# CME 213

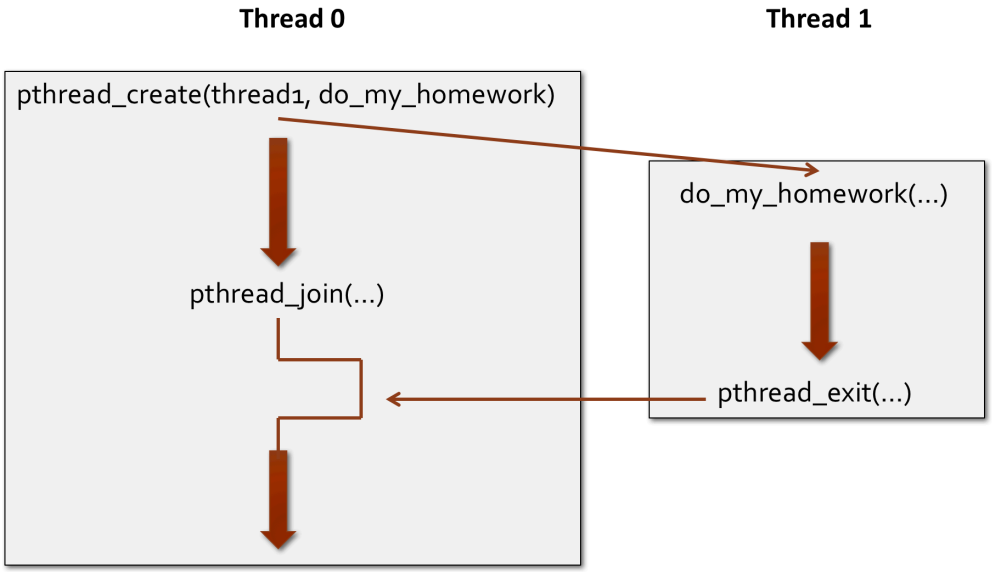
## Spring 2017

Eric Darve

Stanford University

# PTHREADS

- Low-level API
- Implemented in C
- `pthread_create`
- `pthread_exit`
- `pthread_join`

