Deadlocks
Detection and Avoidance

Prof. Bracy and Van Renesse
CS 4410
Cornell University

based on slides designed by Prof. Sirer

System Model

- There are non-shared computer resources
  - Maybe more than one instance
  - Printers, Semaphores, Tape drives, CPU
- Processes need access to these resources
  - Acquire resource
    - If resource is available, access is granted
    - If not available, the process is blocked
  - Use resource
  - Release resource
- Undesirable scenario:
  - Process A acquires resource 1, and is waiting for resource 2
  - Process B acquires resource 2, and is waiting for resource 1

  $\Rightarrow$ Deadlock!

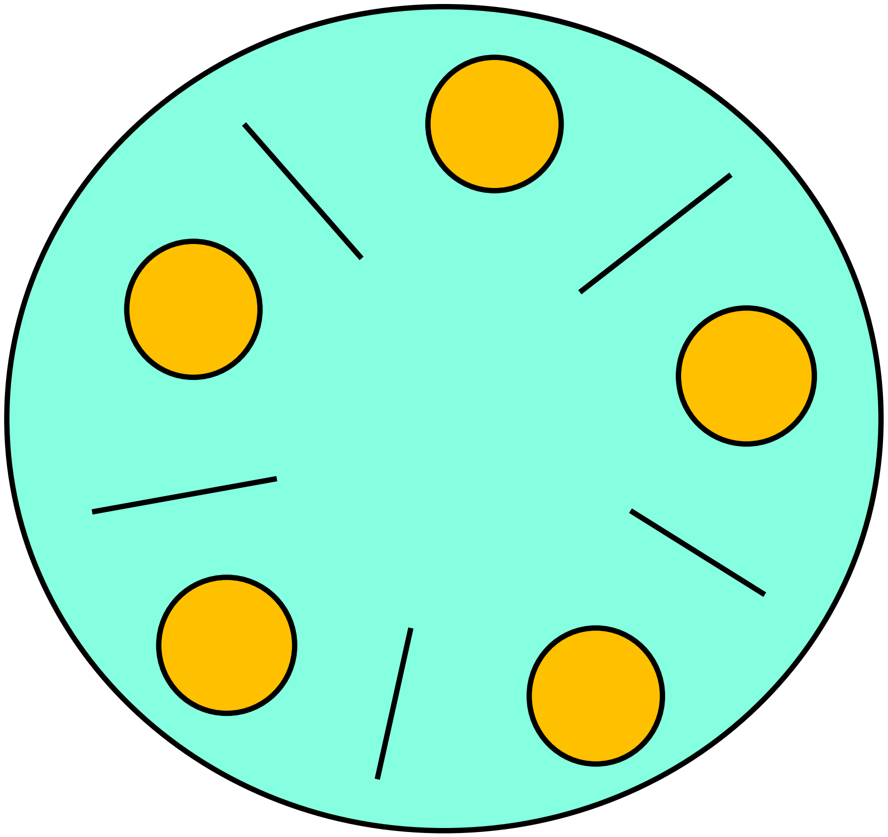
Example 1: Semaphores

```
semaphore: file_mutex = 1             /* protects file resource */
           printer_mutex = 1          /* protects printer resource */
```

```
Process A code:
 {
    /* initial compute */
   P(file_mutex )
   P(printer_mutex)

  /* use both resources */

   V(printer_mutex)
   V(file_mutex )
}
```

```
Process B code:
 {
    /* initial compute */
   P(printer_mutex)
   P(file_mutex)

  /* use both resources */

   V(file_mutex)
   V(printer_mutex)
}
```

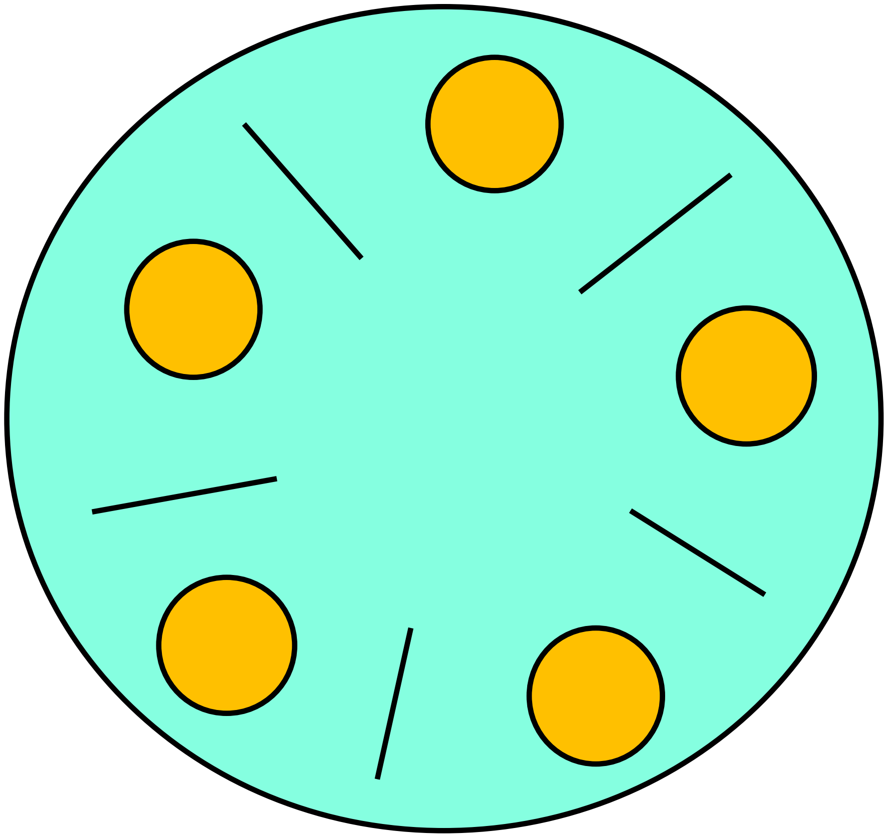
Example 2: Dining Philosophers



```
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:


      # om nom nom
```

- Philosophers go out for Chinese food
- They need exclusive access to two chopsticks to eat their food

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)
      # om nom nom
      V(right)
      V(left)
```

- Philosophers go out for Chinese food
- They need exclusive access to two chopsticks to eat their food

Classic Deadlock

Four Conditions for Deadlock

Necessary conditions for deadlock to exist:

- **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 $\{P_1, P_2, \ldots P_N\}$, such that
    - P_1 is waiting for P_2, P_2 for P_3, …. and P_N for P_1

All four conditions must hold for deadlock to occur

(Edward Coffman, 1971)

Real World Deadlocks?

