FIT2100 Practical #4
Understanding Threads
and Thread Synchronisation
Week 9 Semester 2 2020

October 11, 2020

Revision Status:

`$Id: FIT2100-Practical-04.tex, Version 2.0 2018/09/04 18:30 Jojo $`

`Updated by Muhammed Tawfiqul Islam, Oct 2020.`

Contents

1 Background

In this practical, you will learn the concept of *threads* in the Unix/Linux environment (Section 2) and understand how synchronisation can be supported during the execution of multiple threads (Section 3).

The pre-lab preparation (defined under **Pre-class exercise** subsections, Section 2.5 and Section 3.2) should be completed before attending the lab session. The practical tasks (Section 2.6 and Section 3.3) are to be assessed in the lab.

Before you attempt any of the tasks in this prac, create a folder named `PRAC04` under the FIT2100 folder (`~/Document/FIT2100`). Save all your source files under this `PRAC04` folder.

2 Understanding Threads

(**Note:** You should complete the reading from Sections 2.1 to 2.4.)

2.1 What are Threads?

Technically, a *thread* is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.

A thread is a semi-process that has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them).

A *thread group* is a set of threads all executing inside the same process (see Figure 1). They all share the same memory, and thus can access the same global variables, the same heap memory, the same set of file descriptors, etc. All these threads execute in *parallel* (i.e. using time slices, or if the system has several processors, then really in parallel).

2.2 Pthreads

Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other, making it difficult for programmers to develop portable threaded applications.

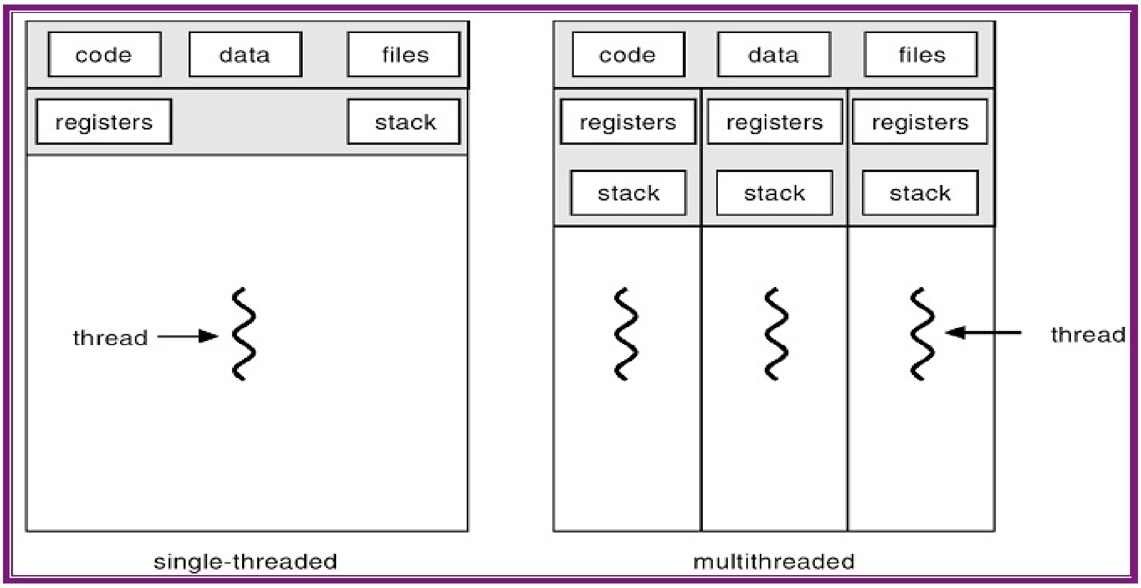

Figure 1: Single-Threaded and Multi-Threaded Programs (Processes)

In order to take full advantage of the capabilities provided by threads, a standardised programming interface was required. For Unix/Linux systems, this interface has been specified by the **IEEE POSIX 1003.1c** standard (1995). Implementations that adhere to this standard are referred to as POSIX threads, or *Pthreads*. Most hardware vendors now offer Pthreads in addition to their proprietary threads.

