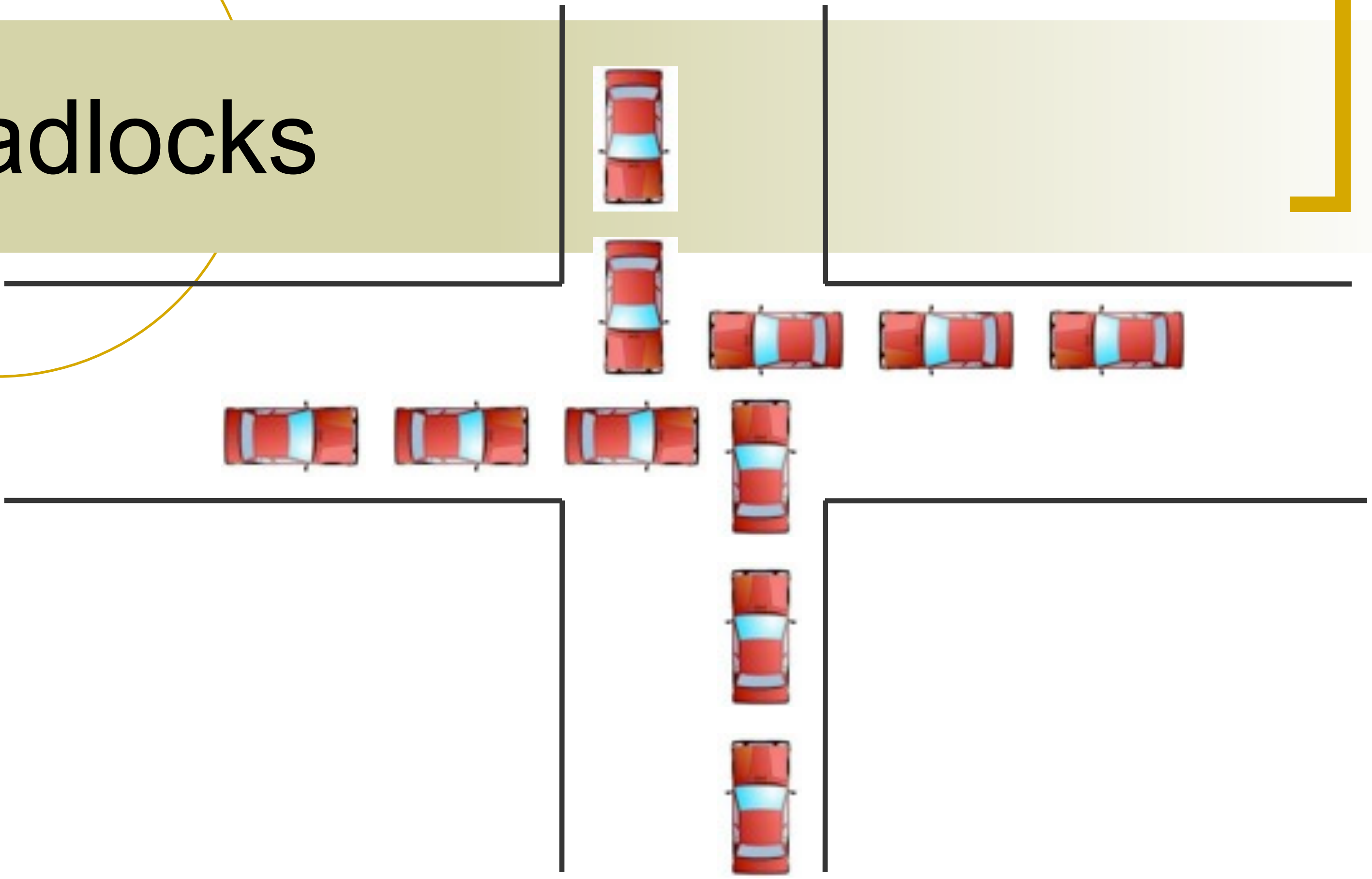
Deadlocks



Upcoming schedule

- Deadlock avoidance
- Homework 2 solutions (if time)

