OPERATING SYSTEMS LAB

LAB MANUAL # 12

Instructor: Muhammad Abdullah Orakzai

Semester Fall 2021

Course Code: CL2006

Fast National University of Computer and Emerging Sciences, Peshawar

Department of Computer Science

Contents

POSIX Threads

POSIX Threads

POSIX thread library is a standard thread library for C/C++. Using the POSIX thread library, we can create a new concurrent process execution flow such that our program can then handle multiple execution paths. As a result, our program would appear to do many things at the same time.

What is a Thread?

A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called *lightweight processes*.

What are the differences between process and thread?

Threads are not independent of one other like processes as a result threads shares with other threads their **code section**, **data section** and OS resources like open files and signals. But, like process, a thread has its own program counter (PC), a register set, and a stack space.

Why Multithreading?

Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.

Threads operate faster than processes due to following reasons:

1) Thread creation is much faster.
2) Context switching between threads is much faster.
3) Threads can be terminated easily
4) Communication between threads is faster.

Thread Basics

A thread can be defined as a separate stream of instructions within a process. From developer point of view, a thread is simply a function or procedure, that has its own existence and runs independently from the program's main() procedure/function.

To visualize, imagine a program C program with a number of functions. This program can be run by entering the executable **a.out** on the command interface. Then imagine each function being scheduled by the operating system. This would be a **multi-threaded** program.

Unlike child processes, a thread doesn't know which thread is responsible for its creation, neither does it maintain a list of current active threads inside a process. Within the same process, a thread may share the process instructions (text section), global data (data section) and open files (file descriptors). Each thread has a unique **thread ID**, **set of registers** and **stack**. These will be individual to a thread itself.

Thread Creation

A thread is created using the pthread_create() call. Just like process creation using fork(), a pthread_create() call will return certain integer integer numbers upon successfull completion of the call.

In a **Unix/Linux operating system**, the **C/C++ languages** provide the POSIX thread (pthread) standard API (Application program Interface) for all thread related functions. It allows us to create multiple threads for concurrent process flow. It is most effective on multiprocessor or multi-core systems where threads can be implemented on a kernel-level for achieving the speed of execution. Gains can also be found in uni-processor systems by exploiting the latency in IO or other system functions that may halt a process.

We must include the **pthread.h** header file at the beginning of the script to use all the functions of the pthreads library. To execute the **C** file, we have to use the -pthread or -lpthread in the command line while compiling the file.

```
cc -pthread file.c or

cc -lpthread file.c
```

The **functions** defined in the **pthreads library** include:

1. **pthread_create:** used to create a new thread
   Syntax:

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

Parameters:

thread: pointer to an unsigned integer value that returns the thread id of the thread created.

attr: pointer to a structure that is used to define thread attributes like detached state, scheduling policy, stack address, etc. Set to **NULL** for default thread attributes.

start_routine: pointer to a subroutine that is executed by the thread. The return type and parameter type of the subroutine must be of type void *. The function has a single attribute but if multiple values need to be passed to the function, a **struct** must be used.

arg: pointer to void that contains the arguments to the function defined in the earlier argument.

2. **pthread_exit***:* used to terminate a thread
   Syntax:

```
void pthread_exit(void *retval);
```

Parameters: This method accepts a mandatory parameter **retval** which is the pointer to an integer that stores the return status of the thread terminated. The scope of this variable must be global so that any thread waiting to join this thread may read the return status.

3. **pthread_join:** used to wait for the termination of a thread.
   Syntax:

```
int pthread_join(pthread_t th,
                 void **thread_return);
```

Parameter: This method accepts following parameters:

th: thread id of the thread for which the current thread waits.

thread_return: pointer to the location where the exit status of the thread mentioned in th is stored.

4. **pthread_self:** used to get the thread id of the current thread.
   Syntax:
   ```
   pthread_t pthread_self(void);
   ```

5. **pthread_equal:** compares whether two threads are the same or not. If the two threads are equal, the function returns a non-zero value otherwise zero.
   Syntax:
   ```
   int pthread_equal(pthread_t t1,

                     pthread_t t2);
   ```
   Parameters: This method accepts following parameters:
   a. t1: the thread id of the first thread
   b. t2: the thread id of the second thread

6. **pthread_cancel:** used to send a cancellation request to a thread
   Syntax:
   ```
   int pthread_cancel(pthread_t thread);
   ```
   Parameter: This method accepts a mandatory parameter **thread** which is the thread id of the thread to which cancel request is sent.

