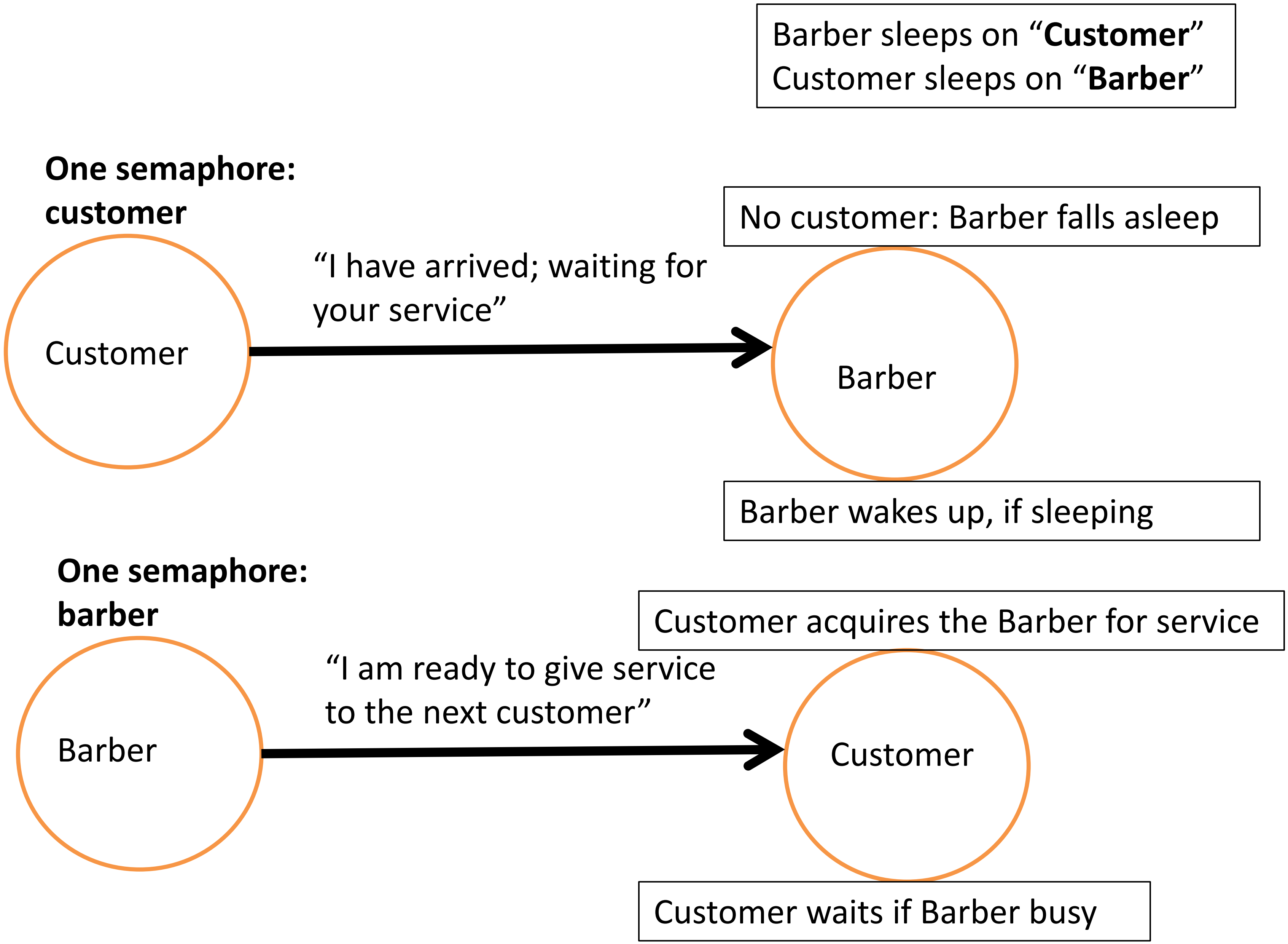
Tutorials

The Sleeping Barber Problem

Challenges

- Actions taken by barber and customer takes unknown amount of time (checking waiting room, entering shop, taking waiting room chair)
- Scenario 1
  - Customer arrives, observe that barber busy
  - Goes to waiting room
  - While he is on the way, barber finishes the haircut
  - Barber checks the waiting room
  - Since no one there, Barber sleeps
  - The customer reaches the waiting room and waits forever
- Scenario 2
  - Two customer arrives at the same time
  - Barber is busy
  - Both customers try to occupy the same chair!



Barber sleeps on "**Customer**"
Customer sleeps on "**Barber**"
One semaphore: customer
No customer: Barber falls asleep
"I have arrived; waiting for your service"
Customer
Barber
Barber wakes up, if sleeping
One semaphore: barber
Customer acquires the Barber for service
"I am ready to give service to the next customer"
Barber
Customer
Customer waits if Barber busy

