CS 521: Systems Programming

Condition Variables

Lecture 19

Waiting for Changes [1/2]

- We discussed how busy waiting is one way to prevent access to a **critical section**
- Unfortunately, busy waiting is very inefficient!
- So, we have a better way: **mutexes**
- What about when we want to wait for something to happen before our thread does its work?
  - For example: I will wait until I receive a "ready" message before I process a file

Waiting for Changes [2/2]

- We could busy wait on a variable to change
  - Once the change happens, we know we can proceed
  - Once again, this is inefficient
- Consider:
  - We have two threads, A and B
  - Thread A preprocesses the input file
  - Thread B calculates the statistics
  - In this case, thread B needs to wait for A
    - There is a conditional dependency

Waiting on a… Condition!

- To wait for something to happen, we can use **condition variables**
- Condition variables have two related functions:
  - **wait** – wait for the condition to become true
  - **signal** – inform the waiting thread that the condition has changed
- When a thread is waiting, it blocks

Blocking vs. Busy Waiting

- The big difference between blocking and actively waiting is efficiency
- Rather than constantly checking, go to sleep and let the operating system wake you up when something happens
  - Are we there yet?
    - Are we there yet?
      - Are we there yet?
        - Are we there yet?
          - Are we there yet?

Alternatives [1/2]

- You might be inclined to use plain mutexes to achieve the functionality we're discussing here
- After all, if you have Thread B try to lock a mutex that is already locked by Thread A, you can ***sort of*** pull this off
  - Thread B will block until Thread A unlocks it
- This has some big disadvantages though…
  - Can we think of all the issues here?

Alternatives [2/2]

- How about having both threads lock a mutex as the very first thing they do?
  - Race condition!
- Maybe Thread A creates Thread B?
  - Wouldn't work with more than these two threads though
  - …and doesn't that defeat the purpose? (Just use Thread A then!)
- Or the main thread locks a mutex, creates Thread A, unlocks, then creates Thread B
  - No!! Still has a race condition 

Condition Variables

- Often, mutual exclusion is not enough…
- The point of a condition variable is to give us the ability to *signal* other threads and *wait* for things
- This helps solve a classic problem: bounded producer/consumer
  - Task queue with a maximum size
  - One or more threads are *producers* that add tasks
  - One or more threads are *consumers* that remove (and process) the tasks

Initializing a Condition Variable

- Initialization is just like a mutex:

```
pthread_cond_t cond_variable =
PTHREAD_COND_INITIALIZER;
```

- Note: to use a condition variable, you also need a mutex
  - Why? This protects the condition variable logic

Using a Condition Variable [1/2]

- `pthread_cond_init(&cond, NULL);`
- `pthread_cond_wait(&cond, &mutex);` – waits for a signal
- `pthread_cond_signal(&cond);` – signals a waiting thread
  - We don't have control over ***which*** thread will wake up
- `pthread_cond_broadcast(&cond);`
  - Signals all waiting threads

Using a Condition Variable [2/2]

```
void *thread_a(void *) {
    pthread_mutex_lock(&mutex);
    while (!condition) {
        /* Note: mutex is released + reacquired here: */
        pthread_cond_wait(&cond, &mutex);
    }
    /* Do the work we were waiting to do! (mutex reacquired) */
    pthread_mutex_unlock(&mutex);
}

void *thread_b(void *) {
    pthread_mutex_lock(&mutex);
    /* Do whatever thread A is waiting for us to do ... */
    /* Signal the other thread! */
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}
```

The Condition

Let's say we have a whiteboard and we want to make sure only four students can use it at a time:

```
pthread_mutex_lock(&mutex);
while (students_at_board > 4) { // <--- our condition
    /* Note: mutex is released here: */
    pthread_cond_wait(&cond, &mutex);
    /* Note: mutex is reacquired here */
}
```

Common Producer/Consumer Setup

- Let's use condition variables to implement producer-consumer synchronization
- (A work queue)
  - Thread 1:

    Producer – creates the tasks
  - Thread 2…N:

    Consumers – wait for tasks and carry them out
- Usually it's easier to produce the tasks than consume them (resulting in this one-to-many setup), but that doesn't have to always be true!

Web Server: Prod/Con [1/3]

- Imagine you are writing a web server
- The server listens for incoming requests and places them in a queue to be handled by worker threads
  - We can't have the main thread serve the requests directly because then it can't listen for more connections
    - Web servers need to handle thousands of concurrent clients!
- Ok, protect the queue with a mutex + condition variable and have **N** threads wait for work

Web Server: Prod/Con [2/3]

- Now the main thread can accept connections and the worker threads serve up the HTML pages, images, etc.
- Except… we still have a problem
- What happens if there are **too many** incoming connections?
  - If we use a fixed-sized queue, we'll eventually run out of space and segfault
  - If we use a flexible queue (say, an `elist`), then we could run out of memory
- We need to be able to stop producing temporarily

Web Server: Prod/Con [3/3]

- How can we stop production?
- …
- Easy answer: don't put any more connections in the work queue
  - But how do we know when it has space again?
- You guessed it! Another condition variable!
  - Block (wait) if the queue is currently full, and signal after each thread completes its work
  - Places ***backpressure*** on the network connection

Building a Barrier [1/2]

- We can also use condition variables to build a barrier
  - Helpful on systems like macOS that do not support them
- `pthread_cond_broadcast` can signal all waiting threads
- Each thread that calls `barrier()` increments a counter
  - Once the counter hits **N**, then do a broadcast to all waiting threads

Building a Barrier [2/2]

```
void barrier(void)
{
    pthread_mutex_lock(&bar_mut);
    bar_count++;
    if (bar_count == thread_count) {
        bar_count = 0;
        pthread_cond_broadcast(&bar_cond);
    } else {
        // Wait unlocks mutex and puts thread to sleep.
        //    Put wait in while loop in case some other
        // event awakens thread.
        while (pthread_cond_wait(&bar_cond, &bar_mut) != 0);
        // Mutex is relocked at this point.
    }
    pthread_mutex_unlock(&bar_mut);
}
```