7. **pthread_detach:** used to detach a thread. A detached thread does not require a thread to join on terminating. The resources of the thread are automatically released after terminating if the thread is detached.
   Syntax:
   ```
   int pthread_detach(pthread_t thread);
   ```
   Parameter: This method accepts a mandatory parameter **thread** which is the thread id of the thread that must be detached.

Example # 01:
A simple C program to demonstrate use of pthread basic functions

Please note that the below program may compile only with C compilers with pthread library.

thread.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> //Header file for sleep(). man 3 sleep for details.
#include <pthread.h>

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{
	sleep(1);
	printf("Printing Hello World from Thread \n");
	return NULL;
}

int main()
{
	pthread_t thread_id;
	printf("Before Thread\n");
	pthread_create(&thread_id, NULL, myThreadFun, NULL);
	pthread_join(thread_id, NULL);
	printf("After Thread\n");
	exit(0);
}
```

In main() we declare a variable called thread_id, which is of type pthread_t, which is an integer used to identify the thread in the system. After declaring thread_id, we call pthread_create() function to create a thread.

pthread_create() takes 4 arguments.

The first argument is a pointer to thread_id which is set by this function. The second argument specifies attributes. If the value is NULL, then default attributes shall be used.

The third argument is name of function to be executed for the thread to be created.

The fourth argument is used to pass arguments to the function, myThreadFun.

The pthread_join() function for threads is the equivalent of wait() for processes. A call to pthread_join blocks the calling thread until the thread with identifier equal to the first argument terminates.

How to Compile the above program:

To compile a multithreaded program using gcc, we need to link it with the pthreads library. Following is the command used to compile the program.

gcc thread.c -lpthread ----> for compilation

./a.out ----> for execution

OR

gcc -o thread thread.c -lpthread ----→ for compilation

./thread -----→ for execution

Example # 02:

Study the format of the code below especially the usage of the bold text.

thread1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr );
int main()
```

```
{
    pthread_t thread1, thread2;
    //In the following code we are assigning the address of
    //the strings to the pointer *message1 and *message2 .
    char message[15]="Thread 1";
    char *message1 = message;
    //char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int return_value1, return_value2;
   /* Create independent threads each of which will execute function */
    return_value1 = pthread_create( &thread1, NULL, print_message_function, message1);
   return_value2 = 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",return_value1);
    printf("Thread 2 returns: %d\n",return_value2);
    exit(0);
    return 0;
}
void *print_message_function( void *ptr)
{
    char *message;
    message = (char *) ptr;    // type casting of pointer to char
    printf("%s \n", message);
}
```

Compile the above code. Compilation of multithreaded programs are differently from the normal method. For threads, the -lpthread argument is provided as an addition.

When this program runs, there will be a total of 3 threads running _in_ this process; main, thread1, and thread2. Three additional kernel level threads will be required for servicing these three user-level threads. (Remember, Linux uses 1:1 thread model).

When the thread is created using pthread_create(), the main thread proceeds executing with the remainder of instructions. Meanwhile, the newly created thread will complete executing as well. If in case the main() thread finishes executing before any of the other threads, the process will exit. Any thread which had not finished will be interrupted before its termination. To avoid this, the pthread_join() call can be used. With this, the main() thread will wait for successful completion of any thread specifed in the join call.

Actually, pthread_join() is the opposite of pthread_create(). Pthread_create() will split our single threaded process into two-threaded process. Pthread_join() will join back the two threaded process into a single threaded process.

Read the manual pages and find out the answers to the following:

Q1: What is the pthread_create() and the pthread_join() calls doing?

Q2: In the pthread_create() call, what are the 4 parameters?

Q3: In pthread_create() call, the 4th parameter is used for passing a pointer to argument of a function. What will we need to do if we want to pass multiple arguments to that function?

Q4: In the pthread_join() call, what are the 2 paramters?

Q5: What is the purpose of the return_value1 and return_value2 variables? Find out the contents of both these variables using a printf function. What do they contain?

