POSIX Semaphores

#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
• initialize the semaphore pointed to by sem.
• pshared indicates whether the semaphore is local to the current
  process (pshared is 0) or is to be shared between several
  processes (pshared is not 0).
• The initial value of the semaphore is set to value.
• Return 0 on success, -1 on error

6.1

POSIX Semaphores

int sem_wait(sem_t * sem);
• Suspend the calling thread until the semaphore’s value > 0,
  then atomically decreases the value.
• Return 0 on success, -1 on error

int sem_post(sem_t * sem);
• Atomically increases the value of the semaphore. If the
  semaphore’s value becomes greater than 0, a process or
  thread blocked in a sem_wait call will be woken up.
• Return 0 on success, -1 on error

int sem_destroy(sem_t * sem);
• Destroy a semaphore
• No threads should be waiting on the semaphore when called.
• Return 0 on success, -1 on error

6.2
/**
 * A simple semaphore demonstration.
 * Link with the pthread library:
 * g++ thisfile.cc -lpthread
 */

#include <pthread.h>
#include <iostream>
#include <semaphore.h>
using namespace std;

sem_t s1, s2;

void *thread1(void *x)
{
    cout << "1) Thread 1 signaling\n";
    sem_post(&s2);
    sem_wait(&s1);
    cout << "4) Thread 1 past wait\n";
    return NULL;
}

int main()
{
    sem_init(&s1, 0, 0);
    sem_init(&s2, 0, 0);
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    sem_destroy(&s1);
    sem_destroy(&s2);
    cout << "Threads are done\n";
    return 0;
}

Pthread Mutexes

- A mutex is useful for protecting shared data structures from concurrent modifications
- A mutex has two possible states: *unlocked* (not owned by any thread) and *locked* (owned by one thread).
  - Initial state is unlocked
- A mutex can never be owned by two different threads simultaneously.
- #include <pthread.h>
Creating/Destroying Mutexes

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

- Initialize the mutex object according to the attributes specified in mutexattr.
  - If mutexattr is NULL, default attributes are used.
- Return 0 on success; otherwise, an error number is returned.

```c
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Destroys a mutex object. The mutex must be unlocked when called.
- Return 0 on success; otherwise, an error number is returned.

6.5

Locking/Unlocking Mutexes

```c
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- If mutex is unlocked, it becomes locked and owned by the calling thread.
- If mutex is locked, the calling thread is suspended until it obtains the lock.
- Return 0 on success; otherwise, an error number is returned.

```c
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Unlocks the mutex.
- Return 0 on success; otherwise, an error number is returned.

6.6
A Mutex Example

```c
char buffer;
int buffer_has_item = 0;
pthread_mutex_t mutex;

main()
{
    pthread_t reader;
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&reader, NULL, reader_function, NULL);
    writer_function();
    pthread_join(reader, NULL);
}
```

A Mutex Example (Cont.)

```c
void writer_function(void)
{
    while(1) {
        pthread_mutex_lock( &mutex );
        if ( buffer_has_item == 0 ) {
            buffer = make_new_item();
            buffer_has_item = 1;
        }
        pthread_mutex_unlock( &mutex );
        sleep(2);
    }
}
```

```c
void* reader_function(void*)
{
    while(1) {
        pthread_mutex_lock( &mutex );
        if ( buffer_has_item == 1 ) {
            consume_item( buffer );
            buffer_has_item = 0;
        }
        pthread_mutex_unlock( &mutex );
        sleep(2);
    }
}
```
Pthread Condition Variables

- A condition variable allows a thread to suspend execution until a condition on shared data holds.
- A condition variable is always used in conjunction with a mutex variable.
- #include <pthread.h>

Creating/Destroying Condition Variables

```c
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
```

- Initialize the condition variable cond using the attributes specified in cond_attr.
  - If cond_attr is NULL, use default attributes.
- Return 0 on success; otherwise, an error number is returned.

```c
int pthread_cond_destroy(pthread_cond_t *cond)
```

- Destroy the condition variable cond.
- No threads must be waiting on the condition variable on entrance.
- Return 0 on success; otherwise, an error number is returned.
Waiting/Signaling on Condition Variables

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
• The mutex must be locked by the calling thread on entrance.
• Atomically unlocks the mutex and suspends the thread execution until the condition variable is signaled.
• Before returning to the calling thread, pthread_cond_wait re-lock the mutex.
• Return 0 on success; otherwise, an error number is returned

int pthread_cond_signal(pthread_cond_t *cond)
• Wakeup one of the threads that are waiting on cond. If no threads are waiting on cond, nothing happens.
• Should be called after mutex is locked, and must unlock mutex in order for pthread_cond_wait() to complete.
• Return 0 on success; otherwise, an error number is returned

/*
* A simple illustration of pthread condition variables
*
* Link with the pthread library: g++ thisfile.cc -lpthread
*/
#include <iostream>
#include <pthread.h>
void* thread2(void*)
{
  int i;
  for (i=0; i<10; i++) {
    cout << "Thread 2 setting x to " << i << "\n";
    pthread_mutex_lock(&mutex);
    x = i;
    if (x>5) pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}
int x;
pthread_mutex_t mutex;
pthread_cond_t cond;
void* thread1(void*)
{
  // wait until x>5
  pthread_mutex_lock(&mutex);
  while (x<=5) {
    pthread_cond_wait(&cond, &mutex);
    cout << "Thread 1 awake...x>5\n";
    return NULL;
  }
  return NULL;
}
int main()
{
  pthread_mutex_init(&mutex, NULL);
  pthread_cond_init(&cond, NULL);
  pthread_t t1, t2;
  pthread_create(&t1, NULL, thread1, NULL);
  pthread_create(&t2, NULL, thread2, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return 0;
}