Lecture 20
The Dining Philosophers Problem

ECE 422: Reliable and Secure Systems Design

UNIVERSITY OF ALBERTA

Instructor: An Ran Chen
Term: 2024 Winter

Schedule for today

- Key concepts from last classes
- The Dining Philosophers Problem
  - Race condition
  - Deadlocks
  - Starvation
- A solution to the Dining Philosophers Problem
  - Atomic locks for resource reservation
  - Critical section
- TODOs
  - Distributing the midterm

Diffie-Hellman algorithm

Diffie-Hellman is a key exchange algorithm used with symmetric encryption.

- It allows both parties to agree on an identical secret key
- Without having to share the actual key in the communication channel
- This is used for key exchange, and not encryption/decryption

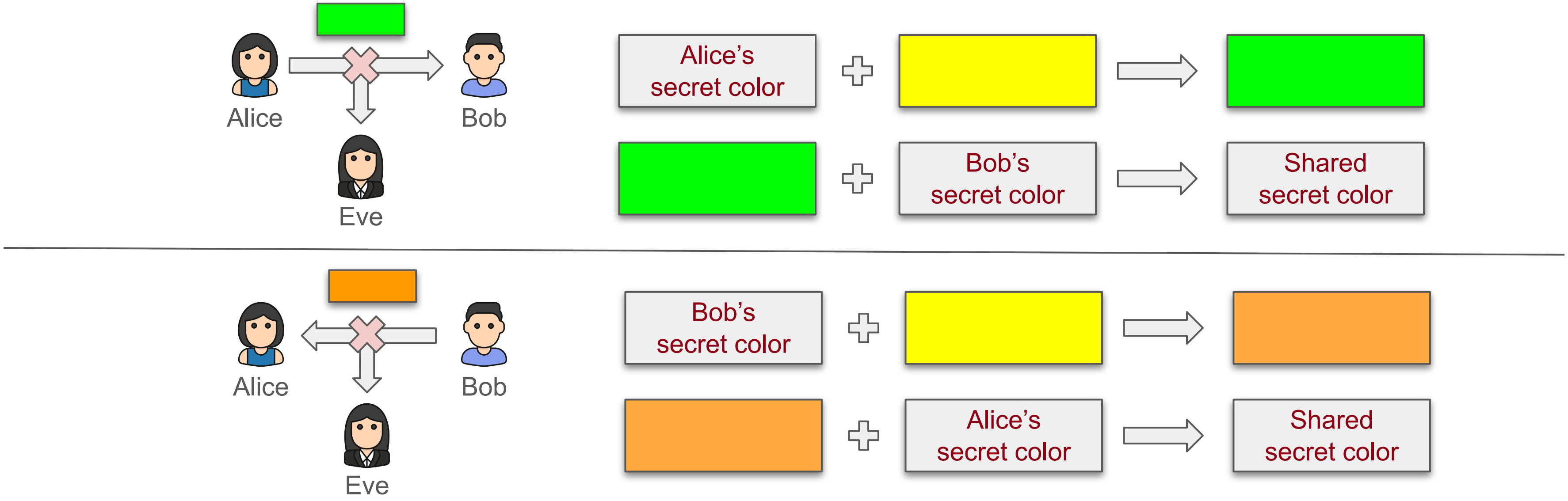
Overall idea behind DF key exchange algorithm:

- Neither parties choose the key explicitly
- Both parties contribute in calculating the secret key together
- The calculated secret key can then be used in symmetric encryption.

Analogous to finding a shared secret color

Can Eve come up with the shared secret color, brown?

- Eve has access to the starting color (yellow), and the mixed colors (green and orange), but not the secret colors (red and blue).



DF algorithm cheat sheet

DF algorithm as a 4-steps process:

Step 1: Alice and Bob agree on some public parameters

Step 2: Alice and Bob come up with a public integer

Step 3: Alice and Bob exchange their own public integer

Step 4: Alice and Bob compute the shared secret key

Public integers

- $A = g^a \mod(p)$
- $B = g^b \mod(p)$

Shared secret key

- $K = g^{ab} \mod(p) = A^b \mod(p)$
- $K = g^{ba} \mod(p) = B^a \mod(p)$

Schedule for today

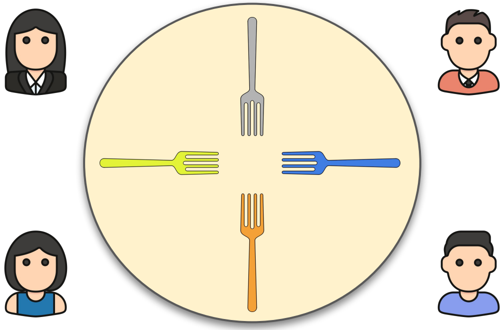
- Key concepts from last classes
- The Dining Philosophers Problem
  - Race condition
  - Deadlocks
  - Starvation
- A solution to the Dining Philosophers Problem
  - Atomic locks for resource reservation
  - Critical section
- TODOs
  - Distributing the midterm

