Lab 7

Pthreads

HelloWord Program Using Pthreads

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
//thread function definition
void *slave(void *argument) {
   int tid = *((int *) argument);
   printf("Hello World! It's me, thread %d!\n", tid);
   return NULL;
}

int main(int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   int thread_args[NUM_THREADS];
   int i;

   for (i = 0; i < NUM_THREADS; i++) {                       // create threads
      thread_args[i] = i;
      printf("In main: creating thread %d\n", i);
      if ( pthread_create(&threads[i], NULL, slave, (void *) &thread_args[i] ) != 0)
              printf("Pthread_create fails");
   }

   for (i = 0; i < NUM_THREADS; i++) {                       // join threads
      if ( pthread_join(threads[i], NULL) != 0 )
              printf("Pthread_join fails");
      printf("In main: thread %d is complete\n", i);
   }
   printf("In main: All threads completed successfully\n");
   return 0;
}
```

To Compile

- gcc -lpthread pthread1.c

Or

- gcc pthread1.c -lpthread

Problem 1: Scalar Product Using Pthreads

- Write a scalable parallel program that takes, as a command line argument, the name of a file, Input1.txt, containing an integer N and 2 vectors of integers of size N. Your program should calculate the Scalar Product of the two vectors, using PThreads.
- **Sample input:** **Scalar Product:**

3 *s = u.v= 823*

11

20

7

42

10

23

Steps for the Solution

- Vectors A and B should be declared as global variables
- The total Sum is also declared as global variable
- Open the file for reading and read the size N
- Dynamically allocate vectors A and B
- Read the components of A and B from the file
- Write the function to be executed by every thread, which is to compute the product of one element in A with the corresponding element in B and update the global Sum variable
- Create N threads
- Join the threads

Race Condition

- A race condition will arise when multiple threads try to update the global memory location Sum
- A mutex should be used!

Pthread Mutex Functions

- In the dot product example, we ensure absence of races by employing a a **mutex** (MUTual EXclusion) resource. A mutex can be owned, i.e. "locked", by at most one thread at any given time.
- **int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *mutexattr);**
- **int pthread_mutex_lock(pthread_mutex_t *mutex); int pthread_mutex_unlock(pthread_mutex_t *mutex); int pthread_mutex_destroy(pthread_mutex_t *mutex);**
- **pthread_mutex_init(mutex, attr)** initialises **Code** with given attributes **attr** (usually **attr=NULL** for defaults).
- **pthread_mutex_lock(mutex)** blocks execution of calling thread until it obtains the lock over the mutex.
- **pthread_mutex_unlock(mutex)** releases the lock over the mutex. **pthread_mutex_destroy(mutex)** tears-down the mutex.

Pthread Mutex Function

- In our code, the mutex is also declared as a global variable:
  - pthread_mutex_t *mutex_scalarproduct;
- The mutex should be used by every thread to prevent race conditions while updating the Sum

Problem 2: Finite Difference

- *Description:*
- In a one-dimensional finite difference problem, we have a vector $X^{(0)}$ of size *N* and must compute $X^{(T)}$ where:
- $0 < i < N-1$ and $0 <= t < T$: $\mathbf{X_i^{(t+1)} = (X_{i-1}^{(t+1)} + 2X_i^{(t)} + X_{i+1}^{(t)}) / 4}$
- That is, we must repeatedly update each element of X, with no element being updated in step *t+1* until its neighbors have been updated in step *t*.
- *Your program:*
- Write a scalable parallel program that takes, as a command line argument, the name of a file, Input2.txt, containing a value T (number of steps), followed by integer N, and an array X of size N considered as the initial vector (at step *t*=0). Your program should update the vector $X^{(0)}$ until reaching $X^{(T)}$.

Steps for the solution

- Launch N threads. Every thread will update one element of X in every iteration.
- To ensure threads are synchronized in the same iteration, we employ a **barrier**

Barrier Operations

- **int pthread_barrier_init(pthread_barrier_t *barrier, pthread_barrierattr_t *restrict attr, unsigned count);**
- **int pthread_wait(pthread_barrier_t *barrier);**
  int pthread_barrier_destroy(pthread_barrier_t *barrier);
- **pthread_barrier_init(b,attr,count)** initialises **b** for **count** participants.
- **pthread_barrier_wait(b)** blocks the calling thread on barrier **b**, until all participants reach the barrier. The barrier is reset for re-use on return.
- **pthread_barrier_destroy(b)** tears-down **b**.

Condition Variables

- In many a cases a thread has to wait for a certain synchronisation condition to hold. Using a mutex we can repeatedly test for the condition, as illustrated above. This is a costly “busy-wait” scheme and may also generate high contention between threads, due to repeated lock acquisition and release.
- Ideally, we would like that the thread suspends execution and relinquishes the lock until the condition holds. This type of synchronisation is provided by **condition variables**, also known as **monitors**.

Condition Variables in Pthreads

- **int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);**
- **int pthread_cond_destroy(pthread_cond_t *cond);**
- **int pthread_cond_wait(pthread_cond_t *cond,**
- **pthread_mutex_t *mutex);**
- **int pthread_cond_signal(pthread_cond_t *cond);**
- **int pthread_cond_broadcast(pthread_cond_t *cond); int pthread_cond_destroy(pthread_cond_t *cond);**
- **pthread_cond_init** / **pthread_cond_destroy** are used for initialization / tear-down (as usual).
- **pthread_cond_wait(cond,mutex)**: waits on **cond**, releasing the lock on **mutex** while blocked, and re-acquiring it when unblocked.
- **pthread_cond_signal(cond)** unblocks **one** thread waiting on **cond**. **pthread_cond_broadcast(cond)** unblocks **all** threads waiting on
- **cond**.

Condition Variable Example Use

```
// Waiting thread
pthread_mutex_lock(&mutex);
while (! some_condition())
        pthread_cond_wait(&cond, &mutex);
pthread_mutex_unlock(&mutex)

// Signalling thread
pthread_mutex_lock(&mutex);
if (some_condition() )
        pthread_cond_signal(&cond); // or broadcast
pthread_mutex_unlock(&mutex);
```