Operating Systems

Lecture 5.4

Dead Locks

Instructor: Muhammad Arif Butt, PhD

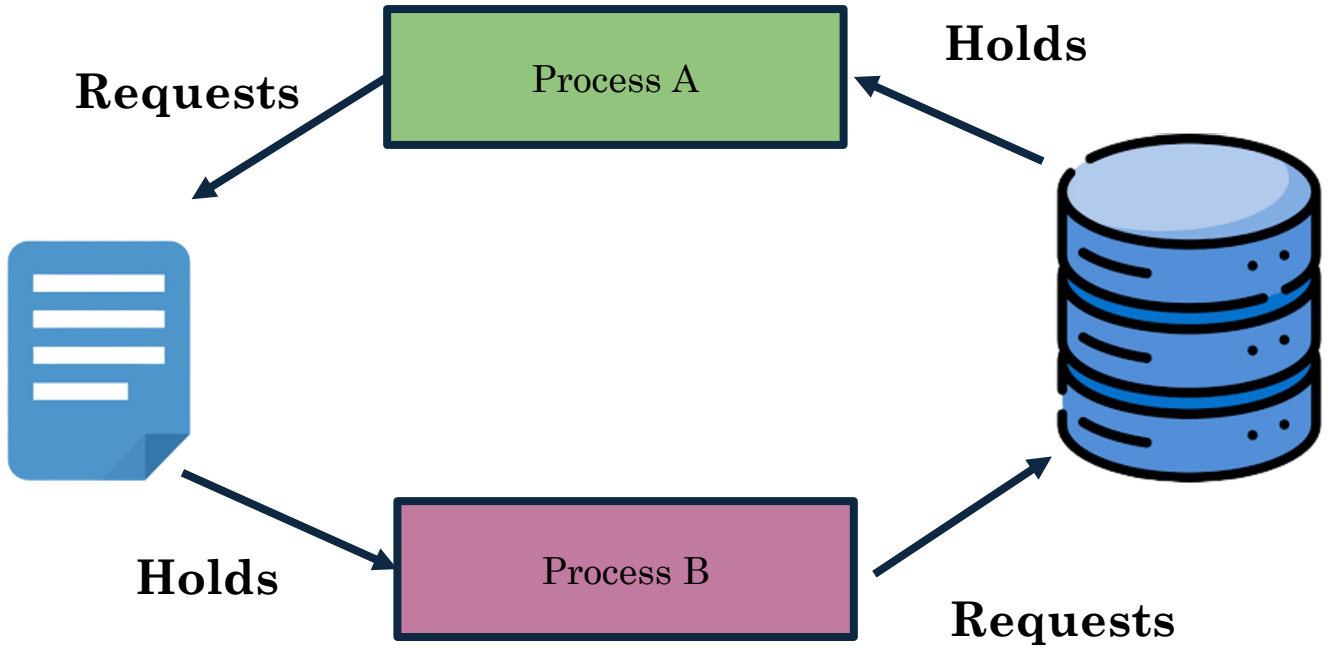
Lecture Agenda

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


Deadlock

Deadlock Problem

- A set of processes is said to be in a deadlock 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.**

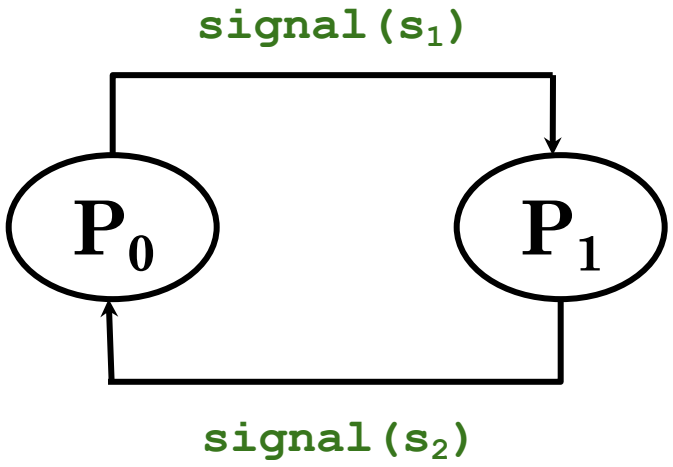


Deadlock Problem - Examples

- 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 deadlock
- A person going down a ladder while another person is climbing up the ladder
- Two cars crossing a single lane bridge from opposite direction.
- Two trains travelling toward each other on the same track.
- 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 - Semaphores

```
semaphore s1 = 1;
semaphore s2 = 1;
```



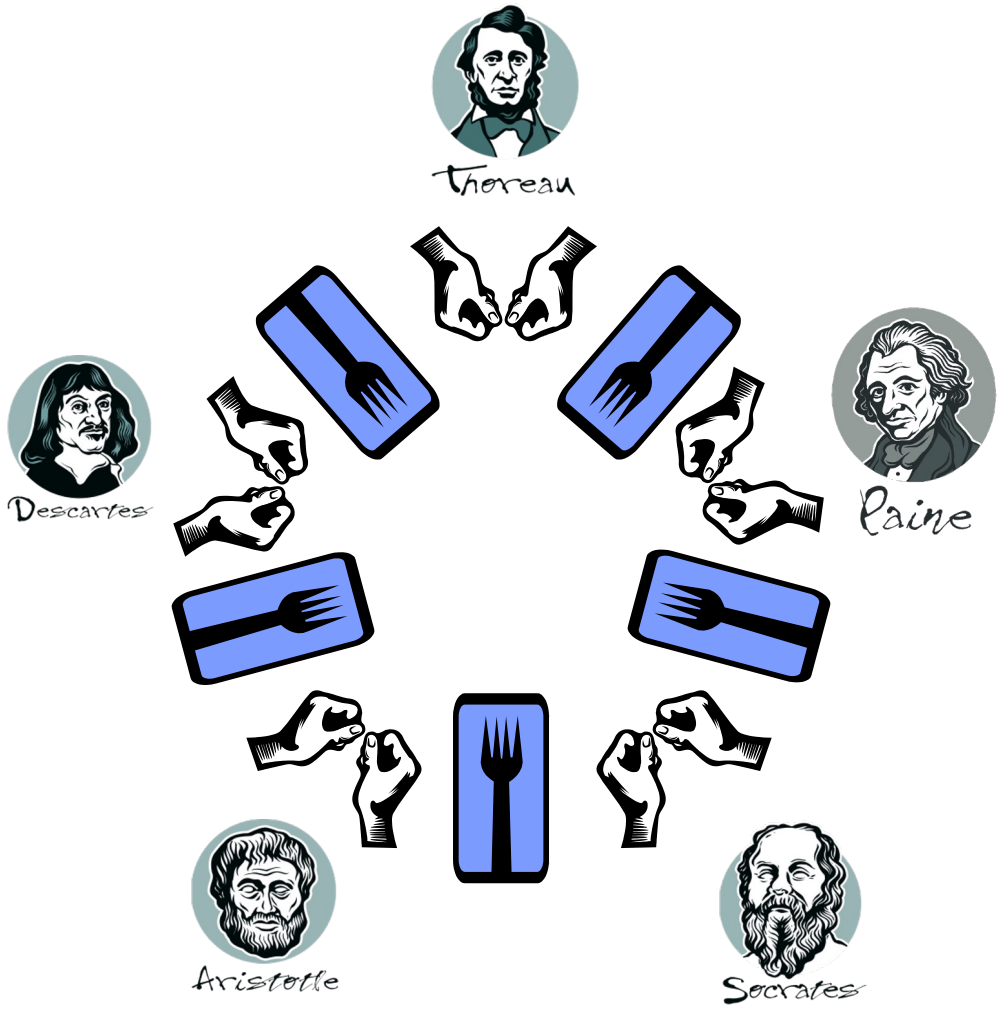
P_0	P_1
wait(s1);	wait(s2);
wait(s2);	wait(s1);
.	.
signal(s1);	signal(s2);
signal(s2);	signal(s1);

s1	s2
0	0

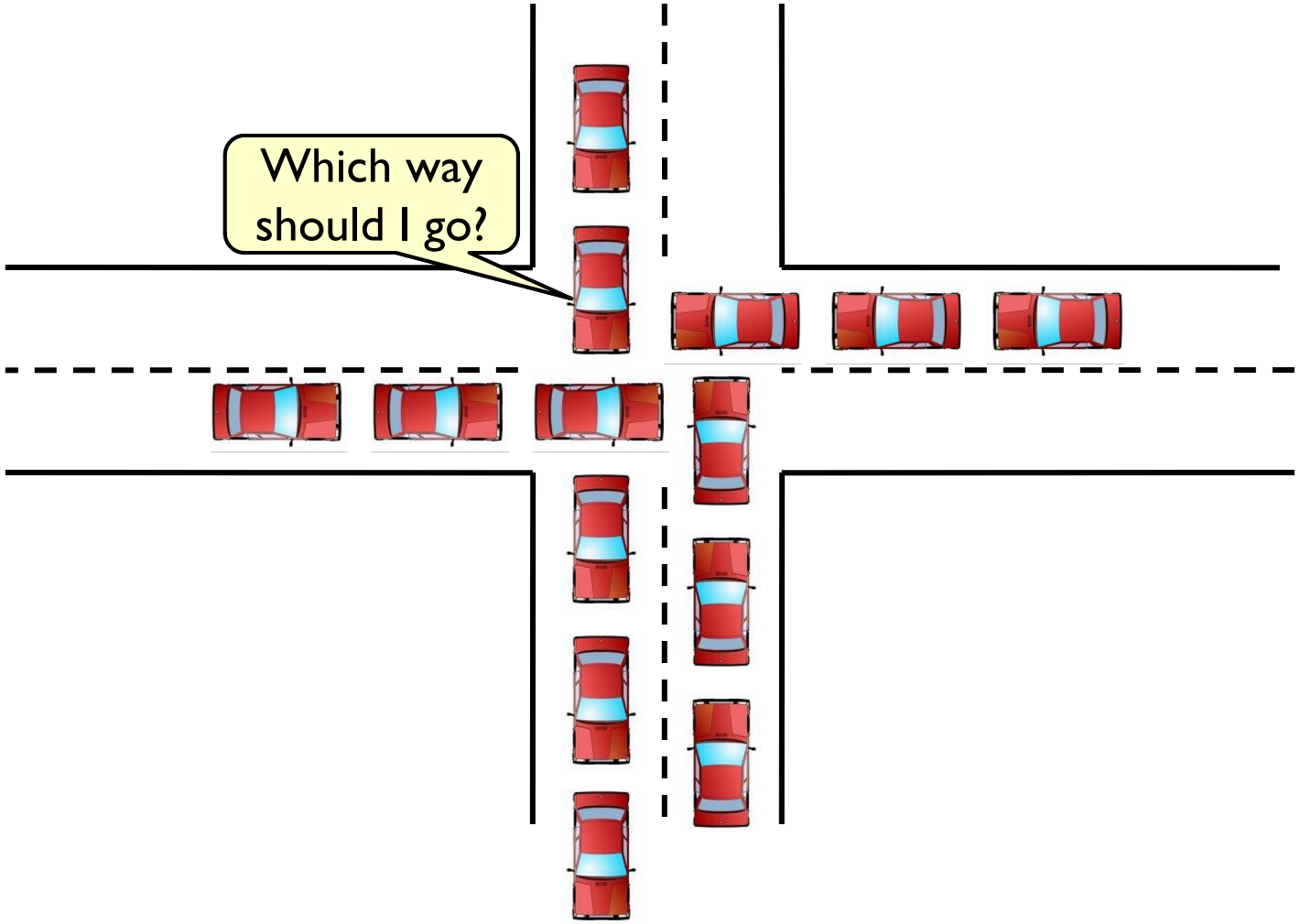
Blocked ...

