to Process



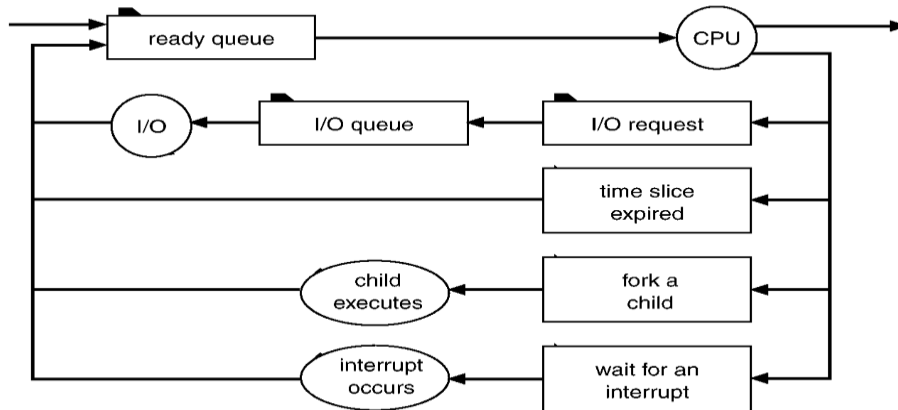
Process Scheduling Queues

** Job queue – set of all processes in the system.
** Ready queue – set of all processes residing in main memory, ready and waiting to execute.
** Device queues – set of processes waiting for an I/O device.
** Process migration between the various queues.

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



Schedulers

** Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue.

** Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU.

**Short-term scheduler is invoked very frequently (milliseconds) (must be fast).

**Long-term scheduler is invoked very infrequently (seconds, minutes) (may be slow).

**The long-term scheduler controls the degree of multiprogramming.

**Processes can be described as either:

**I/O-bound process – spends more time doing I/O than computations, many short CPU bursts.

**CPU-bound process – spends more time doing computations; few very long CPU bursts.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Short notes on context switching	Dec 2009	2
Q.2	Explain structure of PCB	Dec 2012 Dec 2009	7 3
Q.3	Types of Schedulers	Dec 2012	7

Unit-02/Lecture-02

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

Process Creation
**Parent process create children processes, which, in turn create other processes, forming a tree of processes.
**Resource sharing
**Parent and children share all resources.
**Children share subset of parent's resources.
**Parent and child share no resources.
**Execution
**Parent and children execute concurrently.
**Parent waits until children terminate.
**Address space
**Child duplicate of parent.
**Child has a program loaded into it.
UNIX examples
****fork** system call creates new process
****exec** system call used after a **fork** to replace the process' memory space with a new program.

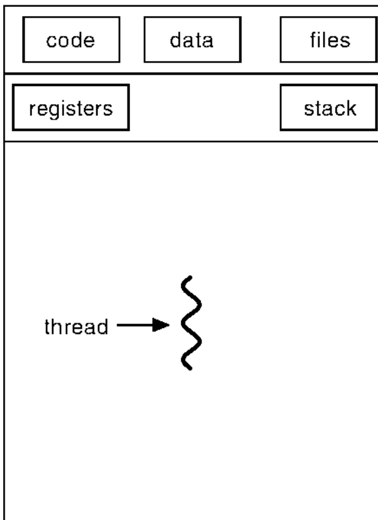
Process Termination
Process executes last statement and asks the operating system to decide it (exit**).
Output data from child to parent (via **wait).
**Process' resources are deallocated by operating system.
Parent may terminate execution of children processes (abort**).
**Child has exceeded allocated resources.
**Task assigned to child is no longer required.
**Parent is exiting.
**Operating system does not allow child to continue if its parent terminates.
**Cascading termination.

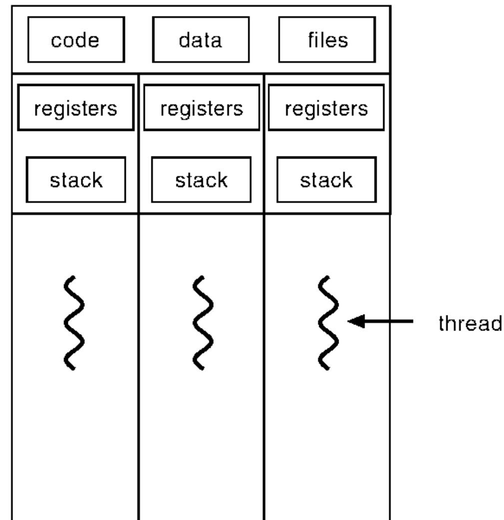
Cooperating Processes
**Independent process cannot affect or be affected by the execution of another process.
**Cooperating process can affect or be affected by the execution of another process
Advantages of process cooperation

**Information sharing
**Computation speed-up
**Modularity
**Convenience

we dont take any liability for the notes correctness. http://www.rgpvonline.com

Thread
Single and Multithreaded Processes





single-threaded multithreaded

Benefits

Responsiveness
Resource Sharing
Economy
Utilization of MP Architectures

User Threads

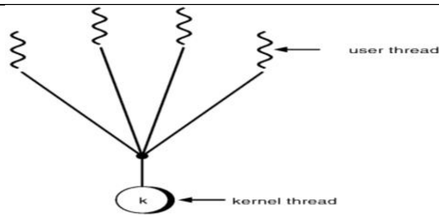
**Thread management done by user-level threads library Examples
- POSIX Pthreads
- Mach C-threads
- Solaris threads

Kernel Threads
**Supported by the Kernel
Examples
- Windows 95/98/NT/2000
- Solaris
- Tru64 UNIX
- BeOS
- Linux

Multithreading Models

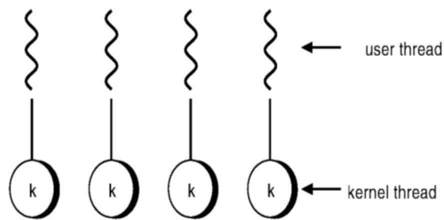
1) **Many-to-One**

Many user-level threads mapped to single kernel thread.
Used on systems that do not support kernel threads



