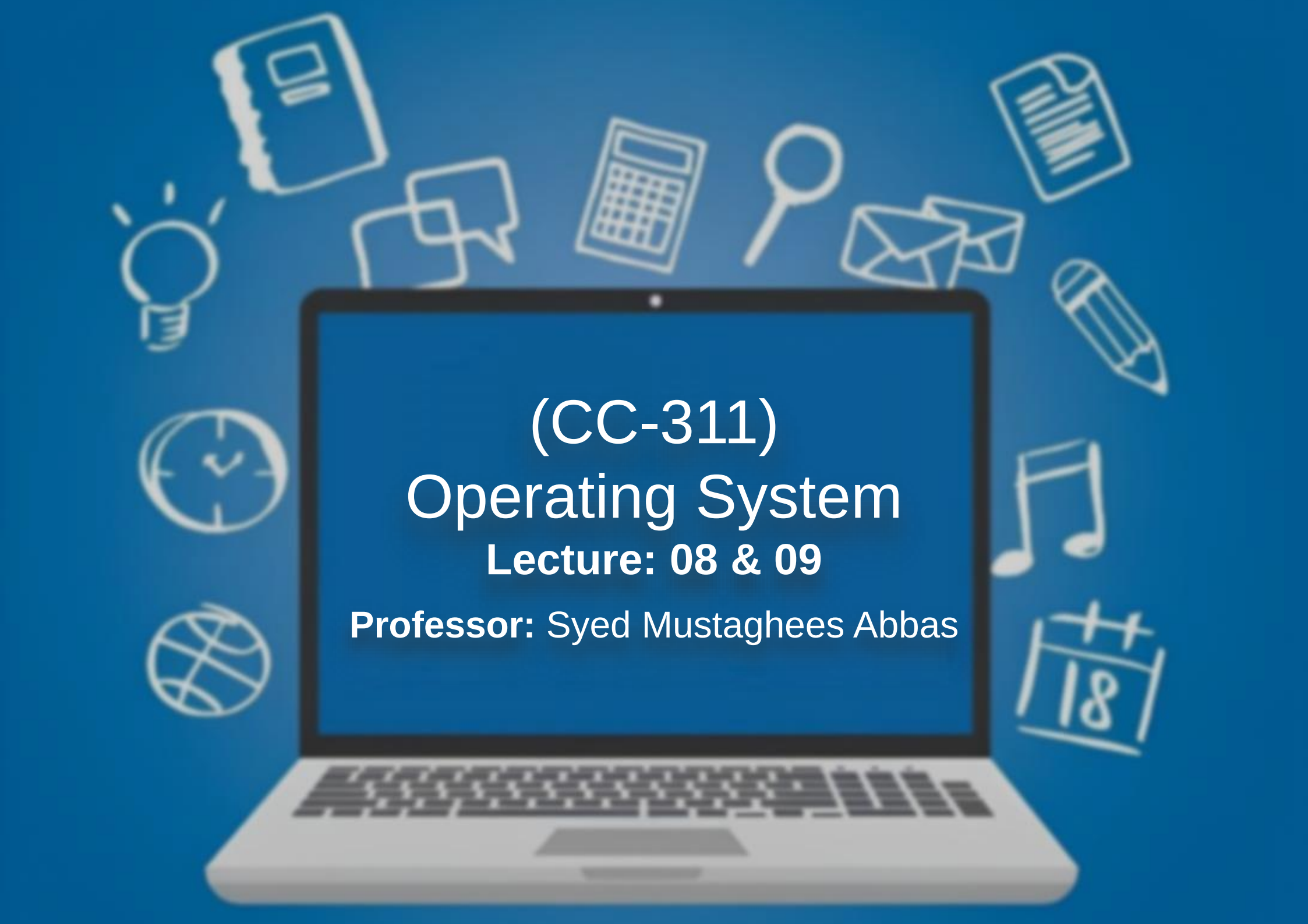

(CC-311)
Operating System
Lecture: 08 & 09
Professor: Syed Mustaghees Abbas

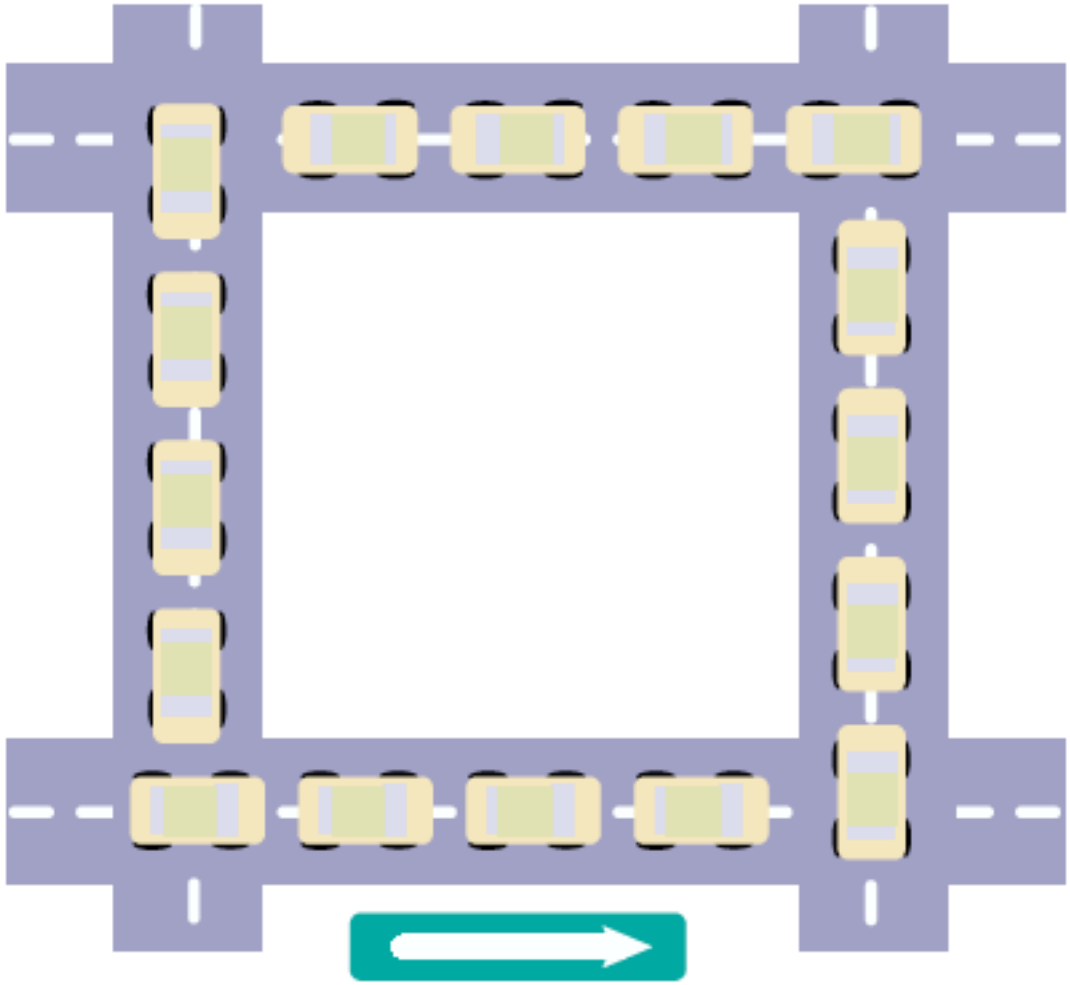
Deadlocks

The Deadlock Problem

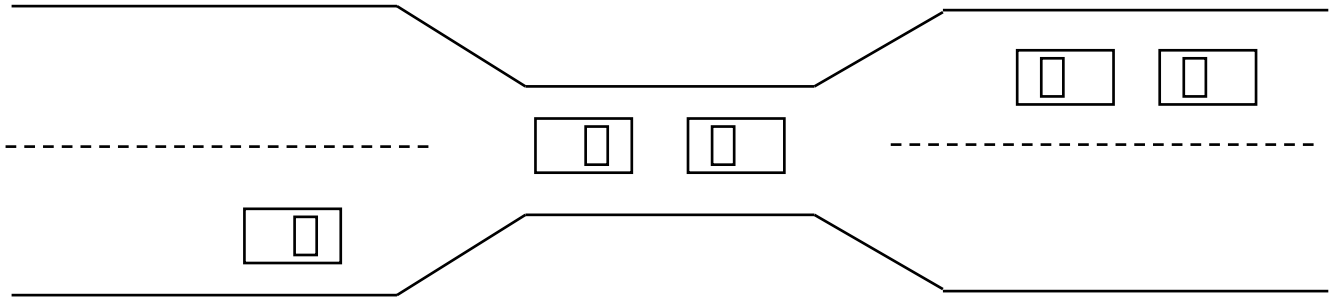
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

- Example
  - Consider a system with three CD RVV drives.
  - Suppose each of three processes holds one of these CD RW drives.
  - If each process now requests another drive, the three processes will be in a deadlock state.
  - Each is waiting for the event "CD RVV is released," which can be caused only by one of the other waiting processes.

The Deadlock Problem



Bridge Crossing Example



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

Why worry about deadlock?

Not all systems need deadlock analysis

- Some are simple
- It may not matter (reboot may be an option)

Some systems are critical

- The control system in your car; on a plane; highenergy physics