Deadlock - Dining Philosopher Problem

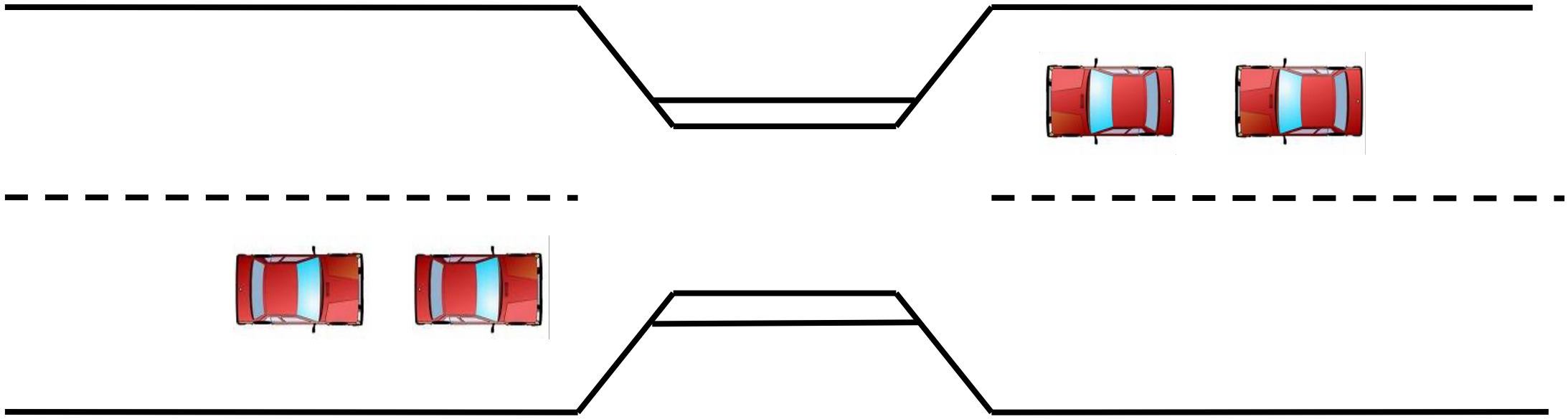
- Consider the dining philosophers problem.
- All five philosophers become hungry at the same time.
- All pick up the chopsticks on their left and look for the right, which was held by the neighboring philosopher.
- No one put the chopstick back and wait for the right chopstick and finally all starved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



Deadlock in the real world

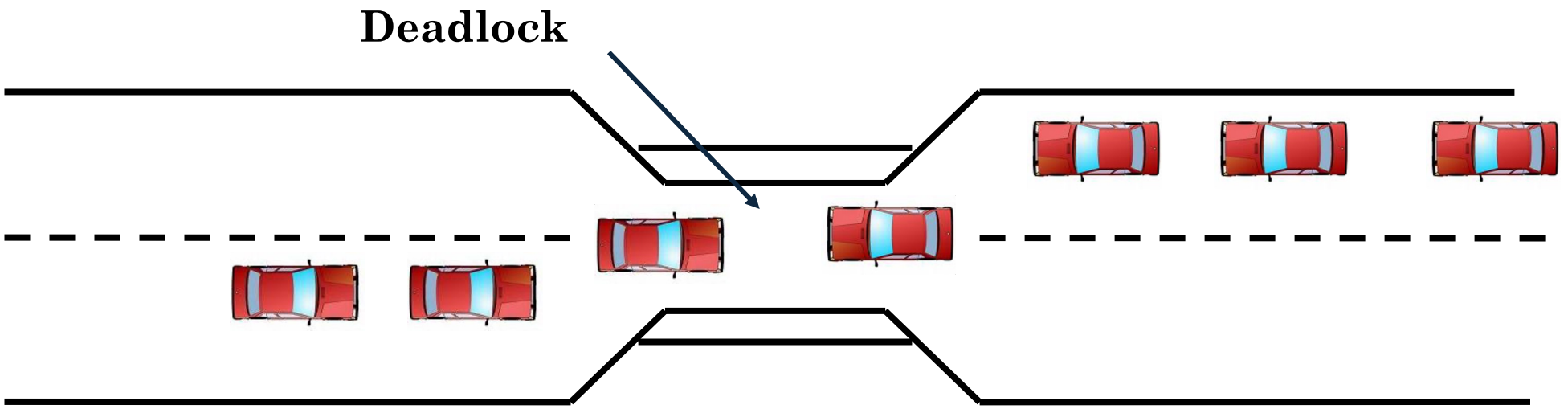


Deadlock: One-lane Bridge



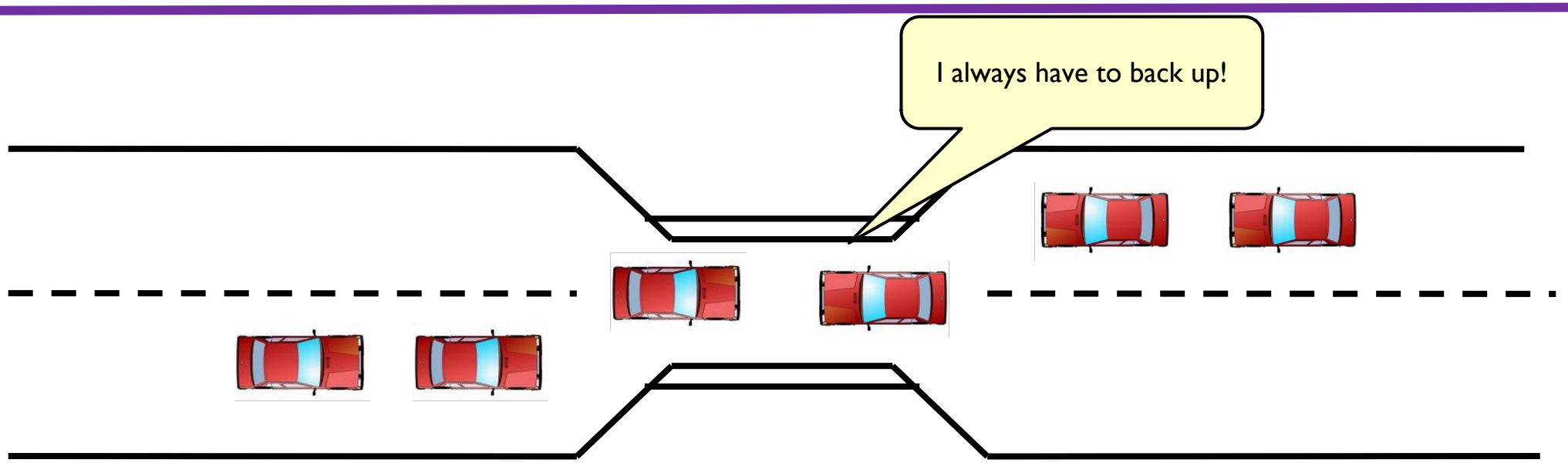
- Traffic only in one direction
- Each section of a bridge can be viewed as a resource

Deadlock: One-lane Bridge



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

Deadlock: Starvation vs Deadlock



- **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 software information (a variable, file, semaphore, record of a database).
- Resources comes in two flavors:
  1. **Preemptable Resources:** Can be taken away from a process without harm.
     - Example: CPU (scheduler can switch a process out and resume later), memory (can be swapped to disk and reloaded).
     - These rarely cause deadlocks because the system can reassign them safely.
  2. **Non-Preemptable Resources:** Cannot be forcibly taken away without breaking the process.
     - Example: Printer (if taken mid-job → garbled print out), CD Burner (if stopped mid-burn, corrupted disc).
     - 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 the following four conditions hold simultaneously (Necessary Conditions)

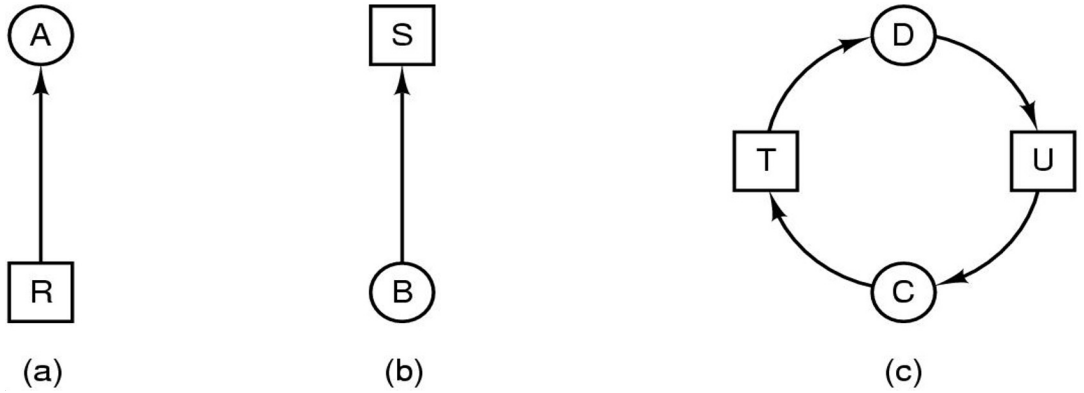
1. **Mutual exclusion**: Only one process can use a resource at any given time i.e. the resources are non-sharable. Only one process can use the printer at a time.
2. **Hold and wait**: A process is holding at least one resource and waiting for additional resources. Process P1 is holding the scanner and waiting for the printer, meanwhile, Process P2 is holding the printer and waiting for the scanner
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

Resource Allocation Graph

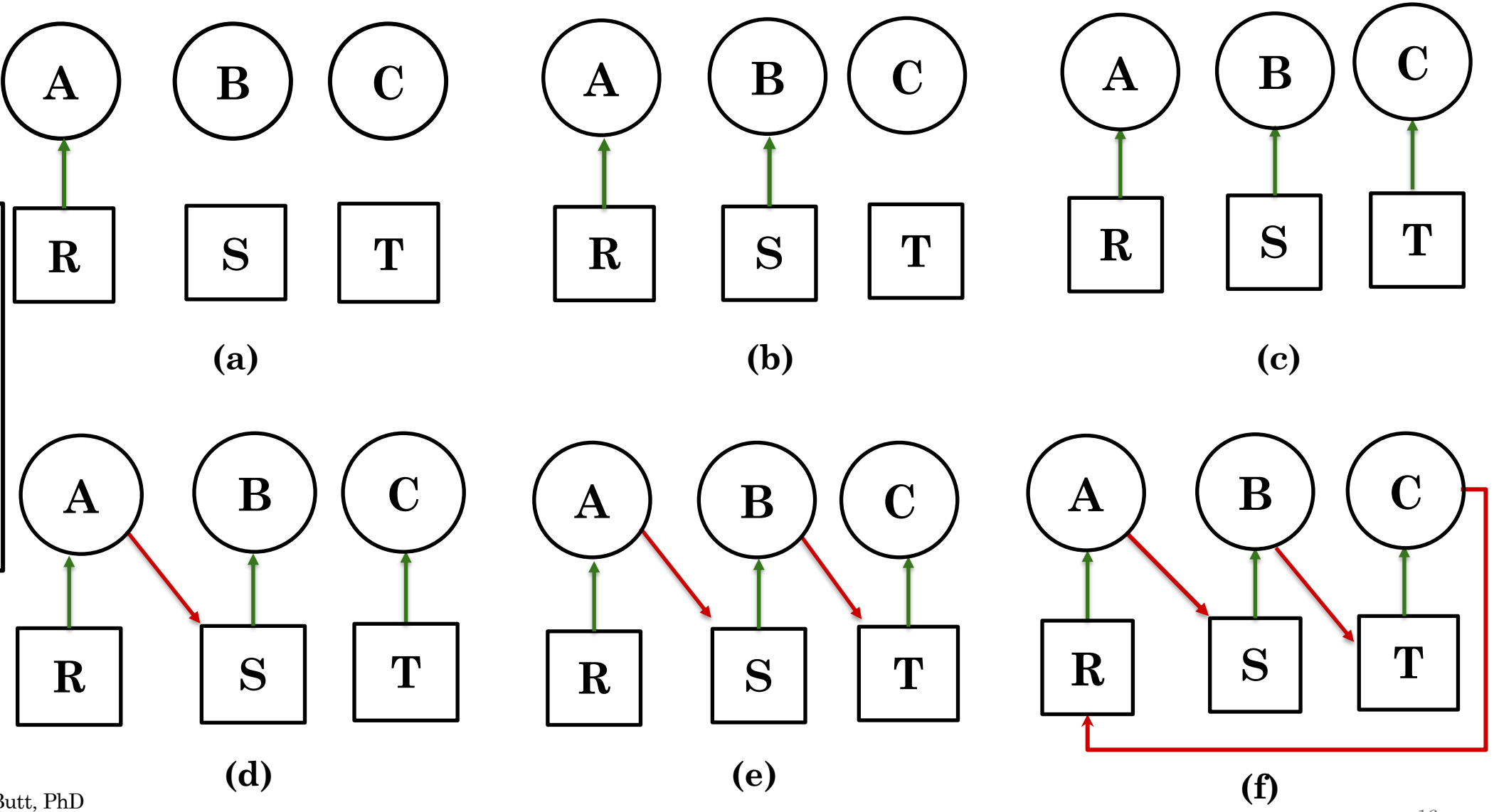
- A Resource Allocation Graph (RAG) is a graphical tool used in operating systems to represent how processes and resources interact. Instead of relying only on tables, RAG provides a visual way to see:
  - Which resources are allocated to which processes?
  - Which processes are waiting for which resources?
  - How many resources are available?
- Making deadlock conditions easy to spot
- Elements of RAG:



1. **Vertices (nodes):**
   - Processes (P_1, P_2, …, P_n) → drawn as circles
   - Resources (R_1, R_2, …, R_m) → drawn as squares (each square represent multiple instances)
2. **Edges (arrows):**
   - Request Edge (Pi → Rj): → Arrow goes from a process to a resource
     - Process Pi is waiting for a resource of type Rj
   - Assignment Edge (Rj → Pi): → Arrow goes from a resource to a process
     - An instance of resource Rj is assigned to process Pi

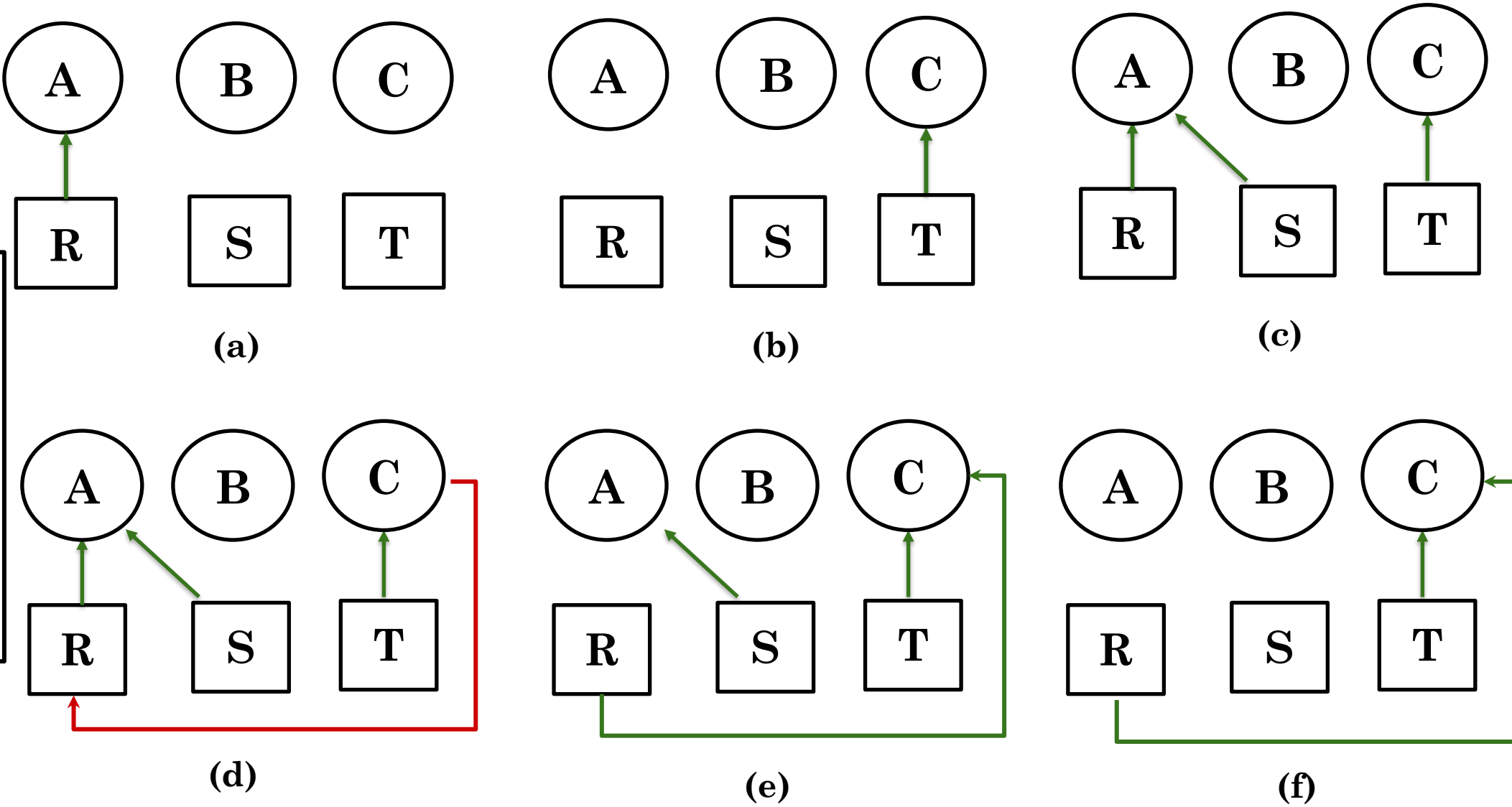
Resource Allocation Graph

a. A requests R
b. B requests S
c. C requests T
d. A requests S
e. B requests T
f. C requests R



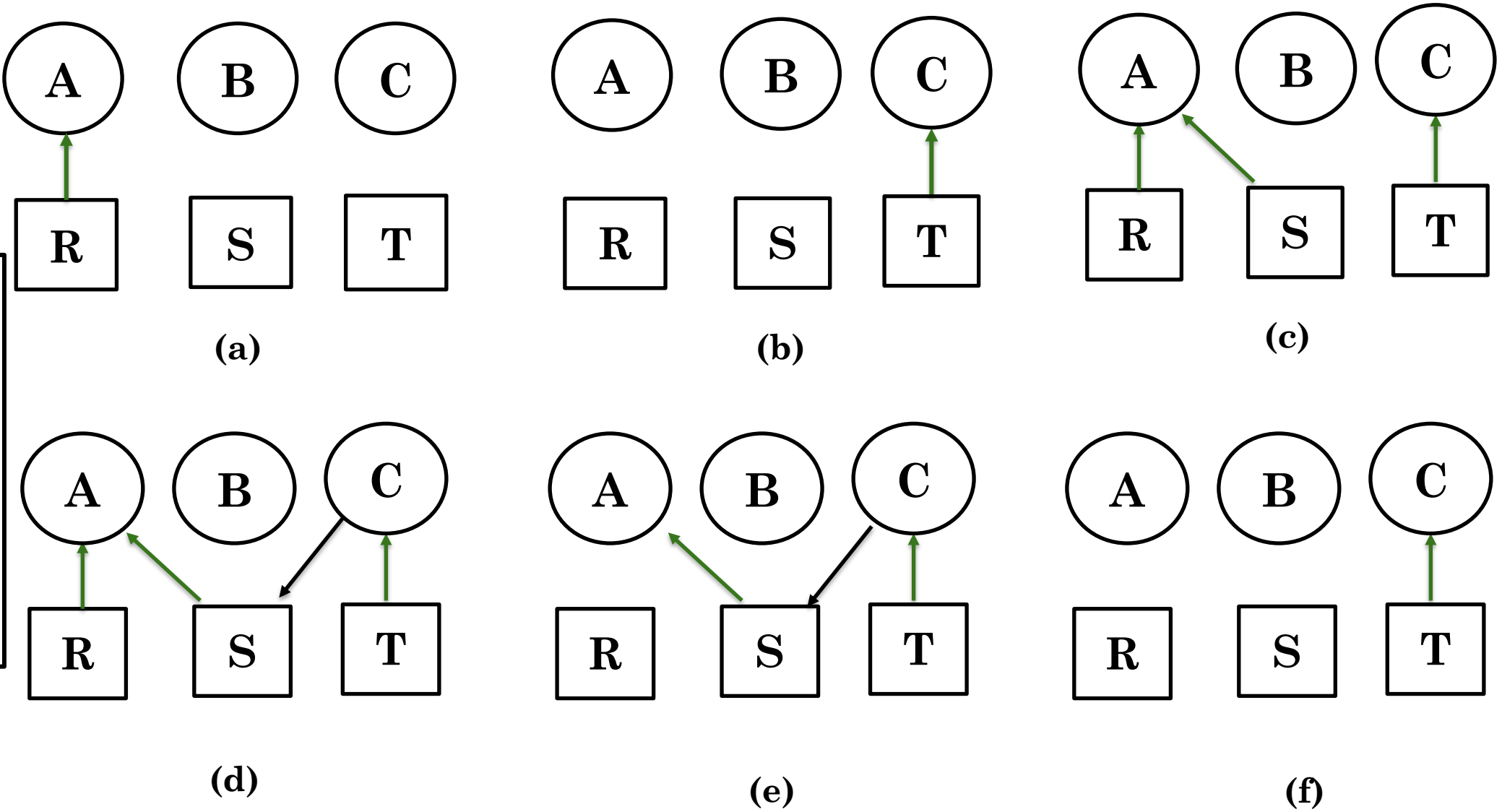
Resource Allocation Graph

a. A requests R
b. C requests T
c. A requests S
d. C requests R
e. A releases R
f. A releases S

