CME 213

SPRING 2019

Eric Darve

So far in CME213

- Multi-threaded programs: based on creating threads that run user-provided functions
- C++ threads:

```
thread t1(f1);
```

- Pthreads:

```
pthread_create(&thread[t], NULL, PrintHello, (void *)t);
```

Pthreads basics

- Include the header file:

```
include <pthread.h>
```

- Compile using:

```
gcc -o hello_pthread hello_pthread.c -lpthread
```

```
hello_pthread.c
```

```
for (t = 0; t < n_thread; t++)
{
    printf("In main: creating thread %ld\n", t);
    pthread_create(&thread[t], NULL, PrintHello, (void *)t);
}

for (t = 0; t < n_thread; t++)
{
    pthread_join(thread[t], (void **)(&thread_result) /*NULL is common*/);
    printf("Thread #%ld just finished; its value is %ld\n", t,
           thread_result);
    assert(ComplexCalculation(t) == thread_result); /* Testing output */
}
```

Reference: thread creation

```
int pthread_create(
   pthread_t *thread,
   const pthread_attr_t *attr,
   void *(*routine)(void*),
   void *arg)
```

- `thread` thread identifier
- `routine` function that will be executed by the thread
- `arg` pointer to the argument value with which the thread function `routine()` will be executed
- `attr` use `NULL` for the time being

Reference: thread termination

A thread terminates when:

1. Thread reaches the end of its thread function, i.e., returns.
2. Thread calls

```
void pthread_exit(void *valuep)
```

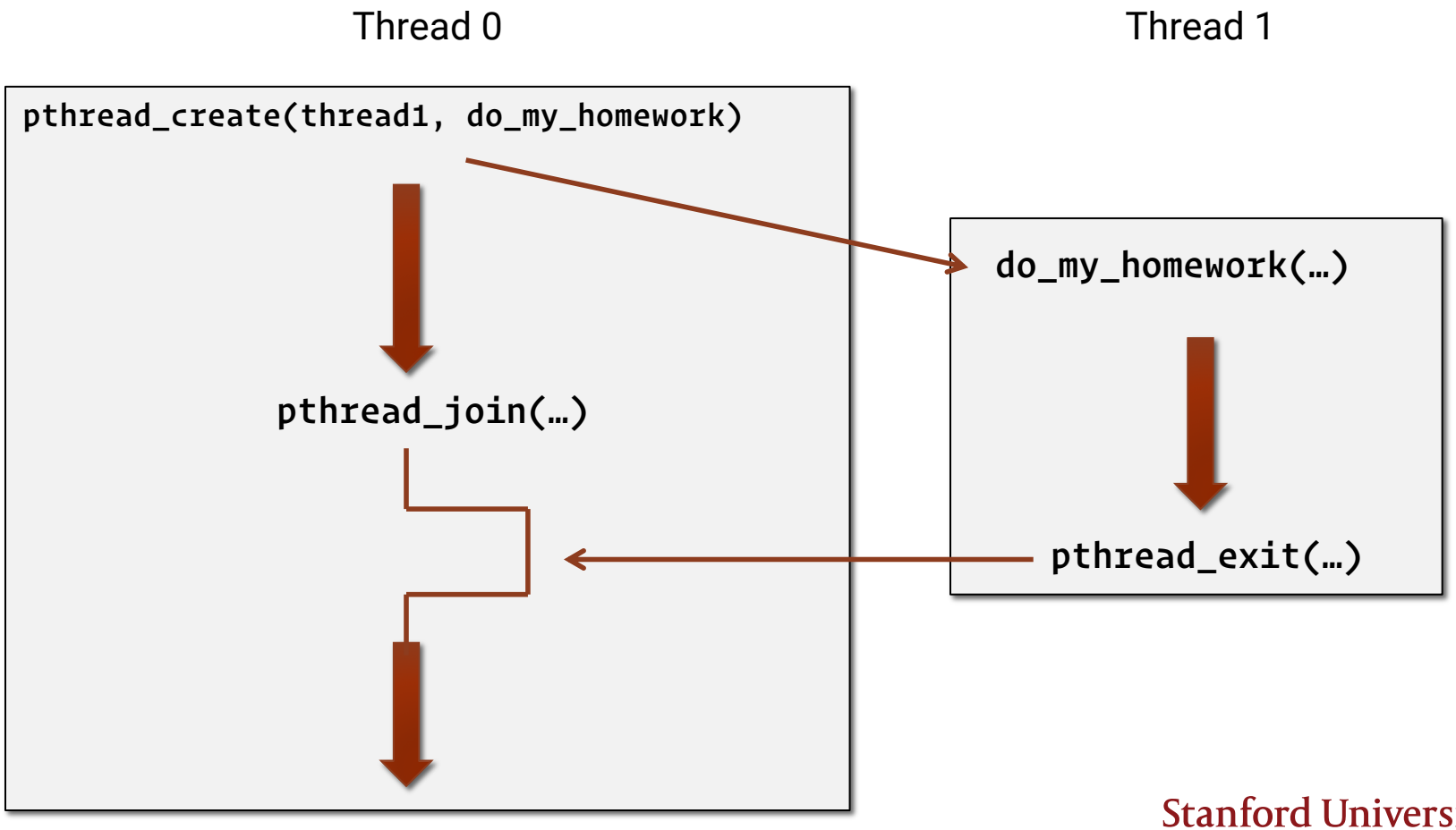
Note:

