Deadlock & Dining with my Phils

Computer Operating Systems, Fall 2025

Instructors: Joel Ramirez

Head TAs: Maya Huizar, Akash Kaukuntla, Vedansh Goenka, Joy Liu

TAs:

Eric Zou	Joseph Dattilo	Aniket Ghorpade	Shriya Sane
Zihao Zhou	Eric Lee	Shruti Agarwal	Yemisi Jones
Connor Cummings	Shreya Mukunthan	Alexander Mehta	Raymond Feng
Bo Sun	Steven Chang	Rania Souissi	Rashi Agrawal
Sana Manesh			

Poll Everywhere

pollev.com/cis5480

- Any planned courses for Spring 2026? Any Questions about PennOS?

Administrivia

- PennOS
  - Groups have been assigned
  - TA’s have been assigned to groups
  - You have the first milestone, due by the end of Next Week!
    - **Shouldn’t be too bad, just a general guide line…**
  - Your group (or at least most of your group) needs to meet with your assigned TA and display the expectations laid out in the PennOS Specification

Administrivia

- PennOS
  - Groups have been assigned
  - TA's have been assigned to groups
  - You have the first milestone, due by the end of Next Week!
    - **Shouldn't be too bad, just a general guide line…**
  - Your group (or at least most of your group) needs to meet with your assigned TA and display the expectations laid out in the PennOS Specification

- Github Gradescope Repo Creator is up!
  - Make sure to adhere to the spec with how to make it!
    - groupnum
    - Account1
    - Account2
    - ….etc

Administrivia

- PennOS
  - Groups have been assigned
  - TA’s have been assigned to groups
  - You have the first milestone, due by the end of Next Week!
    - **Shouldn’t be too bad, just a general guide line…**
  - Your group (or at least most of your group) needs to meet with your assigned TA and display the expectations laid out in the PennOS Specification
- Github Gradescope Repo Creator is up!
  - Make sure to adhere to the spec with how to make it!
    - groupnum
    - Account1
    - Account2
    - ….etc

Once you make a REPO with your group, that is your group. No switching after that. Just come to terms with it…….

Administrivia

- First Milestone
  - Make sure to reach out to your TAs with scheduling logistics by Monday.
  - Please, do not do it last minute as you will be penalized. Do not make this a habit.

Administrivia

- PennOS Advice:
  - Will announce this on Ed as well
  - In your FAT code you may do something like this:

```
lseek(FAT_FD, offset, SEEK_SET);
write(FAT_FD, contents, size);
```

    - Sometimes though, the write and lseek will return a success, but it won't actually write to your file system
    - Most commonly happens with blocks near the end of the FAT
      (as in blocks not in the allocation table but show up shortly after the end of the allocation table)
    - Most likely related to an issue between mmap and write (And empty bytes....)
    - Shows up inconsistently!
    - What's the fix?
      Just do it twice, that usually fixes it.

```
lseek(FAT_FD, offset, SEEK_SET);
write(FAT_FD, contents, size);
lseek(FAT_FD, offset, SEEK_SET);
write(FAT_FD, contents, size);
```

Poll Everywhere

pollev.com/cis5480

- Any planned courses for Fall 2025? Any Questions about PennOS?

Deadlock Prevention Summary

- Prevent deadlocks by removing any one of the four deadlock preconditions
- But eliminating even one of the preconditions is often hard/impossible
  - Mutual Exclusion is necessary in a lot of situations
  - Forcing a lower priority process to release resources early requires rollback of execution
  - Not always possible to know all resources that an operating system or process will use upfront

Lecture Outline

- **Dining Philosophers**
- Deadlock Handling

Dining Philosophers

- Assume the following situation
  - There are N philosophers that are trying to eat rice. (Computer Scientists here...)
  - They only have one chopstick each!
    - ***Need two chopsticks to eat ☹***
  - Alternate between two states:
    - Thinking
    - Eating
  - They are arranged in a circle with a chopstick between each of them
  - ***6 Chopsticks, 6 Philosphers in this example.***

Dining Philosophers

- Philosophers have good table manners
  - **Must acquire two chopsticks to eat**
  - Only one philosopher can have a chopstick at a time
- Useful abstraction / "standard problem" try to achieve:
  - Deadlock Free
    - No state where no one gets to eat
  - Starvation Free
    - Solution guarantees that all philosophers occasionally eat
    - Ideally maximize parallel eating
    - Most difficult to solve.

Dining with my Philz

```
pthread_mutex_t chopsticks[NUM_MUTEX];
```

```
void *start(void *arg){
    int id = *((int*)arg);
    while(true){
        think((int)id);
        eat(id);
    }
}
```

`start` is the entry point for all N threads. They all share N mutexes.

Each thread prints, then thinks, then prints that they'd like to eat.

Let's look at our preliminary `eat()` function.

```
void think(int id){
    printf("Phil %d, is about to think.\n", id);
    printf("Phil %d, is done thinking, time to eat.\n", id);
}
```

First Solution Attempt

- If we number each philosopher 0 – N and then each chopstick is also 0 – N, we can model the problem with mutexes, each chopstick is a mutex and each philosopher is a thread
  - To eat, thread I must acquire lock **`id (Left)`** and **`id + 1 (Right)`**
  - This ensures that each chopstick is only in use by one philosopher at a time

```
void eat(int id){
   pthread_mutex_lock(&chopsticks[id]);
   pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);


   printf("Phil %d, is about to eat!!.\n", id);
   pthread_mutex_unlock(&chopsticks[(id + 1) % NUM_MUTEX]);
   pthread_mutex_unlock(&chopsticks[id]);
}
```

Dining with my Philz, walk through.

- Let’s go ahead and go through an example…
  - Reminder: we number each philosopher 0 – N and then each chopstick is also 0 – N

```
pthread_mutex_lock(&chopsticks[id]);                  (A)
pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]); (B)
```

T-id = 0 T-id = 1 T-id = 2

0 1 2

Dining with my Philz, walk through.

- Let’s go ahead and go through an example…
  - Reminder: we number each philosopher 0 – N and then each chopstick is also 0 – N

```
pthread_mutex_lock(&chopsticks[id]);
pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);
```
(A)
(B)

0 1 2

T-id = 0 T-id = 1 T-id = 2

(A)

(A)

Dining with my Philz, walk through.

- Let's go ahead and go through an example…
  - Reminder: we number each philosopher 0 – N and then each chopstick is also 0 – N

```
pthread_mutex_lock(&chopsticks[id]);                    (A)
pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);  (B)
```

0 1 2

T-id = 0	T-id = 1	T-id = 2
	(A)	
		(A)

Dining with my Philz, walk through.

- Let's go ahead and go through an example…
  - Reminder: we number each philosopher 0 – N and then each chopstick is also 0 – N

```
    pthread_mutex_lock(&chopsticks[id]);
    pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);
```
(A)
(B)

0 1 2

T-id = 0 T-id = 1 T-id = 2

(A)

(A)
(B) This makes thread 2, lock mutex 0.

*(B)

*(A)

* means blocked..

Dining with my Philz, walk through.

- Now, Thread 1 and 0 is blocked!
  - Reminder: we number each philosopher 0 – N and then each chopstick is also 0 – N

```
pthread_mutex_lock(&chopsticks[id]);                    (A)
pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);  (B)
```

0 1 2

T-id = 0	T-id = 1	T-id = 2	
	(A)		
		(A)	
		(B)	This makes thread 2, lock mutex 0.
	*(B)		
*(A)			

* means blocked..

Poll Everywhere

discuss

- To unblock unblock thread 1
  - What needs to happen?

```
    pthread_mutex_unlock(&chopsticks[(id + 1) % NUM_MUTEX]);   (C)
    pthread_mutex_unlock(&chopsticks[id]);                     (D)
```

0 1 2

T-id = 0	T-id = 1	T-id = 2
	(A)	
		(A)
		(B) This makes thread 2, lock mutex 0.
	*(B)	
*(A)		

* means blocked..

Poll Everywhere

pollev.com/cis5480

- What is the worse case scenario here?
  - Reminder: we number each philosopher 0 – N and then each chopstick is also 0 – N

```
    pthread_mutex_lock(&chopsticks[id]);                    (A)
    pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);  (B)
```

T-id = 0 T-id = 1 T-id = 2

0 1 2

Second Attempt: Round Robin

- Our first attempt deadlocks.
- What if we instead we tried doing this “round robin”, we pass around a token that says “it is your turn to eat”
- Can this deadlock?
- What issues arise with this solution?

Second Attempt: Round Robin

- Our first attempt deadlocks.

- What if we instead we tried doing this “round robin”, we pass around a token that says “it is your turn to eat”

- Can this deadlock?

  No

- What issues arise with this solution?

Not parallel, just sequential eating ☹
Everyone guaranteed gets to eat though ☺

Third Attempt: Global Mutex

- What if instead, we add another "global" mutex that controls permission to pick up chopsticks. Once a philosopher has chopsticks, they can release the lock before they eat

- Can this deadlock?

