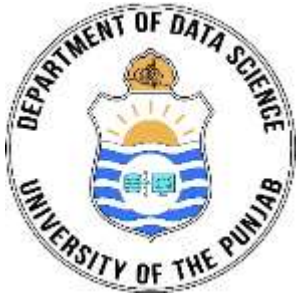


CMP325
Operating Systems
Lecture 17, 18

Dead Locks

Muhammad Arif Butt, PhD

Note:
Some slides and/or pictures are adapted from course text book and Lecture slides of

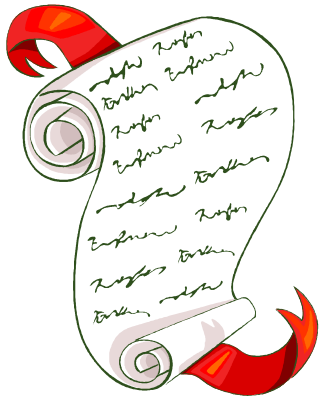
- Dr Syed Mansoor Sarwar
- Dr Kubiatowicz
- Dr P. Bhat
- Dr Hank Levy
- Dr Indranil Gupta

For practical implementation of operating system concepts discussed in these slides, students are advised to watch and practice following video lectures:

OS with Linux:
https://www.youtube.com/playlist?list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8

System Programming:
https://www.youtube.com/playlist?list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW

Today's Agenda



- Introduction to Dead Locks
- Examples of Deadlocks
- Conditions for Deadlocks
- Resource Allocation Graphs
- Deadlock Solutions
  - Prevention
  - Avoidance
  - Detection and Recovery

Deadlock Problem

- A set of processes is said to be in a dead lock state if every process is waiting for an event that can be caused only by another process in the set.
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- A process is said to be dead locked if it is waiting for an event which will never occur.

Deadlock Problem (cont…)

Examples

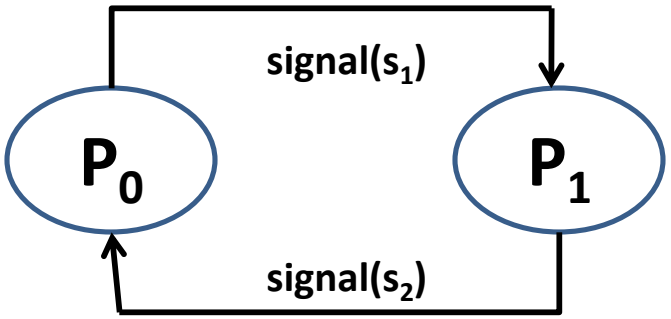
- A table with a writing pad and a pen. Two persons sitting around the table wants to write letter. One picks up the pad and the other grab the pen, causing dead lock
- Two cars crossing a single lane bridge from opposite direction
- A person going down a ladder while another person is climbing up the ladder
- Two trains travelling toward each other on the same track
- Consider a system having two tape drives. P1 and P2 each hold one tape drive and each needs the other one (e.g., to copy data from one tape drive to another)

Deadlock Problem (cont…)

- Consider two semaphores s1 and s2 each initialized to 1.

P_0
```
wait(s1);
wait(s2);
⋮
signal(s1);
signal(s2);
⋮
```

P_1
```
wait(s2);
wait(s1);
⋮
signal(s2);
signal(s1);
⋮
```

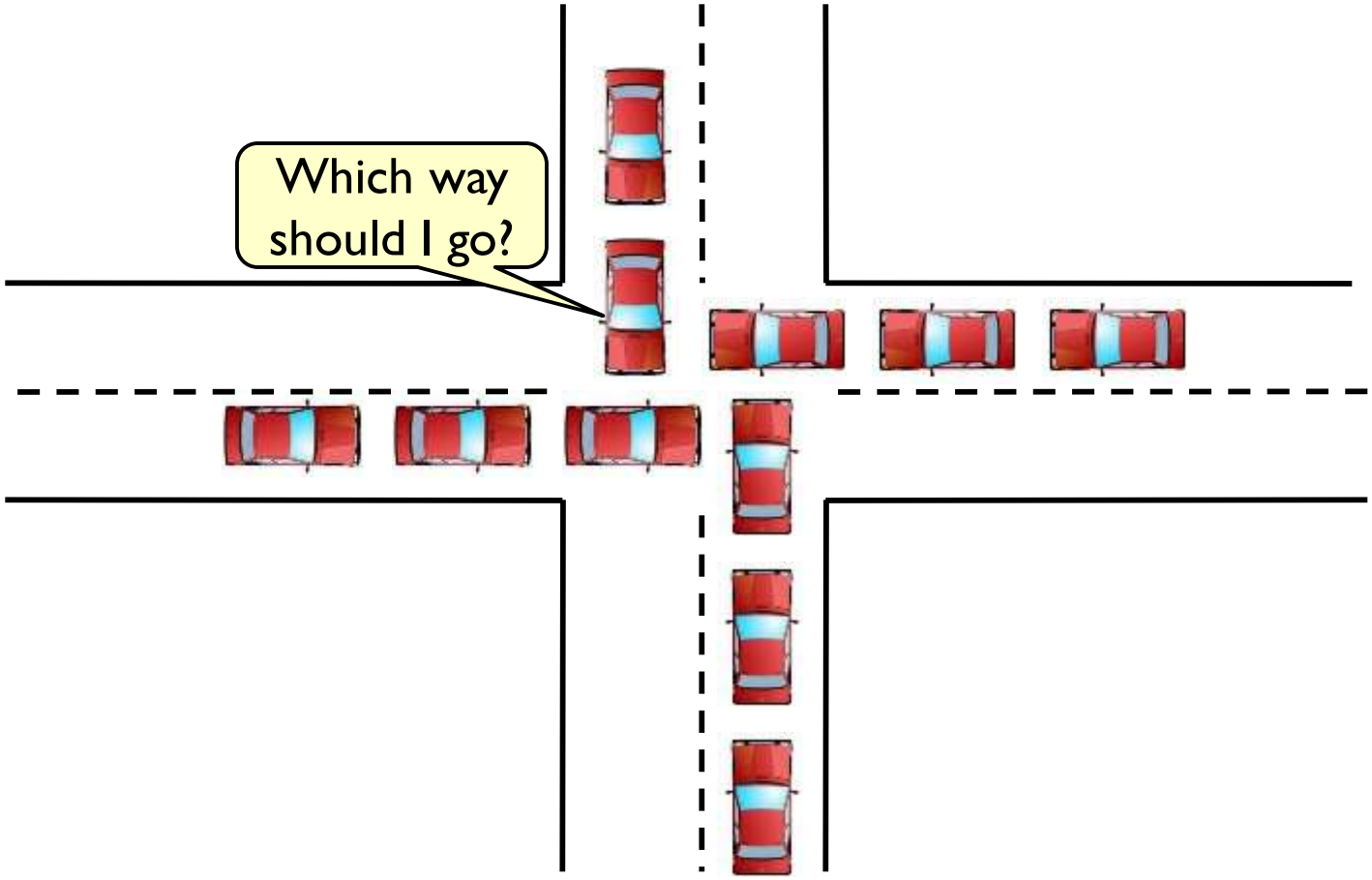


Deadlock Problem (cont…)

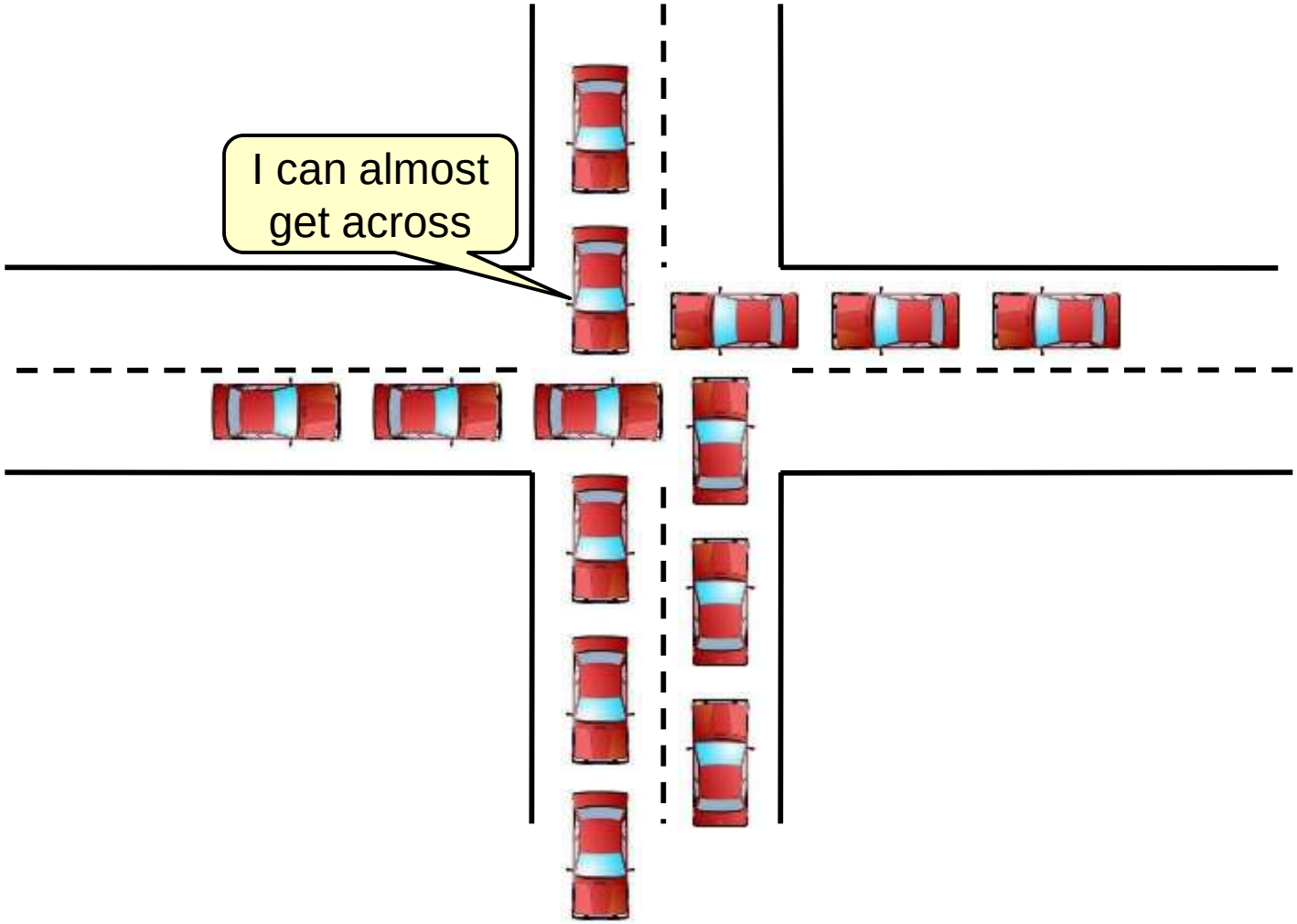
Dining Philosopher Problem

- Consider the dining philosophers problem.
- All five philosophers become hungry at the same time.
- All pick up the chopsticks on their left and look for the right, which was held by the neighboring philosopher.
- No one put the chopstick back and wait for the right chopstick and finally all starved to death

