CS 4410

Operating Systems

Deadlocks
Characterization & Detection

Summer 2016

Cornell University

Today

- Deadlocks
- Detection algorithm

Racing for resources

- Threads are racing to acquire resources.
  - Threads may belong to different processes.
  - Resources may be logical (user data, OS structures) or physical (memory, printer, disk).
- Assume there is a mechanism that coordinates the access of threads to resources.
- This mechanism may be a combination of:
  - Synchronization primitives.
  - The operating system.
  - Resources themselves.

Safety property

- Coordinating threads involves blocking threads until resources are available.
- This coordination mechanism should satisfy the safety property: deadlock freedom!
  - At any point of time, at least one thread should be able to make progress.
- Undesirable scenario:
  - Process A acquires resource 1, and is waiting for resource 2
  - Process B acquires resource 2, and is waiting for resource 1
  - Deadlock!

Deadlock

Example 1: Semaphores

```
semaphore:  mutex1 = 1    /* protects file */
            mutex2 = 1    /* protects printer */
```

```
Process A code:
 {
    /* initial compute */

   P(mutex1)
   P(mutex2)

  /* use file & printer*/

   V(mutex2)
   V(mutex1)
}
```

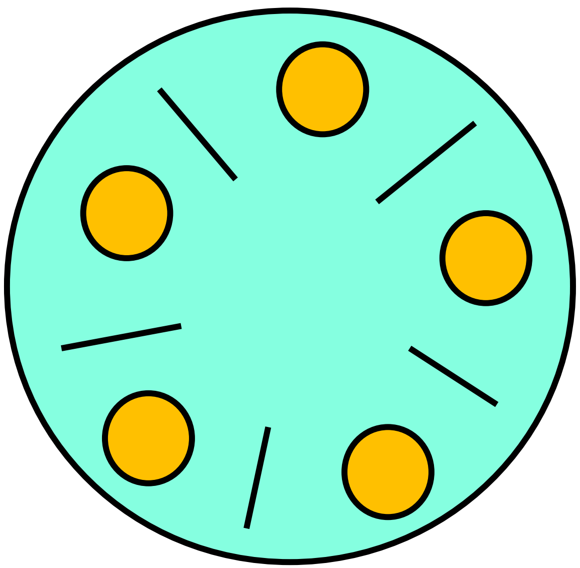
```
Process B code:
 {
    /* initial compute */

   P(mutex2)
   P(mutex1)

  /* use file & printer */

   V(mutex1)
   V(mutex2)
}
```

Example 2: Dining Philosophers



```
class Philosopher:
        chopsticks[N] = [Semaphore(1),…]

        Def __init__(mynum)
                self.id = mynum

        Def eat():
                right = (self.id+1) % N
                left = (self.id-1+N) % N
                while True:
                P(left)
                        P(right)
                # eat
                V(right)
                        V(left)
```

Deadlock

- A cycle of waiting among a set of threads where each thread is waiting for some other thread in the cycle to take some action.
- Caused by the coordination mechanism.

Four Conditions for Deadlock

- **Mutual Exclusion**
  - At least one resource must be held in non-sharable mode.
- **Hold and wait**
  - There exists a process holding a resource, and waiting for another.
- **No preemption**
  - Resources cannot be preempted.
- **Circular wait**
  - There exists a set of processes {P1, P2, … PN}, such that
    - P1 is waiting for P2, P2 for P3, …. and PN for P1.

- If some of these conditions do not hold, then there is no deadlock(necessary conditions).
- If all four conditions hold, then there may not be a deadlock (not sufficient conditions).

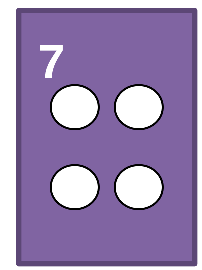
Deadlock Detection

- Stop the world.
- Check if the conditions for which threads are waiting can be ever satisfied.
  - Check if requested resources can ever be allocated to threads.

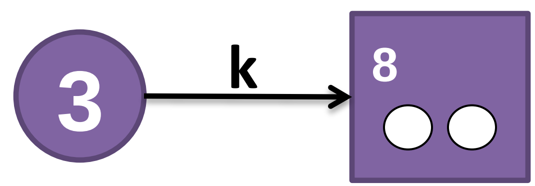
Resource Allocation Graph (RAG)

- 2 kinds of nodes
- A process P_3 represented as:

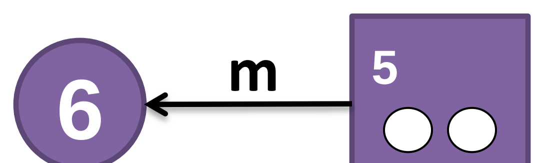
- A resource R_7 represented as:
  - A resource often has multiple identical units, such as "blocks of memory".
  - Represent these as circles in the box.



