Synchronization: Deadlocks

ICS332

Operating Systems

Henri Casanova (henric@hawaii.edu)

Deadlocks

- The previous set of lecture notes talked about race conditions
- In these lecture notes we talk about another common bug that can happen in concurrent programs: deadlocks
- This is a very different kind of bug
- Often not as confusing / difficult to deal with as race conditions
  - Although this is not always true
- Deadlocks are pretty common

Learning from Mistakes: A Comprehensive Study on Real-World Concurrency Bug Characteristics, Lu et al., ASPLOS'08

Application	Fraction of Concurrency Bugs that are Deadlocks
MySQL (database server)	8%
Apache (web server)	23%
Mozilla (web browser)	28%
OpenOffice (office suite)	25%
Overall	29%

Deadlocks

- The name is inspired from real-life situations
- Early 20th Century Kansas legislature proposed bill: ”When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone”
  - Likely not true

A video of this happening

The Classic 2-Lock Example

Thread #1

```
. . .
lock1.lock();
. . .
lock2.lock();
. . .
```

Thread #2

```
. . .
lock2.lock();
. . .
lock1.lock();
. . .
```

The Classic 2-Lock Example

Thread #1	Thread #2

. . .
lock1.lock();
. . .
lock2.lock();
. . .
	. . .
	lock2.lock();
	. . .
	lock1.lock();
	. . .

One possible Execution timeline

```
. . .
lock1.lock(); // Thread #1 acquires lock #1
<context switch to Thread #2>
. . .
lock2.lock(); // Thread #2 acquires lock #2
. . .
lock1.lock(); // Thread #2 BLOCKS because lock #1 is taken
<context switch to Thread #1>
. . .
lock2.lock(); // Thread #1 BLOCKS because lock #2 is taken
```

Both threads are waiting on each other: they are deadlocked

Deadlock Meme

Defining a Deadlock

- The deadlock problem can be formalized and generalized
- We have a system with Resources and Processes
- The Resources:
  - There can be resources of types: R1, R2, . . ., Rm
  - There are multiple resource of each type: e.g., 3 NICs, 4 disks
- The Processes (or Threads):
  - P1, P2, ..., Pn
  - Each process can:
    - Request a resource of a given type and block/wait until one resource instance of that type becomes available
    - Use a resource
    - Release a resource
- In the previous slides we have two processes, P1 and P2 (2 threads), two resource types R1 (one lock, which corresponds to some resource), and R2 (another lock, which corresponds to another resource)
  - These “resources” could be data structures

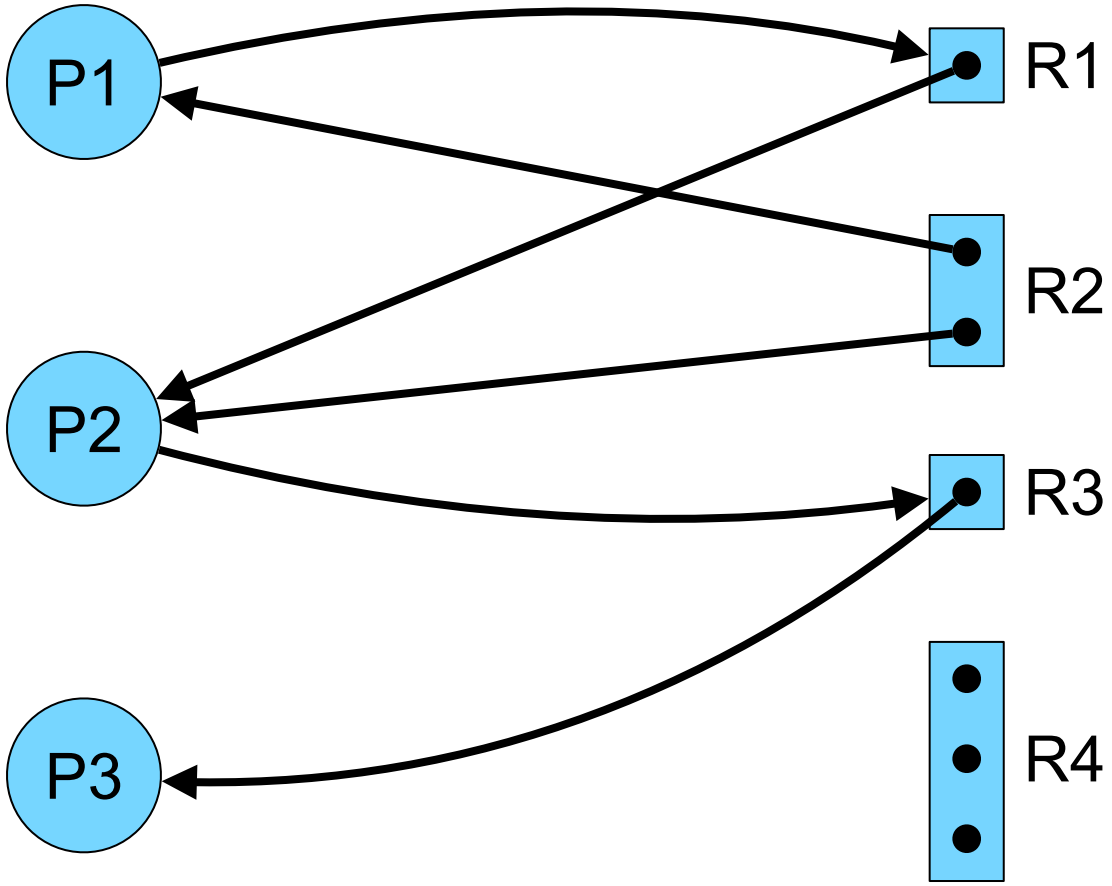
Deadlock State

- A deadlock state happens if every process is waiting for a resource instance that is being held by another process
- Three **necessary** conditions for a deadlock to occur:
  - Mutual exclusion: At least one resource is non-shareable: at most one process at a time can use it
    - In our example: the locks are mutually exclusive
  - No preemption: Resources cannot be forcibly removed from processes that are holding them
    - In our example: only the thread holding a lock can release it
  - Circular wait: There exists a set {P0,P1,...,Pp} of waiting processes such that ($\forall i \in \{0, 1, ...p - 1\}$) Pi is waiting for a resource held by Pi +1 and Pp is waiting for a resource held by P0
    - i.e., There is a circular chain of processes such that each process holds one or more resources that are being requested by the next process in the chain
    - In our example: P1 has lock1 and needs lock2, and P2 has lock2 and needs lock1
- If your program is in a state that meets all three conditions, then it **may** deadlock, otherwise you're safe

Resource Allocation Graph

- Describing the system can be done precisely and easily with a resource-allocation-request graph, where
- The set of vertices is made of:
  - The set of processes {P0, P1, ..., Pn}, and
  - The set of resource types {R0, R1, ..., Rm}
    - Each resource instance is a black dot
- The set of directed edges is made of:
  - Request edges where a request edge is built from a process Pi to a resource Rj if Pi has requested a resource of type Rj
  - Assignment edges where an assignment edge is built from an instance of a resource type Rj to a process Pi if Pi holds a resource instance of type Rj
- Note: if a request can be fulfilled, the assignment edge replaces immediately the request edge
- Let's see it on a picture…

Example Graph



Example from Operating Systems Concepts textbook, Silberschatz et al.

Cycles in the Graph

- Theorem:
  - If the resource-allocation-request graph contains no (directed) cycle, then there is no deadlock in the system
  - If the graph contains a cycle then there may be a deadlock
- **If there is only one resource instance** (black dot) per resource type then we have a Stronger Theorem:
  - The existence of a cycle is a necessary and sufficient condition for the existence of a deadlock
- Let's draw the graph for our 2-thread/2-lock examples…..