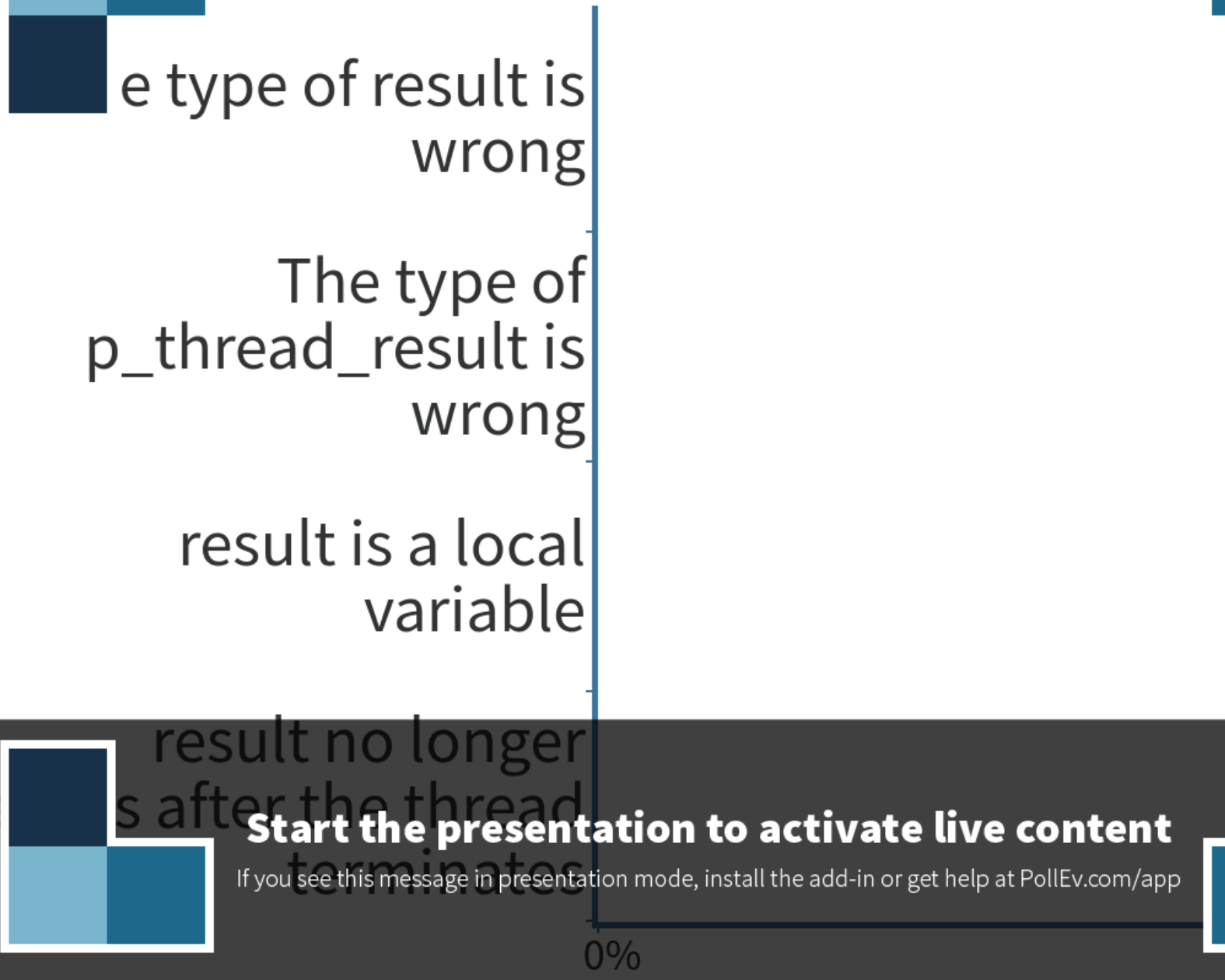


See

**hello_pthread_bug_1.c**

**hello_pthread_bug_2.c**

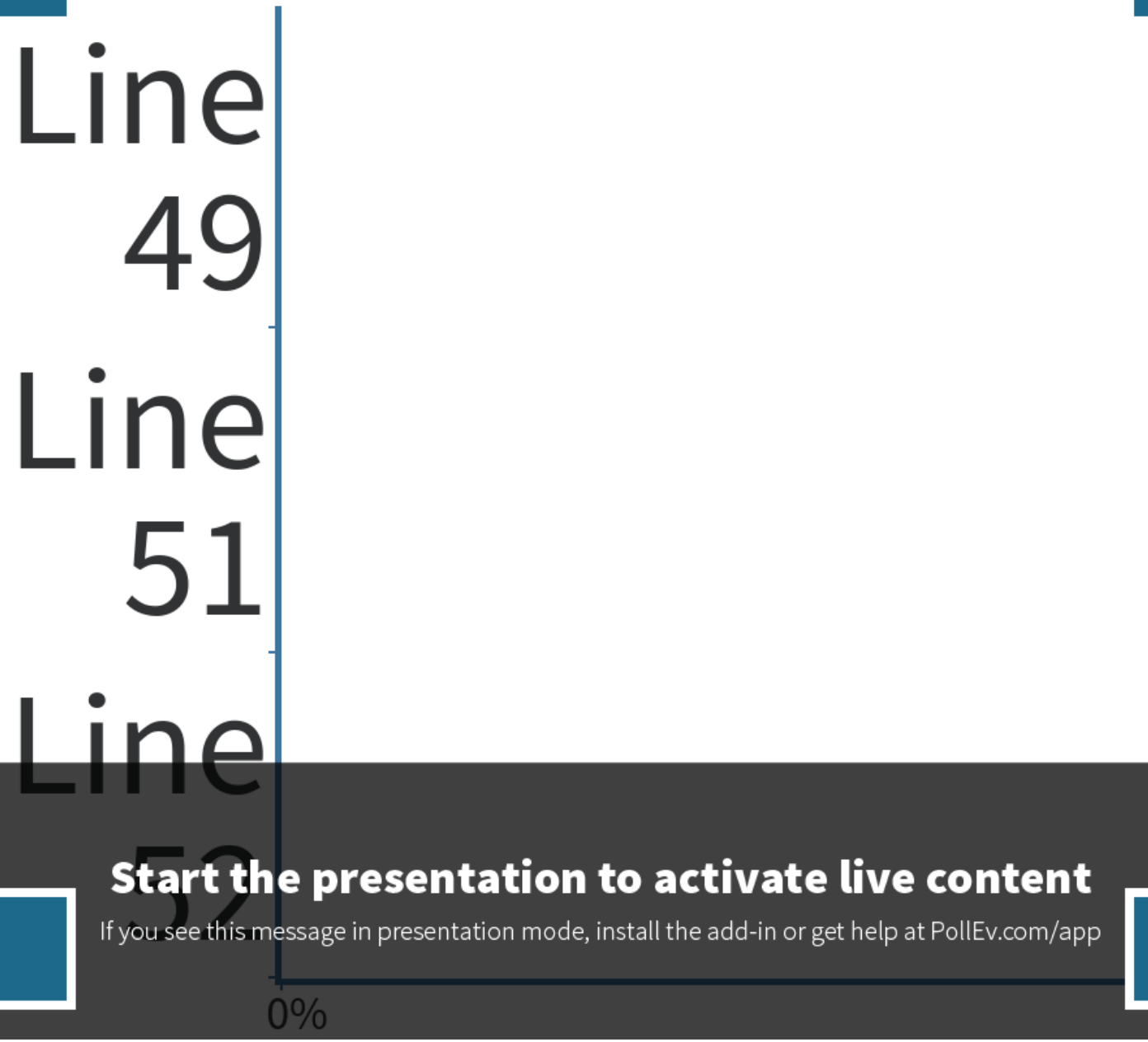
# What's the error in bug_1.c?




