Threads in C
Pthreads synchronization

EPL222 – Lab4

Synchronizing Threads

- Most of threaded programs have threads that interact with one another
  - Interaction in the form of sharing access to variables
    - Multiple concurrent reads (ok)
    - Multiple concurrent writes (not ok, outcome non-deterministic)
    - One write, multiple reads (not ok, outcome non-deterministic)
  - Need to make sure that the outcome is deterministic
    - Synchronization: allowing concurrent accesses to variables, removing non-deterministic outcome by enforcing some order during thread execution
- Three basic synchronization primitives
  - mutex locks
  - condition variables
  - semaphores

Synchronizing Threads

- Mutual exclusion (mutex):
  - guard against multiple threads modifying the same shared data simultaneously
  - provides locking/unlocking critical code sections where shared data is modified
  - each thread waits for the mutex to be unlocked (by the thread who locked it) before performing the code section

```
Thread 1:                      Thread 2:
    insert A to tree               insert B to tree
```

↓

```
Thread 1:                      Thread 2:
    lock(tree)                     lock(tree)
    insert A to tree               insert B to tree
    unlock(tree)                   unlock(tree)
```

Mutex variables

- A typical sequence in the use of a mutex is as follows:
  - Create and initialize a mutex variable
  - Several threads attempt to lock the mutex
  - Only one succeeds and that thread owns the mutex
  - The owner thread performs some set of actions
  - The owner unlocks the mutex
  - Another thread acquires the mutex and repeats the process
  - Finally the mutex is destroyed

Basic Mutex Functions

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- a new data type named pthread_mutex_t is designated for mutexes
- a mutex is like a key (to access the code section) that is handed to only one thread at a time
  - if multiple threads try to gain lock at the same time, the return order is based on priority of the threads
  - higher priorities return first
  - no guarantees about ordering between same priority threads
- the attributes of a mutex can be controlled by using function pthread_mutex_init()
- the lock/unlock functions work in tandem
- the trylock function will attempt to lock the mutex. However, if the mutex is already locked, it will return immediately with a "busy" error code

Mutex example

```
#include <pthread.h>
...
pthread_mutex_t my_mutex;
...
int main(){
   int tmp;
   ...
   // initialize the mutex
   tmp = pthread_mutex_init(&my_mutex, NULL );
   ...
   // create threads
   ...
   pthread_mutex_lock(&my_mutex );
      do_something_private();
   pthread_mutex_unlock(&my_mutex );
   ...
   pthread_mutex_destroy(&my_mutex );
   return 0;
}
```

Whenever a thread reaches the lock/unlock block, it first determines if the mutex is locked.

If so, it waits until it is unlocked.

Otherwise, it takes the mutex and locks it, unlocking it when it's done.

Mutex example

```c
#include <stdio.h>
#include <pthread.h>

#define MAX_SIZE  5
pthread_mutex_t bufLock;
int count;

void producer(char* buf) {
   for(;;) {
      while(count == MAX_SIZE);
      pthread_mutex_lock(&bufLock);
      if(count<MAX_SIZE){
         buf[count] = getChar();
         count++;
      }
      pthread_mutex_unlock(&bufLock);
   }
}
```

```c
void consumer(char* buf) {
   for(;;) {
      while(count == 0);
      pthread_mutex_lock(&bufLock);
      if(count>0){
         useChar(buf[count-1]);
         count--;
      }
      pthread_mutex_unlock(&bufLock);
   }
}

int main() {
   char buf[MAX_SIZE];
   pthread_t p;
   count = 0;
   pthread_mutex_init(&bufLock, NULL);
   pthread_create(&p, NULL, (void*)producer, &buf);
   consumer(buf);
   return 0;
}
```

Condition Variables

- Notice in the previous example a spin-lock was used wait for a condition to be true
  - the buffer to be full or empty
  - spin-locks require CPU time to run
    - waste of cycles
- Condition variables allow a thread to block until a specific condition becomes true
  - recall that a blocked process cannot be run
    - doesn’t waste CPU cycles
  - blocked thread goes to wait queue for condition
- When the condition becomes true, some other thread signals the blocked thread(s)

Condition Variables

- A condition variable is created like a normal variable

```
pthread_cond_t cv;
```

  - must be initialized before being used
  - can only be initialized once

```
int pthread_cond_init(pthread_cond_t *cv,
         const pthread_condattr_t *cvattr);
```

  - *cv*: a pointer to the condition variable to be initialized
  - *cvattr*: attributes of the condition variable – usually NULL

Synchronization with condition variables

```
int pthread_cond_wait(pthread_cond_t *cv,
                      pthread_mutex_t *mutex);
```

- A wait call is used to block a thread on a CV
- Blocks the thread until the specific condition is signalled
  - even after signal, condition may still not be true!
- Should be called with mutex locked
  - the mutex is automatically released by the wait call
  - the mutex is automatically reclaimed on return from wait call (condition is signaled)
- *cv*: condition variable to block on
- *mutex*: the mutex to release while waiting

Synchronization with condition variables

```
int pthread_cond_signal(pthread_cond_t *cv);
```

- A signal call is used to “wake up” a single thread waiting on a condition variable
  - multiple threads may be waiting and there is no guarantee as to which one wakes up first
  - thread to wake up does not actually wake until the lock indicated by the wait call becomes available
  - condition thread was waiting for may not be true when the thread actually gets to run again
    - should always do a wait call inside of a while loop
- Called after mutex is locked, and must unlock mutex after
- *cv*: condition variable to signal on