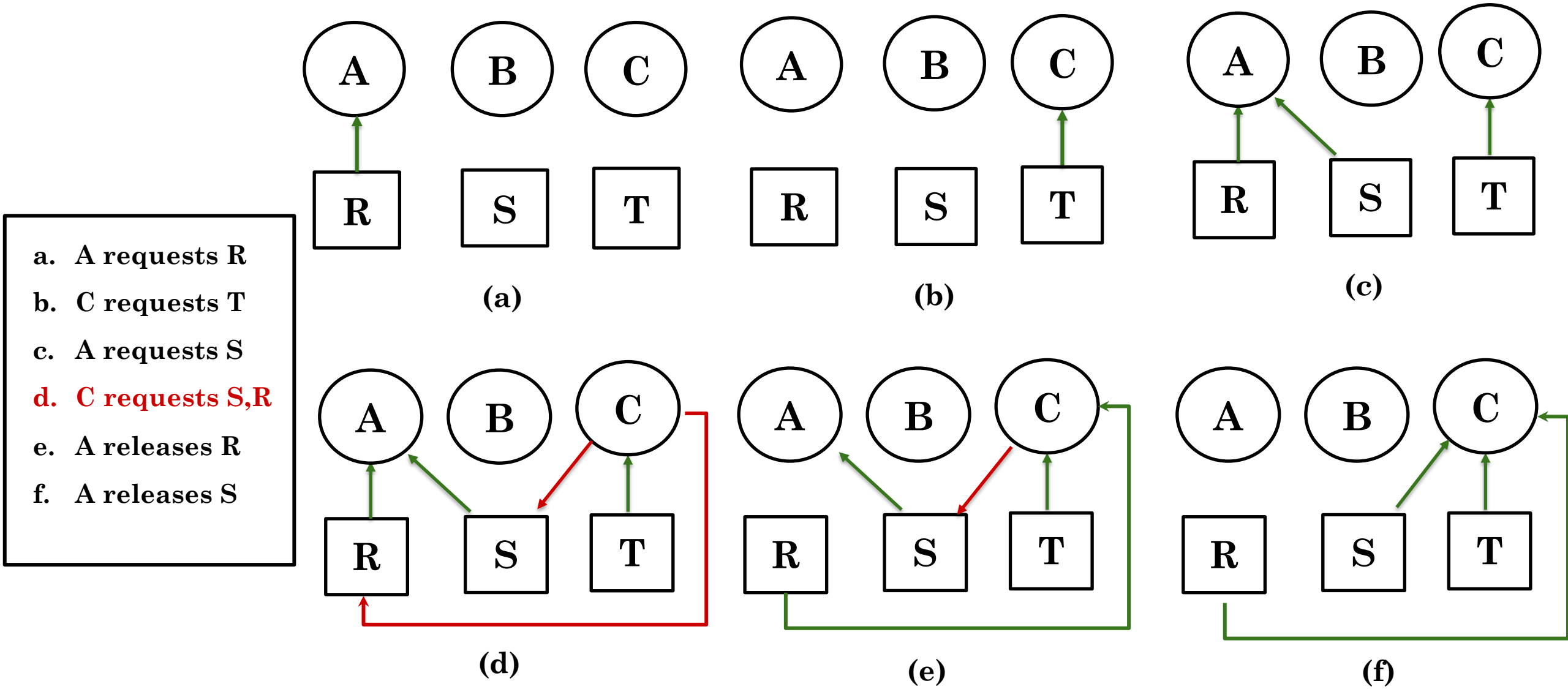


Resource Allocation Graph

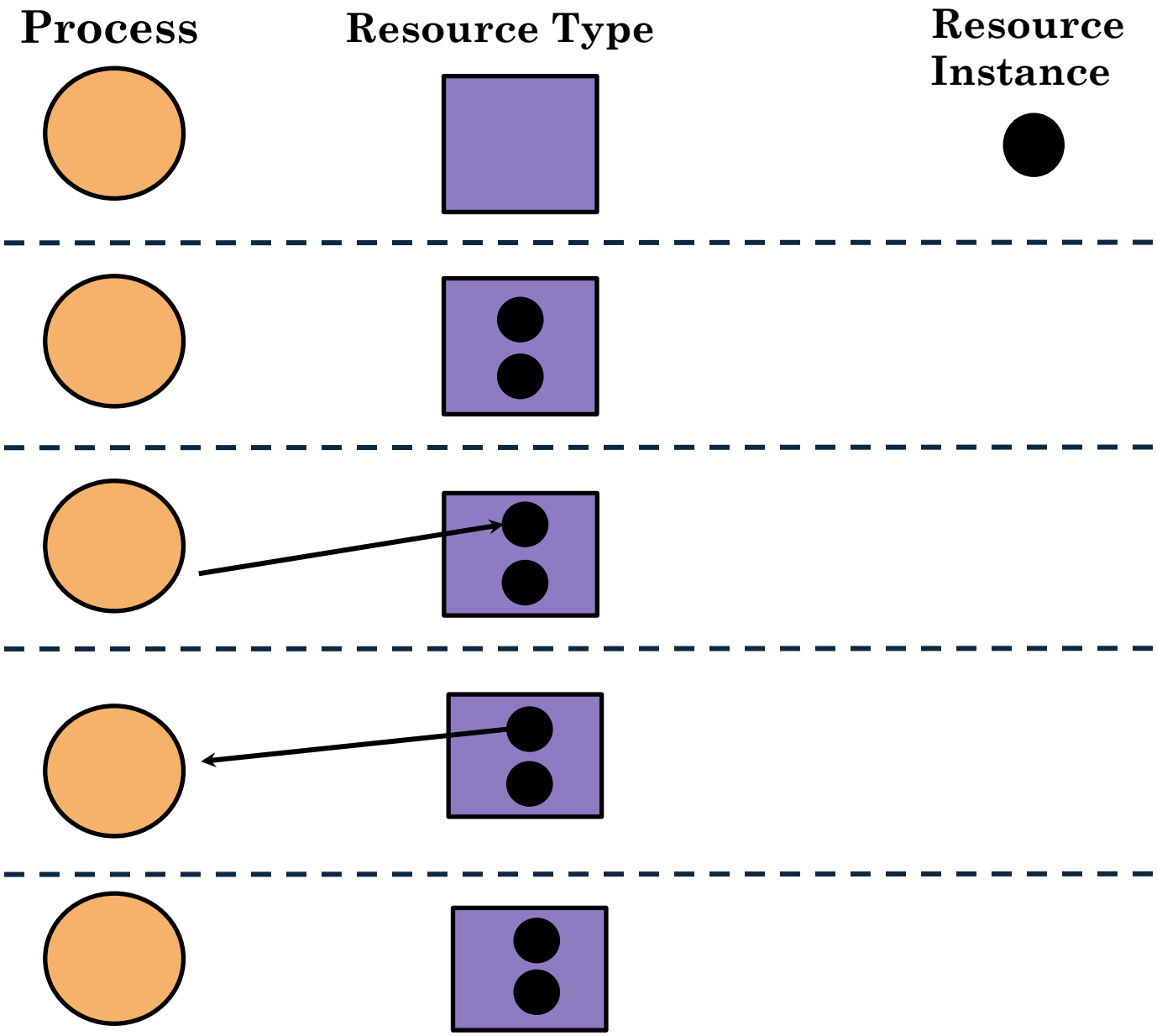
a. A requests R
b. C requests T
c. A requests S
d. C requests S
e. A releases R
f. A releases S