**Start the presentation to activate live content**

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

# ine would you delete in order for bug_2.c to




**Start the presentation to activate live content**

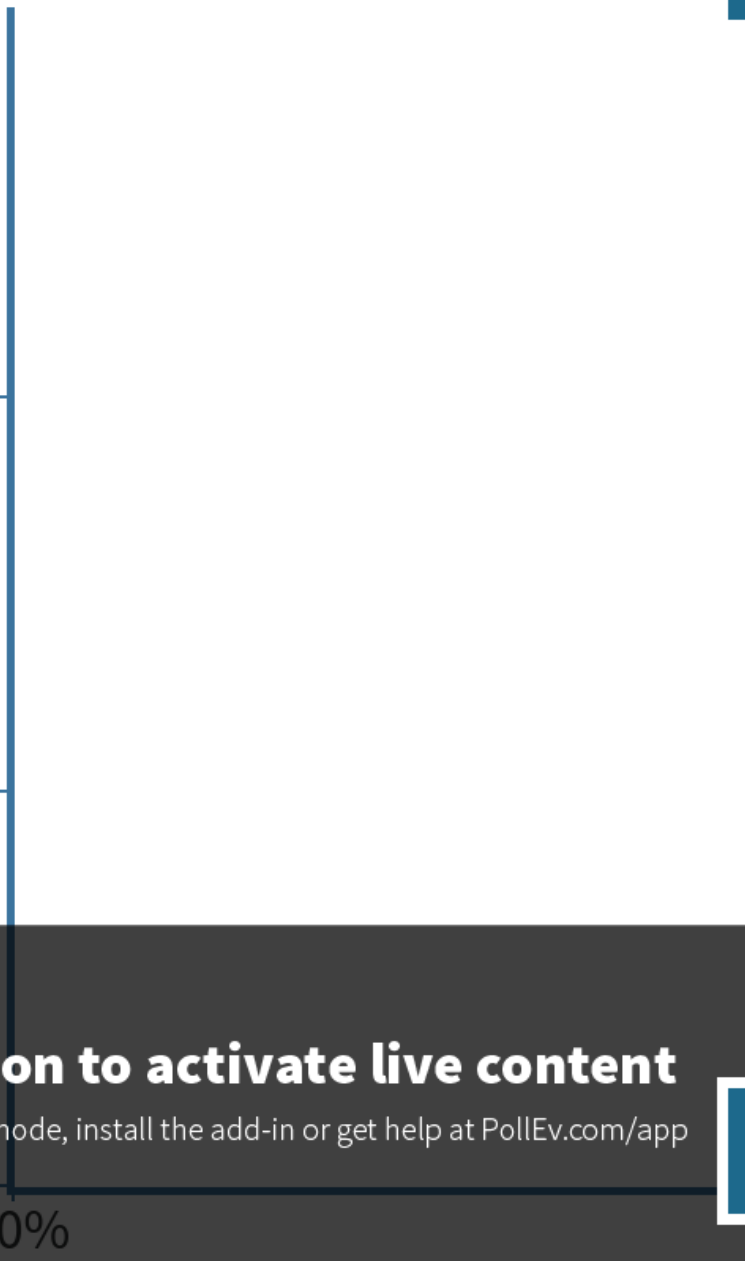
If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