The Sleeping Barber Problem

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* go to sleep if # of customers is 0 */
        down(&mutex);               /* acquire access to 'waiting' */
        waiting = waiting − 1;      /* decrement count of waiting customers */
        up(&barbers);               /* one barber is now ready to cut hair */
        up(&mutex);                 /* release 'waiting' */
        cut_hair( );                /* cut hair (outside critical region) */
    }
}


void customer(void)
{
    down(&mutex);                   /* enter critical region */
    if (waiting < CHAIRS) {         /* if there are no free chairs, leave */
        waiting = waiting + 1;      /* increment count of waiting customers */
        up(&customers);             /* wake up barber if necessary */
        up(&mutex);                 /* release access to 'waiting' */
        down(&barbers);             /* go to sleep if # of free barbers is 0 */
        get_haircut( );             /* be seated and be serviced */
    } else {
        up(&mutex);                 /* shop is full; do not wait */
    }
}
```

Semaphore Barber: Used to call a waiting customer.
Barber=1: Barber is ready to cut hair and a customer is ready (to get service) too!
Barber=0: customer occupies barber or waits

Semaphore customer: Customer informs barber that “I have arrived; waiting for your service”

Mutex: Ensures that only one of the participants can change state at once

The Sleeping Barber Problem

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* go to sleep if # of customers is 0 */
        down(&mutex);               /* acquire access to 'waiting' */
        waiting = waiting − 1;      /* decrement count of waiting customers */
        up(&barbers);               /* one barber is now ready to cut hair */
        up(&mutex);                 /* release 'waiting' */
        cut_hair();                 /* cut hair (outside critical region) */
    }
}

void customer(void)
{
    down(&mutex);                   /* enter critical region */
    if (waiting < CHAIRS) {         /* if there are no free chairs, leave */
        waiting = waiting + 1;      /* increment count of waiting customers */
        up(&customers);             /* wake up barber if necessary */
        up(&mutex);                 /* release access to 'waiting' */
        down(&barbers);             /* go to sleep if # of free barbers is 0 */
        get_haircut();              /* be seated and be serviced */
    } else {
        up(&mutex);                 /* shop is full; do not wait */
    }
}
```

Barber sleeps on "**Customer**"
Customer sleeps on "**Barber**"

For Barber: Checking the waiting room and calling the customer makes the **critical section**

For customer: Checking the waiting room and informing the barber makes its **critical section**

Problem 1

We want to use semaphores to implement a shared critical section (CS) among three processes T1, T2, and T3. We want to enforce the execution in the CS in this order: First T2 must execute in the CS. When it finishes, T1 will then be allowed to enter the CS; and when it finishes T3 will then be allowed to enter the CS; when T3 finishes then T2 will be allowed to enter the CS, and so on, (T2, T1, T3, T2, T1, T3,…).

Write the synchronization solution using a minimum number of binary semaphores and you are allowed to assume the initial value for semaphore variables.

Problem 1

T1	T2	T3
While(true) { Wait(S3); Print(“C”); Signal (S2); }	While(true) { Wait(S1); Print(“B”); Signal (S3); }	While(true) { Wait(S2); Print(“A”); Signal (S1); }

S1=1, S2=0, S3=0

Problem 2

Three concurrent processes X, Y, and Z execute three different code segments that access and update certain shared variables.

Process X executes the P operation (i.e., wait) on semaphores a, b and c;

process Y executes the P operation on semaphores b, c and d;

process Z executes the P operation on semaphores c, d, and a before entering the respective code segments.

After completing the execution of its code segment, each process invokes the V operation (i.e., signal) on its three semaphores.

All semaphores are binary semaphores initialized to **one**.

Which one of the following represents a deadlock-free order of invoking the P operations by the processes?

(A) X: P(a)P(b)P(c) Y: P(b)P(c)P(d) Z: P(c)P(d)P(a)
(B) X: P(b)P(a)P(c) Y: P(b)P(c)P(d) Z: P(a)P(c)P(d)
(C) X: P(b)P(a)P(c) Y: P(c)P(b)P(d) Z: P(a)P(c)P(d)
(D) X: P(a)P(b)P(c) Y: P(c)P(b)P(d) Z: P(c)P(d)P(a)

Problem 3

- The following two functions P1 and P2 that share a variable B with an initial value of 2 execute concurrently.

```
P1()
{
  C = B – 1;
  B = 2*C;
}

P2()
{
  D = 2 * B;
  B = D - 1;
}
```

The number of distinct values that B can possibly take after the execution

Problem 4

Consider the reader-writer problem with designated readers. There are n reader processes, where n is known beforehand. There are one or more writer processes. Items are stored in a buffer. Every item is written by a writer and is designated for a particular reader.

```
semaphore rw_mutex = 1;
semaphore r_mutex[n] = {0, 0, . . . , 0};

reader (i)
{
        wait(r_mutex[i]);
        while (true) {
                wait(rw_mutex);
                Read and remove one item from buffer, that is meant for the i-th reader;
                signal(rw_mutex);
                wait(r_mutex[i]);
        }
}

writer ()
{
        while (true) {
                Generate item for reader i;
                wait(rw_mutex);
                Write (item, i) to buffer;
                signal(rw_mutex);
                signal(r_mutex[i]);
        }
}
```

