Chapter 2 Topics:
Process synchronization:

- Background, critical section problem, understating the race condition and the need for the process synchronization. [From Chapter 1]
- Petersons solution, synchronization Hardware
- Semaphores implementation
- Classical synchronization problem: Bounder buffer, Reader Writer, Dining philosophers

CPU scheduling:

- FCFS, SJF, Priority, Round robin, Multilevel queue and multilevel feedback queue scheduling, Real Time scheduling – Rate Monotonic Scheduling and Deadline scheduling

Deadlocks:

- Necessary conditions, Resource allocation graph, Deadlock prevention, Avoidance, detection and recovery.

PROCESS SYNCHRONIZATION:

Introduction

CPU scheduler switches rapidly between processes to provide concurrent execution. This means that one process may only partially complete execution before another process is scheduled. In fact, a process may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instructions of another process.

Now consider the following producer Consumer Code

Producer Process	**Consumer Process**

while (true) {
/* produce an item in next produced */
while (counter == BUFFER SIZE)
; /* do nothing */
buffer[in] = next produced;
in = (in + 1) % BUFFER SIZE;
counter++;
}
	while (true) {
	while (counter == 0)
	; /* do nothing */
	next consumed = buffer[out];
	out = (out + 1) % BU
	FFER SIZE;
	counter--;
	/* consume the item in next consumed */
	}

The statements “counter++” and “counter—” may be implemented in machine language (on a typical machine) as follows:

Counter ++	***Counter--***
*register*1 = counter *register*1 = *register*1 + 1 counter = *register*1	*register*2 = counter *register*2 = *register*2 − 1 counter = *register*2

The concurrent execution of “counter++” and “counter--” is equivalent to a sequential execution in which the lower-level statements presented are interleaved in some arbitrary order.

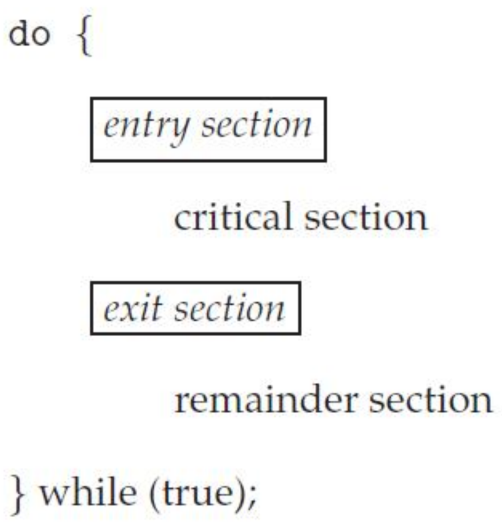
*T*0: *producer* execute *register*1 = counter {*register*1 = 5}
*T*1: *producer* execute *register*1 = *register*1 + 1 {*register*1 = 6}
*T*2: *consumer* execute *register*2 = counter {*register*2 = 5}

*T*3: *consumer* execute *register*2 = *register*2 − 1 {*register*2 = 4}
*T*4: *producer* execute counter = *register*1 {*counter* = 6}
*T*5: *consumer* execute counter = *register*2 {*counter* = 4}

Several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**. **Maintaining data consistency requires** while writing into shared data must be mutually exclusive and need mechanisms to ensure the orderly execution

The Critical-Section Problem

- Consider a system consisting of *n* processes {P_0, P_1, ..., P_{n-1}}. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on.
- 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.
- 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**.
- The general structure of a typical process P_i is shown below :



```
do {
      entry section
          critical section
      exit section
          remainder section
} while (true);
```

A solution to the **critical-section problem must satisfy** the following **three requirements**:
1. **Mutual exclusion**. If process P_i 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 some processes wish to enter their critical sections, for which a selection is made which cannot be postponed indefinitely.
3. **Bounded waiting**. There exists a bound, or limit, on the number of times that **other processes** are allowed to enter their critical sections and makes a bounded waiting for a process

Types of Solutions

- Software solutions : Petersons Algorithm - Algorithms whose correctness does not rely on any other assumptions.
- Hardware solutions - Rely on some special machine instructions.
- Operating System solutions – Semaphore - Provide some functions and data structures to the programmer through system/library calls.
- Programming Language solutions -Monitors - Constructs provided as part of a high level language

Peterson’s Solution

The classic software-based two process solution to the critical-section problem known as Peterson’s solution

The two processes share two variables:
int turn;
Boolean flag[2]
The variable turn indicates whose turn it is to enter the critical section. The flag array is used to indicate if a process is ready to enter the critical section.
flag[i] = true implies that process Pi is ready!

Algorithm for Process Pi

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

Algorithm with explanation

Process Pi	Process Pj

do {
flag[i]=true;*//Pi indicates it wants to enter critical Section//*
turn = j;*//sets turn as j allowing Pj to enter Critical section If Pj wants to//*
while (flag [j] and turn = j);
{//wait//}
// if Pj wants to enter & it is Pj’s turn to enter its critical section then Pi waits until Pi completes its critical section//
critical section*//critical section of Pi//*
flag [i]=false;
	do {
	flag[j]=true;*//Pj indicates it wants to enter critical Section//*
	turn = i;*s//sets turn as i allowing Pi to enter Critical section If Pi wants to//*
	while (flag [j] and turn = j);
	{//wait//}
	// if Pi wants to enter & it is Pi’s turn to enter its critical section then Pj waits until Pi completes its critical section//
	critical section*//critical section of Pj//*
	flag [j]=false;

//sets flag as false indicating Pi no longer wants to enter its critical section// remainder section *//rest of Pi's non critical code//* } while (1);*//executing in endless loop//*	*//sets flag as false indicating Pj no longer wants to enter its critical section//* remainder section *//rest of Pj's non critical code//* } while (1); *//excuting in endless loop//*

This solution guarantees the three requirements: mutual exclusion, progress and bounded waiting.

Multi-process solution

Bakery algorithm supports multiple cooperating processes and solves critical section problem for multiple processes.