- Life-support systems (during surgery, say)
- Online services, where deadlock is expensive

Each of these systems is managed by an OS

Necessary conditions for deadlock

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion.** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. **Hold and wait.** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. **No preemption.** Resources cannot be preempted.; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Necessary conditions for deadlock(cntd)

4. **Circular wait.** A set $\{P_i, ..., P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ..., P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent. However, that it is useful to consider each condition separately.

Resource-Allocation Graph

In some cases deadlocks can be understood more clearly through the use of Resource-Allocation Graphs, having the following properties:

- A set of resource categories, { R_1, R_2, R_3, . . ., R_N}, which appear as square nodes on the graph. Dots inside the resource nodes indicate specific instances of the resource. (E.g. two dots might represent two laser printers.)

- A set of processes, { P_1, P_2, P_3, . . ., P_N }

- **Request Edges** - A set of directed arcs from P_i to R_j, indicating that process P_i has requested R_j, and is currently waiting for that resource to become available.

- **Assignment Edges** - A set of directed arcs from R_j to P_i indicating that resource R_j has been allocated to process P_i, and that P_i is currently holding resource R_j.

Resource-Allocation Graph(cntd)

- Note that a request edge can be converted into an assignment edge by reversing the direction of the arc when the request is granted. (However note also that request edges point to the category box, whereas assignment edges originates from a particular instance dot within the box.)
- The resource-allocation graph shown

