Deadlocks: Detection & Avoidance

CS 4410
Operating Systems

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

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

System Model

Exclusive (one-at-a-time) computer resources

- printers, CPU, memory, shared region to update,
- 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:

```
{
   /* initial compute */

   P(file_mutex)
   P(printer_mutex)

   /* use resources */

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

Process B code:

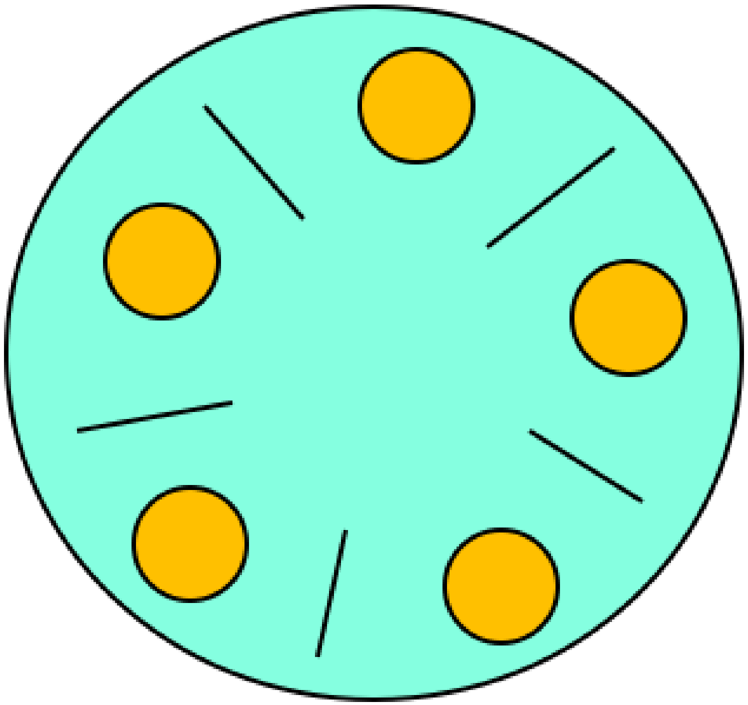
```
{
   /* initial compute */

   P(printer_mutex)
   P(file_mutex)

   /* use 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
   left = (self.id+1) % N
   while True:
         P(chopsticks[left])
         P(chopsticks[right])
         # om nom nom
         V(chopsticks[right])
         V(chopsticks[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 $\Rightarrow$ starvation, but not vice versa

Subject to deadlock $\neq$ will deadlock

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

Four Conditions for Deadlock

Necessary conditions for deadlock to exist:

(1) Mutual Exclusion / Bounded Resources

≥ 1 resource must be held in non-sharable mode

(2) Hold and wait

$\exists$ a process holding 1 resource & waiting for another

(3) No preemption

Resources cannot be preempted

(4) Circular wait

$\exists$ 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

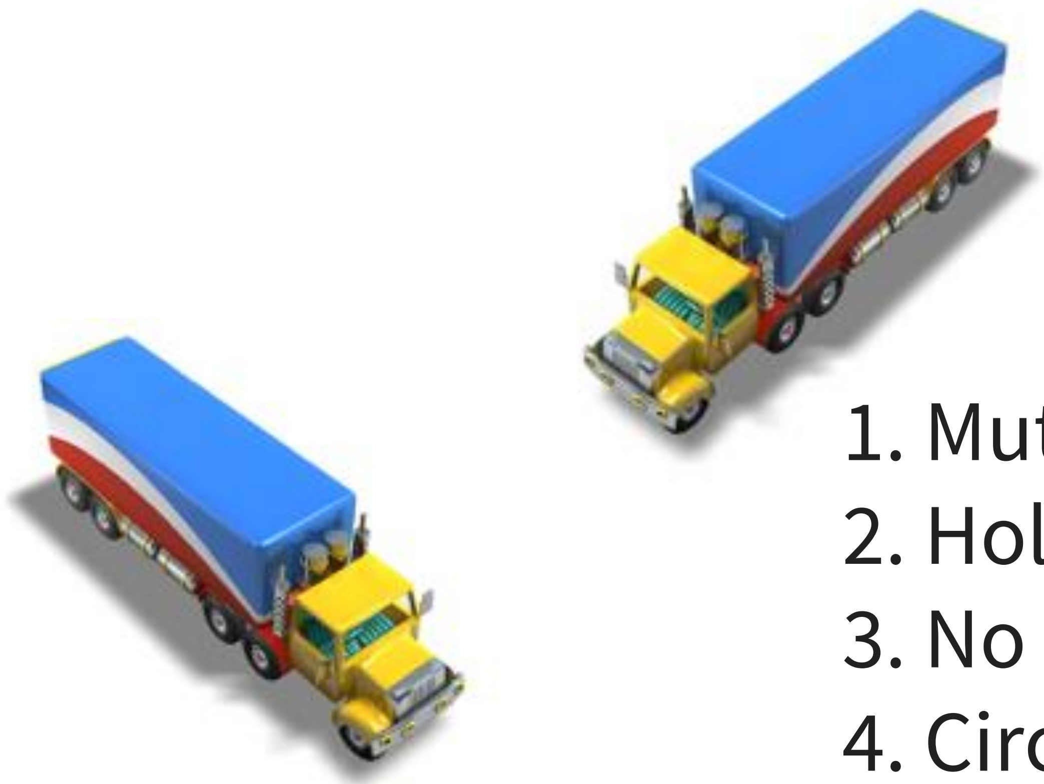
ALL FOUR must hold for deadlock to occur.