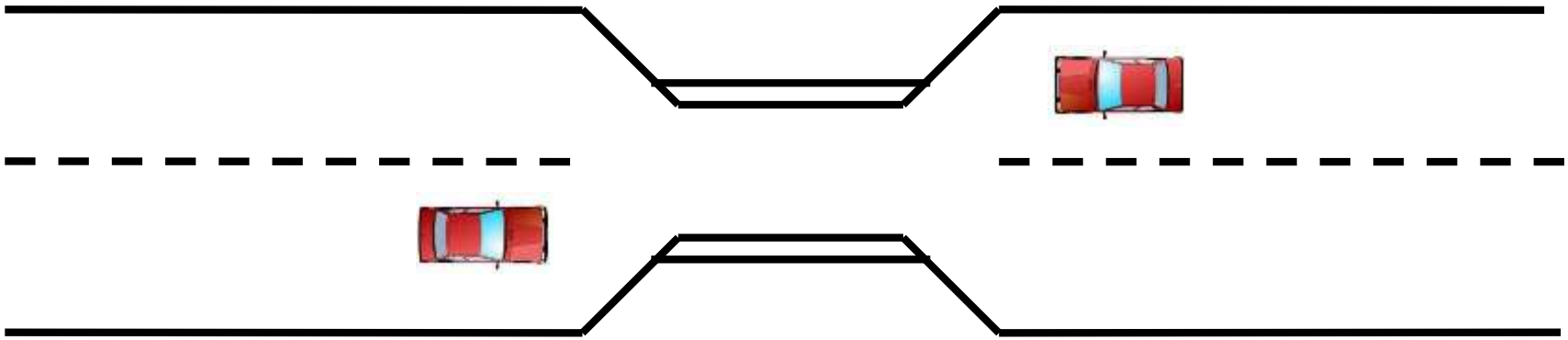
Deadlock in the real world



Deadlock in the real world



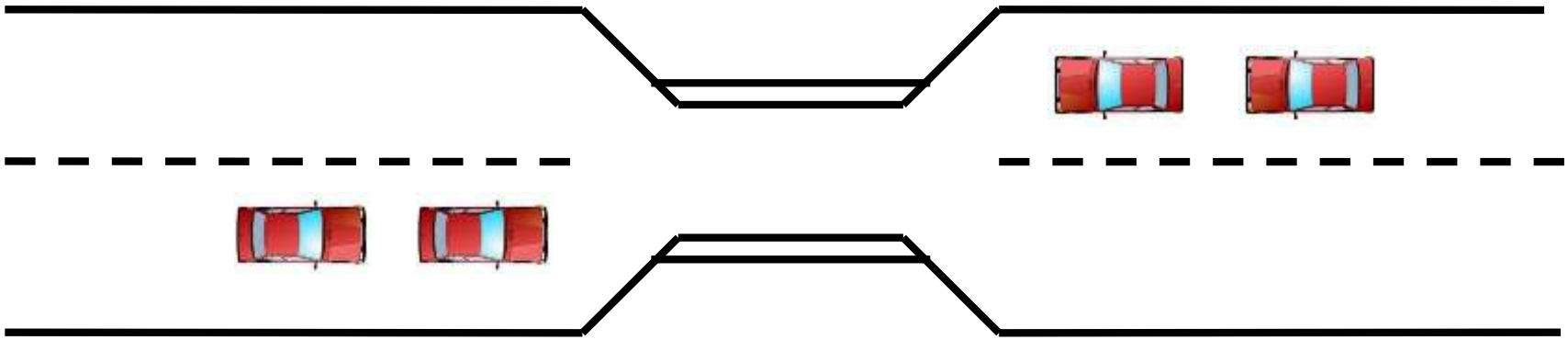
Deadlock: One-lane Bridge



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource

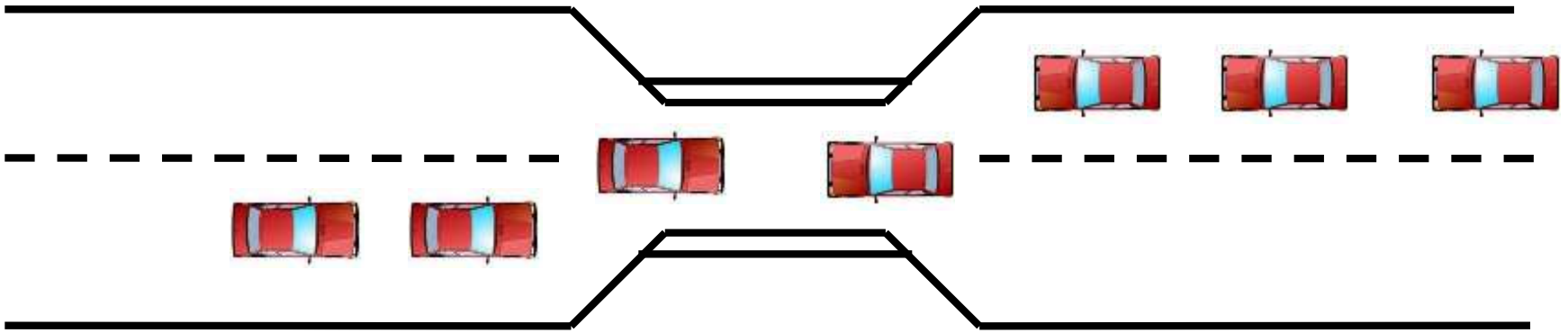
What can happen?

Deadlock: One-lane Bridge



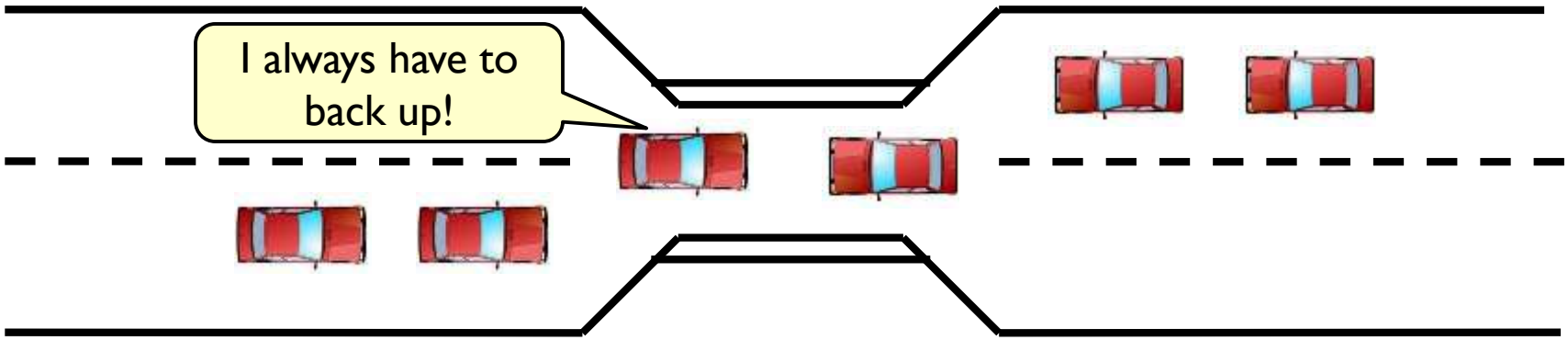
- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- Deadlock
  - Resolved if cars back up (preempt resources and rollback)
  - Several cars may have to be backed up

Deadlock: One-lane Bridge



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- Deadlock
  - Resolved if cars back up (preempt resources and rollback)
  - Several cars may have to be backed up
- But, starvation is possible
  - e.g., if the rule is that Westbound cars always go first when present

Deadlock: One-lane Bridge



- Deadlock vs. Starvation
  - Starvation = Indefinitely postponed
    - Delayed repeatedly over a long period of time while the attention of the system is given to other processes
    - Logically, the process may proceed but the system never gives it the CPU (unfortunate scheduling)
  - Deadlock = no hope
    - All processes blocked; scheduling change won't help

Resources

- Deadlocks occurs when processes have been granted exclusive access to resources; In case of sharable access DL will not occur.
- A resource can be a hardware device (e.g. tape drive, printer, memory, CPU) or a piece of information (a variable, file, semaphore, record of a database).
- Resources comes in two flavors:
  - **Preemptable Resources.** A resource that can be taken away from the process holding it with no ill effects, (sharable resources) e.g. CPU, memory.
  - **Non-Preemptable Resources.** A resource that cannot be taken away from the process holding it, w/o causing the computation to fail, (non sharable resources) e.g. if a process has begun to burn a CD ROM, suddenly taking the CD recorder away from it and giving the CD Recorder to another process will result in a garbled CD.
- In general DL involve non-preemptable resources.
- A process may utilize a resource in following sequence:

Request → Use → Release

Conditions for Deadlock

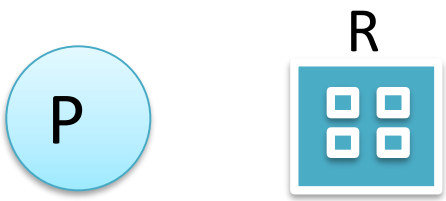
Deadlock can arise if four conditions hold simultaneously.

1. **Mutual exclusion**: Only one process at a time can use a resource.
2. **Hold and wait**: A process holding one or more allocated resources while awaiting assignment of other.
3. **No preemption**: No resource can be forcibly removed from a process holding it.
4. **Circular wait**: There exists a set {P0, P1, …, Pn} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.

P0 → P1 → P2 → … → Pn → P0

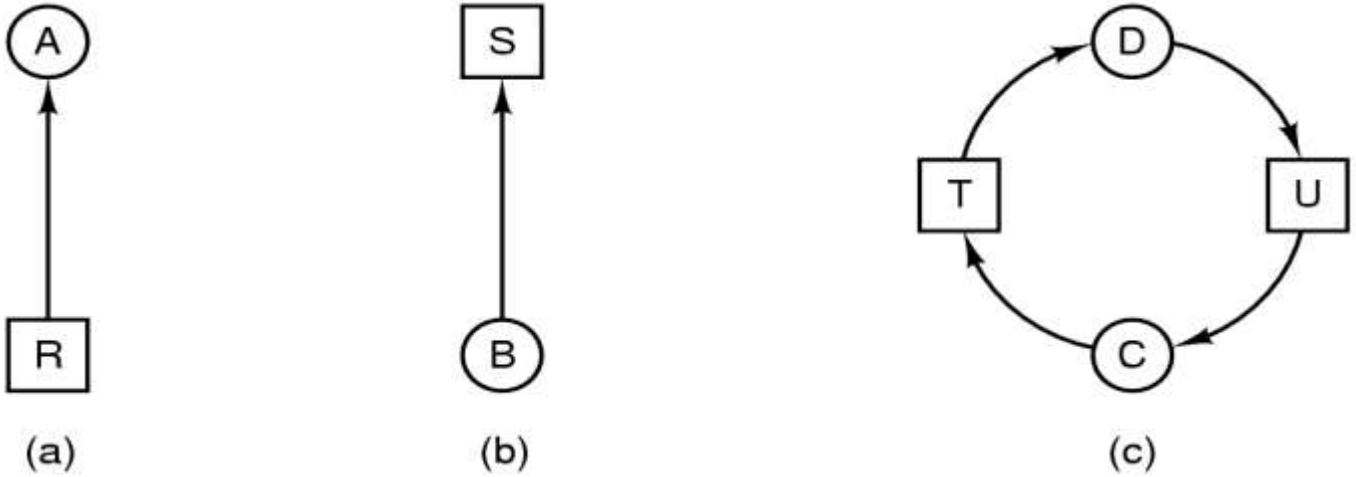
Resource Allocation Graph

- Consider some processes {P0, P1, P2, ….Pn} and some resources {R0, R1, R2, … Rn} within a system.
- Each resource type may have one or more instances.
- Each process utilizes a resource as (request, use, release).
- **RAG**
  - A set of vertices **V** and a set of edges **E**.
  - **V** is partitioned into two types:
    - ✓ **P** = {P1, P2, …, Pn}
    - ✓ **R** = {R1, R2, …, Rm}
  - **E** can also be of two types:
    - ✓ **Request Edge**: P1 → Rj
    - ✓ **Assignment Edge**: Rj → Pi



Resource Allocation Graph

- Modeled with directed graphs



- Resource R assigned to process A
- Process B is requesting/waiting for resource S
- Process C and D are in deadlock over resources T and U

Resource Allocation Graph

A

Request R
Request S
Release R
Release S

(a)

B

Request S
Request T
Release S
Release T

(b)

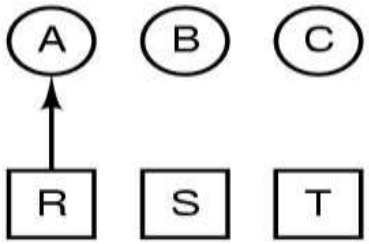
C

Request T
Request R
Release T
Release R

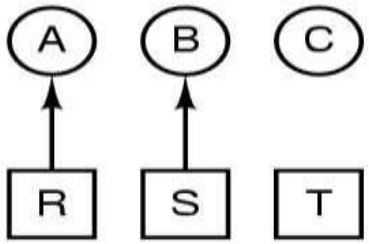
(c)

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

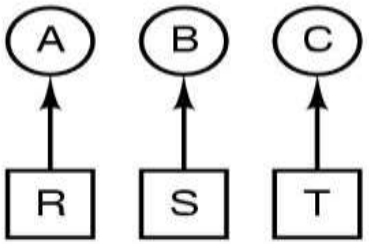
(d)



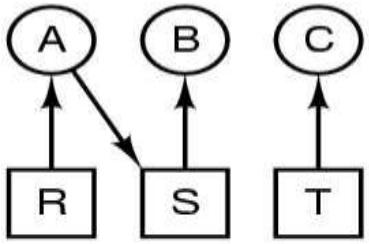
(e)



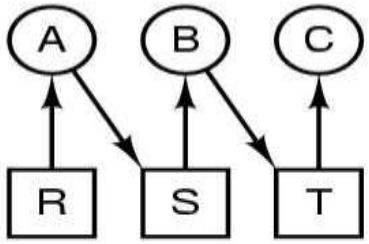
(f)



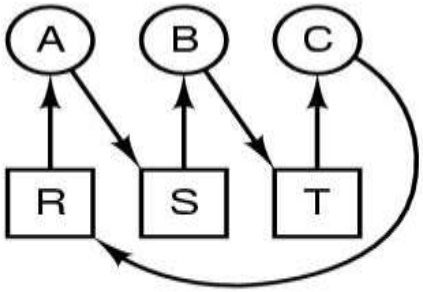
(g)



(h)



(i)

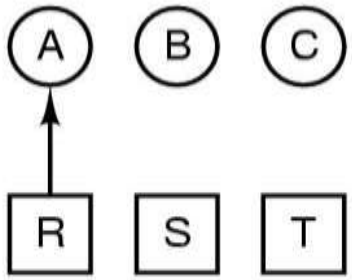


(j)

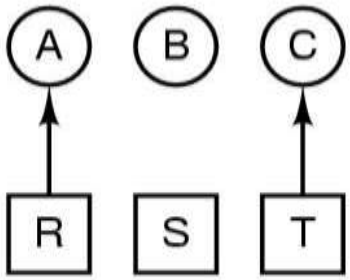
Resource Allocation Graph

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S

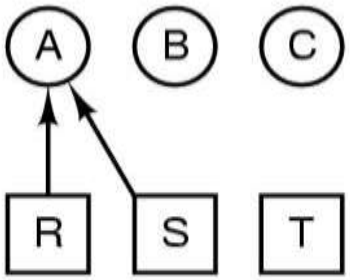
no deadlock



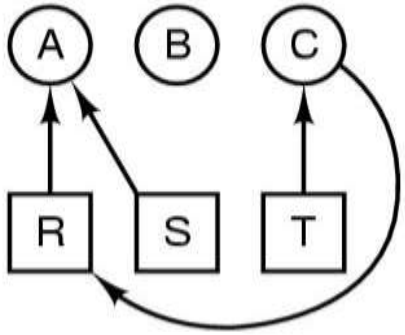
(k)



