After Mid first class: (Process Synchronization and Semaphores)

Process Synchronization

Concurrent access to shared data may result in data inconsistency between 2 processes. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Bounded and Unbounded Buffer

The unbounded buffer places no practical limit on the size of the buffer.

The bounded buffer assumes a fixed buffer size.

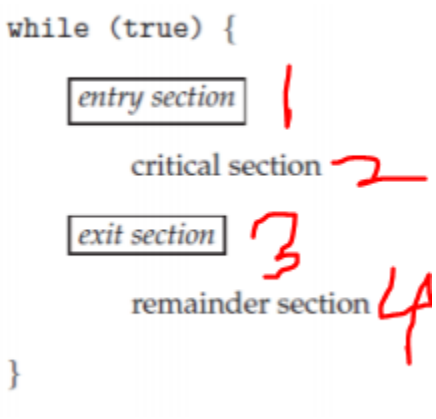
Race Condition

A situation like this, where 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.

The Critical-Section Problem

Critical-Section is a segment or block of code, in which the 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

General Structure of a Code Having Critical Section:



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.

Requirements for Solution for a Critical-Section Problem:

1• **Mutual Exclusion**: Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2• Progress: No process can execute its critical section again just after finishing the execution of it, the process has to finish the execution of exit section and then process is allowed to enter the critical section by taking the permission.

3• **Bounded Waiting:** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Solution of critical section problem:

Paterson's Solution to Bounded Buffer System:

Peterson's Algorithm is used to synchronize two processes. It uses two variables, a bool array flag of size 2 and an int variable turn to accomplish it.

• Applicable for Two Processes only

• Suppose Process i and j (numeric values: 0,1)

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

while (true) {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

}
```

After Mid, 8march, 2nd class: (Semaphores)

Semaphore:

Semaphore is a mechanism that can be used to provide synchronization of processes. (যখন একটা process তার critical section এ প্রবেষ করল তখন semaphore, critical section কে lock করে দেয়, তখন অন্ন কোন process এই critical section এ ধুক্তে পারে না, এর পর যখন critical section এর কাজ শেষ করে বের হয়ে যাবে, তখন সে lock টা খুলে দেয়)

Semaphore is a variable with an underlying counter, this variable is unsigned integer(means not negative)

Variable যদি '-' হই → ব্লক,lock

Variable যদি '+' হই → unlock

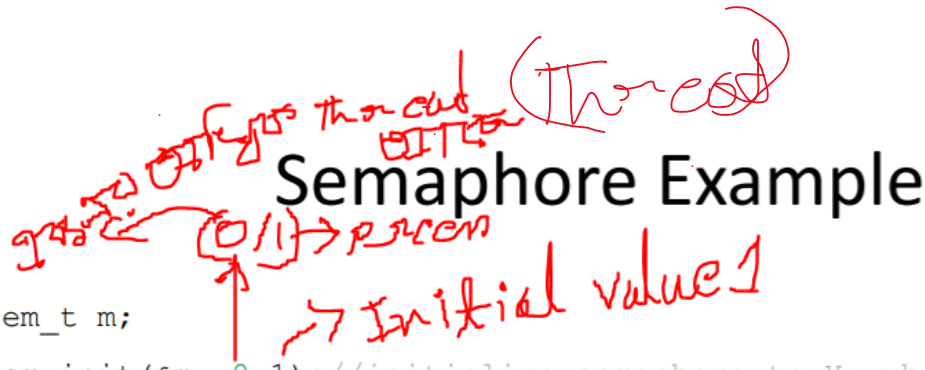
Two functions on a semaphore variable

– Up/post increments the counter

– Down/wait decrements the counter and blocks (০ এর নিচে গেলে ব্লক হয়ে যাবে, সেই process আর critical section এ ঢুকতে পারবে না)

A semaphore with init value 1 acts as a simple lock (binary semaphore = mutex) (0/1)

Semaphore Example



```
sem_t m;
sem_init(&m, 0,1);//initialize semaphore to X; what should X be?
sem_wait(&m);
// critical section here
sem_post(&m);
```

Solving the Producer-consumer Problem using Semaphores:

Need two semaphores for signaling

– One to track empty slots, and make producer wait if no more empty slots

– One to track full slots, and make consumer wait if no more full slots

• One semaphore to act as mutex for buffer