2-Thread/2-Lock examples

Thread #1

```
. . .
lock1.lock();
. . .
lock2.lock();
. . .
```

Thread #2

```
. . .
lock2.lock();
. . .
lock1.lock();
. . .
```

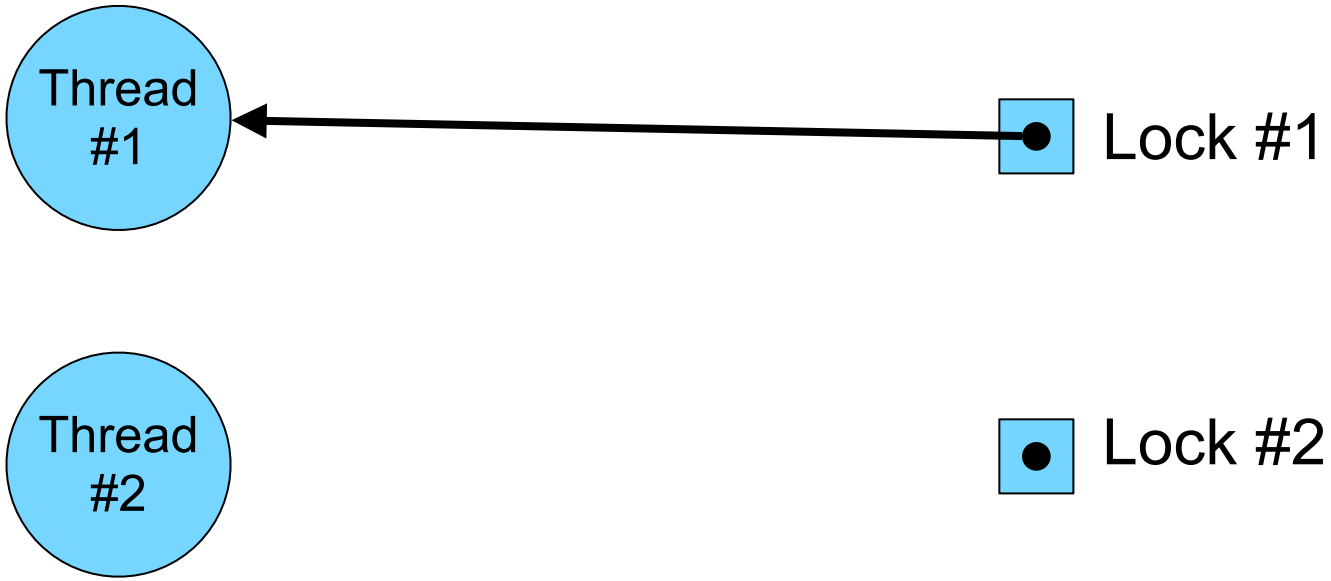
2-Thread/2-Lock examples

Thread #1

```
. . .
lock1.lock();
. . .
lock2.lock();
. . .
```

Thread #2

```
. . .
lock2.lock();
. . .
lock1.lock();
. . .
```



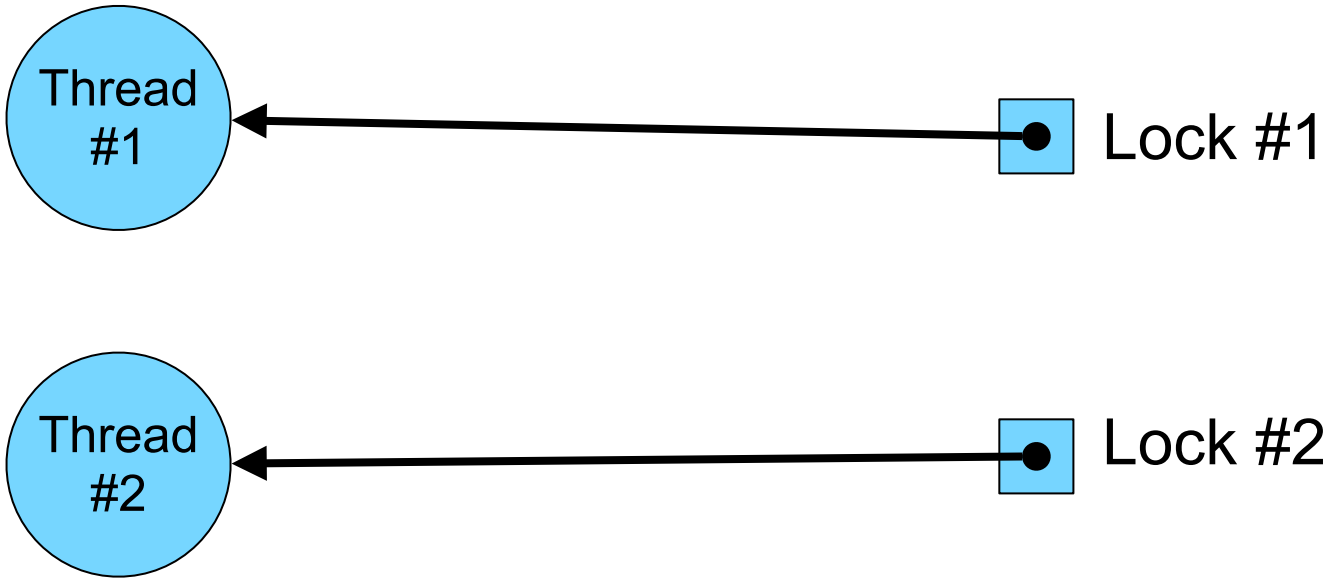
2-Thread/2-Lock examples

Thread #1

```
. . .
lock1.lock();
. . .
lock2.lock();
. . .
```

Thread #2

```
. . .
lock2.lock();
. . .
lock1.lock();
. . .
```