Condition variable example

```
#include <stdio.h>
#include <pthread.h>

#define MAX_SIZE  5
pthread_mutex_t bufLock;
pthread_cond_t notFull, notEmpty;
int count;

void producer(char* buf) {
   for(;;) {
      pthreads_mutex_lock(&bufLock);
      while(count == MAX_SIZE)
         pthread_cond_wait(&notFull,&bufLock);
      buf[count] = getChar();
      count++;
      pthread_cond_signal(&notEmpty);
      pthread_mutex_unlock(&bufLock);
   }
}
```

```
void consumer(char* buf) {
   for(;;) {
      pthread_mutex_lock(&bufLock);
      while(count == 0)
         pthread_cond_wait(&notEmpty, &bufLock);
      useChar(buf[count-1]);
      count--;
      pthread_cond_signal(&notFull);
      pthread_mutex_unlock(&bufLock);
   }
}

int main() {
   char buf[MAX_SIZE];
   pthread_t p;
   count = 0;
   pthread_mutex_init(&bufLock, NULL);
   pthread_cond_init(&notFull, NULL);
   pthread_cond_init(&notEmpty, NULL);
   pthread_create(&p, NULL, (void*)producer, &buf);
   consumer(buf);
   return 0;
}
```

More on Signaling Threads

- The previous example only wakes a single thread
  - not much control over which thread this is
- Perhaps all threads waiting on a condition need to be woken up
  - can do a broadcast of a signal
  - very similar to a regular signal in every other respect

```
int pthread_cond_broadcast(pthread_cond_t *cv);
```

- *cv*: condition variable to signal all waiters on

Semaphores

- Permit a limited number of threads to execute a section of the code
- Similar to mutexes
- should include the *<semaphore.h>* header file
- Semaphore functions do not have **pthread_** prefixes; instead, they have **sem_** prefixes

Semaphores

- pthreads allows the specific creation of semaphores
  - can do increments and decrements of semaphore value
  - semaphore can be initialized to any value
  - thread blocks if semaphore value is less than or equal to zero when a decrement is attempted
  - as soon as semaphore value is greater than zero, one of the blocked threads wakes up and continues
    - no guarantees as to which thread this might be

Basic Semaphore Functions

`sem_t sem;`
- Semaphores are created like other variables

```
int sem_init(sem_t *sem, int pshared,
                            unsigned int value);
```
- initializes a semaphore object pointed to by *sem*
- *pshared* is a sharing option; a value of 0 means the semaphore is local to the calling process
- gives an initial value *(value)* to the semaphore

```
int sem_destroy(sem_t *sem);
```
- frees the resources allocated to the semaphore *sem*
- usually called after pthread_join()
- an error will occur if a semaphore is destroyed for which a thread is waiting

Basic Semaphore Functions

`int sem_post(sem_t *sem);`

- Atomically increases the value of a semaphore by 1, i.e., when 2 threads call sem_post simultaneously, the semaphore's value will also be increased by 2 (there are 2 atoms calling)
- if any threads are blocked on the semaphore, they will be unblocked
- *sem*: the semaphore to increment

`int sem_wait(sem_t *sem);`

- Atomically decreases the value of a semaphore by 1; but always waits until the semaphore has a non-zero value first
- If the semaphore value is greater than 0, the sem_wait() call returns immediately
  - otherwise it blocks the calling thread until the value becomes greater than 0
- *sem*: semaphore to try and decrement

`int sem_trywait(sem_t *sem);`

- The same as sem_wait(), except that if the decrement cannot be immediately performed, then call returns -1 instead of blocking
- *sem*: semaphore to try and decrement

Semaphore example

```
#include <pthread.h>
#include <semaphore.h>
...
void *thread_function( void *arg );
...
// global variable just like mutexes
sem_t semaphore;
...
int main() {
    int tmp;
    ...
    // initialize the semaphore
    tmp = sem_init( &semaphore, 0, 0 );
    ...
    // create threads
    pthread_create( &thread[i], NULL, thread_function, NULL );
    ...
    while ( still_has_something_to_do() )  {
         sem_post( &semaphore );
         ...
    }
    ...
    pthread_join( thread[i], NULL );
    sem_destroy( &semaphore );
    return 0;
}
```

```
void *thread_function( void *arg ){
    sem_wait( &semaphore );
    perform_task_when_sem_open();
    ...
    pthread_exit(NULL);
}
```

Semaphore example

- The main thread increments the semaphore's count value in the while loop
- The threads wait until the semaphore's count value is non-zero before performing perform_task_when_sem_open() and further
- Child thread activities stop only when pthread_join() is called

Semaphore example

```
#include <stdio.h>
#include <semaphore.h>

#define MAX_SIZE  5
sem_t empty, full;

void producer(char* buf) {
   int in = 0;
   for(;;) {
      sem_wait(&empty);
      buf[in] = getChar();
      in = (in + 1) % MAX_SIZE;
      sem_post(&full);
   }
}
```

```
void consumer(char* buf) {
   int out = 0;
   for(;;) {
      sem_wait(&full);
      useChar(buf[out]);
      out = (out + 1) % MAX_SIZE;
      sem_post(&empty);
   }
}

int main() {
   char buffer[MAX_SIZE];
   pthread_t p;
   sem_init(&empty, 0, MAX_SIZE);
   sem_init(&full, 0, 0);
   pthread_create(&p, NULL, (void*)producer, &buffer);
   consumer(buffer);
   return 0;
}
```