Deadlocks:
Detection & Avoidance

CS 4410, Operating Systems

Fall 2016
Cornell University

Rachit Agarwal
Anne Bracy

See: Ch 6 in OSPP textbook

The slides are the product of many rounds of teaching CS 4410 by Professors Sirer, Bracy, Agarwal, George, and Van Renesse.

System Model

There are non-shared computer resources

- 1+ instances (printers, semaphores, CPU, etc.)

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, waits for resource 2
- Process B acquires resource 2, waits for resource 1

➢ **Deadlock!**

Classic Deadlock

Example 1: Semaphores

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

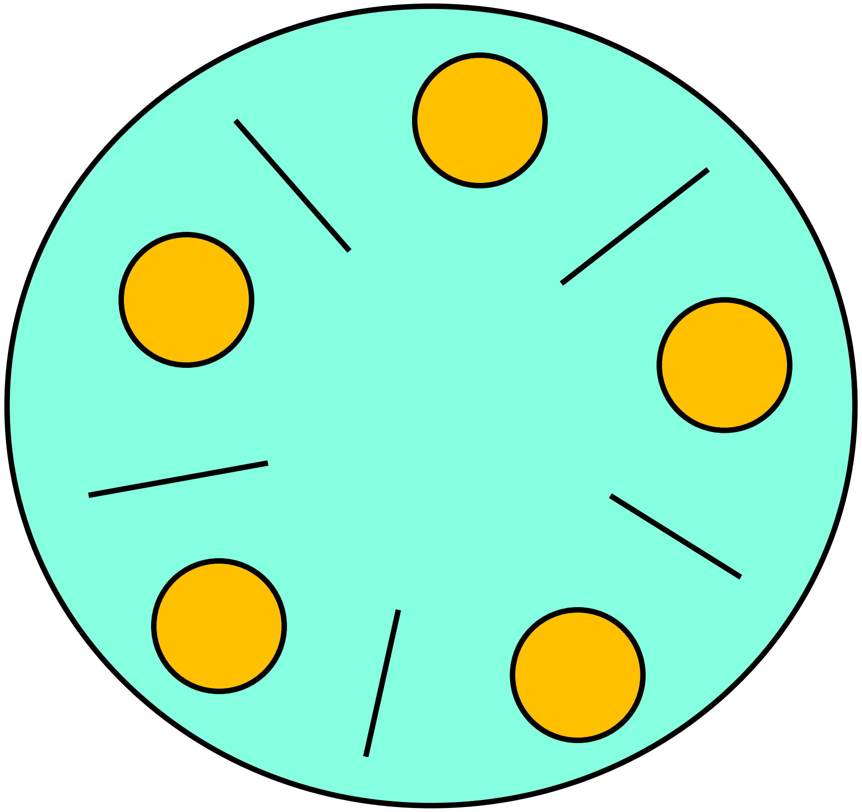
```
Process A code:               Process B code:
 {                             {
   /* initial compute */         /* initial compute */

   P(file_mutex)                 P(printer_mutex)
   P(printer_mutex)              P(file_mutex)

   /* use resources */           /* use resources */

   V(printer_mutex)              V(file_mutex)
   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:
      P(left)
      P(right)
       # om nom nom
      V(right)
      V(left)
```

- Philosophers go out for Chinese food
- Need exclusive access to 2 chopsticks to eat food

Starvation vs. Deadlock

Starvation: thread waits indefinitely

Deadlock: circular waiting for resources

  Deadlock => starvation, but not vice versa

Subject to deadlock ≠ will deadlock

- ➢ Testing is not the solution
- ➢ System must be deadlock-free *by design*

Four Conditions for Deadlock [Coffman 1971]

Necessary conditions for deadlock to exist:

- **Mutual Exclusion / Bounded Resources**

  At least one resource must be held in non-sharable mode
- **Hold and wait**

  ∃ a process holding a resource, and waiting for another
- **No preemption**
  - Resources cannot be preempted
- **Circular wait**
  - ∃ a set of processes $\{P_1, P_2, \ldots P_N\}$, such that

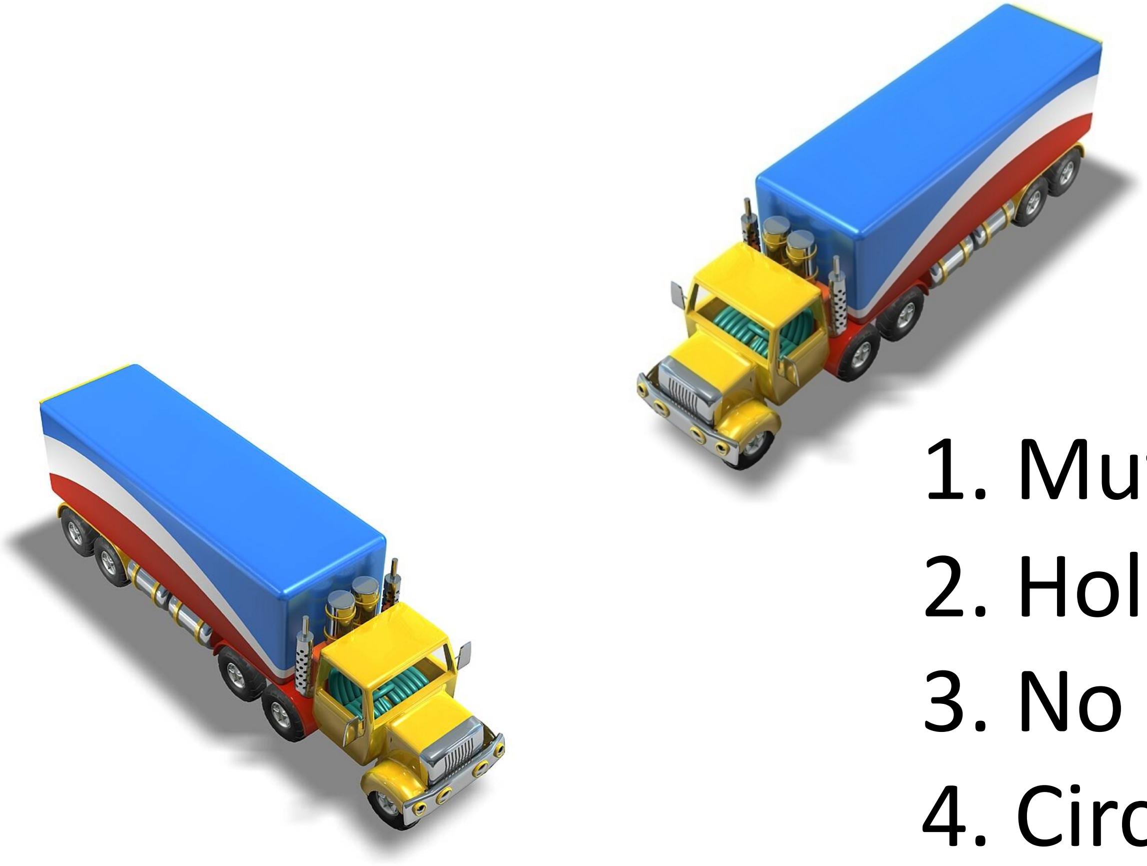
    P_1 is waiting for P_2, P_2 for P_3, …. and P_N for P_1

All four conditions must hold for deadlock to occur.

Note: not just about locks!

Is this a Deadlock?