- What issues arise with this solution?

Poll Everywhere

pollev.com/cis5480

- Can this deadlock? What issues arise with this solution?
- chopstick_gaurd is a global mutex

```
void eat(int id){
    pthread_mutex_lock(&chopstick_gaurd);

    pthread_mutex_lock(&chopsticks[id]);
    pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);

    pthread_mutex_unlock(&chopstick_gaurd);

    printf("Phil %d, is about to eat!!.\n", id);
    pthread_mutex_unlock(&chopsticks[(id + 1) % NUM_MUTEX]);
    pthread_mutex_unlock(&chopsticks[id]);
}
```

Fourth Attempt: More Human Approach

- What if instead, if a philosopher fails to get a chopstick, it puts down any chopsticks it has, waits for a little bit and then tries again?

- Can we do this in code?
  - `pthread_mutex_trylock`: if the lock can't be acquired, return immediately
  - `pthread_mutex_timedlock`: timeout after trying to get a mutex for some specified amount of time

- Can this deadlock?
- What issues arise with this solution?

An example of a final question; Say we replace all `lock` with `trylock` making sure to unlock if we can't acquire both locks, is it possible for all Philosophers to never eat? If it is, show how. If not, explain.

How will try lock spin?

- What is the worse case scenario here?
  - Reminder: we number each philosopher 0 – N and then each chopstick is also 0 – N

```
pthread_mutex_trylock(&chopsticks[id]);
pthread_mutex_trylock(&chopsticks[(id + 1) % NUM_MUTEX]);
```

(A)
(B)

0 1 2

T-id = 0	T-id = 1	T-id = 2
		(A)
	(A)	
(A)		
		^(B)
	^(B)	
^(B)		
		unlock 2
	unlock 1	
unlock 0		

Because each thread failed to grab their right hand chopstick, they release their left hand chopstick and try again!

^ indicates fail

How will try lock spin?

- What is the worse case scenario here?
  - Reminder: we number each philosopher 0 – N and then each chopstick is also 0 – N

```
    pthread_mutex_trylock(&chopsticks[id]);                     (A)
    pthread_mutex_trylock(&chopsticks[(id + 1) % NUM_MUTEX]);   (B)
```

0 1 2

T-id = 0	T-id = 1	T-id = 2
		(A)
	(A)	
(A)		
		^(B)
	^(B)	
^(B)		
		unlock 2
	unlock 1	
unlock 0		

Because each thread failed to grab their right hand chopstick, they release their left hand chopstick and try again!

^ indicates fail

How will try lock spin?

0 1 2

T-id = 0	T-id = 1	T-id = 2
(A)	(A)	(A)
^(B)	^(B)	^(B)
unlock 0	unlock 1	unlock 2
(A)	(A)	(A)
^(B)	^(B)	^(B)
unlock 0	unlock 1	unlock 2

This is a plausible scenario in which each thread, although not deadlocked because resources are eventually released, continues to run and later attempts to reacquire the locks. :/

You ***sorta*** hope that the schedular makes this really really really really unlikely…

But still not impossible.

^ indicates fail

Fifth Attempt: Break the "Symmetry"

- What if the even numbered philosophers and odd numbered philosophers do things differently?
  - Even Numbered: Grab chopstick on their left and then right (Left handed folks)
  - Odd Numbered: Grab chopstick on their right and then left (Right handed folks)

- Can this deadlock?

- What issues arise with this solution?

Poll Everywhere

pollev.com/cis5480

- Is there any way the threads can deadlock now? Assume each thread unlocks in any order.
  - (Ask yourself, which lock will Thread 1 always try to lock first?
  - Who does this compete directly with?)

Threads: 0 & 2

```
pthread_mutex_lock(&chopsticks[id]);                          (A)
pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);        (B)
```

Thread: 1

```
pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);        (B)
pthread_mutex_lock(&chopsticks[id]);                          (A)
```

0 1 2

Poll Everywhere

pollev.com/cis5480

- Assume for the sake of contradiction that there's a deadlock
  - I'm just kidding. Let's walk through an example; we can enumerate them all.

Threads: 0 & 2

```
    pthread_mutex_lock(&chopsticks[id]);                     (A)
    pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);   (B)
```

Thread: 1

```
    pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);   (B)
    pthread_mutex_lock(&chopsticks[id]);                     (A)
```

0 1 2

Changing it up.

```
pthread_mutex_lock(&chopsticks[id]);                 (A)
pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]); (B)
```

Threads; 0 & 2

```
pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]); (B)
pthread_mutex_lock(&chopsticks[id]);                 (A)
```

Thread; 1

0 1 2

T-id = 0 T-id = 1 T-id = 2

(A) (A)

In this state, Thread 2 can not continue. As thread 0 has ownership of lock 0 (it’s right chop)

In this state, Thread 1 can not continue. As thread 2 has ownership of lock 2 (it’s right chop).

In this state, Thread 0 can continue. As thread 1 cannot grab lock 1, because it can’t grab lock 2! ***So no deadlock possible here.***

Changing it up.

```
    pthread_mutex_lock(&chopsticks[id]);                    (A)
    pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);  (B)
```

Threads; 0 & 2

```
    pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);  (B)
    pthread_mutex_lock(&chopsticks[id]);                    (A)
```

Thread; 1

0 1 2

T-id = 0	T-id = 1	T-id = 2
(A)		
		(A)

This is the same as the previous example.

Poll Everywhere

discuss

In this state, which thread(s) still have the opportunity to eat?

```
    pthread_mutex_lock(&chopsticks[id]);                        (A)
    pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);      (B)
```

Threads; 0 & 2

```
    pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);      (B)
    pthread_mutex_lock(&chopsticks[id]);                        (A)
```

Thread; 1

T-id = 0	T-id = 1	T-id = 2
(A)		
	(B)	

Poll Everywhere

discuss

In this state, which thread(s) still have the opportunity to eat?

```
    pthread_mutex_lock(&chopsticks[id]);                    (A)
    pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);  (B)
```

Threads; 0 & 2

```
    pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);  (B)
    pthread_mutex_lock(&chopsticks[id]);                    (A)
```

Thread; 1

0 1 2

T-id = 0	T-id = 1	T-id = 2
(A)		
	(B)	

Here, thread 2 can not continue because thread 1 grabbed lock 2 by executing (B) first!

Poll Everywhere

discuss

In this state, which thread(s) still have the opportunity to eat?

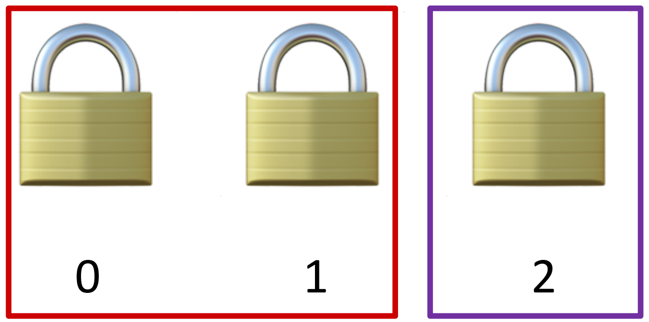
```
    pthread_mutex_lock(&chopsticks[id]);                    (A)
    pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);  (B)
```

Threads; 0 & 2

```
    pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);  (B)
    pthread_mutex_lock(&chopsticks[id]);                    (A)
```

Thread; 1

Thread 0! Thread 1



T-id = 0 T-id = 1 T-id = 2

(A)

(B)

(B)

Here, thread 2 can not continue because thread 1 grabbed lock 2 by executing **(B)** first!

Poll Everywhere

discuss

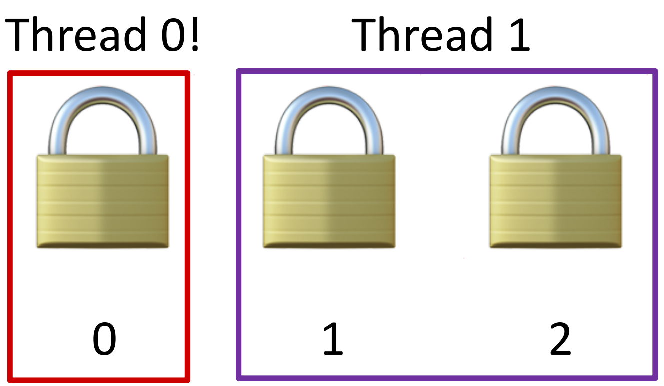
In this state, which thread(s) still have the opportunity to eat?

```
pthread_mutex_lock(&chopsticks[id]);
pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);
```
(A)
(B)

Threads; 0 & 2

```
pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);
pthread_mutex_lock(&chopsticks[id]);
```
(B)
(A)

Thread; 1



T-id = 0	T-id = 1	T-id = 2
(A)		
	(B)	
	(A)	

Here, thread 2 can not continue because thread 1 grabbed lock 2 by executing (B) first!

