Scheduling Algorithm optimization

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

$TAT = FT - AT$

$WT = TAT - BT$

First-come, First-served (FCFS) scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

FCFS : process arrive in order: P_1 P_2 P_3

Gantt chart for schedule:

P_1	P_2	P_3
0 – 24	24 – 27	27 – 30

waiting time for $P_1 = 0$ $P_2 = 24$

$P_3 = 27$

Average waiting time $= \dfrac{(0+24+27)}{3} = 17$

convoy effect :→ short process behind long process.

Shortest-Job-First (SJF) scheduling

Associate with each process the length of its next CPU Burst.

SJF is optimal - gives minimum average waiting time for a given set of processes

Example of SJF

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

SJF scheduling chart

P4	P1	P3	P2

0 3 9 16 24

average waiting time $= \frac{(3+9+16+0)}{4} = 7$

TAT = FT − AT

WT = TAT − BT

Explaining shortest-Remaining-Time-First or SJF (preemptive)

Process	Arrival Time	Burst-Time
P1	0	8→7
P2	1	4→~~3~~→~~2~~→~~2~~→~~1~~→~~0~~
P3	2	9
P4	3	5

P1	P2	P2	P2	P2	P4	P4	P4	P4	P4	P1	P1	P1	P1	P1	P1	P1

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

P3	P3	P3	P3	P3	P3	P3	P3	P3

← 18 19 20 21 22 23 24 25 26 ~~27~~

Process	Arrival Time	BURST Time	completion Time	Turn-around Time	Waiting Time
P1	0	8	17	17	~~8~~ 9
P2	1	4	5	4	0
P3	2	9	26	24	15
P4	3	5	10	7	2

TAT = FT − AT   WT = TAT − BT

$$\frac{9+15+2+0}{4} = \frac{26}{4} = 6.5 \text{ msec}$$

Priority-scheduling

A priority number (integer) is associated with each process

The CPU is allocated to the process with highest priority

preemitive   Non-preemitive

SJF is priority scheduling where priority is the inverse of predicted next CPU Burst time

problem = starvation
solution = Aging

Example of priority - scheduling

process	Burst Time	priority	completion
P1	10	3	
P2	1	1	
P3	2	4	
P4	1	5	
P5	5	2	

priority scheduling gantt - chart

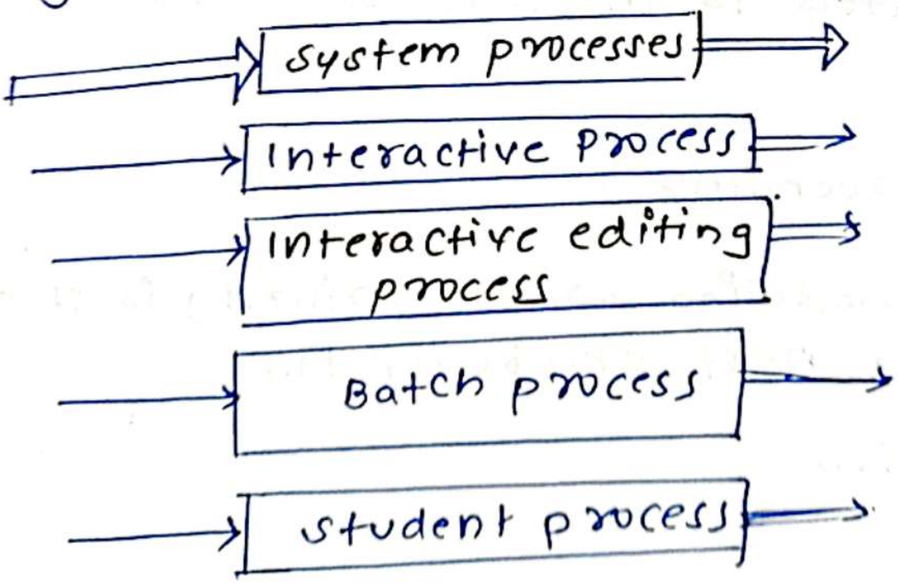
P2	P5	P1	P3	P4

0 1 6 16 18 19

Average waiting time $= \dfrac{0+1+6+16+18}{5} = 8.2$ msec

Multilevel Queue scheduling

highest priority



lowest priority

chp.7 **Deadlocks**

System Model

System consist of resources
 Resources Types $R_1, R_2 \dots R_m$

Each Resource Type has R_i has W_i Instances

Deadlock Arrises if four condn hold simultaneously

- Mutual exclusion
- Hold & Wait
- No preemption
- circular wait

Resource Allocation Graphs (RAGs)