# What happens if you delete line 49 in bug_2.c

The result is correct

The result is wrong

The result is undetermined

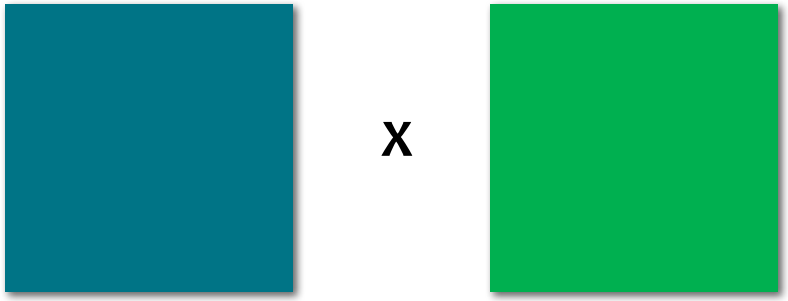



**Start the presentation to activate live content**

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

## EXAMPLE: MATRIX MATRIX PRODUCT

- C = A * B
  where A and B are square matrices.
- See `matrix_prod.c`

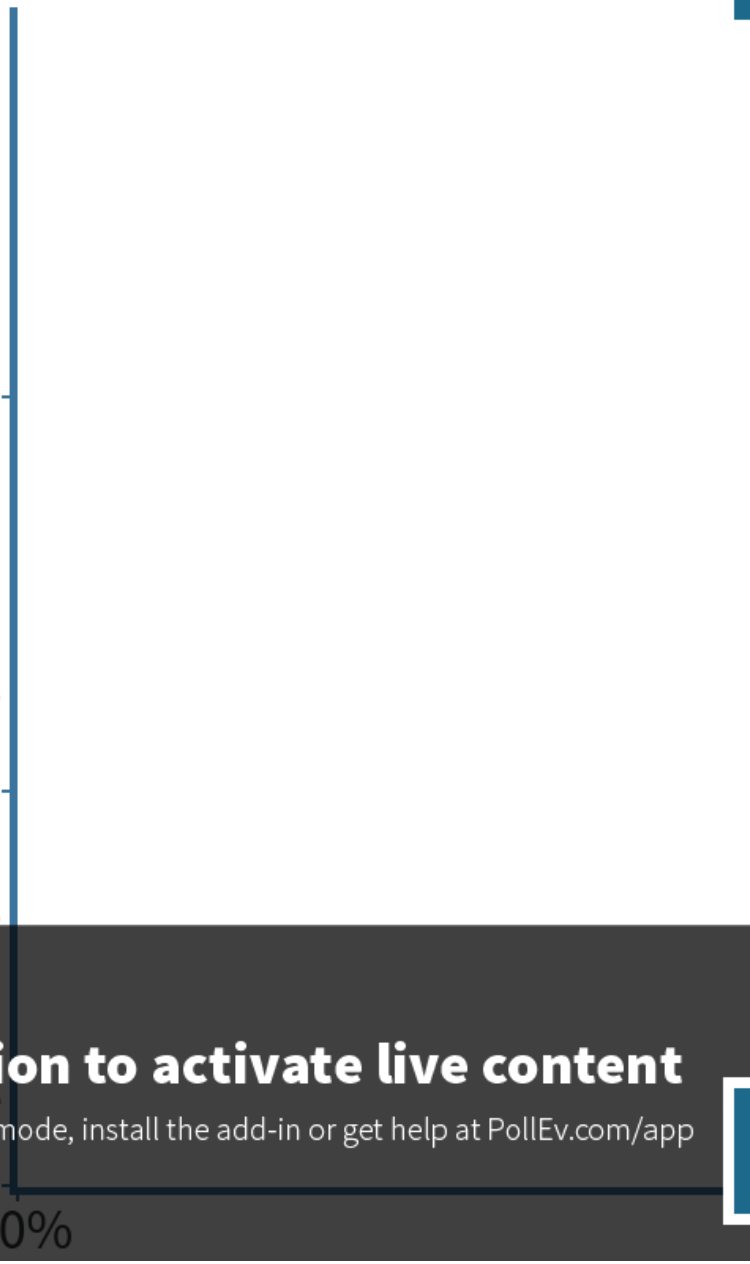


# Why are we casting work to void* in line 179?

So MatrixMult can be a general function

Because this is the type that pthread_create expects

Because this is required by the signature of MatrixMult




**Start the presentation to activate live content**

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

# THREAD COORDINATION

# THE RISKS OF MULTI-THREADED PROGRAMMING

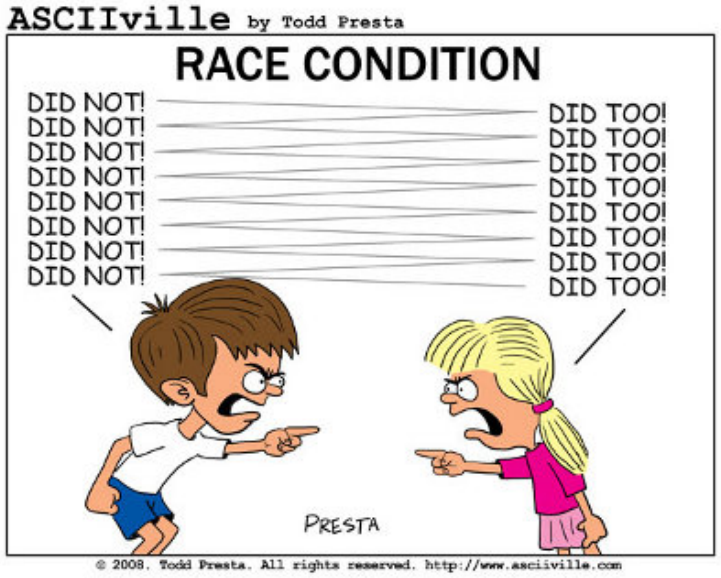
- Let us assume that a well-known bank company has asked you to implement a multi-threaded code to perform bank transactions.
- You start with the modest goal of allowing deposits.
- Clients deposit money and the amount gets credited to their accounts.
- As a result of having multiple threads running concurrently the following can happen:

# A PARALLEL BANK DEPOSIT

| Thread 0 | Thread 1 | Balance |
|---|---|---|
| Client requests a deposit | Client requests a deposit | $1000 |
| Check current balance = $1000 | | |
| | Check current balance = $1000 | |
| Ask for deposit amount = $100 | Ask for deposit amount = $300 | |
| | Compute new balance = $1300 | |
| Compute new balance = $1100 | Write new balance to account | $1300 |
| Write new balance to account | | $1100 |

# RACE CONDITION

- Although the correct balance should be $1,400, it is $1,100.
- The problem is that many operations "take time" and can be interrupted by other threads attempting to modify the same data.
- This is called a race condition: the final result depends on the precise order in which the instructions are executed.



## RACE CONDITION

- This issue is addressed using mutexes (mutex): mutual exclusion.
- They ensure that certain common pieces of data are accessed and modified by a single thread.
- This problem typically occurs when you have a sequence like:
  READ/WRITE, or
  WRITE/READ
  performed by different threads.

Thread 0 wants to add new to-do item.


Thread 0 closes lock. No other thread can open the lock.

Thread 0 is done with the to-do list. It opens the lock.

Thread 1 wants to open the lock. It has to wait.

Thread 1 can close the lock and access the to-do list.

## MUTEX

- **A mutex can only be in two states: locked or unlocked.**
- **Once a thread locks a mutex:**
  - **Other threads attempting to lock the same mutex are blocked**
  - **Only the thread that initially locked the mutex has the ability to unlock it.**
- **This allows to protect regions of code.**

## TYPICAL USAGE

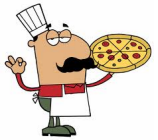
**Mutex use:**

- **Create and initialize a mutex variable.**
- **Threads attempt to lock the mutex.**
- **Only one succeeds, and that thread owns the mutex.**
- **The owner thread performs some set of actions.**
- **The owner unlocks the mutex.**

- **Another thread acquires the mutex and repeats the process.**

# Pizza restaurant

**Pizza cook**



Receives all the orders.
Prepares all the pizzas.