Note: it's not just about locks!

[Coffman 1971]

Is this a Deadlock?

Truck A has to wait for Truck B to move

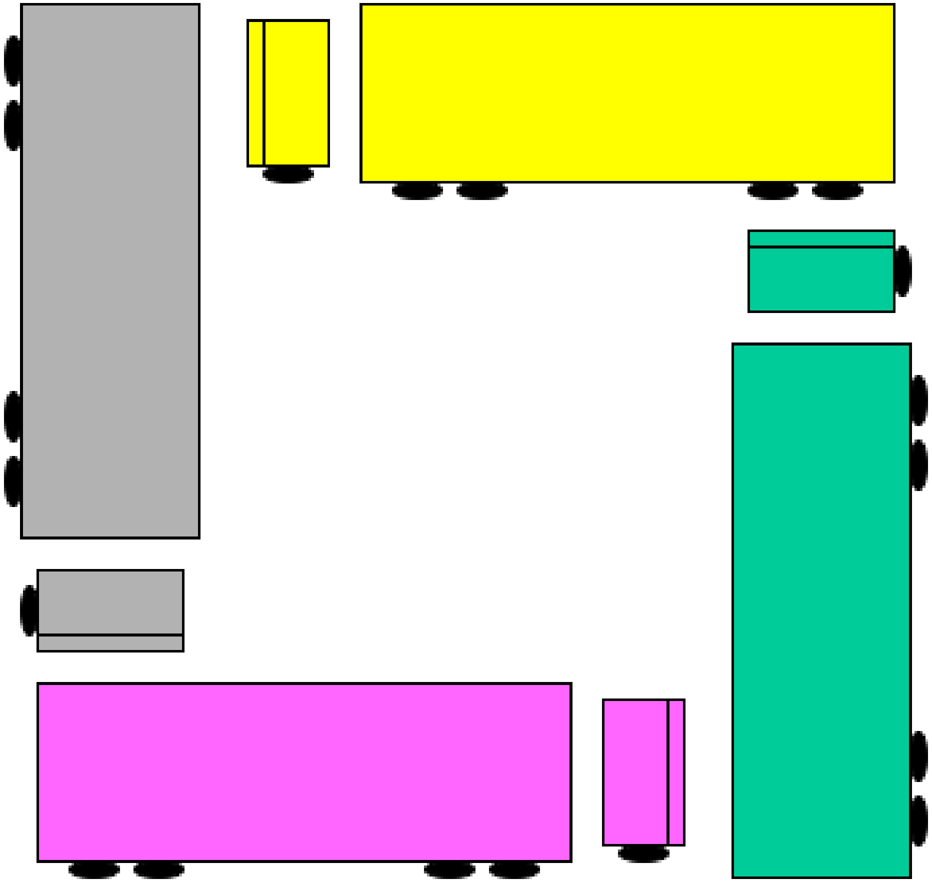


1. Mutual Exclusion
2. Hold & Wait
3. No Preemption
4. Circular Wait

Deadlock?