Fifth Attempt: Break the Symmetry

- What if the even numbered philosophers and odd numbered philosophers do things differently?
  - Even Numbered: Grab chopstick on their left and then right
  - Odd Numbered: Grab chopstick on their right and then left

- Can this deadlock?
  No

- What issues arise with this solution?

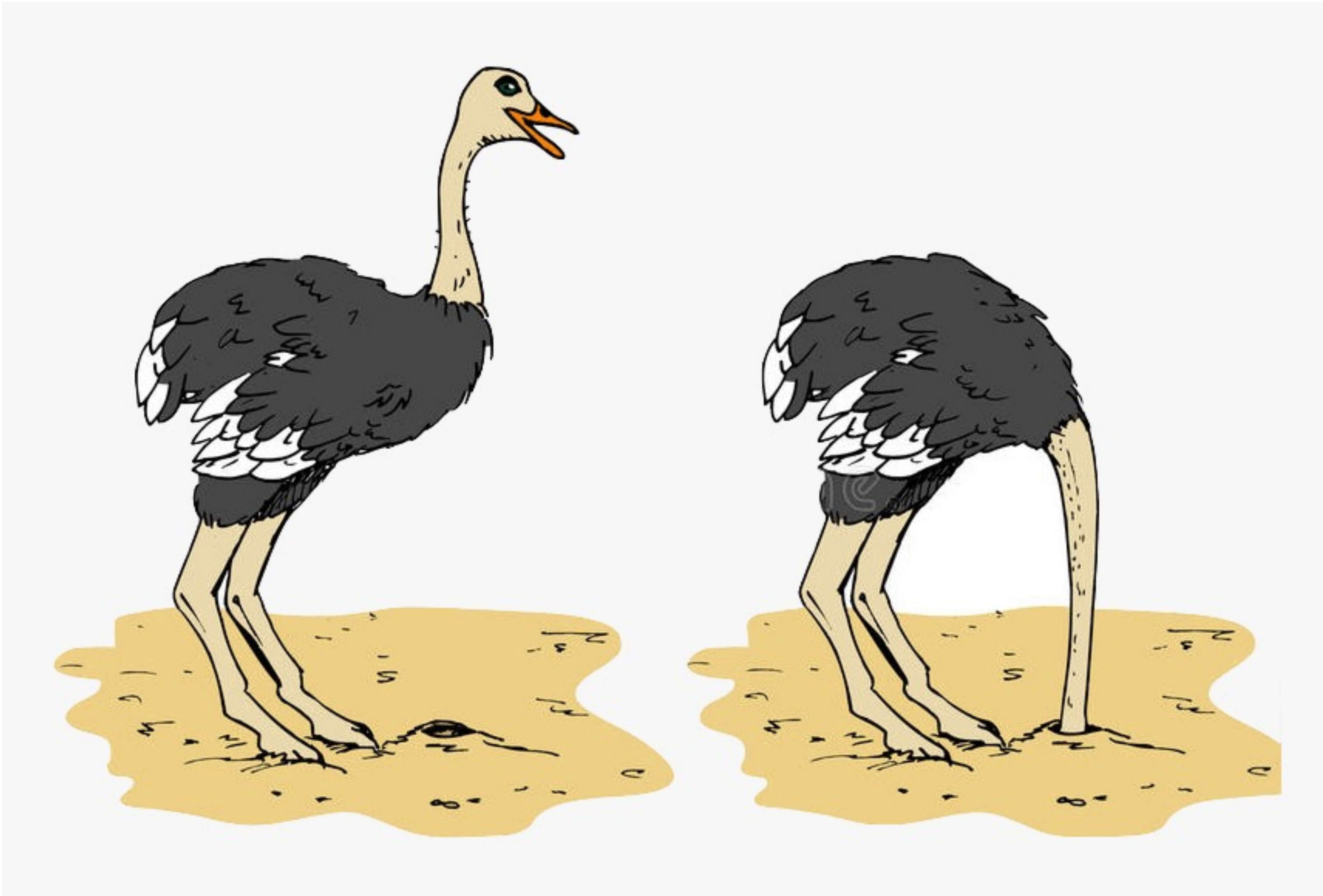
threads may still <u>possibly</u> starve

Lecture Outline

- Dining Philosophers
- **Deadlock Handling**

Deadlock Handling: Ostrich Algorithm

Deadlock Handling: Ostrich Algorithm



Ostriches don't actually do this, but it is an old myth

Deadlock Handling: Ostrich Algorithm

- Ignoring potential problems
  - Usually under the assumption that it is either rare, too expensive to handle, and/or not a fatal error

- Used in real world contexts, there is a real cost to tracking down every possible deadlock case and trying to fix it
  - Cost on the developer side: more time to develop
  - Cost on the software side: more computation for these things to do, slows things down

Deadlock Handling: Prevention

- Ad Hoc Approach
  - Key insights into application logic allow you to write code that avoids cycles/deadlock
  - Example: Dining Philosophers breaking symmetry with even/odd philosophers

- Exhaustive Search Approach
  - Static analysis on source code to detect deadlocks
  - Formal verification: model checking
  - Unable to scale beyond small programs in practice
    Impossible to prove for any arbitrary program (without restrictions)

Detection

- If we can't guarantee deadlocks won't happen, we can instead try to detect a deadlock just before it will happen **and then intervene**.

- Two big parts
  - Detection algorithm. This is usually done with tracking metadata and graph theory
  - The intervention/recovery. We typically want some sort of way to "recover" to a safe state when we detect a deadlock is going to happen

Detection Algorithms

- The common idea is to think of the threads and resources as a graph.
  - If there is a cycle: deadlock
  - If there is no cycle: no deadlock

- Finding cycles in a graph is a common algorithm problem with many solutions.

Poll Everywhere

pollev.com/cis5480

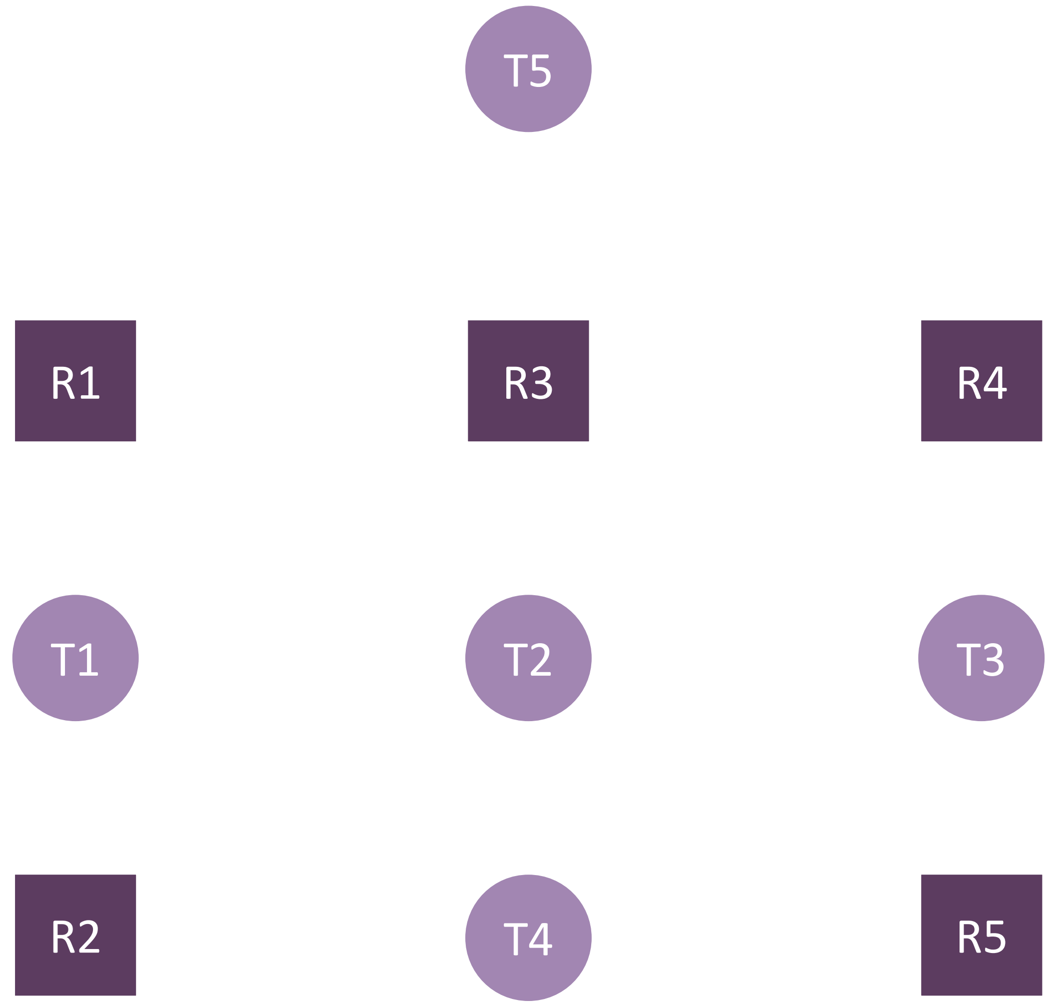
- Consider the following example with 5 threads and 5 resources that require mutual exclusion is this a deadlock?
  - Thread 1 has R2 but wants R1
  - Thread 2 has R1 but wants R3, R4 and R5
  - Thread 3 has R4 but wants R5
  - Thread 4 has R5 but wants R2
  - Thread 5 has R3