**Ask delivery team to deliver pizzas to customers**

**Pizza delivery team**

- Checks the addresses for customers
- Deliver pizza

**Go back; check if there are orders left.**

## Pizza restaurant code

See

`mutex_demo.c`

## SUMMARY OF KEY FUNCTIONS

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
int pthread_mutex_init(pthread_mutex t *mutex,
                       const pthread_mutexattr_t *attr)
```

Initialization of mutex; choose `NULL` for `attr`.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

Destruction of mutex.

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

Locks a mutex; blocks if another thread has locked this mutex and owns it.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Unlocks mutex; after unlocking, other threads get a chance to lock the mutex.

# uld I move line 76 (mutex_lock) to before line
# "global_task_list = global_task_list->next") ?

Yes because the thread is not modifying global_task_list until line 86

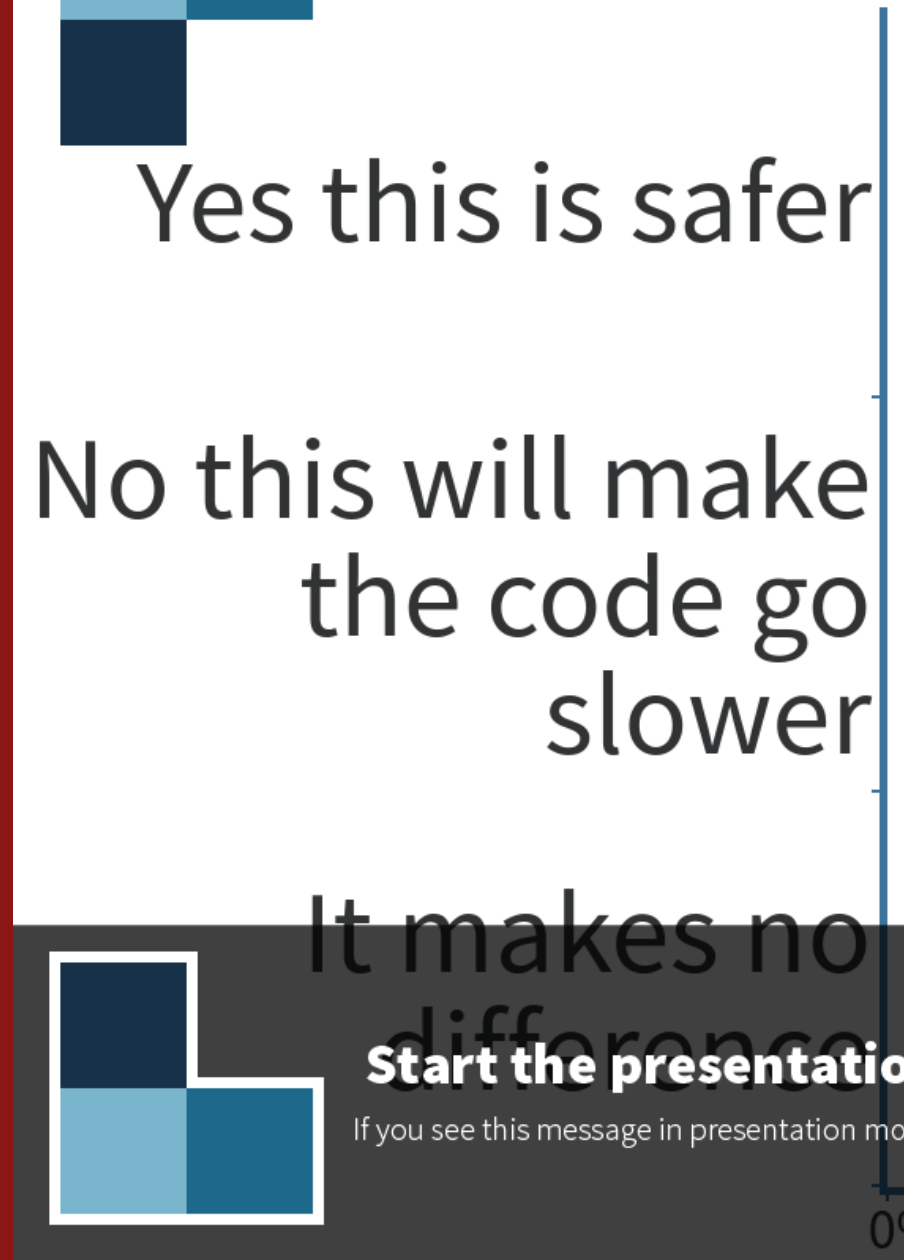