Truck A has to wait for Truck B to move

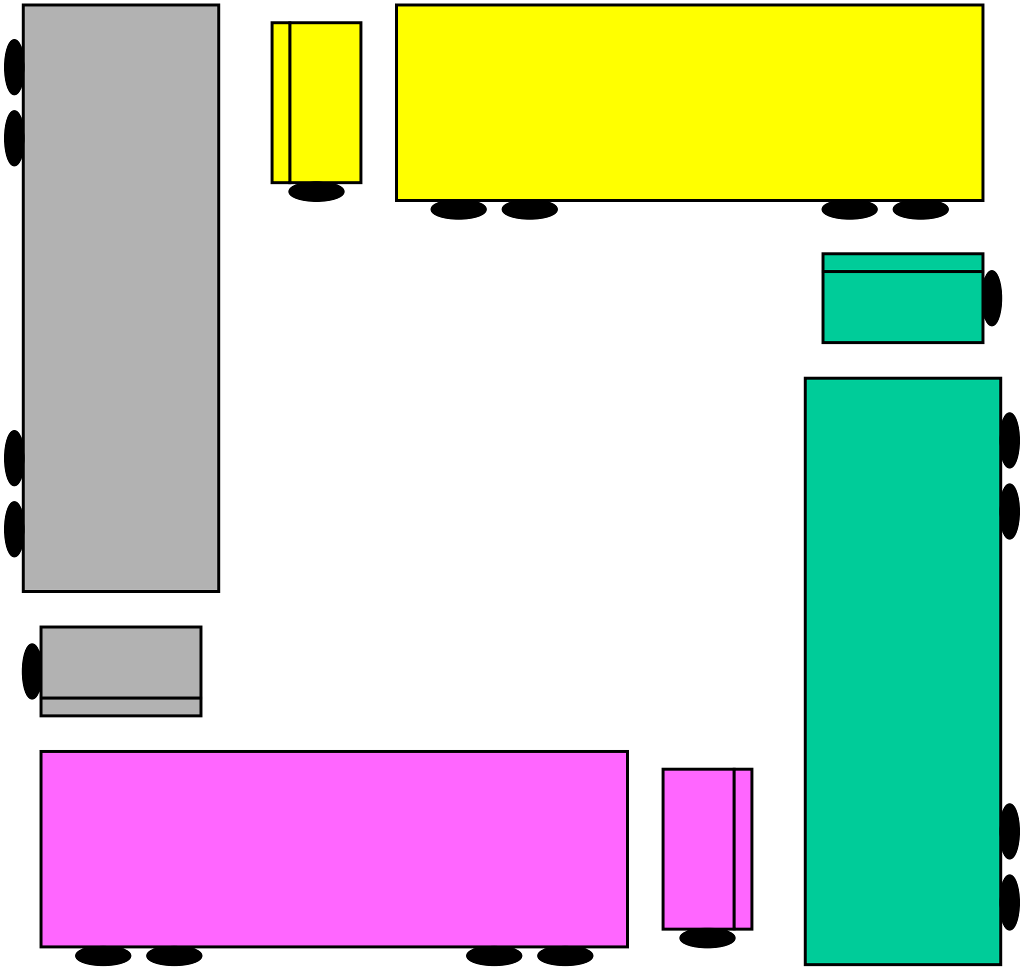


1. Mutual Exclusion
2. Hold & Wait
3. No Preemption
4. Circular Wait

Deadlock?

Is this a Deadlock?

Gridlock

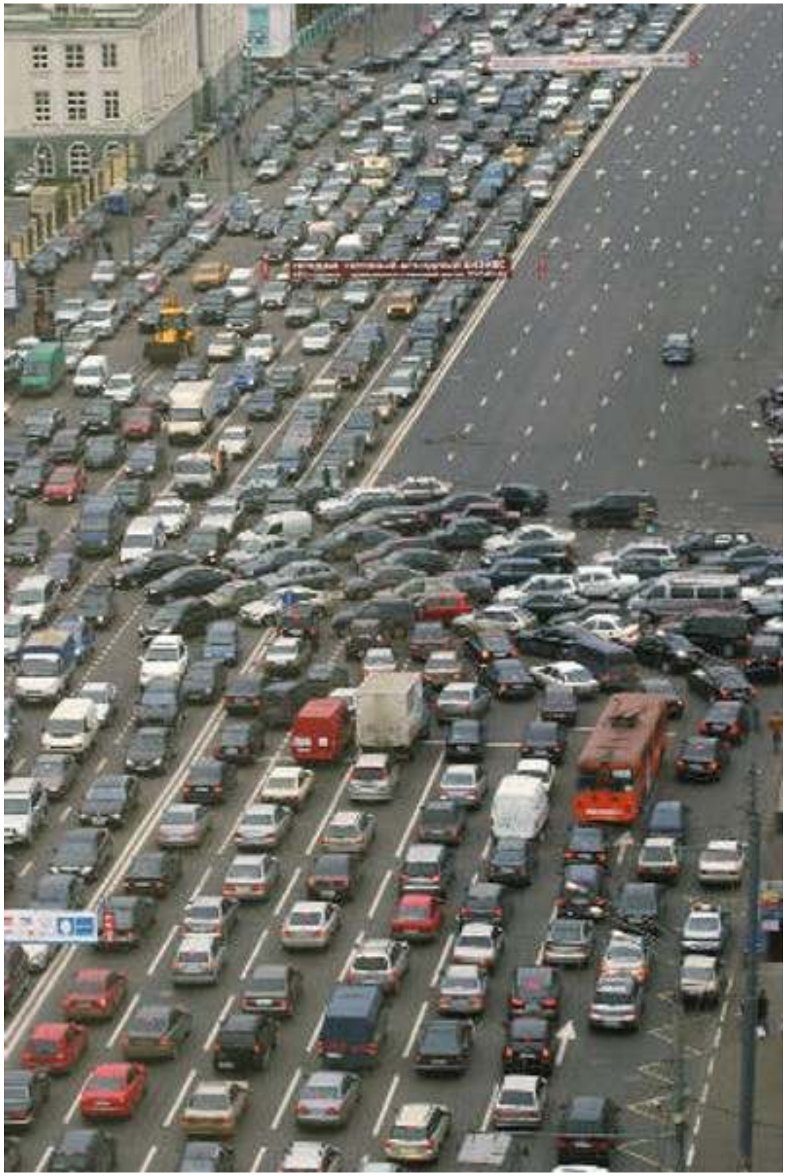


1. Mutual Exclusion
2. Hold & Wait
3. No Preemption
4. Circular Wait

Deadlock?

Is this a Deadlock?

Gridlock



1. Mutual Exclusion
2. Hold & Wait
3. No Preemption
4. Circular Wait

Deadlock?

Is this a Deadlock?

Gridlock

1. Mutual Exclusion
2. Hold & Wait
3. No Preemption
4. Circular Wait

Deadlock?

Deadlock Detection

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

Cycles in graph indicate deadlock

Testing for cycles (= deadlock)