2.3 Are Threads Efficient?

If implemented correctly, threads have some advantages over processes. Compared to the standard `fork()`, threads carry a lot less overhead.

Remember that `fork()` produces a second copy of the calling process. The parent and the child are completely independent, each with its own address space, with its own copies of its variables, which are completely independent of the same variables in the other process.

Threads share a common address space, thereby avoiding a lot of the inefficiencies of multiple processes.

- The kernel does not need to make a new independent copy of the process memory space, file descriptors, etc. This saves a lot of CPU time, making thread creation ten to a hundred times faster than a new process creation. Because of this, you can use a whole bunch of threads and not worry about the CPU and memory overhead incurred. This means you can generally create threads whenever it makes sense in your program.

- Less time to terminate a thread than a process.
- Context switching between threads is much faster than context switching between processes.(Context switching means that the system switches from running one thread or process, to running another thread or process).
- Less communication overheads — communicating between the threads of one process is simple because the threads share the address space. Data produced by one thread is immediately available to all the other threads.

On the other hand, because threads in a group all use the same memory space, if one of them corrupts the contents of its memory, other threads might suffer as well. With processes, the operating system normally protects processes from one another, and thus if one corrupts its own memory space, other processes will not suffer.

2.4 Examples using Threads

Example 1: A responsive user interface

One area in which threads can be very helpful is in *user interface* programs. These programs are usually centred around a loop of reading user input, processing it, and showing the results of the processing. The processing part may sometimes take a while to complete, and the user is made to wait during this operation. By placing such long operations in a separate thread, while having another thread to read user input, the program can be more responsive. It may allow the user to cancel the operation in the middle.

Example 2: A graphical interface

In *graphical* programs, the problem is more severe since the application should always be ready for a message from the windowing system telling it to *repaint* part of its window. If it is too busy executing some other tasks, its window will not respond to the user, leading the user to think the program has crashed. In such a case, it is a good idea to have one thread handle the message loop of the windowing system and always ready to get such requests (as well as user input). Whenever this thread sees a need to do an operation that might take a long time to complete (say, more than `0.2` seconds in the worst case), it will delegate the job to a separate thread.

Example 3: A Web server

The Web server needs to handle several download requests over a short period. Hence, it is more efficient to create (and destroy) a single thread for each request. Multiple threads can possibly be executing simultaneously on different processors.

2.5 Pre-class exercise (1 mark)

Creating and Destroying Threads

When a *multi-threaded* program starts executing, it has one thread running, which executes the main() function of the program. This is already a full-fledged thread, with its own thread ID.

In order to create a new thread, the program should use the pthread_create() function.

Here is how to use it:

```c
#include <stdio.h>      /* standard I/O routines */
#include <pthread.h>    /* pthread functions and data structures */
#include <unistd.h>     /* we will use sleep() later on in prac */
#include <stdlib.h>     /* needed for exit() function */


/* This function is to be executed by the new thread */
void* print_hello(void* data) {
    long int my_data = (long int) data; /* data received by thread */
        //must cast received data back to original data type.

    pthread_detach(pthread_self());
    printf("Hello from new thread - got %ld\n", my_data);

    pthread_exit(NULL);     /* terminate the thread */
}


/* Like any C program, program's execution begins in main */
int main(int argc, char* argv[]) {
    int rc;                 /* value returned from thread creation */
    pthread_t thread_id;    /* thread's ID (a long unsigned integer)*/
    long int t = 11;        /* data passed to the new thread */

    /* create a new thread that will execute 'print_hello()' */
    rc = pthread_create(&thread_id, NULL, print_hello, (void*)t);

    if(rc) {    /* could not create thread */
        printf("\n ERROR: return code from pthread_create is %d \n", rc);
        exit(1);
    }

    printf("\n Created new thread (%lu) ... \n", thread_id);
    pthread_exit(NULL);      /* terminate the thread */
}
```

While the above program does not do anything useful, it will help you understand how threads work. Let us take a step-by-step look at what the program does.