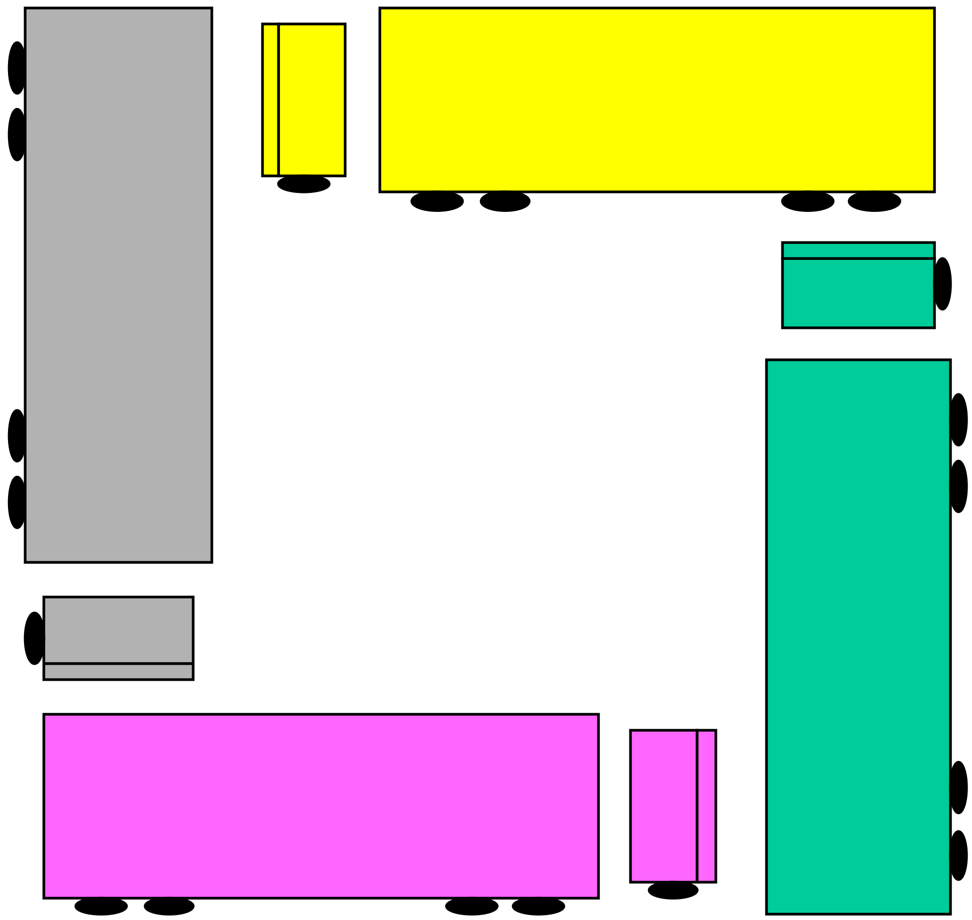
- Truck A has to wait for truck B to move

1. Mutual Exclusion
2. Hold and wait
3. No preemption
4. Circular wait

Deadlock?

Real World Deadlocks?

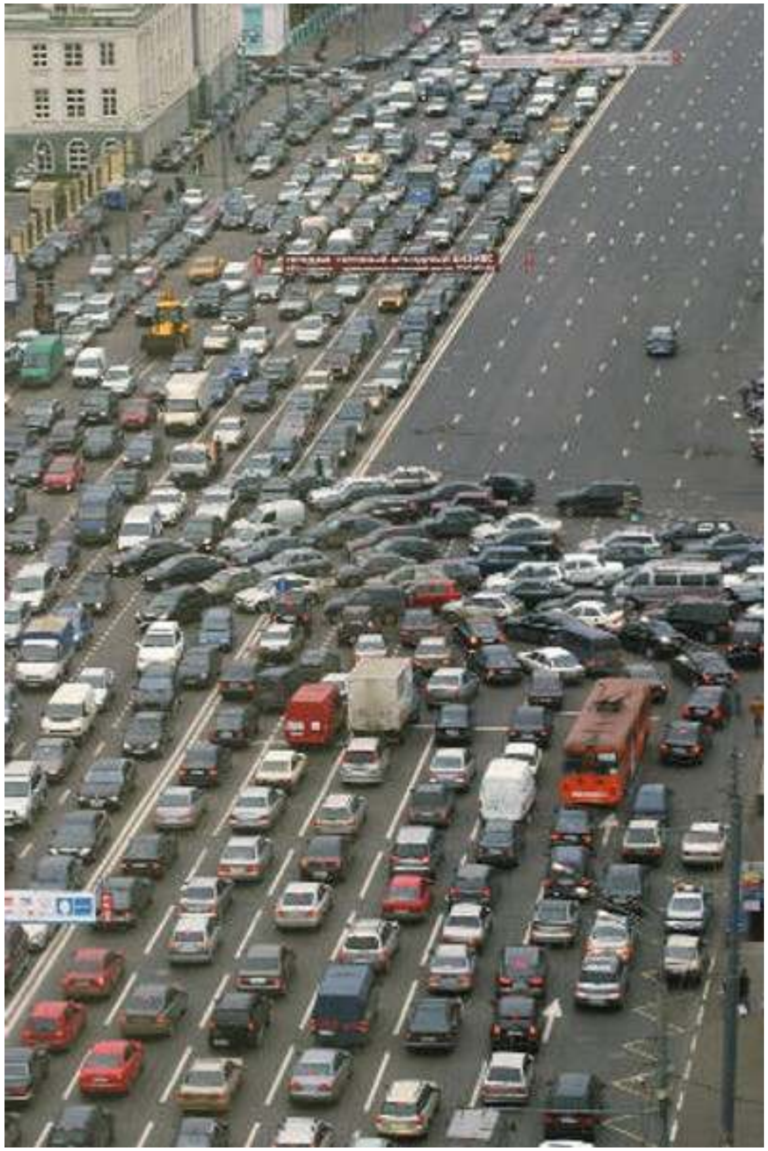
- Gridlock



1. Mutual Exclusion
2. Hold and wait
3. No preemption
4. Circular wait

Deadlock?