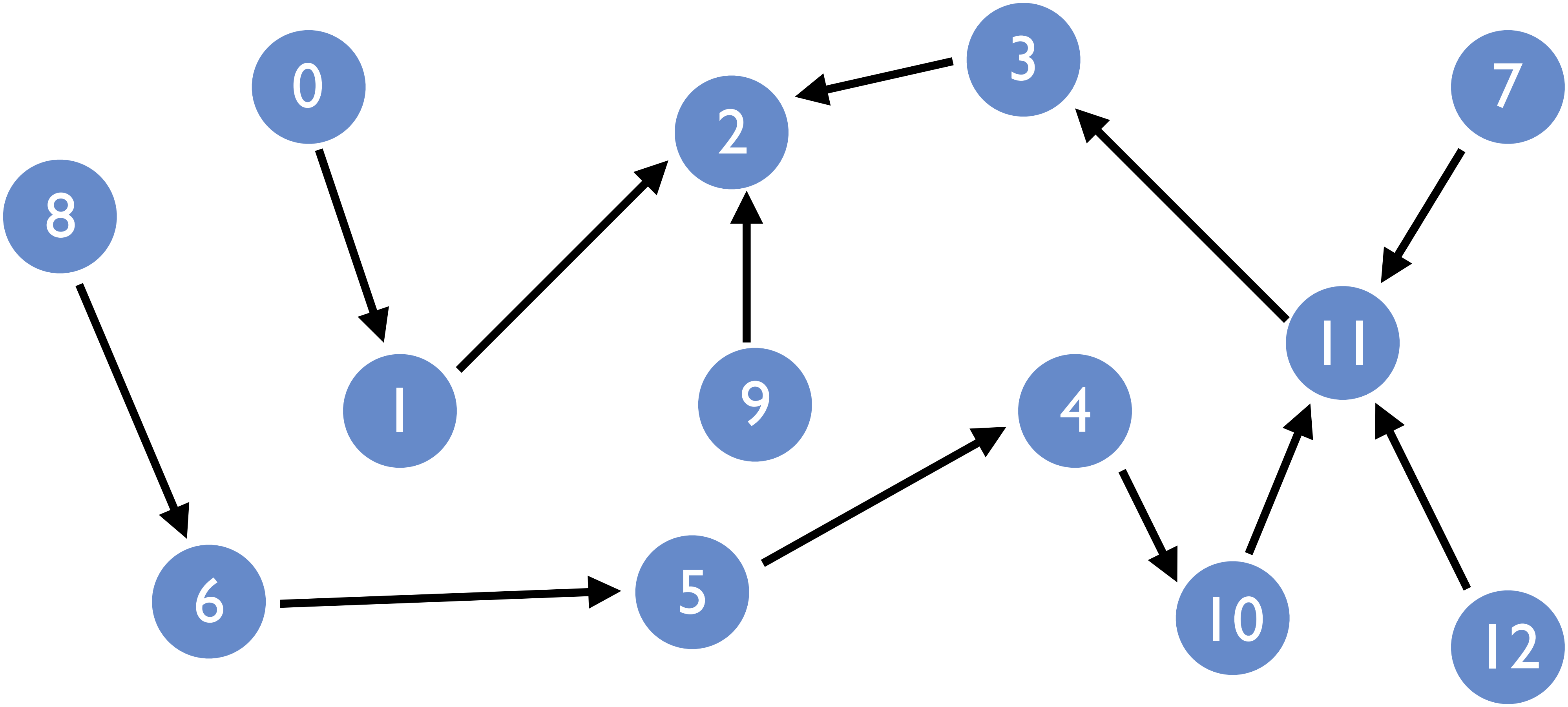
Find a node with no outgoing edges

- 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.

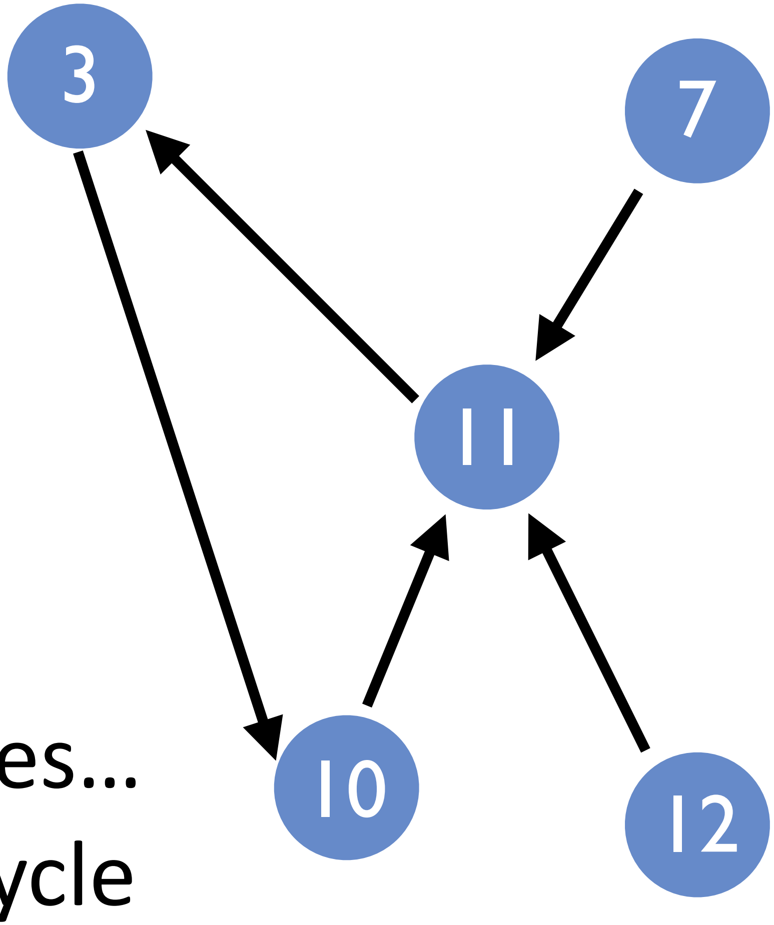
Erase whole graph ↔ graph has no cycles
Graph remains ↔ deadlock
This is a graph reduction algorithm.

Graph Reduction: Example 1



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 2



No node with no outgoing edges…
Irreducible graph, contains a cycle
(only some processes are in the cycle)
➝ 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

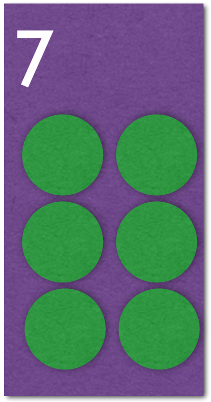
Can we extend our graphs to represent resource wait?

Resource Allocation Graphs

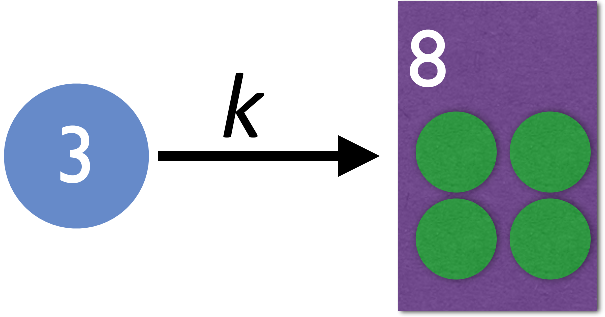
2 kinds of nodes

- A process: P_3 represented as 

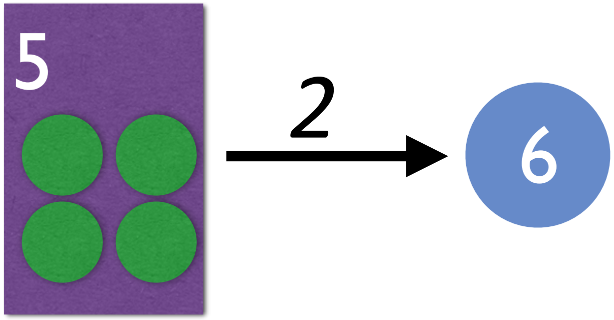
- A resource: R_7 represented as multiple identical units of the resource (e.g., blocks of memory) = circles in box