Passing Multiple Arguments to Thread

Multiple arguments can be passed to a thread through declaring a structure. For example,

```
typedef struct str_thdata
{
   int x;
   int y;
   int z;

} thdata;
```

Would be our thread_data structure containing members x, y, and z.

```
thdata Omar;     /* structs to be passed to threads */
```

Will create an instance of thread_data structure. Its name is given as **Omar**.

```
void print_message_function ( void *ptr )
{
   thdata *my_data;
   my_data = (thdata *) ptr;  /* type cast to a pointer to thdata */
   /* do the work */
   printf("X: %d, Y: %d, Z: %d\n", my_data->x, my_data->y, my_data->z);
   pthread_exit(0);           /* exit */
} /* print_message_function ( void *ptr ) */
```

Is our thread function. Here, we have declared a pointer called my_data and assign it location of argument passed to thread as input. Since this is a pointer to a structure, we will be using the arrow operator (->) instead of the dot (.) operator to access its members.

```
int main()
{
   pthread_t thread1;  /* thread variables */
   thdata Omar;           /* structs to be passed to threads */
```

```
   Omar.x=1;
   Omar.y=2;
   Omar.z=3;
   pthread_create (&thread1, NULL, (void *) &print_message_function, (void *) &Omar);
   pthread_join(thread1, NULL);
   exit(0);
}
```

Here, we assign the members x, y, and z some values. Instead of sending these individual members over to the thread, we send the address of the instance Omar.

Examples

struct_thread.c

```
/* Includes */
#include <unistd.h>     /* Symbolic Constants */
#include <sys/types.h>  /* Primitive System Data Types */
#include <errno.h>      /* Errors */
#include <stdio.h>      /* Input/Output */
#include <stdlib.h>     /* General Utilities */
#include <pthread.h>    /* POSIX Threads */
#include <string.h>     /* String handling */
/* prototype for thread routine */
void print_message_function ( void *ptr );
/* struct to hold data to be passed to a thread this shows how multiple data items can be passed to
a thread */
typedef struct str_thdata
{
   int x;
   int y;
   int z;
} thdata;
```

```
int main()
{
    pthread_t thread1;  /* thread variables */
    thdata Omar;            /* structs to be passed to threads */
    /* initialize data to pass to thread 1 */
    Omar.x=1;
    Omar.y=2;
    Omar.z=3;
    /* create threads 1*/
    pthread_create (&thread1, NULL, (void *) &print_message_function, (void *) &Omar);

    /* Main block now waits for both threads to terminate, before it exits  If main block exits, both
    threads exit, even if the threads have not
      finished their work */
    pthread_join(thread1, NULL);

    exit(0);
}

/**
 * print_message_function is used as the start routine for the threads used it accepts a void pointer
**/
void print_message_function ( void *ptr )
{
    thdata *my_data;
    my_data = (thdata *) ptr;  /* type cast to a pointer to thdata */

    /* do the work */
    printf("X: %d, Y: %d, Z: %d\n", my_data->x, my_data->y, my_data->z);
    pthread_exit(0); /* exit */
```

```
} /* print_message_function ( void *ptr ) */
```

struct_thread1.c

```
#include <unistd.h>     /* Symbolic Constants */
#include <sys/types.h>  /* Primitive System Data Types */
#include <errno.h>      /* Errors */
#include <stdio.h>      /* Input/Output */
#include <stdlib.h>     /* General Utilities */
#include <pthread.h>    /* POSIX Threads */
#include <string.h>     /* String handling */

/* prototype for thread routine */
void print_message_function ( void *ptr );

/* struct to hold data to be passed to a thread
   this shows how multiple data items can be passed to a thread */
typedef struct str_thdata
{
    int thread_no;
    char message[100];
} thdata;

int main()
{
    pthread_t thread1, thread2;  /* thread variables */
    thdata data1, data2;         /* structs to be passed to threads */

    /* initialize data to pass to thread 1 */
    data1.thread_no = 1;
    strcpy(data1.message, "Hello!");
```

```
    /* initialize data to pass to thread 2 */
    data2.thread_no = 2;
    strcpy(data2.message, "Hi!");

    /* create threads 1 and 2 */
    pthread_create (&thread1, NULL, (void *) &print_message_function, (void *) &data1);
    pthread_create (&thread2, NULL, (void *) &print_message_function, (void *) &data2);

    /* Main block now waits for both threads to terminate, before it exits
       If main block exits, both threads exit, even if the threads have not
       finished their work */
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    /* exit */
    exit(0);
} /* main() */

/**
 * print_message_function is used as the start routine for the threads used
 * it accepts a void pointer
**/
void print_message_function ( void *ptr )
{
    thdata *data;
    data = (thdata *) ptr;  /* type cast to a pointer to thdata */
    /* do the work */
    printf("Thread %d says %s \n", data->thread_no, data->message);
```

```
    pthread_exit(0); /* exit */
} /* print_message_function ( void *ptr ) */
```

Thread Termination

There are several ways of terminating threads. Summarised as follows:

- The thread _returns_ from its starting routine.
- Thread makes a call to pthread_exit() call.
- The entire process is terminated due to call to exec() or exit().