The Dining Philosophers Problem

Dining Philosophers Problem is a classical synchronization problem in the operating system.

- Philosophers can either think or eat
- To eat, philosophers needs both their left and right forks
  - Two forks will only be available when the two nearest neighbors are thinking, not eating
  - If not available, they start thinking
  - After they are done eating, they will put down both forks
- To think, philosophers does nothing but thinking

Goal: Designing a solution (algorithm) so that no philosopher starves.



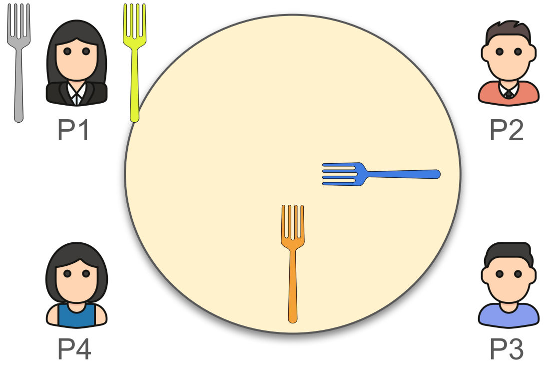
Basic protocol

```
while (true) {
    grab left fork;
    grab right fork;
    eat;
    release left fork;
    release right fork;
    think;
}
```

Example for P1:

- P1 grabs the left and right fork
- P1 eats
- P1 put the left and right fork down
- P1 thinks

What problems can happen here?



Challenges in the Dining Philosophers Problem

There are two challenges in solving the Dining Philosophers Problem:

- Race condition
- Deadlock

Race condition

A race condition may happen when philosophers attempt to grab the same fork at the same time:

- However, the correctness of the process depends on timing
- The action of one philosopher can intervene during another philosopher’s final decision (grabbing or not grabbing) that involves the shared fork
- Therefore, we want to avoid race condition

Example: Between checking the availability of the fork and grabbing the fork, another philosopher intervenes, making the check out of date and the action incorrect

Example of race condition

P1 decides to eat

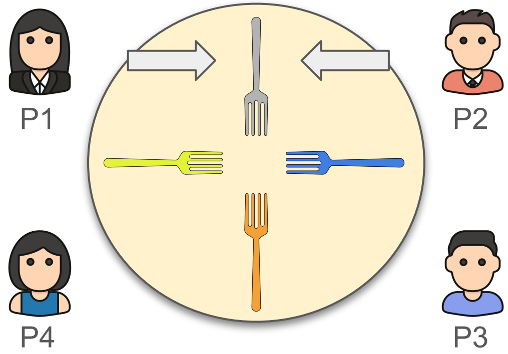
- P1 checks if the grey fork is available
- It is available, therefore P1 will try to grab the grey fork

At the same time, P2 decides to eat

- P2 checks if the grey fork is available
- Because the check happens before P1 actually grabs the grey fork, the fork will also be available to P2

Race condition detected!

- P2 cannot grab the fork as P1 will grab the fork before



Deadlock

Assume that the action of the philosophers are perfectly interleaved:

- Each philosopher grabs the fork on their right
- Then, they try to grab the fork on their left

Deadlock

Assume that the action of the philosophers are perfectly interleaved:

- Each philosopher grabs the fork on their right
- Then, they try to grab the fork on their left

However, as all philosophers grab their right fork at the same time, their left is gone.