Resource Allocation Graph

- We can represent this deadlock with a graph:
  - Each resource and thread is a node
  - If a thread has a resource, draw an arrow pointing at the thread form that resource
  - If a thread wants to acquire a resource but can’t, draw an arrow pointing at the resource from the thread trying to acquire it

Resource Allocation Graph Example

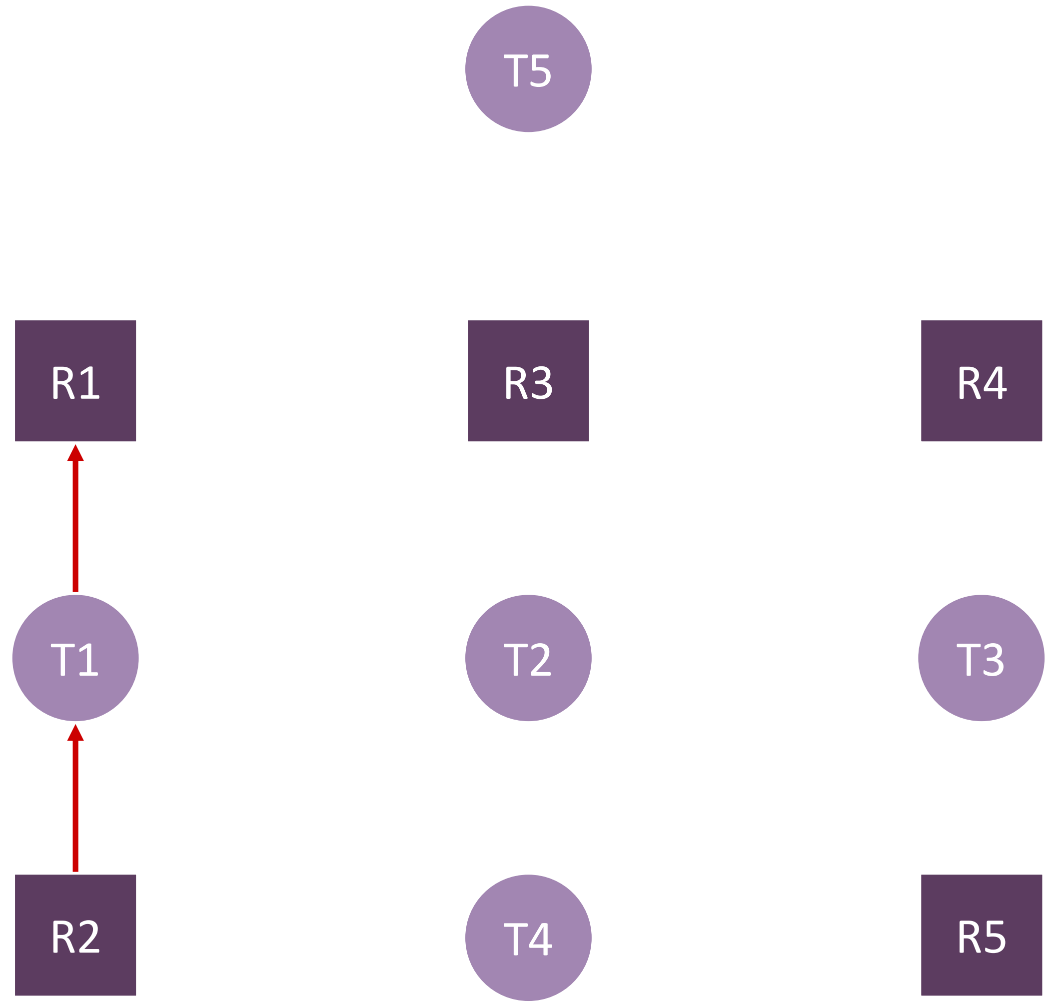
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

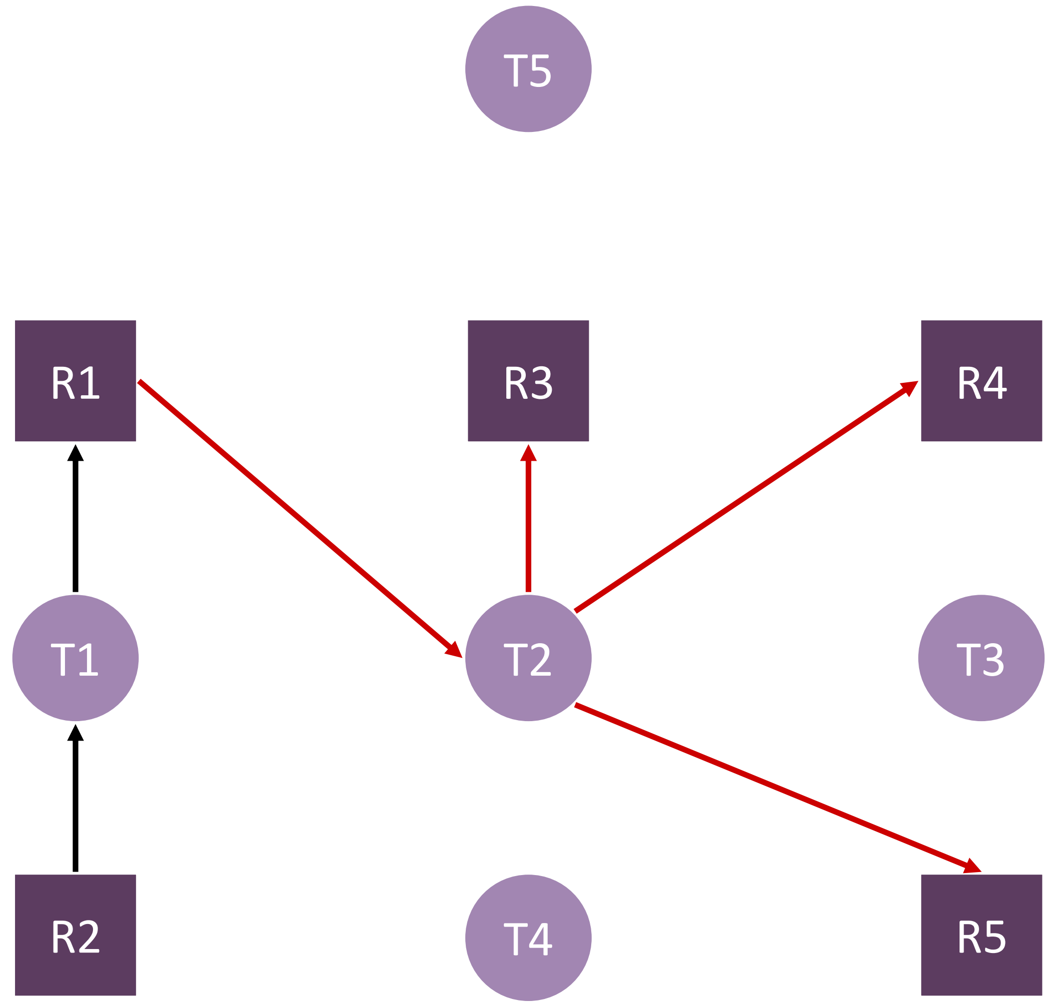
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

