CMPT 300
Operating System I
Deadlock

Dr. Hazra Imran

Summer 2022

Resource-Allocation Graph

- Graph G = (V, E)
  - Set of vertices V, and set of edges E

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system

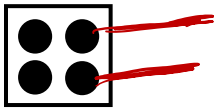
- request edge - directed edge $P_i \rightarrow R_j$

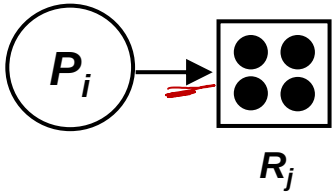
- assignment edge- directed edge $R_j \rightarrow P_i$

Resources
single
multiple
req / assignment edge

Resource-Allocation Graph

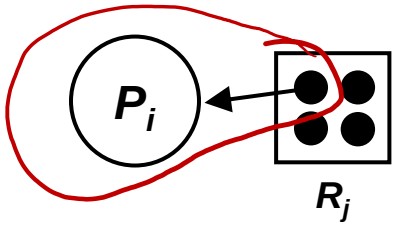
- Process 

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



(request edge)

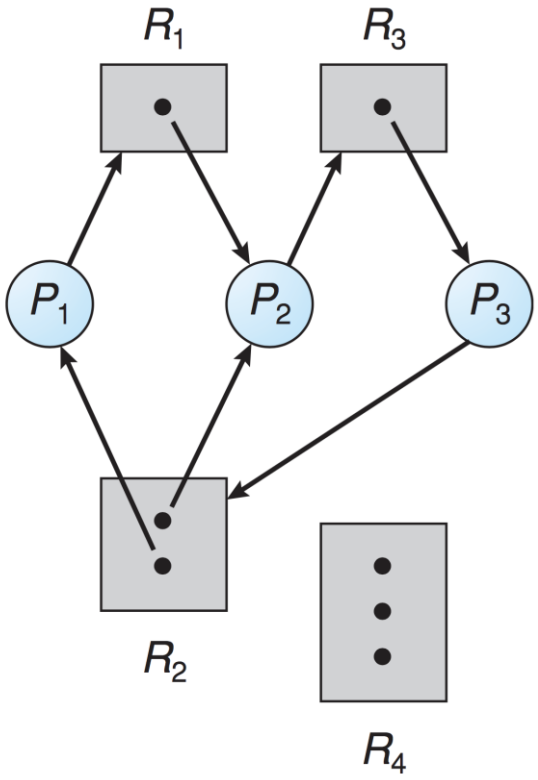
- P_i is holding an instance of R_j



(assignment edge)

RAG Example : Try!

cycle + single instance → deadlock



Detect the cycles?