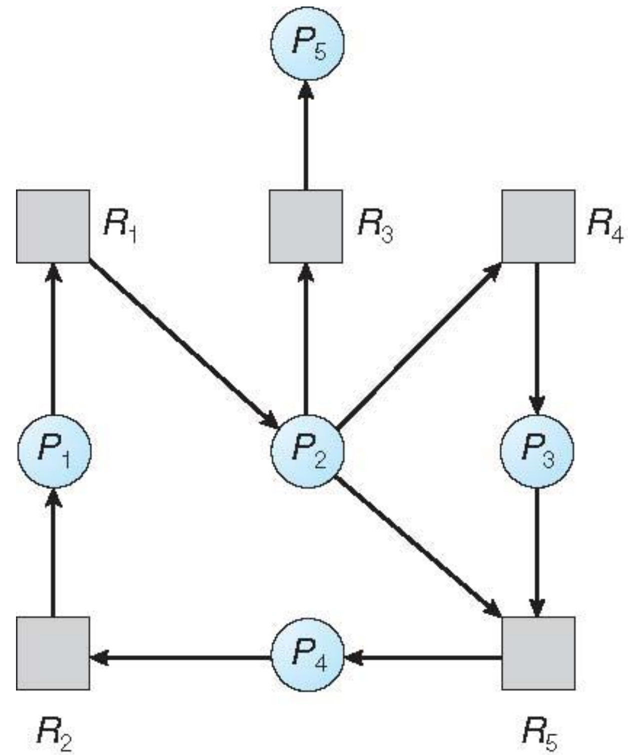
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

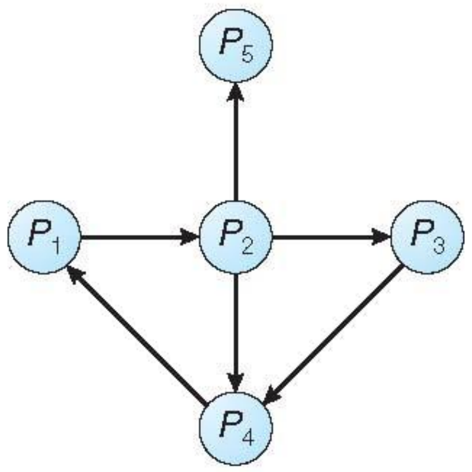
Single Instance of Each Resource Type

- *In resource graph*
  - $P_i \rightarrow R$ *and* $R \rightarrow P_j$
- Maintain *wait-for* graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph.
- If there is a cycle, there exists a deadlock

Resource-Allocation Graph and Wait-for Graph



(a)



(b)

Resource-Allocation Graph

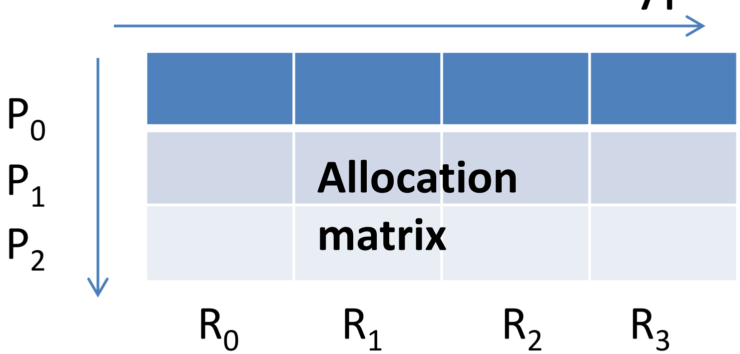
Corresponding wait-for graph

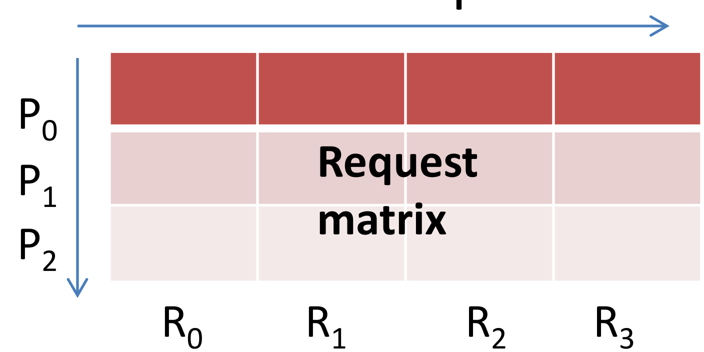
Several Instances of a Resource Type

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

- **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.

Several Instances of a Resource Type

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

Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

Resources available
$(A_1, A_2, A_3, \ldots, A_m)$

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

Detection Algorithm

- Define a relation $\leq$ over two vectors

- X and Y are two vectors of length n

- We say $X \leq Y$

  Iff $X[i] \leq Y[i]$ for all $i=1, 2, \ldots, n$

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

Detection Algorithm (Cont.)

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 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	

- Sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in *Finish*[*i*] = true for all *i*

Example (Cont.)

- P_2 requests an additional instance of type C

	Request
	$A\ B\ C$
P_0	0 0 0
P_1	2 0 2
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

Home work

$$\begin{array}{ccccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ E = (& 4 & 2 & 3 & 1\ ) \end{array}$$

$$\begin{array}{ccccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ A = (& 2 & 1 & 0 & 0\ ) \end{array}$$

Available

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected by deadlock?
- If deadlock frequent
  - Invoke detection algo frequently
- Invoke after each (waiting) resource request
  - Huge overhead
- CPU utilization drops

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
  - Expensive
- Abort one process at a time until the deadlock cycle is eliminated
  - Overhead=> invoke detection algo
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed
  - 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 Preemption

- Selecting a victim – minimize cost
  - (# of resources holding, duration)