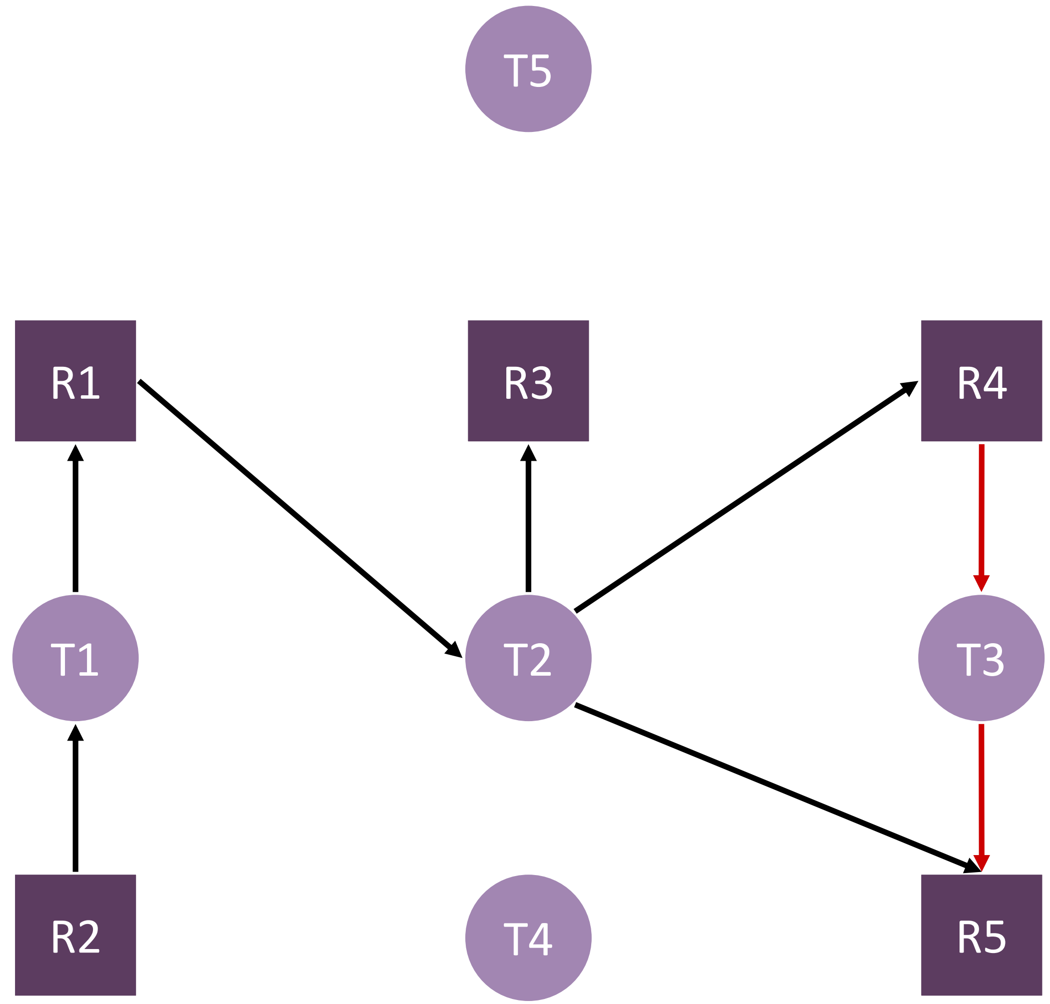
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

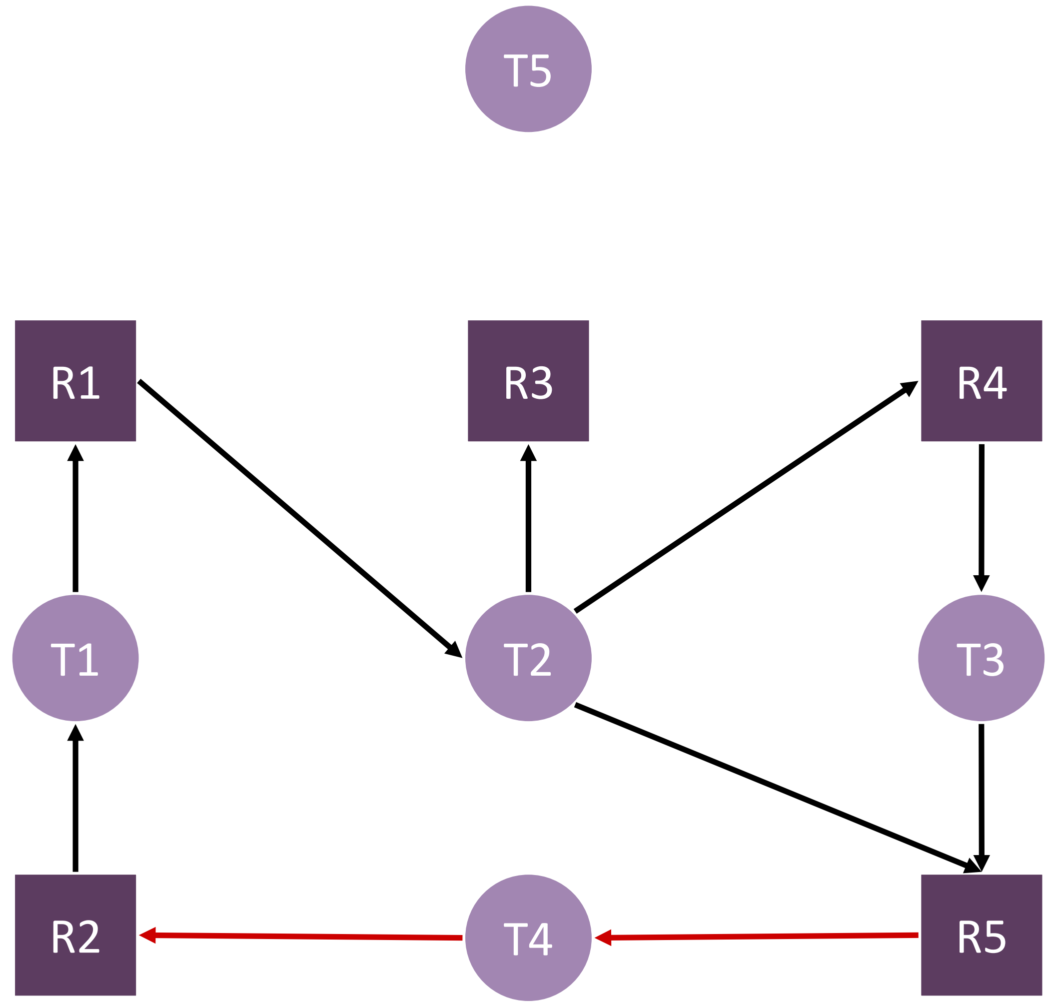
- Thread 1 has R2 but wants R1
- Thread 2 has R1 but wants R3, R4 and R5
- Thread 3 has R4 but wants R5
- Thread 4 has R5 but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

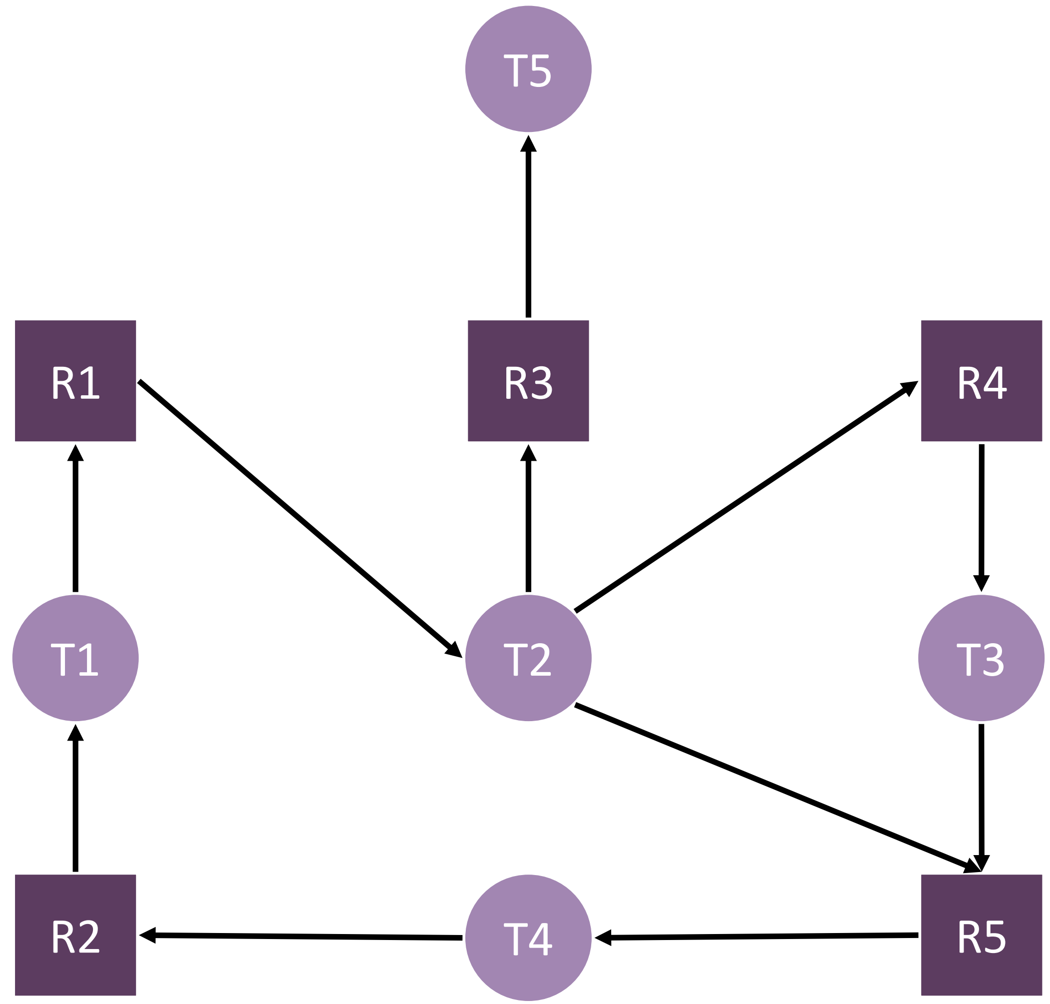
- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3