- Each philosopher waits for the person on their left to release the fork, so they can start eating
- But it will never happen because of a circular chain where the person next to them is also waiting
- This is the problem of deadlock

Example of deadlock

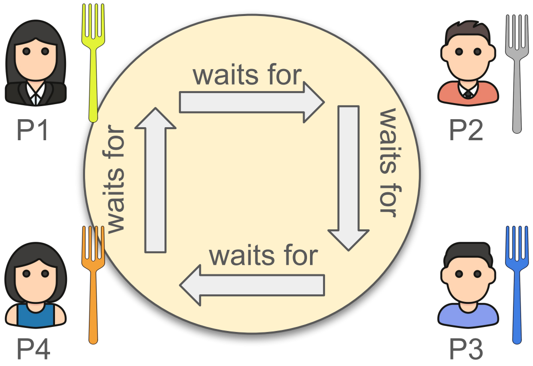
Step 1: Each philosopher grabs their right fork at the same time

- P1 grabs the yellow fork; P2 grabs the grey fork; …

Step 2: Each philosopher grabs their left fork at the same time

- P1 tries to grab the grey fork which is taken, so P1 starts to wait (think);
- P2 tries to grab the blue fork which is taken, so P2 starts to wait (think);
- …

All philosophers hold a fork but are unable to eat.
They will eventually all starve.



A process synchronization problem

The Dining Philosophers Problem illustrates synchronization issues in concurrent processes.

- Philosophers = Processes, programs or threads
- Forks = Shared resources (e.g., files, memory)
- Solution = Algorithm for resource allocation

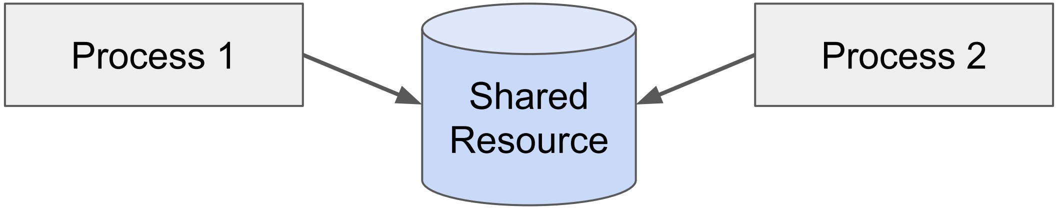
Lack of forks is an analogy to limited (shared) resources in computer programming

- Avoid race condition where some processes may never access the resource
- Avoid deadlocks where processes are mutually locking each other out

Race condition

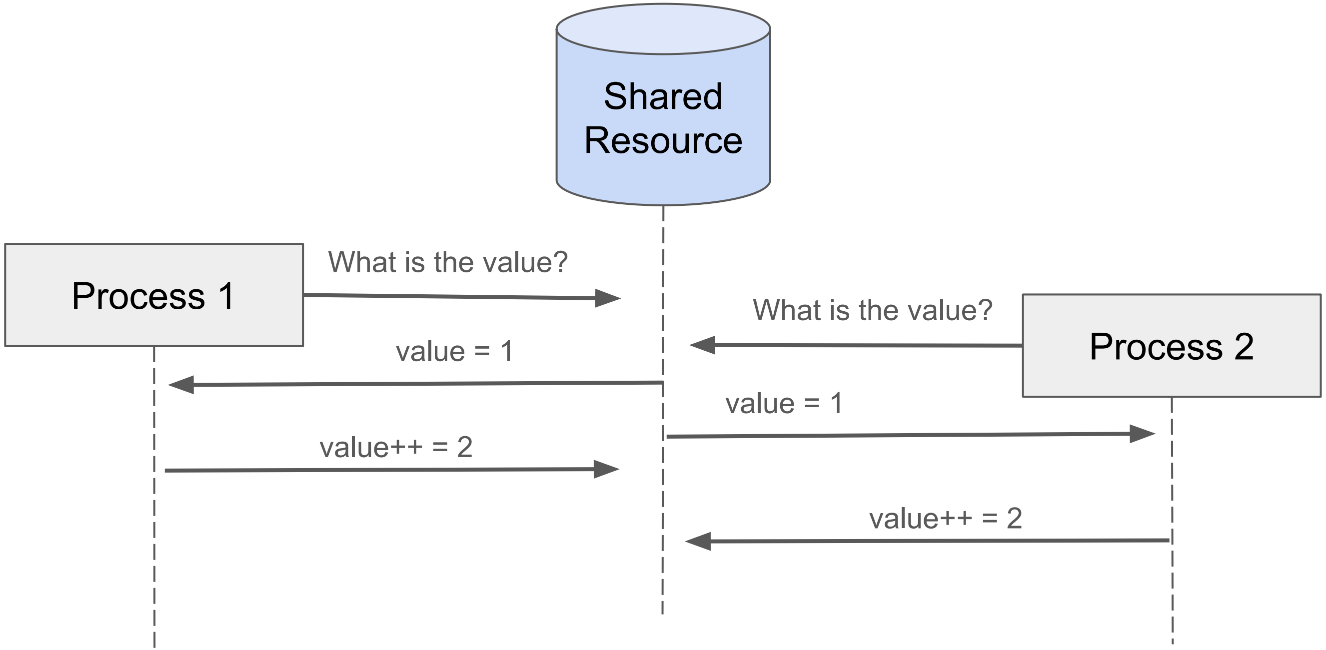
A race condition happens when two processes attempt to access the same resource at the same time. However, the timing or ordering of events affects a program's correctness.