Deadlock in Real Life?

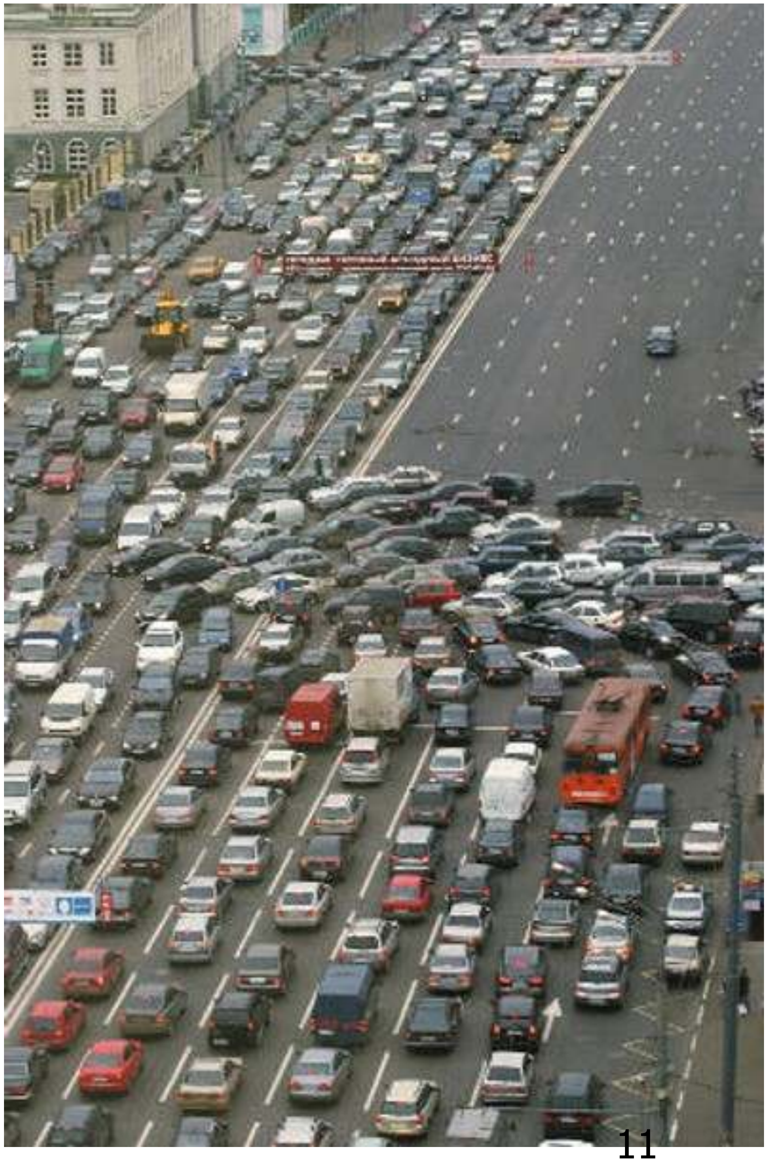


1. Mutual Exclusion
2. Hold and wait
3. No preemption
4. Circular wait

Deadlock?

Deadlock in Real Life?

- No circular wait!
- Not a deadlock!
  - At least, not as far as we can see from the picture
- Will ultimately resolve itself given enough time



Deadlock in Real Life

Avoiding deadlock

- How do cars do it?
  - Try not to block an intersection
  - Must back up if you find yourself doing so
- Why does this work?
  - "Breaks" a wait-for relationship
  - Intransigent waiting (refusing to release a resource) is one of the four key elements of a deadlock

Can we fix Dining Philosophers?

Testing for deadlock

(1) Create a *Wait-For Graph*

- 1 Node per Process
- 1 Edge per Waiting Process, P

  (from P to the process it's waiting for)

Note: Do this in a single instant of time, not as things change

(2) Cycles in graph indicate deadlock

Testing for cycles (= deadlock)

- Find a node with no outgoing edges