Resource Allocation Graph

Resource Allocation Graph Example

- Thread 1 has R2
  but wants R1
- Thread 2 has R1
  but wants R3, R4 and R5
- Thread 3 has R4
  but wants R5
- Thread 4 has R5
  but wants R2
- Thread 5 has R3

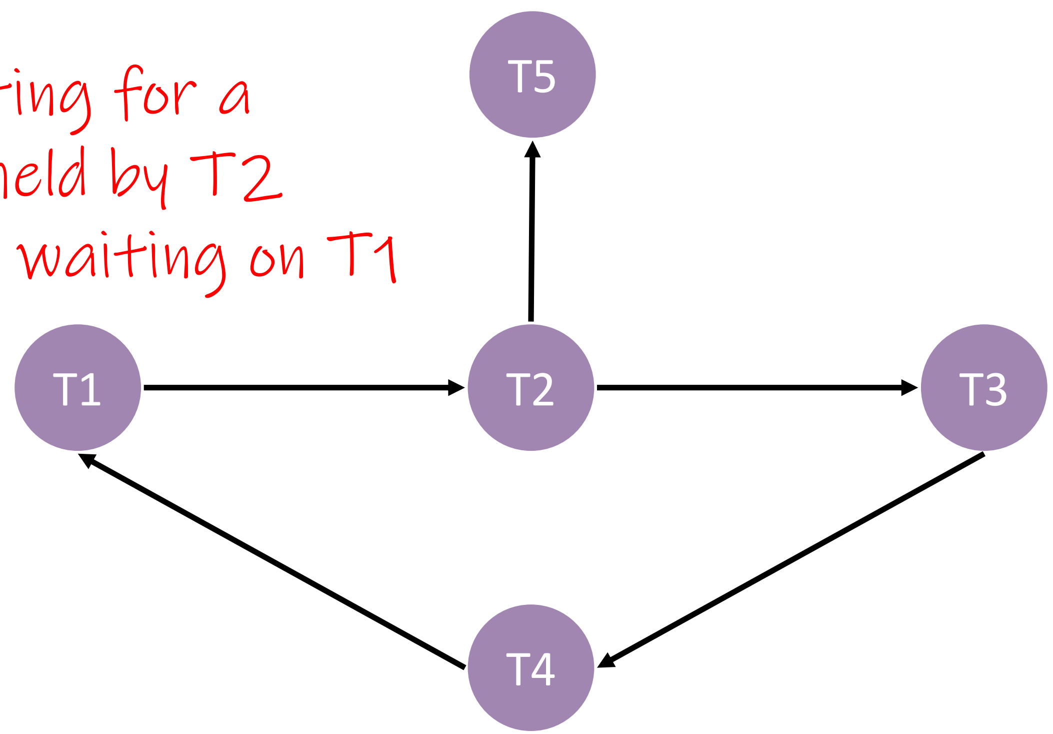


Resource Allocation Graph

Alternate graph

- Instead of also representing resources as nodes, we can have a "wait for" graph, showing how threads are waiting on each other

T1 is waiting for a
resource held by T2
and T4 is waiting on T1



Wait For Graph

Recovery after Detection

- Preemption:
  - Force a thread to give up a resource
  - Often is not safe to do or impossible
- Rollback:
  - Occasionally checkpoint the state of the system, if a deadlock is detected then go back to the checkpointed “Saved state”
  - Used commonly in database systems
  - Maintaining enough information to rollback and doing the rollback can be expensive
- Manual Killing:
  - Kill a process/thread, check for deadlock, repeat till there is no deadlock
  - Not safe, but it is simple

Overall Costs

- Doing Deadlock Detection & Recovery solves deadlock issues, but there is a cost to memory and CPU to store the necessary information and check for deadlock

- This is why sometimes the ostrich algorithm is preferred

Avoidance

- Instead of detecting a deadlock when it happens and having expensive rollbacks, we may want to instead avoid deadlock cases earlier

- Idea:
  - Before it does work, it submits a request for all the resources it will need.
  - A deadlock detection algorithm is run
    - If acquiring those resources would lead to a deadlock, deny the request. The calling thread can try again later
    - If there is no deadlock, then the thread can acquire the resources and complete its task
  - The calling thread later releases resources as they are done with them

Avoidance

- Pros:
  - Avoids expensive rollbacks or recovery algorithms

- Cons:
  - Can’t always know ahead of time all resources that are required
  - Resources may spend more time being locked if all resources need to be acquired before an action is taken by a thread, could hurt parallelizability
    - Consider a thread that does a very expensive computation with many shared resources.
    - Has one resources that is only updated at the end of the computation.
    - That resources is locked for a long time and other threads that may need it cannot access it

Aside: Bankers Algorithm

- This gets more complicated when there are multiple copies of resources, or a finite number of people can access a resources.

- The Banker's Algorithm handles these cases
  - But I won't go into detail about this
  - There is a video linked on the website under this lecture you can watch if you want to know more

Aside: Bankers Algorithm

- This gets more complicated when there are multiple copies of resources, or a finite number of people can access a resources.

- The Banker's Algorithm handles these cases
  - But I won't go into detail about this
  - There is a video linked on the website under this lecture you can watch if you want to know more

That’s all!

- Make sure to try out these examples in the dining.c linked in the schedule to solidify you’re understanding!
  - As practice, try implementing the **try_lock** solution!

Poll Everywhere

pollev.com/cis5480

- To unblock unblock thread 1
  - What needs to happen?

```
    pthread_mutex_unlock(&chopsticks[(id + 1) % NUM_MUTEX]);   (C)
    pthread_mutex_unlock(&chopsticks[id]);                     (D)
```

0 1 2

T-id = 0 T-id = 1 T-id = 2

(A)

(A)
(B) This makes thread 2, lock mutex 0.

*(B)

*(A)

(C)

* means blocked..

Poll Everywhere

pollev.com/cis5480

- To unblock unblock thread 1
  - What needs to happen?

```
    pthread_mutex_unlock(&chopsticks[(id + 1) % NUM_MUTEX]);   (C)
    pthread_mutex_unlock(&chopsticks[id]);                     (D)
```

0 1 2

T-id = 0	T-id = 1	T-id = 2
	(A)	
		(A)
		(B)
	* (B)	
* (A)		
		(C)
		(D)
	(B) Now thread 1, can lock mutex 2 ✅.	

* means blocked..

Poll Everywhere

pollev.com/cis5480

- What is the worse case scenario here?
  - Reminder: we number each philosopher 0 – N and then each chopstick is also 0 – N

```
    pthread_mutex_lock(&chopsticks[id]);                        (A)
    pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);      (B)
```

0 1 2

T-id = 0	T-id = 1	T-id = 2
		(A)

Poll Everywhere

pollev.com/cis5480

- What is the worse case scenario here?
  - Reminder: we number each philosopher 0 – N and then each chopstick is also 0 – N

```
    pthread_mutex_lock(&chopsticks[id]);                    (A)
    pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);  (B)
```

0 1 2

T-id = 0	T-id = 1	T-id = 2
		(A)
	(A)	

Poll Everywhere

pollev.com/cis5480

- What is the worse case scenario here?
  - Reminder: we number each philosopher 0 – N and then each chopstick is also 0 – N

```
pthread_mutex_lock(&chopsticks[id]);                 (A)
pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]); (B)
```

0 1 2

T-id = 0 T-id = 1 T-id = 2

(A)

(A)

(A)

Poll Everywhere

pollev.com/cis5480

- What is the worse case scenario here?
  - Reminder: we number each philosopher 0 – N and then each chopstick is also 0 – N

```
pthread_mutex_lock(&chopsticks[id]);                      (A)
pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);    (B)
```

0 1 2

T-id = 0	T-id = 1	T-id = 2
		(A)
	(A)	
(A)		
		*(B)
	*(B)	
*(B)		

Akin to saying all philz picked up their left chopstick at the same time!

Pure deadlock!

Poll Everywhere

pollev.com/cis5480

- Can this deadlock? What issues arise with this solution?
- `chopstick_gaurd` is a global mutex

```
void eat(int id){
    pthread_mutex_lock(&chopstick_gaurd);

    pthread_mutex_lock(&chopsticks[id]);
    pthread_mutex_lock(&chopsticks[(id + 1) % NUM_MUTEX]);

    pthread_mutex_unlock(&chopstick_gaurd);

    printf("Phil %d, is about to eat!!.\n", id);
    pthread_mutex_unlock(&chopsticks[(id + 1) % NUM_MUTEX]);
    pthread_mutex_unlock(&chopsticks[id]);
}
```

Picking up the chopsticks becomes an "atomic" operation.