Vertices

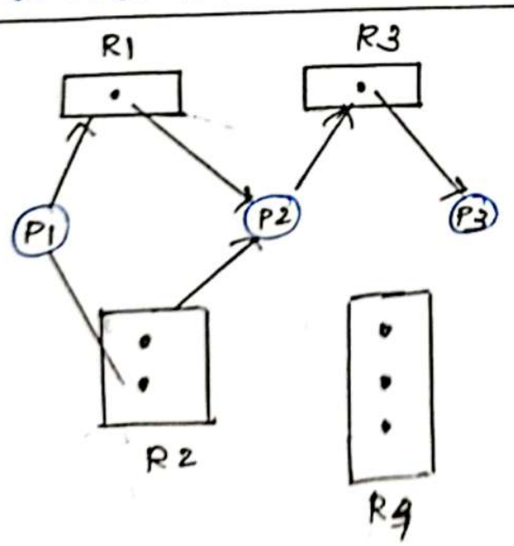
$P = \{P_1, P_2, P_n\}$ process

$R = \{R_1, R_2 \dots R_n\}$ Resource Types

Request edge :- Directed edge $P_i \longrightarrow R_j$

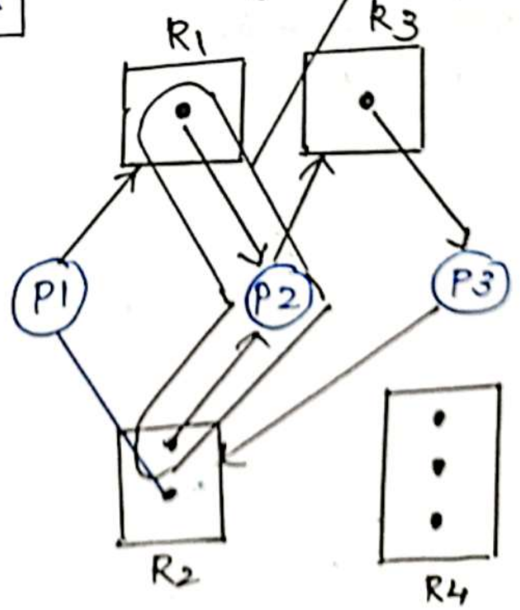
Assignment edge :- Directed edge $R_j \rightarrow P_i$

Resource Allocation Graphs

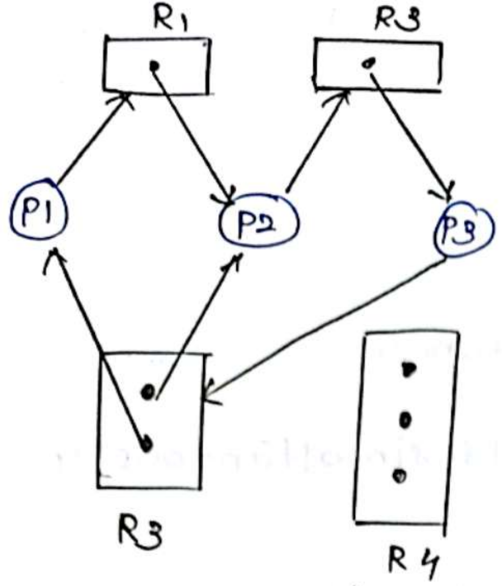


Graph with No Deadlock

Two instances with two different resources assigned to process P2



Resource Allocation Graph with a deadlock



~~All instances are Assigned to one process.~~

Resource Allocation Graph with A Deadlock

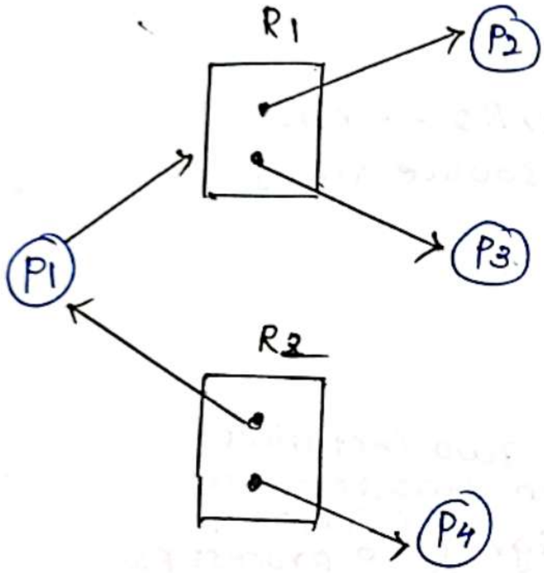
Note

- If Graph contains No cycle → No deadlock

If graph contains cycle

→ only one instance per resource type, then deadlock

→ If several instances per resource types/ possibility of deadlock



All Instances are assigned to one process each.