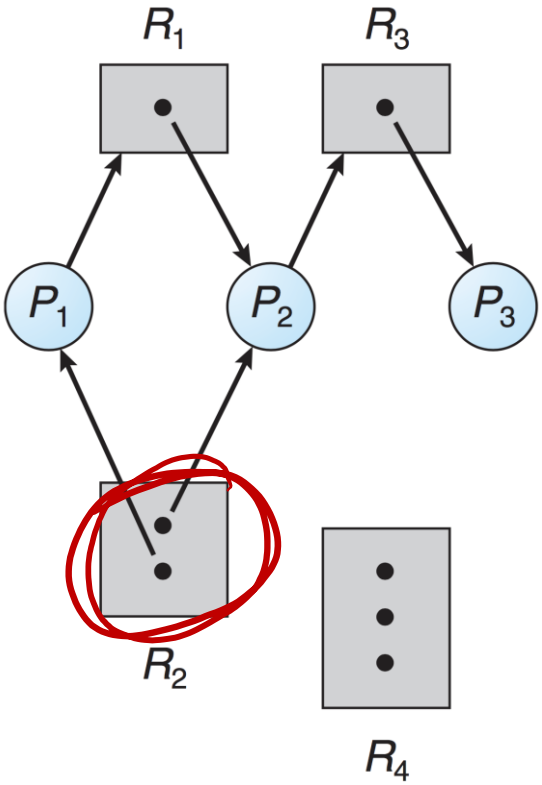
P1→R1→P2→R3→P3→R2→P1

P2→R3→P3→R2→P2

✓ cycle + single → deadlock

cycle + multiple instance → possibility

Resource-Allocation Graph - Clicker



Deadlock

(A) Yes

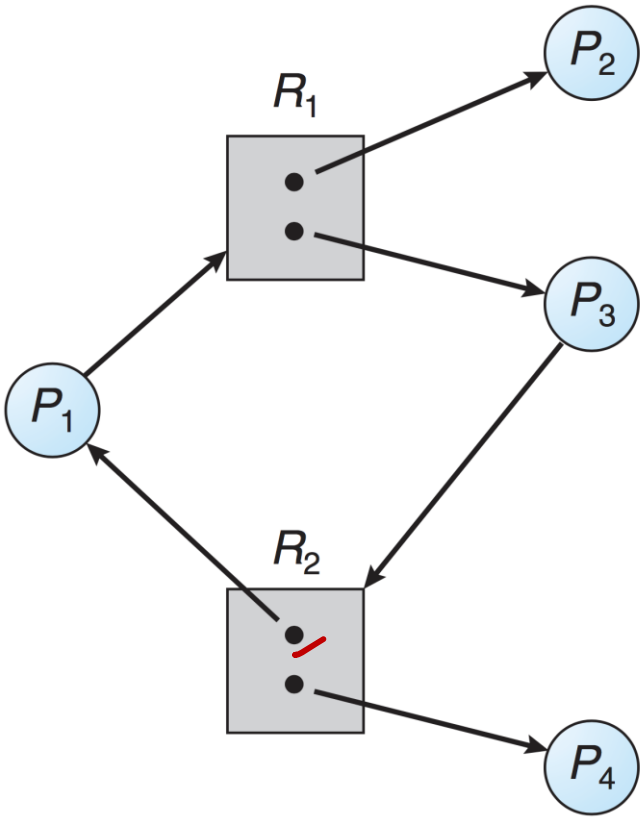
(B) No

P_1 has R_2 & waiting for R_1

P_2 has R_2 & waiting for R_3

P_3 has R_3

Resource-Allocation Graph



Deadlock?

(A) Yes (B) No

P_1 has R_2 & waiting for R_1

P_2 R_1

P_3 R_1 & waiting for R_2

P_4 has R_2

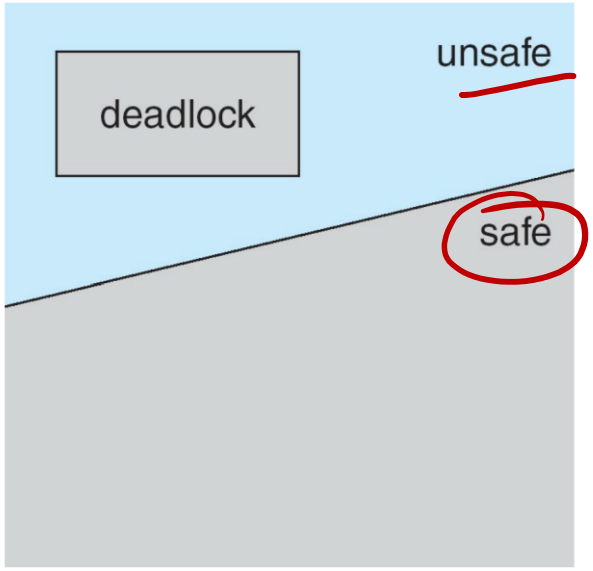
How do we know if we have a deadlock?

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

A cycle is a necessary condition for deadlock, but not a sufficient condition for deadlock.