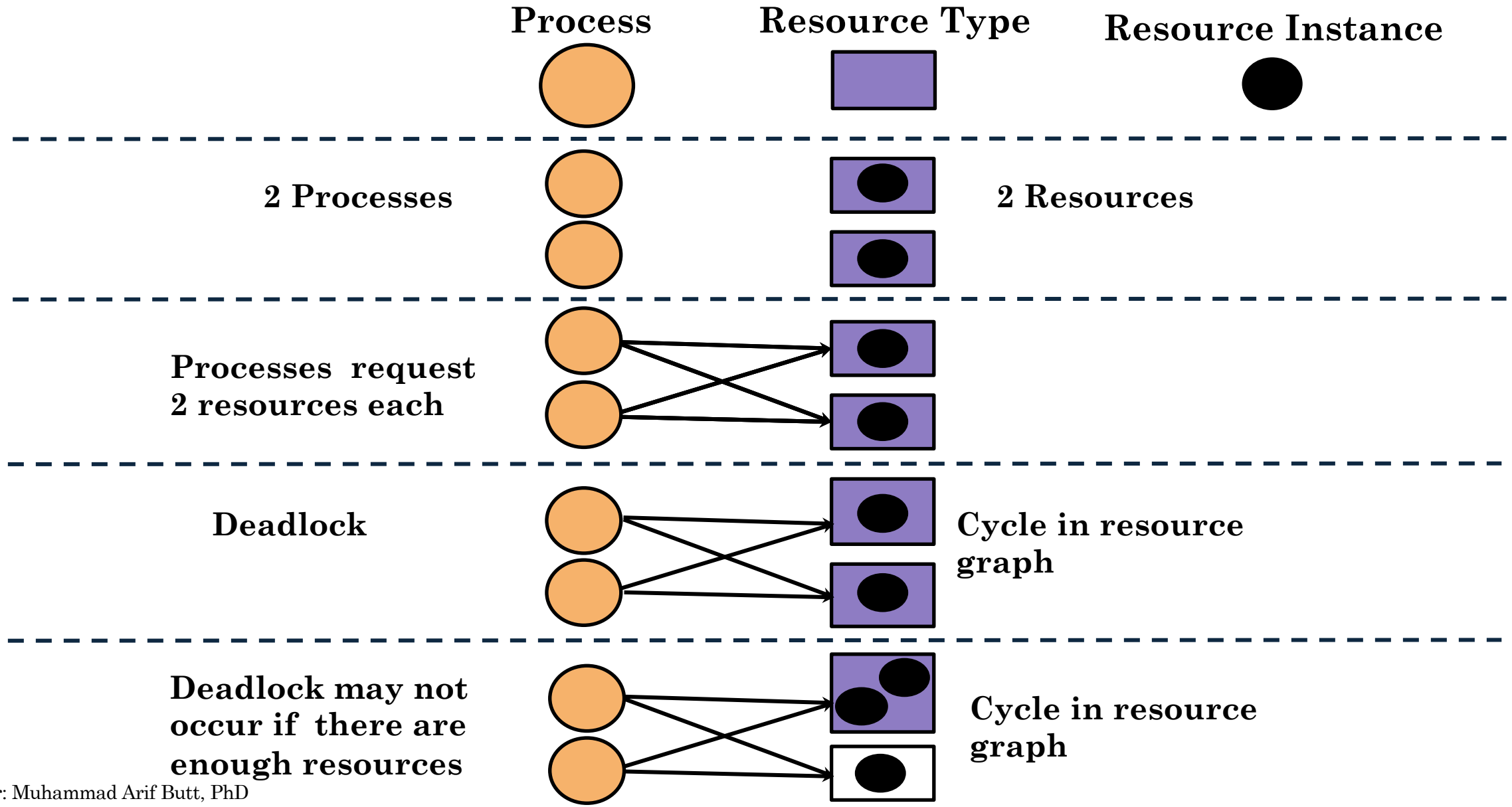
Resource Allocation Graph



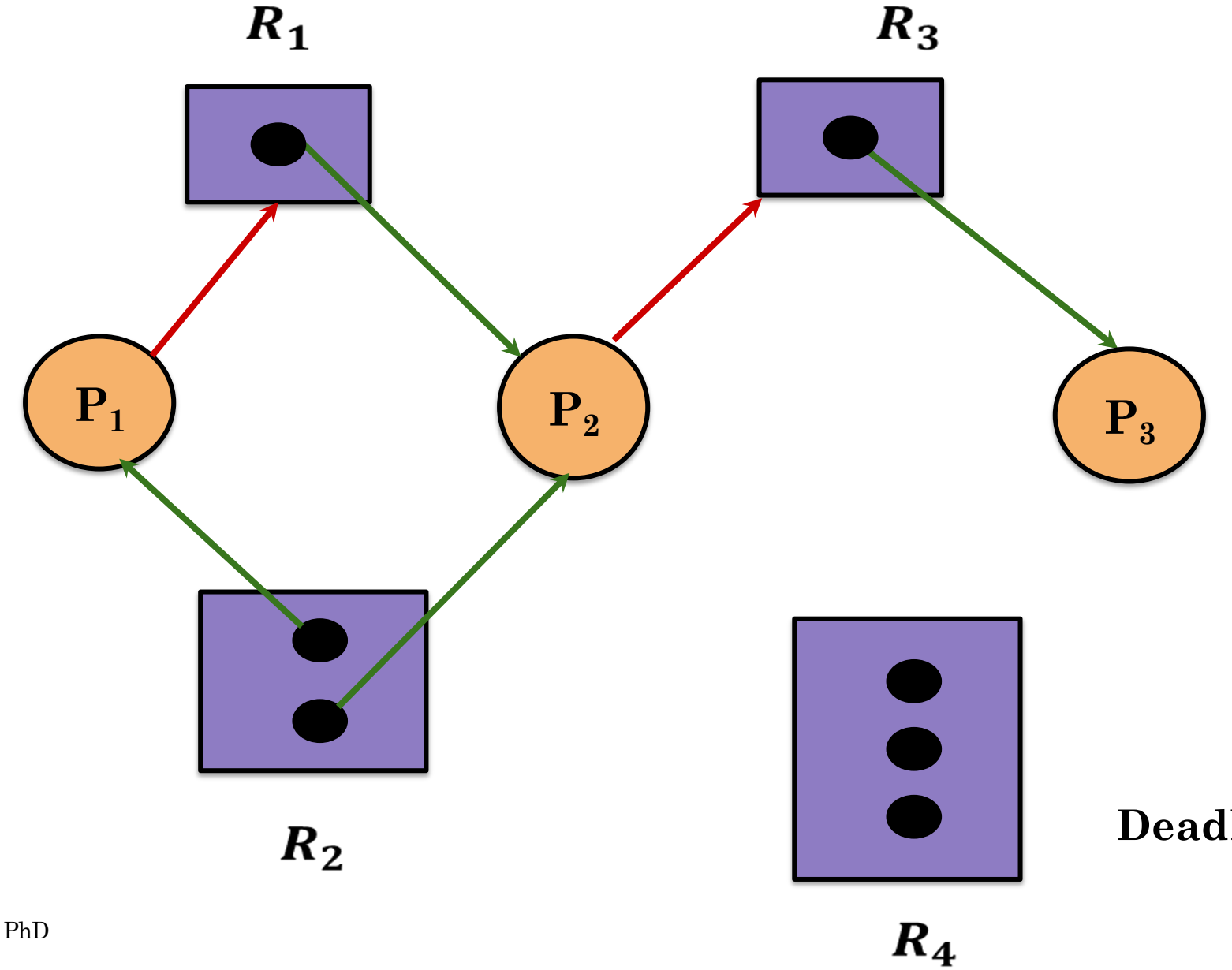
Multiple Instances of a Resource Type



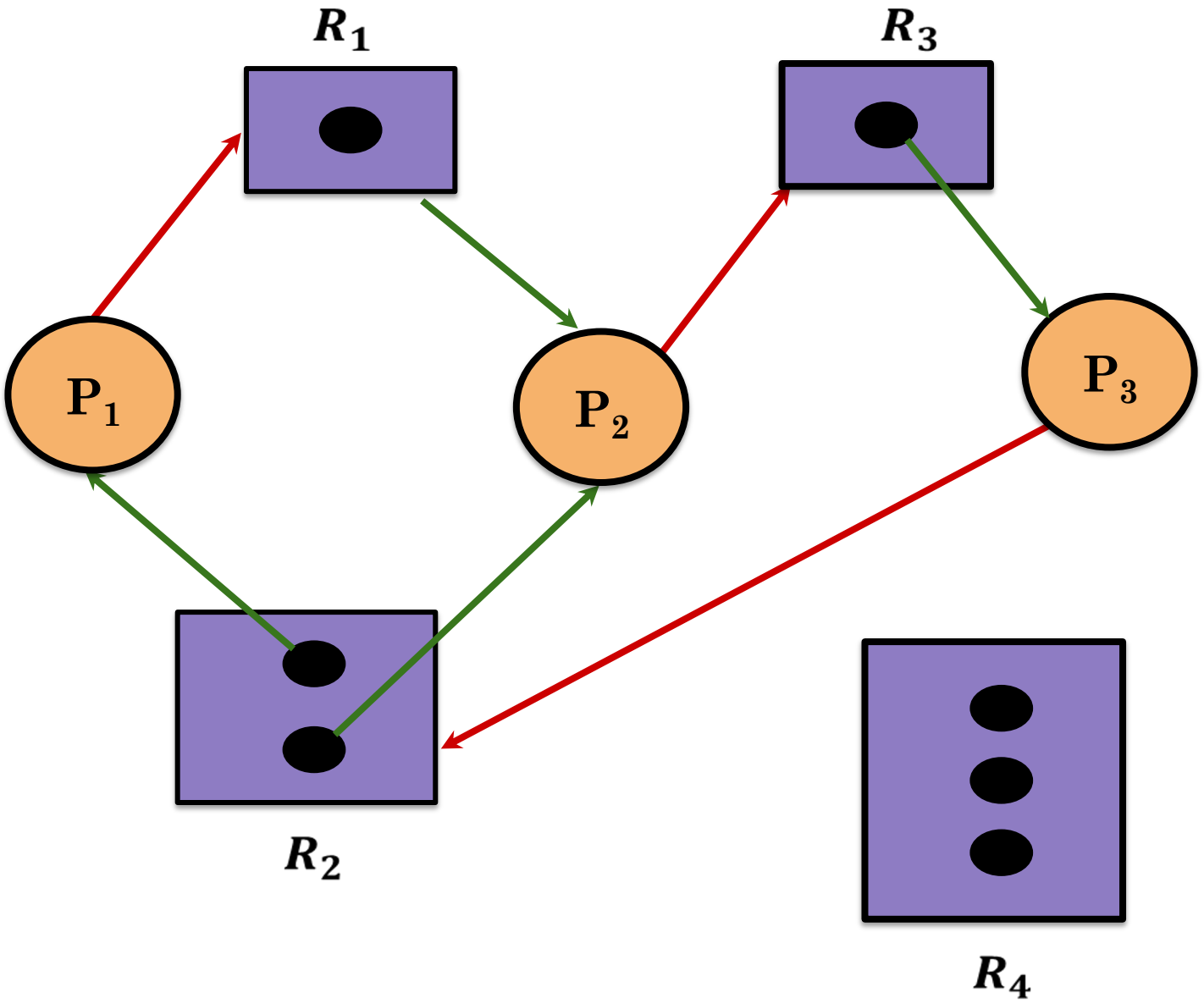
Multiple Instances of a Resource Type



RAG Example 1



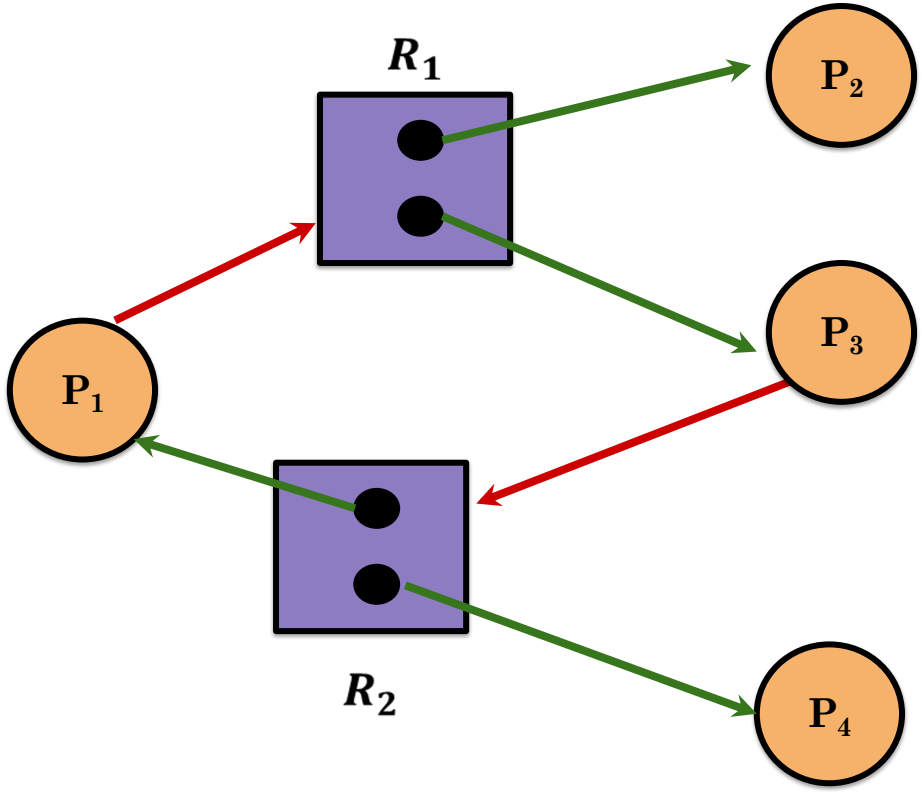
RAG Example 2



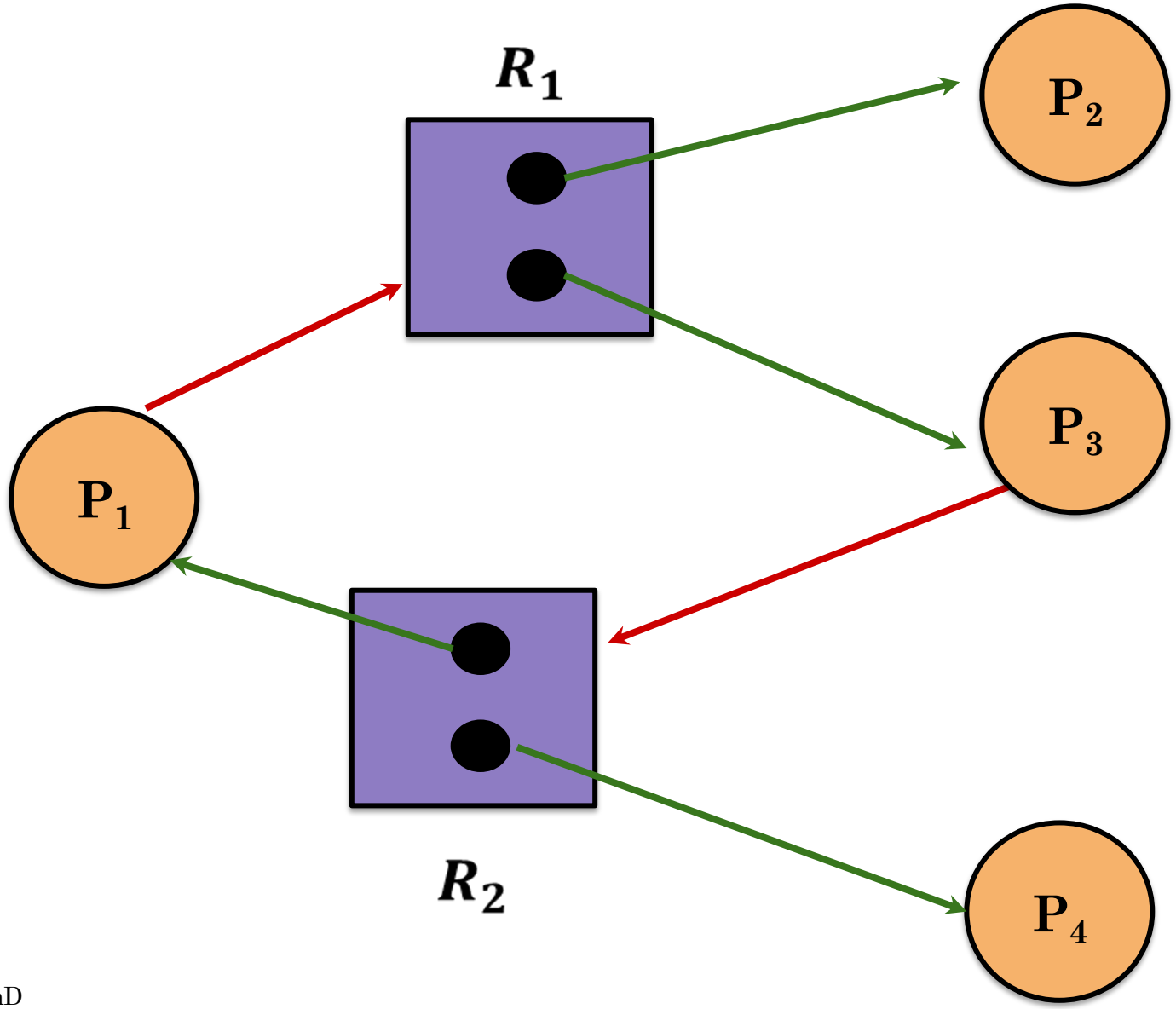
Deadlock?

Basic Facts

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - If only one instance per resource type, then deadlock
  - If several instances per resource type, **possibility** of deadlock



RAG - Cycle and no 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

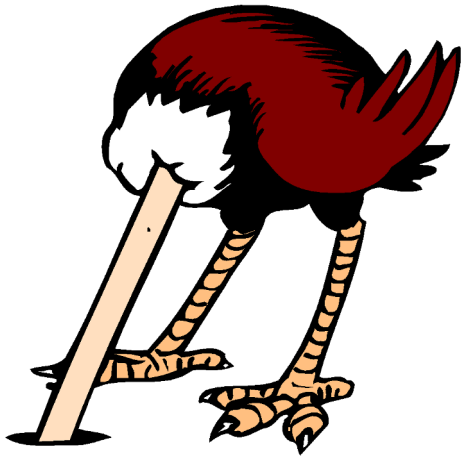
A system has five processes P1 through P5 and four resource types R1 through R4. It has 2 units of each resource type. Given that:

- 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

1. **Prevention:** Design system so that deadlock is impossible. It involves adopting a static policy that disallows one of the four conditions for deadlock.
2. **Avoidance:** Steer around deadlock with smart scheduling. It involves making dynamic choices that guarantee prevention.
3. **Detection & recovery:** Check for deadlock periodically. Recover by killing a deadlocked processes and releasing its resources.
4. **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.




Deadlock
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: At least one resource must be held in a non-shareable mode (only one process can use it at a time)
- Hold and Wait: A process must be holding at least one resource and waiting to acquire additional resources currently held by other processes.
- No Preemption: Resources cannot be preempted (forcibly removed from a process); they can only be released voluntarily by the process holding them
- Circular Wait: A set of processes $\{P_0, P_1, ..., P_n\}$ exists such that P_0 waits for P_1, P_1 waits for P_2, ..., P_n waits for P_0

When all of the above four conditions are present simultaneously:

- Processes cannot share resources (mutual exclusion)
- Processes are stuck holding some resources while waiting for others (hold and wait)
- No external intervention can break the impasse (no preemption)
- The waiting forms a closed loop with no exit (circular wait)

Result: The system reaches a state where no process can proceed, which is the definition of deadlock

Deny Hold and Wait

Prevent processes from holding resources while waiting for others

- **Option 1 - Request All upfront**
  - 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.

```
Process lifecycle:
1. Request ALL needed resources at start
2. Execute using resources
3. Release ALL resources at end
```

- **Option 2 - Release All before request**
  - 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.

```
Process holds R1, needs R2:
1. Release R1
2. Request both R1 and R2 together
3. If successful, continue; otherwise retry
```

Preemption based Prevention

Allow the system to forcibly take resources from processes when needed

- 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.
- This requires that the resources must be preemptible (like CPU, memory pages), not applicable to printers, tape drives, etc.

```
P1 holds R1, requests R2
P2 holds R2, requests R1

Preemption solution:
1. Preempt R2 from P2
2. Give R2 to P1
3. P1 completes and releases R1, R2
4. P2 can now get both resources
```

Resource Ordering (Global Numbering)

This technique assigns unique numbers to all resources and requires processes to request resources only in strictly increasing order

- Assign unique numbers: $R_1(1)$, $R_2(2)$, $R_3(3)$, ..., $R_n(n)$
- Rule: Process can request R_j only if it doesn't hold any R_i where $i \geq j$

- **Example**

```
Resources: Printer(1), Scanner(2), Disk(3), Memory(4)

Valid sequence:
P1: Request Printer(1) → Request Scanner(2) → Request Memory(4) ✓

Invalid sequence:
P1: Hold Memory(4) → Request Printer(1) ✗ (4 > 1)

Why cycles are impossible:
If P1 holds R(i) and wants R(j) where i < j, and P2 holds R(j) and wants R(i), then
      P2 violates the ordering rule, since j > i, preventing the cycle.
```