- Rollback – return to some safe state, restart process from that state

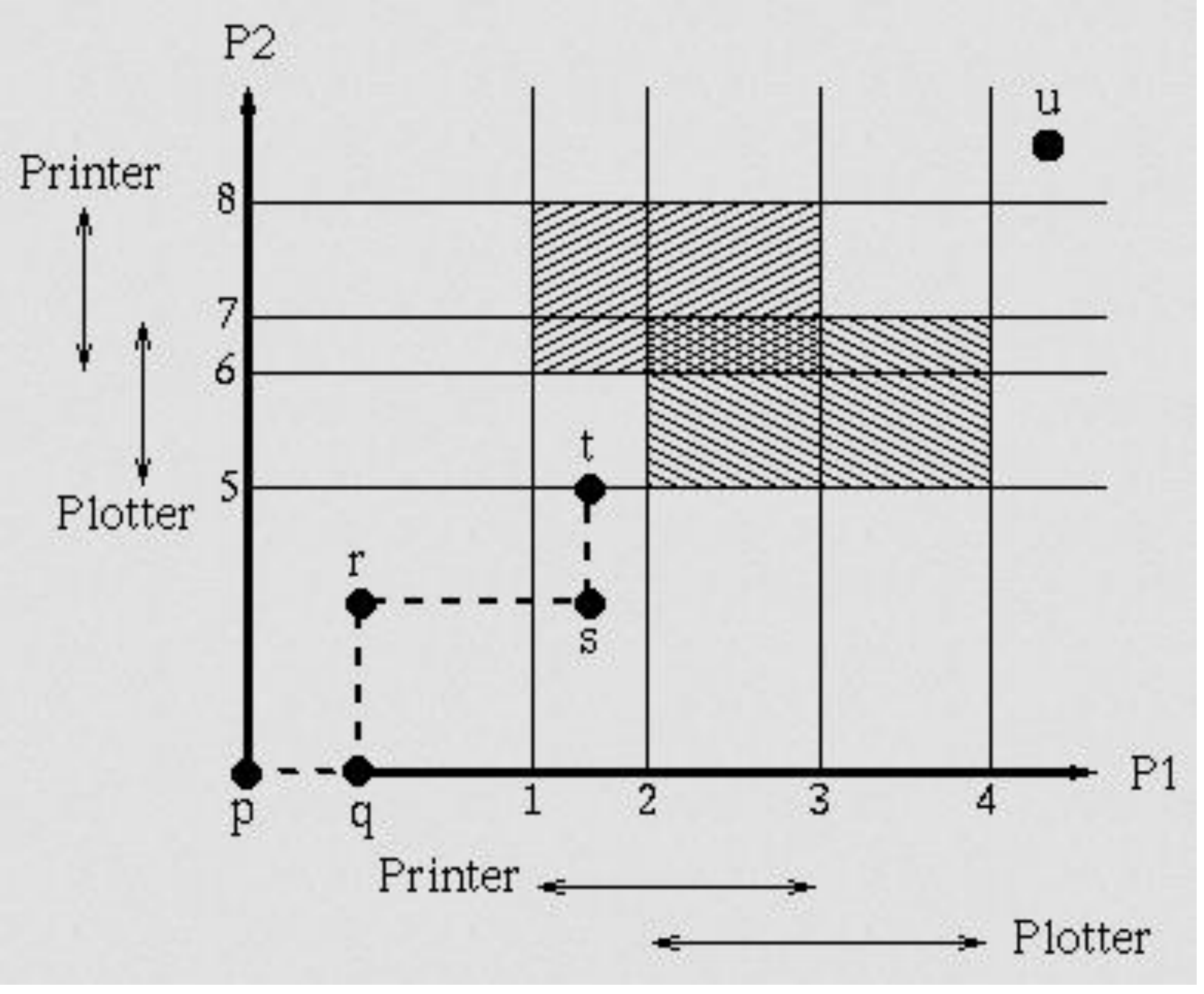
- Starvation – same process may always be picked as victim, include number of rollback in cost factor

Deadlock Avoidance

Requires that the system has some additional *a priori* information available

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the **resource-allocation state** to ensure that there can never be a circular-wait condition
- Resource-allocation ***state*** is defined by the number of **available and allocated** resources, and the **maximum demands** of the processes

Safe State



Safe State

- When a process **requests** an available resource, system must **decide** if immediate allocation leaves the system in a **safe state**

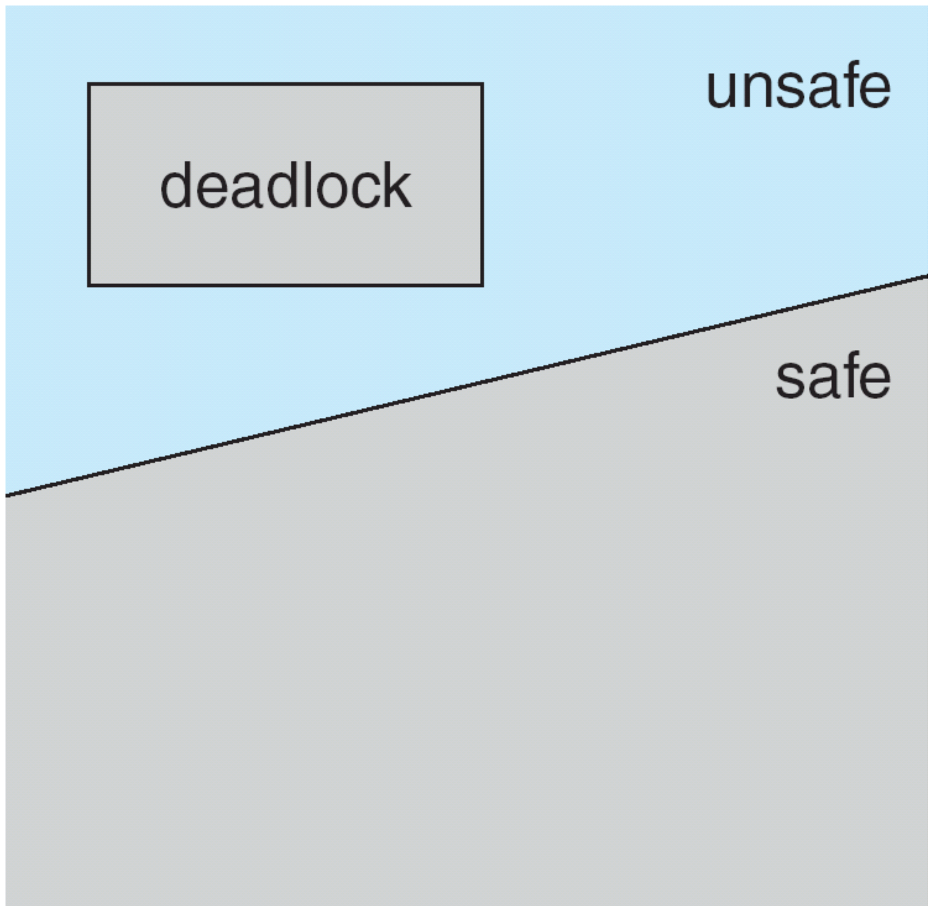
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems
  - such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$

