Recitation 8
3-24-14
Today

• Synchronization
  – Mutex locks
  – Condition Variables (Introduction)
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, you have to identify your critical sections.
- Example: `sum = sum + 1;` (line 16) in `race.c`
- Then create a mutex for each shared resource, and lock and unlock them before and after each critical section.
POSIX Mutex

- **Initialize**
  - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`

- **Lock**
  - `pthread_mutex_lock(&mutex);`

- **Unlock**
  - `pthread_mutex_unlock(&mutex);`
Exercise

• Fix race.c
Mutual Exclusion

- How to fix race.c: (race_fixed.c)

```c
//snip
int sleepFlag = 1;
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);
}
//snip
```
Mutual Exclusion

• How is this different?

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

• Problem: Make a thread wait until some condition is satisfied (Say \( x == y \)). (x,y – shared variables among threads)

• Avoid Busy Waiting (\( while \ (x != y); \)):
  – Consumes unnecessary CPU cycles
  – Depending on Scheduling if other threads never get a chance to execute, shared variables may not change and thread may busy wait forever
Condition Variables

• Correct way:
  – while(true)
    • **Lock** a mutex
    • Test condition \((x == y)\)
    • If true, **unlock** mutex and **exit** loop
    • If false, **suspend** the thread and **unlock** the mutex

Reference: RR § 13.4
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, then try to lock the mutex when the signal arrives.
Pitfalls

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

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_mutex_lock(&amp;mutexA);</td>
<td>pthread_mutex_lock(&amp;mutexB);</td>
</tr>
<tr>
<td>pthread_mutex_lock(&amp;mutexB);</td>
<td>pthread_mutex_lock(&amp;mutexA);</td>
</tr>
<tr>
<td>//Critical section</td>
<td>//Critical section</td>
</tr>
<tr>
<td>pthread_mutex_unlock(&amp;mutexB);</td>
<td>pthread_mutex_unlock(&amp;mutexA);</td>
</tr>
<tr>
<td>pthread_mutex_unlock(&amp;mutexA);</td>
<td>pthread_mutex_unlock(&amp;mutexB);</td>
</tr>
</tbody>
</table>
Pitfalls

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

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

• Questions on material?
• Questions on PA3?