- Friday: Midterm review
- **Monday: Midterm**

How to deal with deadlocks

- **The default**
  - The “ostrich solution”
- **Prevention**
  - design a system in such a way that deadlocks cannot occur, at least with respect to serially reusable resources.
- **Detection**
  - in a system that allows the possibility of deadlock, determine if deadlock has occurred, and which processes and resources are involved.
- **Recovery**
  - after a deadlock has been detected, clear the problem, allowing the deadlocked processes to complete and the resources to be reused. Usually involves destroying (at least one of) the affected processes and starting them over.
- **Avoidance**
  - impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches.

Deadlock Avoidance

- Each resource has multiple instances
- **Resource** = mutex lock, memory, disk, …
- The system needs to know the **maximum** resource requirements of each process ahead of time
  - Process p specifies, for each resource i, **p.Max[i]** = maximum number of instances of i that p can request
- This information will allow us to avoid deadlock.

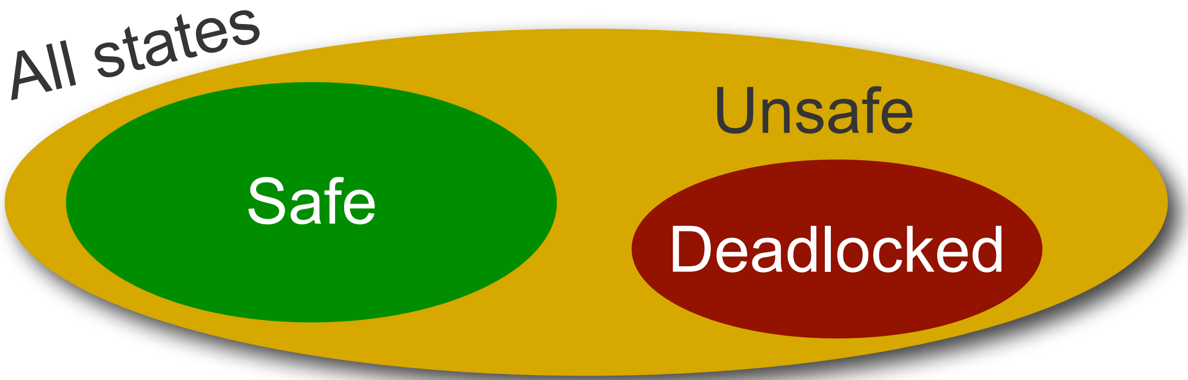
Avoiding deadlock with the Banker's Algorithm

[Dijkstra 1965; Habermann 1969]

- Before starting, each customer tells banker the maximum number of resources it needs
- Customer borrows resources from banker
- Customer returns resources to banker
- Banker only lends if the system will stay in a safe state after the loan

Safe State and Unsafe State

- Safe State
  - Can guarantee: there is some scheduling order in which every process can run to completion regardless of how they request their resources
  - From safe state, the system can guarantee that all processes will finish
- Unsafe state: no such guarantee



In other words...

- **Safe** means there is a way for the programs to finish executing without deadlocking.
- **Our goal:** guide the system down one of those paths successfully.

How to guide the system down a safe path of execution

- Subroutine: is a given state safe?
- When a resource allocation request arrives,
  - Pretend that we approve the request. Would we then be safe? (use subroutine)
  - If so, approve request
  - Otherwise, block process until its request can be safely approved

Subroutine: is a state safe?

- What is a "state"? For each resource,
  - Current amount **available**
  - Current amount **allocated** to each process
  - Future amount **needed** by each process

	Memory	GPU
Free	🟩🟩	
P1 alloc	🟨🟨	🟨
P2 alloc	🟨🟨	
P1 need	🟥🟥	
P2 need	🟥🟥🟥	🟥

Subroutine: is a state safe?