Poll Everywhere

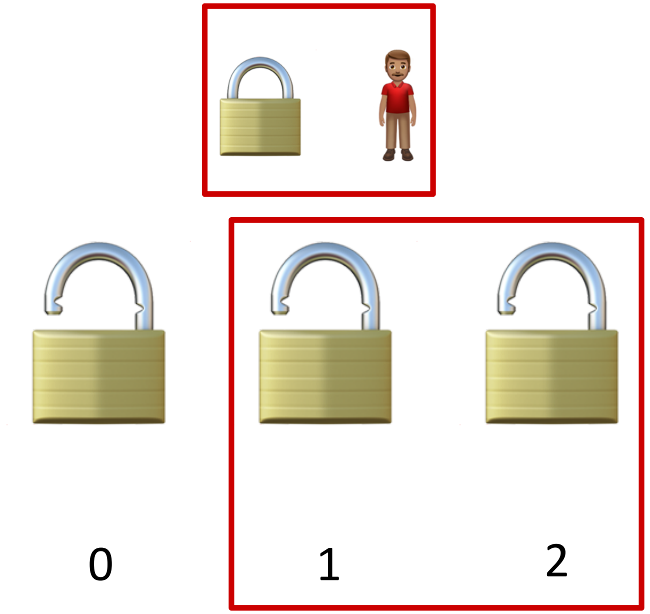
pollev.com/cis5480

This will be our chopstick guard

0 1 2

Poll Everywhere

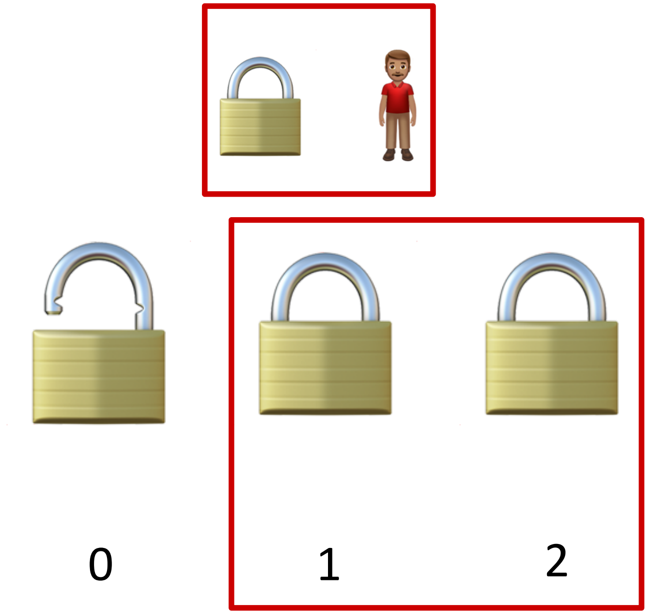
pollev.com/cis5480



- Thread 1 Acquires Lock Guard First!
- Thread 0 and 2 cannot proceed…
- Thread 1 then acquires lock 1 and 2.

Poll Everywhere

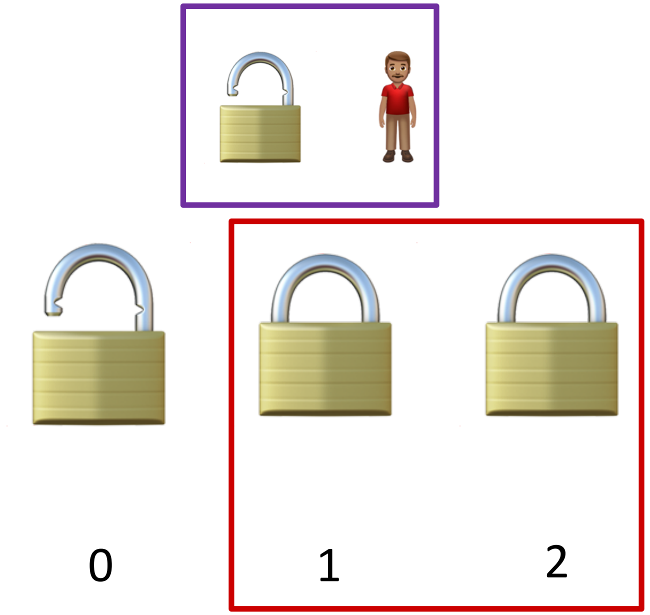
pollev.com/cis5480



- Thread 1 Acquires Lock Guard First!
- Thread 0 and 2 cannot proceed…
- Thread 1 then acquires lock 1 and 2.
- Thread 1 then releases our Lock Guard

Poll Everywhere

pollev.com/cis5480

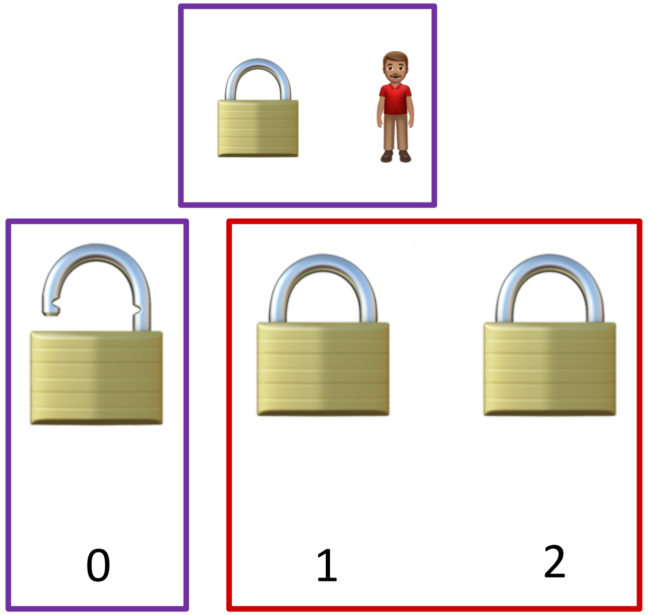


- Thread 1 Acquires Lock Guard First!
- Thread 0 and 2 cannot proceed…
- Thread 1 then acquires lock 1 and 2.
- Thread 1 then releases our Lock Guard

- While Thread 1 is eating, Thread 0 acquires the lock guard!
  - Thread 2 still doesn't progress.

Poll Everywhere

pollev.com/cis5480

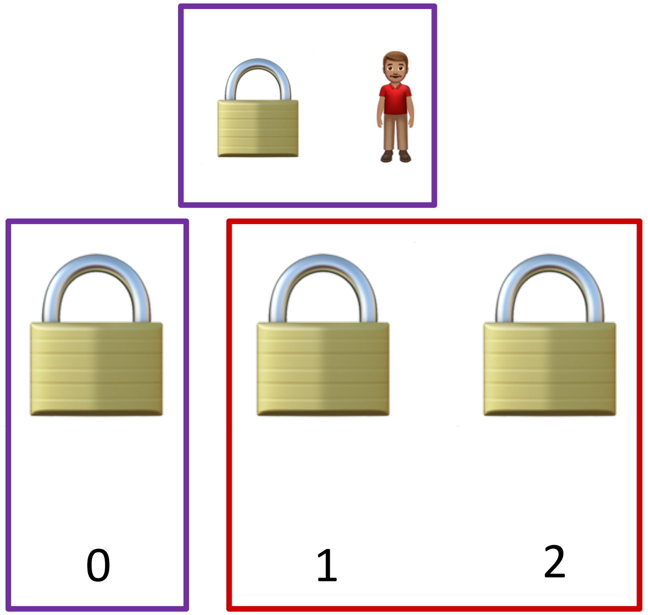


- Thread 1 Acquires Lock Guard First!
- Thread 0 and 2 cannot proceed…
- Thread 1 then acquires lock 1 and 2.
- Thread 1 then releases our Lock Guard

- While Thread 1 is eating, Thread 0 acquires the lock guard!
  - Thread 2 still doesn't progress.
- Thread 0 acquires lock 0!

