PThreads – An Introduction

CS F422 Parallel Computing Tutorial

PThreads

- POSIX Thread
- POSIX – A standard for Unix like OS
- Library linked with C programs

Ref [1]

Things to remember

- Threads within a process share same address space
  - Data, open files, current working directory …
- Each thread has a unique
  - Thread id, registers, stack …

Ref [2]

Execution

- #include <pthread.h>
- gcc –g –Wall –o outputfilename filename.c –lpthread

Ref [1]

E.g. 1

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

void *print_message_function( void *ptr );

main(){
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;

    /* Create independent threads each of which will execute function */

     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

     /* Wait till threads are complete before main continues. Unless we  */
     /* wait we run the risk of executing an exit which will terminate   */
     /* the process and all threads before the threads have completed.   */

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}

void *print_message_function( void *ptr ){
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

Ref [2]

Starting Threads

- int pthread_create(

  pthread_t *thread ,

  const pthread_attr_t *attr,

  void *(*start_routine)(void*),

  void *arg_p);

  - thread – return thread id
  - attr – set to NULL. Attributes include detached state, scheduling policy, scheduling parameter, inheritsched attribute, scope …
  - Void *(*start_routine) – pointer to function to be threaded with single pointer to void argument
  - *arg – pointer to argument of function

Ref [2]

Stopping Threads

- int pthread_join(
  pthread_t thread,
  void** ret_val_p);

- Output E.g. 1

  Thread 1
  Thread 2
  Thread 1 returns: 0
  Thread 2 returns: 0

Ref [2]

Killing threads

- void pthread_exit(void *retval)

- Kills the thread
- Return value of thread stored in retval, which is not of local scope

Ref [2]

Synchronization

- Mutexes
- Joins
- Condition variables

Ref [2]

E.g. 2 Mutexes

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

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main(){
   int rc1, rc2;
   pthread_t thread1, thread2;

   /* Create independent threads each of which will execute functionC */

   if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
      printf("Thread creation failed: %d\n", rc1);

   if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
      printf("Thread creation failed: %d\n", rc2);

   /* Wait till threads are complete before main continues. Unless we  */
   /* wait we run the risk of executing an exit which will terminate   */
   /* the process and all threads before the threads have completed.   */

   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   exit(0);
}

void *functionC(){
   pthread_mutex_lock( &mutex1 );
   counter++;
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );
}
```

Ref [2]

Mutexes

- Prevent data inconsistencies due to race conditions (same memory changed by threads, but output depends on order of operations)
- Serialize shared resources
- In E.g. 2, if locking was not applied, the two threads may have written same result as 1, but with locking correct result is obtained, i.e., 2.

- Output E.g. 2

  Counter value: 1

  Counter value: 2

Ref [2]

E.g. 3 Joins

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

#define NTHREADS 10
void *thread_function(void *);
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main(){
   pthread_t thread_id[NTHREADS];
   int i, j;

   for(i=0; i < NTHREADS; i++)
      pthread_create( &thread_id[i], NULL, thread_function, NULL );

   for(j=0; j < NTHREADS; j++)
      pthread_join( thread_id[j], NULL);

   /* Now that all threads are complete I can print the final result.     */
   /* Without the join I could be printing a value before all the threads */
   /* have been completed.                                                */

   printf("Final counter value: %d\n", counter);
}

void *thread_function(void *dummyPtr){
   printf("Thread number %ld\n", pthread_self());
   pthread_mutex_lock( &mutex1 );
   counter++;
   pthread_mutex_unlock( &mutex1 );
}
```

Ref [2]

Joins

- Wait for a thread to finish
- Say master thread waiting for all threads to complete and then terminate
- In E.g. 3, main thread waits for all to complete and then prints the counter value
- Output E.g. 3

  Thread number 1026

  Thread number 2051 …

  Final counter value: 10

Ref [2]

E.g. 4 Condition Variables

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

pthread_mutex_t count_mutex     = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t condition_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_cond  = PTHREAD_COND_INITIALIZER;

void *functionCount1();
void *functionCount2();
int  count = 0;
#define COUNT_DONE  10
#define COUNT_HALT1  3
#define COUNT_HALT2  6

main(){
   pthread_t thread1, thread2;

   pthread_create( &thread1, NULL, &functionCount1, NULL);
   pthread_create( &thread2, NULL, &functionCount2, NULL);
   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   exit(0);
}

void *functionCount1(){
   for(;;){
      pthread_mutex_lock( &condition_mutex );
      while( count >= COUNT_HALT1 && count <= COUNT_HALT2 )
         pthread_cond_wait( &condition_cond, &condition_mutex );
      pthread_mutex_unlock( &condition_mutex );

      pthread_mutex_lock( &count_mutex );
      count++;
      printf("Counter value functionCount1: %d\n",count);
      pthread_mutex_unlock( &count_mutex );

      if(count >= COUNT_DONE) return(NULL);
   }
}

void *functionCount2(){
    for(;;){
      pthread_mutex_lock( &condition_mutex );
      if( count < COUNT_HALT1 || count > COUNT_HALT2 )
         pthread_cond_signal( &condition_cond );
      pthread_mutex_unlock( &condition_mutex );

      pthread_mutex_lock( &count_mutex );
      count++;
      printf("Counter value functionCount2: %d\n",count);
      pthread_mutex_unlock( &count_mutex );

      if(count >= COUNT_DONE) return(NULL);
    }
}
```

Ref [2]

Condition Variables

- Variable of type pthread_cond_t
- Used with appropriate functions for waiting and later, process continuation
- Condition variable associated with a mutex to avoid race condition

- Functions used in conjunction with condition variable
  - Creating/destroying
    - pthread_cond_init
    - pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
    - pthread_cond_destroy

Ref [2]

Condition Variables

- Waiting on condition
  - pthread_cond_wait
  - pthread_cond_timedwait – limit on blocking time
- Waking threads based on condition
  - pthread_cond_signal
  - pthread_cond_broadcast – wake up all threads blocked by the specified condition variable

- In E.g. 4, functionCount1() halted while count was between COUNT_HALT1 and COUNT_HALT2. functionCount2() increments count during this time. Everything else is random

Ref [2]

Condition Variables

- Output E.g. 4

```
Counter value functionCount1: 1
Counter value functionCount1: 2
Counter value functionCount1: 3
Counter value functionCount2: 4
Counter value functionCount2: 5
Counter value functionCount2: 6
Counter value functionCount2: 7
Counter value functionCount1: 8
Counter value functionCount1: 9
Counter value functionCount1: 10
Counter value functionCount2: 11
```

Ref [2]

Reference

[1] An Introduction to Parallel Programming by Peter S. Pacheco

[2] POSIX thread (pthread) libraries