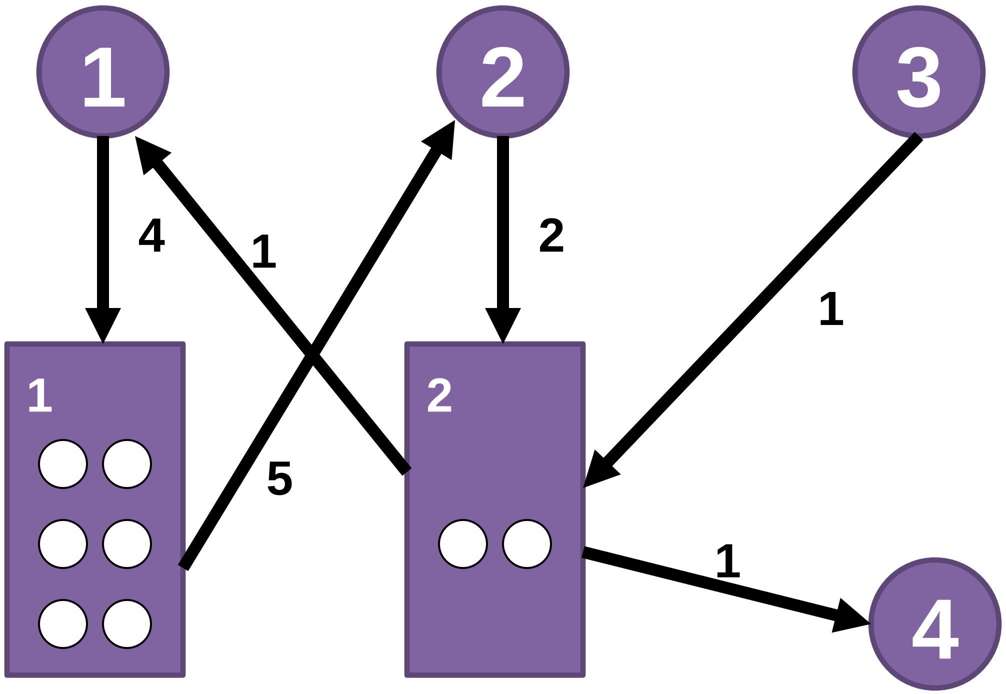
- Edge from P_3 to R_8:
  - P_3 wants k units from R_8.



- Edge from R_5 to P_6:
  - P_6 has m units from R_5.



RAG: Example



Can all requests be satisfied?

Deadlock detection with RAG

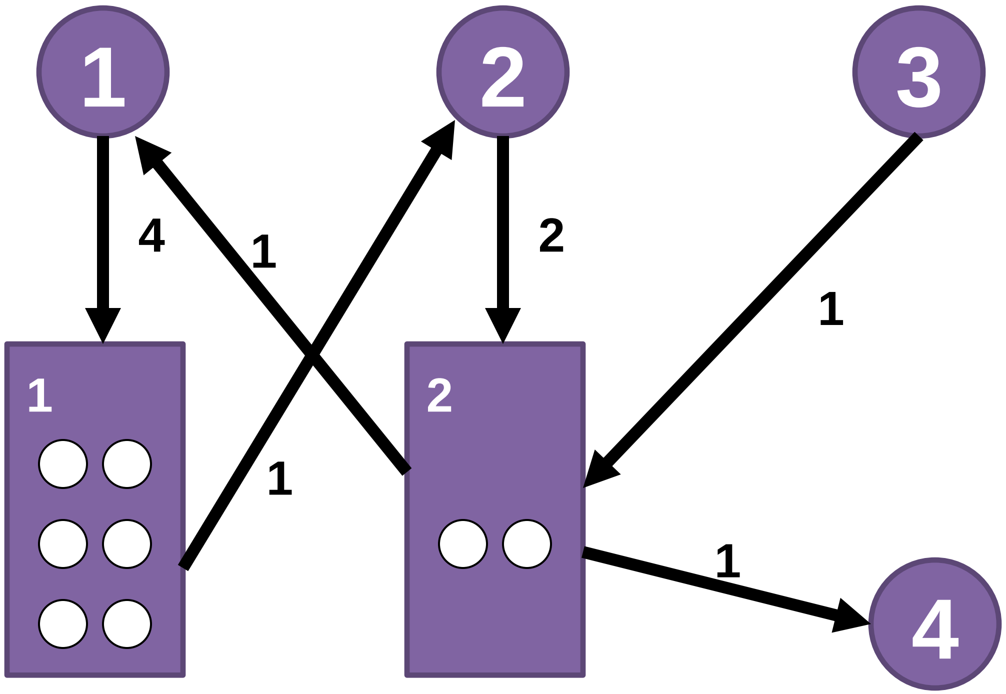
Start satisfying the requests of each process, until:

- no process is left → no deadlock, or
- no remaining request can be satisfied → deadlock.