```
int main(int argc, char *argv[]) {
 // ...
 sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with ...
 sem_init(&full, 0, 0); // ... and 0 are full
 sem_init(&mutex, 0, 1); // mutex=1 because it is a lock
 // ...
}
```

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
 int i;

 for (i = 0; i < loops; i++) {
   sem_wait(&empty);
   sem_wait(&mutex);
   put(i);
   sem_post(&mutex);
   sem_post(&full);
 }
}
```

```
void *consumer(void *arg) {
 int i;

 for (i = 0; i < loops; i++) {
   sem_wait(&full);
   sem_wait(&mutex);
   int_ tmp = get();
   sem_post(&mutex);
   sem_post(&empty);
   printf("%d\n", tmp);
 }
}
```

Reader Writer Problem:

• One database

• Some processes want to read only (Readers)

• Some processes want to read and write (Writers)

Problem: ২writer এক্ সাথে হোলে সমস্যা , কার জায় গায় কে লিখবে!

কিন্তু ২ reader এক সাথে হোলে সমস্যা নাই। আবার reader,write একসাথে হতে পারবে না।

– Reader and Reader →ok

– Reader and writer → not ok

– Writer and reader → not ok

– Writer and Writer → not ok

Reader

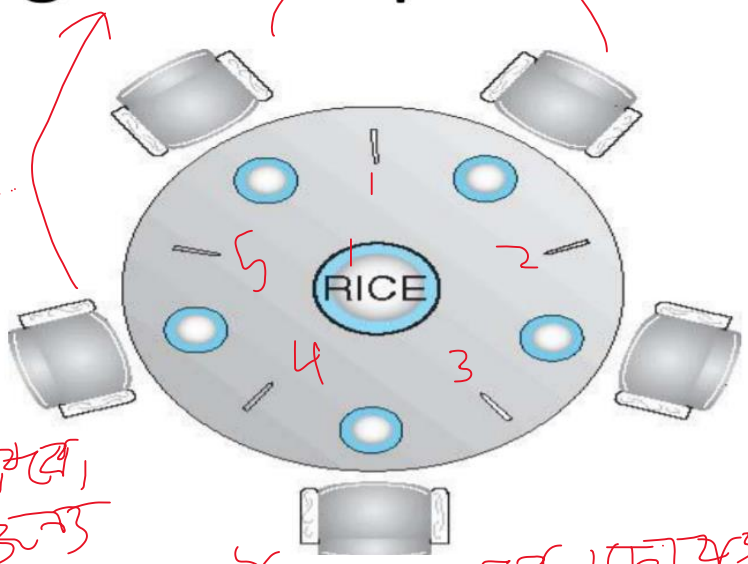
Writer

```
semaphore mutex, wrt;
int readcount;
```

```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
} while (TRUE);
```

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
} while (TRUE);
```

Dining-Philosophers Problem



Shared data – Bowl of rice (data set) and need 2 chopsticks – Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem (Cont.)

```
semaphore chopstick[5];

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

After Mid, 11march, 3rd class: (Deadlock)

What is Deadlock?

Ans: A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.(এক এক্ টা process এক একটা resource আটকায় রাখছে)

Example – System has 2 disk drives – P1 and P2 each hold one disk drive and each needs another one.

Processes P1 , P2 , . . ., P n •

Resource types R1 , R2 , . . ., R m (CPU cycles, memory space, I/O devices)

Each resource type Ri has a number of instances.(2 টা printer)

Each process utilizes a resource as follows:

– request

– use

– release

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.(যেটা ধরে আছে সেটা ছারবে না, আরেক্তা না পাওয়া পরযুনত)

No preemption: A resource can't be preempted from the process by another process, forcefully. For example, if a process P1 is using some resource R, then some other process P2 can't forcefully take that resource. The process P2 can request for the resource R and can wait for that resource to be freed by the process P1.(1printer 100 page print না দিয়ে printer ছারবে না)

Circular wait: Circular wait is a condition when the first process is waiting for the resource held by the second process, the second process is waiting for the resource held by the third process, and so on. At last, the last process is waiting for the resource held by the first process. So, every process is waiting for each other to release the resource and no one is releasing their own resource.