If the main() thread finishes executing before any other thread in the process, all threads will terminate. However, if main() exits using a pthread_exit() call, all other threads in the process will continue to execute. Thus the pthread_exit() call will terminate the main thread but keep on clinging to resources such as process memory space and open file descriptors.

The following code will show both thread creation and thread termination.

threadTermination.c

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void *PrintHello()
{
printf("Hello World! It's me\n");
pthread_exit(0);
}

int main()
{
pthread_t threads[3];
int rc;
```

```
int t;
for(t=0; t<3; t++)
{
  printf("In main: creating thread %d\n", t);
  rc = pthread_create(&threads[t], NULL, PrintHello, NULL);

  if (rc)
  {
    printf("ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
  }
}
pthread_exit(NULL);

}
```

A process can exit at any time when a thread calls the **exit** subroutine. Similarly, a thread can exit at any time by calling the **pthread_exit** subroutine.

Calling the **exit** subroutine terminates the entire process, including all its threads. In a multithreaded program, the **exit** subroutine should only be used when the entire process needs to be terminated; for example, in the case of an unrecoverable error. The **pthread_exit** subroutine should be preferred, even for exiting the initial thread.

Calling the **pthread_exit** subroutine terminates the calling thread. The *status* parameter is saved by the library and can be further used when joining the terminated thread. Calling the **pthread_exit** subroutine is similar, but not identical, to returning from the thread's initial routine. The result of returning from the thread's initial routine depends on the thread:

- Returning from the initial thread implicitly calls the **exit** subroutine, thus terminating all the threads in the process.
- Returning from another thread implicitly calls the **pthread_exit** subroutine. The return value has the same role as the *status* parameter of the **pthread_exit** subroutine.

To avoid implicitly calling the **exit** subroutine, to use the **pthread_exit** subroutine to exit a thread.

Exiting the initial thread (for example, by calling the **pthread_exit** subroutine from the **main** routine) does not terminate the process. It terminates only the initial thread. If the initial thread is terminated, the process will be terminated when the last thread in it terminates. In this case, the process return code is 0.

The following program displays exactly 10 messages in each language. This is accomplished by calling the **pthread_exit** subroutine in the **main** routine after creating the two threads, and creating a loop in the **Thread** routine.

threadTermination1.c

```
#include <pthread.h>    /* include file for pthreads - the 1st */

#include <stdio.h>      /* include file for printf()          */

#include <stdlib.h>

void *Thread(void *string)

{

    int i;

    for (i=0; i<10; i++)

        printf("%s\n", (char *)string);

    pthread_exit(NULL);

}

int main()

{

    char *e_str = "Hello!";

    char *f_str = "FAST !";



    pthread_t e_th;

    pthread_t f_th;

    int rc;
```

```
    rc = pthread_create(&e_th, NULL, Thread, (void *)e_str);
    if (rc)
        exit(-1);
    rc = pthread_create(&f_th, NULL, Thread, (void *)f_str);
    if (rc)
        exit(-1);
    pthread_exit(NULL);
}
```

The **pthread_exit** subroutine releases any thread-specific data, including the thread's stack. Any data allocated on the stack becomes invalid, because the stack is freed and the corresponding memory may be reused by another thread. Therefore, thread synchronization objects (mutexes and condition variables) allocated on a thread's stack must be destroyed before the thread calls the **pthread_exit** subroutine.

Unlike the **exit** subroutine, the **pthread_exit** subroutine does not clean up system resources shared among threads. For example, files are not closed by the **pthread_exit** subroutine, because they may be used by other threads.

Data Sharing between Threads

Compile and run the following code:

threadSharingData.c

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int myGlobal = 0;
void *threadFunction()
{
int i, j;
```

```
for (i = 0; i<5; i++)
{
j = myGlobal;
j = j+1;
printf(".");
fflush(stdout);
sleep(1);
myGlobal = j;
}
}
int main()
{
 pthread_t myThread;
 int i;
 pthread_create(&myThread, NULL, threadFunction,NULL);
for (i = 0; i < 5; i++)
 {
 myGlobal = myGlobal + 1;
 printf("o");
 fflush(stdout);
 sleep(1);
 }
 pthread_join(myThread, NULL);
 printf("\nMy Global Is: %d\n", myGlobal);
 exit(0);
 }
```