Is this a Deadlock?

Gridlock

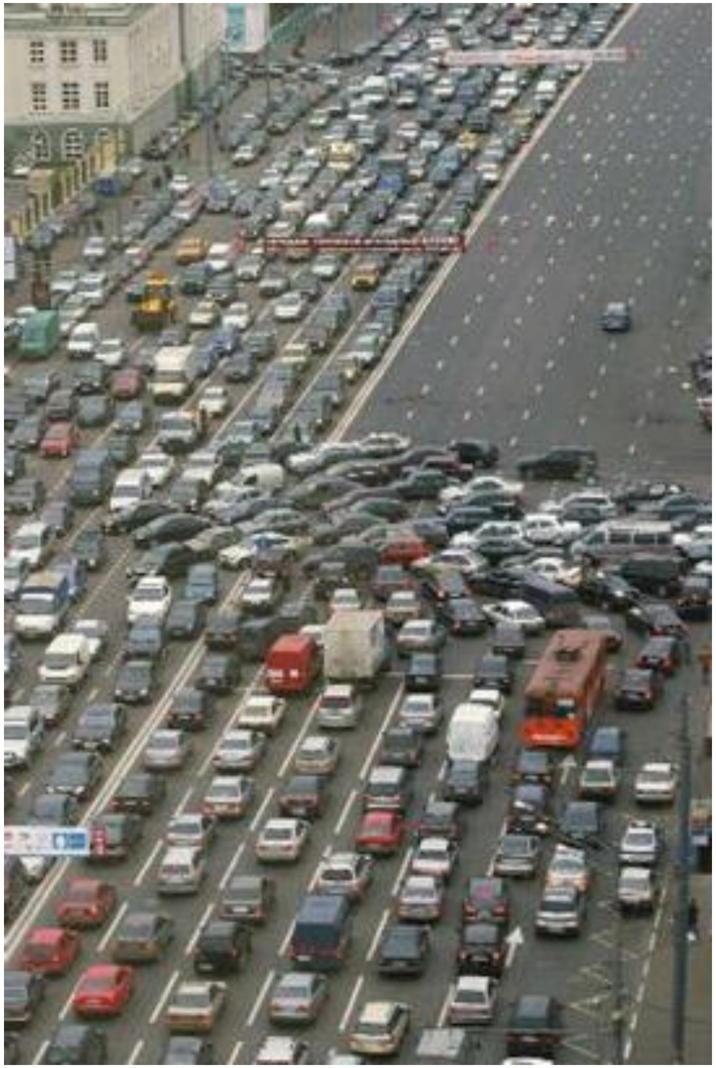


1. Mutual Exclusion
2. Hold & Wait
3. No Preemption
4. Circular Wait

Deadlock?

Is this a Deadlock?

Gridlock



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

Deadlock?

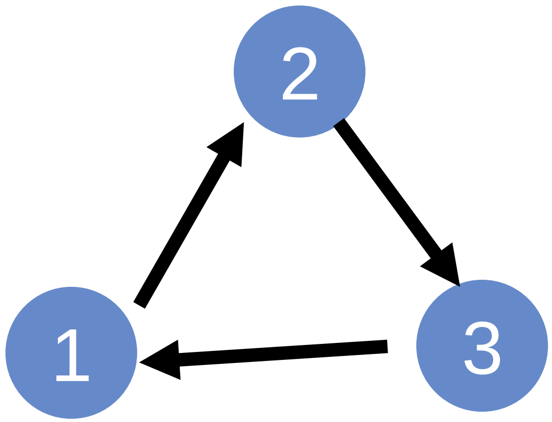
Is this a Deadlock?

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: graph holds for a single instance in time

Cycles in graph indicate deadlock

Testing for cycles (= deadlock)