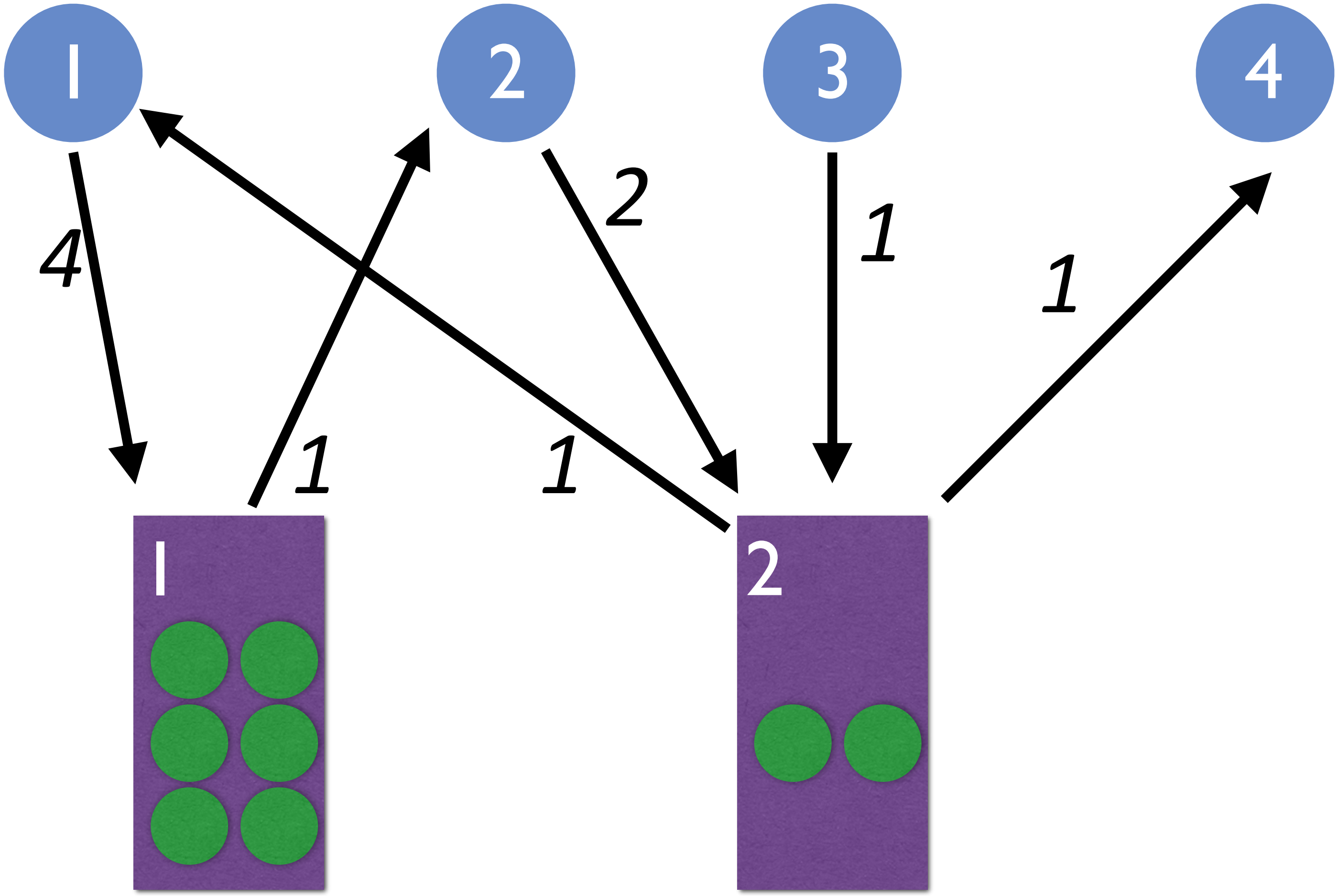
Edge from P_3 to R_8:
"P_3 wants k units of R_8"



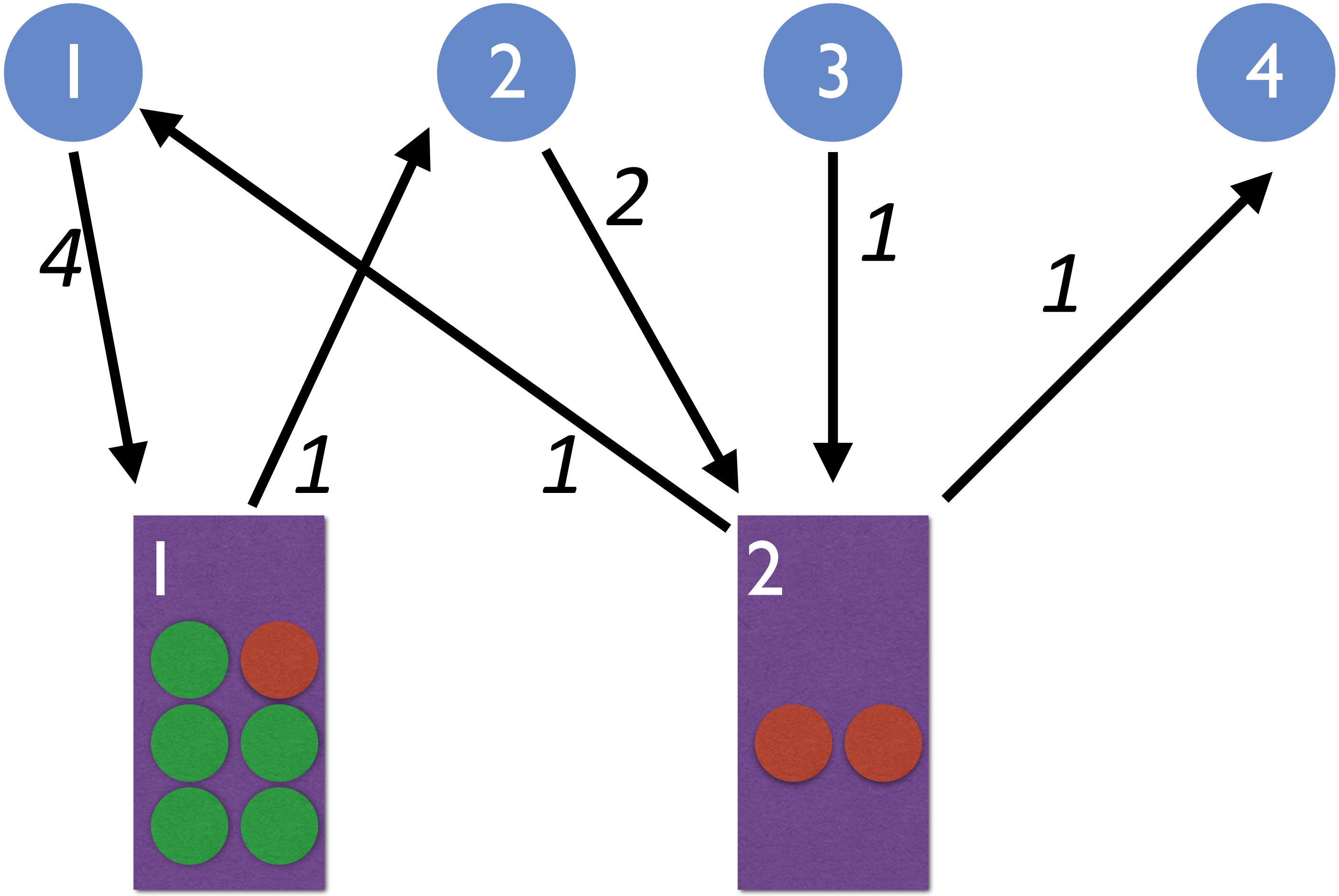
Edge from R_5 to P_6:
"P_6 has k units of R_5"



Example Resource Allocation Graph (RAG)