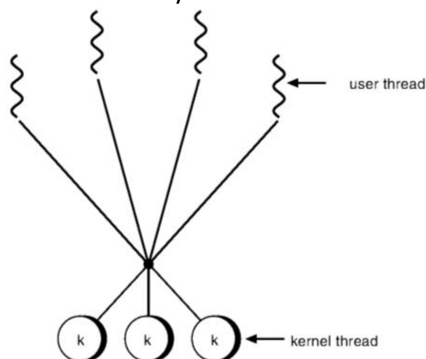
2) One-to-One Model

Each user-level thread maps to kernel thread.
Examples
- Windows 95/98/NT/2000
- OS/2



3) Many-to-Many Model

Allows many user level threads to be mapped to many kernel threads.
Allows the operating system to create a sufficient number of kernel threads.
-Solaris 2
-Windows NT/2000 with the ThreadFiber package



Threading Issues

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

Windows 2000 Threads

**Implements the one-to-one mapping.
**Each thread contains

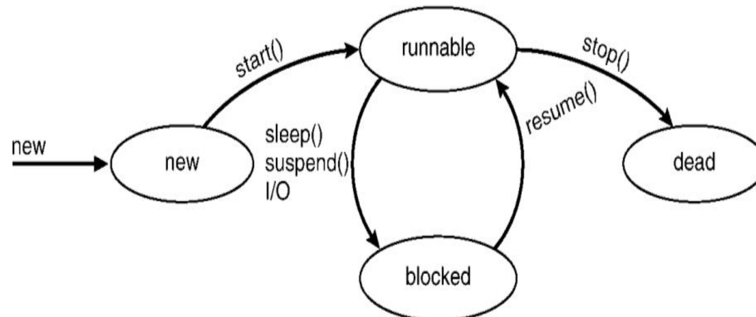
- a thread id
- register set
- separate user and kernel stacks
- private data storage area

Linux Threads

**Linux refers to them as tasks rather than threads.
**Thread creation is done through clone() system call.
**Clone() allows a child task to share the address space of the parent task (process)

Java Threads

**Java threads may be created by:
**Extending Thread class
**Implementing the Runnable interface
****Java threads are managed by the JVM.**



S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Single thread Vs Multithreading issues	Dec 2011 Dec 2009	10 6
Q.2	Discuss Process Creation & termination	Dec 2012	5

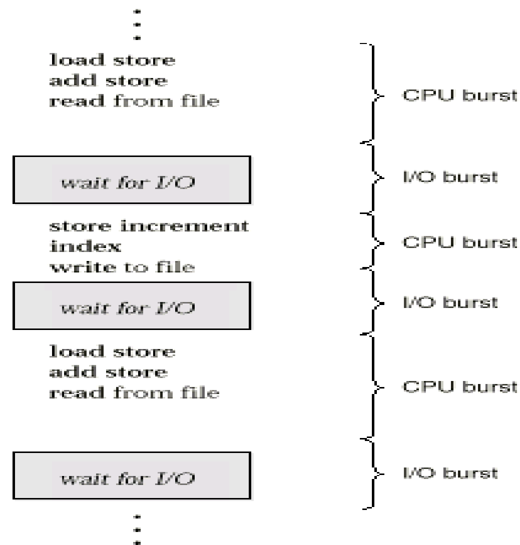
Unit-02/Lecture-03

CPU and I/O Bursts

- a process cycles between CPU processing and I/O activity
- a process generally has many short CPU bursts or a few long CPU bursts

- I/O bound processes have many short CPU bursts
- CPU bound processes have few long CPU bursts

- this can effect the choice of CPU scheduling algorithm used in an OS



Scheduling

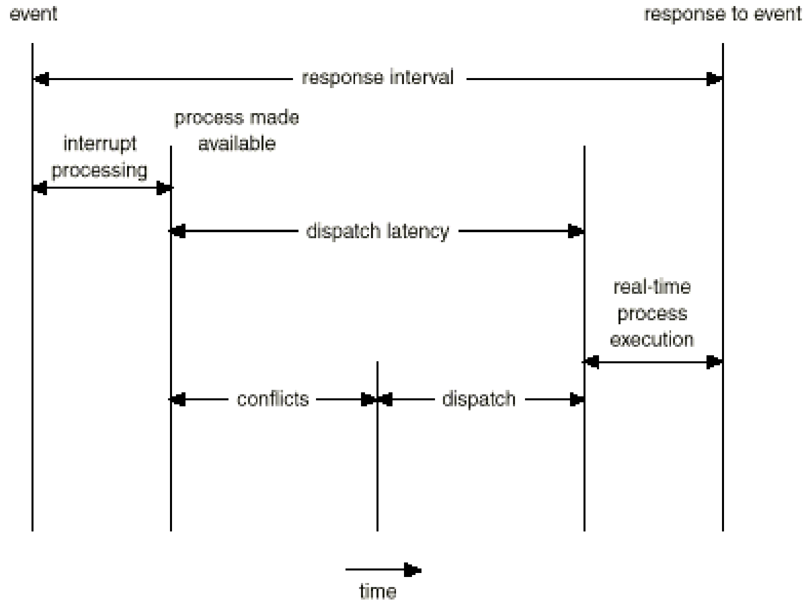
- CPU scheduling decisions may take place when a process
    1. switches from the running to waiting state
    2. switches from the running to ready state
    3. switches from the waiting to ready state
    4. terminates
- scheduling under conditions 1 and 4 is called non-preemptive (context switch is caused by the running program)
- scheduling under conditions 2 and 3 is preemptive (context switch caused by external reasons)

Dispatcher

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

a. switching context
b. switching to user mode
c. jumping to the proper location in the user program to restart that program

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



Scheduling Criteria

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

Optimization Criteria
1. Max CPU utilization
2. Max throughput
3. Min turnaround time
4. Min waiting time
5. Min response time

CPU Scheduling Algorithms

- First-Come, First Serve (FCFS or FIFO) (non-preemptive)
- Priority (e.g., Shortest Job First (SJF; non-preemptive)
- Shortest Remaining Time First (SRTF; preemptive))
- Round Robin (preemptive)
- Multi-level Queue
- Multi-level Feedback Queue