এই ৪টা situation একসাথে আমার system এ occur করে tahole dead lock হবে।

Resource-Allocation Graph:

A set of vertices V and a set of edges E

V is partitioned into two types:

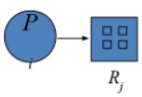
P = {P1, P2, …, Pn}, the set consisting of all the processes in the system

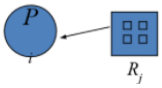
R = {R1, R2, …, Rm}, the set consisting of all resource types in the system

request edge – directed edge Pi → Rj

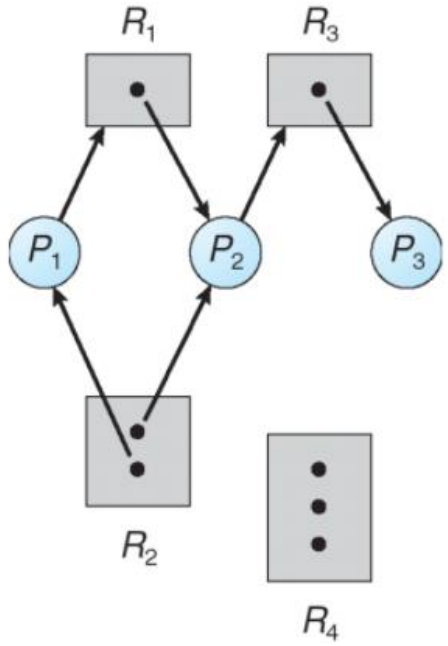
assignment edge – directed edge Rj → Pi

Resource-Allocation Graph (Cont.)

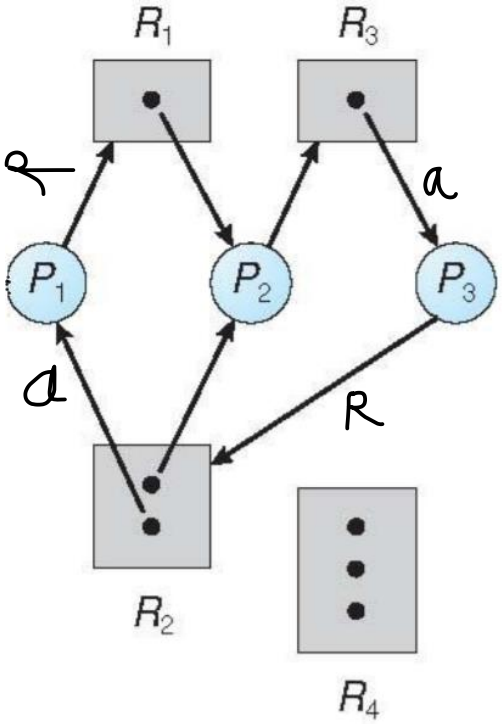
- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

Example of a Resource Allocation Graph:



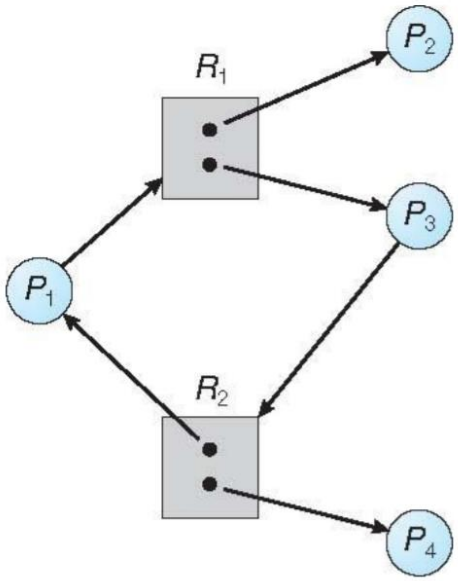
Resource Allocation Graph With A Deadlock:



R = Request

a = Assign

Graph With A Cycle But No Deadlock:



Methods of Handling Deadlocks:

1. Prevent or avoid
2. Detect and recover
3. Ignore (leave it to application softwares)

1 Deadlock Prevention:

Deadlock happens only when Mutual Exclusion, hold and wait, No preemption and circular wait holds simultaneously. If it is possible to violate one of the four conditions at any time then the deadlock can never occur in the system.

Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need

The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

The idea behind the approach is very simple that we have to fail one of the four conditions but there can be a big argument on its physical implementation in the system.

Mutual Exclusion –

not required for sharable resources; must hold for non-sharable resources

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

No Preemption – – If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released – Preempted resources are added to the list of resources for which the process is waiting – Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Safe state: (making a serial of all process as process's requests can be satisfied by currently available resources)

A state of the system is called safe if the system can allocate all the resources requested by all the processes without entering into deadlock.

System is in safe state if there exists a sequence of ALL the processes in the systems such that for each Pi , the resources that Pi can still request can be satisfied by currently available resources.

That is:

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

– When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate.

– When Pi terminates, Pi +1 can obtain its needed resources, and so on.

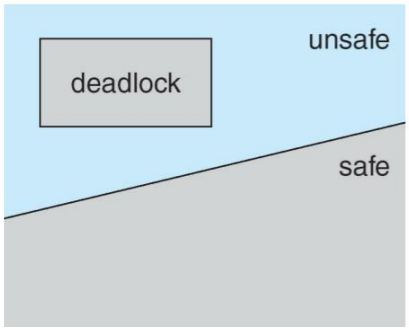
Basic Facts:

If a system is in safe state ⇒ no deadlocks

If a system is in unsafe state ⇒ possibility of deadlock

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

Safe, Unsafe , Deadlock State



Dead lock, Avoidance algorithms: (2types)

• Single instance of a resource type – Use a resource-allocation graph (printer থাকলে 1টাই printer)

• Multiple instances of a resource type – Use the banker's algorithm

Resource-Allocation Graph Scheme: (unsafe state এ জেতেই দিব না)

• Claim edge Pi → Rj indicated that process Pj may request resource Rj ; represented by a dashed line

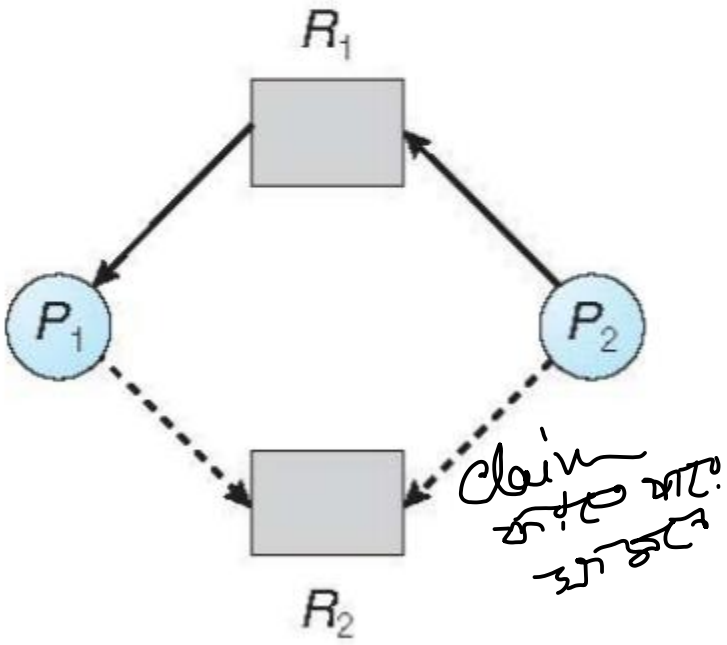
• Claim edge converts to request edge when a process requests a resource

• Request edge converted to an assignment edge when the resource is allocated to the process