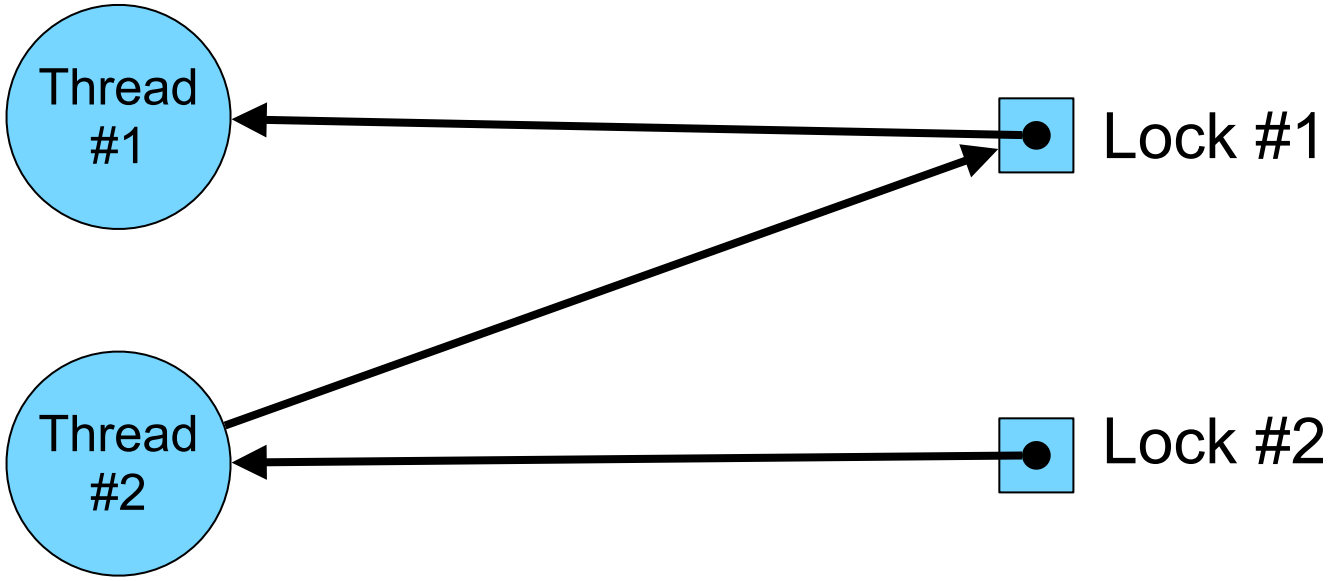
2-Thread/2-Lock examples

Thread #1

```
. . .
lock1.lock();
. . .
lock2.lock();
. . .
```

Thread #2

```
. . .
lock2.lock();
. . .
lock1.lock();
. . .
```



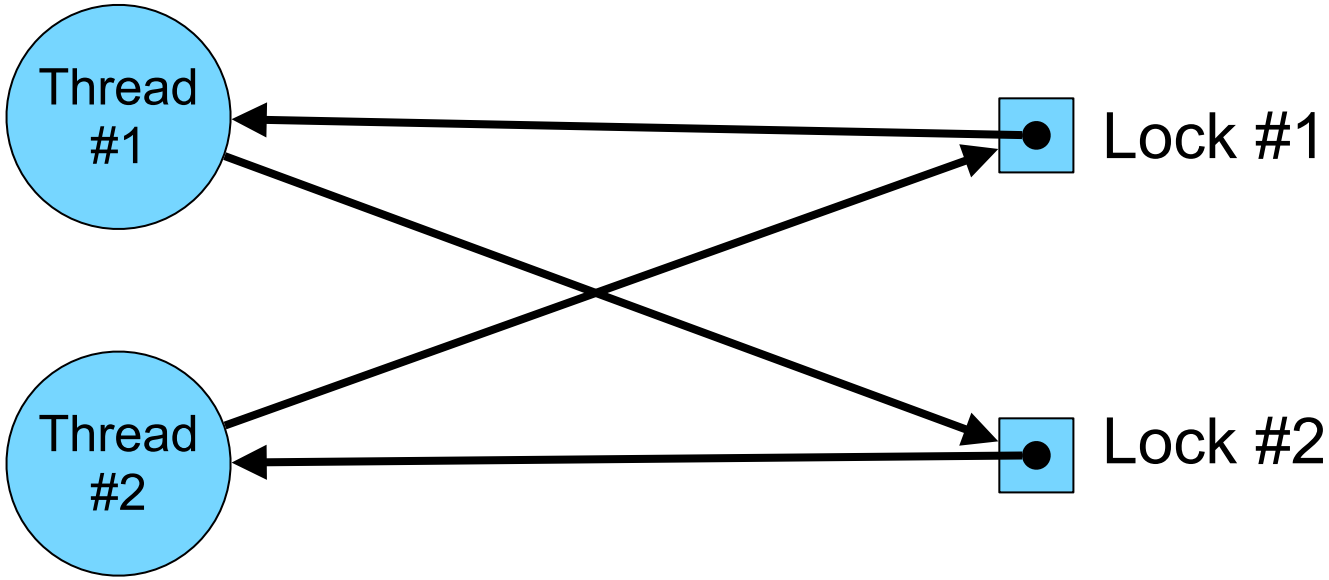
2-Thread/2-Lock examples

Thread #1

```
. . .
lock1.lock();
. . .
lock2.lock();
. . .
```

Thread #2

```
. . .
lock2.lock();
. . .
lock1.lock();
. . .
```



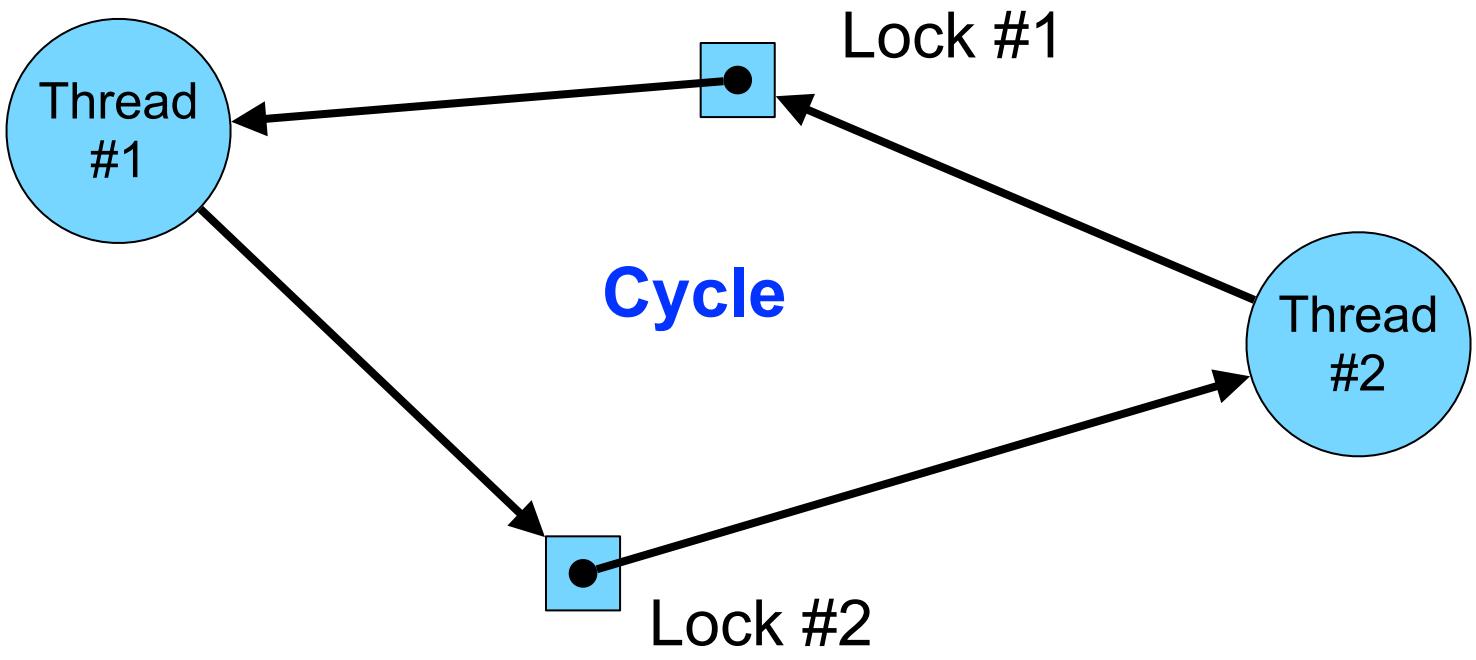
2-Thread/2-Lock examples

Thread #1

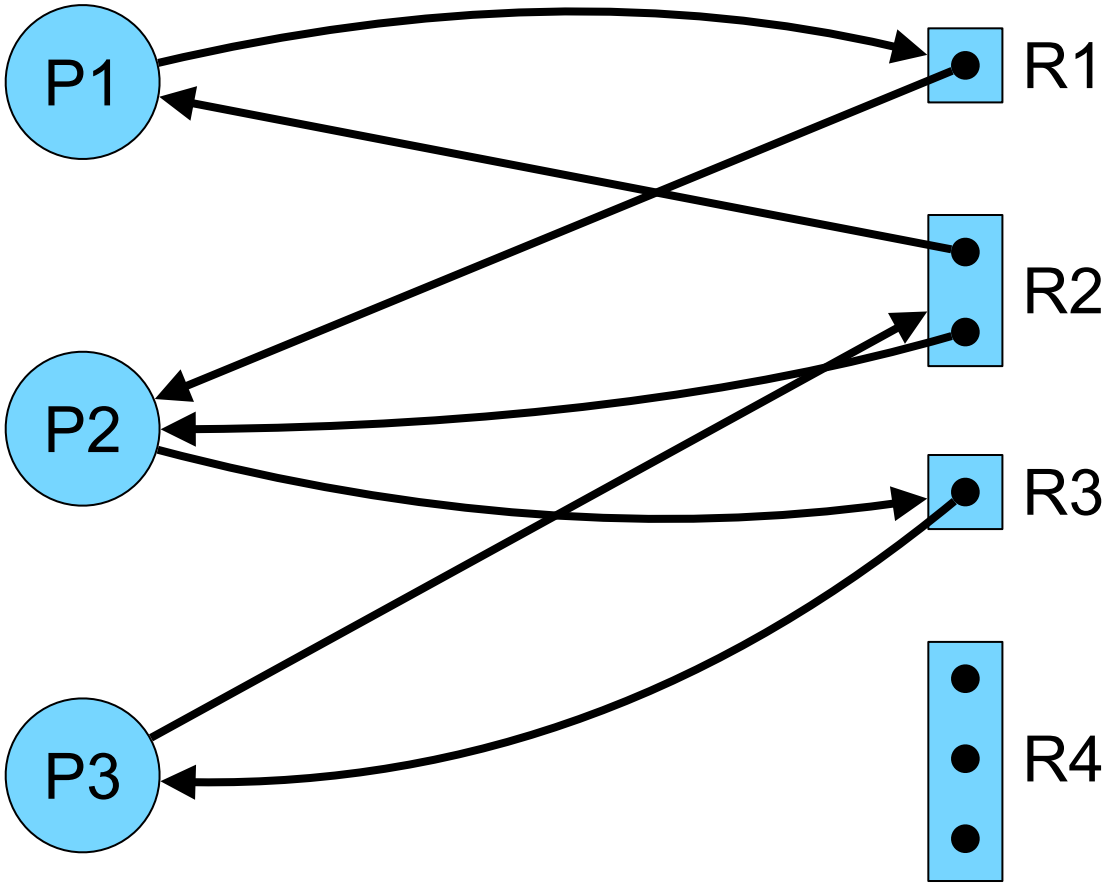
```
. . .
lock1.lock();
. . .
lock2.lock();
. . .
```