- That is:
  - 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 P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State



Three processes P0, P1, P2

Resource R=12

State at time t_0

	Maximum need	Current allocation
P0	10	5
P1	4	2
P2	9	2

Free resource = 3

Safe sequence <P1, P0, P2> **Safe state**

State at time t_1

Allocate one resource to P2

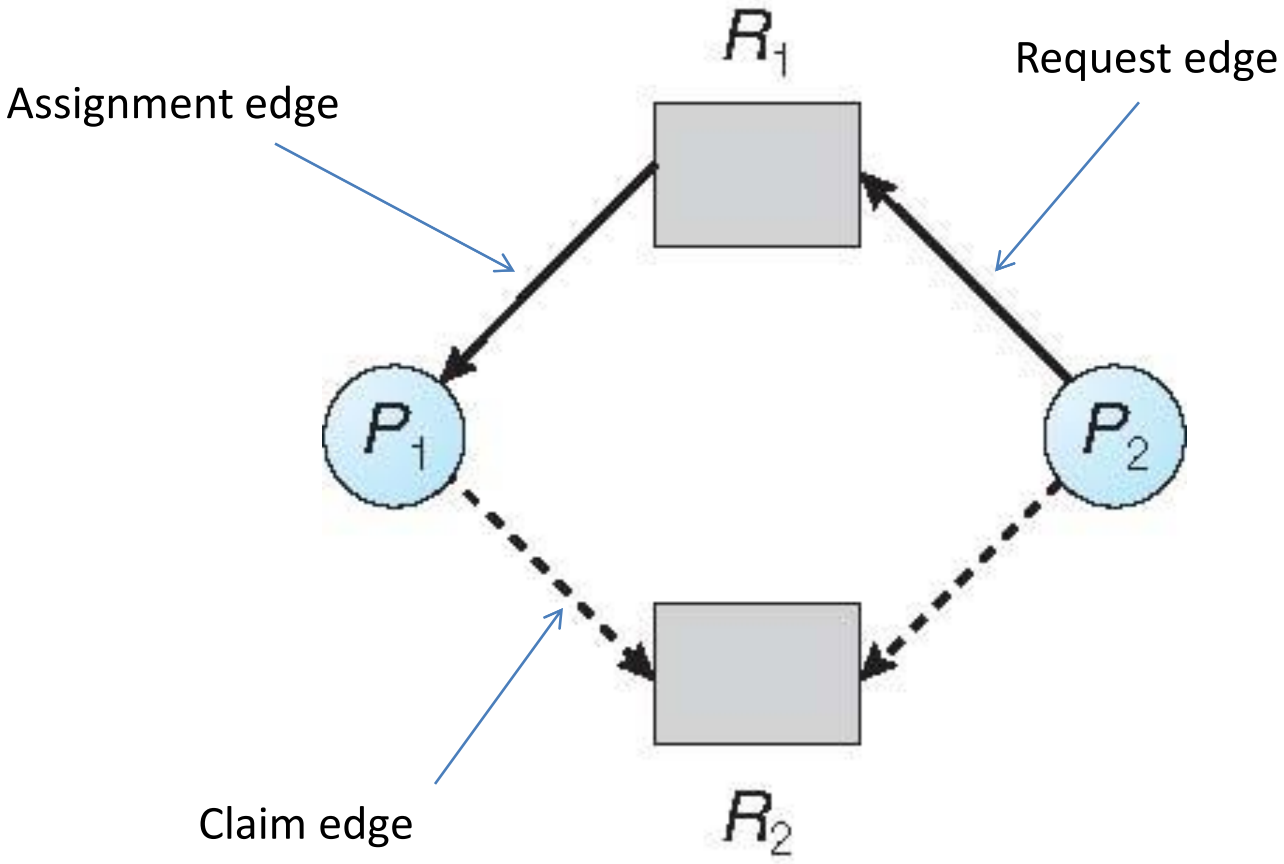
Avoidance algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; 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