RAG reduction

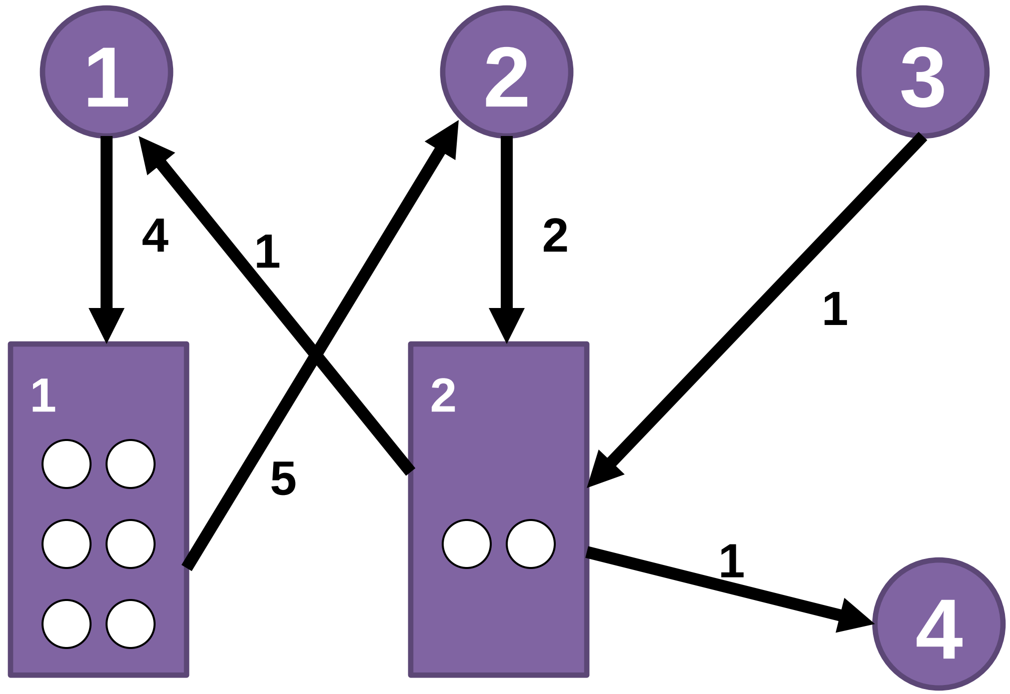
- Find satisfiable process P:
  - available amount of resource ≥ amount requested.
- Erase P.
  - Intuition: Grant the request, let it run, eventually it will release the resource.
- Repeat until all processes gone or irreducible.

Is this graph reducible?



Yes! The system is not deadlocked.

Is this graph reducible?



No! The system is deadlocked.

Detection Algorithm

Data structures:

n:	number of processes
m:	number of resource types
available[1..m]	available[j] is number of available resources of type j
request[1..n,1..m]	current demand of each Pi for each Rj
allocation[1..n,1..m]	current allocation of resource Rj to Pi
free[1..m]	free[j] is number of free resources of type j (not used by any process)
finish[1..n]	true if Pi's request can be satisfied

Detection Algorithm

1. free[] = available[]
2. for all processes i: finish[i] = allocation[i]==[0,0,…,0])
3. find a process i such that finish[i]=false and request[i] <= free

   if no such i exists, goto 7
4. free = free + allocation[i]
5. finish[i]=true
6. goto 3
7. system is deadlocked iff finish[i]=false for some process i

Detection Algorithm: Example

	Allocation			Request			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

- The system is not in a deadlocked state.
- What will happen if P2 makes an additional request for one instance of type R3?

Dealing with Deadlocks

Reactive Approaches:

- Periodically check for evidence of deadlock
  - For example, using a graph reduction algorithm
- Then need a way to recover
  - Could blue screen and reboot the computer
  - Could pick a “victim” and terminate that thread
    - But this is only possible in certain kinds of applications
    - Basically, thread needs a way to clean up if it gets terminated and has to exit in a hurry!
  - Often thread would then “retry” from scratch

(despite drawbacks, database systems do this)

Dealing with Deadlocks

Proactive Approaches:

- Deadlock Prevention and Avoidance
  - Prevent one of the 4 necessary conditions from arising
  - …. This will prevent deadlock from occurring

Today

- Deadlocks
- Detection algorithm

Coming up…

- Next lecture: prevention and avoidance of deadlocks
- HW2: due tonight
- In-class exam: tomorrow, last 30mins