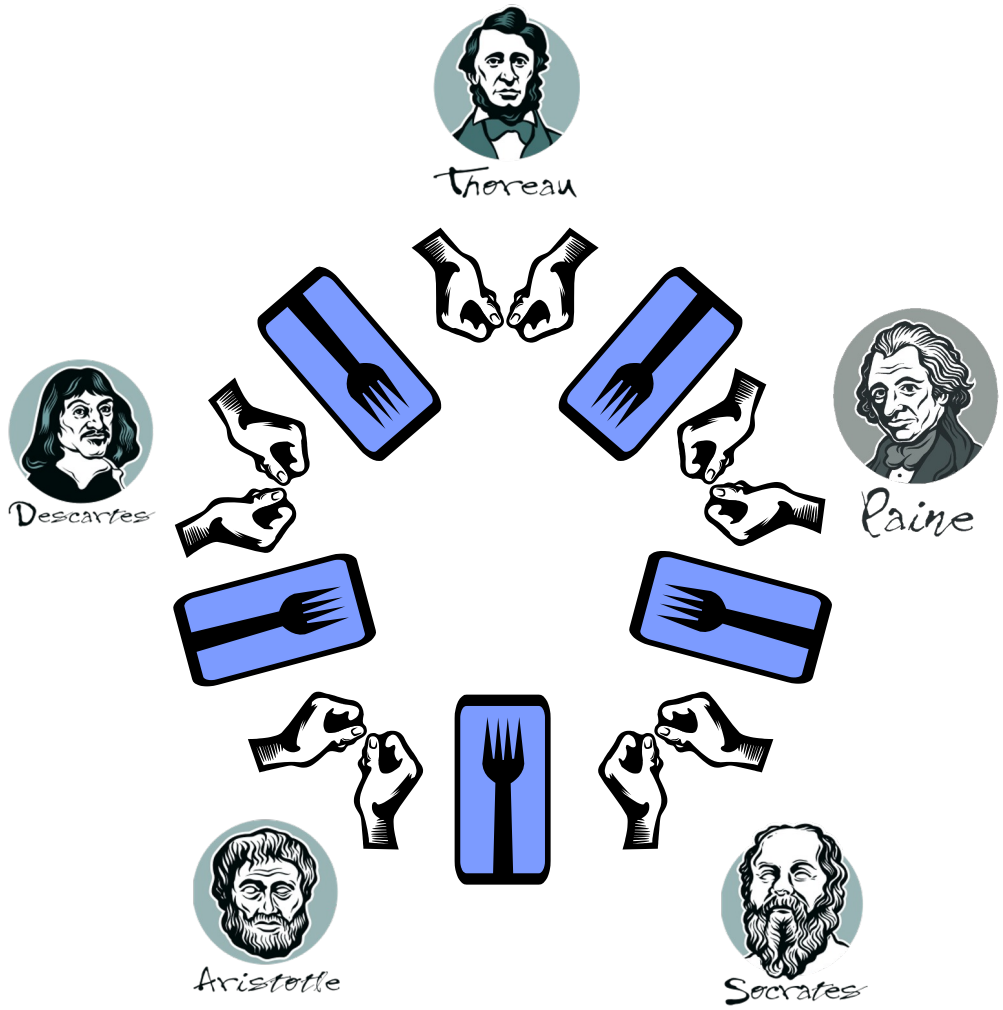
Sample Problem

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.

Dining Philosopher - Unnumbered resources

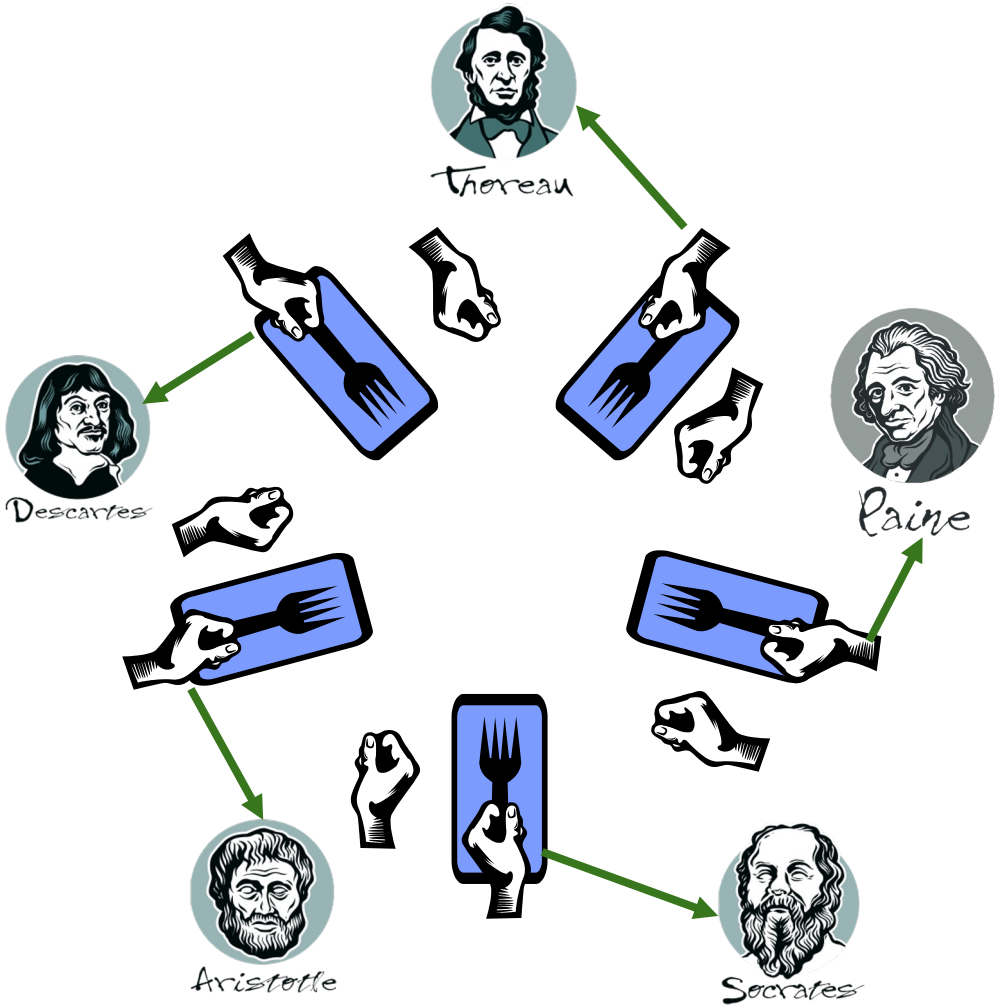
```
# define N 5

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 Philosopher - Unnumbered resources

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

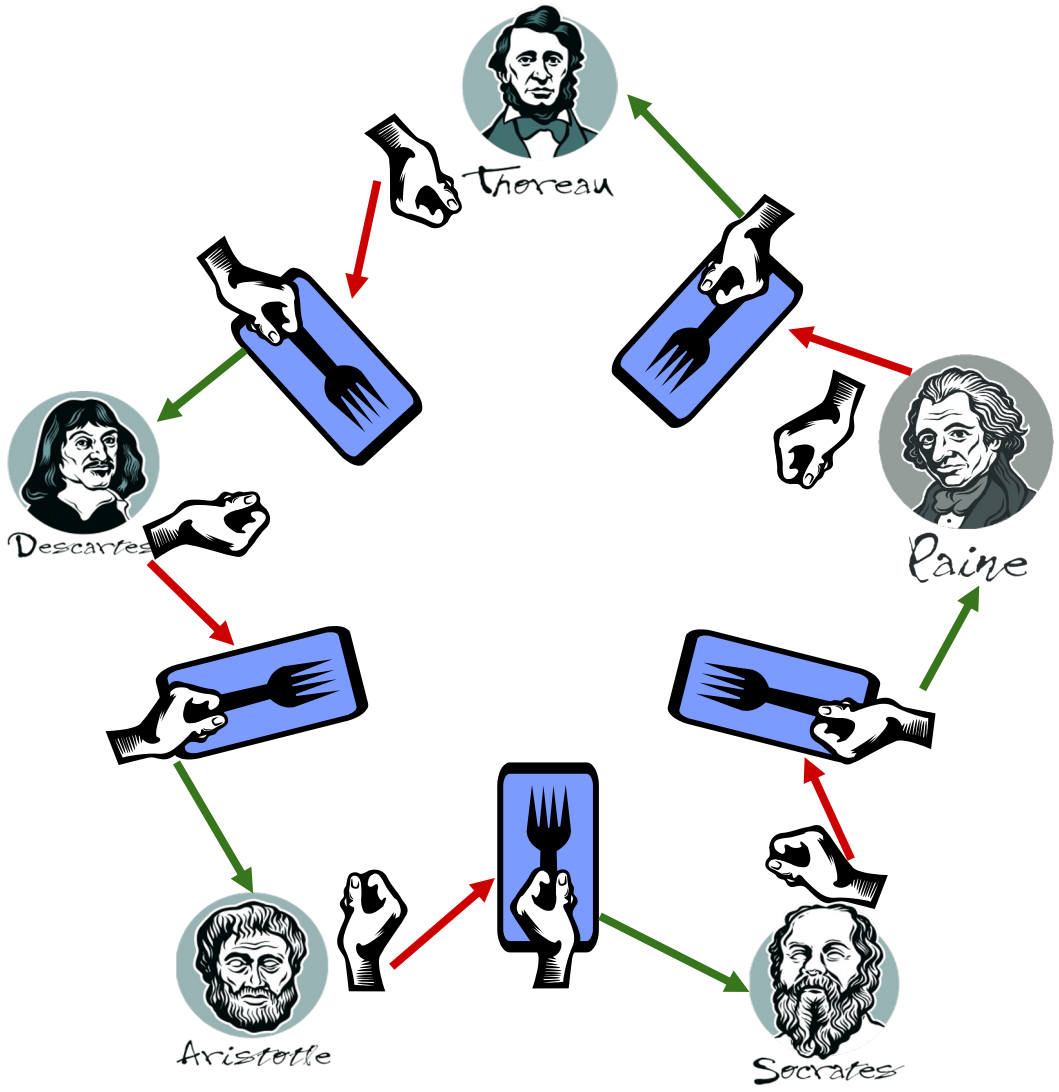
```
# define N 5

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 Philosopher - Unnumbered resources

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

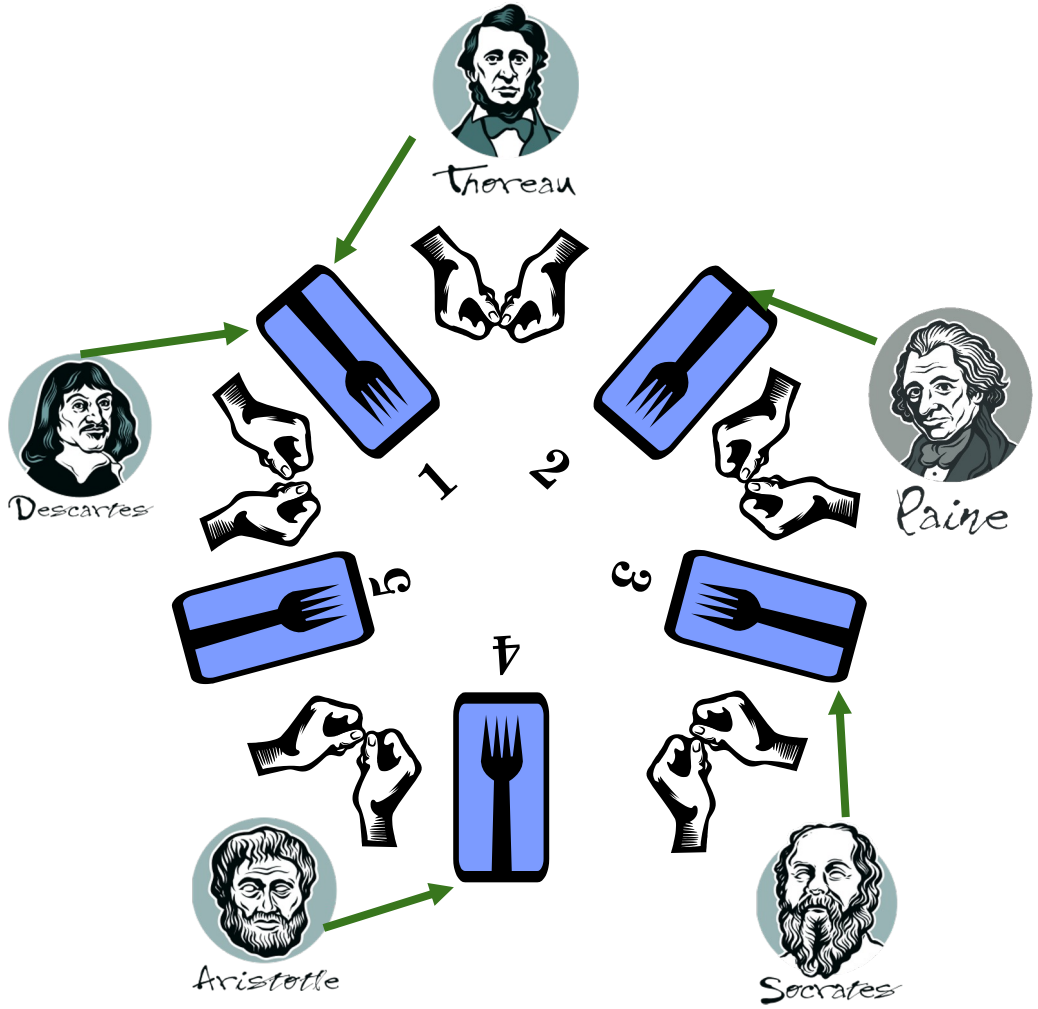
```
# define N 5

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 Philosopher Solution (Global Numbering)

Each philosopher first request lower numbered fork. Note the fork#1 is requested 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));
  }
}
```