General functioning of Bakers algorithms

Its based on a scheduling algorithm commonly used in bakeries

On entering the store, each customer receives a number. The customer with the lowest number is served next. The Customer with the lowest name is served first

Functioning of the Bakery algorithm for process

Each process wanting to enter its critical section chooses a number (choose). The numbering scheme always generates numbers in increasing order. If no processes are in their critical section the process with lowest number is allowed to enter its critical section. If two or more processes have been waiting the process with smallest process token number is allowed

Bakery Algorithm for Process i

```
boolean choosing[n];//willingness to enter//
//initialized to false//
int number[n];//number assigned to each process willing to enter its critical section//
//initialized to 0//
do {
choosing[i]=true;  //process i wants to enter its critical section//
number[i]=max(number[0], number[1],…..number [n– 1])+1;//a number is given to the
process//
choosing[i]=false; //sets willing to enter as false to check status of other processes
before entering into its critical section//
for (j=0; j<n; j++)
{
while (choosing[j]);
// checks if there are any other processes  (j = process 0 to n)willing to enter their critical section
(still choosing a number) wait.
//all willing processes have chosen a number//
While ((number[j]!= 0)&&(number [j,j] < number [i,i]));
//process with least token number
critical section
number[i] = 0;
//process i has finished its critical section //
remainder section
//the non critical section of process i//
} while (1);// continuous loop//
```

Note:- the processes enter in their critical-section on a first-come, first-serve basis.

Bakery Algorithm satisfies mutual exclusion, progress and bounded waiting

Drawback of Software Solutions

Processes that are requesting to enter their critical section are busy waiting (consuming processor time needlessly).

Synchronization hardware:

Hardware instructions are available on many systems and showing how they can be used effectively in solving the critical-section problem. Hardware features can make any programming task easier and improve system efficiency particularly for multiprocessor systems. These hardware instructions are executed **atomically**- as one uninterruptable unit. Two such instructions are test_and_set() and swap()

Test and Set

Test and sets lock if no other process in in Critical section

Test and Set Definition	Test and Set

boolean test and set(boolean *target) {
boolean rv = *target;
*target = true;
return rv;
}
//keeps testing the variable target if false sets it to true & return's false//
	do {
	while (test and set(&lock));
	/* do nothing */
	//test variable lock if false set to true//
	//if lock = false => no other process is in its critical section//
	//if lock is already true => some other process is already in its critical section and wait until released//
	/* critical section */
	lock = false;
	//sets the lock false
	/* remainder section */
	} while (true);

Swap

Process i will wait to enter critical section until lock is true. Once lock becomes false I sets the lock and enters to critical section.

void Swap (boolean *a, boolean *b)
{
boolean temp = *a;
*a = *b;
*b = temp:
}
	do {
	key = TRUE;
	while (key == TRUE)
	Swap (&lock, &key);
	//if lock=false=> no other process is in its critical section//
	//(Key = true ,lock = false) swap =>(lock = true key = false) =>process has set the lock//
	// Key = false process can exit while loop & enter critical section//
	//If lock = true => some other process is in its critical section//
	// critical section
	lock = FALSE;
	//remainder section
	}while(True

Semaphore

A **semaphore** S is an **integer variable** that, apart from initialization, is accessed only through two standard **atomic** operations: **wait() and signal()**. The wait() operation was termed P and signal() as V.

The definition of wait() is as follows:	**The definition of signal() is as follows:**

wait(S) {
while (S <= 0)
; // busy wait
S--;
}
	signal(S) {
	S++;
	}

When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption.

The two types of semaphores are 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.

Mutual exclusion using semaphore
Semaphore mutex;
Mutex is a semaphore variable set to 1 .
//If mutex ≤0 => there is some process in its critical section All other processes has to wait//
// if mutex =>0 => there is no process in critical section next process that sets mutex as ≤0 can enter its critical section//
Process Pi:

```
do {
wait(mutex);  //process tries to set mutex as ≤0//
Critical section;   //process i has set mutex as ≤0 & entered the CS//
signal(mutex);
Remainder section;
} while (1);
```

Advantages of semaphores - Easily implemented for n processes
Disadvantages of Semaphores - Require' s busy waiting(spinlock)

Semaphore avoiding busy waiting

When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. Rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore and the state of the process is switched to the waiting state.
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. Process is then placed in the ready queue.

Structure definition typedef struct{ int value;*//semaphore value//* struct process *L; } semaphore.	The definition of the semaphore operations wait and signal should be modified Wait:- o If semaphore is not +ve then the process can block itself o The block operation:- places the process in a waiting queue. Signal:- o If Queue is not empty wakes the next process from queue

Definition of Wait (S)
wait(S)
{
S.value--;*//process wants to enter CS//*
if (S.value < 0)*//another process is already in its CS//*
{
add this process to S.L;*//insert the process to Q//*
block;*//process blocks itself//*
}
}
	Definition of Signal(S)
	signal(S)
	{
	S.value++;*//process finished CS & exiting//*
	if (S.value <= 0)*//if Q is not empty//*
	{
	remove a process P from S.L;
	//select next process from Q//
	wakeup(P);*//wake the next process from Q//*
	}5
	}

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, we consider a system consisting of two processes, P0 and P1, 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 P0 executes wait (S) and then P1 executes wait (Q). When Po executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until Po executes signal(S). Since these signal () operations cannot be executed, Po and P1 are deadlocked.
Starvation a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Classical synchronization Problem

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

The Bounded-Buffer Problem
/ Producer Consumer Problem

- Producer produces (data) into buffer and Consumer consumes (data) from buffer
- Producer is trying to full the buffer & Consumer it trying to empty the buffer
- Synchronization should ensure
    - Producer should not produce on full buffer
    - Consumer should not consume from empty buffer
    - Producer & consumer should not function at the same time. Production & consumption should be mutually exclusive.