Find a node with no outgoing edges

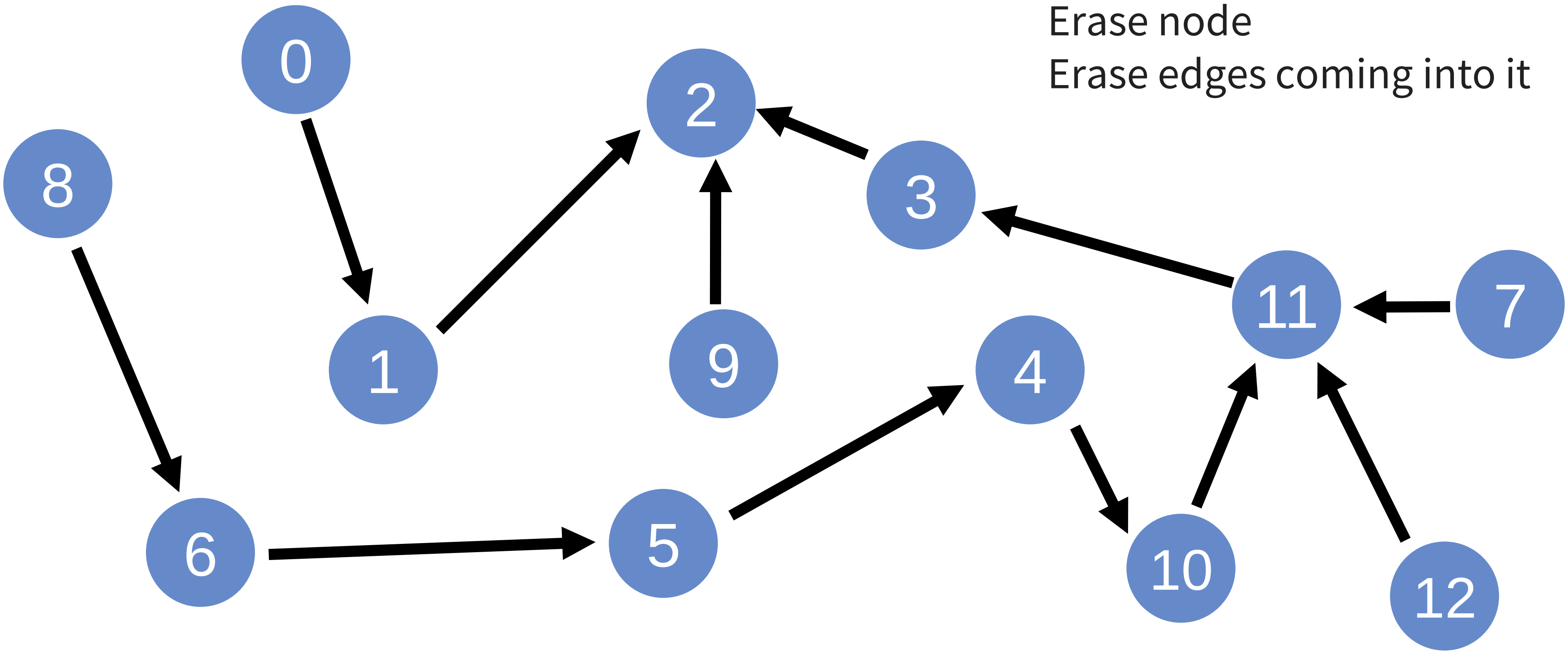
- Erase node
- Erase any edges coming into it

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.