No because of line 79

It depends on what threads are doing


**Start the presentation to activate live content**

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

# ove line 89 (mutex_unlock) to after line 95 (D




**Start the presentation to activate live content**

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

# Deadlock

**Another strange case (but common!) of parallel computing.**

**Let's assume:**

Thread 0 Locks *mutex0*

Thread 1 Locks *mutex1*

Thread 0 Does some work

Thread 1 Does some work

Thread 0 Locks *mutex1*

Thread 1 Locks *mutex0*

The code will never reach this point → Thread 0 Work requiring lock on both mutexes

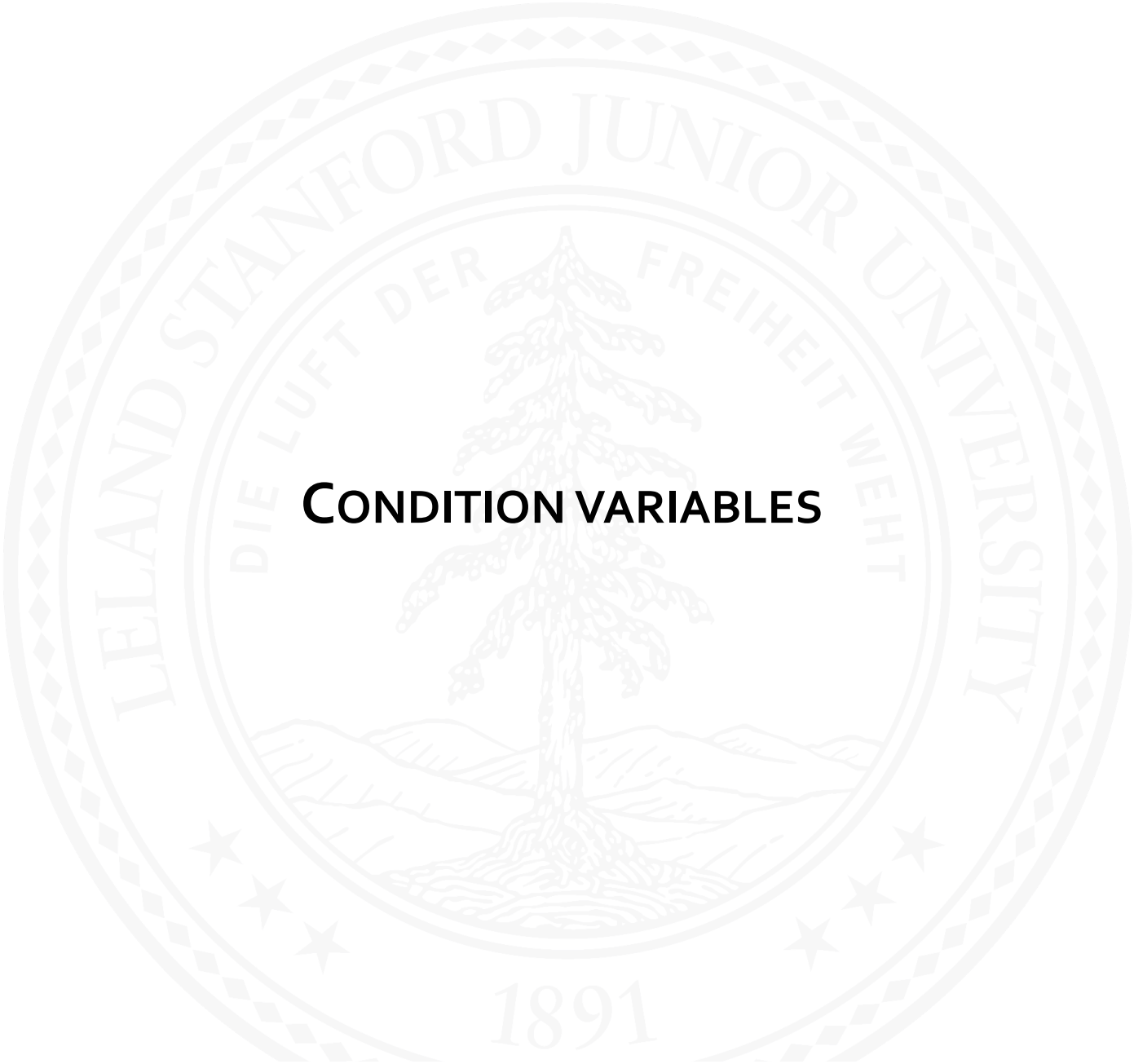
Thread 1 Work requiring lock on both mutexes

Thread 0 Unlocks all mutexes

Thread 1 Unlocks all mutexes


**Solution: threads always lock mutexes in the same order.**

# Condition variables

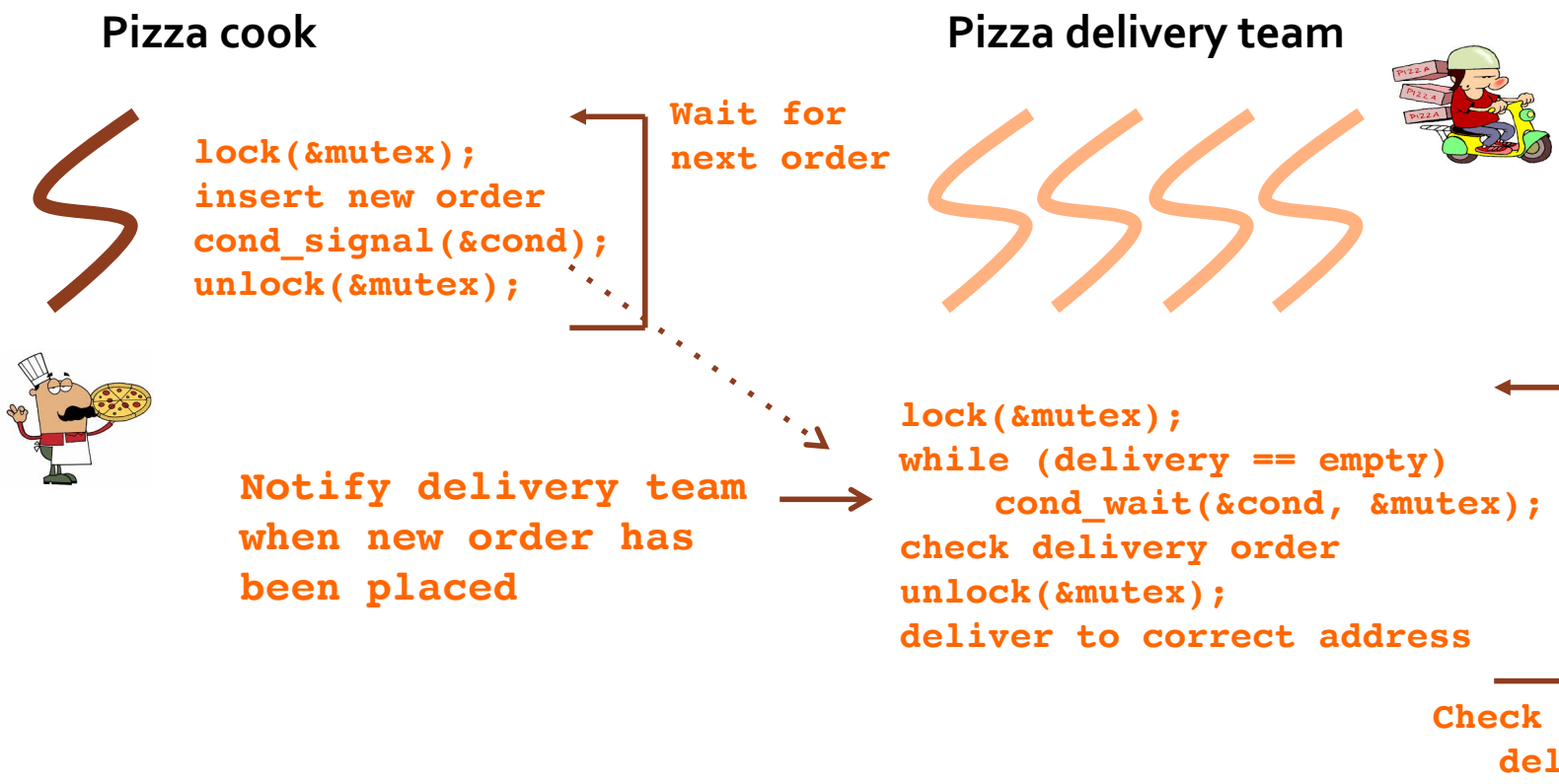