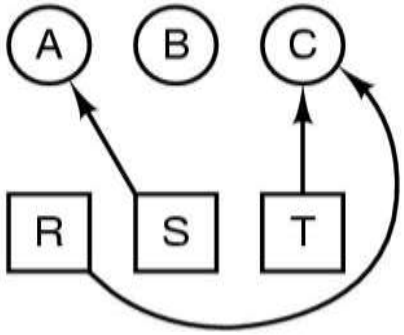
(l)



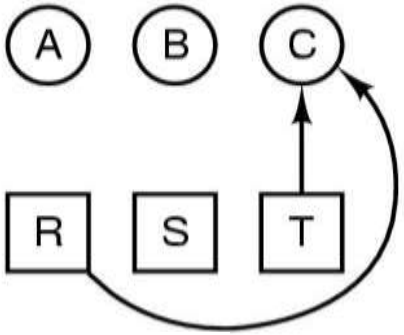
(m)



(n)



(o)



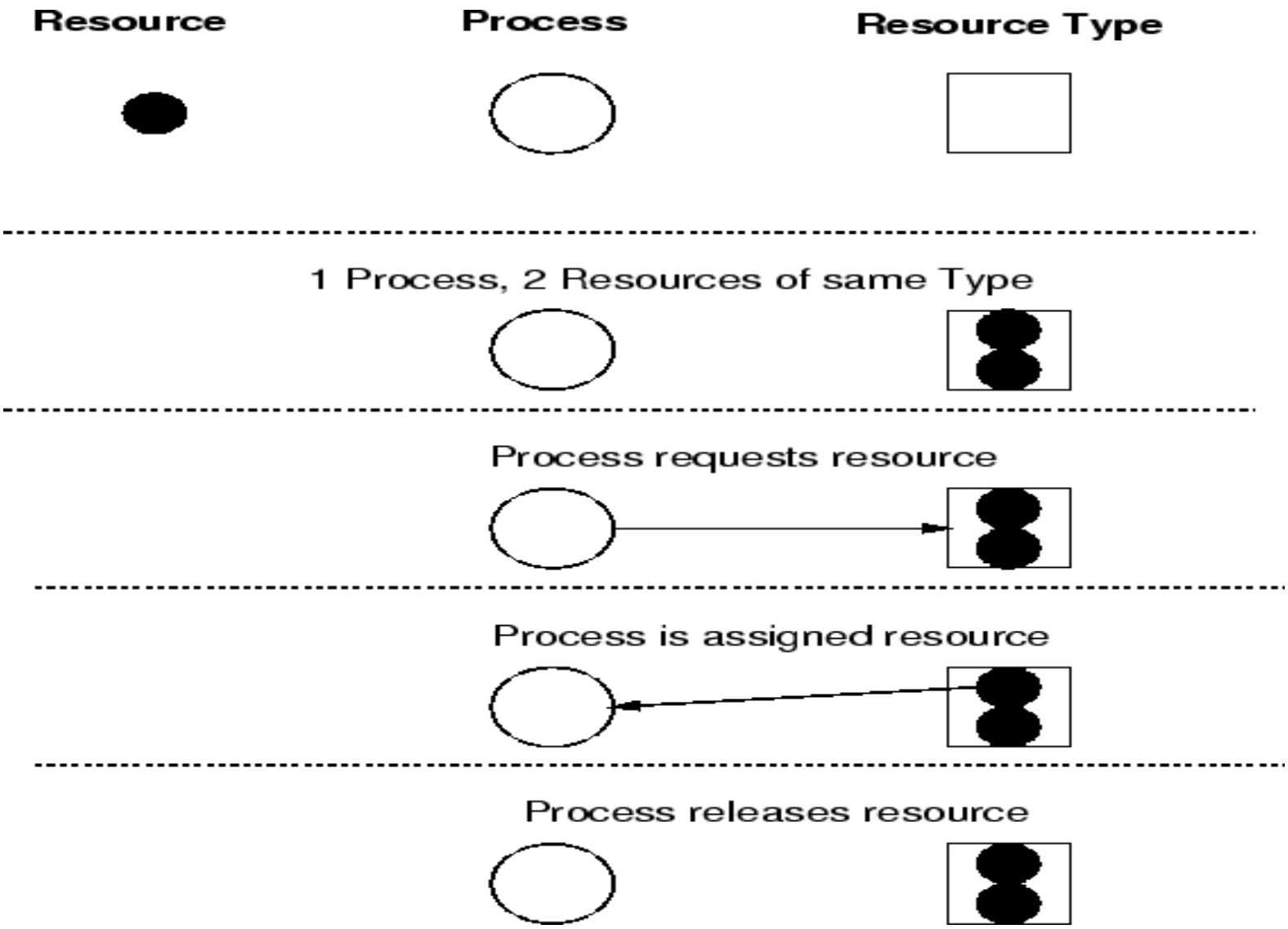
(p)

(q)

a. **Suppose that in step 4 / (o) C requested S instead of requesting R. Would this lead to deadlock?**
b. **Suppose C requested both S and R. Would this lead to deadlock?**