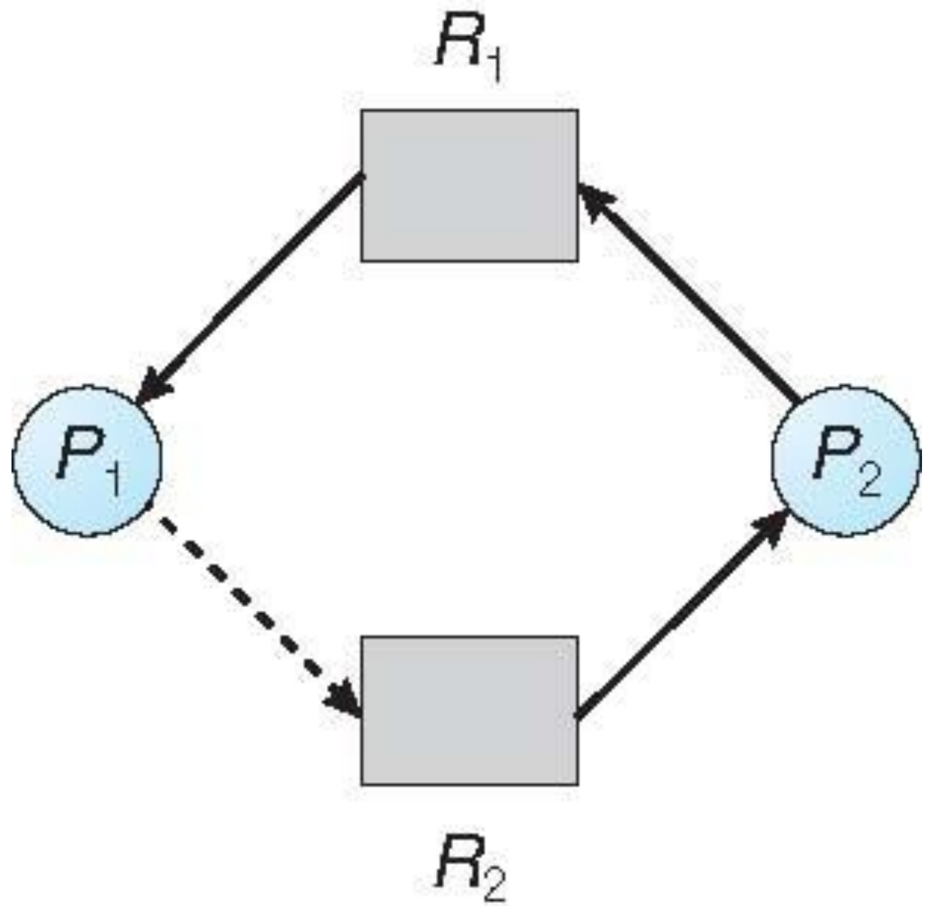
Resource-Allocation Graph



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- 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
- If no cycle
  - Safe state

Unsafe State In Resource-Allocation Graph



Suppose that process P_2 requests a resource R_2

Banker’s Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a set of resources
  - System decides whether the allocation is safe
- When a process requests a resource – not safe?
  - it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

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

- **Available***:* Vector of length m. If available $[j] = k$, there are k instances of resource type 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 P_i 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

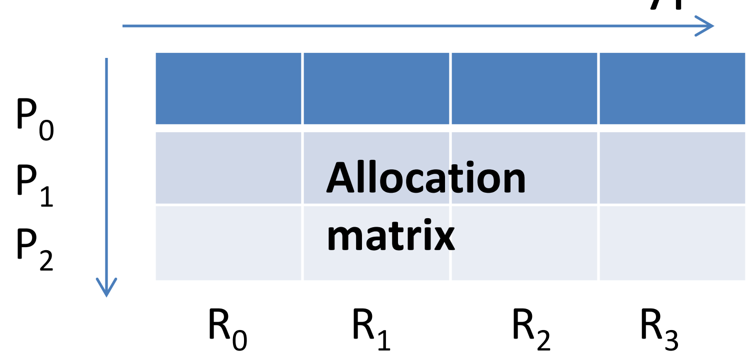
***Need* [*i,j*] = *Max*[*i,j*] – *Allocation* [*i,j*]**

