UFV High Performance Computing
Posix Threads Lab.

THREADS PROGRAMMING in UNIX/LINUX

Thread programming in Linux

The POSIX standard thread library provides an API for C/C++ that allows the creation of threads as parts of a process, the concurrent programming using threads needs less resources than using process (the context switch needs less resources in the thread case than the fork() system call) . Remember that all threads share the address space of the process.

In this lab you will become acquainted with threading basics. The threading system you'll be using is the standard pthreads package. pthreads is standard on UNIX systems, and because C doesn't contain any intrinsic threading primitives, pthreads is just a library of functions that interface to OS syscalls for creating and destroying threads. Because pthreads is a separate library, you'll have to compile you're code a little differently. Essentially, you just have to add the -lpthreads flag to your gcc invocation. For example, if your source file was named threadprog.c, you would compile it so:

$ gcc -lpthread -o myprog myprog.c

Additionally, you will need to include the pthreads header file pthread.h (unsurprisingly).

Note about testing your code. Testing multi-threaded code is a very different proposition from testing single-threaded code. Many multi-threaded errors result from specific interleavings, and as such, are very hard to duplicate. Additionally, it can be helpful to insert print statements throughout the code.

BASIC CONCEPTS for THREADS

The Basic operations for threads programming include the creation, ended, sincronize, scheduling and data Management with threads. The threads inside a process can share .

- global variables
- file descriptors
- signals
- the current working directory

Each thread has a unique:

- thread id number (tid)
- Registers and stack pointer
- Stack for local variables
- Priority

All pthread library functions starting with the name "`pthread_`" returns 0 if there is no problem.

THREAD CREATION and TERMINATION

In pthreads, new threads are created by calling the function pthread create. The actual definition of pthread create is quite verbose:

```
int pthread_create(pthread_t            *thread,
                   const pthread_attr_t *attr,
                   void                 *(*start_routine)(void *),
                   void                 *arg);
```

The function phread_create takes the following four arguments:

• **Thread:** a pointer to a buffer of type pthread_t. 'pthread_create' will write a value that identifies the newly created thread to that buffer. This handle can be used in all subsequent calls to refer to this specific thread.
• **Thread attributes:** a pointer to a structure which is referred to as a thread attribute object. A thread attribute object can specify various characteristics for the new thread. In our example, we pass a value of NULL, indicating that we accept the default characteristics for the new thread. The only really interesting thing, for us at least, is that the pthread attr_t struct tells the system how much memory to allocate for the thread's stack. By default, this is usually 512K.
• **Routine:** a pointer to the routine at which the new thread will start its execution.
• **Argument of the routine:** pointer to a parameter which is passed to the routine.

pthread_create returns a value that indicates whether it has succeeded or failed. A zero value represents success, and a nonzero value indicates and identifies an error.

Pthread Termination

A thread is terminated by calling the function pthread_exit or when it reaches the end of its initial routine.

The function pthread_exit looks as follows:

```
void pthread_exit(void *value_ptr);
```

The argument value_ptr is a pointer to a termination value which the thread wants to share with another thread (see thread synchronisation).

Pthread Synchronisation

In our program, we are using the following function call to synchronise the threads:

```
int pthread_join(pthread_t thread, void **value_ptr);
```

pthread_join is a thread function which is similar to waitpid for processes. The pthread_join call suspends its caller until the thread provided in its first argument exits. The second argument, value_ptr, is a pointer in which the value passed to pthread_exit() by the terminating thread is stored.

Pthread Scheduling

We can order the events in our program by imposing some type of scheduling policy onto them. This can be done in the thread attribute object. However, for our simple program, we are using the function sched_yield:

```
sched_yield();
```

The sched_yield() function forces the running thread to relinquish the processor until it again becomes the head of its thread list. It takes no arguments.

Another way to schedule threads is by using the usleep function. This allows us to implicitly schedule threads by putting the current thread to sleep:

```
Int usleep(usecond_t microseconds);
```

Example1 : pthread1.c (compile with $ gcc -lpthread pthread1.c –o pthread1)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void escribir_mensaje( void *ptr ) {
        char  *mensaje;
        mensaje = (char *) ptr;
        printf("%s \n",mensaje);
}

main() {
        pthread_t tid1, tid2;
        char*mensaje1 = "Hilo 1";
        char*mensaje2 = "Hilo 2";