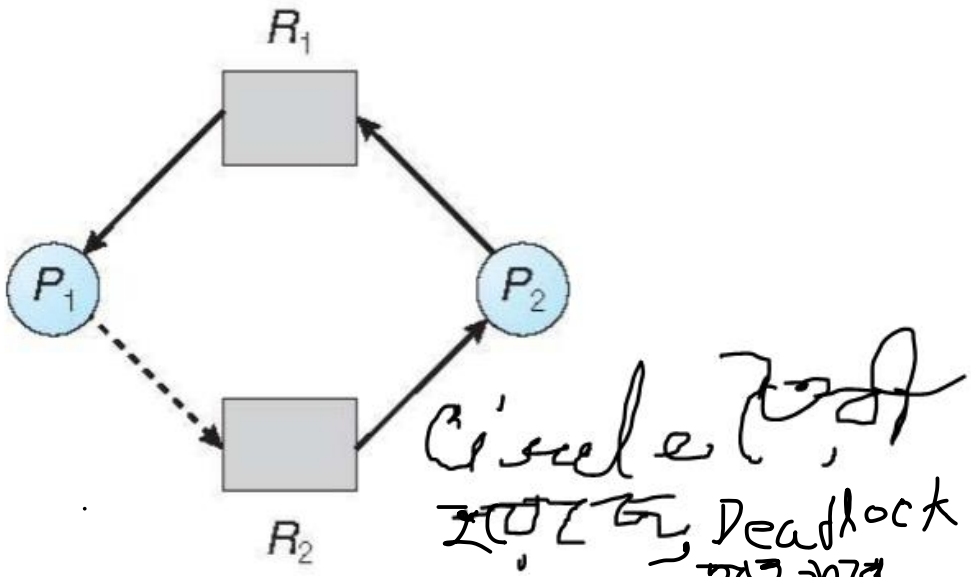
• When a resource is released by a process, assignment edge reconverts to a claim edge

• Resources must be claimed a priori in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm:

• Suppose that process Pi requests a resource Rj

• The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

After Mid, 15march, 4rt class: (Deadlock 2)

Banker's Algorithm: (deadlock avoidance, deadlock detection)

the banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible **amounts of all resources**, after that it makes an check to test for possible activities, before deciding whether allocation should be allowed to continue.

Data Structures for the Banker's Algorithm

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

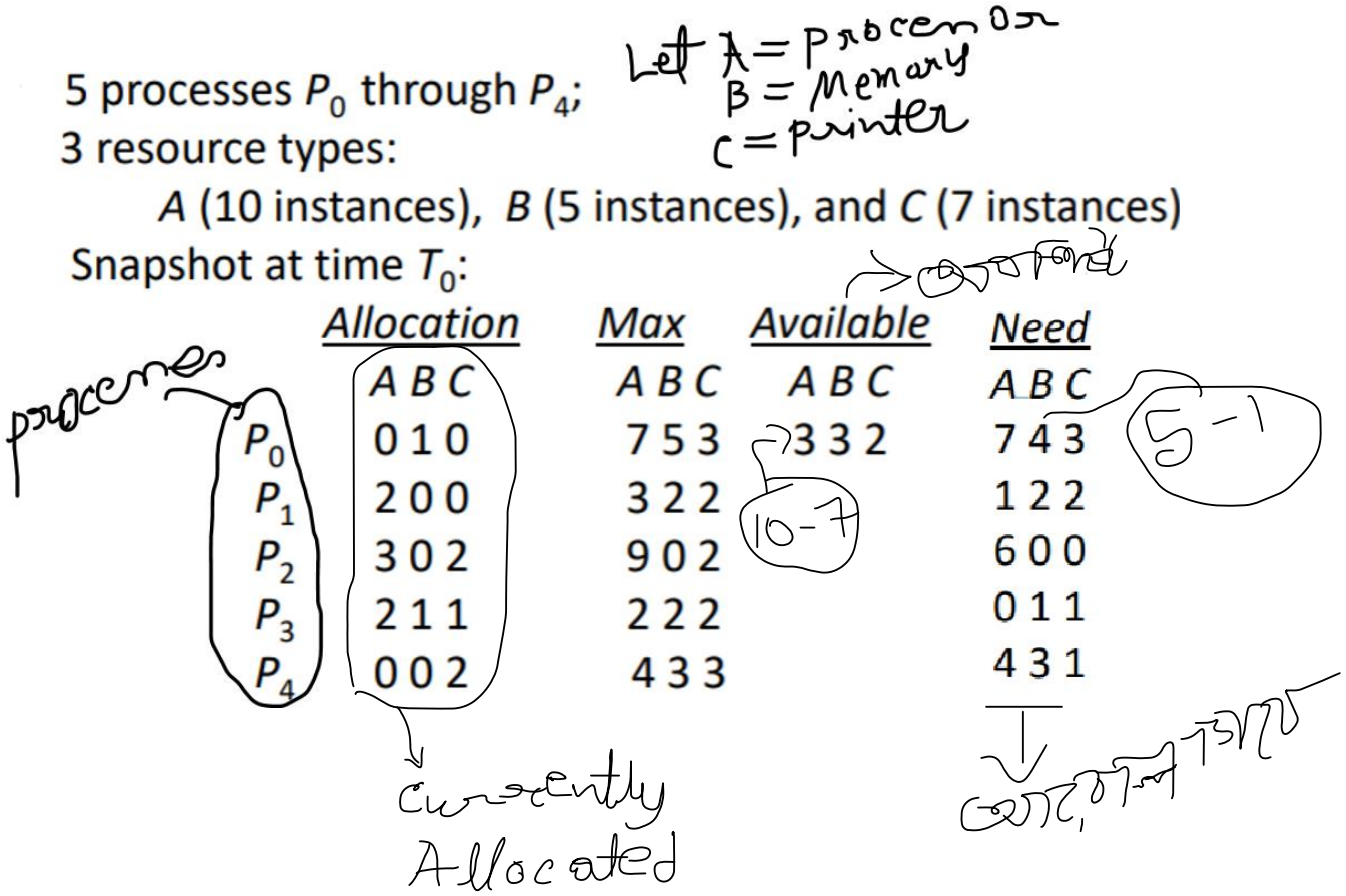
• Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available

