Deadlocks

Addressing Deadlock

- Prevention
  - Design the system so that deadlock is impossible
- Detection & Recovery
  - Check for deadlock (periodically or sporadically) and identify and which processes and resources involved
  - Recover by killing one of the deadlocked processes and releasing its resources
- Avoidance
  - Construct a model of system states, then choose a strategy that, when resources are assigned to processes, will not allow the system to go to a deadlock state
- Manual intervention
  - Have the operator reboot the machine if it seems too slow

Deadlock Avoidance

- Deadlock detection
  - Assumes all resources are requested at start time
- Realistic scenarios
  - Resources are requested incrementally
- Deadlock Avoidance: Basic idea
  - Try to see the worst that could happen
  - Do not grant an incremental resource request to a process if this allocation might lead to deadlock
  - Conservative/pessimistic approach

Deadlock Avoidance

- Assume OS knows
  - Number of available instances of each resource
    - Mutex: a resource with one instance available
    - Semaphore: a resource with possibly multiple “instances” available
  - For each process
    - Current amount of each resource it owns
    - Maximum amount of each resource it might ever need
      - For a mutex this means: Will the process ever lock the mutex?
- Assume processes are independent
  - If one blocks, others can finish if they have enough resources

Deadlock and Resources

- Single instance of each resource
  - Find cycle in resource allocation graph
- Multiple instance of each resource
  - Process can request any number of instances for a given resource
    - May only use some of them

Deadlock Avoidance: Safe vs. Unsafe

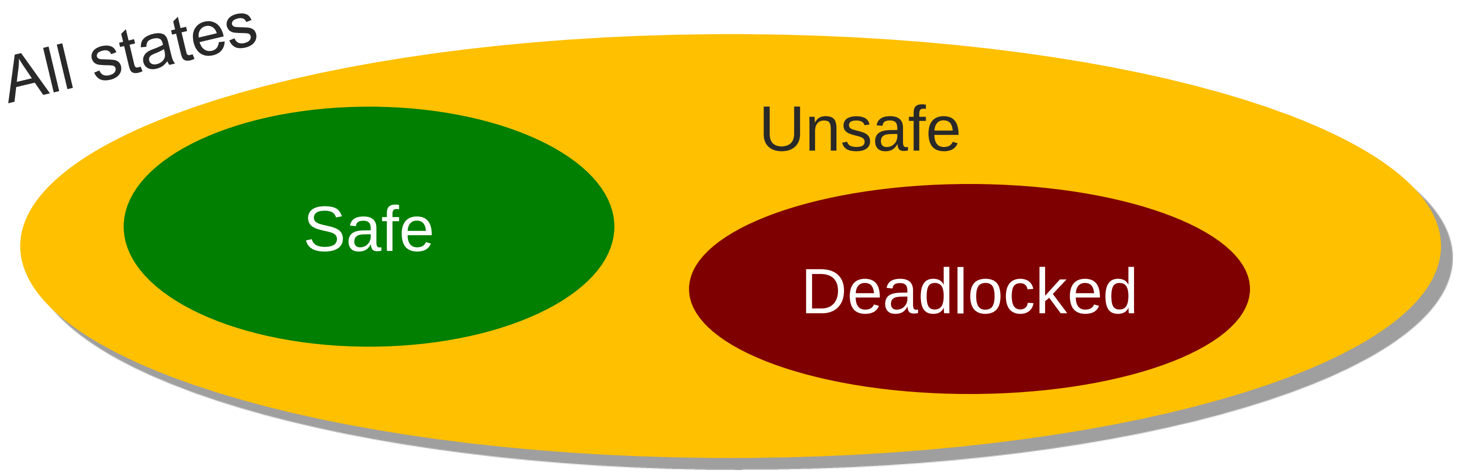
- Approach
  - Define a model of system states (SAFE, UNSAFE)
  - Choose a strategy that guarantees that the system will not go to a deadlock state
- Safe
  - Guarantee
    - There is some scheduling order in which every process can run to completion even if all of them suddenly and simultaneously request their maximum number of resources
  - From a safe state
    - The system can guarantee that all processes will finish