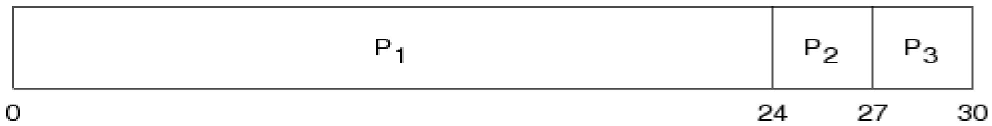
First-Come, First Serve

- non-preemptive scheduling management
- ready queue is managed as a FIFO queue

- example: 3 jobs arrive at time 0 in the following order (batch processing):

Process	Burst Time	Arrival	Start	Wait	Finish	TA
1	24	0	0	0	24	24
2	3	0	24	24	27	27
3	3	0	27	27	30	30

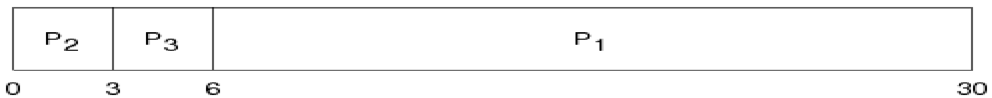
- Gantt chart:



- average waiting time: (0+24+27)/3 = 17
- average turnaround time: (24+27+30) = 27
- consider arrival order: 2, 3, 1

Process	Burst Time	Arrival	Start	Wait	Finish	TA
2	3	0	0	0	3	3
3	3	0	3	3	6	6
1	24	0	6	6	30	30

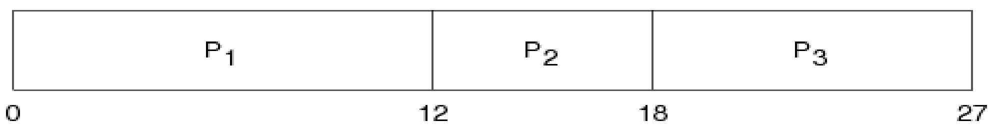
- Gantt chart:



- average waiting time: (0+3+6)/3 = 3
- average turnaround time: (3+6+30) = 13
- another example:

Process	Burst Time	Arrival	Start	Wait	Finish	TA
1	12	0	0	0	12	12
2	6	1	12	11	18	17
3	9	4	18	14	27	23

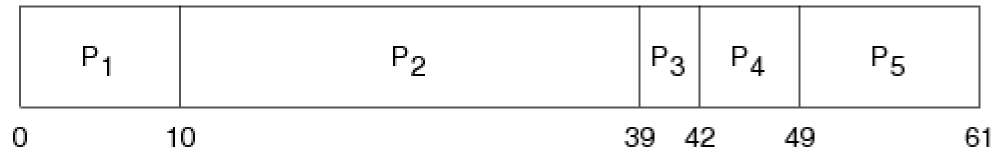
- Gantt chart:



- average waiting time: (0+11+14)/3 = 8.33
- average turnaround time: (12+17+23) = 52/3 = 17.33
- another example:

Process	Burst Time	Arrival	Start	Wait	Finish	TA
1	10	0	0	0	10	10
2	29	0	10	10	39	39
3	3	0	39	39	42	42
4	7	0	42	42	49	49
5	12	0	49	49	61	61

- Gantt chart:



- average waiting time: (0+10+39+42+49)/5 = 28
- average turnaround time: (10+39+42+49+61)/5 = 40.2

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Scheduling Criteria	Dec 2011 June 2010	10 10

Unit-02/Lecture-04

Priority Scheduling

- associate a priority with each process, allocate the CPU to the process with the highest priority
- any 2 processes with the same priority are handled FCFS
- SJF is a version of priority scheduling where the priority is defined using the predicted CPU burst length
- priorities are usually numeric over a range
- high numbers may indicate low priority (system dependent)
- internal (process-based) priorities: time limits, memory requirements, resources needed, burst ratio
- external (often political) priorities: importance, source (e.g., faculty, student)
- priority scheduling can be non-preemptive or preemptive
- problem: starvation --- low priority processes may never execute because they are waiting indefinitely for the CPU
- a solution: aging --- increase the priority of a process as time progresses

Shortest Job First (SJF)

- associate with each process the length of its next CPU burst
- schedule the process with the shortest time
- two schemes
  - non-preemptive: once scheduled, a process continues until the end of its CPU burst
  - preemptive: preempt if a new process arrives with a CPU burst of less length than the remaining time of the currently executing process; known as the Shortest Remaining Time First (SRTF) algorithm
- SJF is provably optimal; it yields a minimum average waiting time for any set of processes
- however, we cannot always predict the future (i.e., we do not know the next burst length)
- we can only estimate its length
- an estimate can be formed by using the length of its previous CPU bursts:

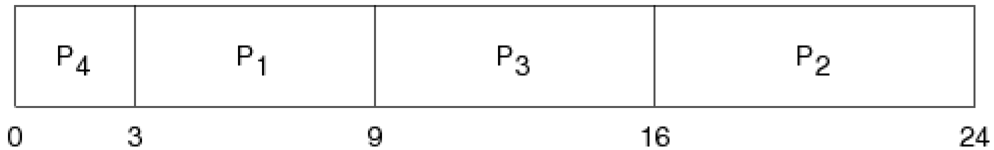
SJF (non-preemptive) examples

- example 1:

Process	Burst Time	Arrival	Start	Wait	Finish	TA
1	6	0	3	3	9	9
2	8	0	16	16	24	24
3	7	0	9	9	16	16
4	3	0	0	0	3	3

we dont take any liability for the notes correctness. http://www.rgpvonline.com

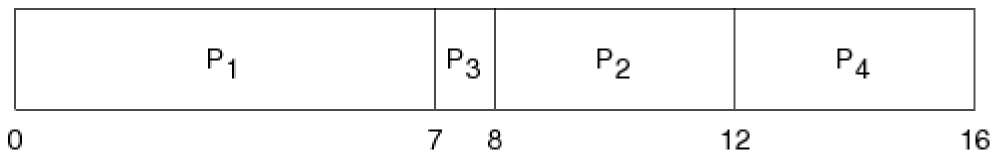
- Gantt chart:



- average waiting time: (3+16+9+0)/4 = 7
- average turnaround time: (9+24+16+3)/4 = 13
- example 2:

Process	Burst Time	Arrival	Start	Wait	Finish	TA
1	7	0	0	0	7	7
2	4	2	8	6	12	10
3	1	4	7	3	8	4
4	4	5	12	7	16	11

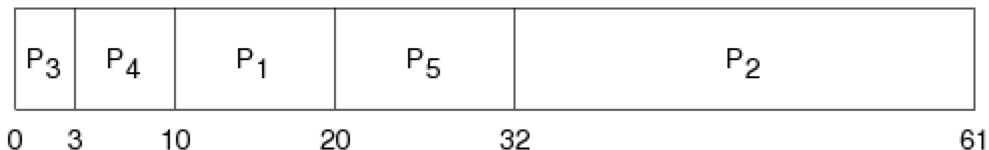
- Gantt chart:



- average waiting time: (0+6+3+7)/4 = 4
- average turnaround time: (7+4+10+11)/4 = 8
- example 3:

Process	Burst Time	Arrival	Start	Wait	Finish	TA
1	10	0	10	10	20	20
2	29	0	32	32	61	61
3	3	0	0	0	3	3
4	7	0	3	3	10	10
5	12	0	20	20	32	32

- Gantt chart:



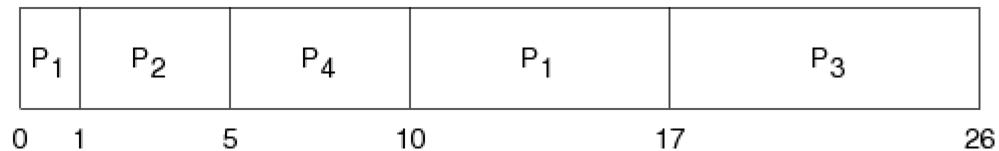
- average waiting time: (10+32+0+3+20)/5 = 13
- average turnaround time: (10+39+42+49+61)/5 = 25.2

SRTF (preemptive) examples

- example 1:

Process	Burst Time	Arrival	Start	Wait	Finish	TA
1	8	0	0	9	17	17
2	4	1	1	0	5	4
3	9	2	17	15	26	24
4	5	3	5	2	10	7

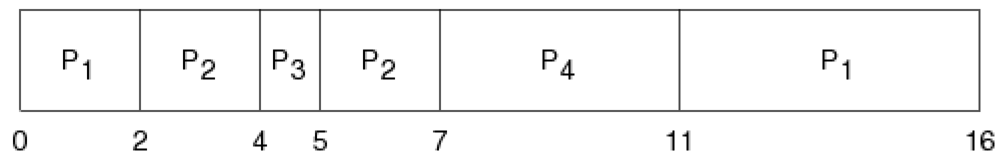
- Gantt chart:



- average waiting time: (9+0+15+2)/4 = 6.5
- average turnaround time: (17+4+24+7)/4 = 13
- example 2:

Process	Burst Time	Arrival	Start	Wait	Finish	TA
1	7	0	0	9	16	16
2	4	2	2	1	7	5
3	1	4	4	0	5	1
4	4	5	7	2	11	6

- Gantt chart:



- average waiting time: (9+1+0+2)/4 = 3
- average turnaround time: (16+5+1+6)/4 = 7

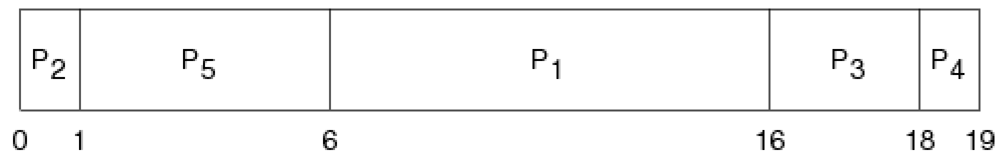
<table>
<tr><th>S.NO</th><th>RGPV QUESTION</th><th>YEAR</th><th>MARKS</th></tr>
<tr><td>Q.1</td><td>CPU Scheduling numerical
SJF,SRTF,FIFO</td><td>Dec
2009</td><td>20</td></tr>
<tr><td>Q.2</td><td>What is pre-emption CPU scheduling ? Compute average turnaround time and waiting time for pre-emptive shortest job first scheduling algorithm for the following data :
<table>
<tr><th>Job</th><th>Arrival Time</th><th>Burst Time</th></tr>
<tr><td>J1</td><td>0</td><td>4</td></tr>
<tr><td>J2</td><td>2</td><td>3</td></tr>
<tr><td>J3</td><td>5</td><td>6</td></tr>
<tr><td>J4</td><td>6</td><td>2</td></tr>
</table></td><td>June
2010
Dec
2012</td><td>10
10</td></tr>
</table>

Unit-02/Lecture-05

Priority scheduling example

Process	Burst Time	Priority	Arrival	Start	Wait	Finish	TA
1	10	3	0	6	6	16	16
2	1	1	0	0	0	1	1
3	2	4	0	16	16	18	18
4	1	5	0	18	18	19	19
5	5	2	0	1	1	6	6

Gantt chart:



average waiting time: (6+0+16+18+1)/5 = 8.2

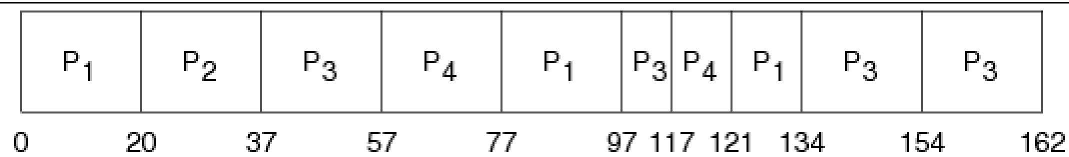
average turnaround time: (1+6+16+18+19)/5 = 12

Round Robin

- time sharing (preemptive) scheduler where each process is given access to the CPU for 1 time quantum (slice) (e.g., 20 milliseconds)
- a process may block itself before its time slice expires
- if it uses its entire time slice, it is then preempted and put at the end of the ready queue
- the ready queue is managed as a FIFO queue and treated as a circular
- if there are n processes on the ready queue and the time quantum is q, then each process gets 1/n time on the CPU in chunks of at most q time units
- no process waits for more than (n-1)q time units
- the choice of how big to make the time slice (q) is extremely important
  - if q is very large, Round Robin degenerates into FCFS
  - if q is very small, the context switch overhead defeats the benefits
- example 1 (q = 20):