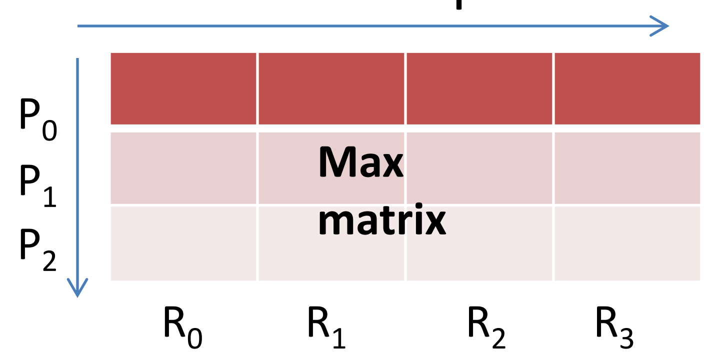
Several Instances of a Resource Type

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

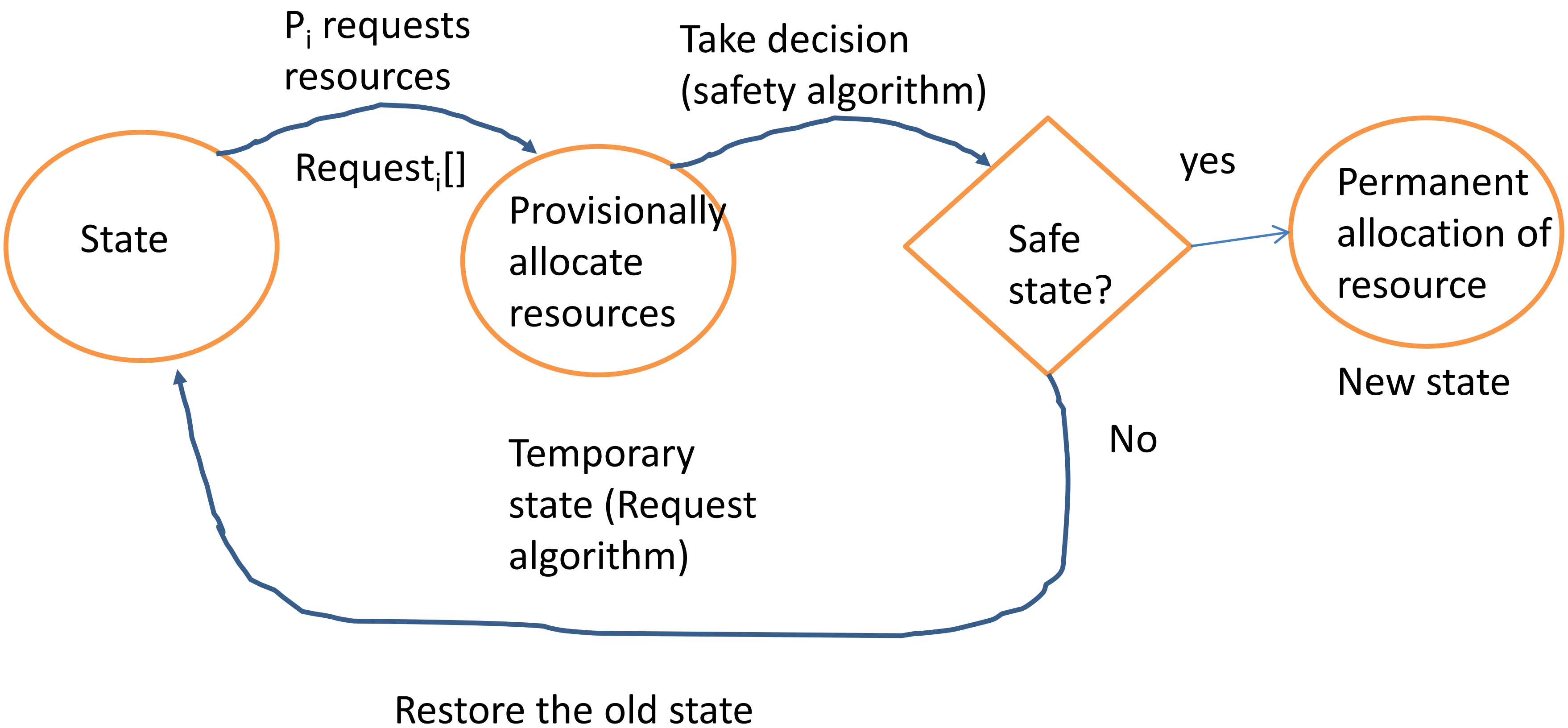
- **Available**: A vector of length m indicates the number of available resources of each type.

- **Allocation**: An n x m matrix defines the number of resources of each type currently allocated to each process.





Deadlock avoidance : Flow chart for P_i



Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

   Work = Available

   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1

2. Find an *i* such that both:

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

   (b) $Need_i \leq Work$

   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

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i.

If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \le Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \le Available$, 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:

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

- *If safe $\Rightarrow$ the resources are allocated to Pi*
- *If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation state is restored*

Example of Banker's Algorithm

- 5 processes P_0 through P_4;
  3 resource types:
  A (10 instances), *B* (5instances), and *C* (7 instances)

Snapshot at time T_0:

	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	

Example (Cont.)

- The content of the matrix *Need* is defined to be *Max* – *Allocation*

	Need
	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

	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	

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example (Cont.)

- The content of the matrix *Need* is defined to be *Max* – *Allocation*

	Need
	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 in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2)$ $\Rightarrow$ true

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
$\boldsymbol{P_1}$	**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	

- Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Deadlock Handling

Strategies for dealing with deadlocks:

1. Detection and recovery. Let deadlocks occur, detect them, take action.
2. Dynamic avoidance by careful resource allocation.
3. Prevention, by structurally negating one of the four required conditions.
4. Just ignore the problem.

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources
  - Read only file
- **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,
  - Allow process to request resources only when the process has none
    - Release all the current resource and then try to acquire
  - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources,
  - Requests another resource that cannot be immediately allocated to it
    - Resources were allocated to some waiting process
  - Preempt the desired resource from waiting process
  - Allocate to current process
  - Cpu Registers
- **Circular Wait** – Impose a total ordering of all resource types
  - Require that each process requests resources in an increasing order of enumeration