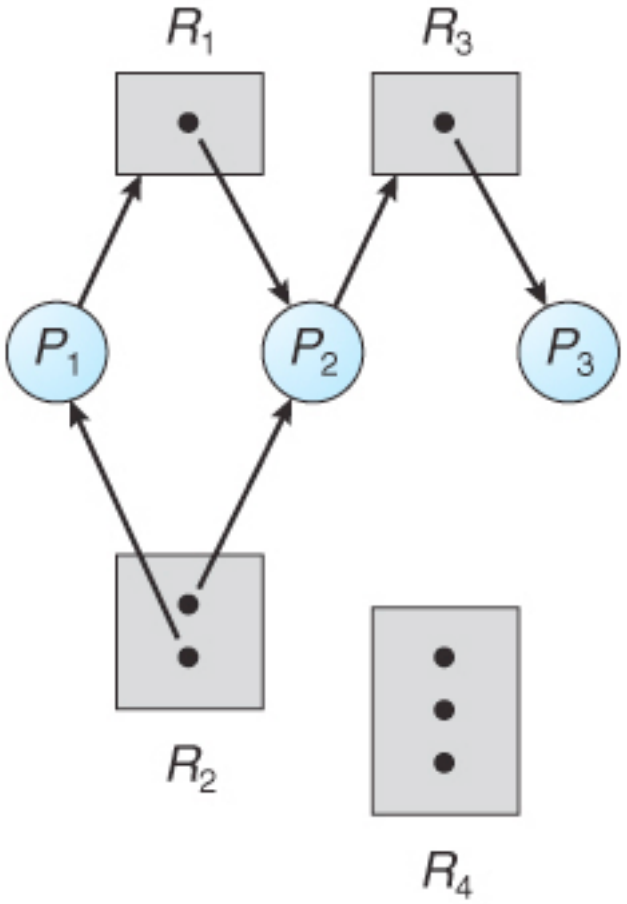
depicts the following situation.

- **The sets P, R, and E:**

P={P_1,P_2,P_3}

R= {R_1,R_2,R_3,R4}

E={p_1->R_1,P_2->R_3,R_1->P_2,R_2->P_1,R_2->P_2,R_3->P_3}



Resource instances:

- One instance of resource type R_1
- Two instances of resource type R_2
- One instance of resource type R_3
- Three instances of resource type R_4

Resource-Allocation Graph(cntd)

- If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are directed graphs.)

- If a resource-allocation graph does contain cycles AND each resource category contains only a single instance, then a deadlock exists.

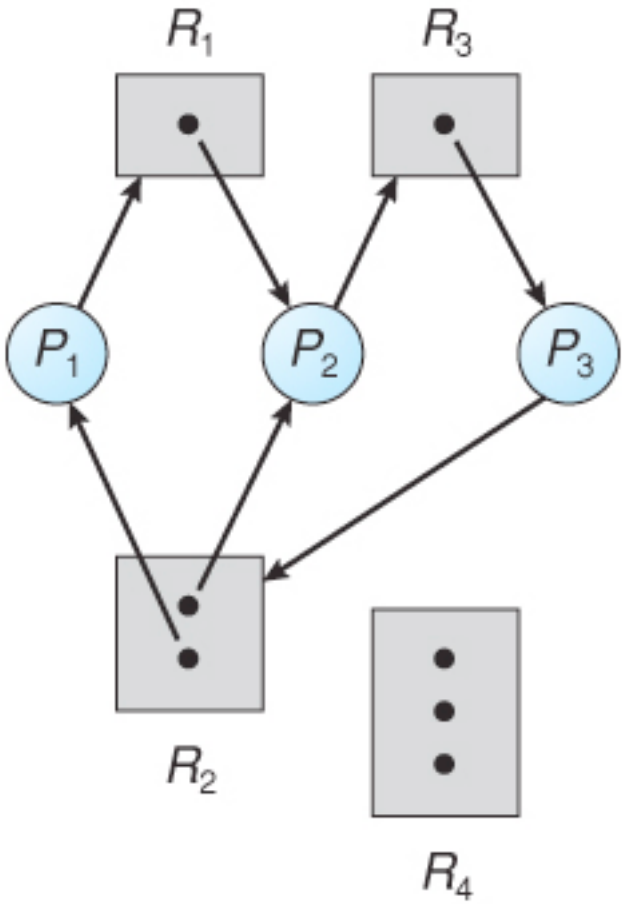
- If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the possibility of a deadlock, but does not guarantee one.

Resource-Allocation Graph(cntd)

- Lets consider the resource-allocation graph given on previous slide. Suppose that process P_3 requests an instance of resource type R_2. Since no resource instance is currently available, a request edge P_3 —>R_2 is added to the graph.