Possibility of Deadlock

- If a system is in the safe state → no deadlocks
- If a system is in an unsafe state → possibility of deadlock



Summary

- If a RAG does not contain a cycle, then there is absolutely no possibility of deadlock
- If a RAG contains a cycle, then there is the possibility of deadlock
- If each resource type has exactly one instance, then a cycle implies that deadlock has occurred
- If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred
- If all instances of a resource are allocated to a process in a cycle, then there is a deadlock

Methods of Handling Deadlocks

- Ensure system will **never** enter a deadlock state
  - Deadlock **Prevention** (1)
  - Deadlock **Avoidance** (2)
- Allow the system to enter deadlock state, then recover
  - Deadlock **Detection and Recovery** (3)
- Ignore the problem and pretend that deadlocks never occur in the system
  - user has to take corrective actions

Ignorance (4)

Deadlock Prevention

Approach:

We don't allow one of the four necessary conditions to hold

- Mutual Exclusion
- Hold and Wait
- No preemption
- Circular wait

Preventing Mutual Exclusion

- use only sharable resources (e.g., a read-only file)
- impossible for practical systems

↓ Concurrency X

Prevent Hold and Wait

Guarantee that whenever a process requests a resource, it does not hold any other resources

Approach 1: process must request all resources upfront, as a single unit.

E.g. do not pick up one chopstick if you cannot pick up the other

Approach 2: only allow a process to request resources only when the process has none allocated to it

starvation problem
inefficient
↓
low resource utilization

Prevent Hold and Wait

Acquire **all** locks at once, **atomically**

```
1 lock(prevention);
2 lock(L1);
3 lock(L2);
4 ...
5 unlock(prevention);
```

Disadvantages:

- Must know which locks to acquire in advance
- Decreased concurrency

Prevent No Preemption (i.e., allow preemption)

- Permit the OS to take away resources from a process.
- If a process that is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
- Preempted resources are added to the list of resources for which the process is waiting
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Prevent No Preemption

Give it a try first, if unsuccessful, give up all already-holding locks and retry

```
while (true) {
  lock(L1);
  if (try_lock(L2) == FAILED) {
    unlock(L1);
  }
  ...
}
```

timeout specified

livelock → overlapping locks

t1 t2 t3

repeating steps

add delay b/w retries

Prevent circular wait

Impose a total ordering on all resource types

- Enforce that each process/thread requests resources in a consistent order
  - Using unique resource identifiers (IDs)
  - Using the lock's address
  - Example: each thread must always acquire the first L1, then L2
- Only allow requests in an increasing order

Time

Thread 1:
```
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
```

Thread 2:
```
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
```

Prevention ⟷ unreasonable

Deadlock Prevention

- Kernel can take preventative steps
  - Resource utilization could be poor
- Or the application programmer can take explicit steps
  - E.g., ordering of lock operations
  - Dealing with preemption

Deadlock Avoidance

- Deadlock prevention techniques place a lot of restrictions on what can be done
  - In particular: allocation decisions are made using uniformly applied rules

- Next approach (**avoidance**): dynamically make allocation decisions on a case-by-case basis

Deadlock Avoidance

- The system's resource-allocation state is defined by the number of available and allocated resources and the maximum possible demands of the processes. (need)
- When a process requests an available resource, the system must decide if immediate allocation leaves the system in a *safe state*.

Deadlock Avoidance by Scheduling

- Know which locks might be requested by which threads
- Schedule threads in a way that guarantees no deadlocks can occur

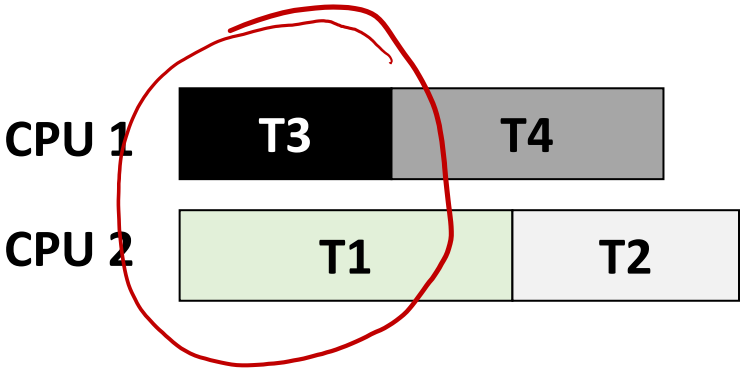
Disadvantage: require global knowledge about all participating threads and resources