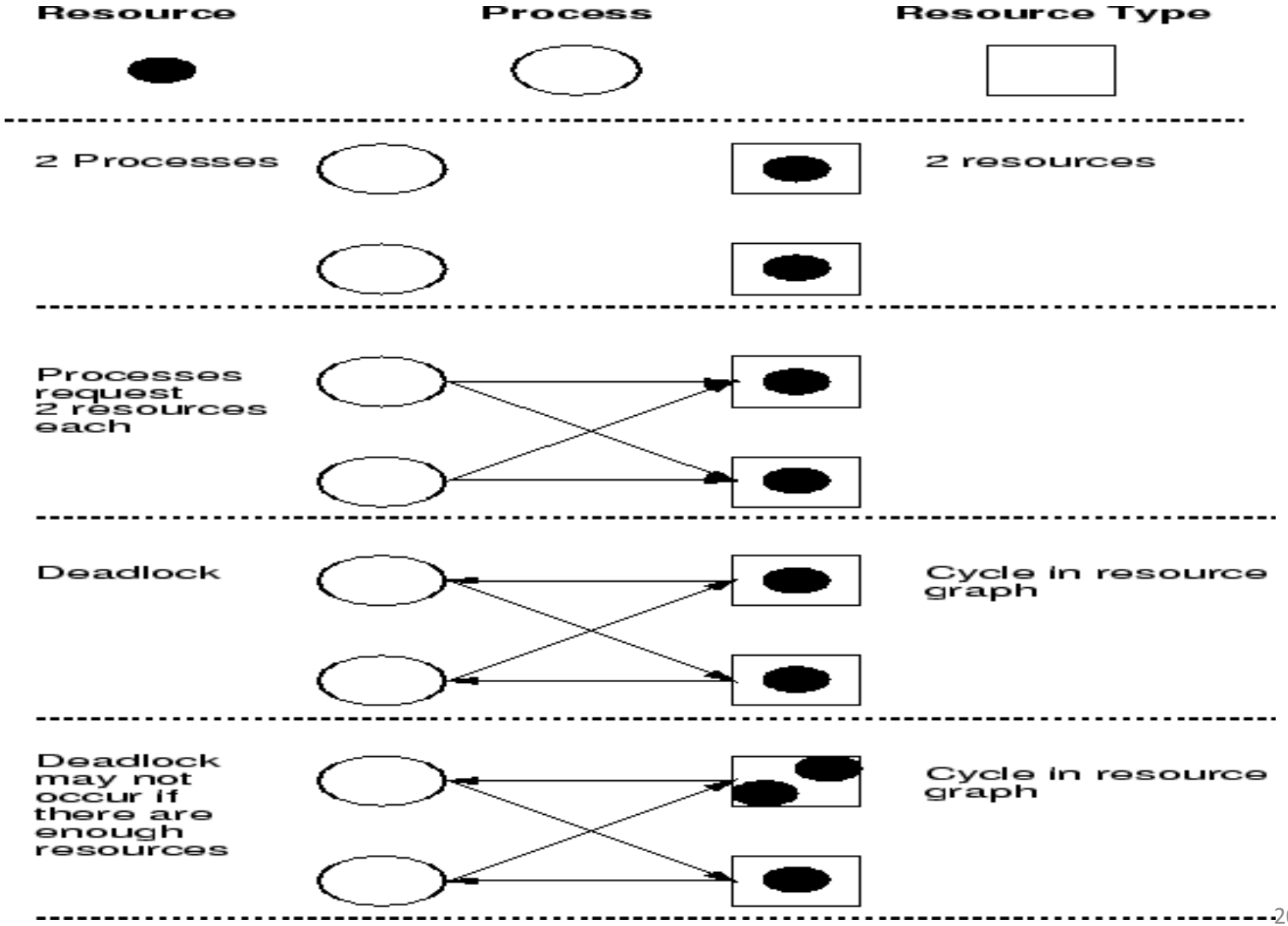
Resource Allocation Graph

RAG with multiple resources of same type

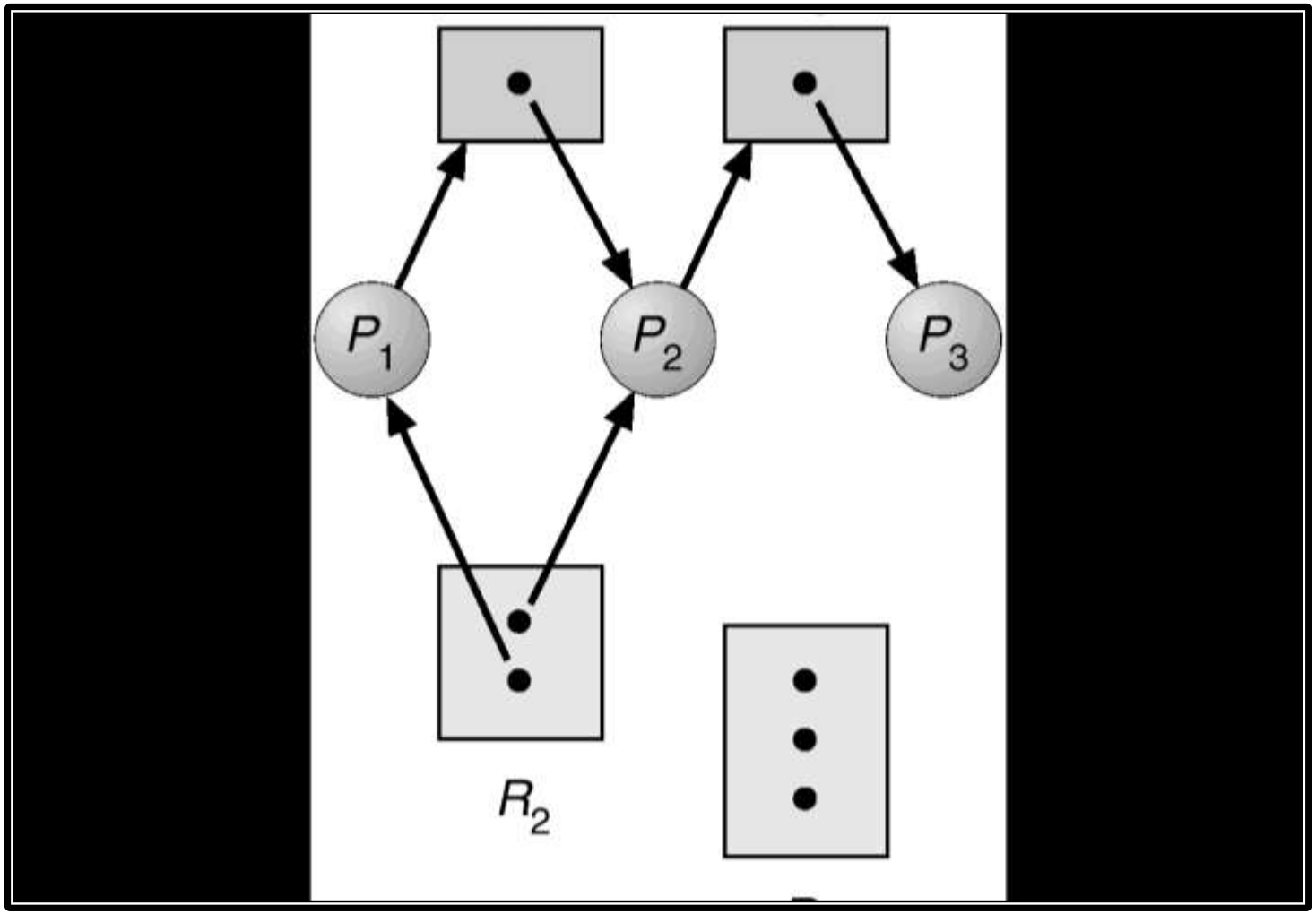


Resource Allocation Graph

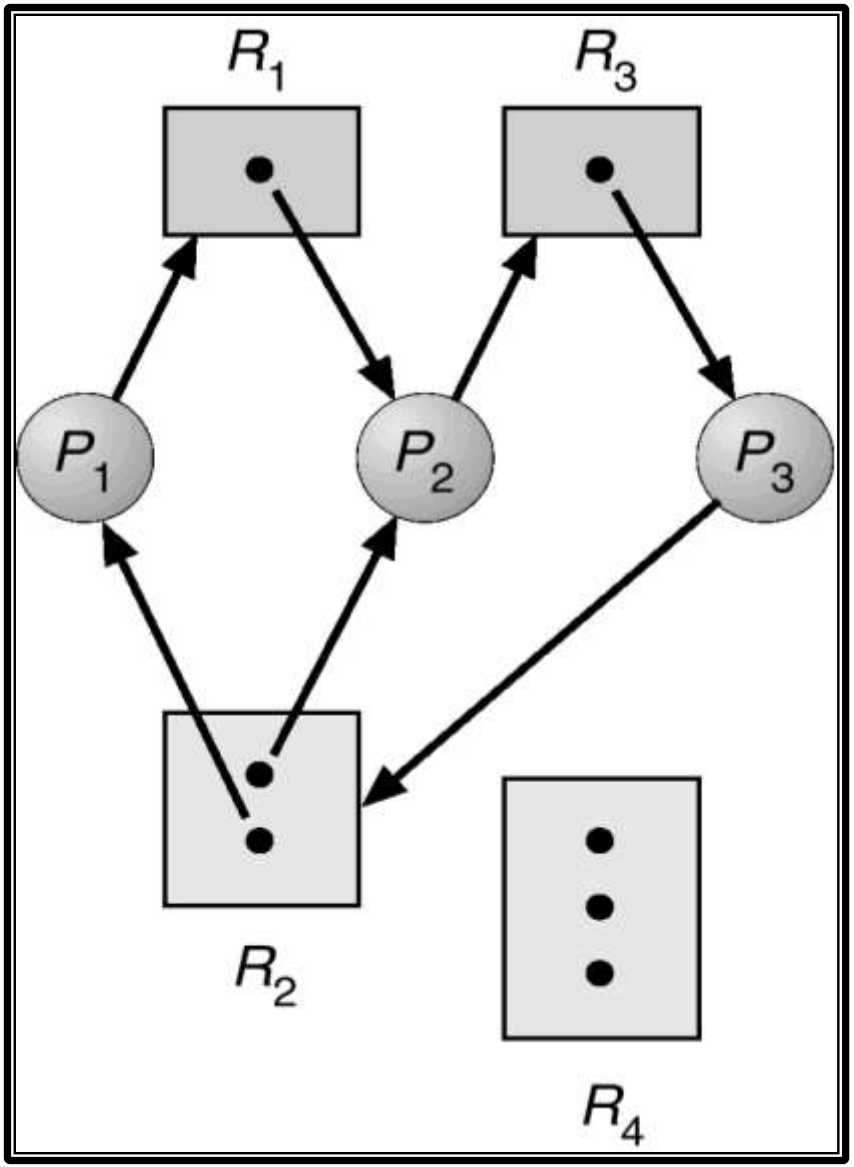
RAG with multiple resources of different types



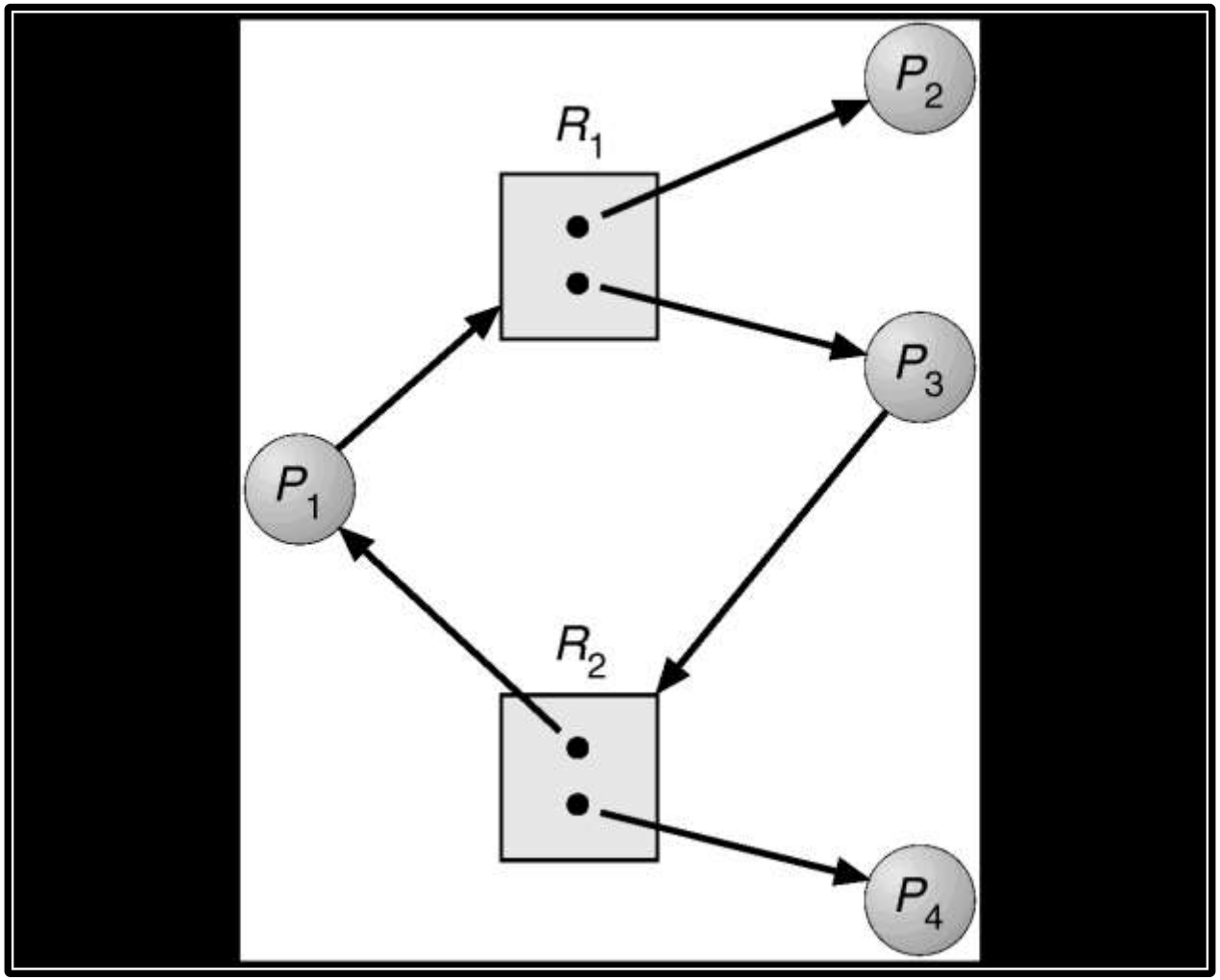
RAG Example 1



RAG Example 2



RAG Example 3



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - If only one instance per resource type, then deadlock.
  - If several instances per resource type, possibility of deadlock.

Sample Problems

Problem 1

A system has four processes P1 through P4 and two resource types R1 and R2. It has 2 units of R1 and 3 units of R2. Given that:

- P1 requests 2 units of R2 and 1 unit of R1.
- P2 holds 2 units of R1 and 1 unit of R2.
- P3 holds 1 unit of R2.
- P4 requests 1 unit of R1.

Show the resource graph for this state of the system. Is the system in deadlock? And if so, which process(es) are involved?

Problem 2

A system has five processes P1 through P5 and four resource types R1 through R4. It has 2 units of each resource type. Given that:

- P1 holds 1 unit of R1 and requests 1 unit of R4.
- P2 holds 1 unit of R3 and requests 1 unit of R2.
- P3 holds 1 unit of R2 and requests 1 unit of R3.
- P4 requests 1 unit of R4.
- P5 holds 1 unit of R3 and 1 unit of R2 and requests 1 unit of R3.

Show the resource graph for this state of the system. Is the system in deadlock? And if so, which process(es) are involved?

Deadlock Solutions

- Generally speaking we can deal with Deadlock problem in one of three ways:
  - Ensure that the system will never enter a deadlock state **(Prevention, Avoidance)**
  - Allow the system to enter a deadlock state and then recover **(Detection & Recovery)**
  - Ignore the problem and pretend that deadlocks never occur in the system **(Do nothing)**

Deadlock Solutions

- **Prevention**
  - Design system so that deadlock is impossible. It involves adopting a static policy that disallows one of the four conditions for deadlock.
- **Avoidance**
  - Steer around deadlock with smart scheduling. It involves making dynamic choices that guarantee prevention
- **Detection & recovery**
  - Check for deadlock periodically. Recover by killing a deadlocked processes and releasing its resources
- **Do nothing**
  - Prevention, avoidance and detection/recovery are expensive. If deadlock is rare, is it worth the overhead? Manual intervention (kill processes, reboot) if needed

PREVENTION

Deadlock Prevention

Restrain the ways resource allocation requests can be made, to ensure that at least one of the four necessary conditions is violated.

- Mutual Exclusion
- Hold and Wait
- No Preemption
- Circular Wait

Deadlock Prevention (cont…)

No Mutual Exclusion

- It is not possible to prevent D/L by denying the mutual exclusion condition only; because we cannot deny mutual exclusion for non-sharable resources like printer.
- We cannot deny mutual exclusion in case of a printer i.e. we cannot preempt a process that has printed half a page and give the printer to another process.

Deadlock Prevention (cont…)

Deny Hold and Wait

- We must guarantee that when ever a process request a resource, it does not hold any other resource.
- Option 1 - Deny wait.
  - Allocate the process with all the requested resources before it starts execution. If one or more resources are busy, nothing would be allocated and the process would just wait. (Low resource utilization)
- Option 2 - Deny hold.
  - A process may request some resources and use them. But before requesting any additional resources, it must release all the resources that are currently allocated. (Possibility of starvation)
- Example – (Tape drive, Disk drive, Printer)

Deadlock Prevention (cont…)

Allow Preemption

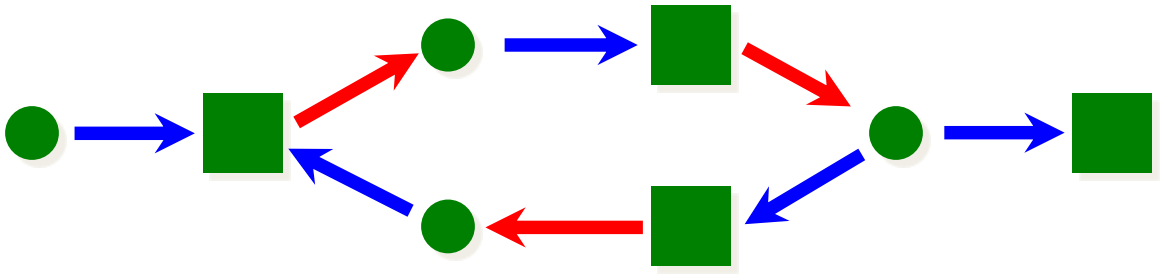
- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
- Process will be restarted only when it can regain its old resources as well as the new ones that it has requested.
- This protocol is often applied to resources whose state can be easily saved and restored later (preemptable resources), such as CPU registers and memory space. It cannot generally be applied to such resources as printers and tape drives. (Its difficult to take a tape drive away from a process that is busy writing a tape).

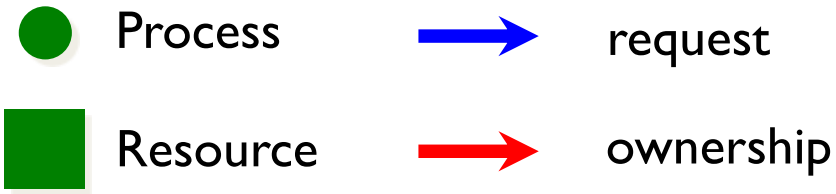
Deadlock Prevention (cont…)

Breaks Circular Wait (Solution 1)

- In resource allocation diagram: **process with an outgoing link must have no incoming links**
- Therefore, cannot have a loop!

Q: Which of these request links would be disallowed?



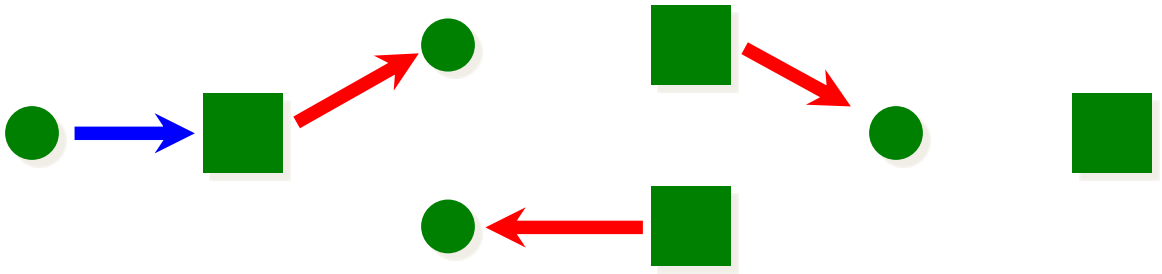


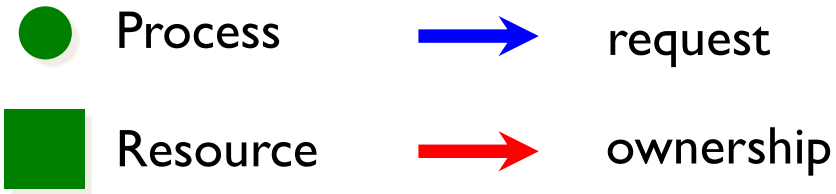
Deadlock Prevention (cont…)

Breaks Circular Wait (Solution 1)

- In resource allocation diagram: **process with an outgoing link must have no incoming links**
- Therefore, cannot have a loop!