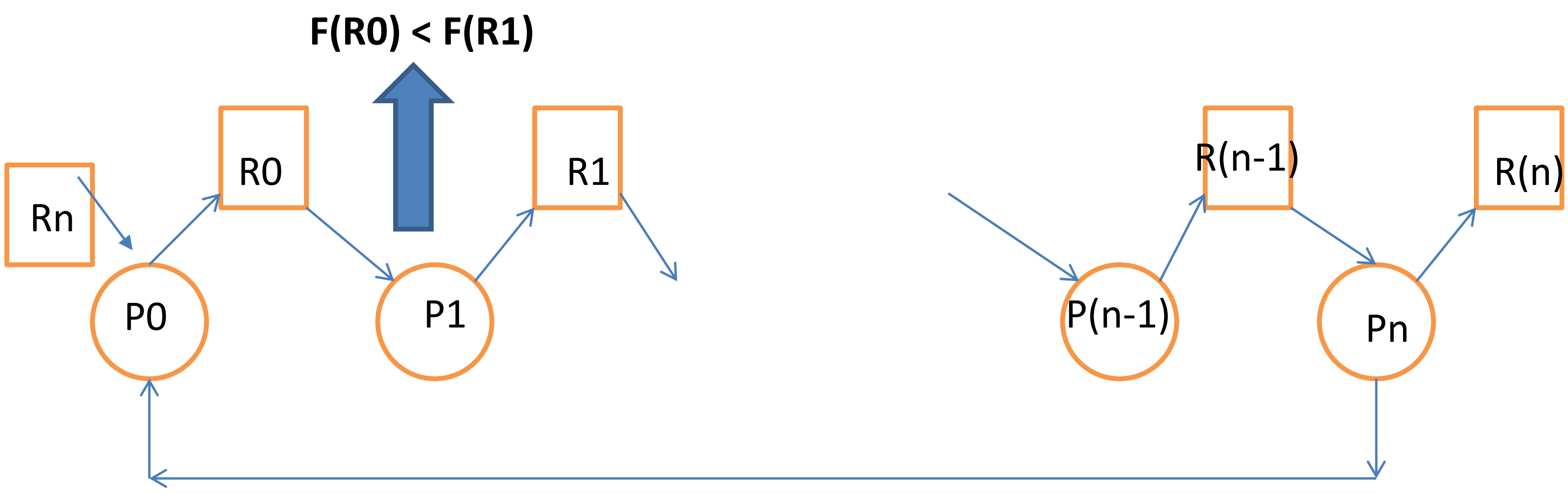
- Let R={R1, R2,……,Rm} set of resource type
- We assign unique integer with each type
- One to one function F:R→N

F(tape drive)=1

F(disk)=5

F(printer)=12

- Protocol: Each process can request resource only in an increasing order.
- Initially request R_i, after that, it can request R_j
  – If and only if $F(R_j)>F(R_i)$
- Currently holding $R_{j;}$ Want to request R_i.
  - Must have released R_j



$$F(R0) < F(R1) < F(R2) < \ldots\ldots\ldots\ldots\ldots\ldots < F(Rn) < F(R0)$$

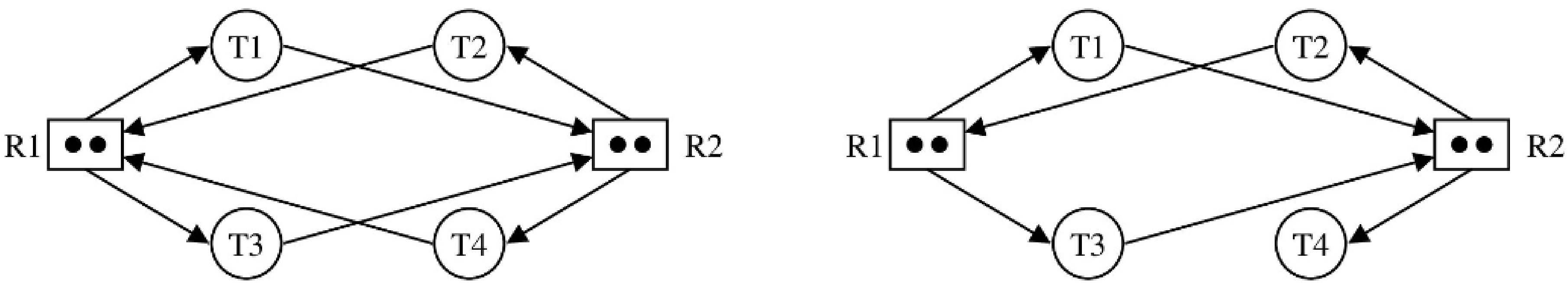
Problem 1

A system is having 3 user processes each requiring max 2 units of resource R.

What is the minimum number of units of R such that no deadlock will occur?

Problem 2

Deadlock or not? Justify.



(a) (b)

Problem 3

Q1: A single processor system has three resource types X, Y, and Z, which are shared by three processes. There are 5 units of each resource type.

Allocation

	X	Y	Z
P0	1	2	1
P1	2	0	1
P2	2	2	1

Request

	X	Y	Z
P0	1	0	3
P1	0	1	2
P2	1	2	0

(i) Is the system in a safe state? What is the safe sequence?

(ii) What will happen if process P_1 requests two additional instances of resource type C?

.

Answer:

(i) According to the question-
Total = [X Y Z] = [5 5 5] , Total _Allocation = [X Y Z] = [5 4 3]
Now, Available = Total – Total_Allocation = [5 5 5] – [5 4 3] = [0 1 2]

• Step: With the instances available currently, only the requirement of process P1 can be satisfied. So, process P1 is allocated the requested resources. It completes its execution and then frees up the instances of resources held by it.
(Then, Available = [0 1 2] + [2 0 1] = [2 1 3]

By repeating the above step, we will get the following

--→ P0, Available = [3 3 4]
-→ P2, Available = [5 5 5]
-→ There exists a safe sequence P1, P0, P2 in which all the processes can be executed.

(ii)New_Request = P1 [0 0 2], so Now, available becomes [0 1 0],