Thread #2

```
. . .
lock2.lock();
. . .
lock1.lock();
. . .
```



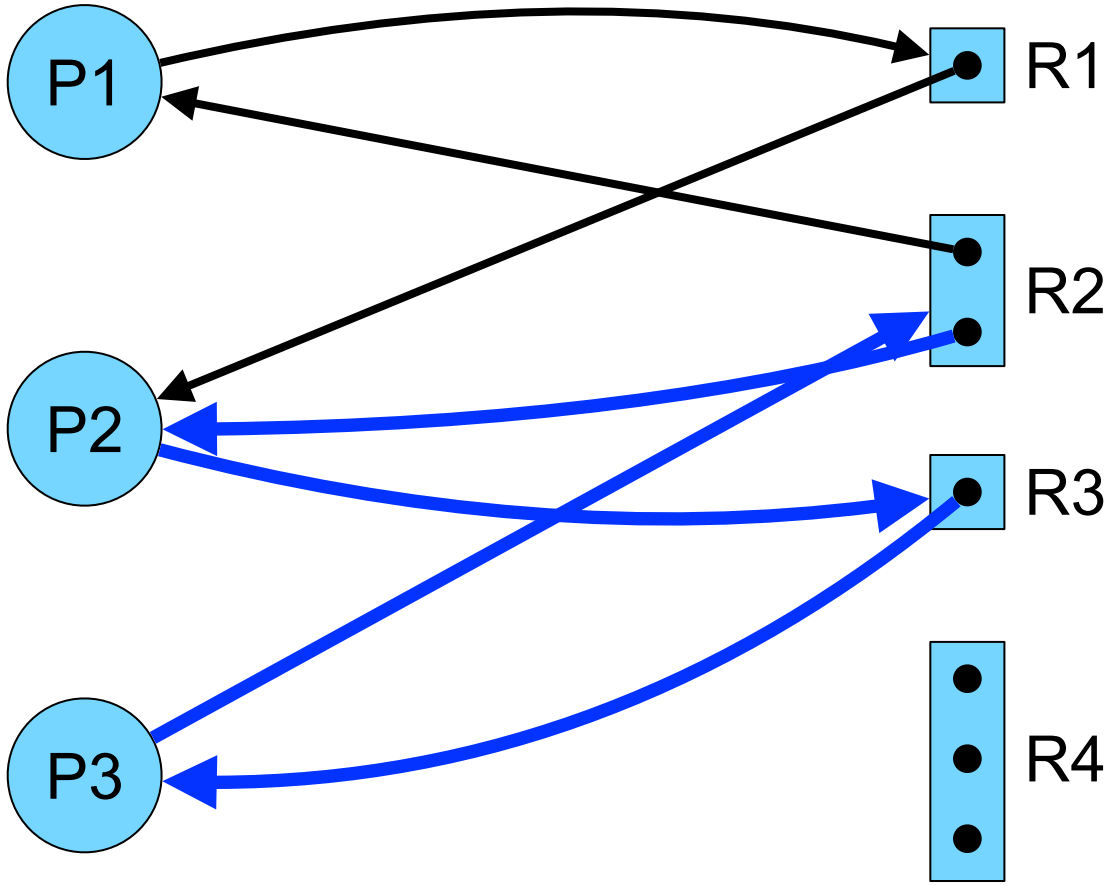
Example Graph: Cycle?



Example from Operating Systems Concepts textbook, Silberschatz et al.

Example Graph: Cycle?

The blue edges form a cycle

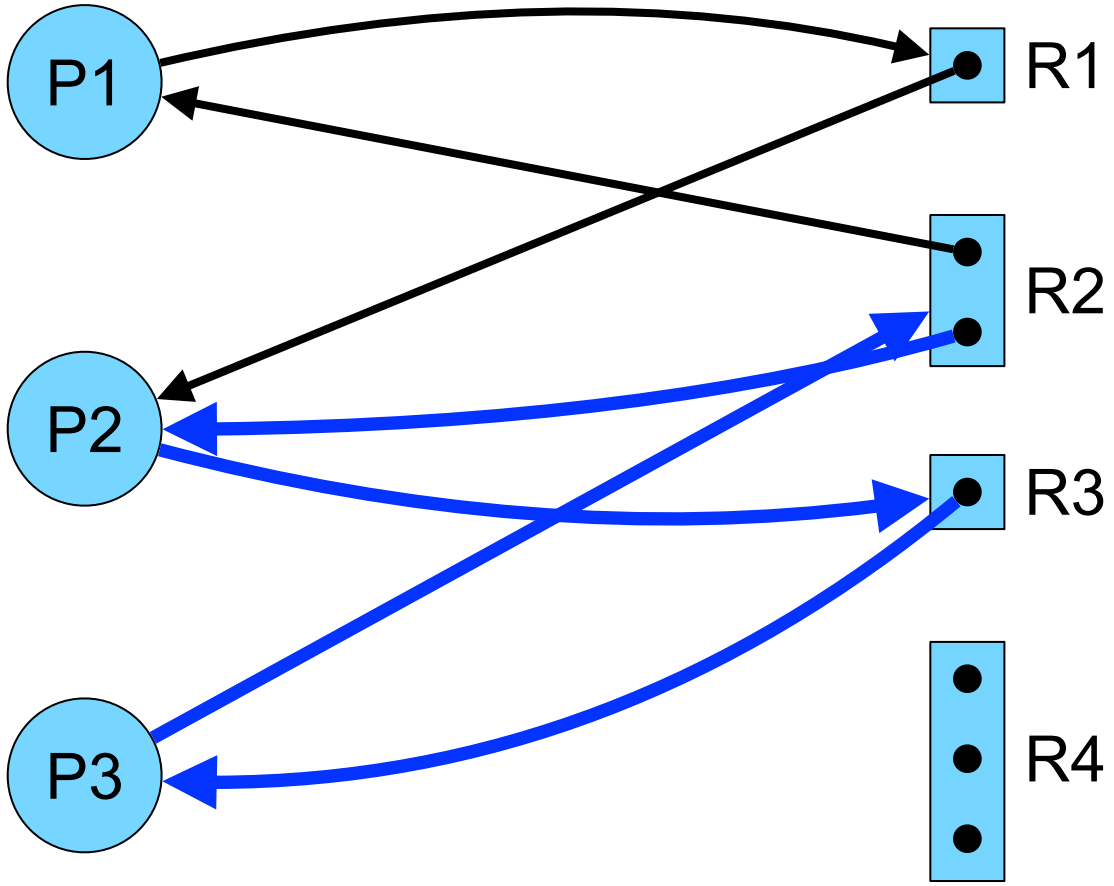


Example from Operating Systems Concepts textbook, Silberschatz et al.