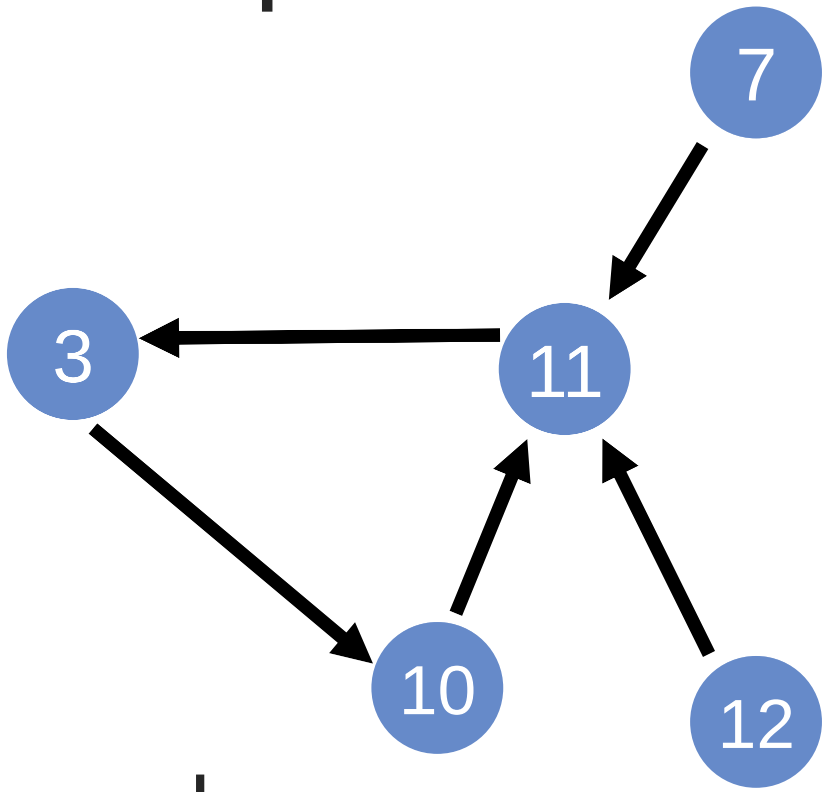
Graph Reduction: Example 1

Find node w/o outgoing edges
- Erase node
- Erase edges coming into it



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

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

Proactive Responses to Deadlocks

Let's not deadlock, okay?

- **Deadlock Prevention:** make it impossible
  - Prevent 1 of the 4 necessary conditions from arising…. … *disaster averted!*

Deadlock Prevention: Negate 1 of 4

#1: Mutual exclusion / Bounded Resources

- Make resources sharable without locks?
- Make more resources available?
- Not always possible (*e.g.*, printers)

Deadlock Prevention: Negate 1 of 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 4

#3: No preemption

Allow runtime system to pre-empt:

1. Requesting processes' resources if all not available
2. Resources of waiting processes to satisfy request

Good when easy to save/restore state of resource

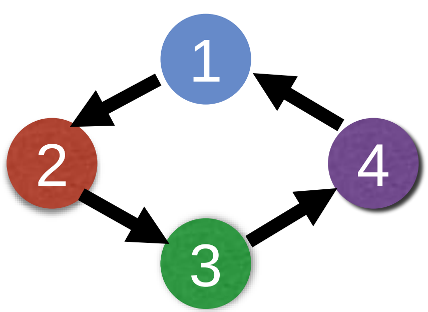
- CPU registers
- memory virtualization (page memory to disk, maybe even page tables)

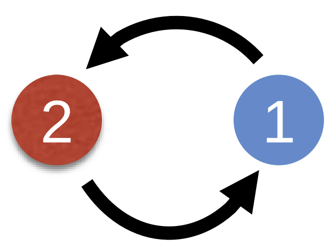
Deadlock Prevention: Negate 1 of 4

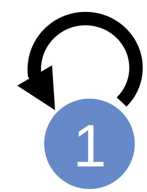
#4: Circular Wait

- 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?

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

def __init__(mynum)
  self.id = mynum

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

1. Bounded Resources
2. Hold & Wait
3. No Pre-emption
4. Circular Wait

Can we prevent one of these conditions? Ideas?

Proactive Responses to Deadlocks

Let’s not deadlock, okay?

- **Deadlock Prevention:** make it impossible
  - Prevent 1 of the 4 necessary conditions from arising…. … *disaster averted!*
- **Deadlock Avoidance:** make it not happen
  - Think before you act

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:

- It is possible to avoid deadlock and eventually grant all resource requests by careful scheduling
- May require delaying a resource request even when resources are available!

Unsafe state:

- Some sequence of resource requests can result in deadlock even with careful scheduling