  - Erase node
  - Erase any edges coming into it

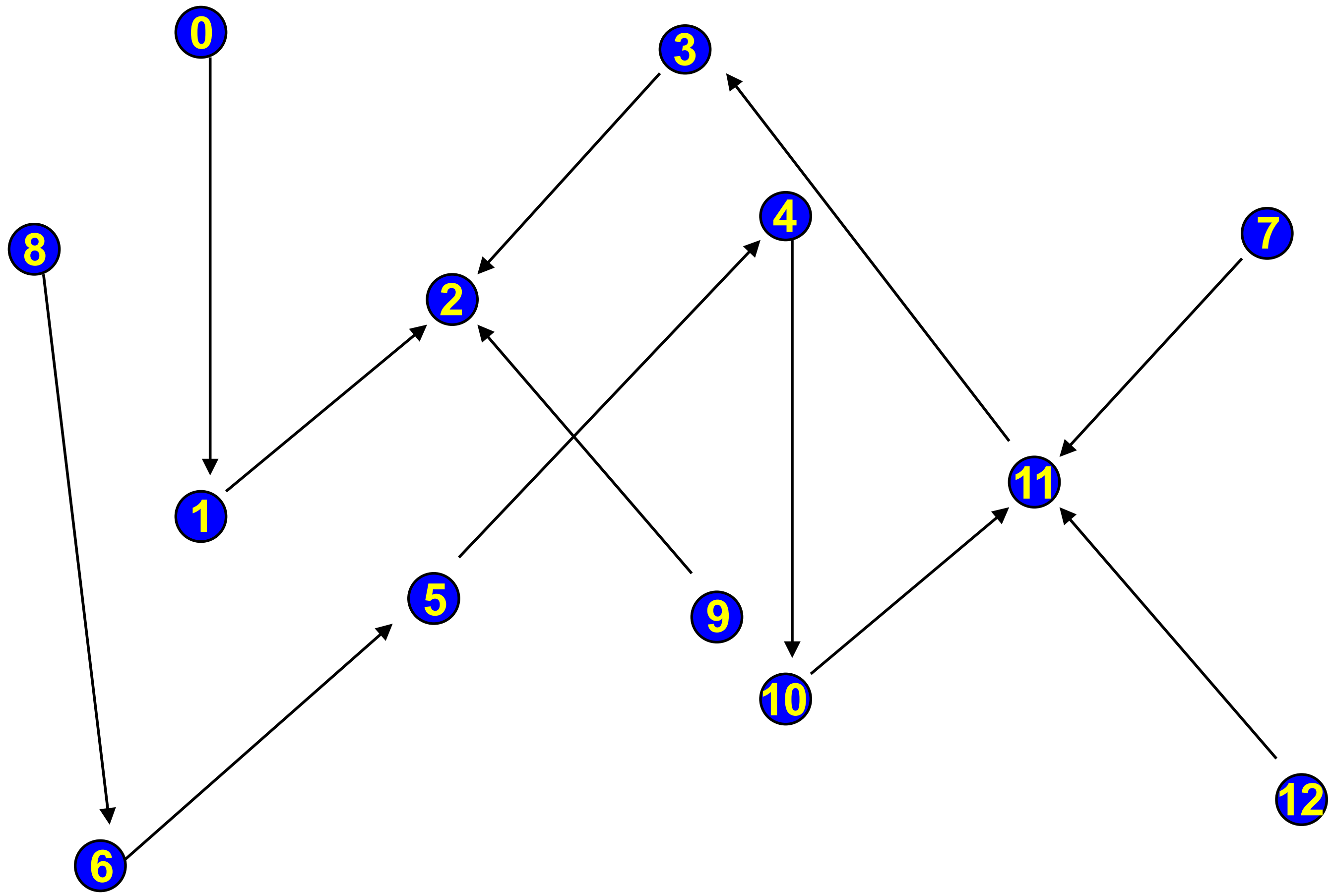
  Intuition: This was a process waiting on nothing. It will eventually finish, and anyone waiting on it, will no longer be waiting.

Erase whole graph $\Leftrightarrow$ graph has no cycles

Graph remains $\Leftrightarrow$ deadlock

This is a graph reduction algorithm.

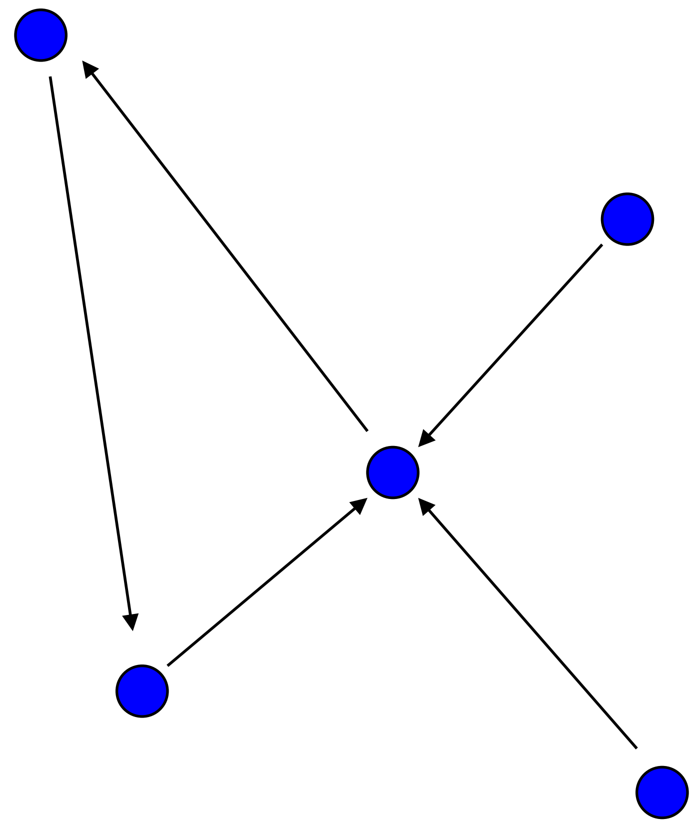
Graph reduction example



This graph can be "fully reduced", hence there was no deadlock at the time the graph was drawn.

(Obviously, things could change later!)

Graph reduction example



Irreducible graph

- contains a cycle

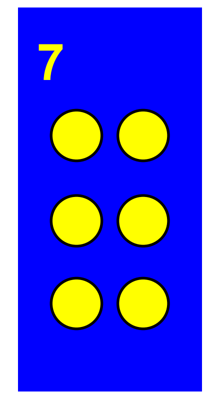
  (only some processes are in the cycle)
- represents a deadlock

Resource waits

- Processes usually don’t wait for each other
  - They wait for resources used by other processes
  - P1 needs access to the critical section of memory P2 is using
- Can we extend our graphs to represent resource wait?