- E.g., Two processes that attempt to increase a shared value by 1
- Instead of increasing the value twice, the value is only increased by 1 where the last modification is preserved



Cause of race condition

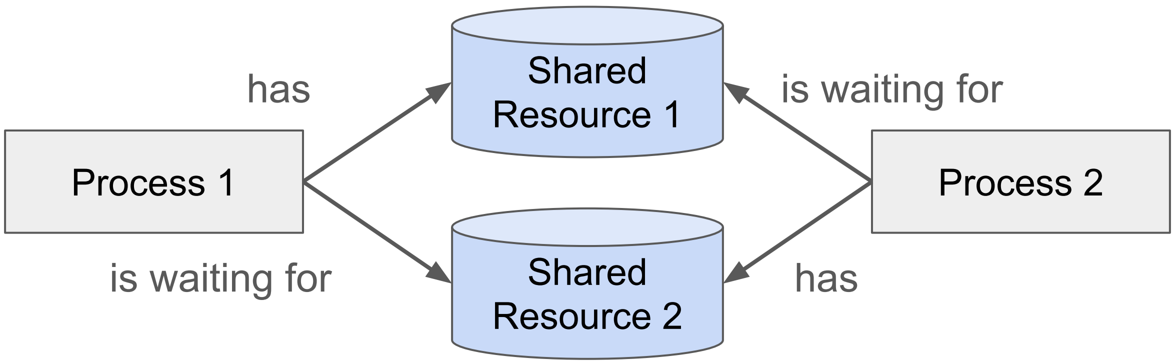
Between checking on a critical resource and acting on that check, another process intervenes, making that check out of date, and the action becomes incorrect.



Deadlock

Deadlock describes a situation in which two processes, sharing the same resources, are preventing each other from accessing the resources.

- Each process is holding the resource the other is waiting for
- But no one is releasing the resource until the other releases it



Necessary conditions for deadlock

Deadlock can only occur in systems where all following conditions are met:

- Mutual Exclusion
  - A resource cannot be used by more than one process at a time
- Hold and Wait
  - At least one process holds one resource while waiting for another
- No Preemption
  - No other process can force one process to release the resource
- Circular Wait
  - Two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds

Questions on deadlock and race condition

Multiple-choice question: A program containing a race condition will always/sometimes/never result in some incorrect behavior?

True/false question: Every deadlock is always starvation but every starvation is not a deadlock.

Questions on deadlock and race condition

Multiple-choice question: A program containing a race condition will always/sometimes/never result in some incorrect behavior?

Answer:

True/false question: Every deadlock is always starvation but every starvation is not a deadlock.