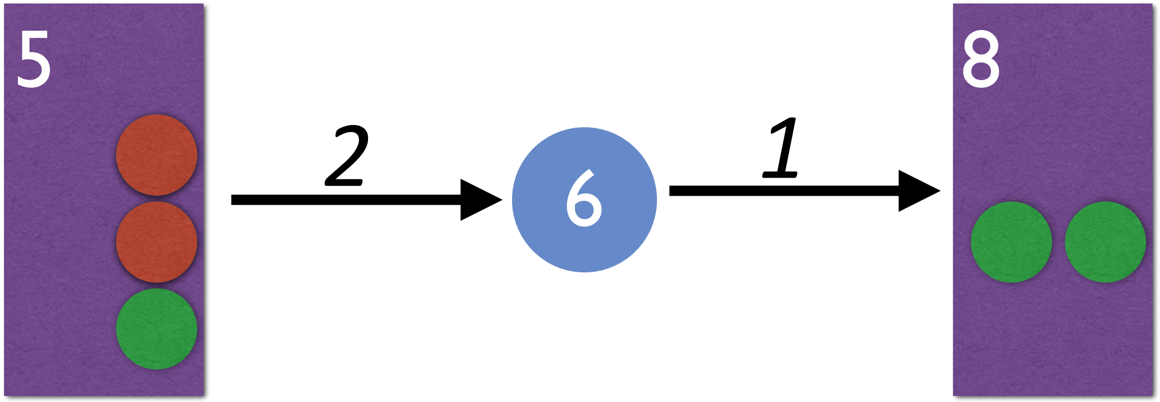
Example Resource Allocation Graph (RAG)



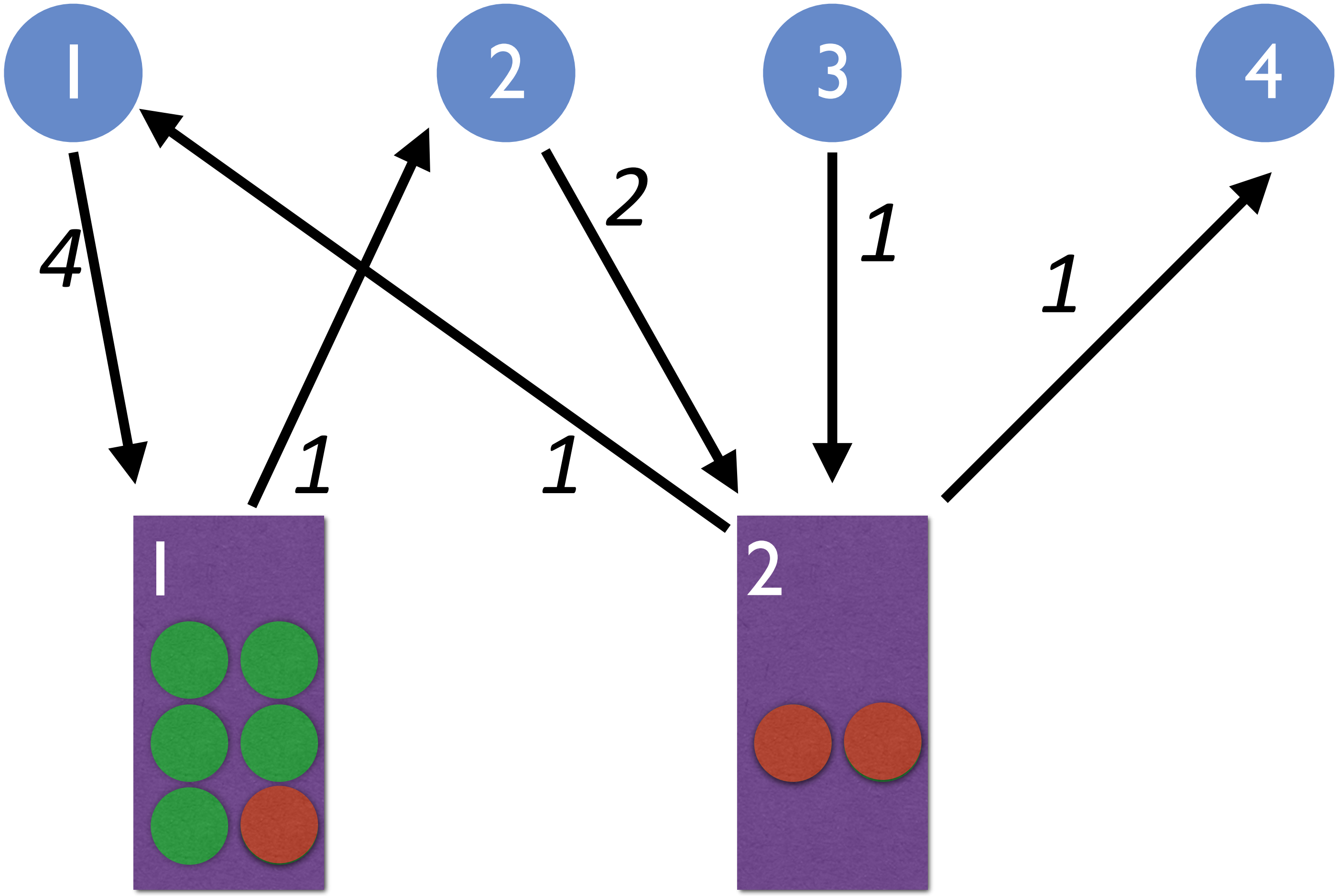
red is optional, but we think it helps…

Reduction Rules

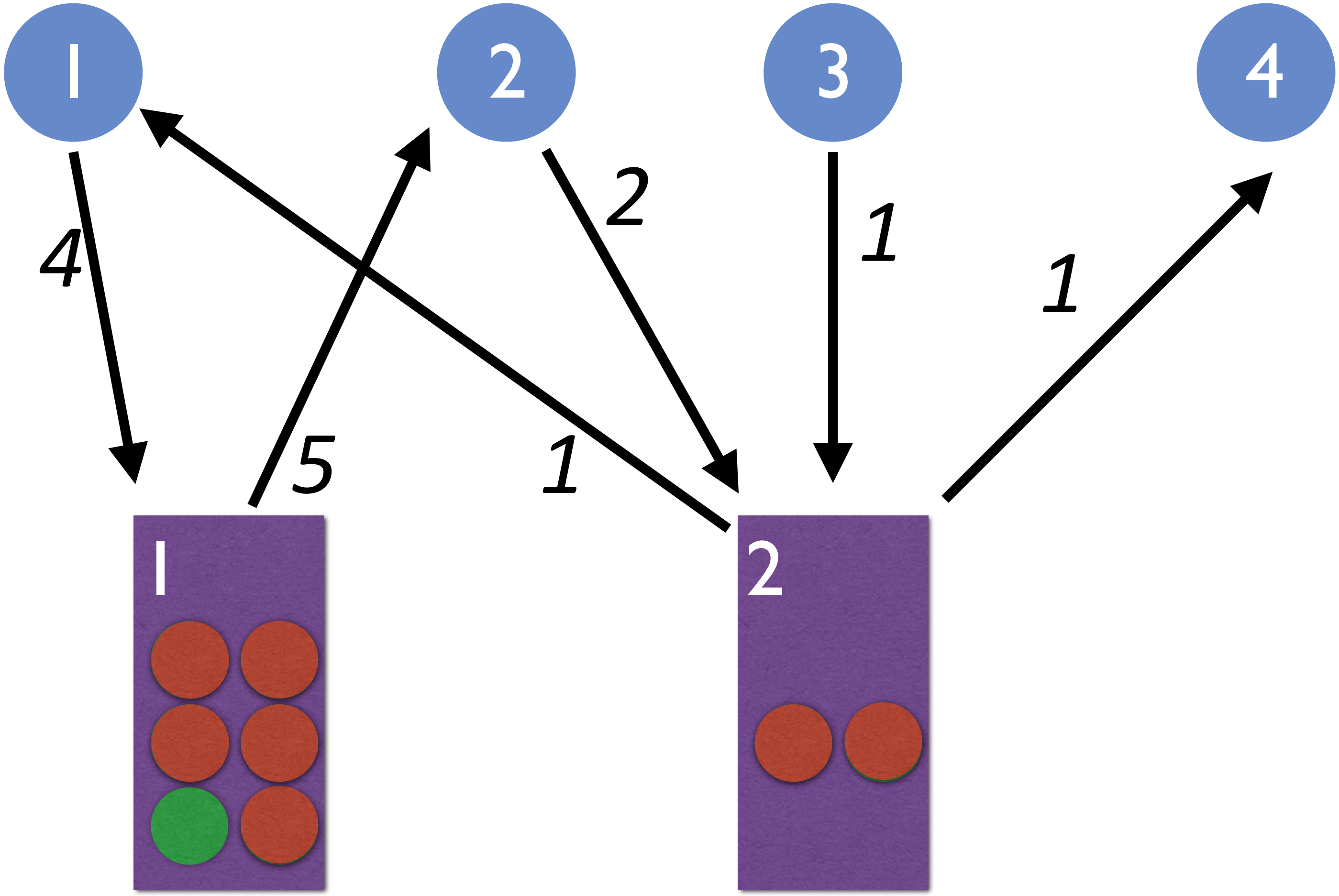
- Find satisfiable process P:
  available amount of resource ≥ amount requested
- Erase P
  Intuition: grant the request, let it run, eventually it will release the resource
- Repeat until:
  all processes gone (yay!) —or— irreducible (boo!)