- Upon termination, a thread releases its runtime stack.
- Therefore, the pointer should point to: 1) a global variable, or 2) a dynamically allocated variable.

```c
void *PrintHello(void *threadid)
{
    long tid = (long)threadid;
    long result;
    result = ComplexCalculation(tid); /* Simulates some useful calculation */
    printf("Hello World! It's me, thread #%ld. My value is %ld!\n", tid,
           result);
    pthread_exit((void *)result /*NULL is common*/);
}
```

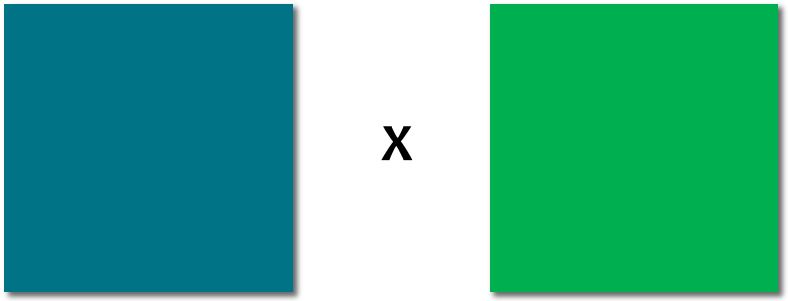
```
int pthread_join(pthread_t thread, void **valuep)
```

- Calling thread waits for `thread` to terminate.
- `pthread_join` is used to synchronize threads.
- `valuep` memory address where the return value of thread will be stored.



Example: matrix matrix product

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



Code snippet

```
/* Create all threads and do the computation */
for (int i = 0; i < n_thread; ++i)
    threads.push_back(thread(MatrixMult, size,
                             i * nrow_per_thread,                    /* row_start */
                             min(size, (i + 1) * nrow_per_thread), /* row_end */
                             ref(mat_c)));
```

```
void MatrixMult(int size, int row_start, int row_end, vector<float> &mat_c)
{
    printf("Processing row %3d to %3d\n", row_start, row_end - 1);

    for (int i = row_start; i < row_end; ++i)
        for (int j = 0; j < size; ++j)
        {
            float c_ij = 0.;

            for (int k = 0; k < size; ++k)
            {
                c_ij += MatA(i, k) * MatB(k, j);
            }

            mat_c[i * size + j] = c_ij;
        }
}
```

Thread coordination

The risks of multi-threaded programming

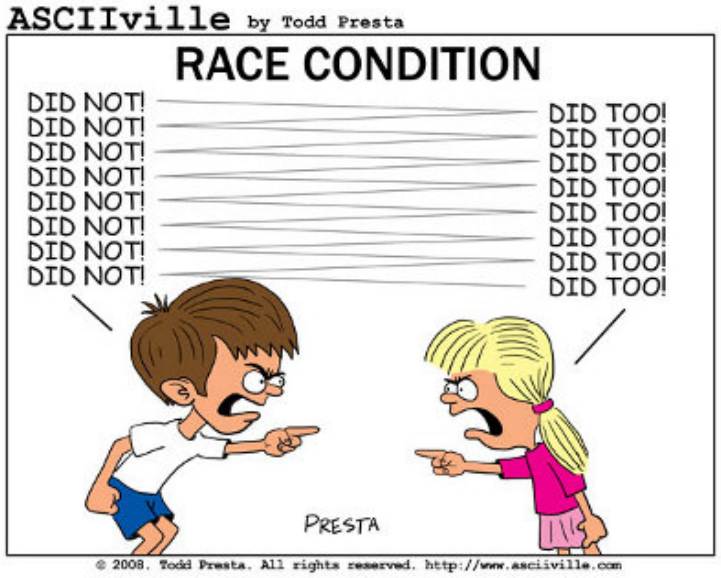
- Let us assume that a well-known bank company has asked you to implement a multi-threaded code to perform bank transactions.
- You start with the modest goal of allowing deposits.
- Clients deposit money and the amount gets credited to their accounts.
- As a result of having multiple threads running 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 1 wants to open the lock.
It has to wait.