Legal Links are:



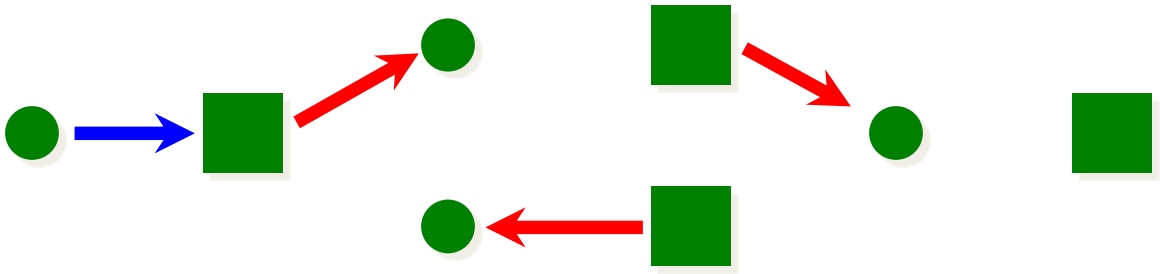


Deadlock Prevention (cont…)

Breaks Circular Wait (Solution 1)

- In resource allocation diagram: **process with an outgoing link must have no incoming links**
- Therefore, cannot have a loop!

Legal Links are:



Very constraining

- Often need more than one resource
- Hard to predict at the beginning what resources you'll need
- Releasing and re-requesting is inefficient, complicates programming, might lead to starvation

Deadlock Prevention (cont…)

Break Circular wait (Solution 2)

- **Break circular wait by requesting resources in order**
- Provide a global numbering to all resources.
- Each process can request resources only in an increasing order of enumeration.
- We assign a unique number to each resource type by using function

$$F: R \rightarrow N$$

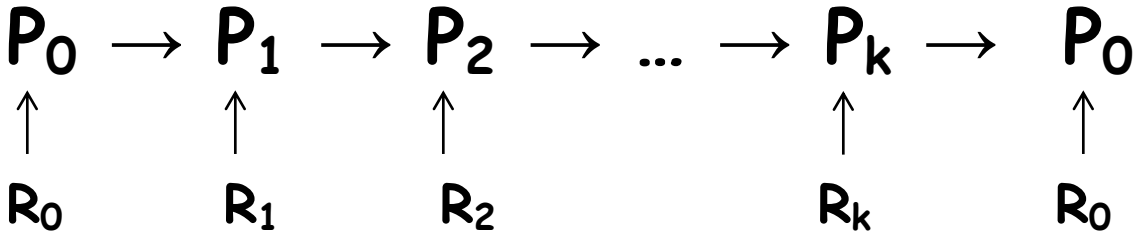
and make sure that processes request resources in an increasing order of enumeration.

- **Example**
  - Tape drive = 1, Disk drive = 5, and Printer = 12.
  - P1 holding Tape can request for both DD and Printer.
  - P2 holding DD can request for Printer but not for TD.

Deadlock Prevention (cont…)

Circular wait (Proof)

- Lets assume that a cycle exist among processes:



P_0 is holding R_0 and waiting for P_1 to release R_1;

P_1 is holding R_1 and waiting for P_2 to release R_2 and so on.

As per rule of circular wait:

$\Rightarrow F(R_0) < F(R_1) < \ldots F(R_k) < F(R_0)$

$\Rightarrow F(R_0) < F(R_0)$, which is impossible

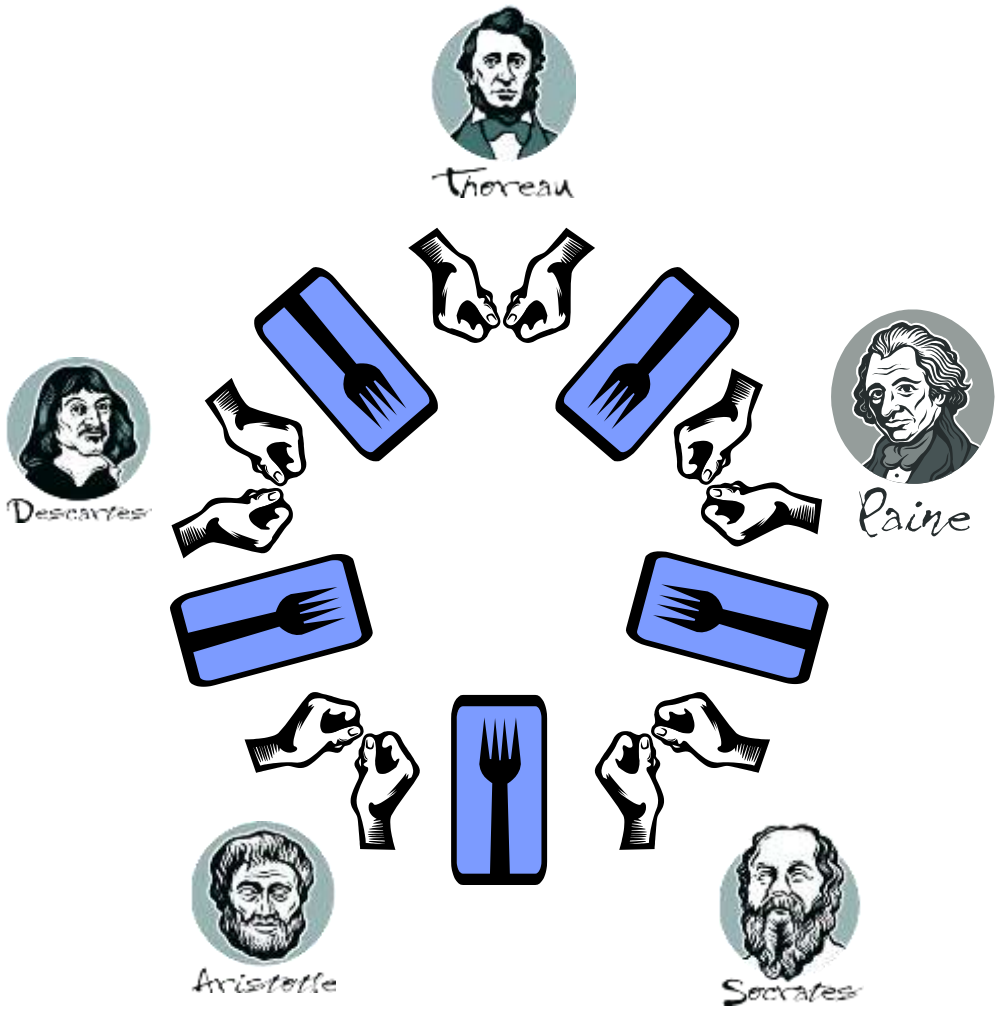
$\Rightarrow$ There can be no circular wait.

Dining Philosophers solution with unnumbered resources

Consider the Dining Philosopher problem

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1)%N);
        eat(); /* yummy */
        put_fork(i);
        put_fork((i+1)%N);
    }
}
```

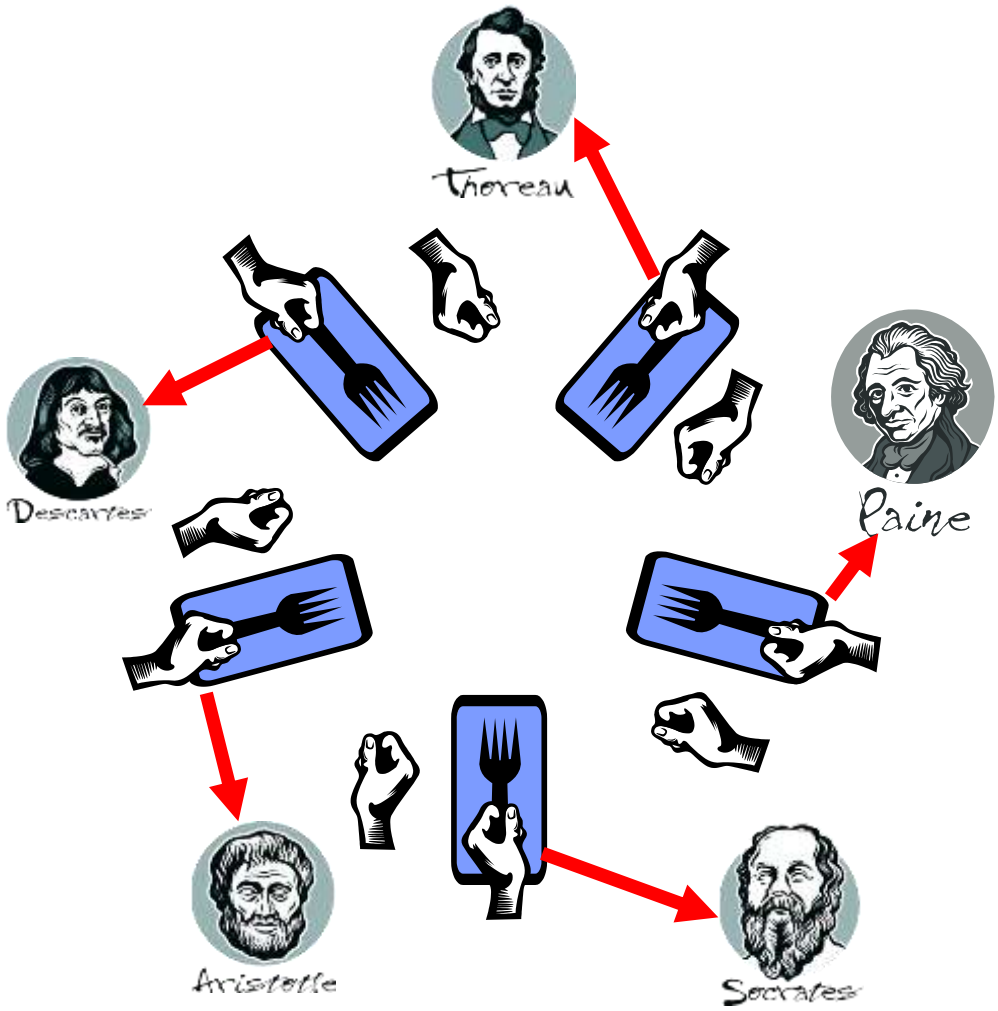


Dining Philosophers solution with unnumbered resources

Each philosopher first request the left fork, which is assigned to him

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1)%N);
        eat(); /* yummy */
        put_fork(i);
        put_fork((i+1)%N);
    }
}
```