Example Graph: Cycle?

The blue edges form a cycle

But are we deadlocked?



Example from Operating Systems Concepts textbook, Silberschatz et al.

Are we Deadlocked?

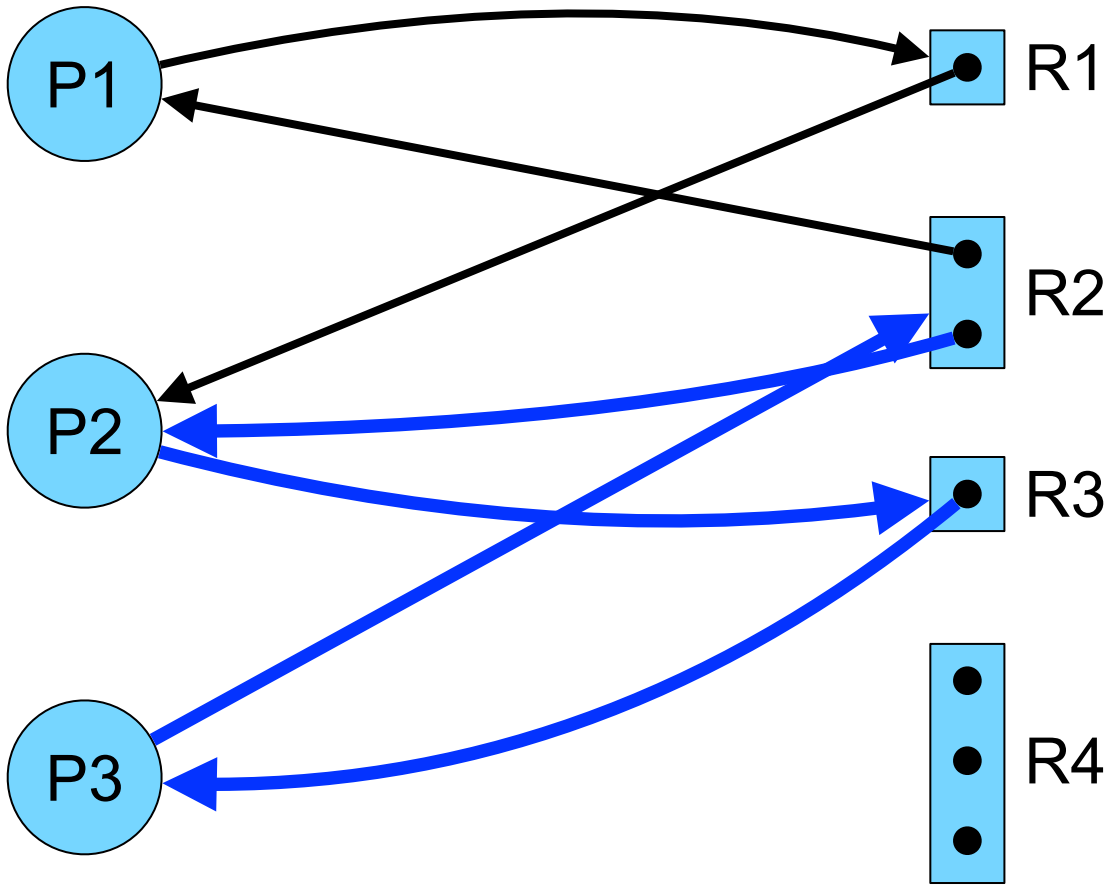
- In the previous example we have a cycle, so there may be a deadlock
  - Because there are multiple resources for some resource types
- We have a deadlock if no process involved in the cycle can make progress
- We can check this as follows:
  - Each process that has all the resources it wants will eventually move on and release its resources
  - So we can remove its incoming resource allocation edges, and perhaps transform some resource request edges into resource allocation edges
  - We keep going…
- Let's look at the example again…

Example Graph: Cycle?

The blue edges form a cycle

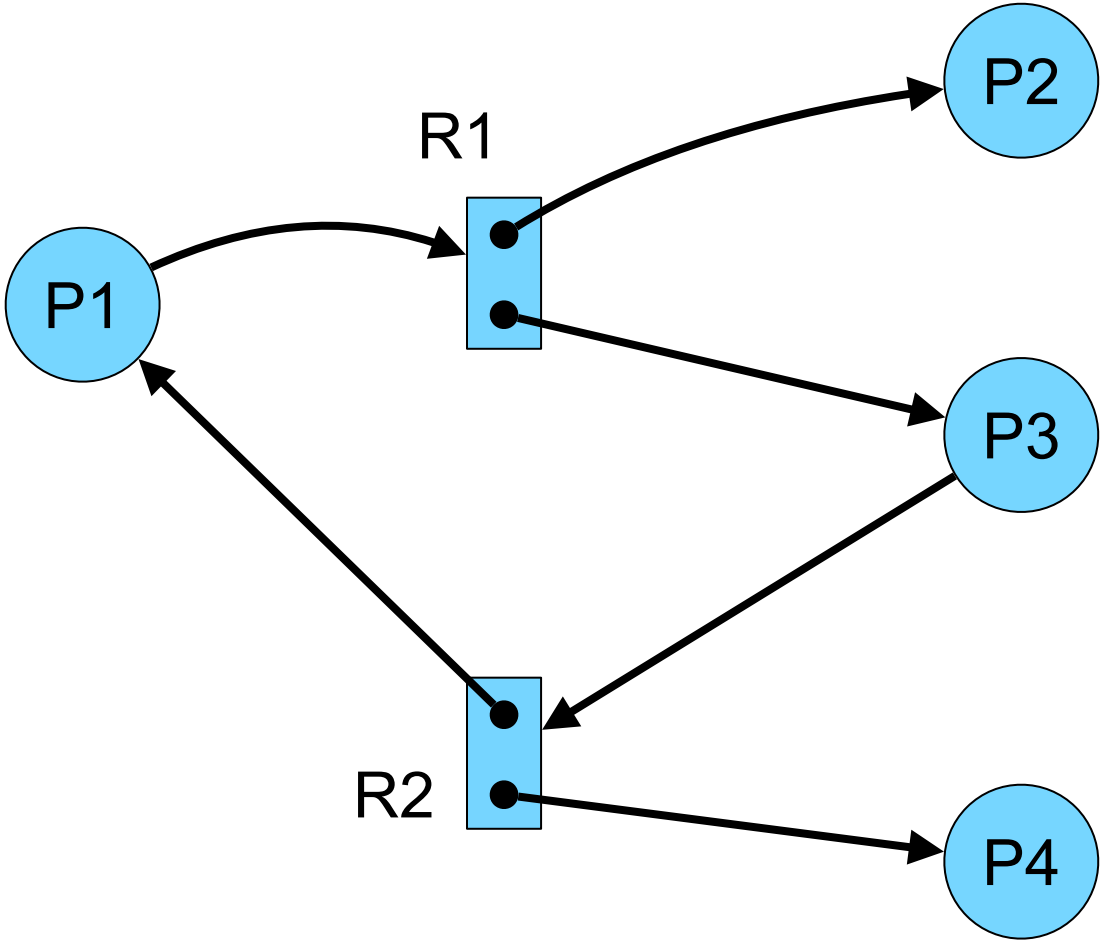
No process can make any progress due to at least one outgoing resource request edge