- At this point, two minimal cycles exist in the system:
- P_1->R_1->P_2->R_3->P_3->R_2->P_1
- P_2->R_3->P_3->R_2->P_2

- Processes P_1, P_2, and P_3 are deadlocked. Process P_2 is waiting for the resource R_3, which is held by process P_3. Process P_3 is waiting for either process P_1 or process P_2 to release resource R_2. In addition, process P_1 is waiting for process P_2 to release resource R_1.

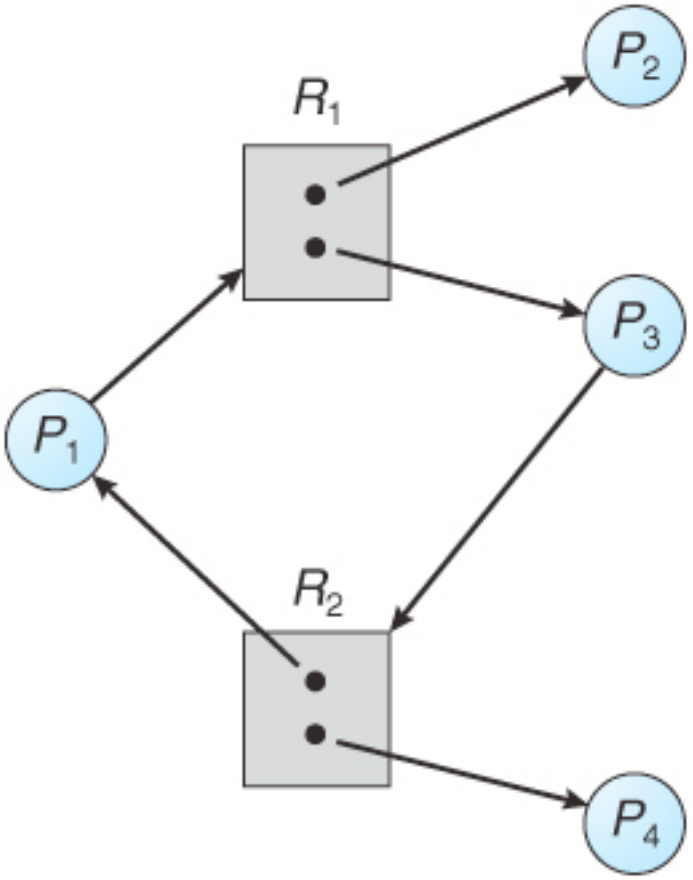


Resource-Allocation Graph(cntd)

- Now consider the resource-allocation graph shown. In this example, we also have a cycle.

- P1->R1->P3->R2->P1

- However, there is no deadlock. Observe that process P4 may release its instance of resource type R?. That resource can then be allocated to P3, breaking the cycle.



Banker’s Algorithm

- Banker's algorithm is a **resource allocation** and **deadlock avoidance** algorithm developed for the safe allocation of predetermined maximum possible amounts of all resources.

- The Banker's algorithm is run by the operating system whenever a process requests resources.

- The algorithm avoids deadlock by denying or postponing the request if it determines that accepting the request could put the system in an unsafe state (one where deadlock could occur).

Banker’s Algorithm (Working)

- For the Banker's algorithm to work, it needs to know three things:

1. How much of each resource each process could possibly request [Max]
2. How much of each resource each process is currently holding [ALLOCATED]
3. How much of each resource the system currently has available [AVAILABLE]

- Resources may be allocated to a process only if it satisfies the following conditions:

- request ≤ max, else set error condition as process has crossed maximum claim made by it.
- request ≤ available, else process waits until resources are available.

Safe and Unsafe States

- A state is considered safe if it is possible for all processes to finish executing.

- Since the system cannot know when a process will terminate, or how many resources it will have requested by then, the system assumes that all processes will eventually attempt to acquire their stated maximum resources and terminate soon afterward.

- The algorithm determines if a state is safe by trying to find a hypothetical set of requests by the processes that would allow each to acquire its maximum resources and then terminate (returning its resources to the system). Any state where no such set exists is an unsafe state.