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

- The `pthread_create()` gets four arguments. The first argument is a pointer to `thread_id`, used by `pthread_create()` to supply the program with the thread‘s identifier. The second argument is used to set some attributes for the new thread. In our case we supplied a `NULL` pointer to tell `pthread_create()` to use the default values.

  Notice that `print_hello()` accepts a `void*` as an argument and also returns a `void*` as a return value. This shows us that it is possible to use a `void*` to pass an arbitrary piece of data to our new thread, and that our new thread can return an arbitrary piece of data when it finishes.

  How do we pass our thread an arbitrary argument? Easy. We use the fourth argument to the `pthread_create()` call. If we do not want to pass any data to the new thread, we set the fourth argument to `NULL`. The `pthread_create()` returns zero on success and a non-zero value on failure.

  Note that the `void*` data type is normally meant for storing a memory address rather than an integer value. We are casting an integer into a `void*` and then back again in the new thread. This works (since a memory address technically *is* an integer value), and is a common ‘trick’ used in C programming, but it's technically a misuse of a data type. Can you think of a better way to pass data using a `void` pointer?

- After `pthread_create()` successfully returns, the program will consist of two threads. This is because the main program is also a thread and it executes the code in the `main()` function in parallel to the thread it creates. Think of it this way: if you write a program that does not use POSIX threads at all, the program will be single-threaded (this single thread is called the "main" thread).

- The call to `pthread_exit()` causes the current thread to exit and free any thread specific resources it is taking.

Now, download or enter the above code (in a file named `hello.c`).

In order to compile a multi-threaded program using gcc, **we need to link it with the pthreads library.** Assuming you have this library already installed on your system, here is how to compile our first program:

```
$ gcc hello.c -o hello -lpthread
```

Compile the source code and run the hello executable.

The output should be similar to:

```
Created new thread (208316031)...
Hello from new thread - got 11
```

2.6 Practical Tasks

2.6.1 Task 1 (1 mark)

A thread can get its own *thread id* by calling the pthread_self() function, which returns the thread id of type pthread_t:

```
pthread_t tid;
tid = pthread_self();
```

Now, duplicate the program hello.c from Section 2.5 and name the duplicated copy as hello1.c. Modify the code for hello1.c to print out the *thread id* for both threads. Recompile and run the hello executable. The new output should be similar to:

```
I am thread 3083799360. Created new thread (3083710031)...
Hello from new thread 3083710031 - got 11
```

Now modify the code so that the "main" thread passes its own *thread id* as the data to the "new" thread it creates. Recompile and run the hello executable. The output should be similar to:

```
I am thread 3083799360. Created new thread (3083710031)...
Hello from new thread 3083710031 - got 3083799360
```

2.6.2 Task 2 (1 mark)

There are several ways for threads to terminate. One way to safely terminate is to call the `pthread_exit()` function, which is the equivalent of `exit()` for processes.

In this exercise, first duplicate the program `hello1.c` from Task 1 (Section 2.6.1) and name the duplicated copy as `hello2.c`. Modify the `hello2.c` program as follows.

In the `print_hello()` function, add a line before the `printf` call with `sleep(1);`. This should be the first line of the function. In the `main()` function, comment out the last statement line, which contains the `pthread_exit()` call. Recompile and run the `hello` executable.

Explain what happens by giving a reason and document your explanation.

Now, put the `pthread_exit()` call back in the main program, but remove it from the `print_hello()` function. Also add the sleep call to the `main()` function, just before the second `printf` call, and remove it from the `print_hello` function. Recompile and run the `hello` executable.

Again, explain what happens by giving a reason and document your explanation.

(**Note:** It is not necessary to use the `pthread_exit()` at the end of the main program. Otherwise, when it exits, all running threads will be killed.)