Deadlock Avoidance - Example 1

- Suppose we have two cores and four threads
- Lock acquisition demands of the threads:

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

- A smart scheduler could compute that as long as T1 and T2 are not run at the same time, no deadlock could ever arise

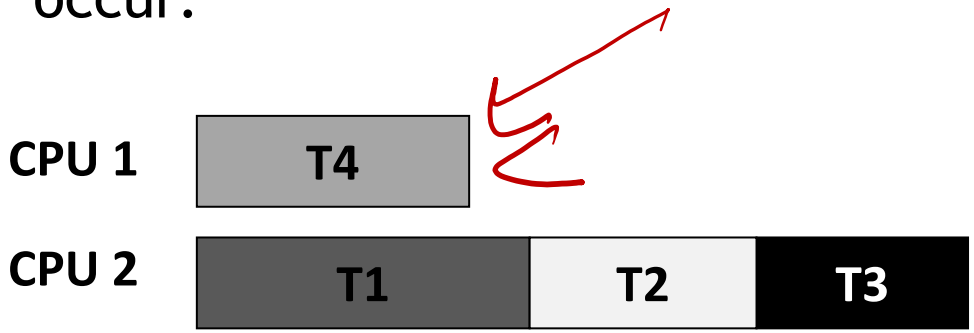


Deadlock Avoidance - Example 2

- More contention for the same resources

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

- A possible schedule that guarantees that no deadlock could ever occur:



Disadvantage: can possibly increase the time needed to complete the jobs

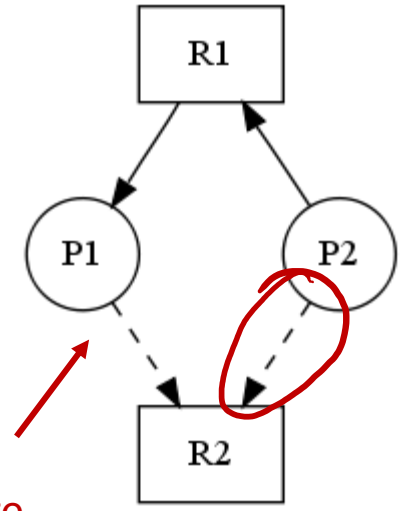
Deadlock Avoidance Algorithms

Two deadlock avoidance algorithms:

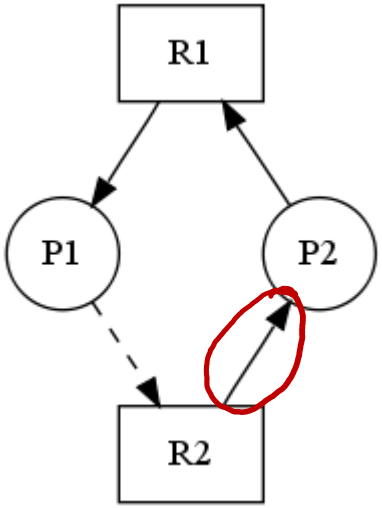
- resource-allocation graph algorithm
- Banker's algorithm – classical

only applicable when we only have 1 instance of each resource type

Resource-allocations graphs for deadlock avoidance



Claim (future request) edge



- Suppose that process Pi requests a resource Rj
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a **cycle** in the RAG

Banker algorithm

- Multiple instances
- Each process has to a priori specify its maximum requirement of resources.
- A process is admitted for execution only if its maximum requirement of resources is within the system capacity of resources
  - When a process requests a resource, it may have to wait
  - When a process gets all its resources it must return them in a finite amount of time

Banker's needs to know **three** things:

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

need

Example

5 processes and 3 resources R1 (10 instances), R2(5 instances) and R3 (7 instances)

Process	Allocated			Max			Available			Need		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	0	1	0	7	5	3	3	3	2	7	4	3
P2	2	0	0	3	2	2				1	2	2
P3	3	0	2	9	0	2				6	0	0
P4	2	1	1	2	2	2				0	1	1
P5	0	0	2	4	3	3				4	3	1

Step 1: Compute Need Matrix

Need [i] = Max[i] - Allocated[I]

Step 2: Check if system is in safe state?