Is this graph reducible?



Is this graph reducible?



Deadlock Detection Algorithm

Data structures:

`n:` number of processes

`m:` number of resource types

`avail[1..m]:` avail[j]: # of *currently* available type j resources

`alloc[n][m]:` current allocation of resource R_j to P_i

`req[n][m]:` current demand of each P_i for each R_j
(in addition to what has already been allocated)

```
1. free[] = avail[]
2. for all processes i: finish[i] = (alloc[i] == [0,0,…, 0])
3. find process i such that finish[i] = 0 and req[i] ≤ free
      if no such i exists, goto 7
4. free = free + alloc[i]
5. finish[i] = true
6. goto 3
7. system is deadlocked iff finish[i] = 0 for some process i
```

Example

`4 processes, 3 resource types,`
`avail[0,0,1]:     1 type-2 resource available`

allocation

	R_0	R_1	R_2
P_0	1	1	1
P_1	2	1	2
P_2	1	1	0
P_3	1	1	1

request

	R_0	R_1	R_2
P_0	3	2	1
P_1	2	2	1
P_2	0	0	1
P_3	1	1	1

`free[0,0,1]`

```
1. free[] = avail[]
2. for all processes i: finish[i] = (alloc[i] == [0,0,…, 0])
3. find process i such that finish[i] =0 and req[i] ≤ free[i]
      if no such i exists, goto 7
4. free = free + alloc[i]
5. finish[i] = true
6. goto 3
7. system is deadlocked iff finish[i] = 0 for some process i
```

Example

```
4 processes, 3 resource types,
avail[0,0,1]:     1 type-2 resource available
```

allocation

	R_0	R_1	R_2
P_0	1	1	1
P_1	2	1	2
P_2	1	1	0
P_3	1	1	1

request

	R_0	R_1	R_2
P_0	3	2	1
P_1	2	2	1
P_2	0	0	1
P_3	1	1	1

```
free[0,0,1]
finish[0,0,0,0]
```

```
1. free[] = avail[]
2. for all processes i: finish[i] = (alloc[i] == [0,0,…, 0])
3. find process i such that finish[i] =0 and req[i] ≤ free[i]
      if no such i exists, goto 7
4. free = free + alloc[i]
5. finish[i] = true
6. goto 3
7. system is deadlocked iff finish[i] = 0 for some process i
```

What about a process with a request that currently has nothing allocated? Since it holds no resources it will not participate in the hold-and-wait deadlock circle, so we ignore it.

Example

`4 processes, 3 resource types,`
`avail[0,0,1]:     1 type-2 resource available`

allocation

	R_0	R_1	R_2
P_0	1	1	1
P_1	2	1	2
P_2	1	1	0
P_3	1	1	1

request

	R_0	R_1	R_2
P_0	3	2	1
P_1	2	2	1
P_2	0	0	1
P_3	1	1	1

`free[0,0,1]`
`finish[0,0,0,0]`

```
1. free[] = avail[]
2. for all processes i: finish[i] = (alloc[i] == [0,0,…, 0])
3. find process i such that finish[i] =0 and req[i] ≤ free[i]
      if no such i exists, goto 7
4. free = free + alloc[i]
5. finish[i] = true
6. goto 3
7. system is deadlocked iff finish[i] = 0 for some process i
```

Example

4 processes, 3 resource types,
avail[0,0,1]: 1 type-2 resource available

allocation

	R_0	R_1	R_2
P_0	1	1	1
P_1	2	1	2
P_2	1	1	0
P_3	1	1	1

request

	R_0	R_1	R_2
P_0	3	2	1
P_1	2	2	1
P_2	-	-	-
P_3	1	1	1

free[1,1,1]
finish[0,0,0,0]

```
1. free[] = avail[]
2. for all processes i: finish[i] = (alloc[i] == [0,0,…, 0])
3. find process i such that finish[i] =0 and req[i] ≤ free[i]
      if no such i exists, goto 7
4. free = free + alloc[i]
5. finish[i] = true
6. goto 3
7. system is deadlocked iff finish[i] = 0 for some process i
```

Example

4 processes, 3 resource types,
avail[0,0,1]:   1 type-2 resource available

allocation

	R_0	R_1	R_2
P_0	1	1	1
P_1	2	1	2
P_2	-	-	-
P_3	1	1	1

request

	R_0	R_1	R_2
P_0	3	2	1
P_1	2	2	1
P_2	-	-	-
P_3	1	1	1

free[1,1,1]
finish[0,0,1,0]

```
1. free[] = avail[]
2. for all processes i: finish[i] = (alloc[i] == [0,0,…, 0])
3. find process i such that finish[i] =0 and req[i] ≤ free[i]
      if no such i exists, goto 7
4. free = free + alloc[i]
5. finish[i] = true
6. goto 3
7. system is deadlocked iff finish[i] = 0 for some process i
```

Example

4 processes, 3 resource types,
avail[0,0,1]:   1 type-2 resource available

allocation

	R_0	R_1	R_2
P_0	1	1	1
P_1	2	1	2
P_2	-	-	-
P_3	1	1	1

request

	R_0	R_1	R_2
P_0	3	2	1
P_1	2	2	1
P_2	-	-	-
P_3	1	1	1

free[1,1,1]
finish[0,0,1,0]

```
1. free[] = avail[]
2. for all processes i: finish[i] = (alloc[i] == [0,0,…, 0])
3. find process i such that finish[i] =0 and req[i] ≤ free[i]
      if no such i exists, goto 7
4. free = free + alloc[i]
5. finish[i] = true
6. goto 3
7. system is deadlocked iff finish[i] = 0 for some process i
```

Example

```
4 processes, 3 resource types,
avail[0,0,1]:     1 type-2 resource available
```

allocation

	R_0	R_1	R_2
P_0	1	1	1
P_1	2	1	2
P_2	-	-	-
P_3	1	1	1

request

	R_0	R_1	R_2
P_0	3	2	1
P_1	2	2	1
P_2	-	-	-
P_3	-	-	-

```
free[2,2,2]
finish[0,0,1,0]
```

```
1. free[] = avail[]
2. for all processes i: finish[i] = (alloc[i] == [0,0,…, 0])
3. find process i such that finish[i] =0 and req[i] ≤ free[i]
      if no such i exists, goto 7
4. free = free + alloc[i]
5. finish[i] = true
6. goto 3
7. system is deadlocked iff finish[i] = 0 for some process i
```