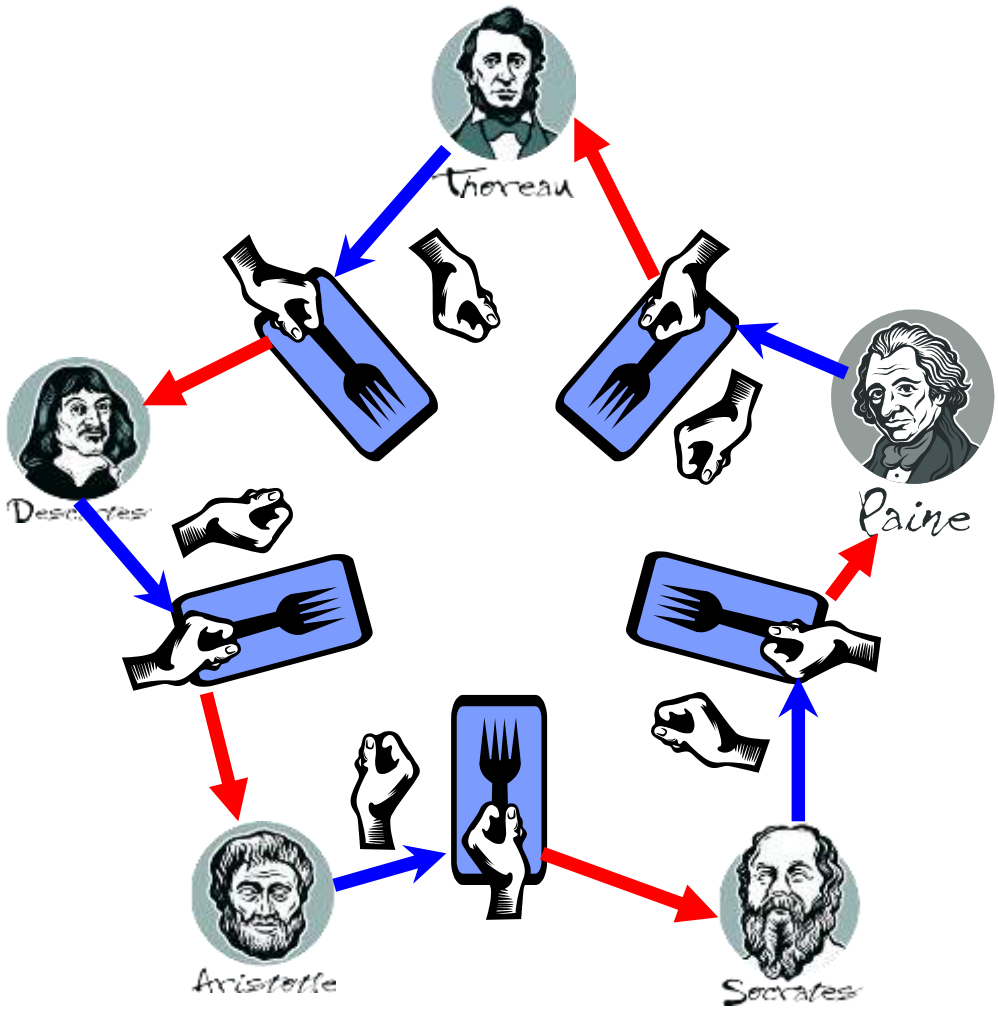


Dining Philosophers solution with unnumbered resources

Each philosopher then request the right fork, which makes a cycle

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1)%N);
        eat(); /* yummy */
        put_fork(i);
        put_fork((i+1)%N);
    }
}
```

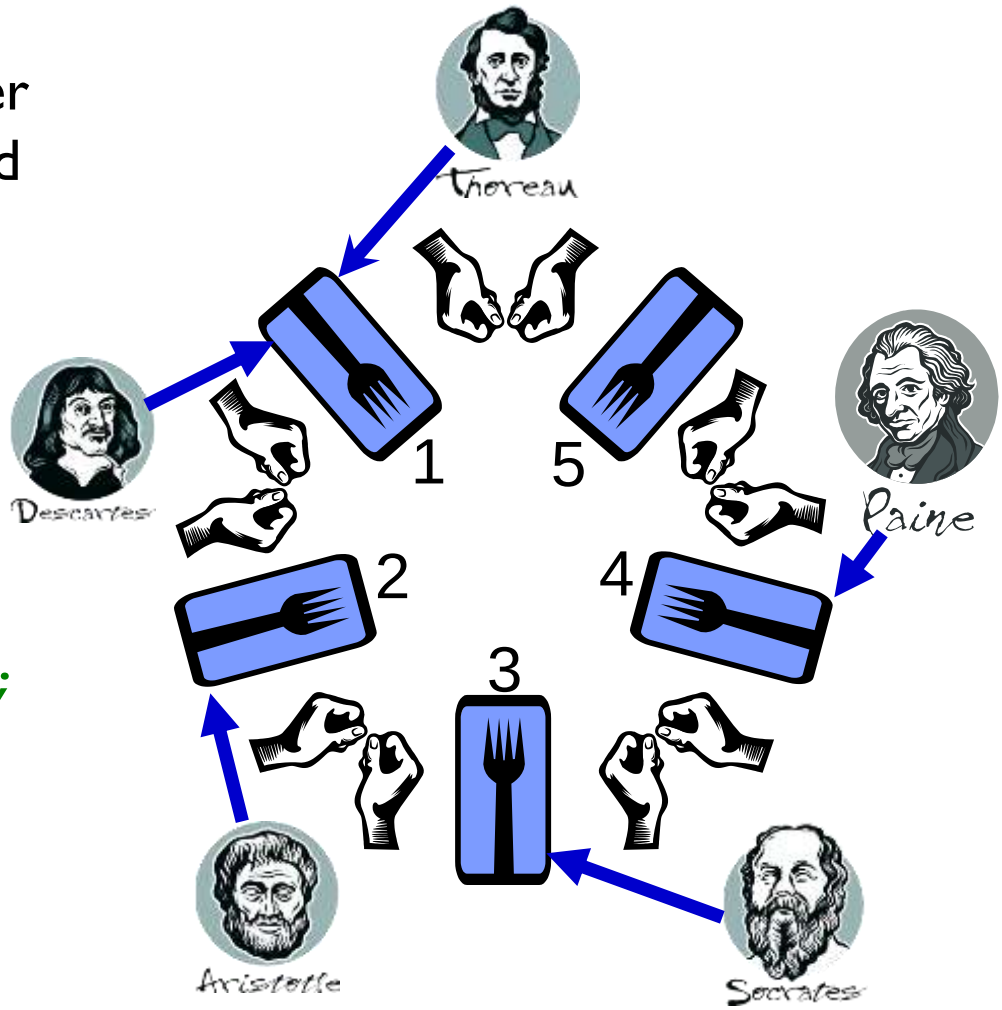


Dining Philosophers solution with numbered resources

Each philosopher first request lower numbered fork. Fork#1 is requested for by two philosophers

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```

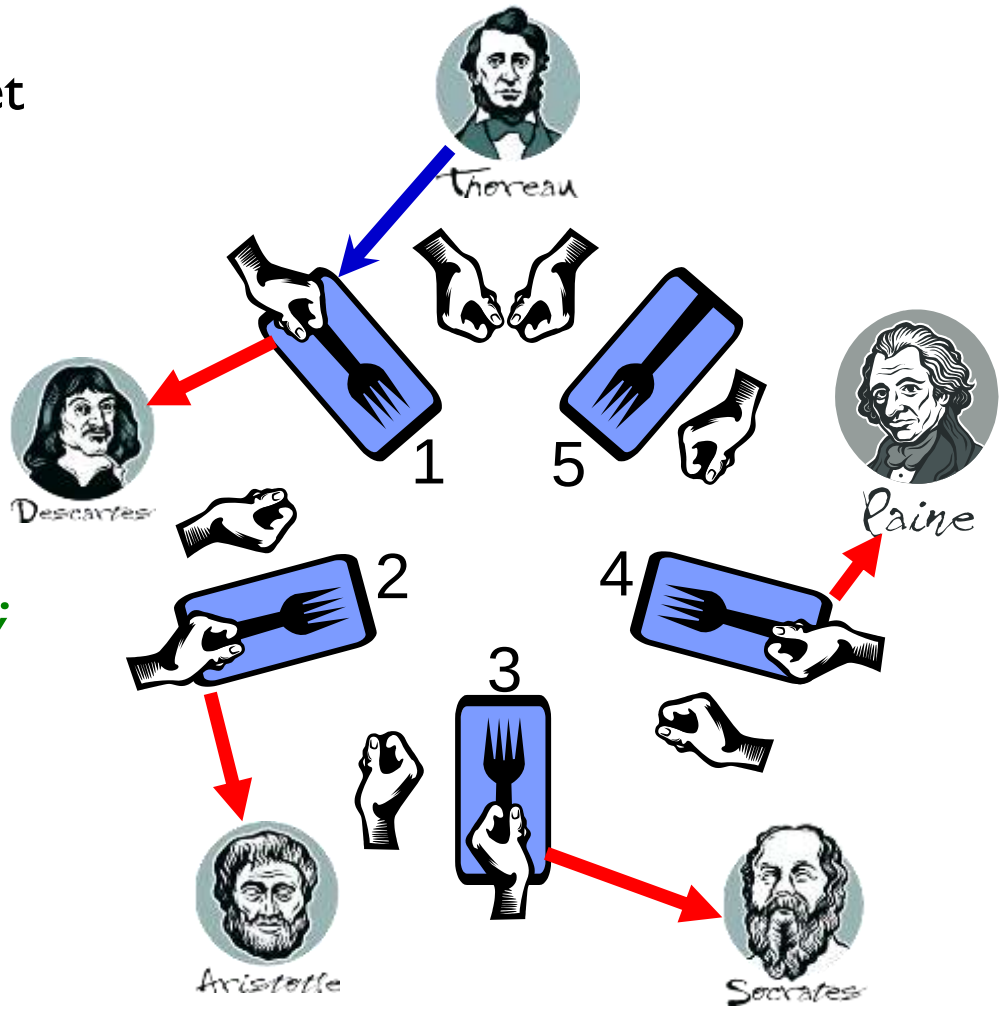


Dining Philosophers solution with numbered resources

One of the philosophers doesn't get the fork.

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```

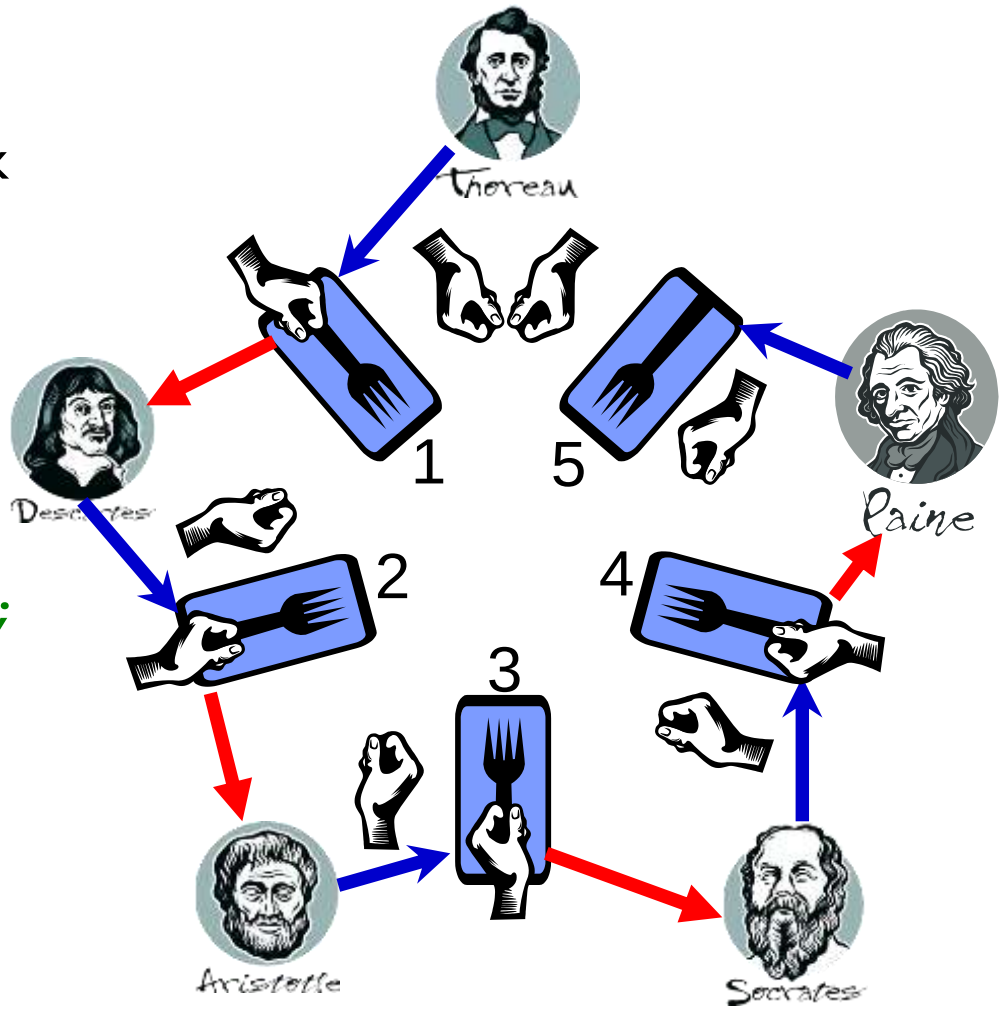


Dining Philosophers solution with numbered resources

Philosophers holding one resource then, request higher numbered fork

```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```

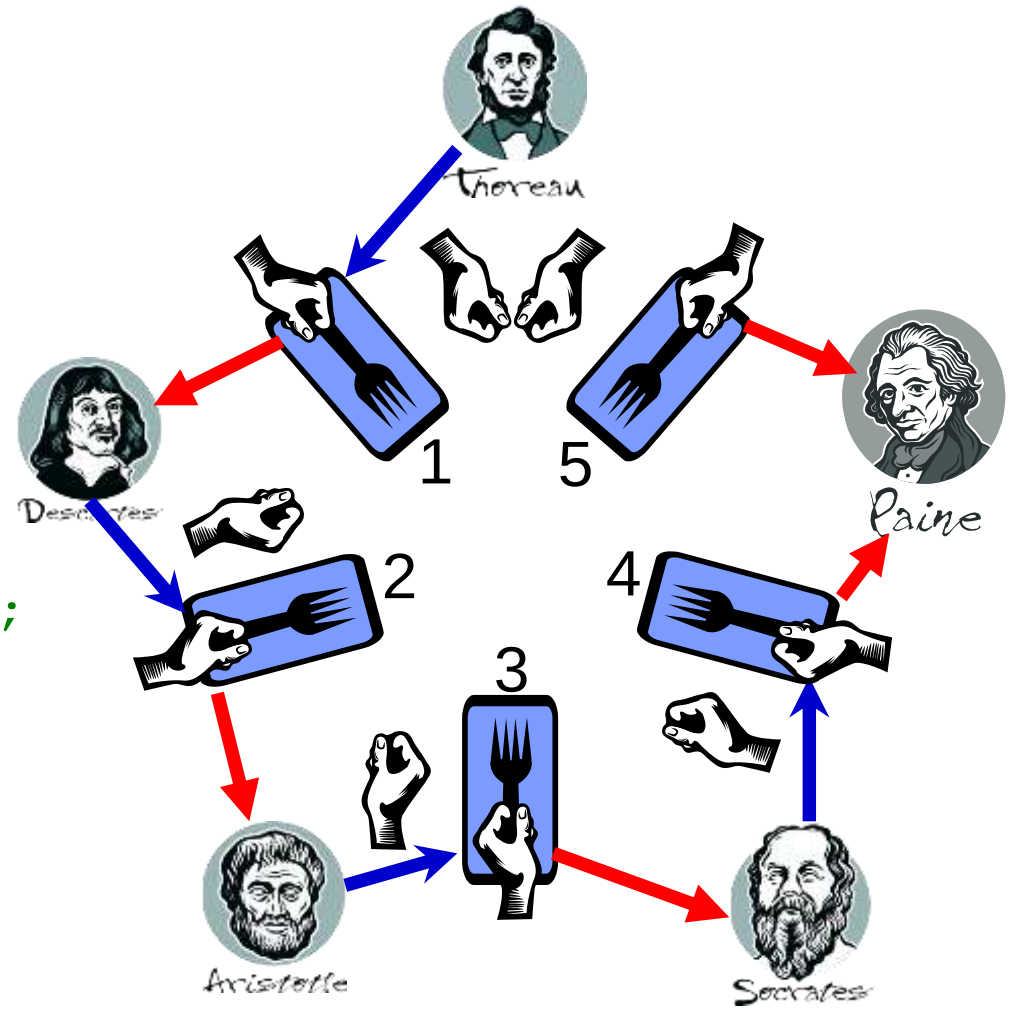


Dining Philosophers solution with numbered resources

One philosopher can eat!