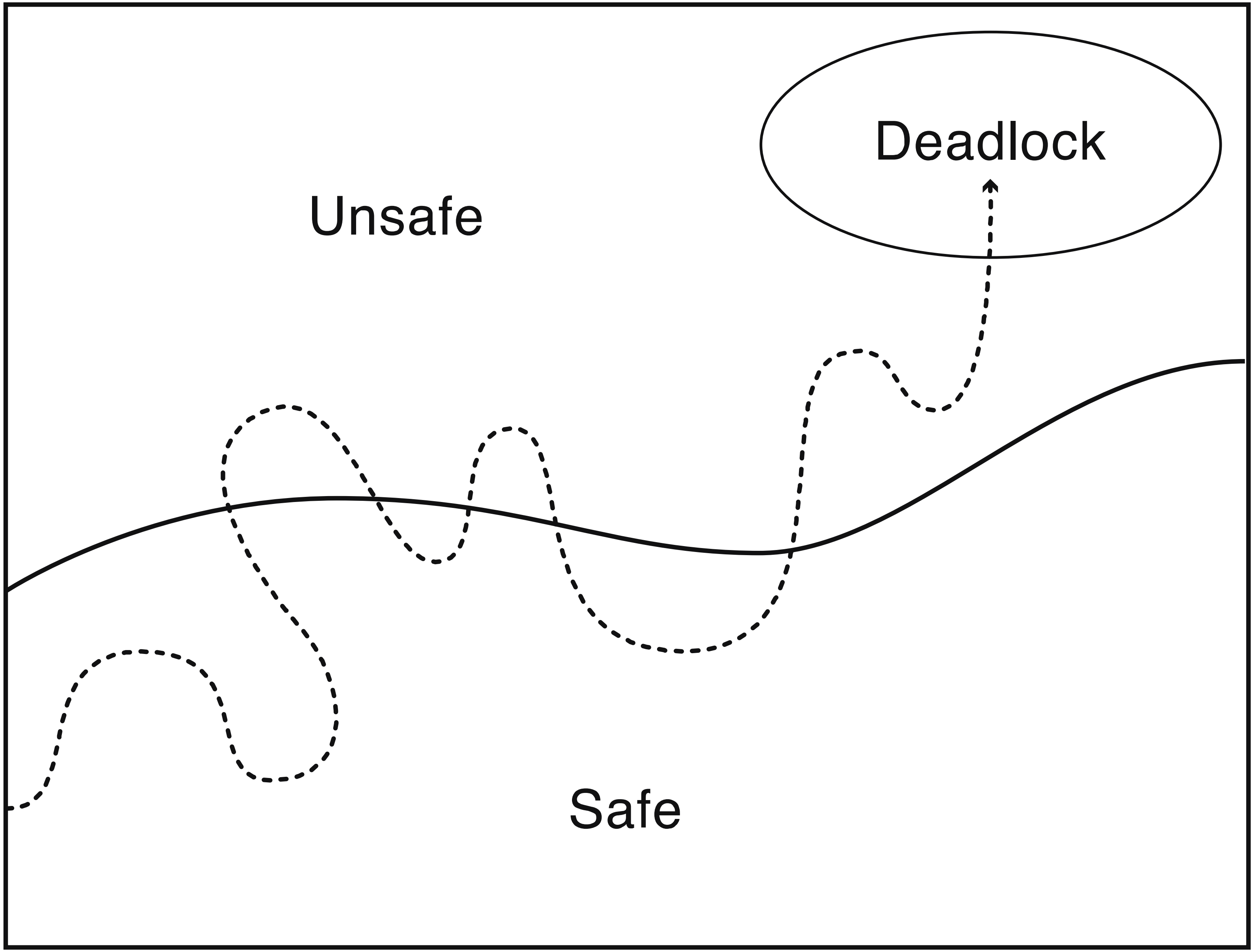
Doomed state:

- 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
  - block new requests until safe order is executed

- Avoids circular wait condition
  - 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 still 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

- from 10,000 feet:
  - Process declares its worst-case needs, asks for what it "really" needs, a little at a time
  - Algorithm decides when to grant requests
    - Build a graph assuming request granted
    - Reducible? yes: grant request, no: wait

Problems:

- Fixed number of processes
- Need worst-case needs ahead of time
- Expensive

Reactive Responses to Deadlocks

If neither avoidance or prevention is implemented, deadlocks can (and will) occur. Now what?

Detect & Recover

Deadlock Detection

- Track resource allocation (who has what)
- Track pending requests (who's waiting for what)

When should we run this?

- For each request?
- After each unsatisfiable request?
- Hourly?
- Once CPU utilization drops below a threshold?
- Some combination of these?

Deadlock Recovery

Blue screen & reboot?

Kill one/all deadlocked processes

- Pick a victim
- Terminate
- Repeat if needed

Preempt resource/processes till deadlock broken

- Pick a victim (# resources held, execution time)
- Rollback (partial or total, not always possible)
- Starve (prevent process from being executed)

Summary

Prevent

- Negate one of the four necessary conditions.

Avoid

- Schedule processes really carefully (?)

Detect

- Determine if a deadlock has occurred

Recover

- Kill or rollback