Example

4 processes, 3 resource types,
avail[0,0,1]:   1 type-2 resource available

allocation

	R_0	R_1	R_2
P_0	1	1	1
P_1	2	1	2
P_2	-	-	-
P_3	-	-	-

request

	R_0	R_1	R_2
P_0	3	2	1
P_1	2	2	1
P_2	-	-	-
P_3	-	-	-

free[2,2,2]
finish[0,0,1,1]

```
1. free[] = avail[]
2. for all processes i: finish[i] = (alloc[i] == [0,0,…, 0])
3. find process i such that finish[i] =0 and req[i] ≤ free[i]
      if no such i exists, goto 7
4. free = free + alloc[i]
5. finish[i] = true
6. goto 3
7. system is deadlocked iff finish[i] = 0 for some process i
```

Example

4 processes, 3 resource types,
avail[0,0,1]: 1 type-2 resource available

allocation

	R_0	R_1	R_2
P_0	1	1	1
P_1	2	1	2
P_2	-	-	-
P_3	-	-	-

request

	R_0	R_1	R_2
P_0	3	2	1
P_1	2	2	1
P_2	-	-	-
P_3	-	-	-

free[2,2,2]
finish[0,0,1,1]

```
1. free[] = avail[]
2. for all processes i: finish[i] = (alloc[i] == [0,0,…, 0])
3. find process i such that finish[i] =0 and req[i] ≤ free[i]
       if no such i exists, goto 7
4. free = free + alloc[i]
5. finish[i] = true
6. goto 3
7. system is deadlocked iff finish[i] = 0 for some process i
```

Example

`4 processes, 3 resource types,`
`avail[0,0,1]:     1 type-2 resource available`

allocation

	R_0	R_1	R_2
P_0	1	1	1
P_1	-	-	-
P_2	-	-	-
P_3	-	-	-

request

	R_0	R_1	R_2
P_0	3	2	1
P_1	-	-	-
P_2	-	-	-
P_3	-	-	-

`free[4,3,4]`
`finish[0,1,1,1]`

```
1. free[] = avail[]
2. for all processes i: finish[i] = (alloc[i] == [0,0,…, 0])
3. find process i such that finish[i] =0 and req[i] ≤ free[i]
      if no such i exists, goto 7
4. free = free + alloc[i]
5. finish[i] = true
6. goto 3
7. system is deadlocked iff finish[i] = 0 for some process i
```

Example

```
4 processes, 3 resource types,
avail[0,0,1]:     1 type-2 resource available
```

allocation

	R_0	R_1	R_2
P_0	1	1	1
P_1	-	-	-
P_2	-	-	-
P_3	-	-	-

request

	R_0	R_1	R_2
P_0	3	2	1
P_1	-	-	-
P_2	-	-	-
P_3	-	-	-

```
free[4,3,4]
finish[0,1,1,1]
```

```
1. free[] = avail[]
2. for all processes i: finish[i] = (alloc[i] == [0,0,…, 0])
3. find process i such that finish[i] =0 and req[i] ≤ free[i]
      if no such i exists, goto 7
4. free = free + alloc[i]
5. finish[i] = true
6. goto 3
7. system is deadlocked iff finish[i] = 0 for some process i
```

Question #1

Does order of reduction matter?

Answer: No.
Explanation: an unchosen candidate at one step remains a candidate for later steps. Eventually—regardless of order—every node will be reduced.

Question #2

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.
Explanation: 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

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

Explanation: 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
  - Blue screen and reboot the computer
  - Pick a "victim" and terminate that thread
    (Only possible in certain kinds of applications)
  - Have threads "retry" from scratch

(despite drawbacks, database systems do this)

Dealing with Deadlocks (2)

Proactive Approaches:

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

Deadlock Prevention : negate 1 of the 4

1. Mutual exclusion / Bounded Resources:

- Make resources sharable without locks?
- Make more resources available?
- Example: reserve space in TCB for thread to be inserted into a waiting list or the ready list.
- Not always possible (e.g., printers)

Deadlock Prevention : negate 1 of the 4

2. Hold and wait

Don’t hold resources when waiting for another

- re-write code: have these 2 fns acquire/release

```
Module:: foo()  {
  lock.acquire();
  doSomeStuff();
  otherModule->bar();
  doOtherStuff();
  lock.release(); }
```

→

```
Module:: foo()  {
    doSomeStuff();
    otherModule->bar();
    doOtherStuff();
}
```

- Request all resources before execution begins
  - Processes don’t know what they 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 last two problems 😞

Deadlock Prevention : negate 1 of the 4

3. No preemption:

- Make resources pre-emptable by runtime system
  1. Preempt requesting processes’ resources if all not available
  2. Preempt resources of waiting processes to satisfy request

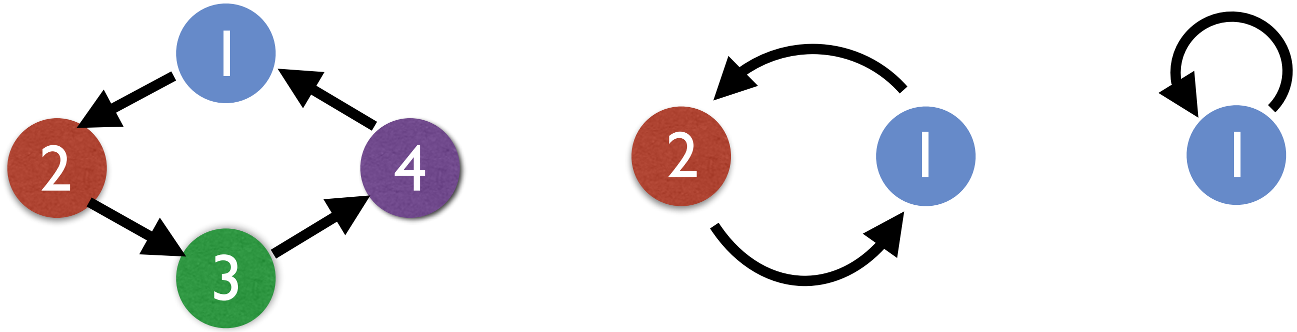
- Good when easy to save and restore state of resource
  - CPU registers
  - memory virtualization (page memory to disk, maybe even page tables)

Deadlock Prevention : negate 1 of the 4

4. Circular Wait

- Single lock for entire system?
- Impose partial ordering on resources, request in order

  Intuition: Cycle requires an edge from low to high, and from high to low numbered node, or to same node



Preventing Dining Philosophers Deadlock?

1. **Mutual Exclusion / Bounded Resources**

2. **Hold and wait**

3. **No preemption**

4. **Circular wait**

Deadlock Avoidance

How do cars do it?

- Try not to block an intersection
- Don’t drive into the intersection if you can see that you’ll be stuck there.

Why does this work?

- Prevents a wait-for relationship
- Cars won’t take up a resource if they see they won’t be able to acquire the next one…

Deadlock Dynamics

Safe state:

- For any possible sequence of future resource requests, it is possible to eventually grant all requests
- May require waiting even when resources are available!