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

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: only one thread at a time can execute that code.

Typical usage

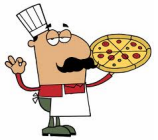
Mutex use:

- 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

`mutex_demo.c`

```c
while (1)
{
    pthread_mutex_lock(&mutex);

    if (global_task_list != NULL)
    {
        my_task = global_task_list;                /* Pop a task */
        Wait();   /* This causes several threads to use the same task pointer */
        global_task_list = global_task_list->next; /* Move forward by one task */
        pthread_mutex_unlock(&mutex);

        /* At this point, the thread delivers the pizza. */
        /* The mutex is unlocked so that other threads can work. */
        printf("Thread %ld: %s\n", thread_id, my_task->work_to_do);
        Delivery();

        /* Work is done */
        free(my_task->work_to_do); /* Free task data */
        free(my_task);             /* Free task itself */
    }
    else
    {
        pthread_mutex_unlock(&mutex); /* Don't forget to unlock the mutex */
        pthread_exit(NULL);
    }
}
```

Reference: summary of key functions

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

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

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

- Destruction of mutex.

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

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

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

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

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

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

Time ↓

Solution: threads always lock mutexes in the same order.

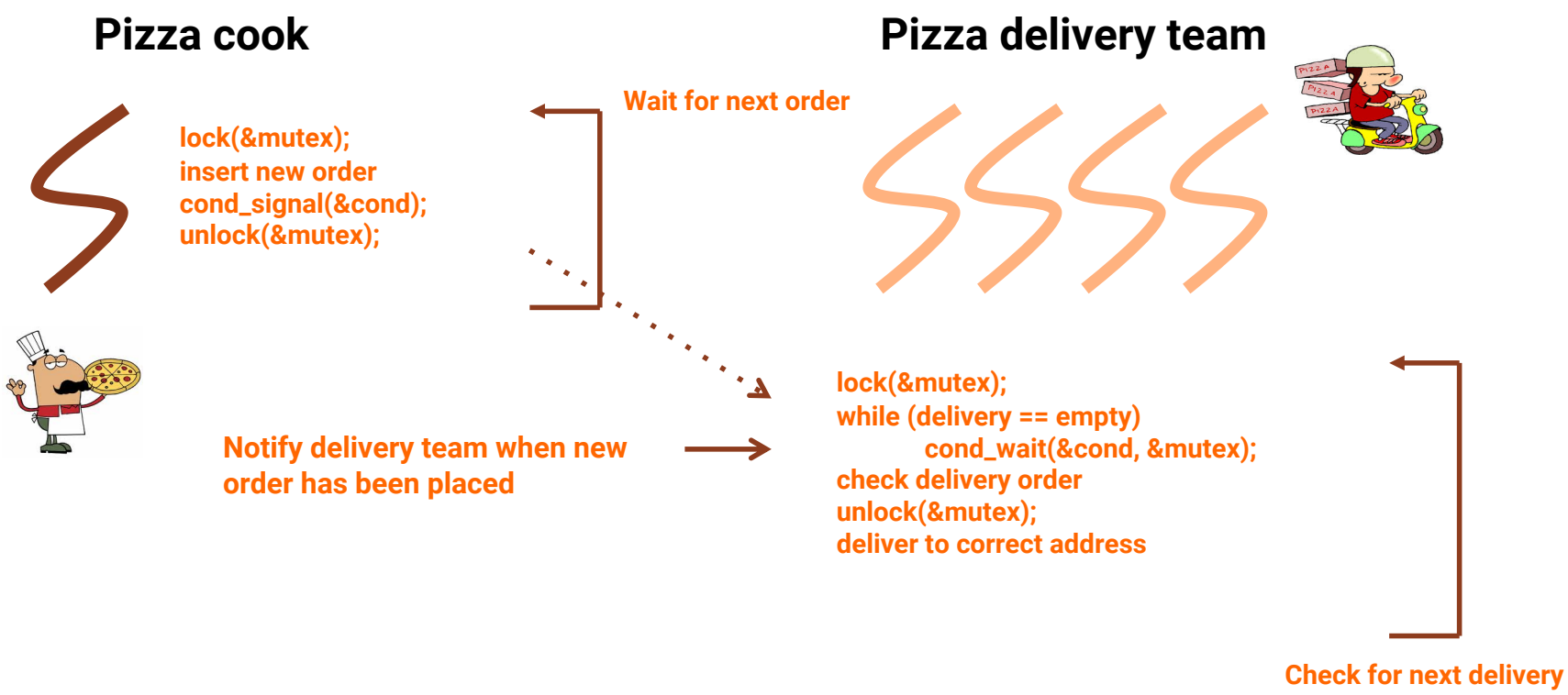
Condition variables

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.



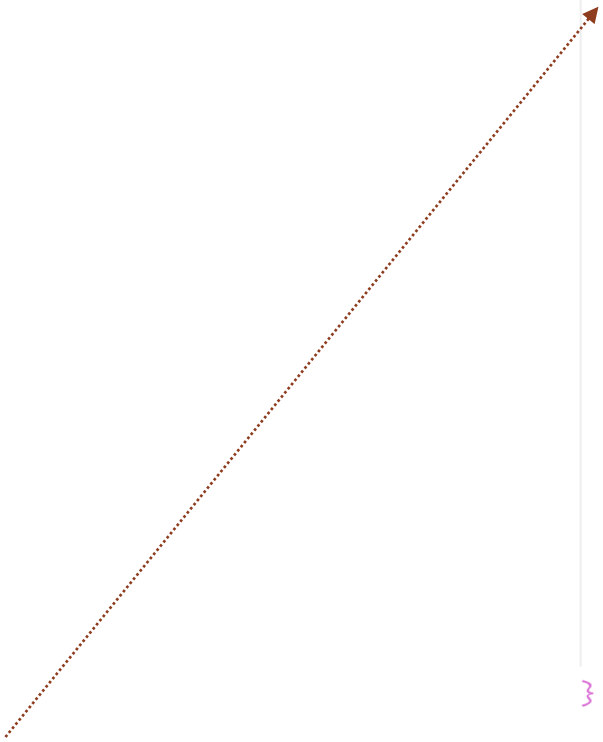
Condition variable example

`cond_var.c`

```c
while (tasks_to_do--)
{
    RandomWait();
    pthread_mutex_lock(&mutex);
    while (global_task_list == NULL)
    {
        printf("Consumer thread %ld: waiting for next task\n", thread_id);
        pthread_cond_wait(&cond, &mutex);
        /* Mutex unlocks while waiting.
           Locks when returning from pthread_cond_wait(). */
    }
    my_task = global_task_list; /* Task that thread is going to process */
    global_task_list = global_task_list->next; /* Next task */
    pthread_mutex_unlock(&mutex);

    printf("Consumer thread %ld: grabbed task %s\n", thread_id,
           my_task->work_to_do);
    printf("Consumer thread %ld: mutex unlocked\n", thread_id);

    /* At this point, the thread delivers the pizza. */
    /* The mutex is unlocked so that other threads can work. */
    free(my_task->work_to_do);
    free(my_task);
}
```



```c
pthread_cond_signal(&cond); /* Wake up a consumer. */
```

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:

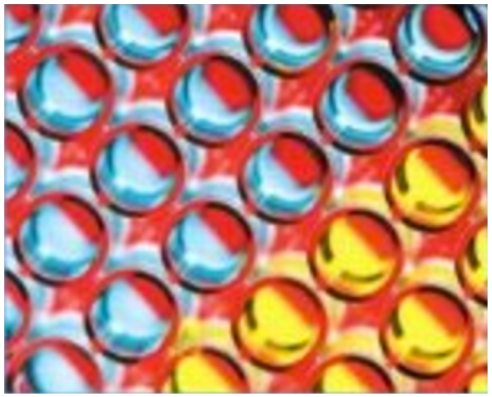
- The thread releases the mutex `mutex`.
- The thread waits until a signal is sent.
- Upon receiving a signal, the thread locks `mutex` and proceeds.

