Mutex Locks

- Mutex Lock: Protects access to a shared resource
  - A thread locks a resource before accessing it
  - Another thread will have to wait for the resource to be unlocked
  - The first thread would unlock the resource after accessing it
- Mutex locks can be used to protect both shared resources and critical sections

Mutex States

- Locked: A single thread holds the mutex
  - Another thread trying to lock mutex will be blocked
  - Queue of blocked threads
- Unlocked: No thread holds the mutex
  - A thread trying to lock mutex will succeed and get control of the mutex

Synchronization Mechanisms

- Mutex Locks
  - Enforce protection and mutual exclusion
- Condition variables
  - Allow atomic checking of conditions
Mutex Operations

- Lock: Gain control of the mutex
  - Get lock if free
  - Block if already locked
- Unlock: Release the mutex
  - Unblock a waiting thread if any
  - Unblocked thread becomes new owner of mutex
- Trylock: Check for availability
  - Lock if available, otherwise don't block
- All these operations are atomic

Mutex Example

```
counter=1;
mutex mut;
Thread A: Thread B:
lock(mut); lock(mut);
counter++; counter--;
unlock(mut); unlock(mut);
```

- Whichever thread executes lock first gets to execute completely
- The other thread waits for its turn
- Ensures consistent value of counter

Mutex Properties

- Simple and efficient synchronization mechanism
- Typically meant to be held for short durations
- Useful for short critical sections. Examples:
  - Modifying/reading a variable value
  - Modifying pointers in a shared data linked list
- Provide exactly-one-at-a-time mutual exclusion

POSIX Mutex: pthread_mutex_t

- Initialization: pthread_mutex_init
  - Can also be set to PTHREAD_MUTEX_INITIALIZER
  - Should be done exactly once
- Destruction: pthread_mutex_destroy
- Locking/Unlocking Operations:

```
pthread_mutex_lock(pthread_mutex_t *mutex);
pthread_mutex_unlock(pthread_mutex_t *mutex);
pthread_mutex_trylock(pthread_mutex_t *mutex);
```
POSIX Mutex: Example

```c
int counter=1;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

Thread A:
pthread_mutex_lock(&mut);
counter++;
pthread_mutex_unlock(&mut);
```

Producer-Consumer Problem

```c
Producer:
while(1)
{
    produce(item);
    put(item, buffer);
}

Consumer:
while(1)
{
    item = get(buffer);
    consume(item);
}
```

- What's the critical section?
- How do we protect the critical section?

Producer-Consumer Synchronization: Using Mutex Locks

```c
Buffer_type buffer;
mutex mut;

Producer:
while(1)
{
    produce(item);
    lock(mut);
    put(item, buffer);
    unlock(mut);
}

Consumer:
while(1)
{
    lock(mut);
    item = get(buffer);
    unlock(mut);
    consume(item);
}
```

- What happens if buffer is empty?

Handling Empty Buffer: Using Mutex Locks

```c
Buffer_type buffer;
mutex mut;

Producer:
while(1)
{
    produce(item);
    lock(mut);
    int num_items=0;
    put(item, buffer);
    num_items++;
    unlock(mut);
}
```

```c
Consumer:
while(1)
{
    lock(mut);
    while (num_items==0)
        /* loop */;
    item = get(buffer);
    num_items--;
    unlock(mut);
    consume(item);
}
```
Conditional Execution

- Mutex locks:
  - Control access to a shared variable or a code segment
  - Each thread eventually accesses the variable or executes the code segment
  - Typical waiting time is small
- What if we want to execute a code segment only under certain circumstances?
  - The waiting time could be unbounded

Conditional Execution: Example

Condition to wait for: \(x == 0\)

- Naive approach: Busy Waiting

Busy Waiting

```c
int x;

Thread A:
while (x != 0) /* loop */
    do_something();

Thread B:
    x = 0;
```

Busy Waiting: Problems

```c
int x;

Thread A:
while (x != 0) /* loop */
    do_something();

Thread B:
    x = 0;
```

- Wastes CPU cycles
- Might prevent other threads from running, and even changing \(x\)
- Race conditions
Conditional Execution: Mutex Lock

- Do the following:
  - Lock a mutex
  - Test the condition \( x = 0 \)
  - If true, unlock the mutex and continue
  - If false, block and unlock the mutex

- Problem: In what order to block and unlock the mutex?
  - Block first: Mutex remains locked, other threads cannot access \( x \)
  - Unlock first: How to be notified about condition

- Need a simple way of testing and blocking

Condition Variables

- Provide atomic way of testing conditions and blocking if required
- Used in conjunction with a mutex
  - Mutex used to protect access to shared data
  - Condition variable used to signal possible satisfaction of condition, e.g.: \( x = 0 \)

Condition Variables: Operations

- Uses a mutex lock
- wait:
  - Atomically unlocks mutex and blocks
  - Thread owns mutex when it returns from wait
- signal:
  - Notifies a waiting thread about a condition
- signalAll/broadcast:
  - Notifies all waiting threads about a condition
- These operations are atomic

Condition Variables: Example

```
int x;

Thread A:
while (x!=0)
  /* loop */;
do_something();

Thread B:
x=0;
```
Condition Variables: Example

```c
int x;
cond_var cond;
mutex mut;

Thread A:
lock(mut);
while (x!=0)
    wait(cond, mut);
do_something();
unlock(mut);

Thread B:
lock(mut);
x=0;
signal(cond);
unlock(mut);
```

Condition Variables: Usage

- wait operation:
  - A thread must hold mutex when calling wait
  - Blocking and releasing of mutex is an atomic operation done inside wait
- signal operation:
  - A thread receiving signal may still wait to grab the mutex if not free
  - It must check condition even after being signaled on condition variable

Producer-Consumer Synchronization: Using Mutex Locks

```c
Buffer_type buffer;
mutex mut;

Producer:
while(1) {
    produce(item);
    lock(mut);
    put(item, buffer);
    unlock(mut);
}

Consumer:
while(1) {
    lock(mut);
    item = get(buffer);
    unlock(mut);
    consume(item);
}
```

What happens if buffer is empty?

Producer-Consumer: Handling Empty Buffer Using Condition Variables

```c
Buffer_type buffer; int num_items=0;
mutex mut; cond_var cond;

Producer:
while(1) {
    produce(item);
    lock(mut);
    put(item, buffer);
    num_items++;
signal(cond);
unlock(mut);
}

Consumer:
while(1) {
    lock(mut);
    while (num_items==0)
        wait(cond, mut);
    item = get(buffer);
    num_items--;
unlock(mut);
    consume(item);
}
**POSIX Condition Variable: pthread_cond_t**

- Initialization: pthread_cond_init
  - Can also be set to PTHREAD_COND_INITIALIZER
  - Should be done exactly once
- Destruction: pthread_cond_destroy
- Wait/signal Operations:
  ```c
  pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
  pthread_cond_timedwait(..., struct timespec *time);
  pthread_cond_signal(pthread_cond_t *cond);
  pthread_cond_broadcast(pthread_cond_t *cond);
  ```

**POSIX Condition Variable: Example**

```c
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
pthread_mutex_t mut;

pthread_mutex_lock(&mut);
while (x!=y) {
  pthread_cond_wait(&cond, &mut);
  do_something();
}
pthread_mutex_unlock(&mut);

x=y;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mut);
```