CSCI 4061: Recitation 8
11-2-15
(Happy November!)
Today: Synchronization

Sync methods:
- Mutex
- Condition variables
- Exercises
Synchronization

- A section of code which works with shared resources can have unpredictable results depending on the order in which the threads execute.

- This is called a **critical section**.
Mutual Exclusion

• A **mutex** lets you lock a code section so that only one thread at a time executes a critical section.

• First, identify the critical sections.

• Which line is a critical section in the following code?
for (i=0; i<NTHREADS; i++)
    pthread_create(&threads[i], &attr, &do_work, NULL);

void *do_work() {
    int i;
    for (i=0; i<LOOPS; i++) {
        if(sleepFlag)
            usleep(rand()%100);
        sum = sum + 1; }
    pthread_exit(NULL);
}
for (i=0; i<NTHREADS; i++)
    pthread_create(&threads[i], &attr, &do_work, NULL);
...

void *do_work() {
    int i;
    for (i=0; i<LOOPS; i++) {
        if(sleepFlag)
            usleep(rand()%100);
        sum = sum + 1;
    }
    pthread_exit(NULL);
}
Mutex

• Identify critical sections,
• Then create a mutex for each shared resource.
• Lock and unlock them before and after each critical section.
Mutex signatures

#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

... 

void *do_work() {
    int i;
    for (i=0; i<LOOPS; i++) {
        if(sleepFlag)
            usleep(rand()%100);
        pthread_mutex_lock(&mutex);
        sum = sum + 1;
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}
How is this different?

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *do_work() {
    pthread_mutex_lock(&mutex);
    int i;
    for (i=0; i<LOOPS; i++) {
        if(sleepFlag){
            if(sleepFlag)
                usleep(rand() % 100);
            sum = sum + 1;
        }
    }
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}
```
Conditional Variables

• Useful when you only want to enter a critical section under certain conditions.

• Avoid “busy waiting”, where a single thread wastes time repeatedly checking for a condition while waiting for it to become true. (“Are we there yet? Are we there yet? Are we there yet? Are we there yet? Are we there yet? Are we there yet? Are we there yet?”)
Conditional Variables

• To create a conditional variable:
  
  ```c
  pthread_cond_t condvar = PTHREAD_COND_INITIALIZER;
  ```

• To wait for a conditional variable:
  
  ```c
  pthread_cond_wait(&condvar, &mutex);
  ```

• This will unlock the mutex, wait for a signal `condvar`, then try to lock the mutex when the signal arrives.
Conditional Variables

• Example: Producer-Consumer Problem
• Consumer has to wait while the buffer is empty, producer has to wait while buffer is full.

• `busywait.c` contains a busy wait, `condvar.c` shows how to fix this with conditional variables.
Pitfalls

• Deadlock: All threads are waiting for each other, then none finish.

• Example:

  Thread 1
  ```c
  pthread_mutex_lock(&mutexA);
  pthread_mutex_lock(&mutexB);
  //Critical section
  pthread_mutex_unlock(&mutexB);
  pthread_mutex_unlock(&mutexA);
  pthread_mutex_unlock(&mutexA);
  pthread_mutex_unlock(&mutexB);
  ```

  Thread 2
  ```c
  pthread_mutex_lock(&mutexB);
  pthread_mutex_lock(&mutexA);
  //Critical section
  pthread_mutex_unlock(&mutexA);
  pthread_mutex_unlock(&mutexA);
  pthread_mutex_unlock(&mutexB);
  ```
Pitfalls

• Livelock: All threads are constantly responding to each other and create a (potentially) infinite loop.

• Starvation: One thread hogs resources while another is constantly waiting for it.
Exercise

• busywait.c
• condvar.c
• deadlock.c
• deadlock_single_lock.c
• deadlock_total_lock.c
• deadlock_trywait.c
• race.c
• race_fixed.c
Questions?