Process	Burst Time	Arrival	Start	Wait	Finish	TA
1	53	0	0	?	134	134
2	17	0	20	?	37	37
3	68	0	37	?	162	162
4	24	0	57	?	121	121

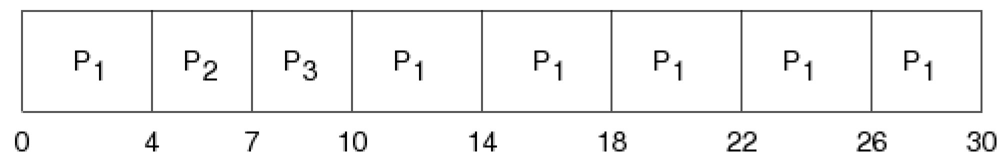
- Gantt chart:



- waiting times:
  - p_1: (77-20) + (121-97) = 81
  - p_2: (20-0) = 20
  - p_3: (37-0) + (97-57) + (134-117) = 94
  - p_4: (57-0) + (117-77) = 97
- average waiting time: (81+20+94+97)/4 = 73
- example 2 (q = 4):

Process	Burst Time	Arrival	Start	Wait	Finish	TA
1	24	0	0	6	30	30
2	3	0	4	4	7	7
3	3	0	7	7	10	10

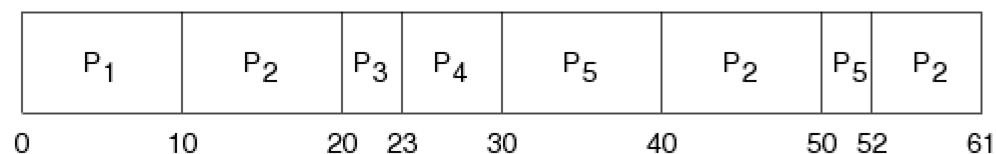
- Gantt chart:



- average waiting time: (6+4+7)/3 = 5.67
- average turnaround time: (30+7+10)/3 = 15.67
- example 3 (q = 10):

Process	Burst Time	Arrival	Start	Wait	Finish	TA
1	10	0	0	0	10	10
2	29	0	10	32	61	61
3	3	0	20	20	23	23
4	7	0	23	23	30	30
5	12	0	30	40	52	52

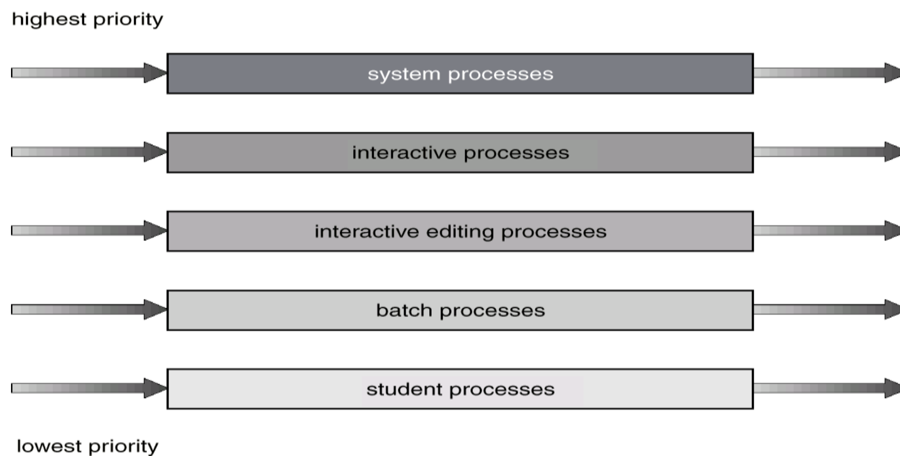
- Gantt chart:



- average waiting time: (0+32+20+23+40)/5 = 23
- average turnaround time: (10+39+42+49+61)/5 = 35.2

Multilevel Queue

- the ready queue is managed as multiple queues based on various characteristics. For instance,
  - foreground (interactive)
  - background (batch)
- each queue uses a particular scheduling algorithm. For instance,
  - foreground (round robin)
  - background (FCFS)
- scheduling must be done between queues:
  - fixed priority (may lead to starvation) (e.g., foreground jobs have absolute priority over background jobs)
  - time slice per queue

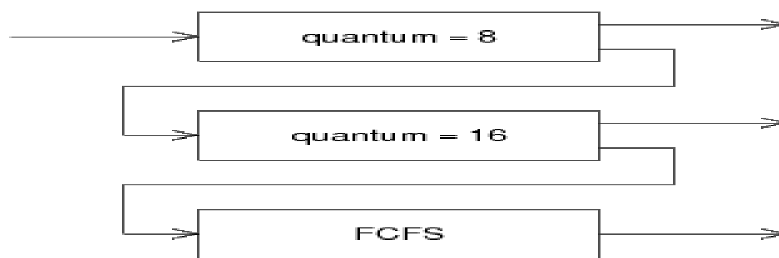


S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Consider the following data : Job | Burst Time | Priority 1 | 10 | 3 2 | 1 | 1 3 | 2 | 3 4 | 1 | 4 5 | 5 | 2 (i) Give a Gantt chart illustrating the exemption of these using FCFS, RR (quantum = 1, and a non-pre-emptive priority scheduling alg.) (ii) What is turn around time, waiting time and response time for each of the above scheduling alg ?	Dec 2012	20
Q.2	Calculate the average turn around time, average waiting time, throughput and processor utilization for the following set of processes. Process Processing time P_1 20 P_2 1 P_3 10 P_4 5 Quantum is 3. Use Round Robin scheduling policy. Draw the Gantt chart also.	Dec 2013	14

Unit-02/Lecture-06

Multilevel Feedback Queue

- processes move between the various queues
- a multilevel feedback queue is characterized by
  - number of queues
  - scheduling algorithm for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine on which queue a process begins (each time it returns to the ready state)
- example:
  - 3 queues
  - fixed priority based on length of CPU burst
  - RR for 1st queue, FCFS for last queue
  - each process begins on top queue (quantum = 8)



Multiple-Processor Scheduling

1. CPU scheduling more complex when multiple CPUs are available.
2. Homogeneous processors within a multiprocessor.
3. Load sharing
4. Asymmetric multiprocessing – only one processor accesses the system data structures, alleviating the need for data sharing.