Bounded-Buffer Problem
Shared data

semaphore full; *//semaphore to ensure producer dose produce to a full buffer//*
semaphore empty; *//semaphore to ensure consumer dose consume from a empty buffer//*
semaphore mutex; *//semaphore to ensure producer & consumer remain mutually exclusive //*

// initially full = 0, empty = n, mutex = 1//

Process Producer	**Process Consumer**

do {
…
produce an item in next_p
//producer has produced the data to be written into buffer//
…
wait(empty); *//if empty≤0 => no empty buffer space =>producer wait//*
//ensures producer dose not write into a full buffer//
//empty --;producer has filled one empty//
wait(mutex);*//ensures mutual exclusion//*
…
add next_p to buffer
…
signal(mutex);
//producer has exited CS (writing into buffer)//
signal(full);
//full ++; producer has filled buffer by 1 entry//
} while (1);
	do {
	wait(full); *//if full ≤ 0 => no elements in buffer =>buffer empty=>consumer wait//*
	//ensures consumer dose not read from empty buffer//
	wait(mutex); *//ensures mutual exclusion//*
	…
	remove an item from buffer to next_c
	…
	signal(mutex);
	//signals consumer is exiting CS(reading from buffer)//
	signal(empty);
	//empty ++;consumer created 1 empty space by consuming//
	…
	consume the item in next_c
	//consumer is consuming the data from buffer//
	…
	} while (1)

Readers and Writers Problem

Readers and writers that share a data resource. Some of these processes may want only to read (readers) whereas others may want to update (read and write) (writers)
o If two readers access the shared data simultaneously - No adverse effects
o Two writers access the resource simultaneously - Data might become inconsistent
o A writer and a reader access the resource simultaneously - Data might become inconsistent
Synchronization tool has to ensure

o Any no of readers can access the shared data - No reader should wait for other readers
o When writer is accessing neither reader nor writer should access shared resource
o Once a writer is ready, that writer performs its write as soon as Possible

- If a writer is waiting to access the object, no new Readers can start

o Provide mutual exclusion for updating read count/read/write semaphore

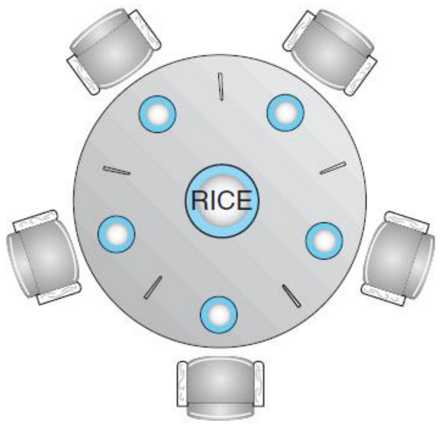
Shared data semaphore write; *//semaphore to ensure consumer dose consume from a empty buffer//* semaphore mutex; *//semaphore to ensure producer & consumer remain mutually exclusive //* int read_count = 0; *//maintains count of number of readers//* *// initially write = 1, mutex = 1, read_count = 0//*	

Process Reader:
 	
wait(mutex);
//provides mutual exclusion for updating read_count//
//process increments readers count defore starting Read//
read_count++;
if (read_count == 1)
wait(write); *r//writer is blocked on first reader*
Signal(mutex);
// mutual exclusion for updating read_count is completed//
Reading is performed
wait(mutex);
//provides mutual exclusion for updating read_count//
//process decrements readers count after finishing Read//
read_count– –;
if (read_count == 0)
signal(write);
//writer signalled on last reader//
signal(mutex);
// mutual exclusion for updating read_count is completed//
	Process Writer:
	
	Wait(write);
	//requesting Mutual exclusion for writing//
	//=> readers =0 && no other writer accessing Data//
	Writing is performed
	Signal(write);
	//writing completed exiting mutual exclusion//

The Dining-Philosophers Problem

5 philosophers share a common circular table surrounded by five chairs

- In the center of the table is a bowl of rice, A chopstick is placed between each pair of adjacent philosophers.
- Each philosopher can either think or eat.
- If a philosopher wants to eat, he will need both chopsticks
- Each philosopher can pick up an adjacent chopstick, when available and put down the chop stick after eating.
- Chopsticks must be picked up and put down one by one.



Solution :
 	
Semaphore chopstick[5]
 	
Algorithm
 	
do {
wait(chopstick[i]);
wait(chopstick[(i+1) % 5]);
. . .
/* eat for awhile */
. . .
signal(chopstick[i]);
signal(chopstick[(i+1) % 5]);
. . .
/* think for awhile */
. . .
} while (true);
	This solution is incorrect: -
	• it allows the system to reach deadlock
	Consider situation
	• all philosopher has picked up their left chopsticks
	• each philosopher is waiting for his right chopstick
	Proposed solutions
	• Allow at most four philosophers to be sitting simultaneously at the table (allow only even No of philosophers)
	• Allow a philosopher to pick up her chopsticks only if both chopsticks are available
	asymmetric solution; that is,
	• an odd philosopher first picks up left chopstick and then right chopstick,
	• an even philosopher first picks up right chopstick and then left chopstick

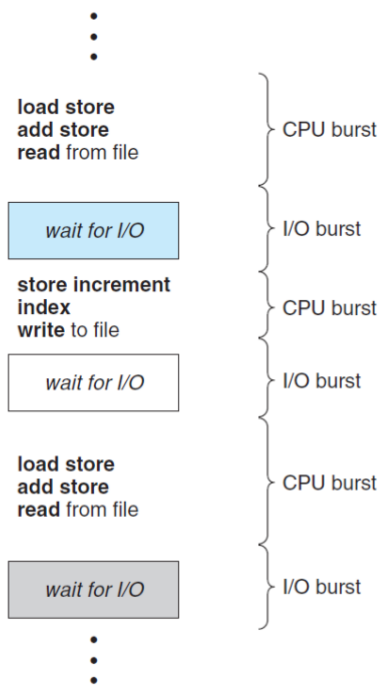
CPU SCHEDULING

Introduction

CPU-I/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes:

- Process execution consists of a cycle of CPU execution and I/O wait.
- Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution (Fig a).
- The durations of CPU bursts have been measured extensively.

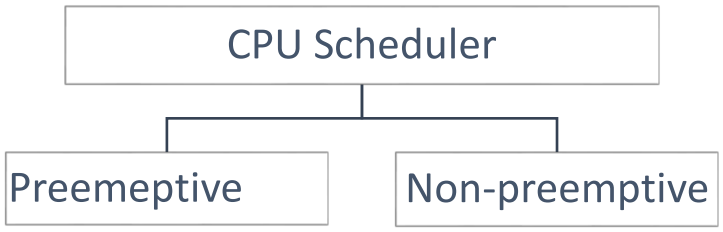


CPU Scheduler
Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler).
The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process. Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.
Dispatcher
Another component involved in the CPU-scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves:

o Switching context
o Switching to user mode
o Jumping to the proper location in the user program to restart that program

The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency.