We have a **deadlock**



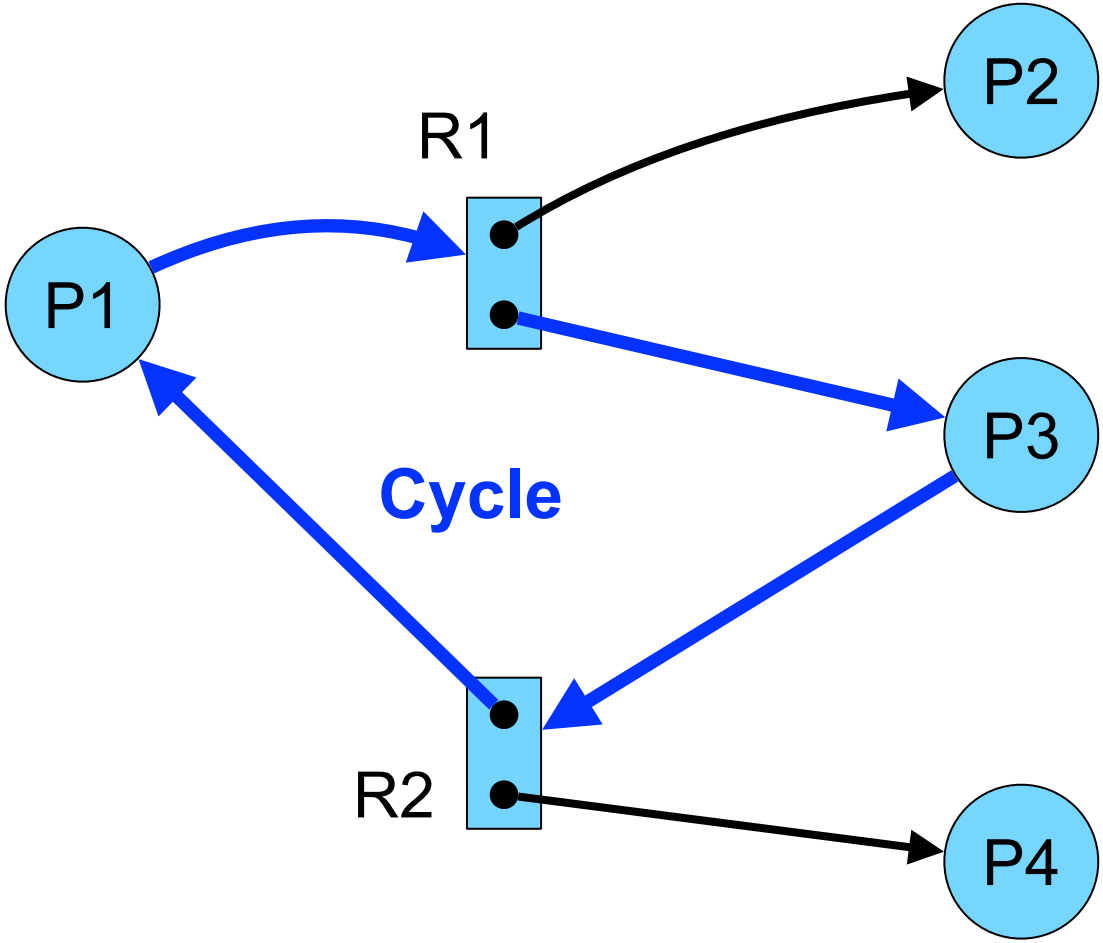
Example from Operating Systems Concepts textbook, Silberschatz et al.

Another example



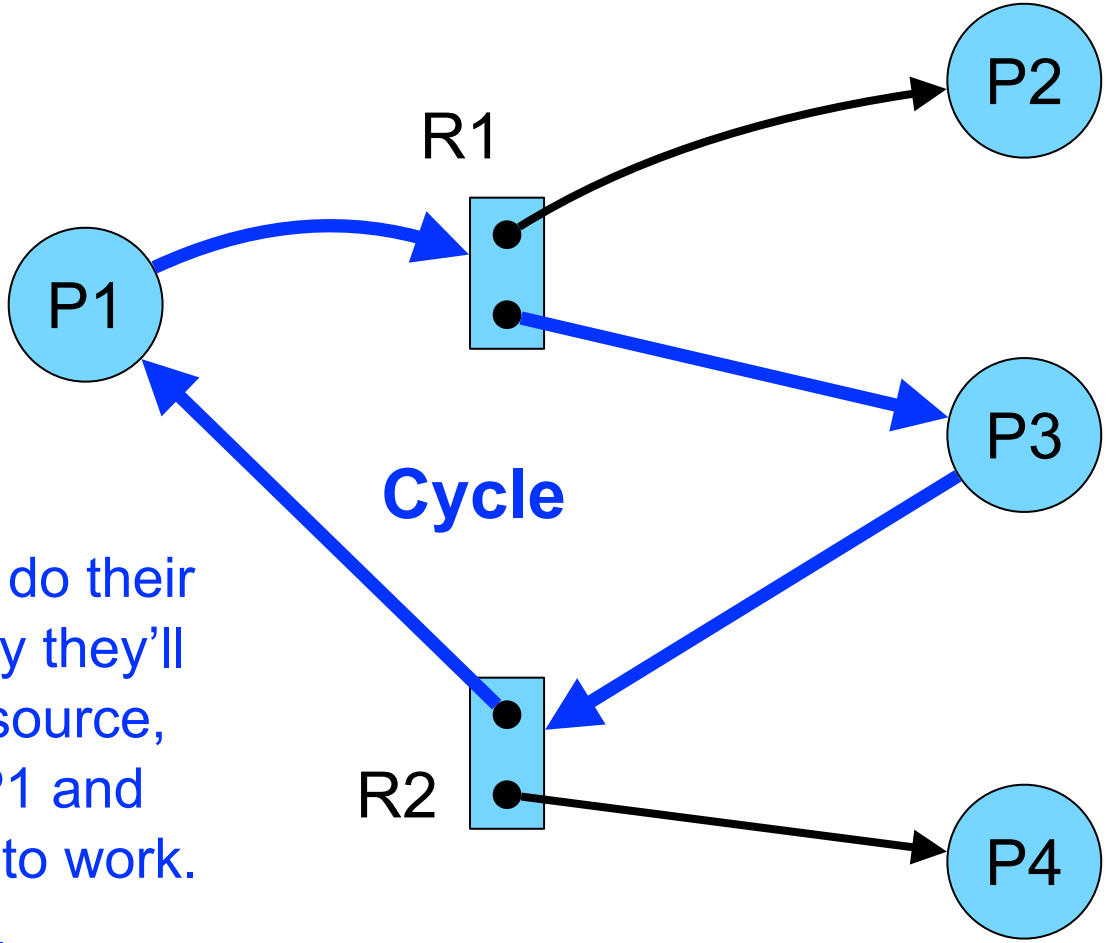
Example from Operating Systems Concepts textbook, Silberschatz et al.

Another example



Example from Operating Systems Concepts textbook, Silberschatz et al.