Condition signal

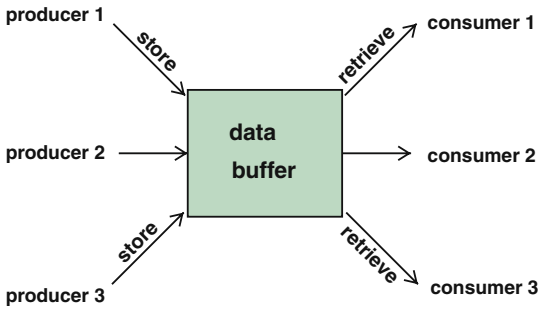
```
int pthread_cond_signal(pthread_cond_t *cond)
```

- Wakes up a single thread waiting on the variable `cond`.
- Usually, around `pthread_cond_signal`, there is code that reads/modifies the condition and needs to be protected using a mutex.
- `pthread_cond_signal` is commonly placed inside the mutex-protected block, although this is not necessary.

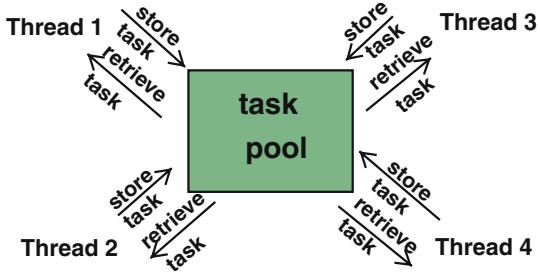
Parallel patterns and other features of Pthreads



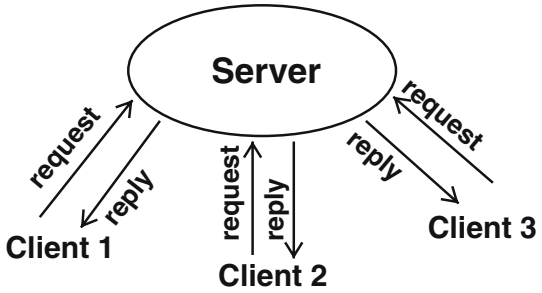
Example of useful parallel programming patterns



Producer consumer

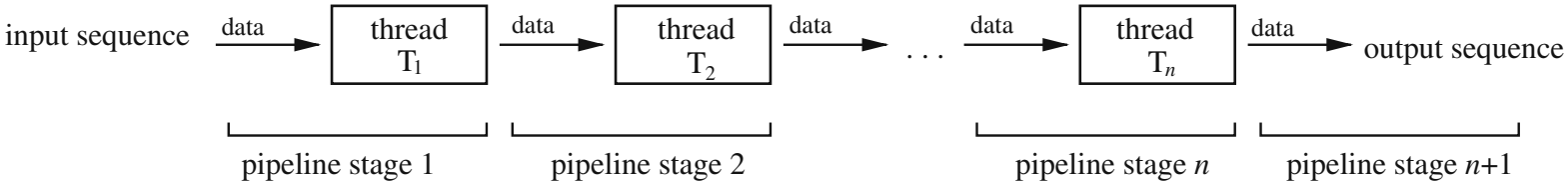


Task pool

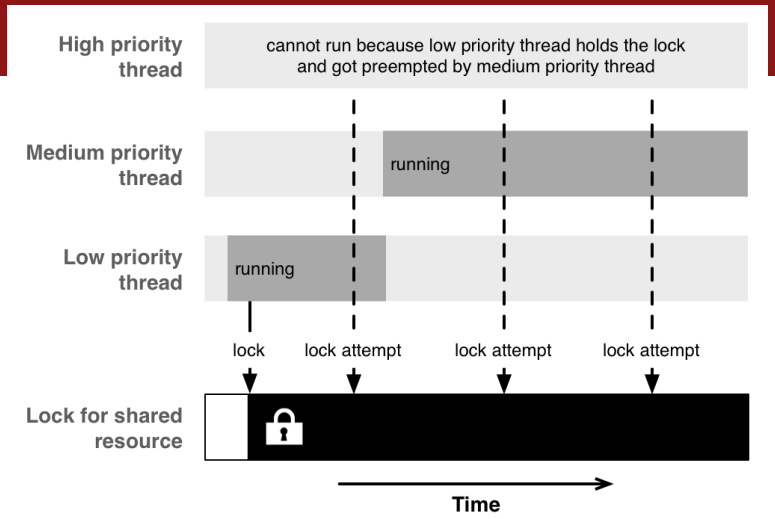


Client server

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