CSC369 Tutorial 3
Synchronization Primitives
Synchronization Mechanisms

- Locks
  - Very primitive constructs with minimal semantics
- Semaphores
  - A generalization of locks
  - Easy to understand, hard to program with
- Condition Variables
  - Constructs used in implementing monitors (more on this later)
Locks (Mutexes)

• Synchronization mechanisms with 2 operations: acquire(), and release()

• In simplest terms: an object associated with a particular critical section that you need to “own” if you wish to execute in that region
  • semantics identical to spinlocks that we’ve seen before

• Simple semantics to provide mutual exclusion:
  acquire(lock);
    // CRITICAL SECTION
  release(lock);

• Downsides:
  • Can cause deadlock if not careful
  • Cannot allow multiple concurrent accesses to a resource
POSIX Locks

• POSIX locks are called mutexes (since locks provide mutual exclusion)

• A few calls associated with POSIX mutexes:
  pthread_mutex_init(mutex, attr)
    • Initialize a mutex
  pthread_mutex_destroy(mutex)
    • Destroy a mutex
  pthread_mutex_lock(mutex)
    • Acquire the lock
  pthread_mutex_trylock(mutex)
    • Try to acquire the lock (more on this later...)
  pthread_mutex_unlock(mutex)
    • Release the lock
Initializing & Destroying
POSIX Mutexes

• POSIX mutexes can be created statically or dynamically
  • Statically, using `PTHREAD_MUTEX_INITIALIZER`
    
    ```
    pthread_mutex_t mx = PTHREAD_MUTEX_INITIALIZER;
    ```
    
    • Will initialize the mutex will default attributes
    • Only use for static mutexes; no error checking is performed
  
  • Dynamically, using the `pthread_mutex_init` call
    
    ```
    int pthread_mutex_init(pthread_mutex_t *mutex, const
    pthread_mutexattr_t *attr);
    ```
    
    • `mutex`: the mutex to be initialized
    • `attr`: structure whose contents are used at mutex creation to determine
      the mutex’s attributes
      • Same idea as `pthread_attr_t` attributes for threads

• Destroy using `pthread_mutex_destroy`
  
  ```
  int pthread_mutex_destroy(pthread_mutex_t *mutex);
  ```
  
  • `mutex`: the mutex to be destroyed
  • Make sure it’s unlocked! (destroying a locked mutex leads to undefined
    behaviour...)
Acquiring and Releasing POSIX Locks

• Acquire
  int pthread_mutex_lock(pthread_mutex_t *mutex);
  • mutex: the mutex to lock (acquire)
  • If mutex is already locked by another thread, the call will block until the mutex is unlocked
  int pthread_mutex_trylock(pthread_mutex_t *mutex);
  • mutex: the mutex to TRY to lock (acquire)
  • If mutex is already locked by another thread, the call will return a “busy” error code (EBUSY)

• Release
  int pthread_mutex_unlock(pthread_mutex_t *mutex);
  • mutex: the mutex to unlock (release)
Banking Example

• Bank account balance maintained in one variable “int balance”

• Transactions: deposit or withdraw some amount from the account (+/- balance)

• Unprotected, concurrent accesses to your balance could create race conditions
  • A specific example?
Banking Example

- Thread 1 withdraws 100

  ```java
  int new_balance = balance - amount;
  balance = new_balance;
  ```

- Thread 2 withdraws 100

  ```java
  int new_balance = balance - amount;
  balance = new_balance;
  ```

- End with balance – 100 instead of balance – 200
- Bank error in your favour? Cold be the other way around!
- Idea: put a lock around the code that modifies balance so only a single thread accesses it at any given time
Banking Example

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 200
int balance=0;
pthread_mutex_t bal_mutex;

int main (int argc, char *argv[]) {
    pthread_t thread[NUM_THREADS];
    pthread_mutex_init(&bal_mutex, NULL);
    for (int t = 0; t < NUM_THREADS; t += 2) {
        int rc = pthread_create(&thread[t], NULL, deposit, (void*)10);
        if (rc != 0) {
            printf("ERROR: pthread_create() returned %d\n", rc);
            exit(-1);
        }
        rc = pthread_create(&thread[t+1], NULL, withdraw, (void*)10);
        if (rc != 0) {
            printf("ERROR: pthread_create() returned %d\n", rc);
            exit(-1);
        }
    }
    //...
```
Banking Example

//...
for (int t = 0; t < NUM_THREADS; t++) {
    void *status = NULL;
    int rc = pthread_join(thread[t], &status);
    if (rc != 0) {
        printf("ERROR; return code from "
                "pthread_join() is %d\n", rc);
        exit(-1);
    }
}
printf("Final Balance is %d.\n", balance);
return 0;
void *deposit(void *amt) 
{
    pthread_mutex_lock(&bal_mutex);
    
    // CRITICAL SECTION
    int amount = (int)amt;
    int new_balance = balance + amount;
    balance = new_balance;

    pthread_mutex_unlock(&bal_mutex);

    return NULL;
}

void *withdraw(void *amt) 
{
    pthread_mutex_lock(&bal_mutex);
    
    // CRITICAL SECTION
    int amount = (int)amt;
    int new_balance = balance - amount;
    balance = new_balance;

    pthread_mutex_unlock(&bal_mutex);

    return NULL;
}