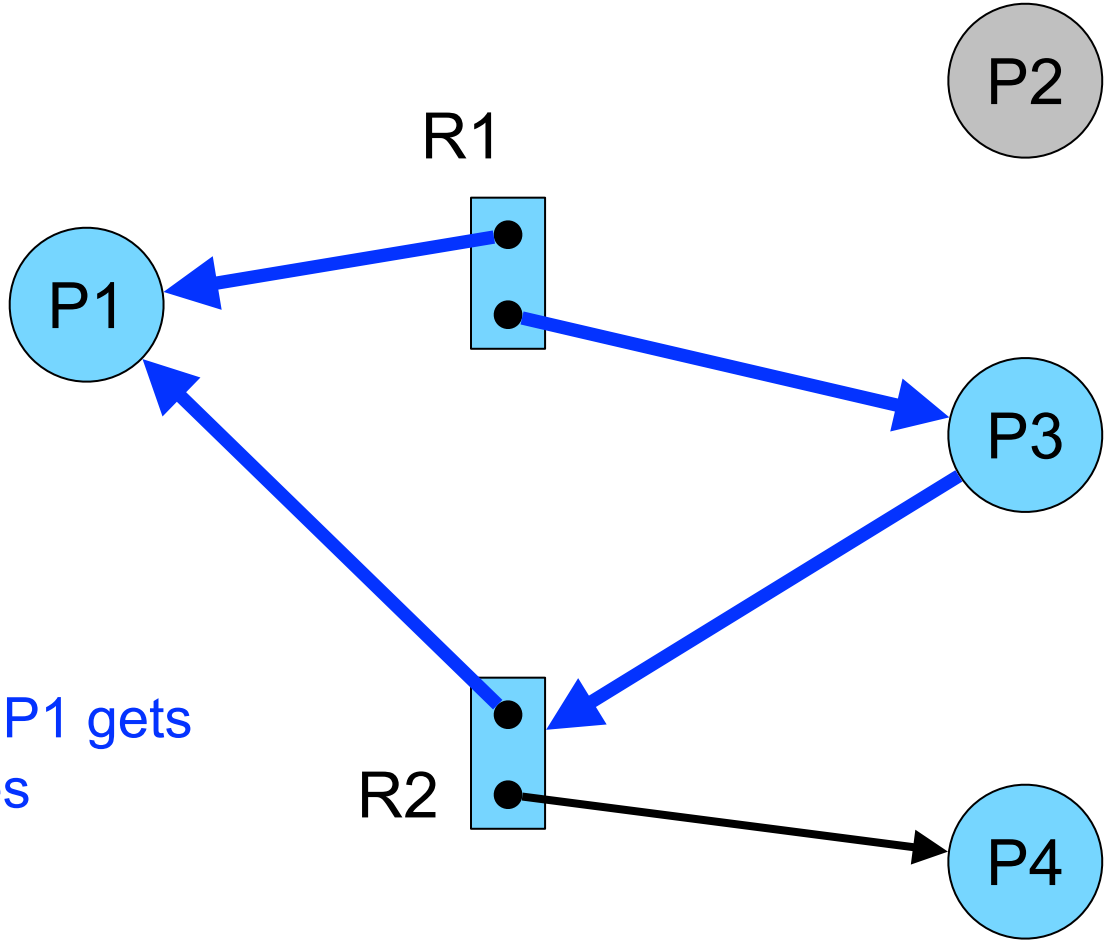
Another example



P2 and P4 can do their work, eventually they'll release one resource, which means P1 and P3 will be able to work. No deadlock….

Example from Operating Systems Concepts textbook, Silberschatz et al.

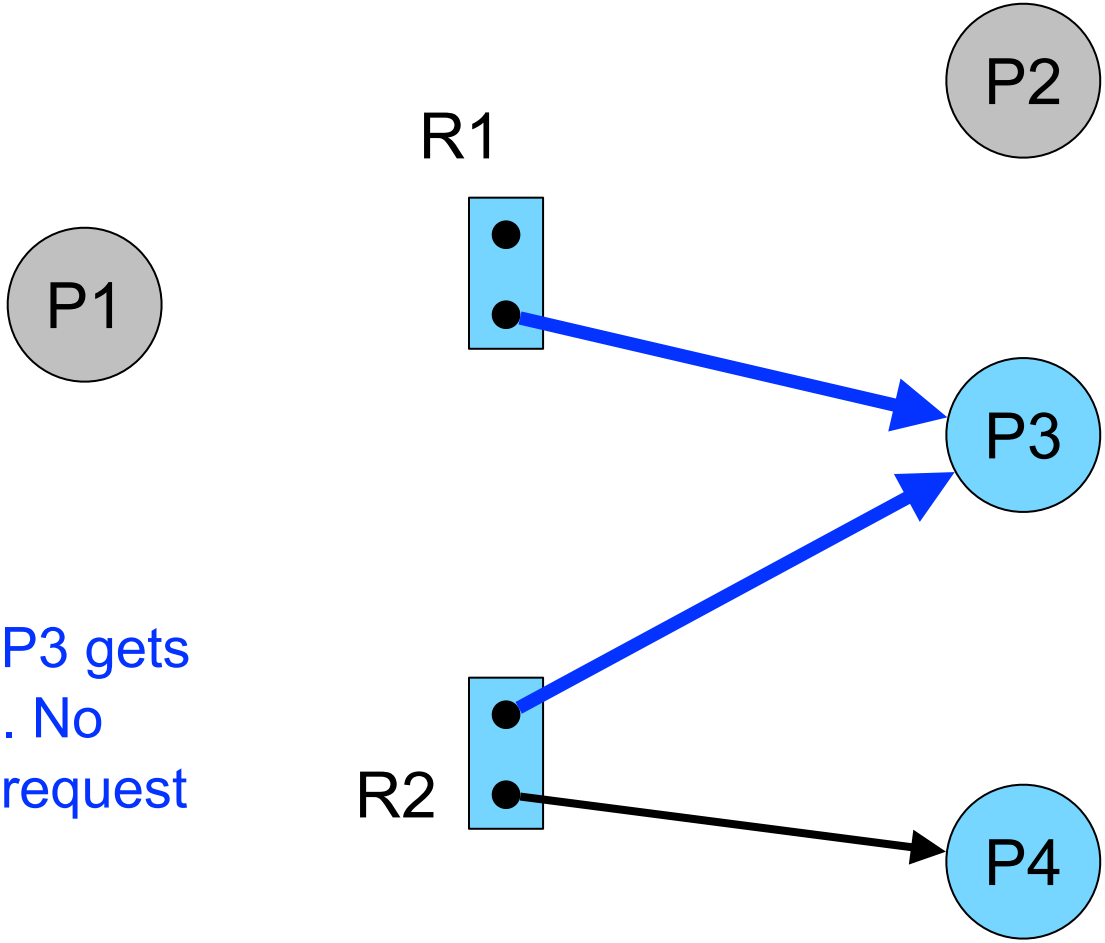
Another example



P2 terminates, P1 gets its R1 resources

Example from Operating Systems Concepts textbook, Silberschatz et al.

Another example



P1 terminates, P3 gets its R2 resource. No more resource request edges.

Example from Operating Systems Concepts textbook, Silberschatz et al.

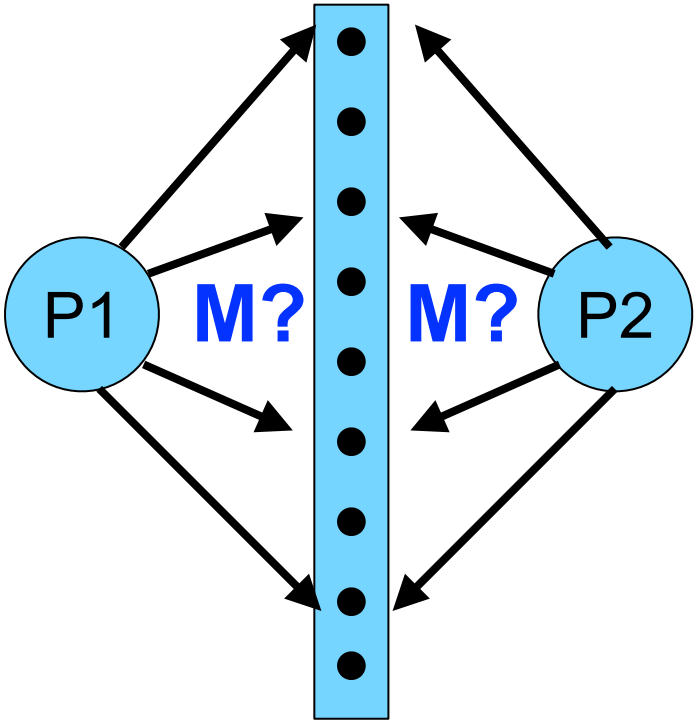
In-Class Exercise

- 9 locks
- 2 threads, each running:

```
while (true) {
  for (i=0;i < M; i++) {
    <acquire one lock>
  }

  // do something useful

  for (i=0;i < M; i++) {
    <release one lock>
  }
}
```



Question: What is the largest value of M that leads to no deadlock?