- **Safe** = there is an execution order which can finish
- Pessimistic assumption: processes never release resources until they're done
- State is safe when there is an ordering of processes P1, P2, ..., Pn such that for all processes *i* and resources *j*,

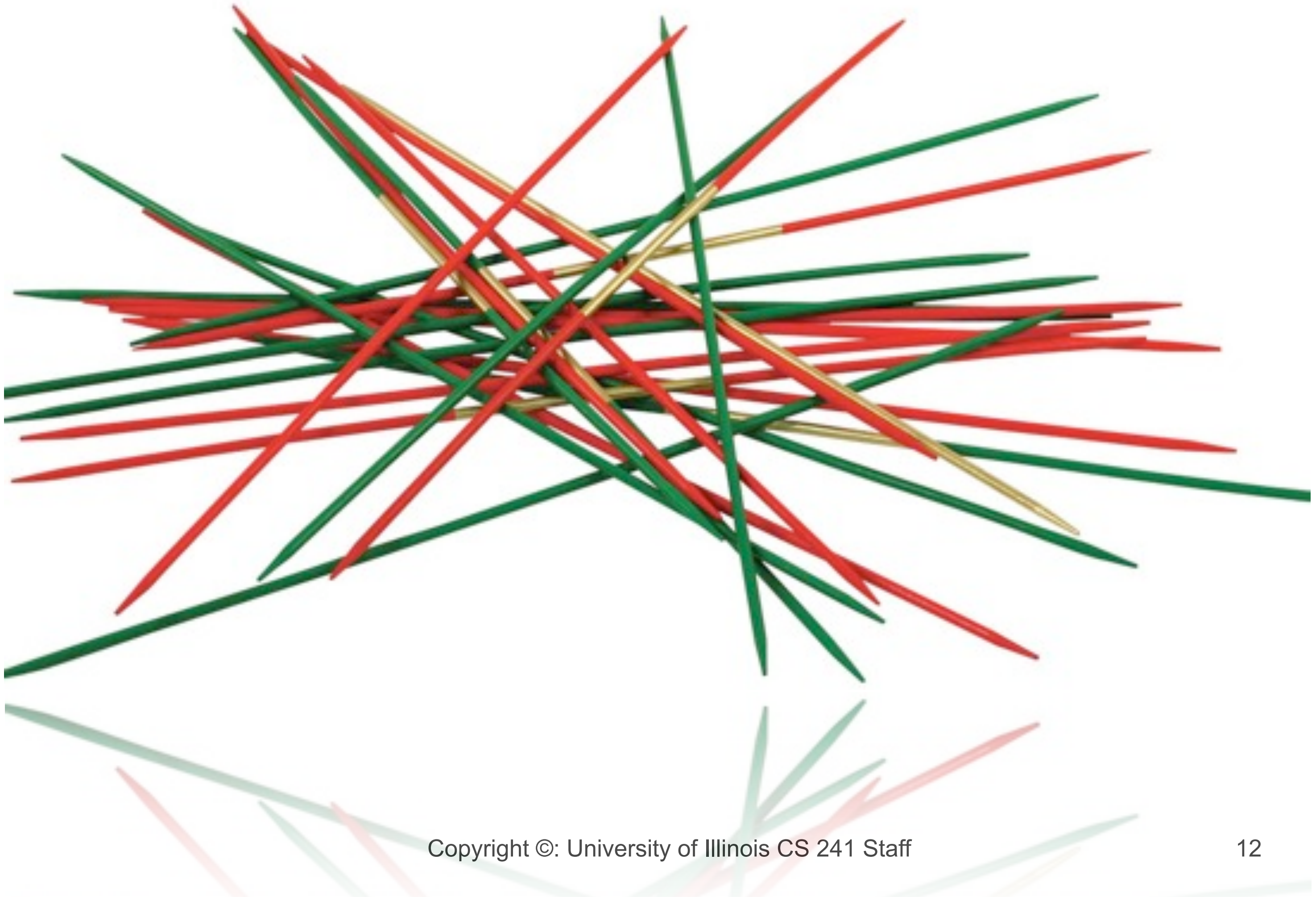
$$\text{need}(i,j) \leq \text{free}(i,j) + \sum_{k<i} \text{alloc}(k,j)$$

Subroutine: is a state safe?