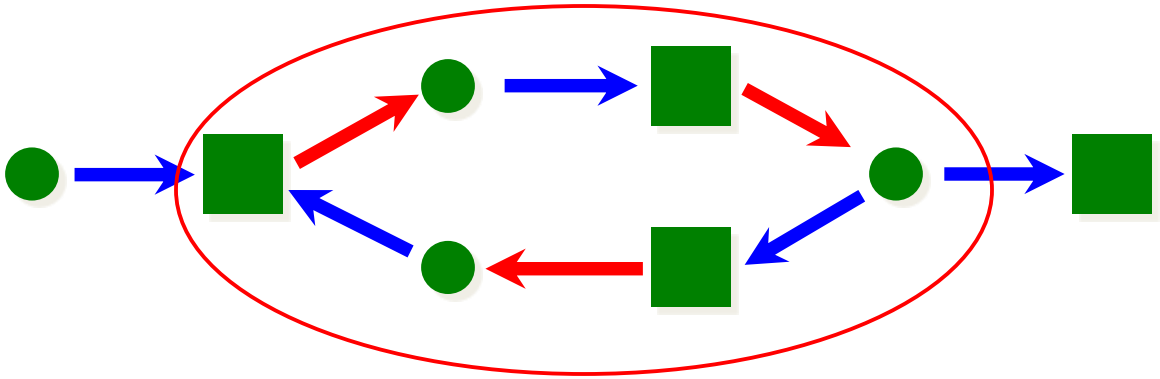
```
# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```



Ordered Resource requests Prevent Deadlock

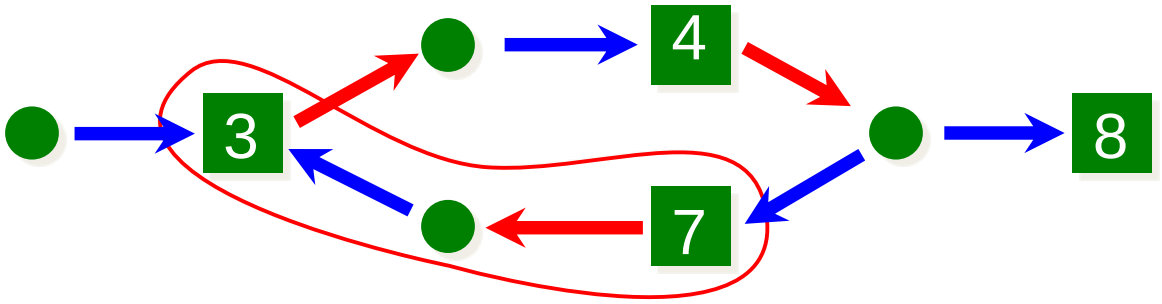
- Without numbering



Cycle!

Ordered Resource requests Prevent Deadlock

- With numbering

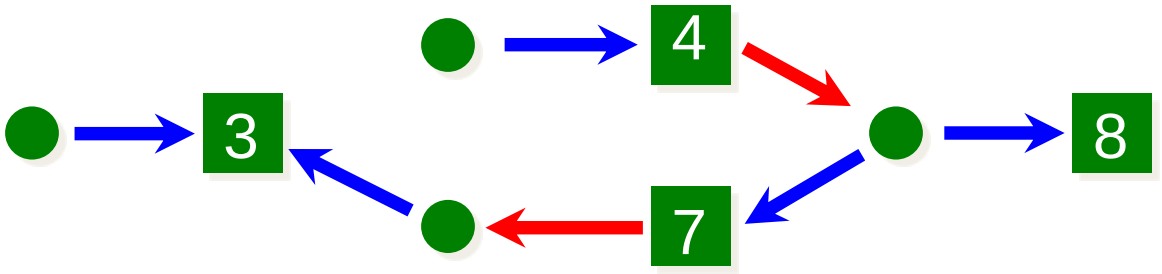


Contradiction:
Must have requested 3 first!

Ordered Resource requests Prevent Deadlock

Are we always in trouble without ordering resources?

- No, not always:



- Ordered resource requests are sufficient to avoid deadlock, but not necessary
- Convenient, but may be conservative

Sample Problem

Problem

Consider the deadlock situation that could occur in the dining-philosophers problem. When the philosophers obtain the chopsticks one at a time. Discuss how the four necessary conditions for deadlock indeed hold in this setting. Discuss how deadlocks could be avoided by eliminating any one of the four conditions.

AVOIDANCE

Deadlock Avoidance

- Deadlock avoidance is subtly different from deadlock prevention: we allow the three necessary conditions for deadlock, but we dynamically allocate resources in such a way that deadlock never occurs. There are two principal approaches.
  1. **Do not start a process if its demands might lead to deadlock.** This strategy is very conservative and thus inefficient.
  2. **Do not grant a resource request if this allocation might lead to deadlock.** The basic idea is that a request is granted only if some allocation of the remaining free resources is sufficient to allow all processes to complete.
- Both strategies require processes to state their resource requirements in advance.

Deadlock Avoidance (cont…)

- The implementation of deadlock avoidance requires that the system has an advanced information available about the future use of resources by processes.
- Simplest way is 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 total number of resources available in the system, maximum demands of the processes and the number of resources currently allocated to the processes.

Deadlock Avoidance (cont…)

- **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 a safe state if there is at least one sequence of allocation of resources to processes that does not result in a dead lock.** (Sequence of processes does not matter).
- **Safe Sequence** is the sequence of execution of processes in which OS fulfills requests of all the processes and still avoids dead lock.
- **Basic Facts.**
  - If a system is in safe state ⇒ no deadlocks.
  - If a system is in unsafe state ⇒ possibility of deadlock due to the behavior of processes.
- DL Avoidance ensure that a system never enters in unsafe state.
- A system can be in safe or unsafe state. Unsafe state does not necessarily means that system is in DL. It may lead to a DL.

Sample Problem

- System with 12 tape drives and three processes. Current system state is as shown

Process	Max Need	Allocated
P_0	10	5
P_1	4	2
P_2	9	2

- Is the system in safe state, if yes give a safe sequence?

Sample Problem

- Assuming that P2 requests and is allocated one more instance of tape drive. The new system state will be:

Process	Max Need	Allocated
P_0	10	5
P_1	4	2
P_2	9	3

- Is the system in safe state, if yes give a safe sequence? If not explain which process may cause a dead lock.

Sample Problem

- Given 5 total units of the resource, tell whether the following system is in a safe or unsafe state.

Process	Max Need	Allocated
P_1	2	1
P_2	3	1
P_3	4	2
P_4	5	0

Sample Problem

- Given a total of 10 units of a resource type, and given the safe state shown below, should P_2 be granted a request of 2 additional resources? Show your work.

Process	Max Need	Allocated
P_1	5	2
P_2	6	1
P_3	6	2
P_4	2	1
P_5	4	1

Sample Problem

Problem 1

- Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock-free.

Problem 2

- A system has two processes and three identical resources. Each process needs a maximum of two resources. Is deadlock possible? Explain your answer.

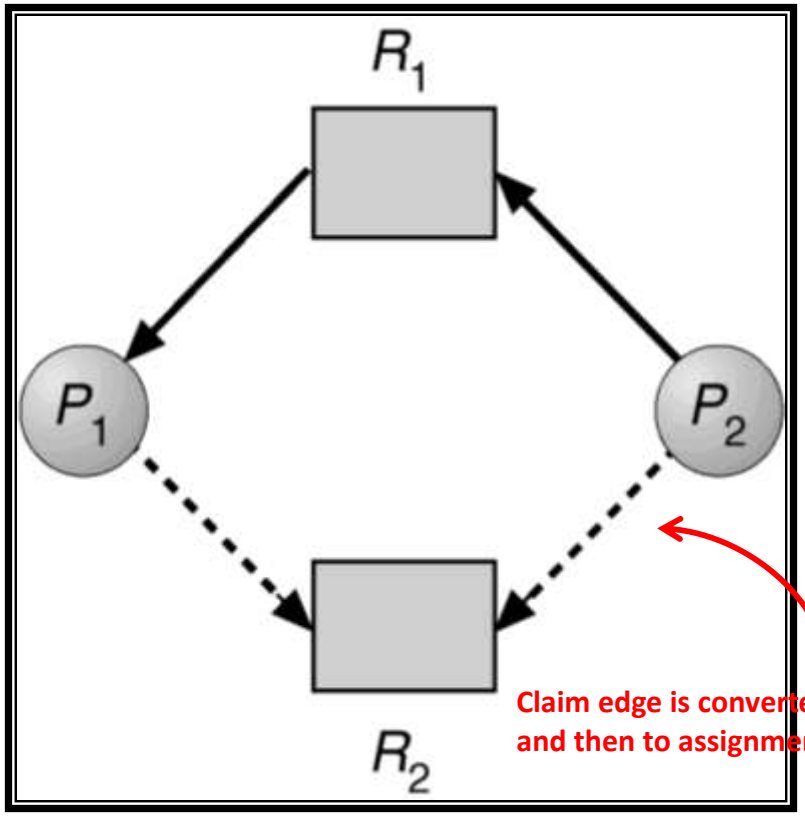
Problem 3

- A computer has six tape drives, with n processes competing for them. Each process may need two drives. For which values of n is the system deadlock free?

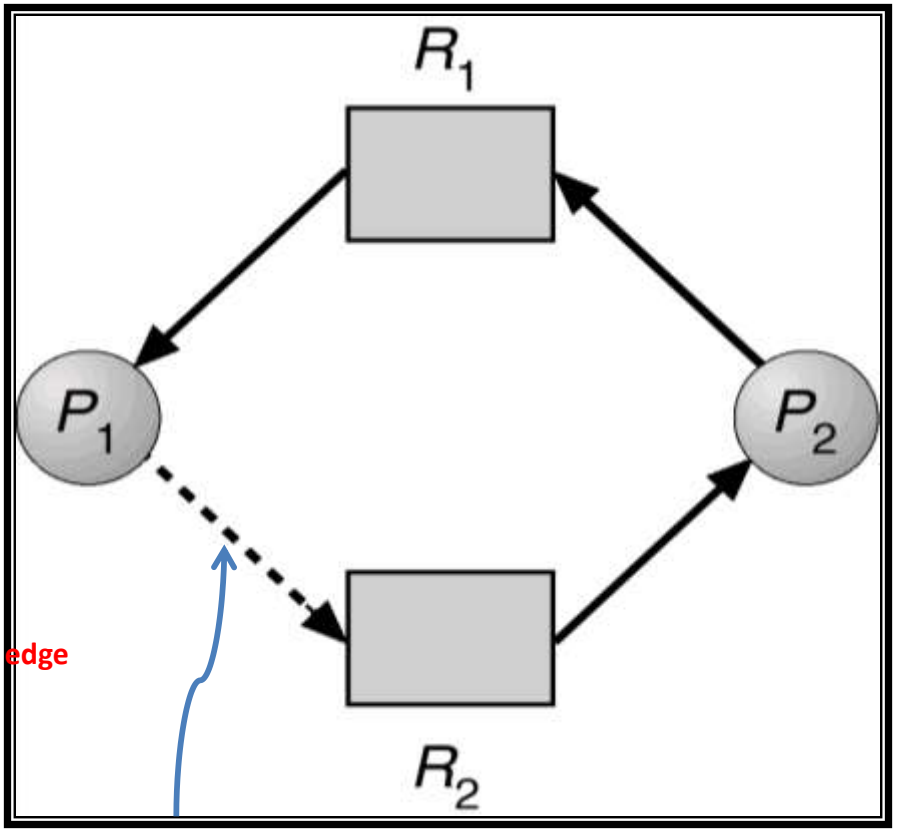
RAG Algorithm

- Used for DL avoidance in case of single instance of each resource.
- **Claim edge** Pi $\rightarrow$ Rj indicates that process Pi may request an instance of resource Rj; represented by a dashed line.
- Claim edge converts to **request edge** when a process requests a resource.
- When a resource is assigned to a process, request edge reconverts to an **assignment edge**.
- All processes will inform the algo in advance which all resources they will be requiring in their life cycle.

RAG Algorithm (cont…)



Claim edge is converted to request edge
and then to assignment edge



Before converting a claim edge to request edge,
we need to check whether it will create a cycle in the directed Graph or not.

RAG Algorithm (cont…)

As seen in previous slides, in our digraph model with one resource of one kind, the detection of a deadlock requires that we detect a directed cycle in a processor resource digraph. This can be simply stated as follows:

- Choose a process node as a root node to initiate a depth first traversal.
- Traverse the digraph in depth first mode.
- Mark process nodes as we traverse the graph.
- If a marked node is revisited then a deadlock exist.

Banker's Algorithm

- This is a deadlock avoidance algorithm for resources with multiple instances.
- **When a process initiates a request the algorithm checks whether after the grant of this request the system will remain in safe state. If yes the request is granted, if not the request is denied.**
- Algorithm is based on resource denial if there is a suspected risk of a deadlock.
- Each process must in advance inform the algo the number of instances it may require in its life time.
- When a process requests a resource it may have to wait, i.e. in case the system is going to unsafe state by allocation of that resource; the process must wait.
- When a process gets all its resources it must return them in a finite amount of time.
- Example. Suppose U go to a bank with Rs. 100,000/ cheque and request to get the money as Rs.10/ notes only.

DS for Banker's Algorithm

- n = Number of processes.
- m = Number of resource types.
- **Available**: Vector of length m, indicates the number of available instances of resources of each type. **Available[j] = k** means that there are k available instances of resource type R_j.
- **Max**: n x m matrix, indicates the maximum demand of resources of each process. **Max[i,j] = k** means that process P_i may request at most k instances of resource type R_j.
- **Allocation**: n x m matrix, indicates the no of instances of resources of each type currently allocated to each process. **Allocation[i,j] = k** means that P_i is currently allocated k instances of resource type R_j.
- **Need**: n x m matrix indicates the remaining resource need of each process. **Need[i,j] = k** means P_i may need k more instances of R_j to complete its task.

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

Safety Algorithm

- Used to determine whether or not a system is in safe state.
- Check the system state and sees if there is any process whose request can be fulfilled. If yes note it down in safe sequence and then add its allocation vector in available vector.
- So now we have got a new available vector with some new instances of various resource types.
- The above step is repeated with all the processes and finally either we will be able to finish all the processes with a safe sequence OR we may get stuck.
