CS472 - Computer Architecture Lecture 6

D. Kevin McGrath

April 18, 2012
Thread synchronization

- Need for synchronization
- Mutexes
- Condition variables
1. Need for synchronization

2. Mutexes

3. Condition variables
Race conditions and critical sections

- A race condition is a flaw where the output or result of the process is unexpectedly and critically dependent on the sequence or timing of other events.
- Section of code known as the “critical region”
- Example: Bank transactions

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read balance: $1000</td>
<td>Read balance: $1000</td>
<td>$1000</td>
</tr>
<tr>
<td>Deposit: $200</td>
<td>Deposit: $200</td>
<td>$1000</td>
</tr>
<tr>
<td>Update balance: $1000 + $200</td>
<td>Update balance: $1000 + $200</td>
<td>$1200</td>
</tr>
</tbody>
</table>

Oopsie!
Synchronization

- In the above example, the balance should have been protected by synchronization constructs.
- The read/update operation on balance is the critical region.
- Alternatively, the operations of reading and updating should have been done *atomically*.
- Unfortunately, there are no atomic operations which can perform such an update.
- Other constructs must be used, such as mutexes.
Outline

1. Need for synchronization
2. Mutexes
3. Condition variables
Mutexes

- A *mutex* is a mutual exclusion lock
- Ensures only a single thread can access a critical region at any given time
- Attempts to do so in a way that prevents *serialization* of the critical region
- Must be created and initialized prior to use.
  - `pthread_mutex_t mutex;`
  - `pthread_mutexattr_t attr;`
  - `pthread_mutex_init(mutex, attr);`
  - `pthread_mutex_destroy(mutex);`
  - `pthread_mutexattr_init(attr);`
  - `pthread_mutexattr_destroy(attr);`
Mutexes creation and destruction

- Mutex variables must be declared with type `pthread_mutex_t`, and must be initialized before they can be used. There are two ways to initialize a mutex variable:
  - Statically:
    ```c
    pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
    ```
  - Dynamically, using `pthread_mutex_init()`

- Dynamic creation can be used to set attributes (not all are supported by all implementations)
  - Protocol - specifies protocol to prevent priority inversion
  - Priority ceiling - priority ceiling of mutex
  - Process-shared - specifies the process sharing of the mutex
Mutex usage

- Functions:
  - `pthread_mutex_lock(mutex);`
  - `pthread_mutex_trylock(mutex);`
  - `pthread_mutex_unlock(mutex);`

- A thread must acquire a lock on the specified mutex
- If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked
  - Cannot recursively lock a mutex
- `pthread_mutex_trylock()` will attempt to lock a mutex, but will not block
  - Instead, returns an error code EBUSY
  - Can be used for busy waiting on a mutex
- `pthread_mutex_unlock()` used to unlock a mutex
  - Can’t unlock already unlocked mutex
  - Can’t unlock a mutex you did not lock
Mutex Usage

- Mutexes must be locked upon entry to a critical region, unlocked upon exit
- If multiple threads blocking on a single mutex, order of wake up is arbitrary
- Entirely voluntary mechanism
- Any thread chooses to ignore the mechanism, everything breaks
- All burden on programmer
Outline

1. Need for synchronization
2. Mutexes
3. Condition variables
Condition variables

- Often, events need to occur at specific values of a variable
  - Bank balance drops below 0
  - Thermostat hits target temperature (plus \( \delta \))
- With only mutexes, threads would have to lock, check value, unlock
  - This overhead is not good
  - Amounts to busy waiting, which is a bad choice if the wait is potentially unbounded
- Condition variables provide a solution to this issue
  - Always used in conjunction with a mutex
  - Used to signal other threads
Need for synchronization

Mutexes

Condition variables

**pthread_once**

- `pthread_once_t *once_control`
- `void (*init_routine)(void)`
- `pthread_once (once_control, init_routine);`
- Useful for executing a routine exactly once in a process
- Control variable must be initialized
  - `pthread_once_t once_control = PTHREAD_ONCE_INIT;`
  - Do not make copies of this variable
- Subsequent calls after first have no effect