        /* Crea 2 hilos independientes y ambos ejecutan la funcion */
        /* escribir_mensaje*/

        pthread_create(&tid1, NULL, (void*)&escribir_mensaje, (void*) mensaje1);
        pthread_create(&tid2, NULL, (void*)&escribir_mensaje, (void*) mensaje2);

        /* Esperar por la terminacion de los hilos, antes de que el */
        /* hilo principal continue. Si no se espera, se corre el */
        /* riesgo de que el proceso termine (y por tanto todos sus */
        /* hilos) antes de que los hilos hayan terminado realmente */

        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);

        exit(0);
}
```

Example 2 ping-pong.c

Consider other simple example with Pthreads (ping-pong.c). The next program creates two threads which both execute the function ***display*** with their own argument, “ping” and “PONG” respectively.

Compiling Pthreads with gcc requires the -pthread flag:

gcc -lpthread ping-pong.c -o ping-pong

To run this sample program, you have to set the value N when you call your program, for example: ./ping-pong 1000

```
#include <stdio.h>
#include <stdlib.h> // necessary for atoi
#include <pthread.h>
int N; // global variable


void *display(void *arg)
{
        int n=0;
        while(n < N)
        {
                n++;
                printf("%s ", (char *)arg); // casting of void pointer
                usleep(1000); // yield the processor to another thread
        }
        printf("\nBye bye from %s thread\n", (char *)arg);
}

int main(int argc, char *argv[]) {
pthread_t thread0, thread1;
char *message0="ping", *message1="PONG";
N = atoi(argv[1]);
pthread_create(&thread0, NULL, display, (void *)message0);
pthread_create(&thread1, NULL, display, (void *)message1);
pthread_join(thread0, NULL); // wait on thread0 exit
pthread_join(thread1, NULL); // wait on thread1 exit
printf("Bye bye from main thread\n");
return 0;
}
```

Passing Arguments to Threads

The pthread_create() routine permits the programmer to pass one argument to the thread start routine. For cases where multiple arguments must be passed, this limitation is easily overcome by creating

- a structure which contains all of the arguments, and then passing a pointer to that structure in the pthread_create() routine.
- All arguments must be passed by reference and cast to (void *).

This example shows how to setup/pass multiple arguments via a structure. Each thread receives a unique instance of the structure.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8
char *messages[NUM_THREADS];


struct thread_data
{
```

```
   int thread_id;
   int  sum;
   char *message;
};
struct thread_data thread_data_array[NUM_THREADS];


void *PrintHello(void *threadarg)
{
   int taskid, sum;
   char *hello_msg;
   struct thread_data *my_data;
   sleep(1);
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   printf("Thread %d: %s  Sum=%d\n", taskid, hello_msg, sum);
   pthread_exit(NULL);
}
int main(int argc, char *argv[])
{
pthread_t threads[NUM_THREADS];
int *taskids[NUM_THREADS];
int rc, t, sum=0;
messages[0] = "English: Hello World!";
messages[1] = "French: Bonjour, le monde!";
messages[2] = "Spanish: Hola al mundo";
messages[3] = "Klingon: Nuq neH!";
messages[4] = "German: Guten Tag, Welt!";
messages[5] = "Russian: Zdravstvytye, mir!";
messages[6] = "Japan: Sekai e konnichiwa!";
messages[7] = "Latin: Orbis, te saluto!";
for(t=0;t<NUM_THREADS;t++) {
  sum = sum + t;
  thread_data_array[t].thread_id = t;
  thread_data_array[t].sum = sum;
  thread_data_array[t].message = messages[t];
  printf("Creating thread %d\n", t);
  rc = pthread_create(&threads[t], NULL, PrintHello, (void *)
       &thread_data_array[t]);
  if (rc) {
    printf("ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
    }
  }
pthread_exit(NULL);
}
```

THREAD SYNCRONIZATION

When múltiple threads are running they will invariably need to communicate with each other in order synchronize their execution. One main benefit of using threads is the ease of using synchronization facilities.

Why we need Synchronization and how to achieve it?

Suppose the multiple threads share the common address space (thru a common variable), and then there is a problem.

THREAD A	**THREAD B**
x = commonvariable ;	y = commonvariable ;
x++ ;	y— ;
common_variable = x ;	common_variable = y ;

If threads execute this code independently it will lead to garbage. The access to the common variable by both of them simultaneously is prevented by having a lock, performing the action and then releasing the lock.

Mutex Variables:

Overview

Mutex is a shortened form of the words "mutual exclusión".

Mutex variables are one of the primary means of implementing thread synchronization.