Resource Allocation Graphs

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

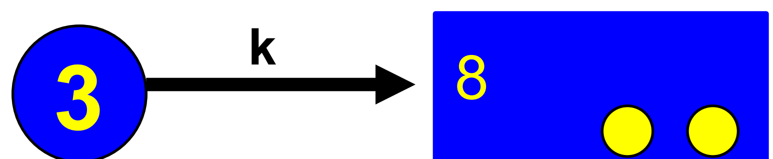
  - **A resource:** R_7 will be represented as: 

    - multiple identical units of the resource (e.g., blocks of memory) = circles in the box

- **Edge from P_3 to R_8:**

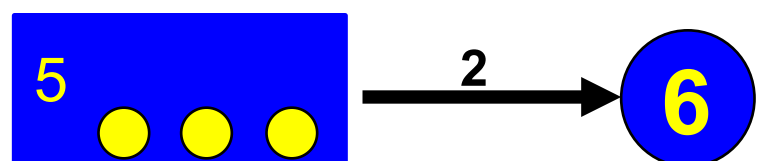
  "P_3 wants *k* units of R_8"

  (default k = 1)

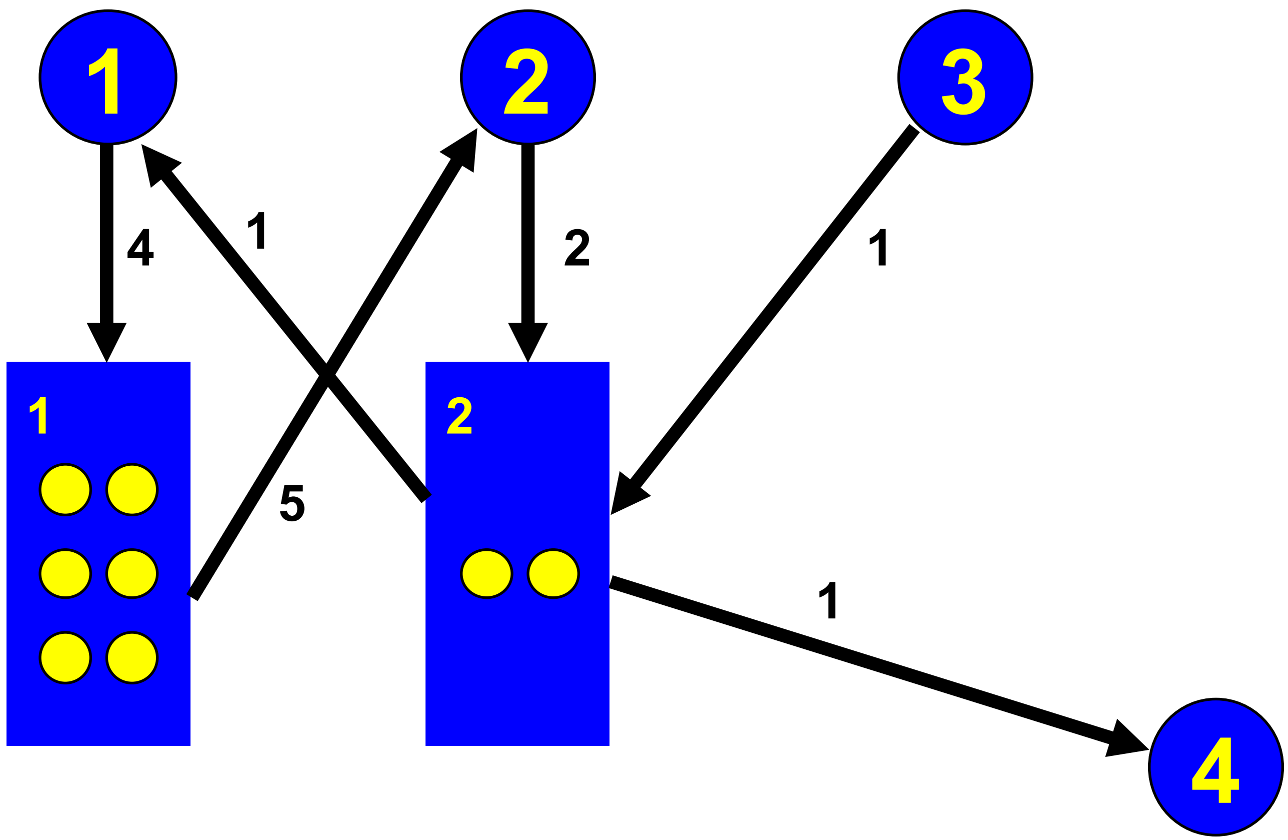


- **Edge from R_5 to P_6 :**