Poll Everywhere

pollev.com/cis5480

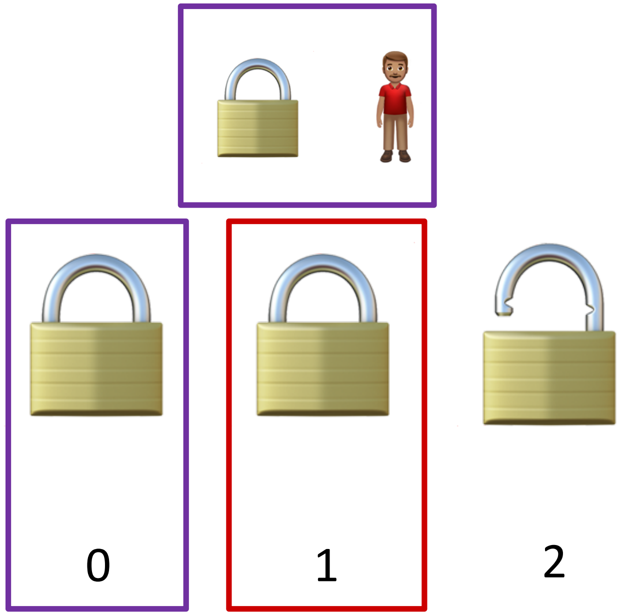


- Thread 1 Acquires Lock Guard First!
- Thread 0 and 2 cannot proceed…
- Thread 1 then acquires lock 1 and 2.
- Thread 1 then releases our Lock Guard

- While Thread 1 is eating, Thread 0 acquires the lock guard!
  - Thread 2 still doesn't progress.
- Thread 0 acquires lock 0!
- Thread 0 is stuck acquiring lock 1 as Thread 1 is eating!

Poll Everywhere

pollev.com/cis5480



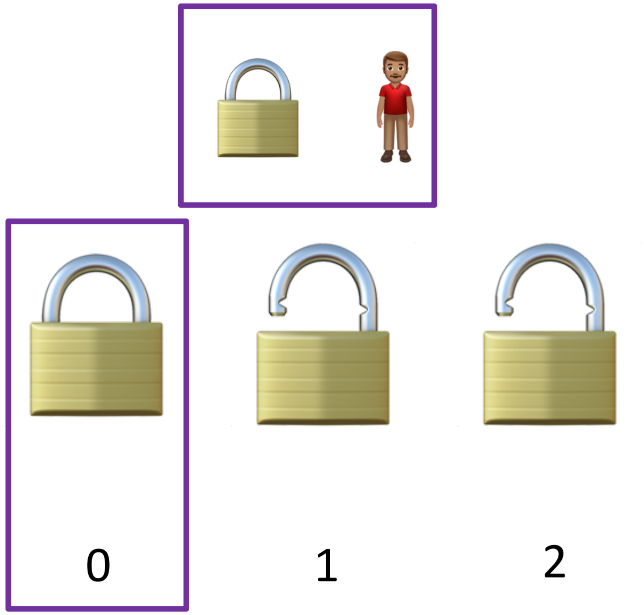
- Thread 1 Acquires Lock Guard First!
- Thread 0 and 2 cannot proceed…
- Thread 1 then acquires lock 1 and 2.
- Thread 1 then releases our Lock Guard

- While Thread 1 is eating, Thread 0 acquires the lock guard!
  - Thread 2 still doesn't progress.
- Thread 0 acquires lock 0!
- Thread 0 is stuck acquiring lock 1 as Thread 1 is eating!

- Thread 1 finishes eating! It releases Lock 2 first and then lock 1!

Poll Everywhere

pollev.com/cis5480



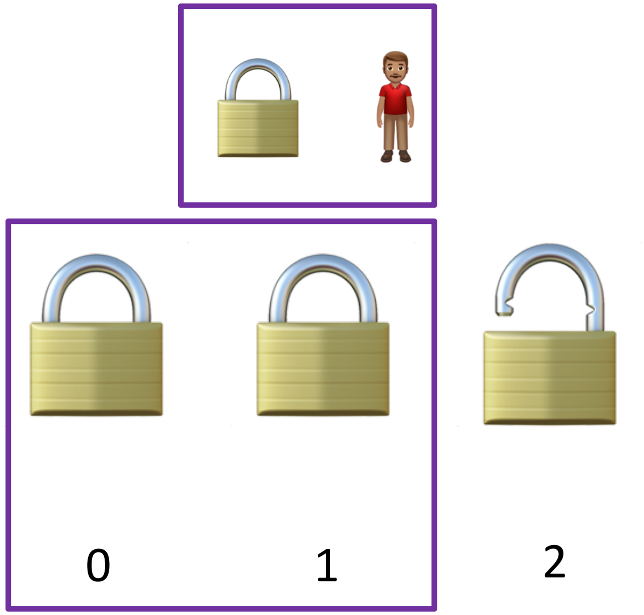
- Thread 1 Acquires Lock Guard First!
- Thread 0 and 2 cannot proceed…
- Thread 1 then acquires lock 1 and 2.
- Thread 1 then releases our Lock Guard

- While Thread 1 is eating, Thread 0 acquires the lock guard!
  - Thread 2 still doesn't progress.
- Thread 0 acquires lock 0!
- Thread 0 is stuck acquiring lock 1 as Thread 1 is eating!

- Thread 1 finishes eating! It releases Lock 2 first and then lock 1!
  - The moment it releases lock 1, Thread 0 acquires it!

Poll Everywhere

pollev.com/cis5480



- Thread 1 Acquires Lock Guard First!
- Thread 0 and 2 cannot proceed…
- Thread 1 then acquires lock 1 and 2.
- Thread 1 then releases our Lock Guard

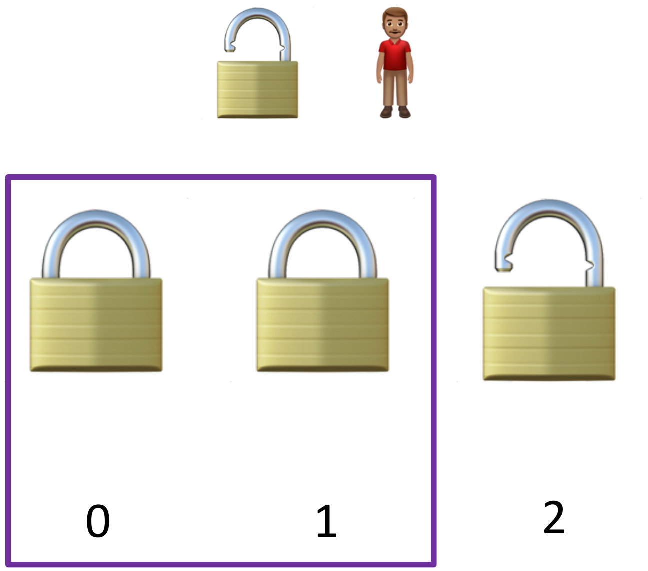
- While Thread 1 is eating, Thread 0 acquires the lock guard!
  - Thread 2 still doesn't progress.
- Thread 0 acquires lock 0!
- Thread 0 is stuck acquiring lock 1 as Thread 1 is eating!

- Thread 1 finishes eating! It releases Lock 2 first and then lock 1!
  - The moment it releases lock 1, Thread 0 acquires it!

- Now Thread 0 releases the lock guard as it has both chopsticks!

Poll Everywhere

pollev.com/cis5480



- Thread 1 Acquires Lock Guard First!
- Thread 0 and 2 cannot proceed…
- Thread 1 then acquires lock 1 and 2.
- Thread 1 then releases our Lock Guard

- While Thread 1 is eating, Thread 0 acquires the lock guard!
  - Thread 2 still doesn't progress.
- Thread 0 acquires lock 0!
- Thread 0 is stuck acquiring lock 1 as Thread 1 is eating!

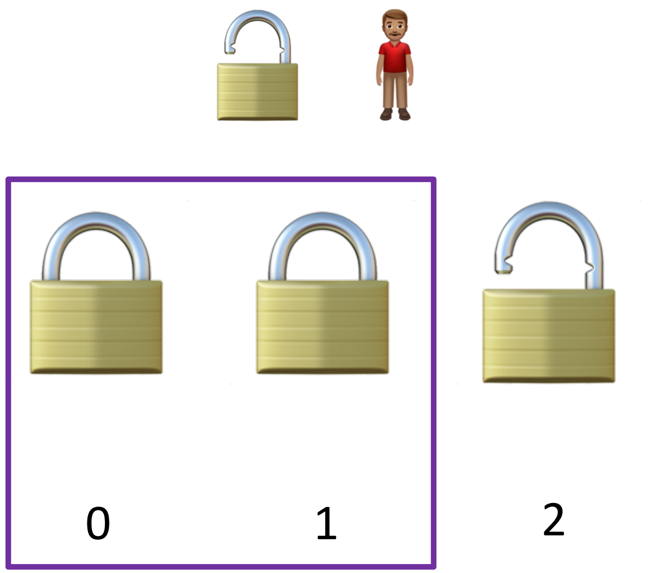
- Thread 1 finishes eating! It releases Lock 2 first and then lock 1!
  - The moment it releases lock 1, Thread 0 acquires it!

- Now Thread 0 releases the lock guard as it has both chopsticks!

- And now Thread 1 and 2 compete for who acquires the lock guard!
  - The cycle continues…

Poll Everywhere

pollev.com/cis5480



Starvation is still possible here as Thread 1 and Thread 0 could always acquired the lock guard before Thread 2.