A mutex variable acts like a "lock" protecting access to a shared data resource. The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex variable 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. Threads must "take turns" accessing protected data.

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". A typical sequence in the use of a mutex is as follows:

Create and initialize a mutex variable 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.
When several threads compete for a mutex, the losers block at that cali an unblocking call is available with "trylock" instead of the "lock" call.

Creating / Destroying Mutexes :

pthread_mutex_init (pthread_mutex_t mutex, pthread_mutexattr_t attr)
pthread_mutex_destroy (pthread_mutex_t mutex)
pthreadmutexattrinit (pthread_mutexattr_t attr)
pthread_mutexattr_destroy (pthread_mutexattr_t attr)
pthread_mutex_init( ) creates and initializes a new mutex mutex object, and sets its attributes according to the mutex attributes object, attr. The mutex is initially unlocked.
Mutex variables must be of type pthread_mutex_t
The attr object is used to establish properties for the mutex object, and must be of type pthread_mutexattr_t if used (may be specified as NULL to accept defaults).
If implemented, the pthread_mutexattr_init( ) and pthread_mutexattr_destroy( ) routines are used to create and destroy mutex attribute objects respectively.
pthread_mutex_destroy( ) should be used to free a mutex object which is no longer needed.

Locking / Unlocking Mutexes :
pthread_mutex_lock (pthread_mutex_t mutex)
pthread_mutex_trylock (pthread_mutex_t mutex)
pthread_mutex_unlock (pthread_mutex_t mutex)

The pthread_mutex_lock( ) routine is used by a thread to acquire a lock on the specified mutex variable. If the mutex is already locked by another thread, the call will block the calling thread until the mutex is unlocked.
pthread_mutex___trylock( ) will attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately. This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.
Mutex contention: when more than one thread is waiting for a locked mutex, which thread will be granted the lock first after it is released? Unless thread priority scheduling (not covered) is used, the assignment will be left to the native system scheduler and may appear to be more or less random.
pthread_mutex_unlock( ) will unlock a mutex if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data.
An error will be returned if:
If the mutex was already unlocked
If the mutex is owned by another thread

Example of using MUTEX

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

void*funcionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  ct = 0;

main() {
int rc1,rc2;
pthread_t tid1, tid2;
/* Create two threads executing funcionC */
if( (rc1=pthread_create( &tid1, NULL, &funcionC, NULL)) ) {
printf("Error creating thread11: %d\n", rc1);
}
if( (rc2=pthread_create( &tid2, NULL, &funcionC, NULL)) ) {
printf("Error creating thread2 2: %d\n", rc2);
}

/* Wait until thread terminate*/
pthread_join(tid1, NULL);
pthread_join(tid2, NULL);
exit(0);
}
```

```
void*funcionC()
{
 pthread_mutex_lock(&mutex1);
ct++;
printf("Counter Value: %d\n",ct);
pthread_mutex_unlock(&mutex1);
}
```

ASSIGNMENT

1.- Write a program where the main thread ask for a number and two strings. Creeate 2 tasks (threads) that prints in the screen the task ID and the previous strings a number of times equal to the number introduced un the main . The main task must waiting to the termination of the and the repeat the whole operation.

Assuming the name of the program is "repeat_string" the behavior would be similar to this

```
$ ./repeat_string
Input a number : 2
Input first string: What happen?
Input second string: Nothing happen
thread (1026): 1 What happen?
thread (1026): 2 What happen?
```

thread (2051): 1 Nothing happen
thread (2051): 2 Nothing happen
Input a number

:
$

2.- Write a program for adition of two 3x3 matrix. Ask for the matrix elements in main and create 3 tasks each add one row with other. then in main print the result

$$(A + B)[i, j] = A[i, j] + B[i, j]$$

$$\begin{bmatrix} 1 & 3 & 2 \\ 1 & 0 & 0 \\ 1 & 2 & 2 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 5 \\ 7 & 5 & 0 \\ 2 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1+1 & 3+0 & 2+5 \\ 1+7 & 0+5 & 0+0 \\ 1+2 & 2+1 & 2+1 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 7 \\ 8 & 5 & 0 \\ 3 & 3 & 3 \end{bmatrix}$$

3.- Write a thraded program for find a the number of occurrences of a determined mumber in a vector, read a vector of 20 elements and divide it in four parts, then each task must find the occurrences of the number in his part. Print the results in the main thread. Suppose we are finding the number of occurrences of number 6. Use mutex for mutual exclusion when tasks are using the vector.