BSc (Hons) in Information Technology

Lab Exercise 5

SE2042 Operating Systems

Semester 1

Learning Objectives: This practical will help you to learn about UNIX threads. (Pthreads).

Basic Concepts

A thread is a stream of instructions that can be scheduled as an independent unit. It is easier to understand a thread within the context of a process. A process is created by an operating system; a process contains information about resources, such as process id, file descriptors, etc.; in addition it contains information pertaining to the execution state, such as program counter and stack. The concept of a thread requires that we make a separation between these two kinds of information in a process:

- Resources - these are available to the entire process e.g., program instructions, global data, working directory, etc..
- Schedulable entities - these would include program counters and stacks.

A thread is an entity within a process, which consists of the schedulable part of the process. Each process can have many threads, which share the resources within a process (address space, for example). As compared to the cost of creating a process, a thread can be created with much less expense, because there is much less intervention on the part of the operating system.

Pthreads

An unambiguously defined interface is essential if threads are to be useful to programmers that wish to take advantage of the capabilities provided by threads. While hardware vendors each have their own implementation of threads which differ from one another, the emerging standard on Unix systems is specified by IEEE POSIX 1003.1c -1995. A threads implementation that follows this standard is referred to as Pthreads. Most hardware vendors now tend to offer Pthreads. In addition to their proprietary API's. A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization .API specifies behavior of the thread library, implementation is up to development of the library. Implementation from system to system varies and each may have different limitations and deficiencies.

The three major classes of Pthread calls are as following:

1. The first class of functions work directly on threads - creating, detaching, joining, etc. These are adjoined by thread attribute functions that set or modify attributes of the threads, (joinable, scheduling etc.)
2. The second class of functions work on mutexes - creating, destroying, locking and unlocking. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
3. The third class work on condition variables - creating and destroying, waiting and signaling. As before these are accompanied by condition variable attribute functions that modify or set attributes of condition variables, (e.g. shared or non-shared).

Simple Explanation of a Mutex

Mutex is a shortened form of the words "mutual exclusion". The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful.

No other thread can own that mutex until the owning thread unlocks that mutex. A typical sequence in the use of a mutex is as follows:

- Create a mutex
- 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

Very often the action performed by a thread owning a mutex is the updating of global variables. This is a safe way to ensure that when several threads update the same variable.

The final value is the same as what it would be if only one thread performed the update. The variables being updated belong to a "critical section".

Using Pthread Library

Programs that use Pthreads need to include pthread.h. You will also need to link the pthread library into your program by adding a "-lpthread" to your compiler command line.

pthread_self - Returns thread identifier for the calling thread

```
#include <pthread.h>
        pthread_t pthread_self( void );
```

pthread_create - Creates a new thread of control

```
#include <pthread.h>
int pthread_create( pthread_t * thread, pthread_attr_t * attr,
                                        void * (*start_routine)(void *), void * arg );
```

Returns 0 to indicate success.
NonZero return is error code.

Parameters:
thread – will contain thread id of new thread. attr -Specifies the attributes of thread to be created (NULL assumes defaults) start_routine -function to used as start of new thread arg -Arguments to pass to new thread start routine

pthread_exit - Terminate the calling thread

```
#include <pthread.h>
void pthread_exit( void *retval  );
```

retval -specifies the threads return status. This will be made available to any pthread_join on the thread.

pthread_join- Waits for a thread to terminate

```
#include <pthread.h>
int pthread_join( pthread_t thread, void **value_ptr );
```

thread – thread id to wait for

value_ptr – value given to pthread_exit on the terminating thread

Returns 0 to indicate success. NonZero return is error code Multiple simultaneous calls for same thread are not allowed.

pthread_attr_init - Initializes the thread attribute object to the default values

```
#include <pthread.h>
int pthread_attr_init( pthread_attr_t *attr  );
```

Returns 0 to indicate success and NonZero return an error code.

Pthread Example 01:

Create a C program file call example1.c and type the following program. Execute the code using appropriate libraries.

```
#include <pthread.h>
#define NUM_THREADS 25
int thread_routine (int  x)
  {
       printf ( "I'm Thread %2d my TID is %u\n", x, pthread_self () );
       pthread_exit(0);
  }

int main ()  {
          pthread_attr_t thread_attr;
          pthread_t tids[NUM_THREADS];
          int x;
          pthread_attr_init (&thread_attr);
                    for (x = 0; x < NUM_THREADS; x++)
                    {
                    pthread_create (&tids[x], &thread_attr, (void *)thread_routine,
                    (void *) x);
                    }
          printf ("Waiting for threads to  finish\n");
          for (x = 0; x < NUM_THREADS; x++)
          {
          pthread_join (tids[x], NULL);
          }
     printf ("All treads are now finished\n");
}
```

Pthread example 02

Create a C program file call example2.c and type the following program. Understand the function of each statement of the program.

```
#include <pthread.h>
#include <stdio.h>
#define BSIZE 4
#define NUMITEMS 30

        typedef struct
        {
          char buf[BSIZE];
          int occupied;
          int nextin, nextout;
          pthread_mutex_t mutex;
          pthread_cond_t more;
          pthread_cond_t less;
        } buffer_t;

buffer_t buffer;

void * producer(void *);
void * consumer(void *);

#define NUM_THREADS 2
pthread_t tid[NUM_THREADS];

main( int argc, char *argv[] )
{
  int i;

  pthread_cond_init(&(buffer.more), NULL);
  pthread_cond_init(&(buffer.less), NULL);

  pthread_create(&tid[1],   NULL,    consumer, NULL);
  pthread_create(&tid[0], NULL, producer, NULL); for
  ( i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);

    printf("\n main() reporting that all %d threads have terminated\n", i);
}
```

```
void * producer(void *  parm)
{
  char item[NUMITEMS]="IT'S A SMALL WORLD, AFTER ALL.";
  int i;

  printf("producer started.\n");

  for(i=0;i<NUMITEMS;i++)
   {
     if (item[i] == '\0') break;

     pthread_mutex_lock(&(buffer.mutex));

     if (buffer.occupied >= BSIZE)
        printf("producer waiting.\n");
     while (buffer.occupied >= BSIZE)
            pthread_cond_wait(&(buffer.less),& buffer.mutex) );
     printf("producer executing.\n");

     buffer.buf[buffer.nextin++] = item[i];
     buffer.nextin %= BSIZE;
     buffer.occupied++;


     pthread_cond_signal(&(buffer.more));
     pthread_mutex_unlock(&(buffer.mutex));
   }

  printf("producer exiting.\n");
  pthread_exit(0);
}
```

```
void * consumer(void * parm)
{
  char item;
  int i;

  printf("consumer started.\n");

  for(i=0;i<NUMITEMS;i++){

  pthread_mutex_lock(&(buffer.mutex) );

  if (buffer.occupied <= 0)
     printf("consumer is waiting.\n");
  while(buffer.occupied <= 0)
     pthread_cond_wait(&(buffer.more),&(buffer.mutex) );
  printf("consumer executing.\n");

  item = buffer.buf[buffer.nextout++];
  printf("%c\n",item);
  buffer.nextout %= BSIZE;
  buffer.occupied--;


  pthread_cond_signal(&(buffer.less));
  pthread_mutex_unlock(&(buffer.mutex));
  }
  printf("consumer exiting.\n");
  pthread_exit(0);
}
```