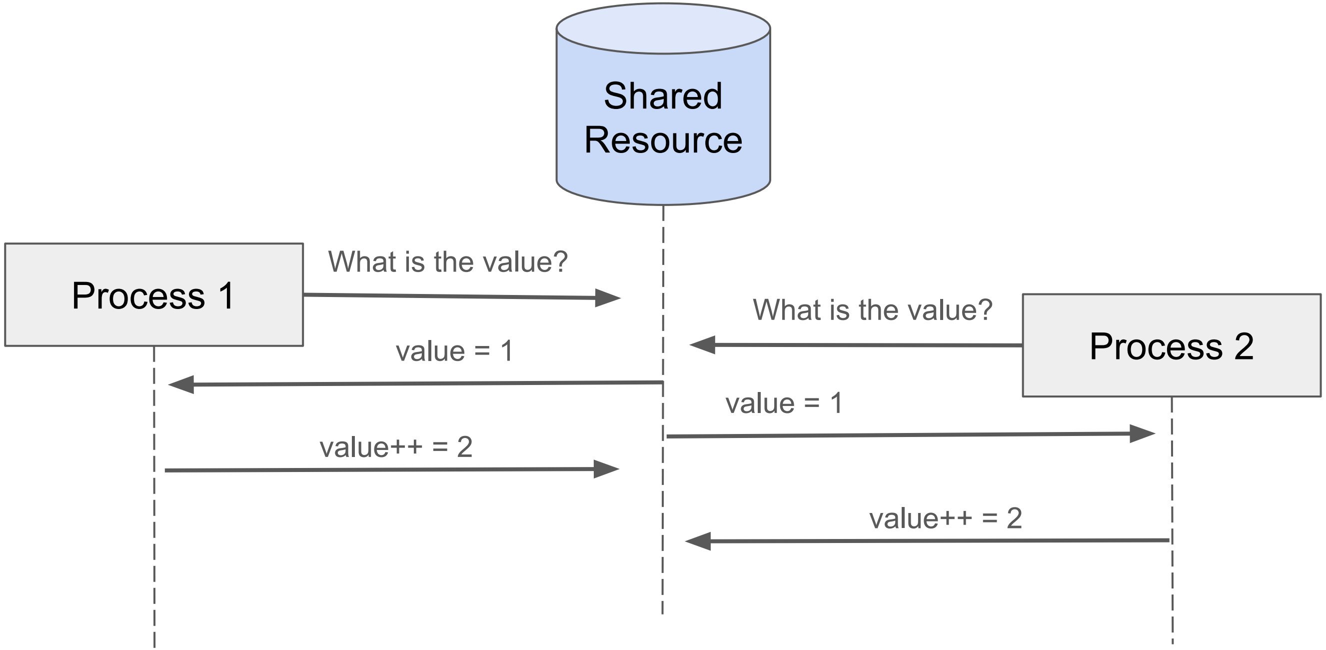
Answer:

Schedule for today

- Key concepts from last classes
- The Dining Philosophers Problem
  - Race condition
  - Deadlocks
  - Starvation
- A solution to the Dining Philosophers Problem
  - Atomic locks for resource reservation
  - Critical section
- TODOs
  - Distributing the midterm