Flush() is used to force a write of all user-space buffered data. Since we have specified stdout, therefore it will write it to standard output. We have two threads here again as well. The main thread has a for loop which is printing the character "**o**". The threadFunction thread also has a for loop which is printing the character " **.** ". You will notice thread-scheduling in action when you see an output of:

Output:

uhammad@muhammad-VirtualBox:~/CProgramming$ ./threadSharingData

o.o..o.o.o

My Global Is: 6

Is everything _ne with this output? Think about the myGlobal value ... Should it be 6?

Or should it be 10 (2 For loops each running for 5 iterations)?

Now comment the sleep line (both or just one) and check the output. It should be 10.

How come a sleep(1) can make such a difference? Here is a little explanation:

We are using the sleep() call to impose a rudimentary form of synchronization between both threads. With sleep, CPU alternates between both threads a total number of 5 times. Each time the myGlobal variable is overwritten by the threadFunction thread. Without sleep, there is just one alternation. The my-Global variable first counts upto 5, and then it counts upto 5 again, totalling 10. But there is no synchronization between threads in this way.

If we want to impose synchronization and at the same time preserve data integrity, we would be needing something more accurate than a simple sleep() call.

Synchronization through Mutex

Every process has a certain portion of code which is called the "critical section" of a process. As an example, this is the area where a

process may:

1. Change shared variables
2. Write to a File
3. Use a shared resource

It would be desirable that if one process is working in this region, then another process should not be allowed to modify the contents of any shared variable within it. If, and only if, that process leaves the critical section, then another process may be allowed access to that shared region. In other words, such a region is "mutually exclusive" to one-and-only-one process at a time.

Ideally, if a process enters this region, it should lock it. Any process trying to access it in the meanwhile will not gain any access because it is locked. Once the process leaves this region, it will un-lock it, rendering it open for anybody else.

We will take this concept of critical-section and mutual exclusion and apply it to our problem in this Section. Here, we apply our lock and unlock mechanism with the help of Mutexes (taken from Mutually Exclusives). If a thread is currently locked into its critical section, another thread trying to access it will go into sleep mode. Compile and run the code given. Here, the lock is our entry section and the unlock is our exit section.

mutexSynchronization.c

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int myGlobal = 0;
pthread_mutex_t myMutex;
void *threadFunction()
 {
   int i, j;
   for (i = 0; i<5; i++)
   {
    pthread_mutex_lock(&myMutex);
    j = myGlobal;
    j = j+1;
    printf(".");
```

```
    fflush(stdout);
    sleep(1);
    myGlobal = j;
    pthread_mutex_unlock(&myMutex);
   }
 }
int main()
{
 pthread_t myThread;
 int i;
 pthread_create(&myThread, NULL, threadFunction, NULL);
 for (i = 0; i < 5; i++)
 {
   pthread_mutex_lock(&myMutex);
   myGlobal = myGlobal + 1;
   pthread_mutex_unlock(&myMutex);
   printf("o");
   fflush(stdout);
   sleep(1);
 }
 pthread_join(myThread, NULL);
 printf("\nMy Global Is: %d\n", myGlobal);
 exit(0);
}
```

Exercise

A Fibonacci series is a set of numbers where every n_{th} number is the sum of the n^{th}-1 and n^{th} -2 numbers. The only exception to this rule is the 1st and 2nd numbers which are 0 and 1 respectively. The following code can find the n^{th} number in the Fibonacci sequence:

fibonacci.c

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int fib(int n)
{
 if (n<=0)
    return 0;
 else if (n==1)
    return 1;
 else
    return fib(n-1)+fib(n-2);
}
int main()
{
 int find = 40;
 printf("Element No. %d in series is: %d",find, fib(find) );
 exit(0);
}
```

Note that the call to fib() function is recursive. Modify the above code so that each fib() call is implemented in a separate thread.

References

https://www.geeksforgeeks.org/multithreading-c-2/

https://www.geeksforgeeks.org/thread-functions-in-c-c/

https://www.ibm.com/docs/en/aix/7.2?topic=programming-terminating-threads