Types of CPU schedulers
CPU Scheduler can be classified in to two types
Based on when the scheduler will choose the next job for execution



Non Preemptive Scheduler
Once a process is in the running state, it will continue until it terminates or blocks itself

- Cannot force stop a running processThus a Non-preemptive scheduler will select next process Only when previous process has released the CPU & CPU is idle

Preemptive Scheduler
Currently running process may be interrupted and moved to the Ready state by OS

- Can force stop a running process if a higher importance
- (priority) process is available in ready Queue
- Allows for better service
- Prevents any one process from monopolizing
- Provides provision for priority of important processes

Thus, a preemptive scheduler will select next process when

- Previous process has released the CPU & CPU is idle
- Whenever a new process enters the ready queue

Scheduling Criteria
Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

The criteria include the following:

- **CPU Utilization:** Keep CPU utilization as high as possible.
- **Throughput:** Number of processes completed per unit time.
- **Turnaround Time:** Mean time from submission to completion of process.
- **Waiting Time:** Amount of time spent ready to run but not running.
  Amount of time spent waiting in ready Q waiting for CPU
- **Response Time:** Time between submission of requests and first response to the request.

Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms:

i. First-come, First served scheduling
ii. Shortest-Job-First scheduling
iii. Shortest-remaining-time-first scheduling
iv. Priority scheduling
v. Round-Robin scheduling
vi. Multilevel Queue scheduling
vii. Multilevel Feedback Queue scheduling
viii. Multiprocessor scheduling
ix. Real time scheduling

First-Come First-Served (FCFS) Scheduling

- Runs the processes in the order they arrive in the Ready Queue
- Non preemptive algorithm
- Removes a process from the processor only if it goes into the waiting state or terminates

Advantages

- Very simple
- Very little scheduler overhead (low dispatch latency)

Disadvantages

- Long average and worst-case waiting times
- Poor dynamic behavior (convoy effect - short process behind long process)

Analysis

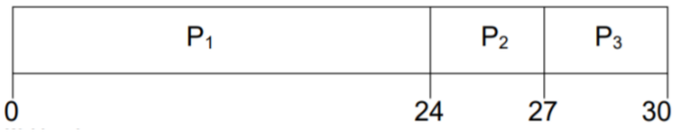
- Good for long processes /batch processing
- Poor performance for short /interactive processes

Example:

Process	Burst Time
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order : P1 , P2 , P3 and are served in FCFS order,

• The Gantt Chart for the schedule is:



<u>Waiting time:</u>

- The waiting time is 0 milliseconds for process P1,

- 24 milliseconds for process P2, and
- 27 milliseconds for process P3.
- Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.

Turnaround time:

- Turnaround time for P1 = 24
- Turnaround time for P2 = 24 + 3
- Turnaround time for P3 = 24 + 3 + 3

Shortest-Job-First scheduling

- A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst.
- When the CPU is available, it is assigned to the process that has the **smallest next CPU burst**. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
- The SJF algorithm can be either preemptive or nonpreemptive. The basic SJF is non preemeptive and shortest remaining time next is SJF preempptive version.

Advantages

- Very simple
- Minimal waiting times

Disadvantages

- little scheduler overhead(some dispatch latency)
- has to rely on user estimates of burst times
- might cause starvation of jobs with long CPU bursts

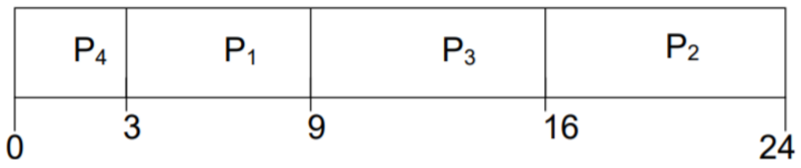
Analysis

- Good for lots of short processes
- No effect for batch processing

Example:

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Suppose that the processes arrive in the order : P1 , P2 , P3 and P4 are served in SJF order,
The Gantt Chart for the schedule is:



<u>Waiting time:</u>

- The waiting time is 3 milliseconds for process P1,
- 16 milliseconds for process P2,
- 16 milliseconds for process P3, and
- 0 milliseconds for process P4.
- Thus, the average waiting time =(3 + 16 + 9 + 0)/4 = 7 milliseconds.

Turnaround time:

- Turnaround time for P1 = 9
- Turnaround time for P2 = 24
- Turnaround time for P3 = 16
- Turnaround time for P4 = 3
- Average Turnaround time = (9+24+16+3)/ 4 = 13 milliseconds

Pre-emptive SJF / Shortest remaining time process next (SRTN)

A pre-emptive SJF algorithm will pre-empt the currently executing process, whereas nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst.

Advantages

- Very simple
- Minimal waiting times

Disadvantages

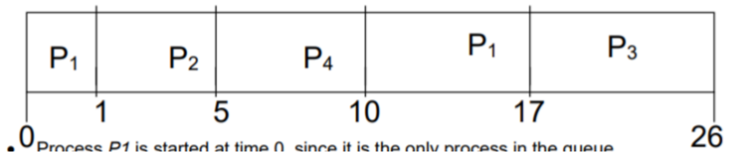
- **little scheduler overhead(some dispatch latency)**
- has to rely on user estimates of burst times
- might cause starvation of jobs with long CPU bursts

Analysis

- Good for lots of short processes
- No effect for batch processing

Process	Arrival time	Burst Time
P1	0	6
P2	1	8
P3	2	7
P4	3	3

Suppose that the processes arrive in the order : P1 , P2 , P3 and P4 are served in SJF order,
The Gantt Chart for the schedule is:


P1
P2
P4
P1
P3
0
1
5
10
17
26
Process P1 is started at time 0, since it is the only process in the queue

- Process P2 arrives at time 1.
- Among P1 and P2 as P2 is having shortest time, P2 will get executed first.
- The remaining time for process P1 (7 milliseconds) is larger than the time required by
- process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.
- After completion of P2, P1 is scheduled.
- And then in the last P4 which is having larger time is sheduled.