# ARE WE THERE YET?


- There is a common situation in computing: you have a pool of threads that are ready to perform tasks, but there is nothing to do yet. So they have to wait until some work becomes available.
- Instead of constantly checking the to-do list, they can simply wait and be awaken when a certain condition is met.

# WAITING ON A CONDITION

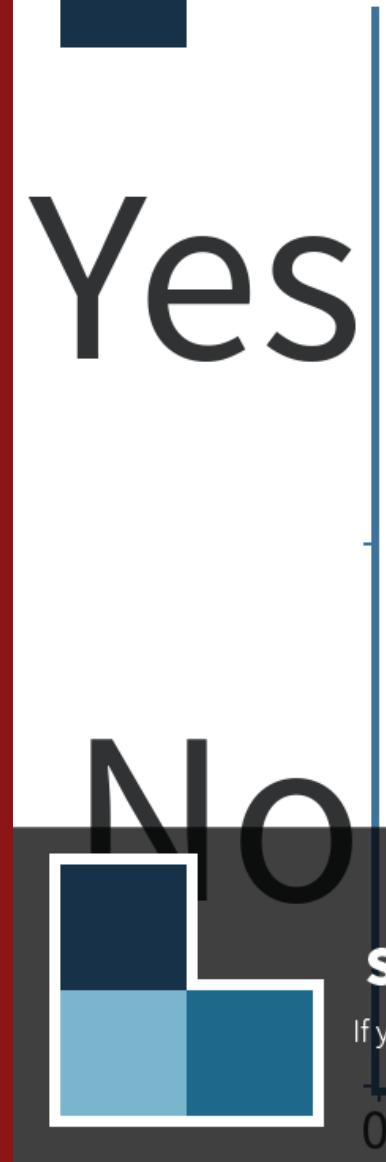
- Application: pizza delivery boys waiting on pizza cook.
- This is the producer-consumer model.



## CONDITION VARIABLE EXAMPLE

**`cond_var.c`**

# s the mutex unlocked when the thread enter pthread_cond_wait?




**Start the presentation to activate live content**

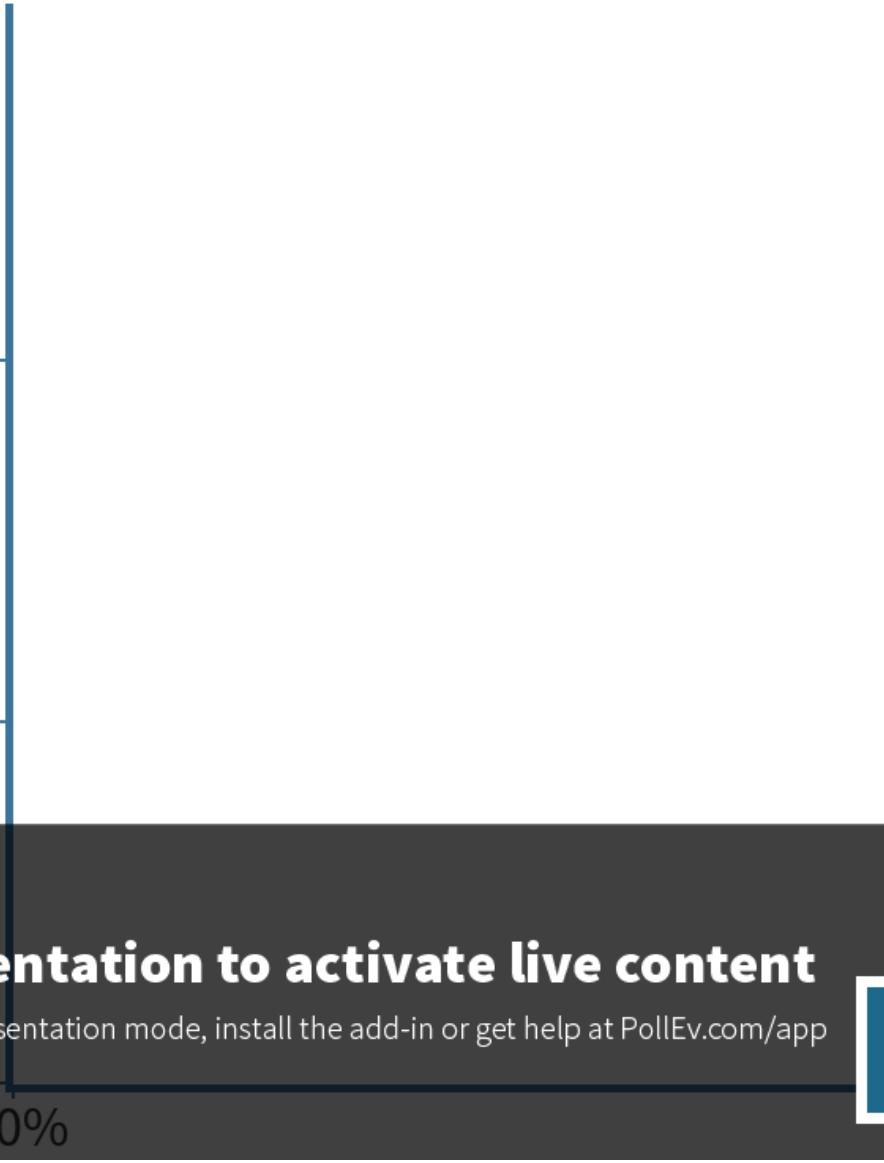
If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

# ppens to the mutex after a thread receives a c signal?

It is unlocked

It is locked

Nothing




**Start the presentation to activate live content**

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

## CONDITION VARIABLE

- Requires a condition variable and a mutex.
- The mutex is used to protect the access to the condition variable.