Real-Time Scheduling

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

Algorithm Evaluation

- which algorithm should be used in a particular system?
- how should the parameters (e.g., q, number of levels) be defined?
- on which criteria do we base our decisions?

Approaches to evaluation

- deterministic modeling
- queue models
- simulation
- implementation

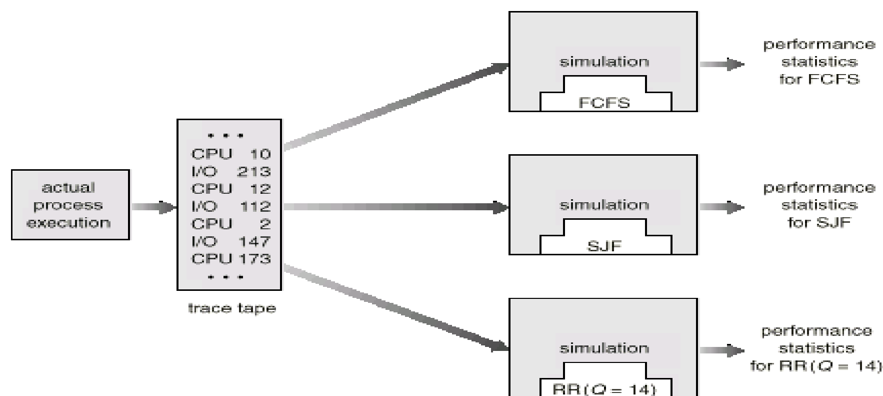
Deterministic modeling

- define a workload and compare it across algorithms
- simple to execute and results in distinct values to compare
- however, the results apply only to that case and cannot be generalized
- a set of workload scenarios with varying characteristics can be defined and analyzed
- must be careful about any conclusion drawn

Queuing models

- n = average queue length
- W = average waiting time in the queue
- λ = average arrival rate
- Little's Formula: $n = \lambda * W$
- Little's formula can be applied to the CPU and ready queue, or the wait queue for any device
- values can be obtained by measuring a real system over time and mathematically estimating
- the estimates are not always accurate due to:
  - complicated algorithms
  - assumptions
- therefore, the queuing model may not reflect reality to the level needed

Evaluation of CPU Schedulers by Simulation



S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	FCFS, RR, Multilevel feedback queue	Dec 2012	14

Unit-02/Lecture-07

Process Synchronization

1. Concurrent access to shared data may result in data inconsistency.
2. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
3. Shared-memory solution to bounded-butter problem (Chapter 4) allows at most n – 1 items in buffer at the same time. A solution, where all N buffers are used is not simple.
a. Suppose that we modify the producer-consumer code by adding a variable counter, initialized to 0 and incremented each time a new item is added to the buffer

Bounded-Buffer

Shared data

```
#define BUFFER_SIZE 10
typedef struct { …..
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
Producer process
item nextProduced;
while (1) {
while (counter == BUFFER_SIZE)
; /* do nothing */
buffer[in] = nextProduced;
in = (in + 1) % BUFFER_SIZE;
counter++;
}
Consumer process
item nextConsumed;
while (1) {
while (counter == 0)
; /* do nothing */
nextConsumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter--;
}
```

1. The statements counter++; counter--; must be performed atomically.
2. Atomic operation means an operation that completes in its entirety without interruption.
3. The statement "count++" may be implemented in machine language as: register1 = counter register1 = register1 + 1 counter = register1
4. The statement "count—" may be implemented as: register2 = counter register2 = register2 – 1 counter = register2
5. If both the producer and consumer attempt to update the buffer concurrently, the assembly

language statements may get interleaved.
6. Interleaving depends upon how the producer and consumer processes are scheduled.

1. Assume counter is initially 5. One interleaving of statements is: producer: register1 = counter (register1 = 5) producer: register1 = register1 + 1 (register1 = 6) consumer: register2 = counter (register2 = 5) consumer: register2 = register2 – 1 (register2 = 4) producer: counter = register1 (counter = 6) consumer: counter = register2 (counter = 4)
2. The value of count may be either 4 or 6, where the correct result should be 5.

The critical section problem

Given two more more processes (or threads) which share a resource (e.g., variable or device), we must often synchronize their activity. Must satisfy to one degree or another the concepts of mutual exclusion, progress, and bounded waiting.

Example: consider only two processes

critical section (<cs>): instructions which access shared resource

We must establish mutual exclusion: no two processes can be in their <cs> at the same time.

```
process producer {

  while (true) {
    while (count == BUFFER_SIZE);
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
  }
}
process consumer {

   while (true) {
     while (count == 0);
     --count;
     item = buffer[out];
     out = (out - 1) % BUFFER_SIZE;
   }
}
```

A race condition: a situation where multiple processes access and manipulate the same data concurrently and the outcome of the execution depends on the order in which the instructions execute.

A solution must satisfy three requirements:

- **mutual exclusion**: only one process may execute its critical section at once.
- **progress**: if no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes not executing in their remainder sections can participate in the decision on which process will enter its critical section

next, and this decision cannot be postponed indefinitely.

- **bounded waiting**: this is a limit on the number of times other processes are allowed to enter their critical section after a process has made a request to enter its critical section and before that request is granted.