Waiting time:

- Waiting time for process p1 = (10 - 1)
- Waiting time for process p2 = (1-1)
- Waiting time for process p3 = (17-2)
- Waiting time for process p4 = (5-3)
- The average waiting time = ((10 - 1) + (1 - 1) + (17-2) + (5-3))/4 = 26/4

= 6.5 milliseconds.

Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

Turnaround time:

- Turnaround time for P1 = (0+8)=8

- Turnaround time for P2 = (7 + 4)
- Turnaround time for P3 = (15+9)
- Turnaround time for P4 = (9 + 5)
- Average Turnaround time = (8+11+24+14) / 4 = 14.25 milliseconds.

Priority Scheduling

- Can be pre-emptive or non-pre-emptive.
- A priority (an integer) is associated with each process. Priorities can be assigned either as Externally (by user) or Internally (by Operating System).
- The CPU is allocated to the process with the highest priority (Smallest integer means highest priority).
- In preemptive priority scheduling, If running process is not the process with highest priority it is pre-empted (forced to release CPU) and CPU will be scheduled for the newly arrived high priority process.
- · Algorithm is initiated each time a process enters ready Q

Advantages

- Important tasks can be processed quicker

Disadvantages

- Starvation or indefinite blocking – low priority processes may never execute
- Solution Aging – increasing the priority of the process. With time

Analysis

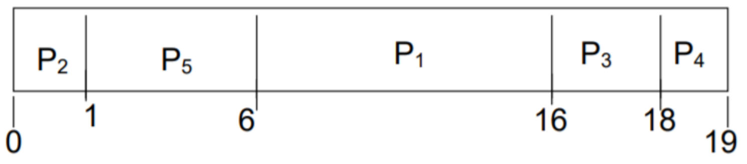
- Good for implementing system rules & priorities
- Might result in starvation of low priority jobs

Non preemptive priority Example

As an example, consider the following set of processes, assumed to have arrived at time 0, in the order P1, P2, ….., P5, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Gantt Chart



- Process P2 is started at time 0, since it is having the highest priority among the processes in the queue.
- After execution of P2 process, process P5 is started at time 1.
- Then P1,P3 and P4 as P2 processes are executed. is having shortest time, P2 will get executed first.

Waiting time:

- Waiting time for process p1 = 6
- Waiting time for process p2 = 0

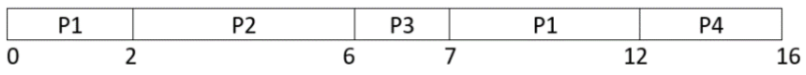
- Waiting time for process p3 = 16
- Waiting time for process p4 = 18
- Waiting time for process p5 = 1
- The average waiting time = (6+0+16+18+1)/5 = 41/5 = 8.2 milliseconds.

Turnaround time:

- Turnaround time for P1 = 6 + 10
- Turnaround time for P2 = 0 + 1
- Turnaround time for P3 = 16 + 2
- Turnaround time for P4 = 18 + 1
- Turnaround time for P5 = 1 + 5
- Average Turnaround time = (16+1+18+19+6) / 5 =   12 milliseconds

Preemptive priority example

Process	Arrival time	Burst Time	Priority
P1	0	7	3
P2	2	4	1
P3	4	1	2
P4	5	4	4



Waiting time:

- Waiting time for process p1 = 7-2-0=5
- Waiting time for process p2 = 2-2=0
- Waiting time for process p3 = 6-4=2
- Waiting time for process p4 = 12-5=7
- The average waiting time = (5+0+2+7)/4 = 3.5 milliseconds.

Turnaround time:

- Turnaround time for P1 = 11
- Turnaround time for P2 = 4
- Turnaround time for P3 = 3
- Turnaround time for P4 = 11
- Average Turnaround time = (11+4+3+11) / 4 =   7.25 milliseconds

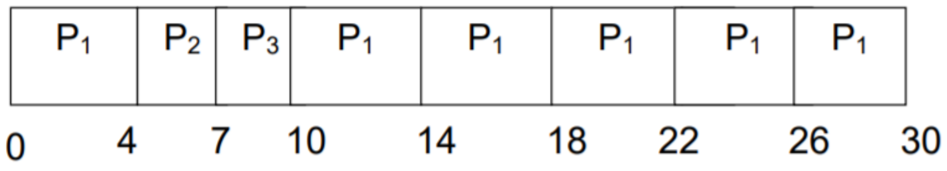
Round Robin Scheduling

- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes.
- A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds.

- The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to t time quantum.
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after t time quantum, and dispatches the process. One of two things will then happen:
    - The process may have a CPU burst of less than t time quantum.
        - In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.
    - Otherwise, if the CPU burst of the currently running process is longer than t time quantum, the timer will go off and will cause an interrupt to the operating system.
        - A context switch will be executed, and the process will be put at the tail of the ready queue.
        - The CPU scheduler will then select the next process in the ready queue.

 The average waiting time under the RR policy is often long.
 Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P1	24
P2	3
P3	3



Quantum time = 4 ms

If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds.
- Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2
- Since process P2 does not need 4 milliseconds, it quits before its time quantum expires.
- The CPU is then given to the next process, process P3.
- Once each process has received t time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is:

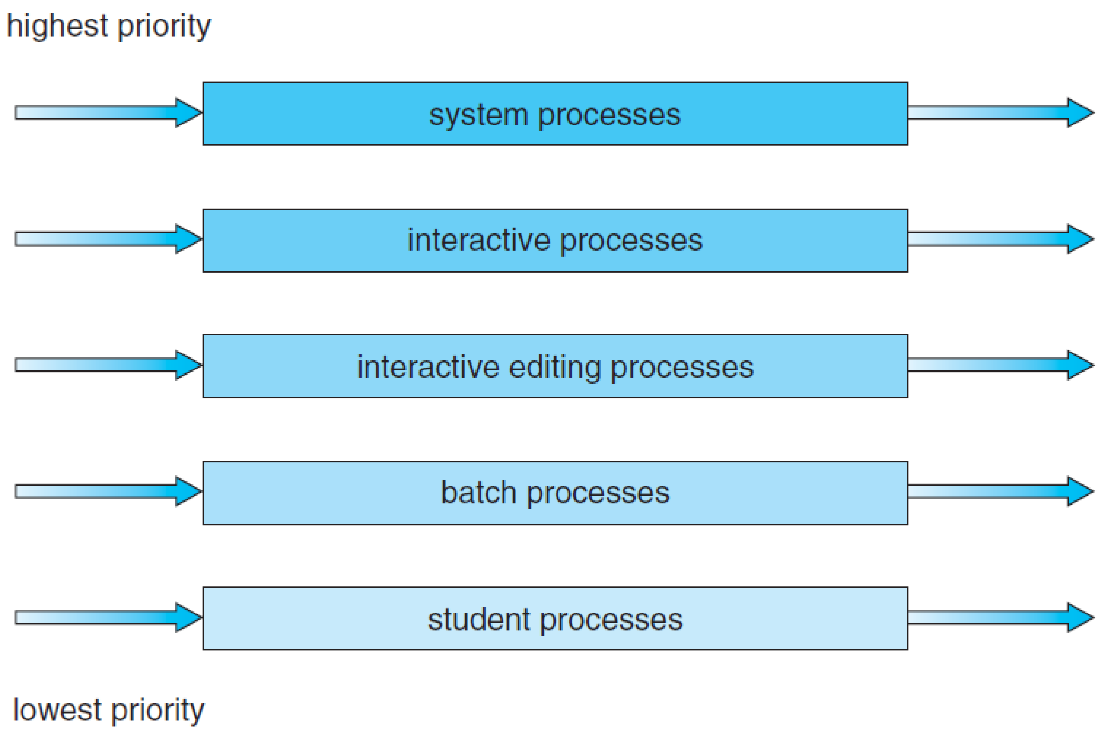
Waiting time:
- Waiting time for process p1 = (10 - 4)
- Waiting time for process p2 = 4
- Waiting time for process p3 = 7
- The average waiting time = ((10-4)+4+7) /3 = 17/3 = 5.66 milliseconds.

Turnaround time:
- Turnaround time for P1 = (10-4) + 24
- Turnaround time for P2 = 4 + 3
- Turnaround time for P3 = 7 + 3
- Average Turnaround time = (30+7+10) / 3 = 15.6 milliseconds

Multilevel Queue scheduling

- Another class of scheduling algorithm has been created for situations in which processes are easily classified into different groups. For example, a common division is made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.

- A multilevel queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type
- Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes.
    - The foreground queue might be scheduled by an RR algorithm,
    - while the background queue is scheduled by an FCFS algorithm.

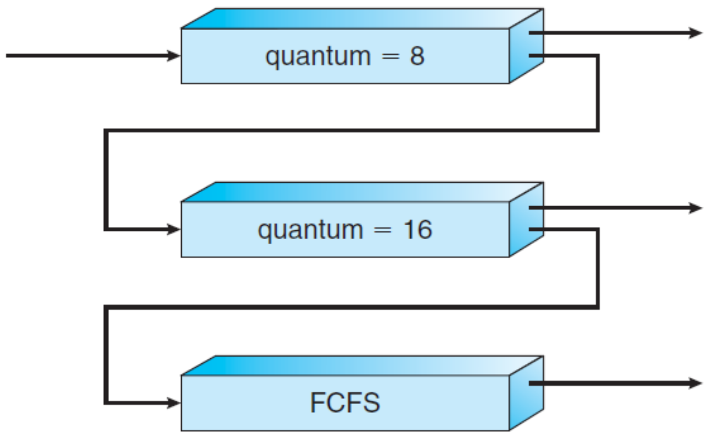


Advantages
Tasks can be categorized
Disadvantages

- Could result in starvation of low priority jobs
- Once a process is assigned to a Q it cannot be changed
- Solution Multi Level Feedback Queues

Multilevel Feedback Queue scheduling

- The multilevel feedback-queue scheduling algorithm, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and



interactive processes in the higher-priority queues.

- Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

For example, consider a multilevel feedback-queue scheduler with three queues, numbered from 0 to 2 .

- The scheduler first executes all processes in queue 0.
- Only when queue 0 is empty will it execute processes in queue 1.
- Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty.
- A process that arrives for queue 1 will preempt a process in queue 2.
- A process that arrives for queue 0 will in turn, preempt a process queue 1.

In general, a multilevel feedback-queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher priority queue
- The method used to determine when to demote a process to a lower priority queue
- The method used to determine which queue a process will enter when that process needs service

Real time scheduling

Correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced. Tasks or processes attempt to control or react to events that take place in the outside world during execution. These events occur in "real time" and process must be able to keep the events with them.

Examples: process control in industrial plants, robotics, air traffic control, telecommunications, military command and control systems.

There are two types of real time systems. They are:

- Hard real-time task - one that must meet its deadline, otherwise it will cause unacceptable damage or a fatal error to the system
- Soft real-time task - has an associated deadline that is desirable but not mandatory and may find small relaxation to schedule and complete the task even if it has passed its deadline.

Real-time operating systems have requirements in five general areas:

- Determinism - whether the system has sufficient capacity to handle all requests within the required time
- Responsiveness - Concerned with how long, after acknowledgment, it takes an operating system to service the interrupt
- User control - User should be able to distinguish between hard and soft tasks and to specify relative priorities within each class
- Reliability - Real-time systems respond to and control events in real time so loss or degradation of performance may have catastrophic consequences.
- Fail-soft operation - A characteristic that refers to the ability of a system to fail in such a way as to preserve as much capability and data as possible.

- Real-time operating systems are designed with the objective of starting real-time tasks as rapidly as possible and emphasize rapid interrupt handling and task dispatching. Priorities provide a crude tool and do not capture the requirement of completion (or initiation) at the most valuable time

A. Rate Monotonic scheduling

The rate-monotonic scheduling algorithm schedules periodic tasks using a static priority policy with preemption. If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority process.Uponentering the system, each periodic task is assigned a priority inversely based on its period. The shorter the period, the higher the priority; the longer the period, the lower the priority. The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often.
Furthermore, rate-monotonic scheduling assumes that the processing time of a periodic process is the same for each CPU burst.