Data Structures for the Banker's Algorithm

Let *n* = number of processes, *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*].

Safety Algorithm

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

   Work = *Available*

   Finish [*i*] = *false* for *i* - 1,3, …, *n.*
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.

Example of Banker's Algorithm

- 5 processes P_0 through P_4; 3 resource types *A* (10 instances), *B* (5 instances, 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 of Banker's Algorithm

- 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.HOW?

For P1

- Available resources – P1(Need)
- <3,3,2> - <1,2,2> = <2,1,0>

- So P1's request is satisfied and after using the resources it returns the resources to system.
- Available = <2,1,0> + P1(Max) = <2,1,0> + <3,2,2> = <5,3,2>

For P3

- Available resources – P3(Need)
- <5,3,2> - <0,1,1> = <5,2,1>

- So P3's request is satisfied and after using the resources it returns the resources to system.
- Available = <5,2,1> + P3(Max) = <5,2,1> + <2,2,2> = <7,4,3>

For P4

- Available resources – P4(Need)
- <7,4,3> - <4,3,1> = <3,1,2>

- So P4's request is satisfied and after using the resources it returns the resources to system.
- Available = <3,1,2> + P4(Max) = <3,1,2> + <4,3,3> = <7,4,5>

For P2

- Available resources – P2(Need)
- <7,4,5> - <6,0,0> = <1,4,5>

- So P2's request is satisfied and after using the resources it returns the resources to system.
- Available = <1,4,5> + P2(Max) = <1,4,5> + <9,0,2> = <10,4,7>

For P0

- Available resources – P0(Need)
- <10,4,7> - <7,4,3> = <3,0,4>

- So P0's request is satisfied and after using the resources it returns the resources to system.
- Available = <3,0,4> + P0(Max) = <3,0,4> + <7,5,3> = <10,5,7>

Example of Banker's Algorithm (cntd)

- Suppose now that process P1 requests one additional instance of resource type A and two instances of resource type C, so Request = (1,0,2).
- First we check that Request < Available i-e <1,0,2> < <3,3,2>, which is true.

- We then pretend that this request has been fulfilled, and we arrive at the following new state:

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

Example of Banker's Algorithm (cntd)

- We must determine whether this new system state is safe.
- To do so, we execute our safety algorithm and find that the sequence <P1, P3, P4, P0, P2> satisfies the safety requirement. Hence, we can immediately grant the request of process P1.
- However when the system is in this state, a request for (3,3,0) by P4 cannot be granted, since the resources are not available.
- Furthermore, a request for (0,2,0) by P0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

The Dining-Philosophers problem

- Multiple resources (Dijkstra 1968)

- “Five philosophers sit around a table, which is set with 5 plates (one for each philosopher), 5 chopsticks, and a bowl of rice. Each philosopher alternately thinks and eats. To eat, he needs the two chopsticks next to his plate. When finished eating, he puts the chopsticks back on the table, and continues thinking.”

- Philosophers are processes, and chopsticks are resources.

The Dining-Philosophers problem

Problems with dining philosophers

- The system may deadlock: if all 5 philosophers take up their left chopstick simultaneously, the system will halt (unless one of them puts one back)

- A philosopher may starve if her neighbors have alternating eating patterns

Simple Solution to Dining philosopher's Problem

- Simple solution to the dining philosopher problem is to restrict the number of philosophers allowed access to the table.

- If there are **N** chopsticks but only **N-1** philosophers allowed to compete for them, at least one will succeed, even if they follow a rigid sequential protocol to acquire their chopsticks.

- This solution is implemented with an integer semaphore, initialized to N-1. This solution avoid deadlock a situation in which all of the philosophers have grabbed one chopstick and are deterministically waiting for the other, so that there is no hope of recovery.

- However, they may still permit *starvation*, a scenario in which at least one hungry philosopher never gets to eat.