  "P_6 has *2* units of R_5"



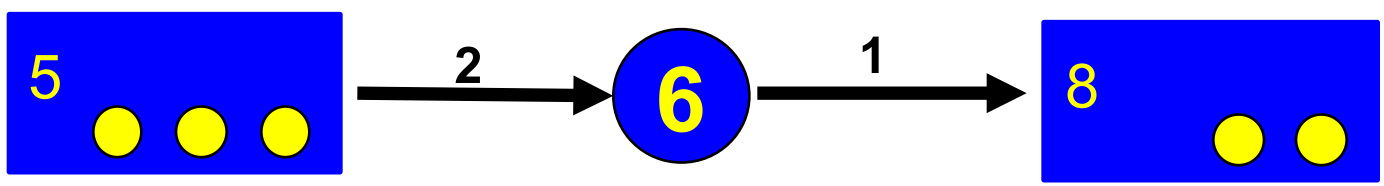
Example RAG



Reduction rules

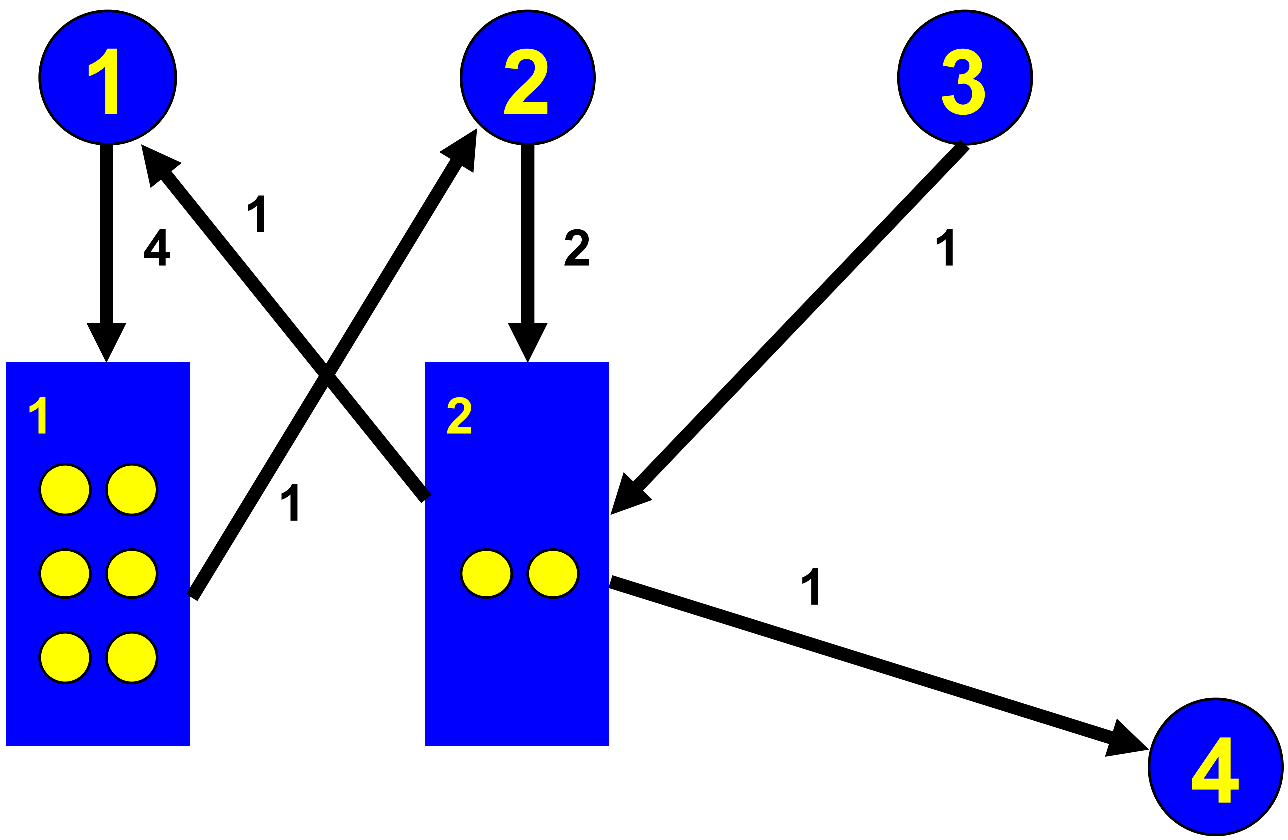
- Find satisfiable process P:
  - available amount of resource ≥ amount requested
- Erase P

  Intuition: Grant the request, let it run, eventually it will release the resource

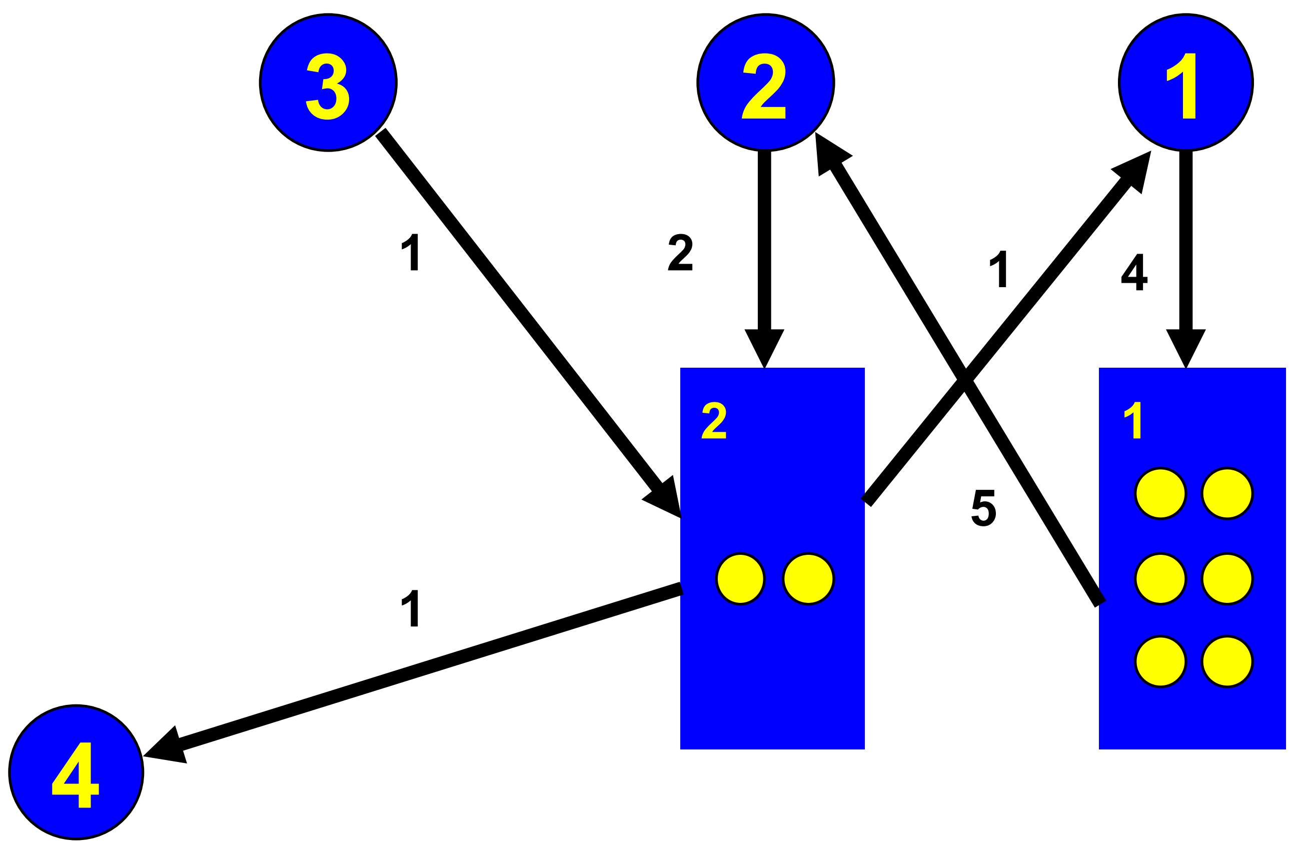


- Repeat until all processes gone (yay!) or irreducible (boo!)

Is this graph reducible?



Is this graph reducible?