Unsafe state:

- Some sequence of resource requests can result in deadlock

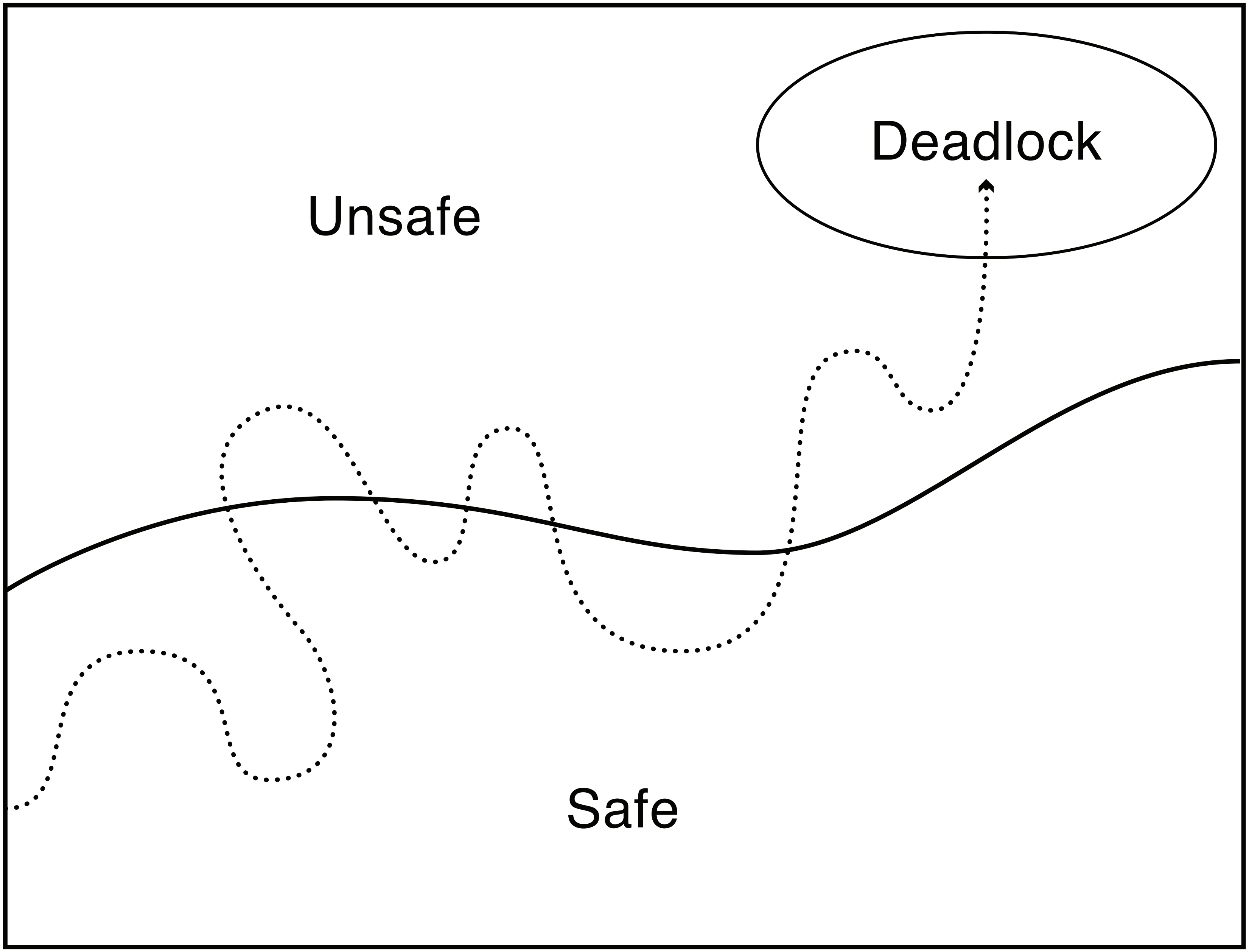
Doomed state:

- All possible computations lead to deadlock

Deadlocked state:

- System has at least one deadlock

Possible System States



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 b/c OS can definitely avoid deadlock
    - by blocking new requests until safe order is executed

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

Safe State Example

Suppose: 12 tape drives and 3 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

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

What if p2 requests 1 drive? Grant or not?

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.
- Hmmm, if Visa hands you all the money your credit lines permit, at the end of the month, will you pay your entire bill?

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 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:` number of processes

`m:` number of resource types

`avail[m]:` avail[j]: # of currently available type j resources

`max[n][m]:` max demand of each P_i for each R_i

`alloc[n][m]:` current allocation of resource R_j to P_i

`need[n][m]:` max # resource R_j that P_i may still request
  (need = max – allocation)

algorithm-internal state:

`finish[n]` – which processors are finished in this scenario

`free[m]` – which resources are available inside path

How to check safety?

```
free[1..m] = available       /* how many resources available */
finish[1..n] = [0..0]        /* none finished yet */

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

Step 2: Found an i:
         finish [i] = 1
         free = free + alloc[i]
         go to Step 1

Step 3: The system is safe iff finish[i] = 1 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 $R_j$ Process $P_i$ 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.
   Tentatively assume we satisfy the request.
Then we would have:
      available = available - request[i]
      alloc[i] = alloc[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 state as is & cause process to wait
```

Banker's Algorithm

State 1

allocation

	A	B	C
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

max

A	B	C
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

available

A	B	C
3	3	2

Is State 1 a safe state?

Is there a sequence of granting processors resources that satisfies everyone?

Banker’s Algorithm

State 1

allocation

	A	B	C
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

max

A	B	C
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

available

A	B	C
3	3	2

State 1 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

State 2

allocation

	A	B	C
P_0	0	1	0
P_1	3	0	2
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

max

A	B	C
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

available

A	B	C
2	3	0

Is State 2 a safe state?

Is there a sequence of granting processors resources that satisfies everyone?

Banker's Algorithm

State 2

allocation

	A	B	C
P_0	0	1	0
P_1	3	0	2
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

max

A	B	C
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

available

A	B	C
2	3	0

State 2 is still safe: safe seq [P1, P3, P4, P0, P2].

Now suppose P4 requests (3,3,0)

- not enough available resources: has to wait

Banker's Algorithm

State 2

allocation

	A	B	C
P_0	0	1	0
P_1	3	0	2
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

max

A	B	C
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

available

A	B	C
2	3	0

State 2 is still safe: safe seq [P1, P3, P4, P0, P2].

Now suppose P0 requests (0,2,0)

- have enough resources, but, hypothetically…

  add it to P0's allocation

  subtract it from Available

Banker's Algorithm

State 3

allocation

	A	B	C
P_0	0	3	0
P_1	3	0	2
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

max

A	B	C
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

available

A	B	C
2	1	0

Is State 3 a safe state?

Is there a sequence of granting processors resources that satisfies everyone?

Banker’s Algorithm

State 3

allocation

	A	B	C
P_0	0	3	0
P_1	3	0	2
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

max

A	B	C
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

available

A	B	C
2	1	0

State 3 is unsafe state (why?)

So P0 has to wait

Problems with Bankers

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

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 the Detection Algorithm?

- For every resource request?
- For every request not immediately satisfiable?
- 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)

The Story So Far

We saw that you can prevent deadlocks.

- By negating one of the four necessary conditions.

We saw that the OS can schedule processes in a careful way so as to avoid deadlocks.

- By preventing circular waiting to ever occur

We discussed options when deadlock has occurs.

The discussion continues…

Transactions / Transactional Memory

- Programming simplicity of coarse-grain locks
- Higher concurrency (parallelism) of fine-grain locks
- Critical sections only serialized if data is actually shared
- No lock acquisition overhead

Transactional Memory

Big idea I: no locks, just shared data

Big idea II: optimistic (speculative) concurrency

- Execute critical section speculatively, abort on conflicts
- "Better to beg for forgiveness than to ask for permission"

Read set: set of shared addresses critical section reads

Example: **`accts[37].bal, accts[241].bal`**

Write set: set of shared addresses critical section writes

Example: **`accts[37].bal, accts[241].bal`**

begin_transaction

- Take a local register checkpoint
- Locally track read set (remember addresses you read)
- See if anyone else is trying to write it
- Locally buffer all of your writes (invisible to other processors)
- Local actions only: no lock acquire

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
end_transaction();
```

end_transaction

- Check read set: is data you read still valid (i.e., no writes to any)
  - **Yes?** Commit transactions: commit writes
  - **No?** Abort transaction: restore checkpoint

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
end_transaction();
```