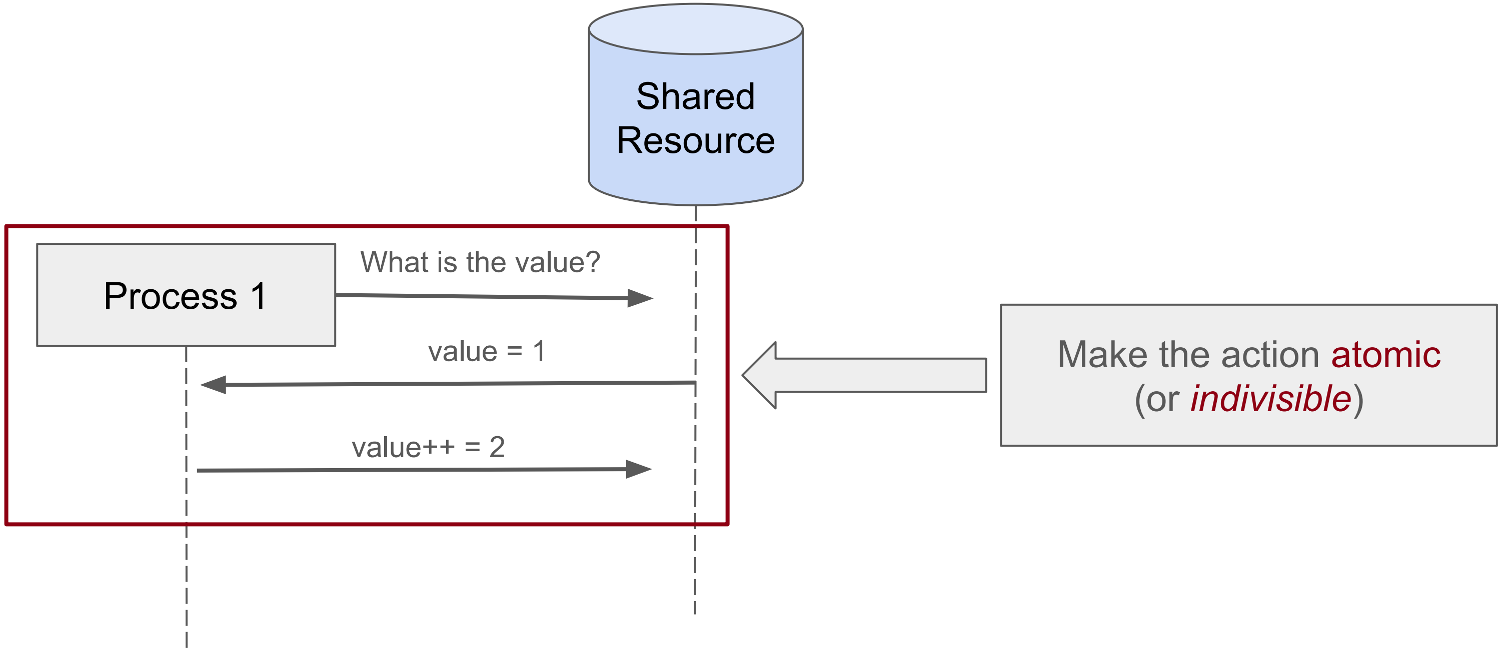
Cause of race condition

One process may intervene during another process' execution, making the check on the critical resource out of date. This is why the action becomes incorrect.



Avoiding race conditions

We must guarantee that no process can intervene during another process' manipulation that involves shared resources.



Deadlock Prevention

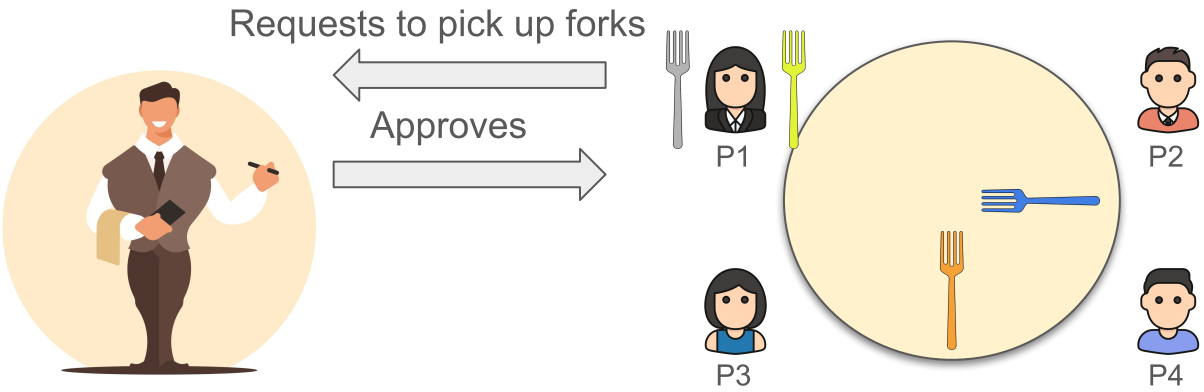
To prevent deadlock, we only need to ensure one of the conditions does not hold:

- Mutual Exclusion
  - Make the resource sharable
  - However, not all resources are sharable, e.g., editing files, analogous to forks
- <u>Hold and Wait</u>
  - Make processes request all resources at once and make them release all resources once done
- No Preemption
  - Make processes release all resources if the request cannot be proceed immediately
  - However, not all resources can be easily preempted, e.g., printing jobs
- Circular Wait
  - Impose an ordering on the resources and processes can only request them in order
  - However, waste of resources for such turn-based solution

Solution: “locking” forks

Introducing a waiter to monitor which forks are been used:

- Before eating, philosophers will ask for permission to pick up both forks
- If both forks are available, the waiter will give the permission to do so

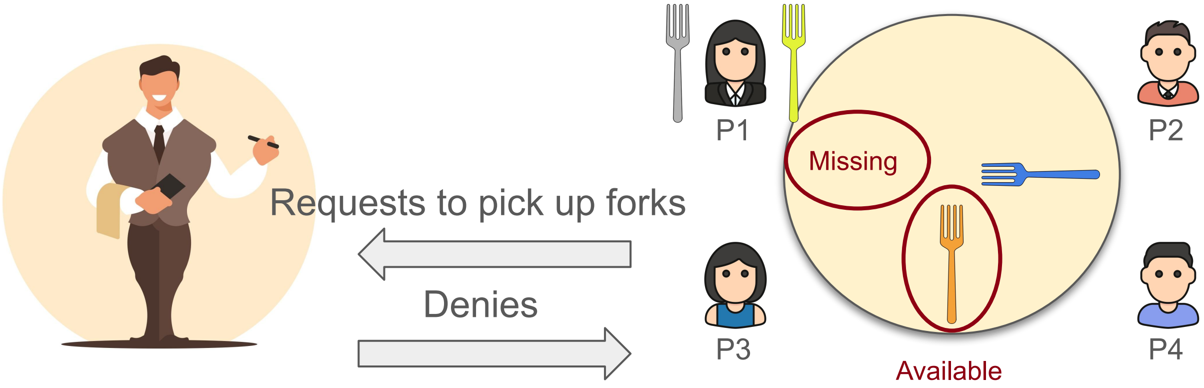


Intuition: Make philosophers request both forks at once, at the same time, each request must be atomic.