• Max: n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj

• Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj

• Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task Need [i,j] = Max[i,j] – Allocation [i,j]

Example of Banker's Algorithm:



Safety Algorithm Example:

m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work = | 3 | 3 | 2 |

	0	1	2	3	4
Finish =	false	false	false	false	false

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $Need_0$ > Work (7,4,3 > 3,3,2)

So P_0 must wait — But Need ≤ Work

For i = 1 — Step 2

$Need_1$ = 1, 2, 2

Finish [1] is false and $Need_1$ < Work (1,2,2 < 3,3,2)

So P_1 must be kept in safe sequence

Work = Work + $Allocation_1$ (3, 3, 2 + 2, 0, 0) — Step 3

Work = | 5 | 3 | 2 |

	0	1	2	3	4
Finish =	false	true	false	false	false

For i = 2 — Step 2

$Need_2$ = 6, 0, 0

Finish [2] is false and $Need_2$ > Work (6, 0, 0 > 5, 3, 2)

So P_2 must wait

For i=3 — Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $Need_3$ < Work (0, 1, 1 < 5, 3, 2)

So P_3 must be kept in safe sequence

Work = Work + $Allocation_3$ (5, 3, 2 + 2, 1, 1) — Step 3

Work = | 7 | 4 | 3 |

	0	1	2	3	4
Finish =	false	true	false	true	false

For i = 4 — Step 2

$Need_4$ = 4, 3, 1

Finish [4] = false and $Need_4$ < Work (4, 3, 1 < 7, 4, 3)

So P_4 must be kept in safe sequence

Work = Work + $Allocation_4$ (7, 4, 3 + 0, 0, 2) — Step 3

Work = | 7 | 4 | 5 |

	0	1	2	3	4
Finish =	false	true	false	true	true

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and Need < Work (7, 4, 3 < 7, 4, 5)

So P_0 must be kept in safe sequence

Work = Work + $Allocation_0$ (7, 4, 5 + 0, 1, 0) — Step 3

Work = | 7 | 5 | 5 |

	0	1	2	3	4
Finish =	true	true	false	true	true

For i = 2 — Step 2

$Need_2$ = 6, 0, 0

Finish [2] is false and $Need_2$ < Work (6, 0, 0 < 7, 5, 5)

So P_2 must be kept in safe sequence

Work = Work + $Allocation_2$ (7, 5, 5 + 3, 0, 2) — Step 3

Work = | 10 | 5 | 7 |

	0	1	2	3	4
Finish =	true	true	true	true	true

Finish [i] = true for $0 \le i \le n$ — Step 4

Hence the system is in Safe state

The safe sequence is P_1, P_3, P_4, P_0, P_2

Resource-Request Algorithm:

Example: P_1 Request (1,0,2)

P_1 process Request → 1 A, 0 B, 2 C

Resource → Request → Grant → Safe sequence

	A	B	C
$Request_1$ =	1	0	2

To decide whether the request is granted we use Resource Request algorithm

Step 1: $Request_1$ (1, 0, 2) < $Need_1$ (1, 2, 2)

Step 2: $Request_1$ (1, 0, 2) < Available (3, 3, 2)

Step 3:

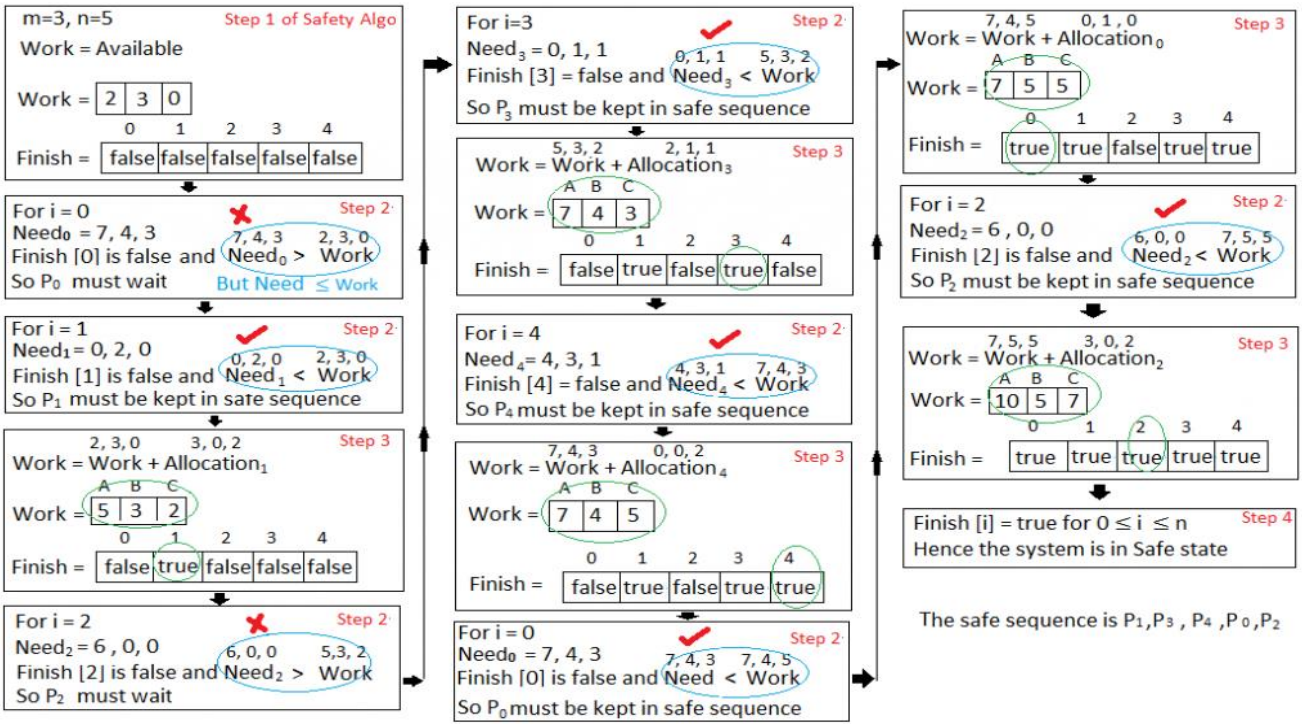
Available = Available – $Request_1$

$Allocation_1$ = $Allocation_1$ + $Request_1$

$Need_1$ = $Need_1$ - $Request_1$

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

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data



- **Now, Home work:**
- **Can request for (3,3,0) by P_4 be granted?**
- **Can request for (0,2,0) by P_0 be granted?**

After Mid, 1march, 5th class: (Deadlock 2)

Deadlock Detection :