B. Dead Line Scheduling

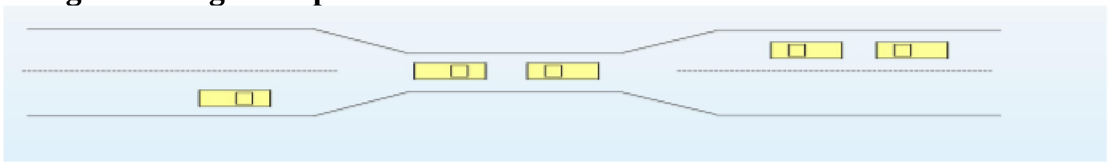
The earlier the deadline, the higher the priority; the later the deadline, the lower the priority.

Deadlocks

Introduction to deadlock

A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.

Bridge Crossing Example



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

System has 2 tape drives. P1 and P2 each hold one tape drive and each needs another one.

System Model

1. Resource types R_1, R_2, . . ., R_m
2. CPU cycles, memory space, I/O devices includes resources
3. Each resource type R_i has W_i instances.

Each process utilizes a resource as follows:

• request
• use
• release

Deadlock Characterization

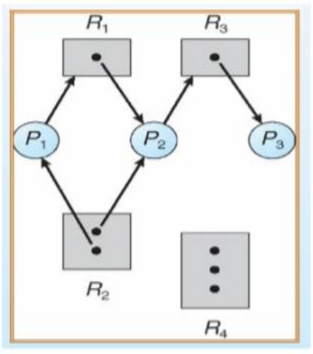
Deadlock can arise if **four conditions hold simultaneously.**

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

Resource-Allocation Graph

A set of vertices V and a set of edges E.
V is partitioned into two types: $P = \{P_1, P_2, \ldots, Pn\}$, the set consisting of all the processes in the system. $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
E (edges) is partitioned into two types as well:
Request edge – directed edge $P_1 \rightarrow R_j$ [defining the process is requesting the resource]
Assignment edge – directed edge $R_j \rightarrow P$ [defining the resource has been assigned to process]

Example of resource allocation graph



Description

- Processes: - P1, P2, P3.
- Resources:-R1, R2, R3, R4.
- R1, R3:- resources with only one instance
- R2:- resource with two instances.
- R4:- resource with four instances
- Process P1 is holding Resource R2 & requesting R1
- Process P2is holding Resource R1 , R2 and requesting R3
- Process P3 is holding resource R3
- Resource R4 Is not assigned to any process

Basic Facts - Describing a deadlock

- if the graph contains no cycles, then no process in the system is deadlocked.
- If the graph does contain a cycle, then a deadlock may exist
- If each resource type has exactly one instance, then a cycle => that a deadlock has occurred.
- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.

Methods for Handling Deadlocks

• Deadlock Prevention - Ensure that the system will never enter a deadlock state
• Deadlock Avoidance - Allow the system to enter a deadlock state and then recover.
• Deadlock Detection and recovery - Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.On detecting the deadlock the system will be recovered.

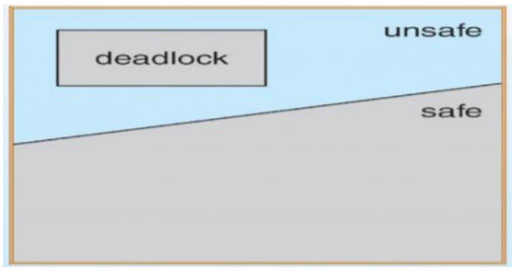
Safe State

- 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 $<P_1, P_2, \ldots, P_n>$ is safe if for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j<I$.

- If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished. When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
- When Pi terminates, Pi+1 can obtain its needed resources, and so on.

Safe, Unsafe , Deadlock State

• 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.



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 we can prevent the occurrence of a deadlock.

Denying/prevent one of these 4 necessary conditions

o Mutual Exclusion condition
o Hold and Wait condition
o No Preemption condition
o Circular Wait condition

- To Prevent/deny Mutual Exclusion condition
  Use only shareable resource and make all resources as shareable. But it's impossible for practical system, where some of the resources like printer cannot be shared.
- Prevent/Deny Hold and Wait condition
  - Method 1 Pre-allocation protocol
    - Require processes to request all the resources before starting. So, process will be allocated all the resources before starting and a process never has to wait for what it needs. The main drawback here is it leads to low utilization of resources and sometimes process may not know required resources at start of run.
  - Method 2
    - Process can request resources only when the process has none. Process must release all holding resources. Request all immediately needed resources. Process never goes into waiting while holding resources.
- Prevent/deny No Preemption condition
  - If a process that is holding some resources requests another resource that cannot be immediately allocated then the process goes into waiting, all resources currently held by the waiting process are released.
  - Preempted resources are added to the list of resources required by the process
  - Process will be restarted only when it can regain its old resources & the new requested resources
  - The draw back here is Some resources (e.g. printer, tap drives) cannot be preempted without serious problems and it may require the job to restart
- Prevent/Deny Circular Wait condition
  (Order resources) each resource type is assigned a unique integer. Process can request for resources only in increasing order. If a process needs several instances of the same resource. It should issue a single request for all of them.

Deadlock Avoidance

- Processes must tell OS in advance how many resources they will request
- OS Should never allocates resources in a way that could lead to a deadlock

Deadlock Avoidance with Resource-Allocation Graph

- This algorithm can be used only for one instance of each resource type.

Definitions

In addition to the request edge and assignment edge a claim edge is also introduced.

Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j in future. Claim edge is Represented by a dashed line.

The algorithm is executed when a process requests a resource. Suppose that process $\mathbf{P_i}$ requests resource $\mathbf{R_j}$. **The request can be granted only If converting the request edge $\mathbf{P_i \rightarrow R_j}$ to an assignment edge does not result in a cycle in the resource-allocation graph.** i.e., a cycle detection algorithm is used.

o If no cycle exits, the resource Rj is assigned to process Pi

o Else the process Pi will have to wait.

Banker's Algorithm

Applicable to system with multiple instances of resource types.