Deadlock Avoidance: Safe vs. Unsafe

- Approach
  - Define a model of system states (SAFE, UNSAFE)
  - Choose a strategy that guarantees that the system will not go to a deadlock state
- Unsafe state: no such guarantee
  - A deadlock state is an unsafe state
  - An unsafe state may not be a deadlock state
  - Some process may be able to complete

Safe vs. Unsafe

- Safe
  - There is a way for all processes to finish executing without deadlocking
- Goal
  - Guide the system down one of those paths successfully

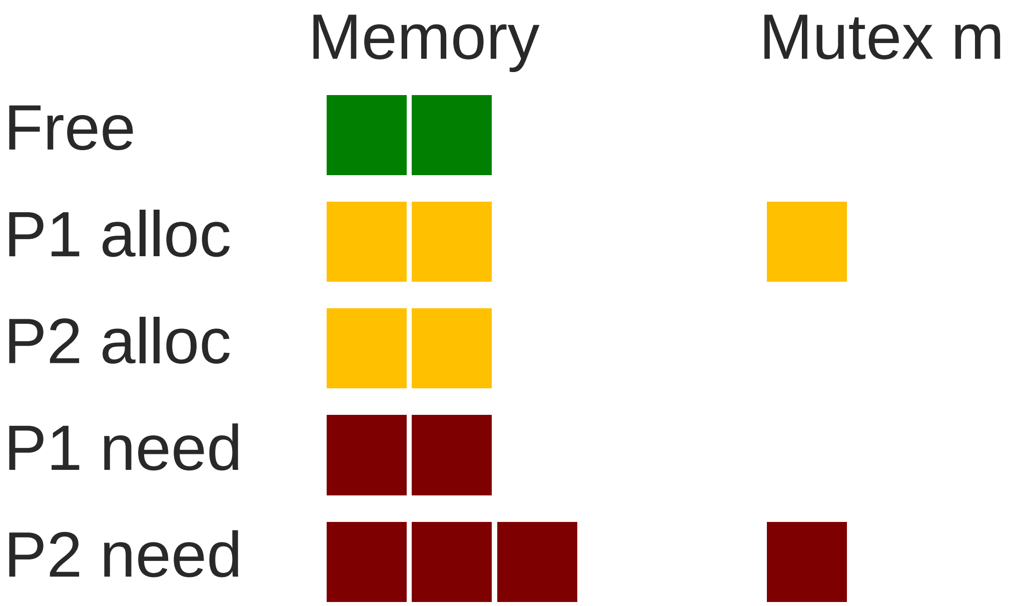


How to guide the system down a safe path of execution

- New function: is a given state safe?
- When a resource allocation request arrives
  - Pretend that we approve the request
    - Call function: Would we then be safe?
  - If safe
    - Approve request
  - Otherwise
    - Block process until its request can be safely approved

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



Is a state safe?

- Safe
  - There is an execution order that can finish
- Pessimistic assumption
  - Processes never release resources until they're done

Is a state safe?