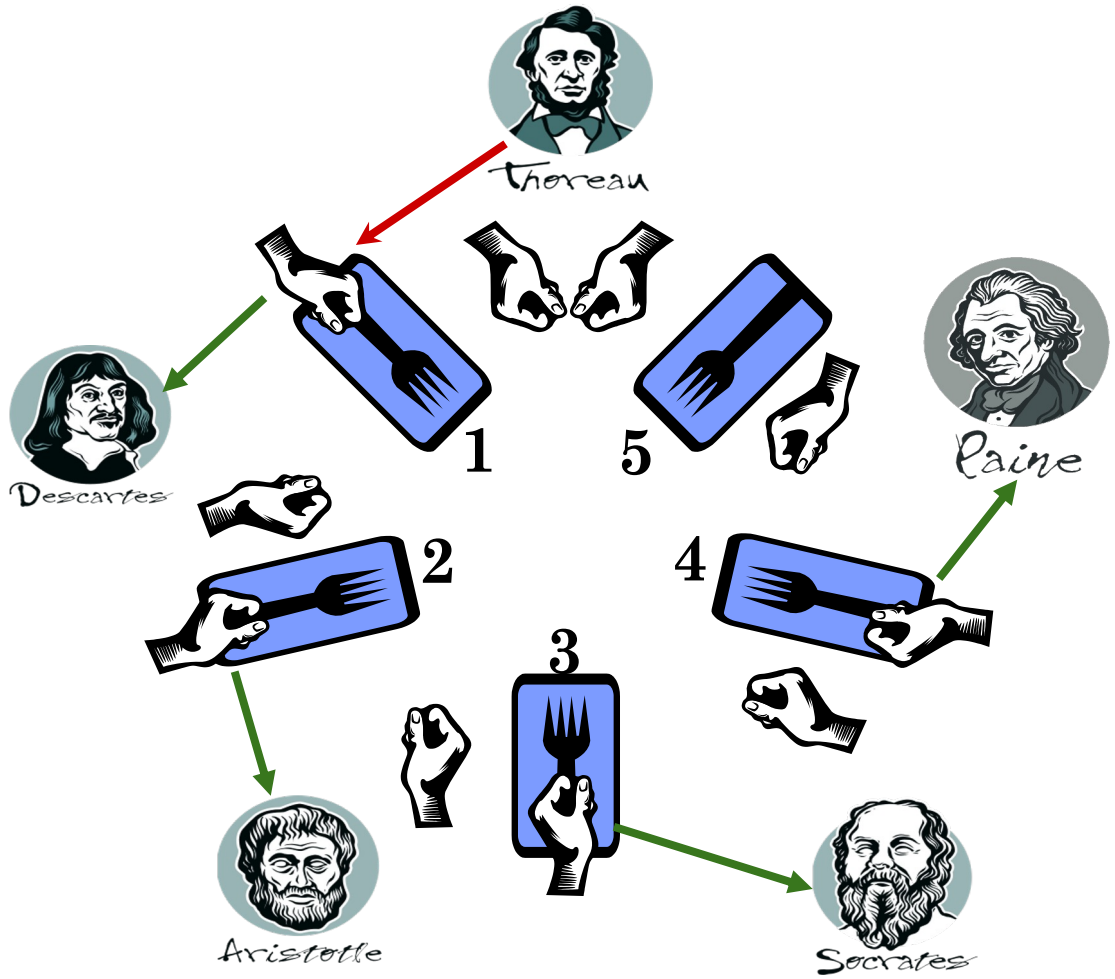


Fork#1 is requested for by two philosophers

Dining Philosopher Solution (Global Numbering)

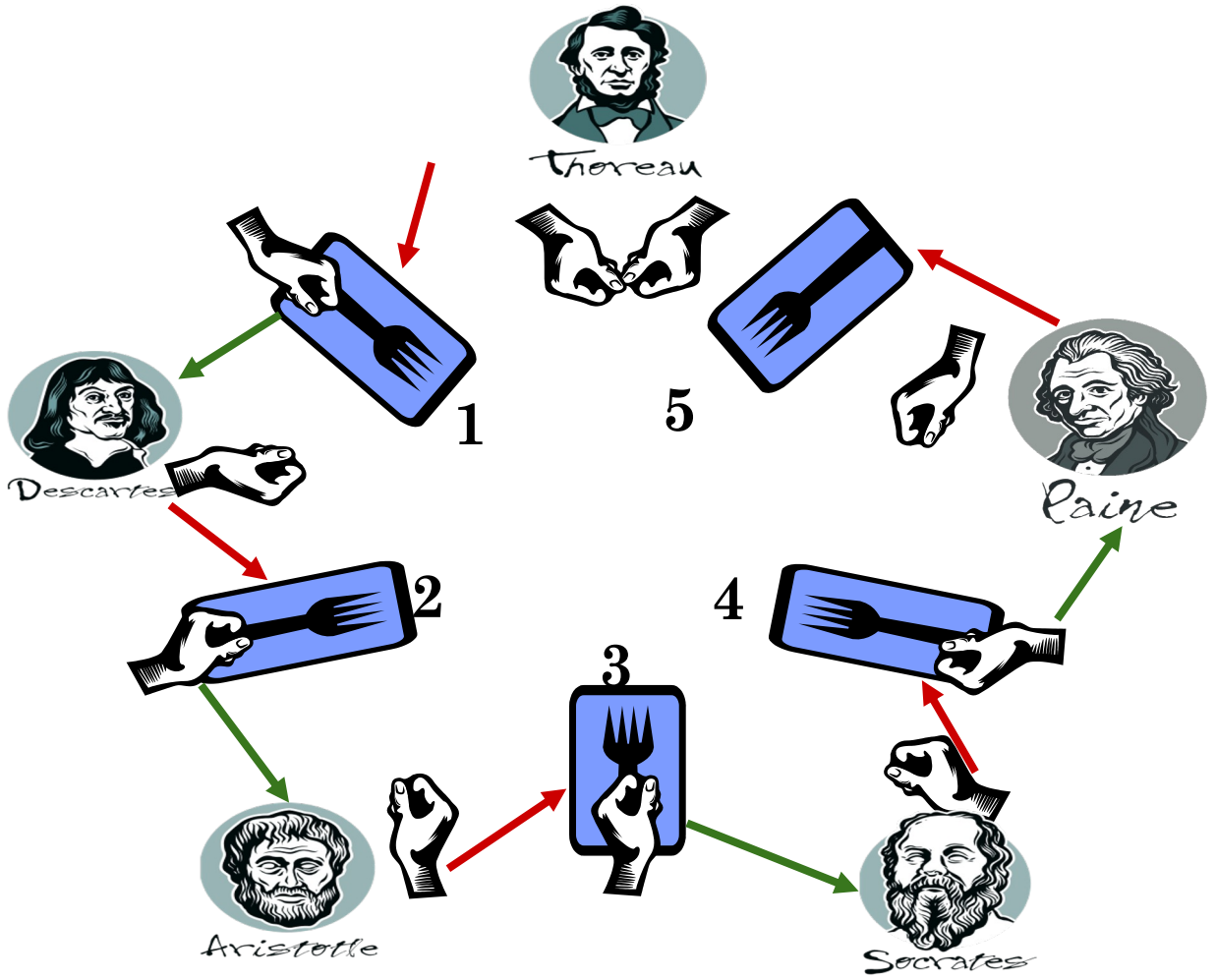
One of the philosophers doesn't get the fork.

```
# 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 Philosopher Solution (Global Numbering)

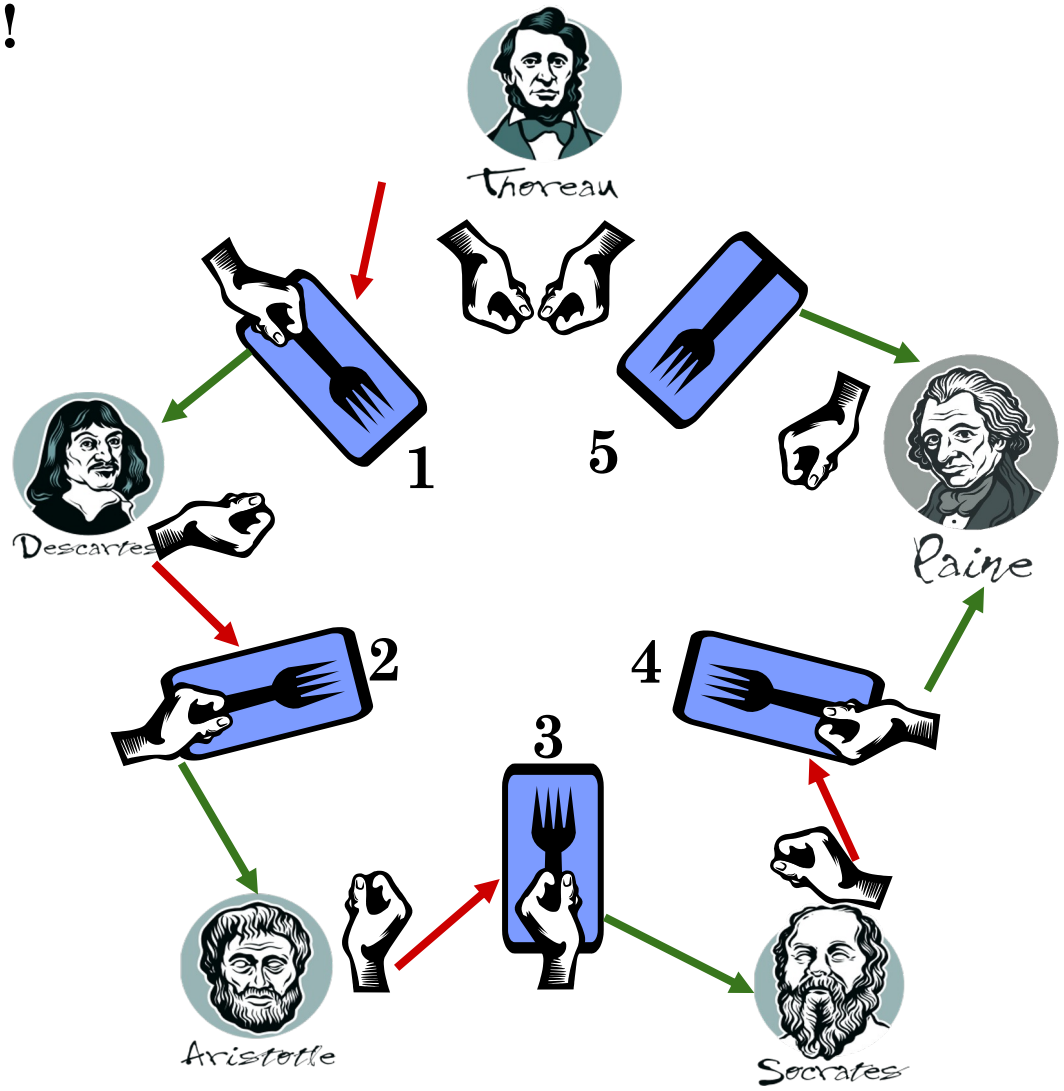
Philosophers holding one resource then, request higher numbered fork



Dining Philosopher Solution (Global Numbering)

One philosopher will always succeed to 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));
  }
}
```



Deadlock AVOIDANCE

Overview of Deadlock Avoidance

- Unlike prevention, avoidance allows all four Coffman conditions to exist in principle but uses dynamic analysis to ensure the system never actually enters a deadlocked state.
- Deadlock avoidance ensures that a system remains in a safe state by carefully allocating resources to processes while avoiding unsafe states that could lead to deadlocks

Key Distinction between Prevention and Avoidance:

- Deadlock Prevention: Structurally eliminates one or more of the four necessary conditions.
- Deadlock Avoidance: Analyzes the current state of the system and determines if granting a resource request will result in a safe state.