- **Safe** = there is an execution order which can finish
- i.e, there is an ordering of processes P1, P2, ..., Pn such that for all processes *i* and resources *j*,
  - P1 can finish using what it has plus what's free
  - P2 can finish using what it has plus what's free, plus what P1 will release when it finishes
  - P3 can finish using what it has, plus what's free, plus what P1 and P2 will release when they finish
  - ...

How do we figure that out? Try all orderings?

Inspiration...



Playing pickup sticks with processes

- Find some process that can finish with what it has plus what's free
- Imagine that the process finishes and releases its resources.
- Repeat until all processes have finished (answer: safe) or we get stuck (answer: unsafe).

Try it: is this state safe?

	Memory	GPU
Free	🟩🟩	
P1 alloc	🟨🟨	
P2 alloc	🟨🟨🟨🟨	🟨
P1 need	🟥🟥🟥	🟥
P2 need	🟥🟥	

Try it: is this state safe?

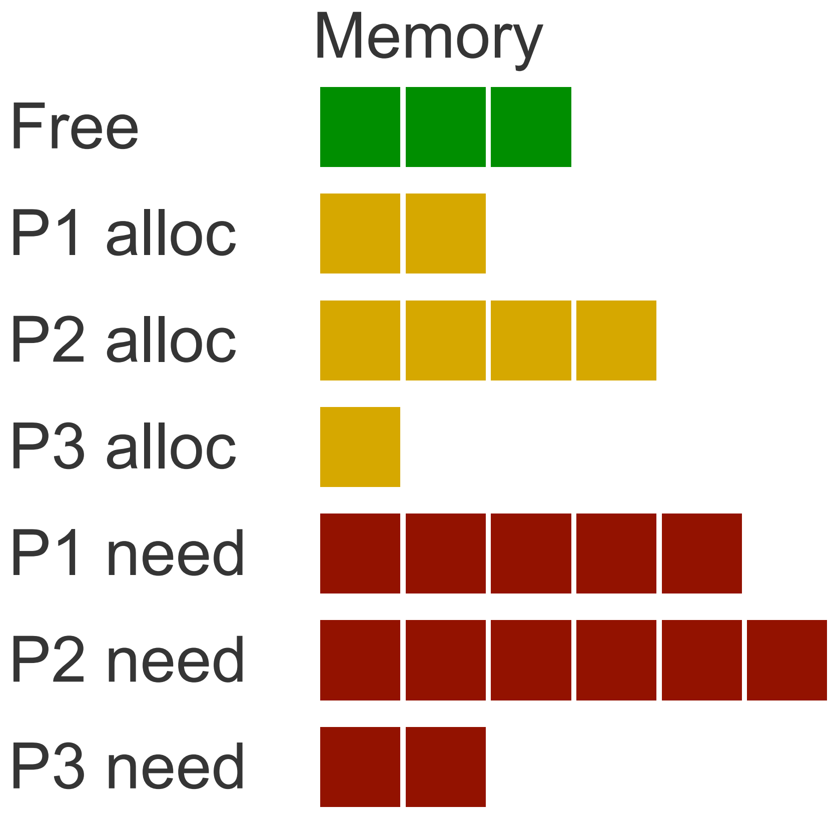
	Memory	GPU
Free	🟩🟩🟩🟩	🟩
P1 alloc	🟨🟨	
P2 alloc	🟨🟨🟨🟨	🟨
P1 need	🟥🟥🟥	🟥
P2 need		

Try it: is this state safe?

	Memory	GPU
Free	🟩🟩🟩🟩	🟩
P1 alloc	🟨🟨🟨🟨🟨	🟨
P2 alloc		
P1 need	🟥🟥🟥	🟥
P2 need		

Yes, it's safe: Order is P2, P1

Example 2: Is this state safe?



Example 2: Is this state safe?