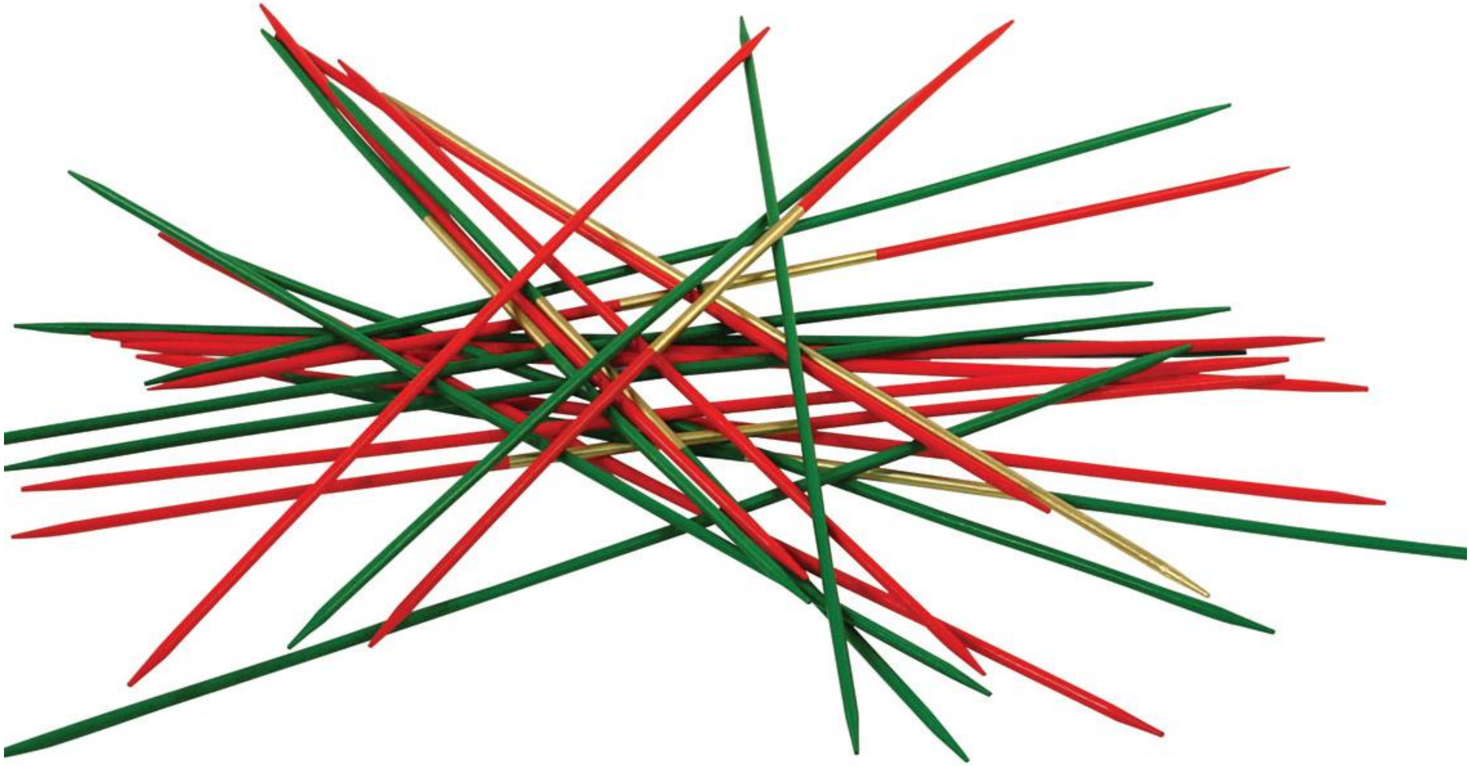
- Safe
  - There is an execution order that can finish
  - **P1** can finish using what it has plus what's free
  - **P2** can finish using what it has plus what's free, plus what **P1** will release when it finishes
  - **P3** can finish using what it has, plus what's free, plus what **P1** and **P2** will release when they finish
  - ...

Is a state safe?

- Search for an order **P1**, **P2**, **P3**, ... such that:
  - **P1**'s max resource needs ≤ what it has + what's free
  - **P2**'s max resource needs ≤ what it has + what's free + what **P1** will release when it finishes
  - **P3**'s max resource needs ≤ what it has + what's free + what **P1** and **P2** will release when they finish

How do we figure that out?
Try all orderings?
How many orderings do we need to find?

Inspiration...