Simple Solution to Dining philosopher’s Problem

- Starvation occurs if the solution allow an individual to eat repeatedly, thus keeping another from getting a chopstick.

- The starving philosopher runs, perhaps, but doesn't make progress. Under some notions of fairness the solutions given above can be said to be correct.

Arbitrator solution using semaphore

- Another approach is to guarantee that a philosopher can only pick up both forks or none by introducing an arbitrator, e.g., a waiter.

- In order to pick up the forks, a philosopher must ask permission of the waiter.

- The waiter gives permission to only one philosopher at a time until he has picked up both his forks. Putting down a fork is always allowed. The waiter can be implemented as a mutex.

- In addition to introducing a new central entity (the waiter), this approach can result in reduced parallelism: if a philosopher is eating and one of his neighbors is requesting the forks, all other philosophers must wait until this request has been fulfilled even if forks for them are still available.

Arbitrator solution using semaphore(cntd)

- Another approach is to guarantee that a philosopher can only pick up both forks or none by introducing an arbitrator, e.g., a waiter.

- In order to pick up the forks, a philosopher must ask permission of the waiter.

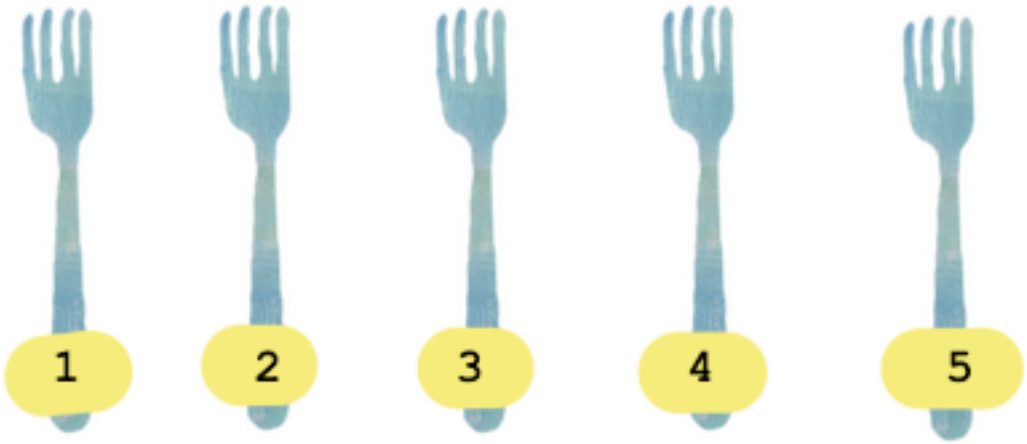
- The waiter gives permission to only one philosopher at a time until he has picked up both his forks. Putting down a fork is always allowed. The waiter can be implemented as a mutex.

- In addition to introducing a new central entity (the waiter), this approach can result in *reduced parallelism*: if a philosopher is eating and one of his neighbors is requesting the forks, all other philosophers must wait until this request has been fulfilled even if forks for them are still available.

Resource Hierarchy solution

- Let’s number the forks:

- Now the rule is, if you’re using two forks, you need to pick up the lower numbered fork first.



Resource Hierarchy solution

Resource Hierarchy solution

- Philosopher#1 picks up fork #1
- Philosopher#2 picks up fork #2
- Philosopher#3 picks up fork #3
- Philosopher#4 picks up fork #4
- Philosopher#5 can’t pick up fork #5! Because he will need two forks and he needs to pick up the lower numbered fork first!

- So fork #5 goes to Philosopher#4 – no deadlock!

Resource Hierarchy solution

- Resource hierarchy avoids deadlocks! But it is slow. Suppose you have forks #3 and #5. Then you decide you need fork #2. Well forks #3 and #5 are larger numbers. So you'll have to:

- put down fork #5
- put down fork #3 (the order you put these down in doesn't matter)
- pick up fork #2
- pick up fork #3

- Wastes a lot of time!