Example of “locking” forks

Given that P1 picked up the forks:

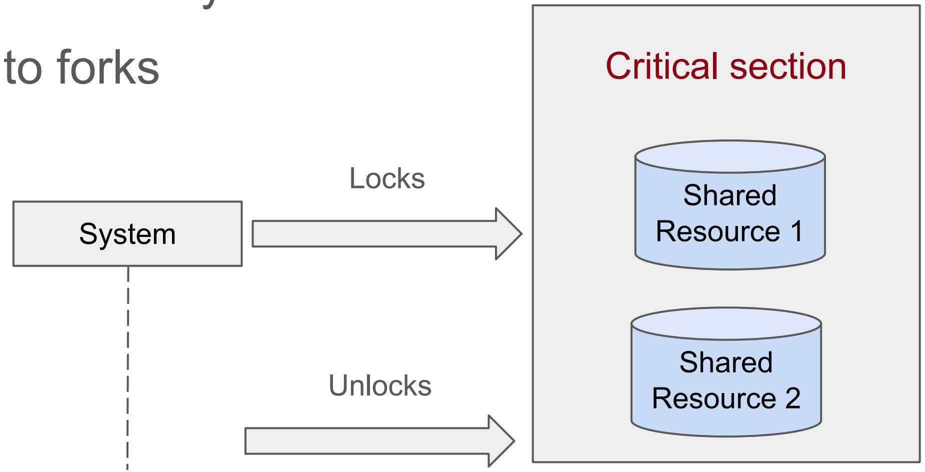
- If P3 (or P2) requests to pick up their forks, their request will be denied
- Because one of the forks is unavailable



Locks for resource reservation

Locks provide a mechanism to prevent multiple processes from accessing the resources in the critical section at the same time.

- Critical section is the part of a program which must be executed by only one process at a time to avoid race conditions
- Example of critical section: shared memory or files
- The critical section is analogous to forks



Locks for resource reservation

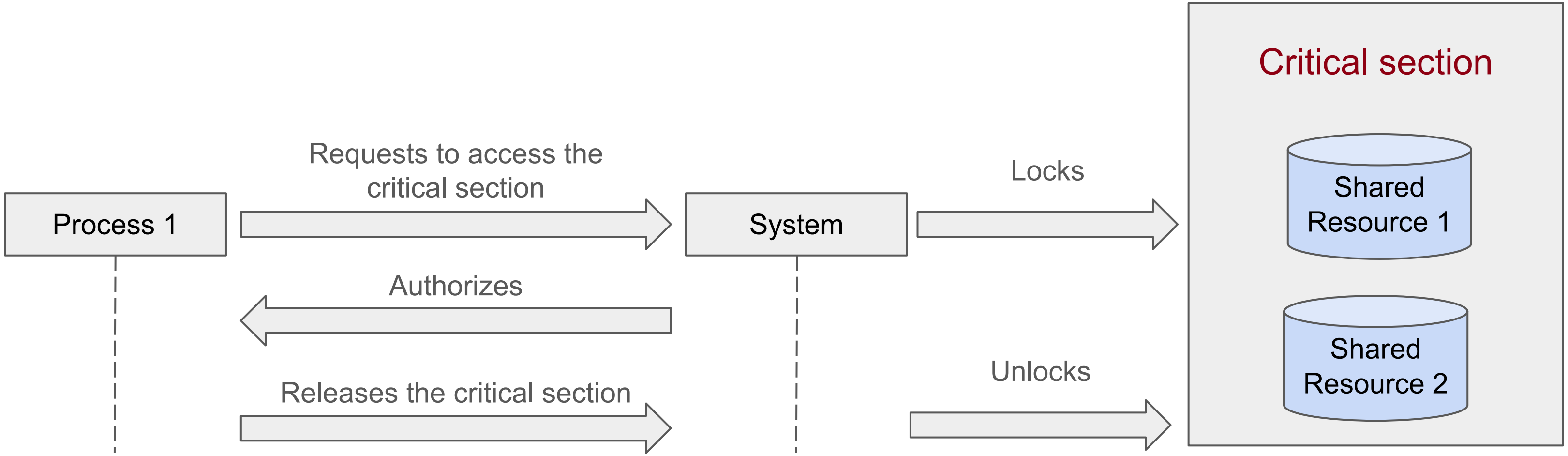
Generic implementation of locks:

- Lock before entering a critical section
- Unlock when leaving a critical section
- Wait and retry if the critical section is locked
- Lock is initially set free

Solution: atomic locking

Introducing atomic locks to the resources:

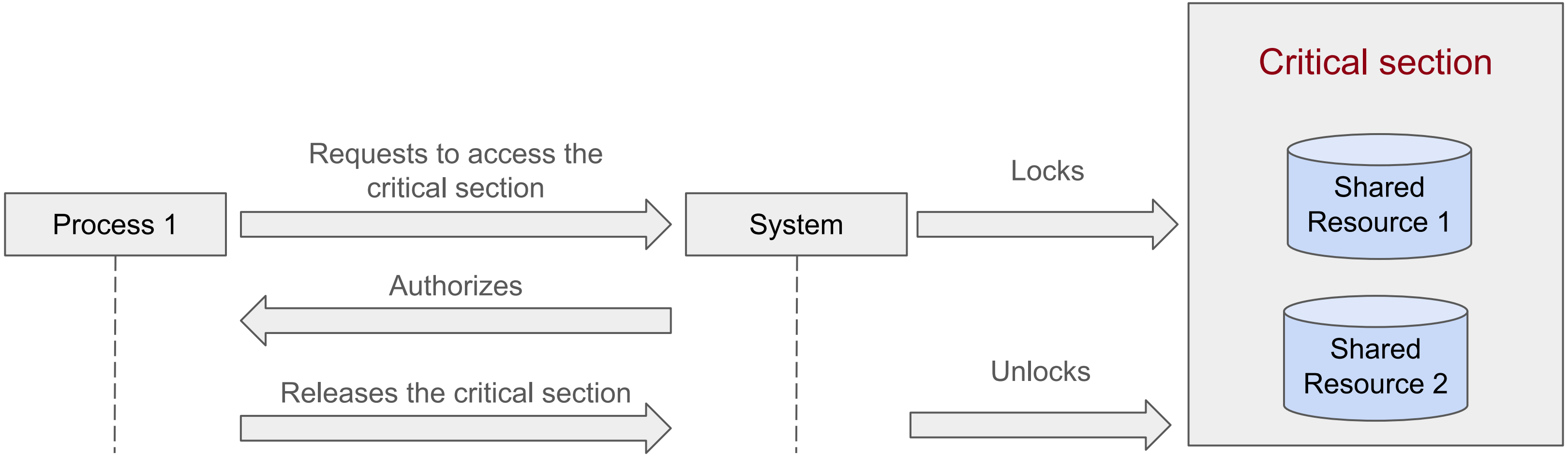
- Ensure that the resource is accessed by only one process at a time
- If the lock is free, the system will allow the process to access the resources and lock the critical section



Solution: atomic locking

Introducing atomic locks to the resources:

The atomic property guarantees that the resource remains locked until Process 1 finishes all its execution.



Other solutions?

- Resource hierarchy solution
- Arbitrator solution
- Chandy/Misra solution
- … and more

Question: Why can’t the philosophers just get more forks?

Answer: But more resources (forks) are expensive. The Dining Philosophers is used to illustrate a synchronization problem using the same resources.

One more question on deadlock

Multiple-choice question: A system that meets the four deadlock conditions will always/sometimes/never result in deadlock?

Next class: introducing the resource allocation graph to answer this question.

TODOs

- Midterm distribution
  - Resources = Classroom tables for distributing the midterms
  - Processes = Students picking up their midterms
  - To have atomic actions = Each student searches for his/her midterm without interference
  - To avoid race condition = Multiple students cannot access the tables at the same time
  - To avoid starvation = Avoid having 73 students waiting on 1 student

Solution? More resources, more instances

- Version A, Last name starting with A to M (Resource 1, instance 1)
- Version A, Last name starting with N to Z (Resource 1, instance 2)
- Version B, Last name starting with A to M (Resource 2, instance 1)
- Version B, Last name starting with N to Z (Resource 2, instance 2)