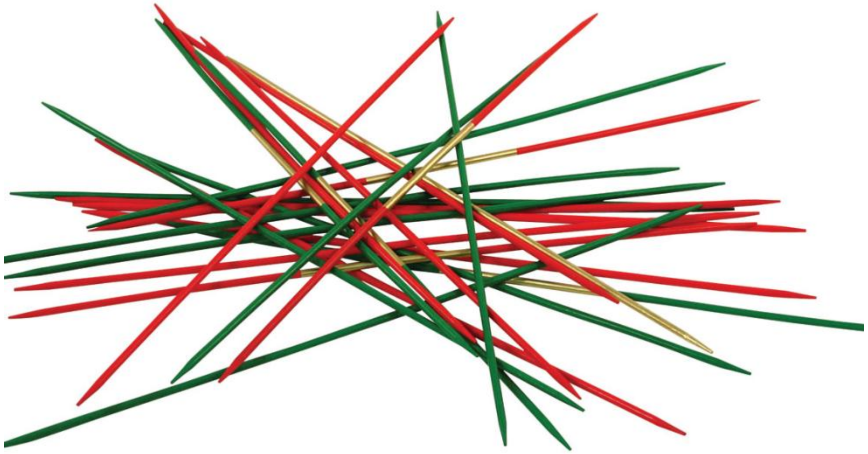


Playing pickup sticks with processes

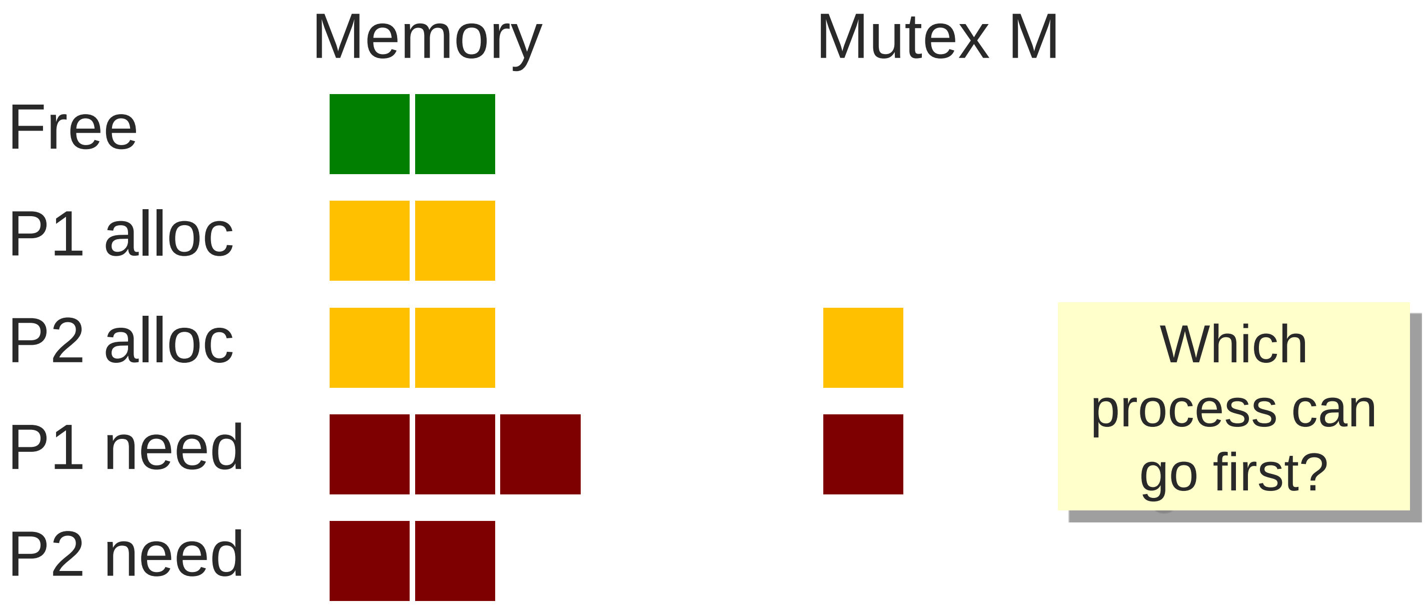
- Pick up
  - Find a stick on top

    = Find a process that can finish with what it has plus what's free
  - Remove stick