```
int pthread_cond_wait(pthread_cond_t *cond,
    pthread_mutex_t *mutex)
```

**Recommended scenario**

```
pthread_mutex_lock(&mutex);
while (!condition_ready())
    pthread_cond_wait(&cond, &mutex);
access_modify_shared_data();

pthread_mutex_unlock(&mutex);
```

When `pthread_cond_wait` is called:

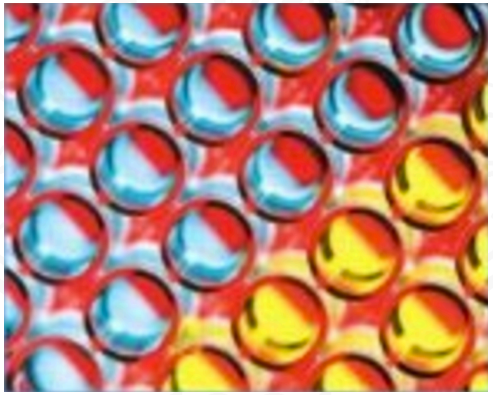
- The thread releases the mutex `mutex`.
- The thread waits until a signal is sent.
- Upon receiving a signal, the thread locks `mutex` and proceeds.

## CONDITION SIGNAL

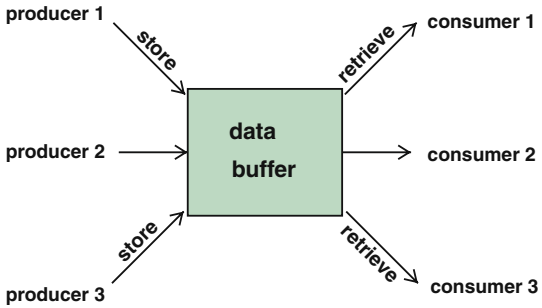
```
int pthread_cond_signal(pthread_cond_t
*cond)
```

- Wakes up a single thread waiting on the variable **`cond`**.
- Typically a mutex is used around **`pthread_cond_signal`** to protect the evaluation of the condition.

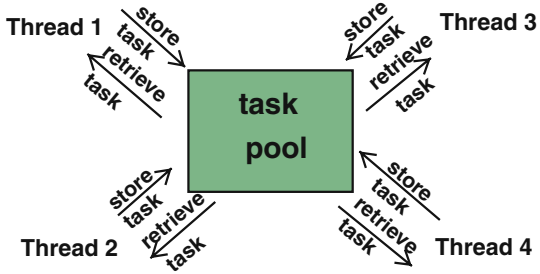
# Parallel patterns and other features of Pthreads



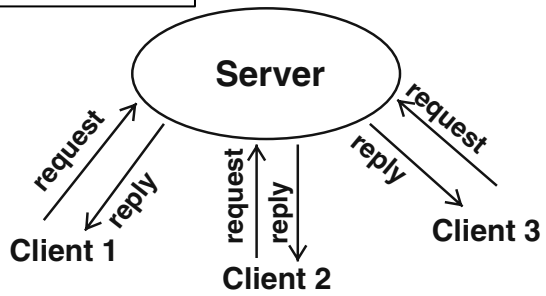
# EXAMPLE OF USEFUL PARALLEL PROGRAMMING PATTERNS




Producer consumer

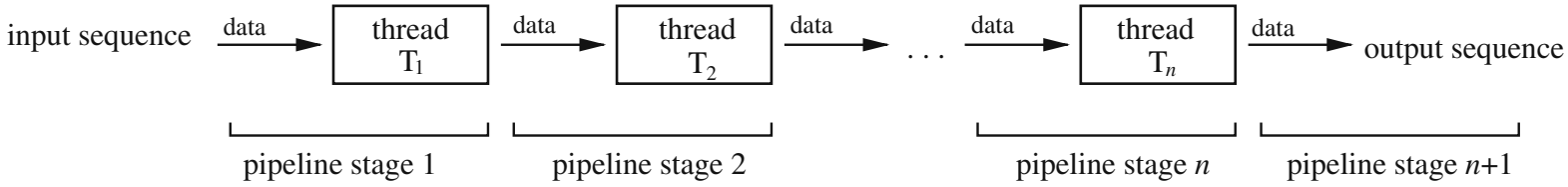



Task pool

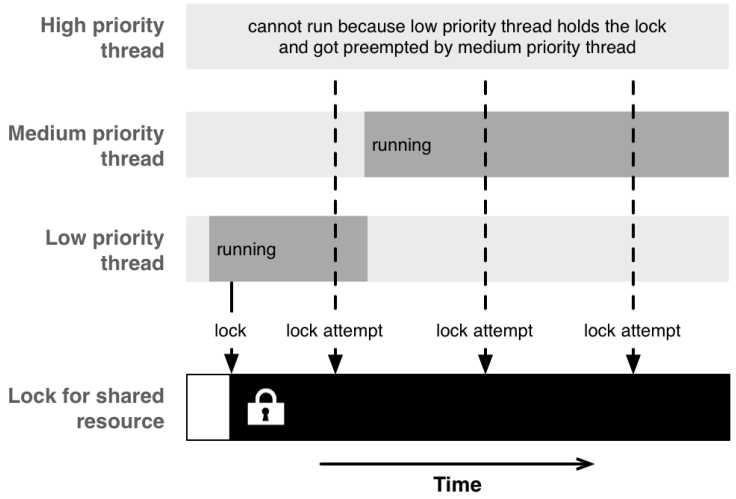



Client server

Pipeline



# Other features




- **Thread scheduling**
  - Implementations will differ on how threads are scheduled to run. In most cases, the default mechanism is adequate.
  - The Pthreads API provides routines to explicitly set thread scheduling policies and priorities which may override the default mechanisms.
- **Thread-specific data: keys**
  - To preserve stack data between calls, you can pass it as an argument from one routine to the next, or else store the data in a global variable associated with a thread.
  - Pthreads provides another, possibly more convenient and versatile, way of accomplishing this, through keys.
- **Priority inversion problems; priority ceiling and priority inheritance**
- **Thread cancellation**
- **Barriers; not always available**
- **Thread safety**