Each process must declare max requirement of each resource type. Max requirement should not exceed total resources. 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.

Working

Banker's algorithm runs each time when a process requests resource. Bankers algorithm will make requested allocation only when it leads the system in safe state.

Data Structures for the Banker's Algorithm

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

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

A. 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 i such that both:
     Finish [i] = false
     $Need_i \leq Work$
    If no such i exists, go to step 4.
3.  Work = Work + $Allocation_i$.
    Finish[i] = true go to step 2.
4.  If Finish [i] == true for all i, then the system is in a safe state.
```

B. **Resource-Request algorithm for Process Pi**

Request = request vector for process P_i.

• If Requesti [j] = k then process Pi wants k instances of resource type R_j.

1. If $\text{Request}_i \leq \text{Need}_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $\text{Request}_i \leq \text{Available}$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

    Available = Available = Request_i;

    Allocationi = Allocationi + Request_i;

    $\text{Need}_i = \text{Need}_i - \text{Request}_i$

If safe $\Rightarrow$ the resources are allocated to Pi.

If UNsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored.

Example of Banker's Algorithm

Total No. processes:- 5 [P0, P1, P2, P3, P4.]

Resource types:- 3 [A, B, C.]

Total Instances of each resource type

- A:- 10 ; B:- 5 ; C:- 7

Snapshot at time T0:

	Allocation	Max	Available
	ABC	ABC	ABC
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

Each process has declared the maximum Nos for each resource type required to complete execution. Declaration is listed in under **Max**

Allocations listed under **Allocation**

Available shows Nos of free resources available with the system

Finished indicates if process is executing /finishes.

Initialize: calculate Need For each process

- Need should list Nos for each resource type further required to complete execution
- Need = Max – Allocation.
- Mark all executing processes finished as false

	Finished	Allocation	Max	Need	Available
		ABC	ABC	A B C	ABC
P0	False	0 1 0	7 5 3	7 4 3	3 3 2
P1	False	2 0 0	3 2 2	1 2 2	
P2	False	3 0 2	9 0 2	6 0 0	
P3	False	2 1 1	2 2 2	0 1 1	
P4	False	0 0 2	4 3 3	4 3 1	

Repeat until all processes finished execution

- Find next executing process such that Need $\leq$ Available
- Checks Need for P0,as Need is higher it checks for P1.
- For P1 Need (1,2,2) $\leq$ Available (3,3,2)
- //assume Resources are allocated to P1
- P1 has completed execution P1 has released resources//
- Mark P1 as finished (i.e., P1 finished –true)

- Like this all other process will execute?
- Executing safety algorithm shows that there is safe sequence <P1,P3,P4,P0,P2> and satisfies safety requirement. So there is no deadlock in the system.
- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?

Note : Look notes for problems

Deadlock Detection and Recovery

Allow system to enter deadlock state, detects the deadlock and recovers from deadlock
=>Requires

- a deadlock detection algorithm
- a deadlock recovery algorithm

Deadlock detection with Single Instance of Each Resource Type

• Maintain wait-for graph
• Nodes are processes.
• $P_i \rightarrow P_j$ if Pi is waiting for P_j [inherited from RAG]
• Periodically invoke an algorithm that searches for a cycle in the graph.
• An algorithm to detect a cycle in a graph, where n is the number of vertices in the graph.

Several Instances of a Resource Type

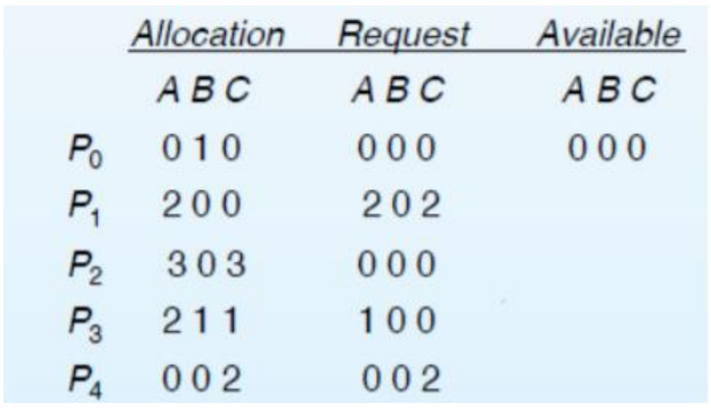
• **Available**: A vector of length m indicates the number of available resources of each type.
• **Allocation**: An n x m matrix defines the number of resources of each type currently allocated to each process.
• **Request**: An n x m matrix indicates the current request of each process. If Request $[i_j] = k$, then process P_i is requesting k more instances of resource type. R_j.

Detection Algorithm

```
1. Let Work and Finish be vectors of length m and n, respectively
   Initialize:
   a. Work = Available
   b. For i = 1,2, …, n,
      if $Allocation_i \neq 0$ then
       Finish[i] = false;
      Otherwise
       Finish[i] = true.
2. Find an index i such that both:
   a. Finish[i] == false
   b. $Request_i \leq Work$

   If no such i exists, go to step 4
3. Work = Work + $Allocation_i$
   Finish[i] = true
   go to step 2.
4. If Finish[i] == false, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover,
   if Finish[i] == false, then $P_i$ is deadlocked.
```

Example of Detection Algorithm
• Five processes P0 through P4; three resource types
A (7 instances), B (2 instances), and C (6 instances).
• Snap Shot at Time T0



	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Sequence <P0,P2,P3,P4,P1> will result in Finish[i] = true for all i and so concludes that there is no deadlock.

But if the Process P2 request a additional resource type as [0,0,1] it will lead to deadlock.

Detection-Algorithm Usage
When, and how often, to invoke depends on:
1. How often a deadlock is likely to occur?
2. How many processes will need to be rolled back?
one for each disjoint cycle If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Recovery from Deadlocks

- Process Termination
- Resource Preemption

Recovery from Deadlock with Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.

To abort one process at a time in which order should we choose to abort?
• Priority of the process.
• How long process has computed, and how much longer to completion?
• Resources the process has used. Resources process needs to complete.
• How many processes will need to be terminated?
• Is process interactive or batch?

Recovery from Deadlock: Resource Pre-emption

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