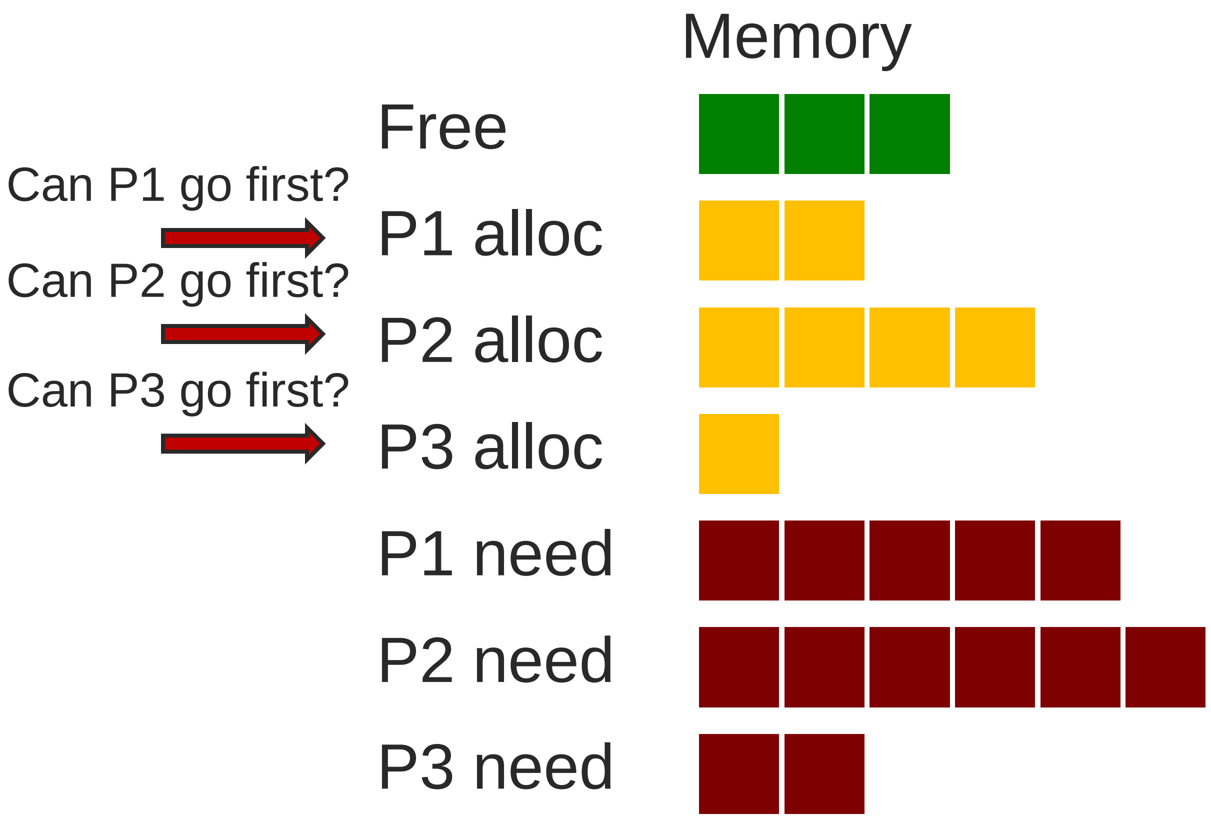
    = Process releases its resources
- Repeat
  - Until all processes have finished
    - Answer: safe
  - Or we get stuck
    - Answer: unsafe



Try it: is this state safe?



Example 2: Is this state safe?



How to guide the system down a safe path of execution

- New function: is a given state safe?
- When a resource allocation request arrives
  - Pretend that we approve the request
    - Call function: Would we then be safe?
  - If safe
    - Approve request
  - Otherwise
    - Block process until its request can be safely approved