Deadlock Detection Algorithm

Data structures:

n:	number of processes
m:	number of resource types
available[1..m]:	available[j] is #available resources of type j
allocation[1..n,1..m]:	current allocation of resource Rj to Pi
request[1..n,1..m]:	current demand of each Pi for each Rj

Deadlock 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

Example

Finished = {F, F, F, F};

Free = Available = (0, 0, 1);

	R_1	R_2	R_3
P_1	1	1	1
P_2	2	1	2
P_3	1	1	0
P_4	1	1	1

Allocation

	R_1	R_2	R_3
P_1	3	2	1
P_2	2	2	1
P_3	0	0	1
P_4	1	1	1

Request

Example

Finished = {F, F, T, F};

Free = (1, 1, 1);

	R_1	R_2	R_3
P_1	1	1	1
P_2	2	1	2
P_3			
P_4	1	1	1

Allocation

	R_1	R_2	R_3
P_1	3	2	1
P_2	2	2	1
P_3			
P_4	1	1	1

Request

Example

Finished = {F, F, T, T};

Free = (2, 2, 2);

	R_1	R_2	R_3
P_1	1	1	1
P_2	2	1	2
P_3			
P_4			

Allocation

	R_1	R_2	R_3
P_1	3	2	1
P_2	2	2	1
P_3			
P_4			

Request

Example

Finished = {F, T, T, T};

Free = (4, 3, 4);

	R_1	R_2	R_3
P_1	1	1	1
P_2			
P_3			
P_4			

Allocation

	R_1	R_2	R_3
P_1	3	2	1
P_2			
P_3			
P_4			

Request

Question 1 you might ask

Does order of reduction matter?

- Answer: **No.**

A candidate node for reduction at step i, and we don't pick it, remains a candidate for reduction at step i+1

Eventually—regardless of order—we'll reduce by every node where feasible

Question 2 you might ask

If a system is deadlocked, could the deadlock go away on its own?

- Answer: **No**, unless someone kills one of the threads or something causes a process to release a resource
- Many real systems put time limits on "waiting" precisely for this reason. When a process gets a timeout exception, it gives up waiting; this can eliminate the deadlock
- Process may be forced to terminate itself because often, if a process can't get what it needs, there are no other options available!

Question 3 you might ask

Suppose a system isn't deadlocked at time T. Can we assume it will still be free of deadlock at time T+1?

- Answer: **No**, because the very next thing it might do is to run some process that will request a resource…

  … establishing a cyclic wait

  … and causing deadlock

Dealing with Deadlocks (1)

Reactive Approaches:

- Periodically check for evidence of deadlock
  - (graph reduction algorithm)
- Need a way to recover
  - Could blue screen and reboot the computer
  - Could pick a "victim" and terminate that thread
    - Only possible in certain kinds of applications
  - Often thread "retry" from scratch

(despite drawbacks, database systems do this)

Dealing with Deadlocks (2)

Proactive Approaches:

- Deadlock Prevention & Avoidance
  - Prevent 1 of the 4 necessary conditions from arising
  - …. This will prevent deadlock from occurring

Deadlock Prevention

Deadlock Prevention

- Can the OS prevent deadlocks?
- Prevention: Negate one of necessary conditions

1. Mutual exclusion:
   - Make resources sharable without locks
   - Not always possible (printers, pinned memory for DMA)
2. Hold and wait
   - Do not hold resources when waiting for another

   ⇒ Request all resources before beginning execution

   – Processes do not know what resources they will need ahead of time

   – Starvation (if waiting on many popular resources)

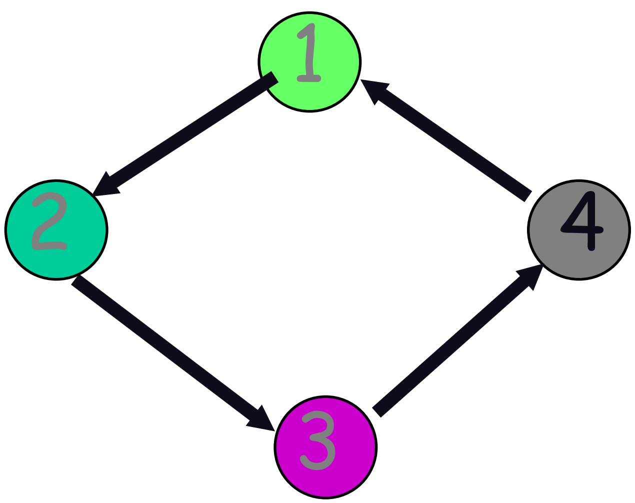
   – Low utilization (need resource only for a bit)
   - Optimization: Release all resources before requesting anything new
     - Still has the last two problems

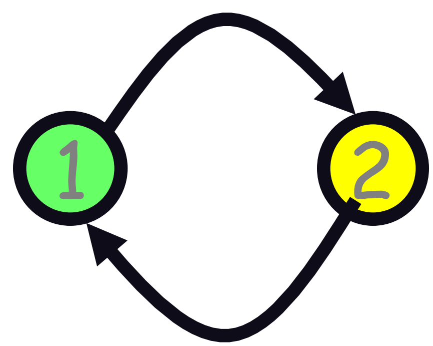
Deadlock Prevention

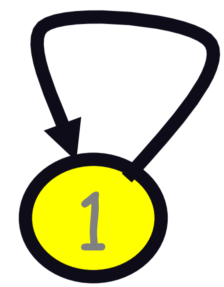
- Prevention cont'd: Negate one of necessary conditions
  3. No preemption:
     - Make resources preemptable (2 approaches)
       - Preempt requesting processes' resources if all not available
       - Preempt resources of waiting processes to satisfy request
     - Good when easy to save and restore state of resource
       - CPU registers, memory virtualization
  4. Circular wait: (2 approaches)
     - Single lock for entire system? (Problems)
     - Impose partial ordering on resources, request them in order

Deadlock Prevention

- Prevention: Breaking circular wait
  - Order resources (lock1, lock2, …)
  - Acquire resources in strictly increasing/decreasing order
  - Intuition: Cycle requires an edge from low to high, and from high to low numbered node, or to same node
  - Ordering not always easy…







Deadlock Avoidance

Deadlock Avoidance

- If we have future information
  - Max resource requirement of each process before they execute
- Can we guarantee that deadlocks will never occur?