Safe vs Unsafe States:

- Safe State: A state is considered safe if there exists a sequence of resource allocations that permits all processes to complete without creating a deadlock.
- Unsafe State: No such sequence exists, making deadlock possible.

Avoidance Strategies (Process Initiation Denial)

Do not start a process if its demands might lead to deadlock

Before starting process P_i,
check: Can system satisfy P_i's maximum needs + all currently running processes' maximum needs?

If NO → Deny process initiation

If YES → Allow process to start

Characteristics:

- Very conservative and inefficient
- Underutilizes system resources
- Simple to implement

Avoidance Strategies (Resource Allocation Denial)

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

Bankers & Safety Algorithm:

Each process within the system must provide all the important necessary details to the operating system, and the system tracks the resource allocation state:

- Total resources in the system
- Resources already allocated
- Maximum demands still possible

Before granting a request, the system checks, "If I grant this, is there some way (a safe sequence) for all processes to finish?"

- If yes → grant
- If no → block the request until it is safe

```
System: 3 processes, 3 resource types
Available: [3, 3, 2]

Process | Allocation| Max Need | Remaining Need
P0      | [0,1,0]   | [7,5,3]  | [7,4,3]
P1      | [2,0,0]   | [3,2,2]  | [1,2,2]
P2      | [3,0,2]   | [9,0,2]  | [6,0,0]

Request: P1 wants [1,0,2]

Safety Check:
1. Grant request: Available becomes [2,3,0]
2. Find safe sequence: P1 → P0 → P2 or P1 → P2 → P0
3. Since safe sequence exists → Grant request
```

Important Terms

- **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 deadlock
  - When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
  - If a system is in safe state , deadlock cannot occur
- **Safe Sequence** is the sequence of execution of processes in which OS fulfills requests of all the processes and still avoids deadlock
- **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
- Deadlock 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 below. Is the system in safe state, if yes give a safe sequence?

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

Sample Problem

System with 12 tape drives and three processes. Current system state is as shown below. Is the system in safe state, if yes give a safe sequence? If not explain which process may cause a deadlock.

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

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

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.

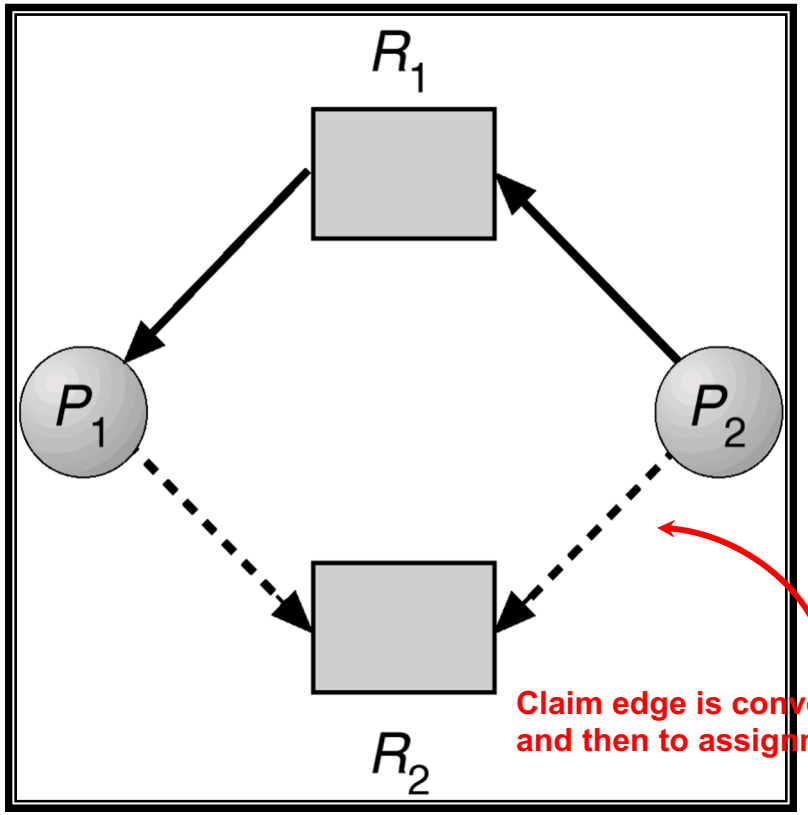
Problem 3

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?

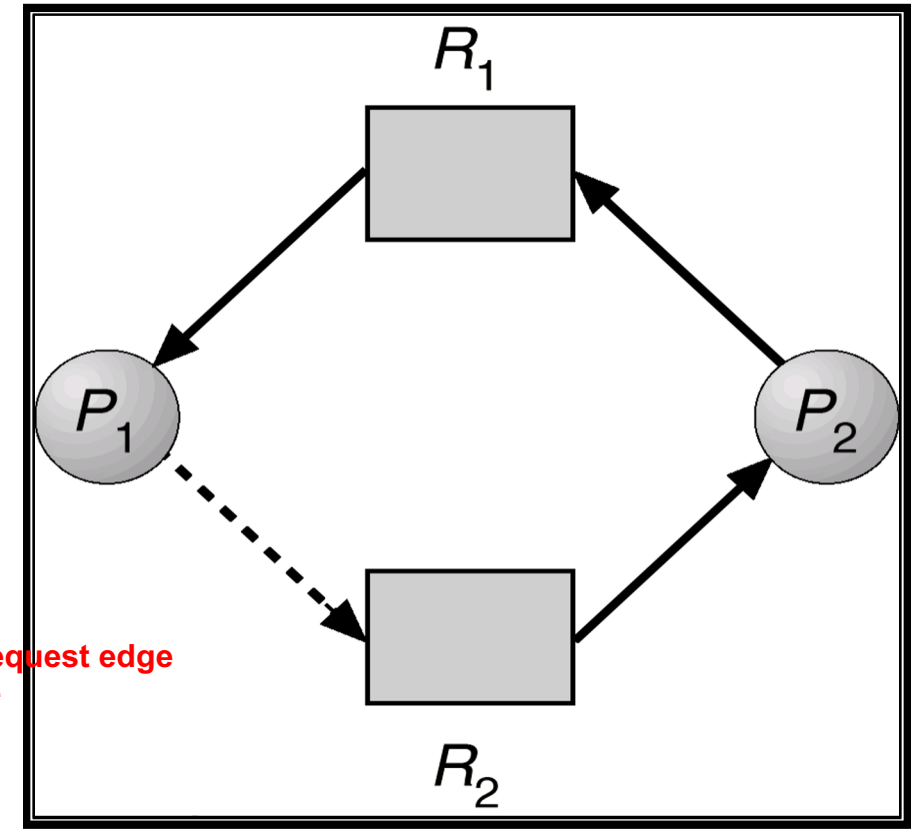
RAG Algorithm for Single Instance Resources

- For systems where each resource type has only one instance, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with claim edges, noted by dashed lines, which point from a process to a resource that it may request in the future.
- **Claim edge** Pi → 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 for Single Instance Resources



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.

51

RAG Algorithm for Single Instance Resources

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:

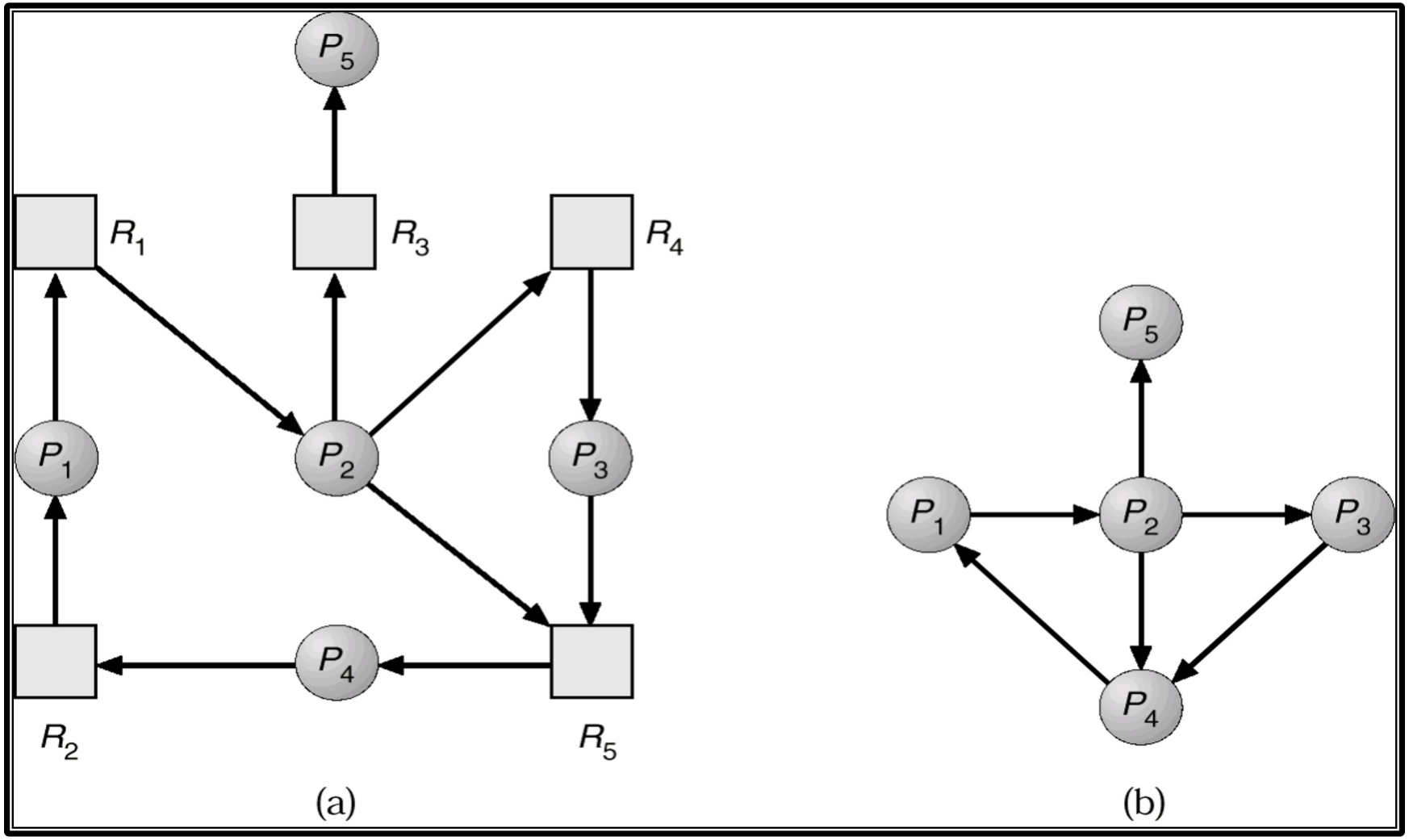
- Choose a process node as a root node to initiate a depth first traversal.
- Traverse the digraph in depth first mode.
- Mark process nodes as we traverse the graph.
- If a marked node is revisited then a deadlock exist.

RAG and Wait for Graph

- The RAG algorithm is expensive as it requires an order of n^2 operations, where n is the number of vertices in the graph.
- To improve the performance, we can use wait-for-graph instead of RAG.
- To derive a wait-for-graph from a RAG, you simply collapse the resource nodes

  $P_i \rightarrow P_j$ means P_i is waiting for P_j
- 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. If a cycle exist in the wait-for-graph we say that a deadlock exist

RAG and Wait for Graph



Resource-Allocation Graph

Corresponding wait-for graph

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

- $Need_i = Max_i$ – Allocation
- 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

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

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

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

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

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

- Final safe sequence:

  <P1, P3, P4, P0, P2>

- Not a unique sequence
- Possible safe sequences for this example:
  1. <P1,P3,P4,P0,P2>
  2. <P1,P3,P4,P2,P0>
  3. <P1,P3,P2,P0,P4>
  4. <P1,P3,P2,P4,P0>
  5. <P1,P3,P0,P2,P4>
  6. <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 the system:

	Allocated	Maximum	Available
Process 0	0 0 1 2	0 0 1 2	1 5 2 0
Process 1	1 0 0 0	1 7 5 0	
Process 2	1 3 5 4	2 3 5 6	
Process 3	0 6 3 2	0 6 5 2	
Process 4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

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

Deadlock

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

Deadlock Detection Algorithms:

1. Construct RAG and run cycle detection algorithm.
2. Construct wait-for-graph from RAG and run cycle detection algorithm.
3. Use Banker's algorithm to detect unsafe state

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 Techniques

Recovery algorithms vary a lot in their severity:

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

Abort Deadlock 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?

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

To do

Coming to office hours does NOT mean that you are academically weak!