Banker's Algorithm

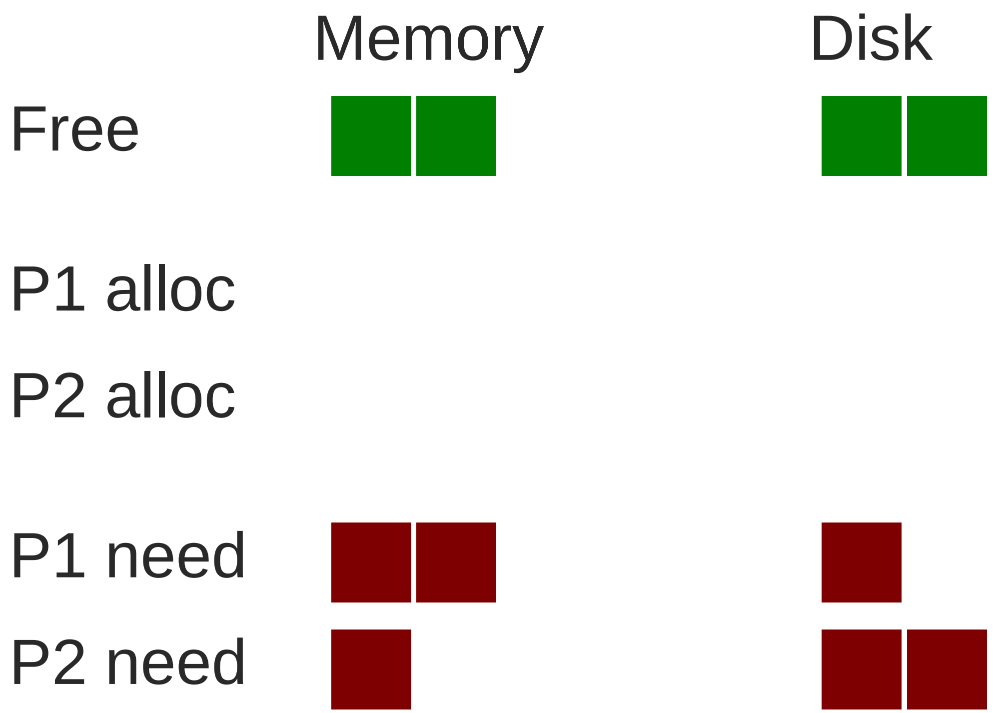
Banker's Algorithm

- Dijkstra, 1965
  - Each customer tells banker the maximum number of resources it needs, before it starts
  - Customer borrows resources from banker
  - Customer returns resources to banker
  - Banker only lends resources if the system will stay in a safe state after the loan
    - Customer may have to wait