- Avoidance Approach:
  - Before granting resource, check if resulting state is **safe**
  - If the state is safe $\Rightarrow$ no deadlock!
  - Otherwise, wait

Safe State

- A state is said to be **safe**, if there exists a sequence of processes [P_1, P_2,..., P_n] such that for each P_i the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all P_j where $j < i$

- State is safe because OS can definitely avoid deadlock
  - by blocking any new requests until safe order is executed

- This avoids circular wait condition from ever happening
  - Process waits until safe state is guaranteed

Safe State Example

- Suppose there are 12 tape drives and three processes, p0, p1, and p2

	max need	current usage	could ask for
p0	10	5	5
p1	4	2	2
p2	9	2	7

3 drives remain (12 - (5+2+2))

- current state is safe because a safe sequence exists: [p1, p0, p2]
  - p1 can complete with remaining resources
  - p0 can complete with remaining+p1
  - p2 can complete with remaining+p1+p0
- if p2 requests 1 drive, then it must wait to avoid unsafe state.

Banker’s Algorithm

- Suppose we know the “worst case” resource needs of processes in advance
  - A bit like knowing the credit limit on your credit cards. (This is why they call it the Banker’s Algorithm)
- Observation: Suppose we just give some process ALL the resources it could need…
  - Then it will execute to completion.
  - After which it will give back the resources.
- Like a bank: If Visa just hands you all the money your credit lines permit, at the end of the month, you’ll pay your entire bill, right?

Banker’s Algorithm

- So…
  - A process pre-declares its worst-case needs
  - Then it asks for what it “really” needs, a little at a time
  - The algorithm decides when to grant requests
- It delays a request unless:
  - It can find a sequence of processes…
  - …. such that it could grant their outstanding need…
  - … so they would terminate…
  - … letting it collect their resources…
  - … and in this way it can execute everything to completion!

Banker’s Algorithm

- How will it really do this?
  - The algorithm will just implement the graph reduction method for resource graphs
  - Graph reduction is “like” finding a sequence of processes that can be executed to completion
- So: given a request
  - Build a resource allocation graph assuming the request is granted
  - See if it is reducible, only actually grant request if so
  - Else must delay the request until someone releases some resources, at which point can test again

Banker's Algorithm

Dijkstra 1977

- Decides whether to grant a resource request.
- Data structures (similar to before):

n:	# of processes
m:	# of resource types
available[1..m]:	available[j] is # of avail resources of type j
max[1..n,1..m]:	max demand of each Pi for each Ri
allocation[1..n,1..m]:	current allocation of resource Rj to Pi
need[1..n,1..m]:	max # resource Rj that Pi may still request
	(*need = max – allocation*)

How to check safety?

free[1..m] = available /* how many resources are available */
finish[1..n] = false (for all i) /* none finished yet */

<u>Step 1:</u> Find a process i such that finish[i] = false and need[i] ≤ free
If f no such i exists, go to Step 3 /* we're done */

<u>Step 2:</u> Found an i:
finish [i] = true
free = free + allocation [i]
go to Step 1

<u>Step 3:</u> The system is safe iff finish[i] = true for all i,

Full Banker's Algorithm

Let process i be the next process that is scheduled to run

Let request[i] be vector of # of resource Rj Process Pi wants in addition to the resources it already has

1. If request[i] > need[i] then error (asked for too much)
2. If request[i] > available then wait (can't supply it now)
3. Resources are currently available to satisfy the request

   Let's tentatively assume that we satisfy the request. Then we would have:

   available = available - request[i]

   allocation[i] = allocation[i] + request[i]

   need[i] = need[i] - request[i]

   Now, check if this would leave us in a safe state:

   if yes, grant the request,

   if no, then leave the state as is and cause process to wait.

Banker's Algorithm: Example

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

this is a safe state:

safe sequence [P1, P3, P4, P2, P0]

Now suppose that P1 requests (1,0,2)

add it to P1's allocation

subtract it from Available

Banker's Algorithm: Example

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	2	3	0
P1	3	0	2	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

This is still safe: safe seq [P1, P3, P4, P0, P2].

In this new state, P4 requests (3,3,0)

- not enough available resources: has to wait

Now P0 requests (0,2,0)

- there are enough resources, but...

Banker's Algorithm: Example

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	3	0	7	5	3	2	1	0
P1	3	0	2	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

This is unsafe state (why?)

So P0 has to wait

Problems with Banker's Algorithm?

Problems with Bankers

- The number of processes is fixed
- Need to know how many resources each process will request ahead of time

The story so far..

- We saw that you can prevent deadlocks.
  - By negating one of the four necessary conditions. (which are..?)
- We saw that the OS can schedule processes in a careful way so as to avoid deadlocks.
  - By preventing circular waiting to ever occur

Deadlock Detection & Recovery

- If neither avoidance or prevention is implemented, deadlocks can (and will) occur.
- Coping with this requires:
  - Detection: finding out if deadlock has occurred
    - Keep track of resource allocation (who has what)
    - Keep track of pending requests (who is waiting for what)
  - Recovery: untangle the mess.
- Expensive to detect, as well as recover

When to run Detection Algorithm?

- For every resource request?
- For every request that cannot be immediately satisfied?
- Once every hour?
- When CPU utilization drops below 40%?
- Some combination of the last two?

Deadlock Recovery

- Killing one/all deadlocked processes
  - Crude, but effective
  - Keep killing processes, until deadlock broken
  - Repeat the entire computation
- Preempt resource/processes until deadlock broken
  - Selecting a victim (# resources held, how long executed)
  - Rollback (partial or total)
  - Starvation (prevent a process from being executed)