• Allow system to enter deadlock state

• Detection algorithm

• Recovery scheme

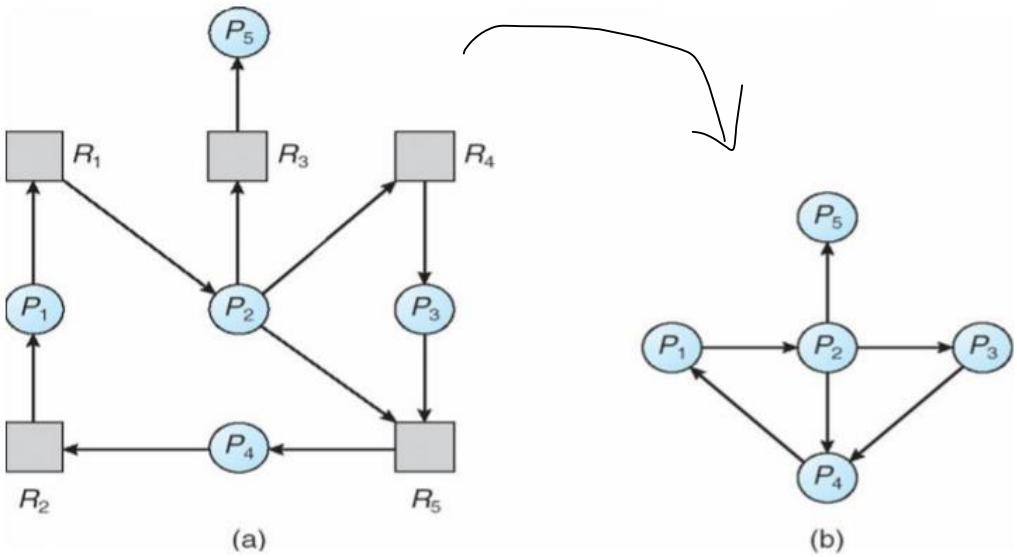
Single Instance of Each Resource Type

• Maintain wait-for graph – Nodes are processes – Pi → Pj if Pi is waiting for Pj

• Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

• An algorithm to detect a cycle in a graph requires an order of n 2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Example of Detection Algorithm:

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

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

1.In this, Work = [0, 0, 0] & Finish = [false, false, false, false, false]
2.i=0 is selected as both Finish[0] = false and [0, 0, 0]<=[0, 0, 0].
Work =[0, 0, 0]+[0, 1, 0] =>[0, 1, 0] & Finish = [true, false, false, false, false].
3.i=2 is selected as both Finish[2] = false and [0, 0, 0]<=[0, 1, 0].
Work =[0, 1, 0]+[3, 0, 3] =>[3, 1, 3] & Finish = [true, false, true, false, false].
4.i=1 is selected as both Finish[1] = false and [2, 0, 2]<=[3, 1, 3].
Work =[3, 1, 3]+[2, 0, 0] =>[5, 1, 3] & Finish = [true, true, true, false, false].
5.i=3 is selected as both Finish[3] = false and [1, 0, 0]<=[5, 1, 3].
Work =[5, 1, 3]+[2, 1, 1] =>[7, 2, 4] & Finish = [true, true, true, true, false].
6.i=4 is selected as both Finish[4] = false and [0, 0, 2]<=[7, 2, 4].
Work =[7, 2, 4]+[0, 0, 2] =>[7, 2, 6] & Finish = [true, true, true, true, true].
Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*
Since Finish is a vector of all true it means **there is no deadlock** in this example.

Example of Detection Algorithm(Cont.)

- P_2 requests an additional instance of type *C*

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

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

Detection-Algorithm Usage

• When, and how often, to invoke depends on:

– How often a deadlock is likely to occur?

– 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 Deadlock:

Process Termination

• Abort all deadlocked processes

• Abort one process at a time until the deadlock cycle is eliminated

• In which order should we choose to abort?

– Priority of the process

– How long process has computed, and how much longer to completion

– Resources the process has used

– Resource’s process needs to complete

– How many processes will need to be terminated

– Is process interactive or batch?

Recovery from Deadlock:

Resource Preemption

• Selecting a victim – minimize cost

• Rollback – return to some safe state, restart process for that state

• Starvation – same process may always be picked as victim, include number of rollbacks in cost factor