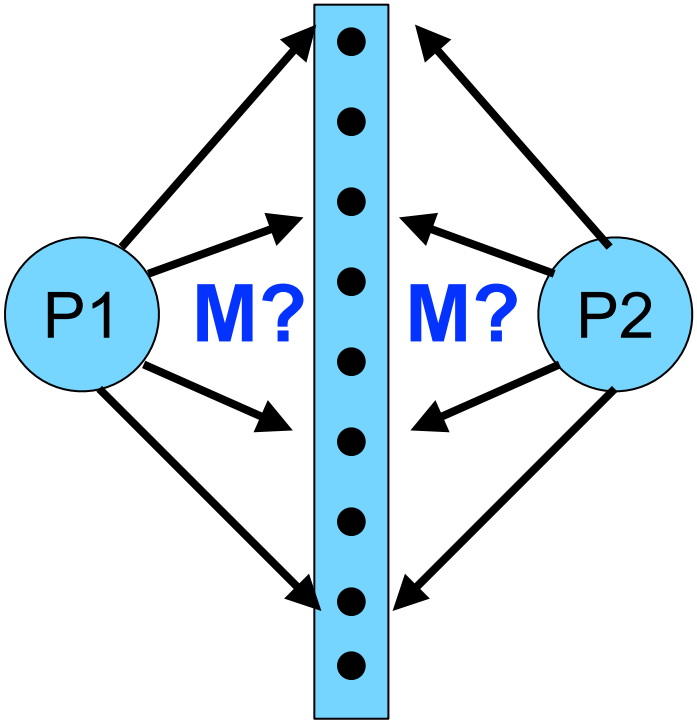
In-Class Exercise

- 9 locks
- 2 threads, each running:

```
while (true) {
  for (i=0;i < M; i++) {
    <acquire one lock>
  }

  // do something useful

  for (i=0;i < M; i++) {
    <release one lock>
  }
}
```



Answer: M=5

If both threads split the locks 4-4, which is the most “dangerous” situation, then one of them will get its 5th and last lock, and we’re ok

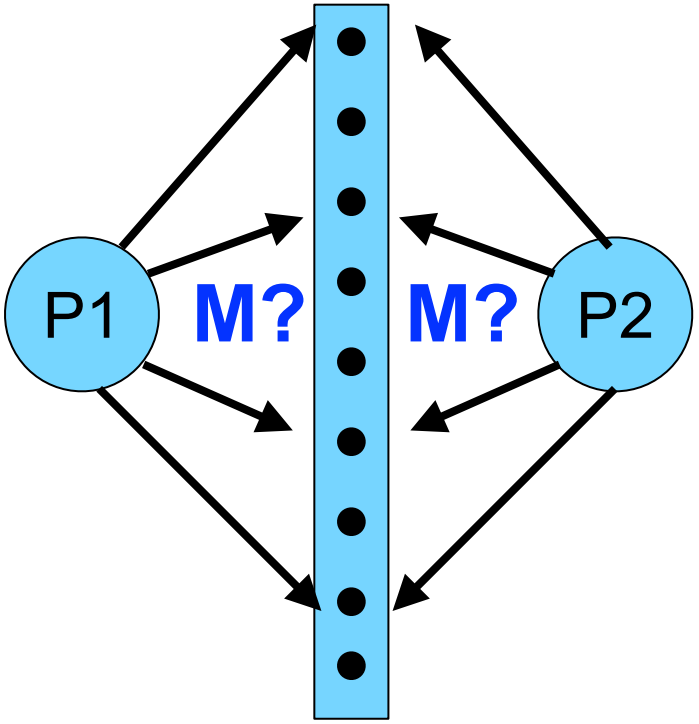
In-Class Exercise

- 9 locks
- 2 threads, each running:

```
while (true) {
  for (i=0;i < M; i++) {
    <acquire one lock>
  }

  // do something useful

  for (i=0;i < M; i++) {
    <release one lock>
  }
}
```



Deadlock for M=6

One thread holds 4 locks, the other holds 5 locks, and we're stuck

Strategies Against Deadlocks

- Prevention — Just build all programs so that at least one of the previous 3 necessary conditions can never be true, a by design approach

- Avoidance — If we are aware of the resources that the processes/threads will use, we could avoid deadlocks, more of a watchdog approach

- Detection and recovery — Use algorithms to detect whether a deadlock has happened and try to recover: a let's fix it approach

Deadlock Prevention ("by design")

- Removing necessary condition #1 (Mutual Exclusion: "At least one resource is non-shareable")
  - Non-shareable resources are too useful to disallow them!
  - A critical section protected by locks, a file open for writing, etc.
- Removing necessary condition #2 (No Preemption: "Resources cannot be forcibly removed")
  - But how do we even program in an environment in which an acquired resource can be taken away at any time?
- Removing necessary condition #3 (Circular Wait)
  - This can be done, e.g., by imposing an ordering on the resources and force processes to acquire them in that order
  - FreeBSD provides an order-verifier for locks (called witness)
  - Lock acquisition order is recorded, and locking locks out of order causes errors/ warnings
  - Useful, but not feasible for all programs
- **Bottom Line:** Deadlock prevention is appealing, but isn't done much at all in practice, save for a few programs that use ordered locks

Deadlock Avoidance ("watchdog")

- One approach:
  - The OS maintain the resource-allocation-request graph at all times
  - Whenever a process requests a resource, the OS determines whether giving that resource to the process would create a cycle in the graph
  - If it would, then reject the request, otherwise, add an edge
  - In a nutshell: never add an edge that would create a cycle
  - Detecting a cycle in a graph with n vertices is usually $O(n^2)$ (i.e., relatively expensive)
  - This approach is sometimes known as a "Graph-based Avoidance Algorithm"
- There are other approaches (e.g., see "Deadlock Avoidance via Scheduling" in OSTEP if you're curious)

- **Bottom Line:** Deadlock avoidance is an interesting idea, but it isn't really done in practice

Deadlock Detection/Recovery ("let's fix it")

- Detection:
  - Use an algorithm to determine whether we're in a deadlock state
  - If only one resource (black dot) per resource type, easy
    - Build the resource-allocation-request graph, and if it has a cycle, we have a deadlock
  - If more than one resource per resource type, harder
    - Use the Banker's Algorithm
  - This takes time, so we can only do this occasionally
- Recovery:
  - Option #1: Process termination
    - Option A: Kill all deadlocked processes
    - Option B: Kill one deadlocked process at a time until no deadlock
    - Dangerous program behaviors are then likely :(
  - Option #2: Resource preemption
    - Select a resource to be preempted
    - Rollback the process that has it (Simplest: Restart the process from scratch; Harder: "Go back till before the lock was acquired")
- **Bottom Line:** these are interesting ideas, but no OS does them

So what do OSes do?

- What do OSes do to help us with deadlocks???

So what do OSes do?

- What do OSes do to help us with deadlocks??? **NOTHING**
- Apparently we can live with this!?!

- Eventually, but very rarely, the deadlock may snowball until the system no longer functions and requires manual intervention (a reboot)
- But typically they remain confined to a program
- Deadlocks occur frequently-ish, and you get no help besides “make sure your code doesn’t have deadlocks”

- In the end there is no good one-size-fits-all solution, as there is no telling what kind of concurrent applications people will be developing

Priority Inversion

- A famous “OS and Deadlocks” problem
  - Assume that there are 3 processes with different priorities: L < M < H
  - H needs a resource currently held by L
  - If M becomes runnable, it will preempt L from running
  - Therefore L will never release the resource
  - And therefore H will never run
  - M has indirectly set the priority of H to the priority of L (since H has to wait for L to release the resource)
- This is called priority inversion
  - Lookup ”Mars Pathfinder priority inversion” for an interesting anecdote
- Solution → priority inheritance: If a process requesting a resource has higher priority than the process locking the resource, the process locking the resource is temporarily given the higher priority.
- This is one thing that some OSes (real-time OSes in particular) implement for you!

Conclusion

- Deadlocks happen when processes/threads wait indefinitely on each other to release resources (e.g., locks)
- Three methods to deal with deadlocks
  - (i) Prevention
  - (ii) Avoidance
  - (iii) Detection/Recovery
- None of them are used much in practice typically, and OSes do nothing
- Bottom line: just be smart and develop software that does not deadlock