Creating semaphores using pthread primitives: pthread_mutex_lock, pthread_cond_wait and pthread_cond_signal

- recall from the last hour we looked at:
  - `pthread_mutex_lock`: allows a thread to lock a mutex
  - `pthread_mutex_unlock`: allows a thread to unlock a mutex
  - `pthread_cond_wait`: allows a thread to wait upon a condition
  - `pthread_cond_signal`: allows a thread to signal a condition

pthread_cond_wait

- notice this function has two parameters
  - `pthread_cond_wait (pthread_cond_t *condition, pthread_mutex_t *mutex)`

- `condition` is a condition variable which can be shared between a number of threads

- the thread will perform a wait upon the condition variable
  - it will unlock the mutex before performing the wait on the condition variable
  - the library will guarentee that the unlock/wait are atomic (another thread will not interrupt this sequence)

Why do we need condition variables, why not use a mutex?

- a mutex provides a lock/unlock
  - but what if we need a conditional lock?

Why do we need conditional lock data type?

- consider trying to implement a conditional lock using a mutex:

```
/* thread A.  */
...
if (some_condition) {
   some_condition = false;
   /* race hazard here.  */
   pthread_mutex_unlock (&mutex);
}
...
```

Why do we need conditional lock data type?

- consider

- ```
  /* thread B.  */
  ...
  if (! some_condition) {
     some_condition = true;
     /* race hazard here.  */
     pthread_mutex_lock (&mutex);
  }
  ...
  ```
  ```

Why do we need conditional lock data type?

- there is a potential race hazard, should the thread scheduler reschedule thread A when thread B is at the race hazard comment then the conditional has been set but the mutex is not locked

Why do we need conditional lock data type?

- the solution is to use a `pthread_cond_wait` and `pthread_cond_signal` pair

Why do we need conditional lock data type?

```
/* thread A.  */
...
pthread_mutex_lock (&mutex);
if (some_condition) {
   some_condition = false;
   /* no race hazard here now.  */
   pthread_cond_signal (&synchro);
}
pthread_mutex_unlock (&mutex);
...
```

Why do we need conditional lock data type?

- consider

- ```
  /* thread B.  */
  ...
  pthread_mutex_lock (&mutex);
  if (! some_condition) {
     some_condition = true;
     /* no race hazard here now.  */
     pthread_cond_wait (&synchro, &mutex);
  }
  pthread_mutex_unlock (&mutex);
  ...
  ```
  ```

Why do we need conditional lock data type?

- the function `pthread_cond_wait (&synchro, &mutex)` will unlock `mutex` before calling wait on `synchro`
  - these two operations are implemented in the function `pthread_cond_wait` as atomic

Implementing full thread semaphores using these primitives

- specification header file

Implementing full thread semaphores using these primitives

mysem.h

```
#include <cstdlib>
#include <iostream>
#include <unistd.h>
#include <pthread.h>
#include <sstream>

class mysem
{
public:
  mysem (void);  /* initialise semaphore with value, 0.  */
  ~mysem (void);
  mysem (const mysem &from);
  mysem (int value);  /* initialise semaphore with value.  */
  void wait (void);
  void signal (void);
private:
  pthread_mutex_t mutex;  /* mutex lock.  */
  pthread_cond_t counter; /* condition lock.  */
  bool waiting;  /* is another thread waiting on this semaphore?  */
  int sem_value; /* a count of outstanding signals.  */
};
```

Implementing full thread semaphores using these primitives

mysem.cpp

```
#include "mysem.h"

mysem::mysem ()
{
  printf ("constructor\n");
  counter = PTHREAD_COND_INITIALIZER;
  mutex = PTHREAD_MUTEX_INITIALIZER;
  waiting = false;
  sem_value = 0;
}

mysem::~mysem ()
{
  printf ("semaphore decons\n");
  pthread_mutex_destroy (&mutex);
  pthread_cond_destroy (&counter);
}
```

Implementing full thread semaphores using these primitives

mysem.cpp

```
mysem::mysem (int value)
{
  printf ("constructor (%d)\n", value);
  counter = PTHREAD_COND_INITIALIZER;
  mutex = PTHREAD_MUTEX_INITIALIZER;
  waiting = false;
  sem_value = value;
}
```

Implementing full thread semaphores using these primitives

mysem.cpp

```
void mysem::wait (void)
{
  printf ("wait\n");
  pthread_mutex_lock (&mutex);
  if (sem_value == 0)
    {
      waiting = true;
      pthread_cond_wait (&counter, &mutex);
      waiting = false;
    }
  else
    sem_value--;
  pthread_mutex_unlock (&mutex);
  printf ("end waite\n");
}
```

Implementing full thread semaphores using these primitives

mysem.cpp

```
void mysem::signal (void)
{
  printf ("signal\n");
  pthread_mutex_lock (&mutex);
  if (waiting)
    pthread_cond_signal (&counter);
  else
    sem_value++;
  pthread_mutex_unlock (&mutex);
  printf ("end signal\n");
}
```

wait

- inside the mutual exclusion section of wait

- **mysem::wait**

```
if (sem_value == 0)
  {
    waiting = true;
    pthread_cond_wait (&counter, &mutex);
    waiting = false;
  }
else
  sem_value--;
```

- notice that the implementation of wait does not decrement the semaphore below zero, but waits for the semaphore to be incremented
  - before it would increment the semaphore
  - thus both the `signal` and `wait` code can agree to optimise the increment and immediate decrement away!

wait

mysem::signal

```
if (waiting)
  pthread_cond_signal (&counter);
else
  sem_value++;
```

Revisit previous tutorial code with the new semaphore implementation

testthreads.cpp

```
#include <cstdlib>
#include <iostream>
#include <unistd.h>
#include <pthread.h>
#include <sstream>
#include "mysem.h"

using namespace std;
mysem mutex (1);
```

Revisit previous tutorial code with the new semaphore implementation

testthreads.cpp

```
void *SleepTime (void *timeToSleep)
{
  (*((int *) timeToSleep))++;
  sleep (0.1); //simulates race condition
  int duration = *((int*) timeToSleep);
  cout << "Sleeping for " << duration << endl;
  mutex.signal ();
  sleep (duration);
  cout << "Slept for " << duration << " seconds." << endl;
  return NULL;
}
```

Revisit previous tutorial code with the new semaphore implementation

testthreads.cpp

```
int main ()
{
  const int NUM_ITTS = 5;
  pthread_t thread[NUM_ITTS];   // we need to remember the thread id.
  mutex.wait ();
  mutex.signal ();
  int time_sleep = 0;
  cout << "spawning threads" << endl;
```

Revisit previous tutorial code with the new semaphore implementation

testthreads.cpp

```
  for (int i = 0; i< NUM_ITTS; i++)
    {
      cout << "time_sleep" << time_sleep << endl;
      mutex.wait ();
      int rc = pthread_create (&thread[i], NULL, SleepTime, &time_sleep);
      if (rc != 0)
     cout << "failure pthread_create" << endl;
    }
  cout << "waiting for all threads to complete" << endl;
  for (int i = 0; i< NUM_ITTS; i++)
    pthread_join (thread[i], NULL);
  return 0;
}
```

Tutorial

- copy the above code in the correct files and verify that the test code works
- implement the shared buffer producer/consumer algorithm using this semaphore data type
- implement the readers/writers algorithm using threads and this data type