Graph with cycle But No Deadlock

Avoidance Algorithms

- single instance over Resource Type → Resource Allocation Graph
- Multiple Instance over Resource Type → Banker's Algorithm

Bankers Algorithm

n = no. of process m = number of Resources

Available available[j] = K there are K instances of Resource type R_j available

MAX n×m if max[i, j] = K then process P_i may request maximum K instances of resource type R_j

Allocation n×m matrix. If allocation[i,j] = K then P_i currently allocated K instances of R_j

Need : Need[i,j] = K then P_i may need K more instance of R_j to complete its task

$$\boxed{Need[i,j] = max[i,j] - allocation[i,j]}$$

Safety - Algorithm

1. Work and Finish length m and n

   Work = Available

   Finish[i] = false for i = 0, 1,

2. Find an i such that

a) Finish[i] = false

b) $Need_i \leq Work$

   if no such i exist goto 4)

3) Work = Work + Allocation

   Finish[i] = true

   go to step 2)

4) Finish[i] == true for all i

   system in a safe state

Reo

Resource - Request Algorithm for process Pi

1. $Request_i \leq Need_i$

2. $Request_i \leq Available$

$$Available = Available - Request_i;$$
$$Allocation = Allocation_i + Request_i;$$
$$Need = Need_i - Request_i;$$

if safe ⇒ The resources are ~~avai~~ allocated to Pi

if unsafe ⇒ Pi must wait, and the old - resource - allocation state is restored

Example of Banker's Algorithm

5 process P_0 through P_4;

3 resource types

A (10 instances), B (5 instances) C (7 instances)

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Need = MAX – Allocation

Need

P	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

The system is safe state ~~until~~ since the sequence <P_1 P_3 P_4 P_2 P_0> satisfies safety criteria.

Deadlock Avoidance

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 examine the resource allocation state to ensure that there never can be a circular-wait condition.

Resource-allocation state is defined by number of available and allocated resources and maximum demands of process

safe state

system is in safe state if there exists a sequence $\langle P1, P2, Pn \rangle$ of All ~~resource~~ ~~pe~~ ~~p~~ the process in the systems such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj with $j < i$

that is

Pi resource needs are not immediately available then Pi can wait until all Pj have finished.

when Pj is finished Pi can ~~obj~~ obtain needed resources, execute, return allocated resources and terminate

when pi terminates Pi+1 can obtain its needful resources and so on.

If a system is in safe state → NO Deadlocks

If a system is in unsafe state → possibility of Deadlock

Avoidance ⇒ Ensure that a system will never enter an unsafe state.

process syncronization

producer and consumer problem -

1) A producer process produce information that is consumed by
2) a consumer process.

one solution to producer and consumer is shared memory

3) A Buffer of items that can be filled by the producer and emptied by consumer

Note: The process of producer and consumer should be syncronized, so consumer does not try to consume a item that has not been produced

Buffers

- **unbounded buffer**: No practical limit on size of Buffer
- **Bounded Buffer**: Assumes a fixed buffer size

A situation like this, where several processes access and manipulate the same data concurrently and the outcome of execution depends on particular order in which the access takes place, is called a race condition.

Clearly, we want the resulting changes not to interfere with one another hence, we need process syncronization.

critical-section problem

critical-section Each segment has a section of code, called a critical section, in which process may be changing common variables, updating a table, writing a file and so on

when one process is executing in its critical section, no other process is to be allowed to execute in its critical section

No two process 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.

General structure of code of process

```
do {
    [entry section]
        critical section
    [exit section]
        remainder section
} while (TRUE)
```

Solution to critical section problem must satisfy three Req.

1. Mutual Exclusion
only one process should be in critical section

2. progress
process not in remainder section can participate in critical section

Bounded waiting
There exist a bound or limit that other process are allowed to enter critical section problem

Peterson's solution

peterson's solution is restricted to two process that alternate execution between their critical section and remainder section.

Let's call the process Pi and Pj

int turn
→ indicates whoese turn is it to enter critical section

Boolean Flag[2]
→ used to indicate if a process is ready to enter its critical section.