Banker's Algorithm: Take 1

For each request

- If approved, would we still be safe?
- If yes
  - Approve
- If no
  - Block

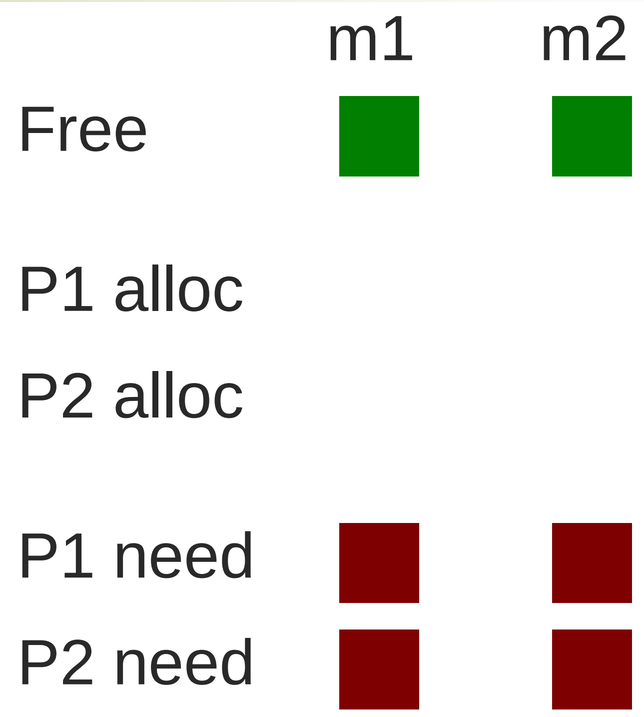


Banker's Algorithm: Take 2

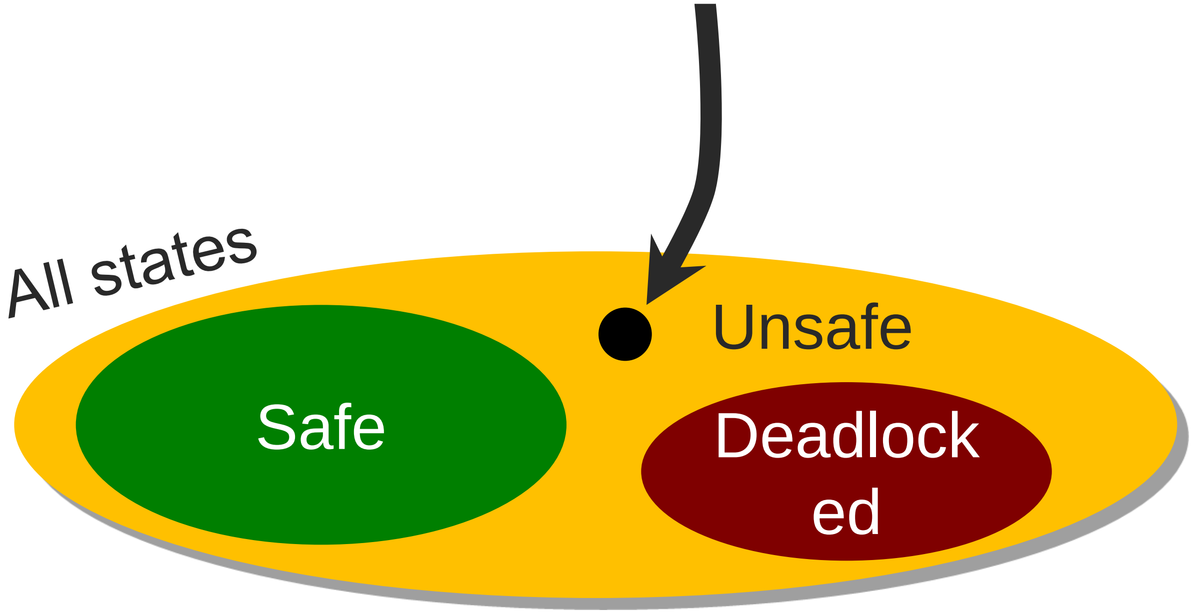
```
mutex m1, m2;
int x, y;

while (1) {
  lock(m1);
  x++;
  unlock(m1);

  lock(m2);
  y++;
  unlock(m2);
}
```



Banker's algorithm example 2



Formalized Banker's Algorithm

- Given
  - **n** resource types
  - **P** processes
  - **p.Max[1…n]**
    - Maximum number of resource **i** needed by **p**
  - **p.Alloc[i]**
    - Number of instances of resource **i** held by **p**
    - **<= p.Max[i]**
  - **Avail [1…n]**
    - Current number of available resources of each type
  - **p.Need[i] = p.MAX[i] - p.Alloc[i]**

- Algorithm:

```
while (there exists a p in P such
    that {for all i (p.Need[i] <=
    Available[i] )}) {

    for (all i) {
        Avail [i] += p.Alloc[i];
        P = P - p;
    }
}
```

- If **P** is empty then system is safe

Current Allocation

Pr	Alloc				Max				Need				Total		
	A	B	C		A	B	C		A	B	C		A	B	C
P0	0	1	0		7	5	3		7	4	3		10	5	5
P1	2	0	0		3	2	2		1	2	2		Available		
P2	3	0	0		9	0	2		6	0	2		A	B	C
P3	2	1	1		2	2	2		0	1	1		3	3	2
P4	0	0	2		4	3	3		4	3	1				

Can P1 request (A:1 B:0 C:2) ?

New Allocation

Pr	Alloc				Max				Need				Total		
	A	B	C		A	B	C		A	B	C		A	B	C
P0	0	1	0		7	5	3		7	4	3		10	5	5
P1	3	0	2		3	2	2		0	2	0		Available		
P2	3	0	0		9	0	2		6	0	2		A	B	C
P3	2	1	1		2	2	2		0	1	1		2	3	0
P4	0	0	2		4	3	3		4	3	1				

Can P0 request (A:0 B:2 C:0) ?

Outcome

- P0’s request for 2 Bs
  - Cannot be granted because
    - Would prevent any other process from completing if they need their maximum claim
- Just Because It’s Unsafe Doesn’t mean it will always deadlock
  - P0 could have been allocated 2 Bs and a deadlock might not have occurred if:
    - P2 didn’t use its maximum resources but finished using the resources it had

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?)