- **The safety algo returns safe or unsafe whatever is applicable to the Bankers Algorithm.**

Resource Request Algorithm

Let **Request$_i$** be the request vector for process P_i. If **Request$_i$ [j] = k** then P_i wants k instances of resource type R_j.

1. **If Request$_i$ ≤ Need$_i$ then**
   go to step 2
   else
   report error (since process has exceeded its maximum claim).
2. **If Request$_i$ ≤ Available then**
   go to step 3
   else
   P_i must wait (since resources are not available).
3. **Update system table:**
   Available = Available – Request$_i$;
   Allocation$_i$ = Allocation$_i$ + Request$_i$;
   Need$_i$ = Need$_i$ – Request$_i$;

Resource Request Algorithm (cont…)

Now the Banker's Algo invokes the safety algo with this new system state and the safety algo tells the banking algo whether the new state will be safe or not.

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

Sample Problem

- Five processes: P_0 … P_4
- Three resource types: *A* (10 instances), *B* (5 instances), *C* (7 instances)
- System state is shown below:

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

Sample Problem (cont…)

- $Need_i = Max_i - Allocation_i$
- Need matrix

Process	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

Sample Problem (cont…)

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

- Safe Sequence: <>

Sample Problem (cont…)

Process	Need			Allocation			Available		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0	5	3	2
P_2	6	0	0	3	0	2			
P_3	0	1	1	2	1	1			
P_4	4	3	1	0	0	2			

- Safe Sequence: <P1>

Sample Problem (cont…)

Process	Need			Allocation			Available		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0	5	3	2
P_2	6	0	0	3	0	2	7	4	3
P_3	0	1	1	2	1	1			
P_4	4	3	1	0	0	2			

- Safe Sequence: <P1, P3>

Sample Problem (cont…)

Process	Need			Allocation			Available		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0	5	3	2
P_2	6	0	0	3	0	2	7	4	3
P_3	0	1	1	2	1	1	7	4	5
P_4	4	3	1	0	0	2			

- Safe Sequence: <P1, P3, P4>

Sample Problem (cont…)

Process	Need			Allocation			Available		
	A	B	C	A	B	C	A	B	C
P_0	7	4	3	0	1	0	3	3	2
P_1	1	2	2	2	0	0	5	3	2
P_2	6	0	0	3	0	2	7	4	3
P_3	0	1	1	2	1	1	7	4	5
P_4	4	3	1	0	0	2	7	5	5

- Safe Sequence: <P1, P3, P4, P0>

Sample Problem (cont…)

- Final safe sequence:

  <P1, P3, P4, P0, P2>

- Not a unique sequence
- Possible safe sequences for this example:

  <P1,P3,P4,P0,P2>, <P1,P3,P4,P2,P0>,
  <P1,P3,P2,P0,P4>, <P1,P3,P2,P4,P0>,
  <P1,P3,P0,P2,P4>, <P1,P3,P0,P4,P2>

Sample Problem

- Four processes: P_1 … P_4
- Three resource types: *A* (9 instances), *B* (3 instances), *C* (6 instances)
- System state is shown below:

Process	Max Matrix			**Allocation Matrix**			Available Vector		
	A	B	C	A	B	C	A	B	C
P_1	3	2	2	1	0	0	0	1	1
P_2	6	1	3	6	1	2			
P_3	3	1	4	2	1	1			
P_4	4	2	2	0	0	2			

Sample Problem

- Five processes: P_0 … P_4
- Three resource types: A (3 instances), B (14 instances), C (12 instances), D (12 instances)
- System state is shown below:

Process	Max Matrix				Allocation Matrix				Available Vector			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	7	5	0	1	0	0	0				
P_2	2	3	5	6	1	3	5	4				
P_3	0	6	5	2	0	6	3	2				
P_4	0	6	5	6	0	0	1	4				

Sample Problem

- A system has four processes and five allocatable resources. The current allocation and maximum needs are as follows:

	Allocated	Maximum	Available
Process A	1 0 2 1 1	1 1 2 1 3	0 0 **x** 1 2
Process B	2 0 1 1 0	2 2 2 1 0	
Process C	1 1 0 1 0	2 1 3 1 0	
Process D	1 1 1 1 0	1 1 2 2 1	

What is the smallest value of x for which this is a safe state?

Sample Problem

- Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix *Need?*
b. Is the system in a safe state?
c. If a request from process *P1 arrives for (0,4,2,0), can the request* be granted immediately?

DETECTION
&
RECOVERY

Deadlock Detection and Recovery

Both prevention and avoidance of deadlock lead to conservative allocation of resources, with corresponding inefficiencies. Deadlock detection takes the opposite approach:

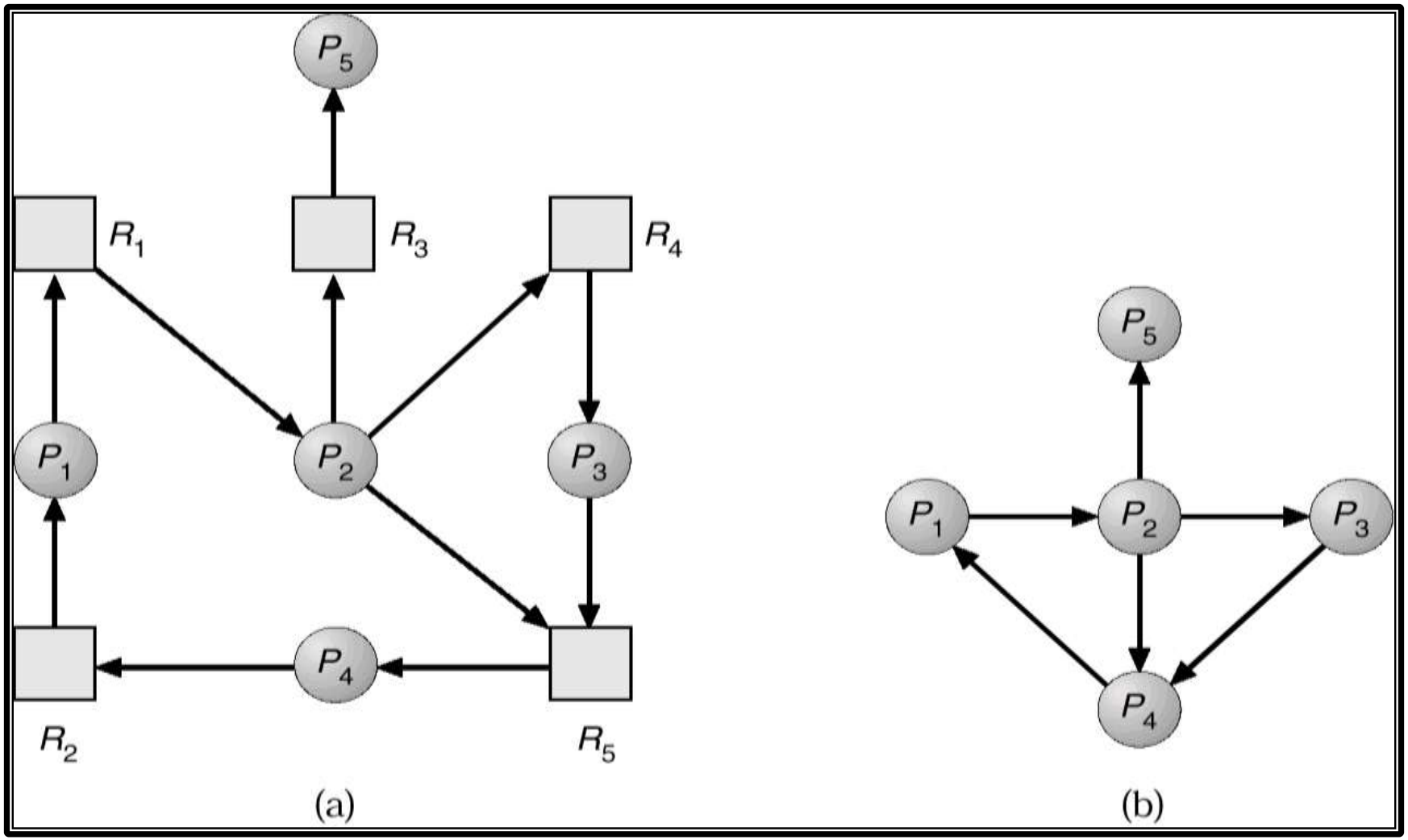
- Make allocations liberally, allowing deadlock to occur (on the assumption that it will be rare);
- Apply a detection algorithm periodically to check for deadlock;
- Apply a recovery algorithm when necessary.

Deadlock Detection Algo for Single Instance of each Resource Type

- This algorithm use wait-for-graph that can be derived from RAG.
- To derive a wait for graph, you collapse the resource node.

  $P_i \rightarrow P_j$ means P_i is waiting for P_j.
- If a cycle exist in the wait for graph we say that a deadlock exist.
- To detect deadlock, the system maintains the wait for graph and periodically invokes an algorithm that searches for a cycle in the wait-for graph.
- The algorithm is expensive—it requires an order of n^2 operations, where n is the number of vertices in the graph.

RAG and Wait for Graph



Resource-Allocation Graph

Corresponding wait-for graph

Deadlock Detection

How often should the detection algorithm be invoked?

1. Every time a request for allocation cannot be granted immediately—expensive but process causing the deadlock is identified, along with processes involved in deadlock.
2. Keep monitoring CPU usage, and when it goes below a certain level, invoke the algorithm.
3. Run it periodically after a specified time interval.
4. Run the algo arbitrarily/randomly—In this case, we may find a number of cycles in the system but may not be able to find out which process has created these cycles.

Deadlock Recovery

Recovery algorithms vary a lot in their severity:

1. Abort all deadlocked processes. Though drastic, this is probably the most common approach!
2. Back-up all deadlocked processes. This requires potentially expensive rollback mechanisms, and of course the original deadlock may recur.
3. Abort deadlocked processes one at a time until the deadlock no longer exists.
4. Preempt resources until the deadlock no longer exists.

- With the last two approaches, processes or resources are chosen to minimize the global "loss" to the set of processes, by minimizing the loss of useful processing with respect to relative priorities.

Deadlock Detection and Recovery

- When a detection algo determines that a dead lock exists, it either let the operator deal with the deadlock manually or recover from dead lock automatically.
- There are two options for breaking a deadlock:
  1. Process Termination.
  2. Resource Preemption.

Deadlock Detection and Recovery

Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated. While selecting the victim process consider the following issues:
  1. Priority of a process.
  2. How long the process has run.
  3. Resources already used by a process.
  4. Further resources the process needs to complete.
  5. How many child processes will be needed to terminate.
  6. Is the process interactive or batch?

Deadlock Detection and Recovery

Resource Preemption

In resource preemption, select a process and take back some resources from that process and allocate those resources to other requesting processes and bring the system out of deadlock. Three important issues to be considered are:

- **Selecting a victim** – Which resources and which processes are to be preempted?
- **Rollback** – If we preempt a resource from a process, what should be done with that process? We need to return the victim to some safe state from where it can be restarted later on.
- **Starvation** – There is a possibility that same process may always be picked as victim, so to avoid this include number of rollbacks in cost factor of victim selection.

SUMMARY

We're done for now, but Todo's for you after this lecture…

- Go through the slides and Book Sections: 7.1 to 7.7
- Try solving the End Exercise Problems given at the end of Chapter 7

If you have problems visit me in counseling hours. . . .