Structure of process pi in peterson's solution

```
do {
flag[i] = true;
 turn = j;
while ( flag[j] && turn==[j]);
   critical section
flag [i] = false;
   remainder section
 } while (TRUE)
```

structure of process pj in peterson's solution

```
do {
flag[j] = true;
 turn = i;
while ( flag[i] &&
  turn == [i]);
critical section
flag [j] = False
remainder section
} while (TRUE);
```

Test and Set Lock

shared lock variable which can either take two values 0 or 1

Before entering into its critical section, a process inquires about lock

if its locked it keeps waiting till free

if not locked, it takes the lock and executes critical section

```
boolean TestAndSET(boolean *target) {
   boolean rv = *target
     *target = TRUE
     return rv;
   }
do {
while (TestAndSEt(&lock));
   // donothing
 // critical section
   lock = FALSE
 } while (TRUE);
```

semaphores

Semaphores is simply a variable which is non-negative and shared between threads. Used to solve critical section problem and to achieve process syncronization

A semaphore S is an integer variable is accessed through two standard atomic operations

wait() signal()

wait() → P "to test"

signal() → V "to increment"

Defination of signal()

```
P(Semaphore S) {
  while (S<=0);
  S--;
}
```

if S<=0 it will tell the process to wait as some other process is in critical condition.

Defination of signal()

```
V (semaphore S) {
  S++;
}
```

Types of Semaphores

Binary Semaphore

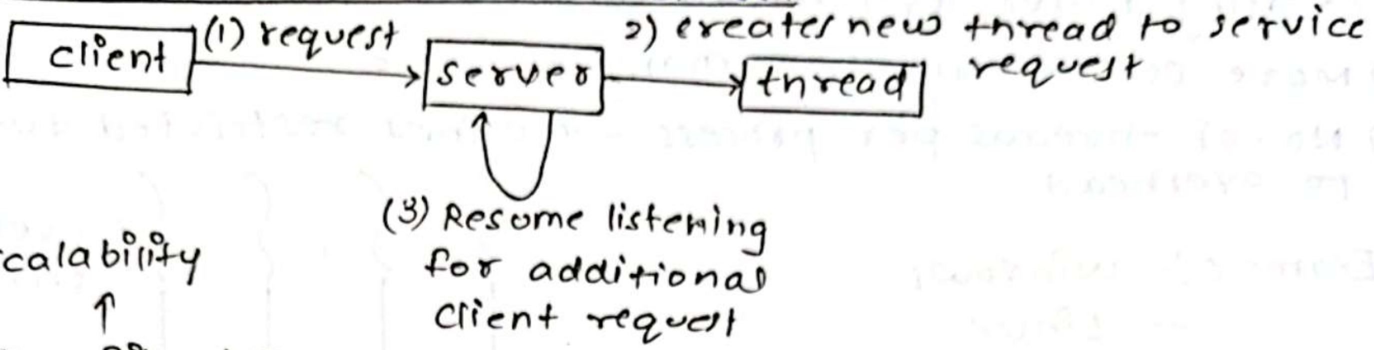
The value of a binary semaphore can range only between 0 and 1. They are known as mutex Locks. Locks that provide mutual exclusion

Counting Semaphores

Its value can change over an unrestricted domain. It is used to control access to a resource that has multiple instances

Threads

Multithreaded server Architecture



Benefits → Scalability
Benefits → Economy
Benefits → Responsiveness
Benefits → Resource sharing

User Threads and kernel threads

user threads: management done by user-Level thread library

kernel threads: supported by the kernel

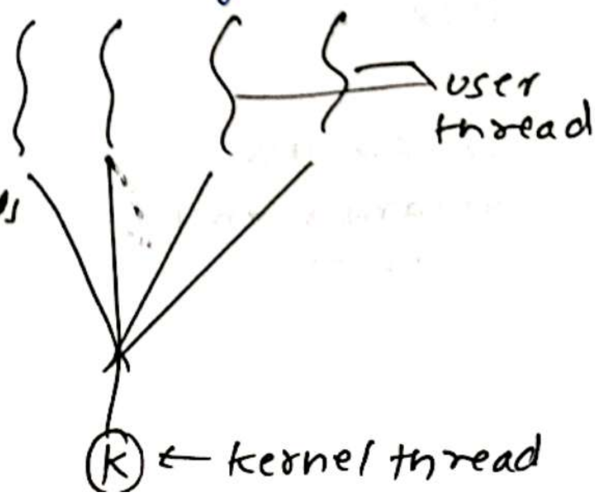
Multithreading Model

- Many-to-one
- one-to-one
- Many-to-Many

Many-to-one threads

1) Many user-level threads mapped to single thread
2) one thread ~~causes~~ blocking causes all to block.
3) Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time