Example

5 processes and 3 resources R1 (10 instances), R2(5 instances) and R3 (7 instances)

Process	Allocated			Max			Available			Need (Max - Allocation)		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	0	1	0	7	5	3	3	3	2	7	4	3
P2	2	0	0	3	2	2				1	2	2
P3	3	0	2	9	0	2				6	0	0
P4	2	1	1	2	2	2				0	1	1
P5	0	0	2	4	3	3				4	3	1

- Check P1

P1 need <7,4,3> but available is <3,3,2>. Check for other processes

- Check P2

P2 needs <1,2,2> and available is <3,3,2> so resources will be allocated.

After finishing P2 will release its allocated resources as well.

Example

5 processes and 3 resources R1 (10 instances), R2(5 instances) and R3 (7 instances)

Process	Allocated			Max			Available			Need (Max - Allocation)		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	0	1	0	7	5	3	5	3	2	7	4	3
P3	3	0	2	9	0	2				6	0	0
P4	2	1	1	2	2	2				0	1	1
P5	0	0	2	4	3	3				4	3	1

- Check P3, resources cannot be allocated. Next process
- Check P4. We can allocate resources.

<P2>

Example

5 processes and 3 resources R1 (10 instances), R2(5 instances) and R3 (7 instances)

Process	Allocated			Max			Available			Need (Max - Allocation)		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	0	1	0	7	5	3				7	4	3
P3	3	0	2	9	0	2				6	0	0
							7	4	3			
P5	0	0	2	4	3	3				4	3	1

- Check P3, resources cannot be allocated. Next process
- Check P4. We can allocate resources.

<P2, P4>

Example

5 processes and 3 resources R1 (10 instances), R2(5 instances) and R3 (7 instances)

Process	Allocated			Max			Available			Need (Max - Allocation)		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	0	1	0	7	5	3				7	4	3
P3	3	0	2	9	0	2				6	0	0
							7	4	3			
P5	0	0	2	4	3	3				4	3	1

- Check P5. We can allocate resources.

<P2, P4>

Example

5 processes and 3 resources R1 (10 instances), R2(5 instances) and R3 (7 instances)

Process	Allocated			Max			Available			Need (Max - Allocation)		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	0	1	0	7	5	3				7	4	3
P3	3	0	2	9	0	2				6	0	0
							7	4	5			

- Check P5. We can allocate resources.

<P2, P4, P5>

Example

5 processes and 3 resources R1 (10 instances), R2(5 instances) and R3 (7 instances)

Process	Allocated			Max			Available			Need (Max - Allocation)		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	0	1	0	7	5	3				7	4	3
P3	3	0	2	9	0	2				6	0	0
							7	4	5			

- Check P1
- We can allocate resources

<P2, P4, P5>

Example

5 processes and 3 resources R1 (10 instances), R2(5 instances) and R3 (7 instances)

Process	Allocated			Max			Available			Need (Max - Allocation)		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1							7	5	5			
P3	3	0	2	9	0	2				6	0	0

- Check P1
- We can allocate resources

<P2, P4, P5, P1>

– system – safe state.

Example

5 processes and 3 resources R1 (10 instances), R2(5 instances) and R3 (7 instances)

Process	Allocated			Max			Available			Need (Max - Allocation)		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1							7	5	5			
P3	3	0	2	9	0	2				6	0	0

- Check P3
- We can allocate resources

Now available will be
<10,5, 7>

<P2, P4, P5, P1, P3>