Basic idea in synchronization: need locks in one form or another

```
while (true)
  // entry section; acquire lock
  // critical section
  // exit section; release lock
  // remainder section
}
```

Three primitive solutions to the critical section problem

- disable interrupts during execution of <cs>
- process p_i {
- while (true) {
- // disable interrupts (a system call)
- // critical section
- // enable interrupts (a system call)
- // remainder section
- }
- }
    - degrades efficiency
    - not possible multiprocessor systems
- hardware instructions (e.g., test-and-set and swap)
- Peterson's solution

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Process synchronization	Dec 2012 Dec 2013	10 7

Unit-02/Lecture-08

Hardware instructions

two new instructions, both versions of a read-modify-write instruction

- test and set
- boolean test-and-set (boolean* target) {
- boolean temp = *target;
- *target = true;
- return temp;
- }
-
- boolean occupied = false;
-
- while (true) {
-
- while (test-and-set (&occupied));
-
- // critical section
-
- occupied = false;
-
- // remainder section
- }

problems? starvation

- swap
- void swap (boolean* x, boolean *y) {
- boolean temp = *x;
- *x = *y;
- *y = temp;
- }
-
- boolean occupied = false;
- boolean p_i_must_wait = true;
-
- while (true) {
-
- do
- swap (&p_i_must_wait, &occupied);
- while (p_i_must_wait);
-
- // critical section
-
- p_i_must_wait = true;
- occupied = false;

we dont take any liability for the notes correctness.
http://www.rgpvonline.com

-
- // remainder section
- }

Peterson's Solution

shared data:

```
int turn;
boolean flag[2];

process p_i {

  while (true) {
    flag[i] = true; // I am ready to enter my <cs>
    turn = j;       // but I give p_j priority

    // as long as p_j wants access and it is p_j's turn, I do no-op
    while (flag[j] && turn == j);

    // critical section

    // I am no longer in my <cs>
    flag[i] = false;

    // remainder section;
  }
}

process p_j {

  while (true) {
    flag[j] = true; // I am ready to enter my <cs>
    turn = i;       // but I give p_i priority

    // as long as p_i wants access and it is p_i's turn, I do no-op
    while (flag[i] && turn == i);

    // critical section

    // I am no longer in my <cs>
    flag[j] = false;

    // remainder section;
  }
}
```

Peterson's solution guarantees mutual exclusion, progress, and bounded waiting.

What is the problem with Peterson's solution?

we dont take any liability for the notes correctness. http://www.rgpvonline.com

busy waiting: the waiting process wastes CPU cycles; in a uniprocessor system, the process waits until its quantum expires

Synchronization Hardware

Atomically swap two variables.

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

Mutual Exclusion with Swap

Shared data (initialized to **false**): **boolean lock;**

boolean waiting[n];

Process Pi

```
do {
key = true;
while (key == true)
Swap(lock,key);
critical section
lock = false;
remainder section
}
```

High-level synchronization solutions

(which rely on primitive solutions)

- semaphores
- monitors

Uses of semaphores

1. Synchronization tool that does not require busy waiting.
2. Semaphore S – integer variable
3. can only be accessed via two indivisible (atomic) operations

wait (S):

while S 0 do no-op; S--;

signal (S):

S++;

Types of semaphores

- binary semaphore (sometimes called a mutex lock): integer value can range only over 0 and 1
- counting semaphore: integer value can range over an unrestricted domain

Semaphore type can be an issue of implementation, or simple how a semaphore is used.

Binary semaphores can be simpler to implement depending on hardware support.

we dont take any liability for the notes correctness. http://www.rgpvonline.com

Counting semaphores can be implemented using binary semaphores

Two Types of Semaphores

1. Counting semaphore – integer value can range over an unrestricted domain.

2. Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement.

3. Can implement a counting semaphore S as a binary semaphore.

Implementing S **as a Binary Semaphore**
Data structures:
binary-semaphore S1, S2;
int C:
Initialization:
S1 = 1
S2 = 0
C = initial value of semaphore **S**
Implementing S
wait operation
wait(S1);
C--;
if (C < 0) {
signal(S1);
wait(S2);
}
signal(S1);
signal operation
wait(S1);
C ++;
if (C <= 0)
signal(S2);
else
signal(S1);

Deadlock and Starvation
Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
Let S and Q be two semaphores initialized to 1
P0 P1
wait(S); wait(Q);
wait(Q); wait(S);
? ?
signal(S); signal(Q);
signal(Q) signal(S);
Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

we dont take any liability for the notes correctness. http://www.rgpvonline.com

Critical Regions

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

2. When a process tries to execute the region statement, the Boolean expression B is evaluated. If B is true, statement S is executed. If it is false, the process is delayed until B becomes true and no other process is in the region associated with v.

Example – Bounded Buffer

1. Shared data:

struct buffer {
int pool[n];
int count, in, out;
}
Bounded Buffer Producer Process

1. Producer process inserts **nextp** into the shared buffer

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

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

region buffer when (count > 0) { nextc = pool[out]; out = (out+1) % n; count--; }
Implementation region x **when** B **do** S

1. Associate with the shared variable x, the following variables:
   i. semaphore mutex, first-delay, second-delay; int first-count, second-count;

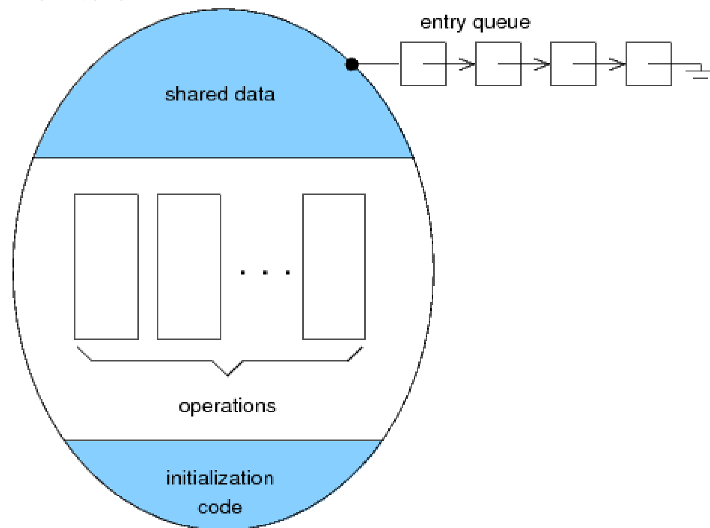
2. Mutually exclusive access to the critical section is provided by **mutex**.
3. If a process cannot enter the critical section because the Boolean expression **B** is false, it initially waits on the **first-delay** semaphore; moved to the **second-delay** semaphore before it is allowed to reevaluate B.