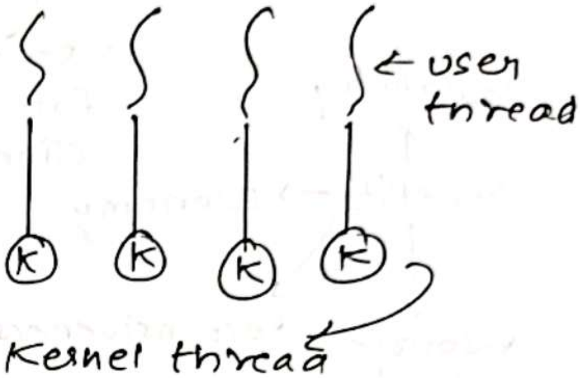
Example :- solaris Green Threads
– GNU Portable Threads



One-to-one

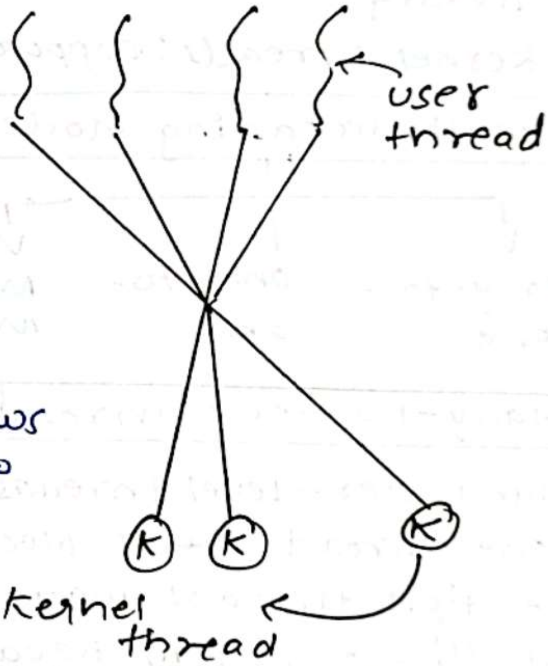
1) Each user-level thread is mapped to kernel thread
2) Creating a user-level thread creates a kernel thread
3) More concurrency than many-to-one
4) No. of threads per process sometimes restricted due to overhead

Example :- Windows
- Linux
- Solaris 9 & Later



Many-to-Many Model

1) Allows many user level threads to be mapped to many kernel threads.
2) Allows the operating system to create a sufficient number of kernel threads.
3) Solaris prior to version 9
4) Windows with the ThreadFiber package

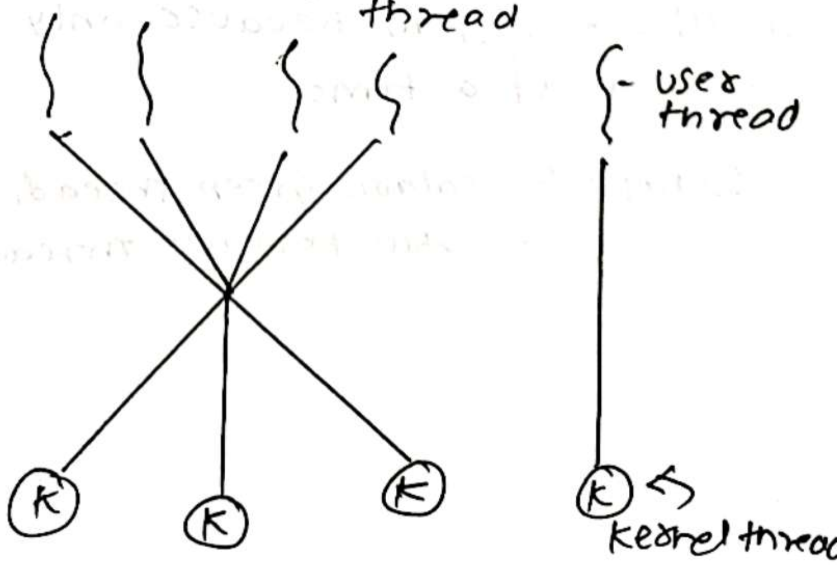


Two-level Model

1) Similar to M:M except that it allows a user thread to be bound to kernel thread

Example

IRIX
HP-UX
Tru64 UNIX
Solaris 8 and earlier



Thread-Libraries

- POSIX Pthread
- Windows
- Java

POSIX (Portable Operating System Interface) is a set of standard operating system interfaces based on the Unix operating system.

The windows thread library is a kernel-level library available on windows system.

Java thread API allows threads to be created and managed directly in Java programs