	Memory
Free	🟩🟩🟩🟩
P1 alloc	🟨🟨
P2 alloc	🟨🟨🟨🟨
P3 alloc	
P1 need	🟥🟥🟥🟥🟥
P2 need	🟥🟥🟥🟥🟥🟥
P3 need	

Unsafe!

How to guide the system down a safe path of execution

- Subroutine: is a given state safe?
- When a resource allocation request arrives,
    - Pretend that we approve the request. Would we then be safe? (use subroutine)
    - If so, approve request
    - Otherwise, block process until its request can be safely approved

Banker's algorithm example

For each request,
- If we approved it, would we still be safe?
- If so, approve
- If not, block

	Memory	Disk
Free	🟩🟩	🟩🟩
P1 alloc		
P2 alloc		
P1 need	🟥🟥	🟥
P2 need	R equest	🟥 R

Banker’s algorithm example

For each request,
- If we approved it, would we still be safe?
- If so, approve
- If not, block

	Memory	Disk
Free	🟩	🟩
P1 alloc		
P2 alloc	🟨	🟨
P1 need	🟥 🟥R	🟥
P2 need		🟥

Banker's algorithm example

For each request,

- If we approved it, would we still be safe?
- If so, approve
- If not, block

	Memory	Disk
Free		🟩
P1 alloc	🟨	
P2 alloc	🟨	🟨
P1 need	🟥	R
P2 need		🟥

Banker's algorithm example

For each request,

- If we approved it, would we still be safe?
- If so, approve
- If not, block

	Memory	Disk
Free		🟩
P1 alloc	🟨	
P2 alloc	🟨	🟨
P1 need	🟥	🟥 Blocked!
P2 need		🟥

Banker's algorithm example

For each request,
- If we approved it, would we still be safe?
- If so, approve
- If not, block

	Memory	Disk
Free		🟩
P1 alloc	🟨	
P2 alloc	🟨	🟨
P1 need	🟥	B locked!
P2 need		R

Banker's algorithm example

For each request,

- If we approved it, would we still be safe?
- If so, approve
- If not, block

	Memory	Disk
Free		
P1 alloc	🟨	
P2 alloc	🟨	🟨🟨
P1 need	🟥	Blocked!
P2 need		

Banker's algorithm example

For each request,
- If we approved it, would we still be safe?
- If so, approve
- If not, block

	Memory	Disk
Free	🟩	🟩🟩
P1 alloc	🟨	
P2 alloc		
P1 need	🟥	Blocked!
P2 need		

Banker's algorithm example

For each request,

- If we approved it, would we still be safe?
- If so, approve
- If not, block

	Memory	Disk
Free	🟩	🟩
P1 alloc	🟨	🟨
P2 alloc		
P1 need	🟥	
P2 need		

Banker's algorithm example 2

```
mutex m1, m2;
int x, y;
```

Two processes doing this:

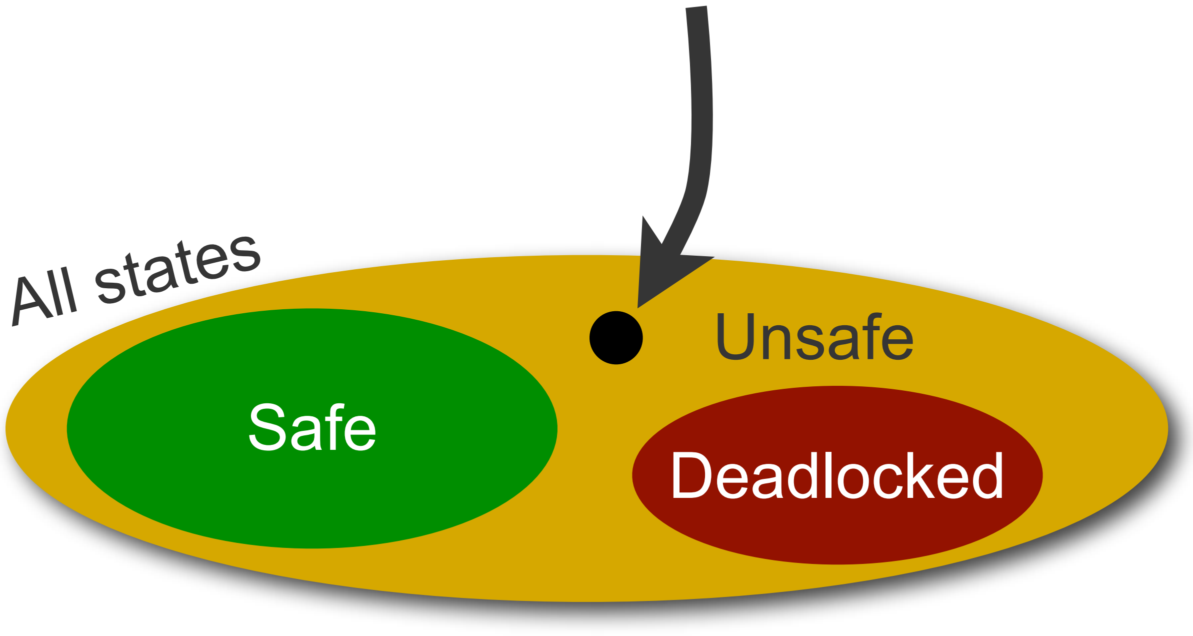
```
while (1) {
    lock(m1); x++; unlock(m1);
    lock(m2); y++; unlock(m2);
}
```

	m1	m2
Free	🟩	🟩
P1 alloc	🟨	
P2 alloc		
P1 need	🟥 R	🟥
P2 need	🟥	🟥 R

While P1 locks m1, P2 can't lock m2!
What did Banker's algorithm get wrong?

Pessimistic assumption: processes never release resources until they're done!

Banker's algorithm example 2

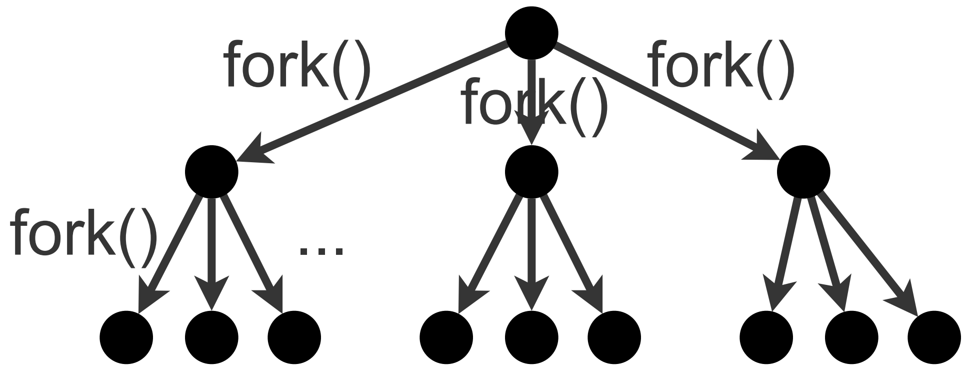


Concluding notes

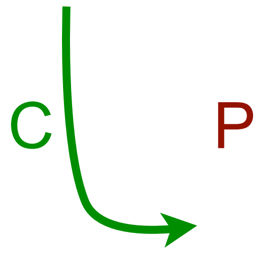
- In general, deadlock detection or avoidance is expensive
- Must evaluate cost and frequency of deadlock against costs of detection or avoidance
- Deadlock avoidance and recovery may cause indefinite postponement
- Unix, Windows use Ostrich Algorithm (do nothing)
- Typical apps use deadlock prevention (order locks)
- Transaction systems (e.g., credit card systems) need to use deadlock detection/recovery/ avoidance/prevention (why?)

Homework 2

Question 2: Balanced ternary tree of depth *N*



A solution



```
create_ternary_tree_of_depth(int N) {
   int i;
   for (i = 1; i < N; i++)
      if (fork())
         if (fork())
            if (fork())
               break;
}
```