Implementation
1. Keep track of the number of processes waiting on **first-delay** and **second-delay**, with **first-count** and **second-count** respectively.
2. The algorithm assumes a FIFO ordering in the queuing of processes for a semaphore.
3. For an arbitrary queuing discipline, a more complicated implementation is required.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Critical section problem & its solutions	Dec 2013	7
Q.2	Use of semaphores with producer consumer problem	Dec 2011	10

Unit-02/Lecture-09

Monitors



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

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

1. To allow a process to wait within the monitor, a **condition** variable must be declared, as
condition x, y;

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

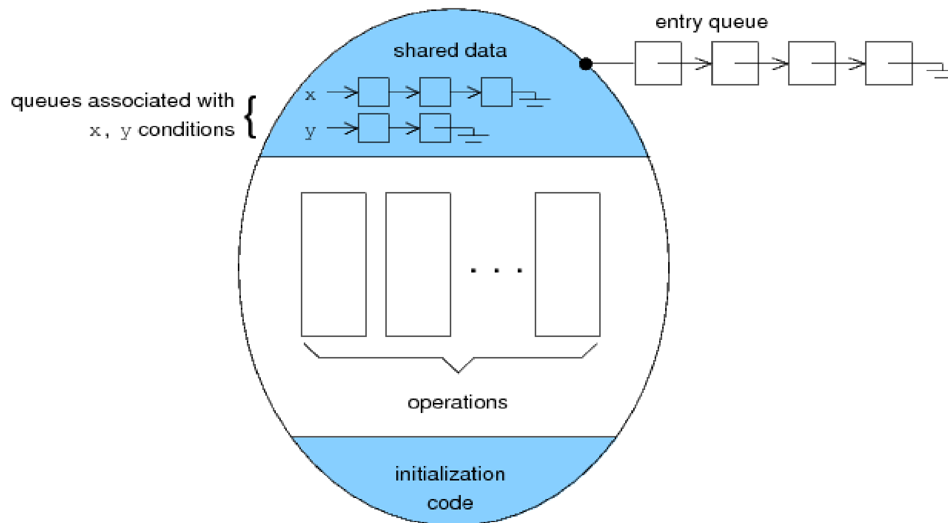
The operation
x.wait(); means that the process invoking this operation is suspended until another process invokes
x.signal();
The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.

Monitor Implementation

1) Conditional-wait construct: **x.wait(c);**
a. **c** – integer expression evaluated when the **wait** operation is executed.
b. value of **c** (a priority number) stored with the name of the process that is suspended.
c. when **x.signal** is executed, process with smallest associated priority number is resumed next.

2) Check two conditions to establish correctness of system:
a. User processes must always make their calls on the monitor in a correct sequence.
b. Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.

Condition variables



Classical probems of synchronization

- The Bounded Buffer Problem (also called the The Producer-Consumer Problem)
- The Readers-Writers Problem
- The Dining Philosophers Problem

These problems are used to test nearly every newly proposed synchronization scheme or primitive.

The Bounded Buffer Problem

Consider:

- a buffer which can store n items
- a producer process which creates the items (1 at a time)
- a consumer process which processes them (1 at a time)

A producer cannot produce unless there is an empty buffer slot to fill.

A consumer cannot consume unless there is at least one produced item.

```
Semaphore empty=N, full=0, mutex=1;

process producer {
  while (true) {
    empty.acquire();
    mutex.acquire();
    // produce
    mutex.release();
    full.release();
  }
}

process consumer {
  while (true) {
    full.acquire();
    mutex.acquire();
    // consume
    mutex.release();
    empty.release();
}
```

The semaphore mutex provides mutual exclusion for access to the buffer.

The Readers-Writers Problem

A data item such as a file is shared among several processes.

Each process is classified as either a reader or writer.

Multiple readers may access the file simultaneously.

A writer must have exclusive access (i.e., cannot share with either a reader or another writer).

A solution gives priority to either readers or writers.

- readers' priority: no reader is kept waiting unless a writer has already obtained permission to access the database
- writers' priority: if a writer is waiting to access the database, no new readers can start reading

A solution to either version may cause starvation

- in the readers' priority version, writers may starve
- in the writers' priority version, readers may starve

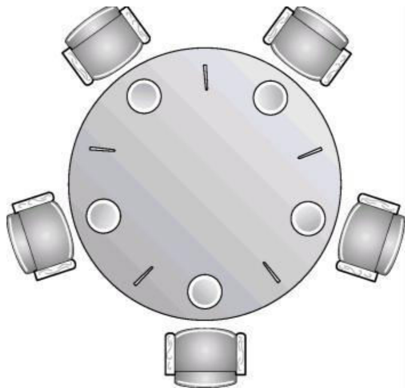
A semaphore solution to the readers' priority version (without addressing starvation):

```
Semaphore mutex = 1;
```

```
Semaphore db = 1;
int readerCount = 0;

process writer {

  db.acquire();
  // write
  db.release();
}
 process reader {
  // protecting readerCount
  mutex.acquire();
  ++readerCount;
  if (readerCount == 1)
    db.acquire();
  mutex.release();

  // read

  // protecting readerCount
  mutex.acquire();
  --readerCount;
  if (readerCount == 0)
    db.release;
  mutex.release();
}
```

readerCount is a <cs> over which we must maintain control and we use mutex to do so.

The Dining Philosophers Problem



n philosophers sit around a table thinking and eating. When a philosopher thinks she does not interact with her colleagues. Periodically, a philosopher gets hungry and tries to pick up the chopstick on his left and on his right. A philosopher may only pick up one chopstick at a time and, obviously, cannot pick up a chopstick already in the hand of neighbor philosopher.

The dining philosophers problems is an example of a large class or concurrency control problems; it is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

A semaphore solution:

```
// represent each chopstick with a semaphore
Semaphore chopstick[] = new Semaphore[5]; // all = 1 initially

process philosopher_i {

  while (true) {
    // pick up left chopstick
    chopstick[i].acquire();

    // pick up right chopstick
    chopstick[(i+1) % 5].acquire();

    // eat

    // put down left chopstick
    chopstick[i].release();

    // put down right chopstick
    chopstick[(i+1) % 5].release();

    // think
  }
}
```

.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Reader Writer problem	June 2010 Dec 2011 Dec 2013	10 10 7
Q.2	Process synchronization	Dec 2012 Dec 2013	10 7