2.6.3 Task 3 (0.5 marks)

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

The first argument to the `pthread_join()` is the identifier of the thread to join. The second argument is a `void` pointer.

```
 pthread_join(pthread_t tid, void* return_value);
```

If the `return_value` pointer is non-`NULL`, the `pthread_join()` will place at the memory location pointed to by the `return_value`, the value passed by the thread `tid` through the `pthread_exit()` call.

If we do not care about the return value of the main thread, we will set it to `NULL`.

Download the code given below `hello3.c`. Compile and run the executable.

Is the output what you expected?

```c
#include <stdio.h>      /* standard I/O routines */
#include <pthread.h>    /* pthread functions and data structures */
#include <unistd.h>
#include <stdlib.h>     /* needed for exit function */


/* This function is to be executed by the new thread */
void* print_hello(void* data) {
    pthread_t tid = (pthread_t) data;   /* data received by thread */

    pthread_join(tid, NULL);        /* wait for thread: tid */

    printf("Hello from new thread %lu - got %lu\n", pthread_self(), tid);
    pthread_exit(NULL);         /* terminate the thread */
}

int main(int argc, char* argv[]) {
    int rc;         /* value returned from thread creation*/
    pthread_t thread_id, parent_id; /* thread IDs */

    parent_id = pthread_self();
    rc = pthread_create(&thread_id, NULL, print_hello, (void*) parent_id);

    if(rc) {    /* could not create thread */
        printf("\n ERROR: return code from pthread_create is %d \n", rc);
        exit(1);
    }

    sleep(1);
    printf("\n Created new thread (%lu) ... \n", thread_id);
    pthread_exit(NULL);
}
```

Note: At any point in time, a thread is either *joinable* or *detached*. (The default state is joinable.)

Joinable threads must be reaped or killed by other threads (using the `pthread_join()` function) in order to free memory resources. Detached threads cannot be reaped or killed by other threads, and resources are automatically reaped on termination. So unless threads need to synchronise among themselves, it is better to call the following instead of the `pthread_join()`:

```c
    pthread_detach(pthread_self());
```

2.6.4 Task 4 (1.5 marks)

Write a program `hellomany.c` that will create `N` number of threads where `N` is specified in the command line, each of which prints out a "`hello`" message and its own thread ID.

To see how the execution of the threads *interleaves*, make the main thread sleep for `1` second for every `4` or `5` threads it creates.

The output of your code should be similar to the following: (Note: on our system, the ID numbers assigned to threads are usually much larger numbers than illustrated below.)

```
I am thread 1. Created new thread (4) in iteration 0...
Hello from thread 4 – I was created in iteration 0
I am thread 1. Created new thread (6) in iteration 1...
I am thread 1. Created new thread (7) in iteration 2...
I am thread 1. Created new thread (8) in iteration 3...
I am thread 1. Created new thread (9) in iteration 4...
I am thread 1. Created new thread (10) in iteration 5...
Hello from thread 6 – I was created in iteration 1
Hello from thread 7 – I was created in iteration 2
Hello from thread 8 – I was created in iteration 3
Hello from thread 9 – I was created in iteration 4
Hello from thread 10 – I was created in iteration 5
I am thread 1. Created new thread (11) in iteration 6...
I am thread 1. Created new thread (12) in iteration 7...
Hello from thread 11 – I was created in iteration 6
Hello from thread 12 – I was created in iteration 7
```

3 Thread Synchronisation

(**Note:** You should complete the reading in Section 3.1.)

3.1 More on Threads

A process as described in the lectures is sometimes called a *heavyweight* process. It contrasts with a thread (or *lightweight* process).

Threads are distinguished from traditional processes in that processes are typically independent, carry considerable state information, and have separate memory address spaces. Multiple threads within a same process, on the other hand, typically share some state information of the process, and share memory and other resources directly.

As mentioned in Section 2.3, context switching between threads in the same process is typically faster than context switching between processes.

As an example, when 40 users are logged on to the same Unix server and are running the same text editor such as `vi`, we have 40 threads. In this example, the 40 threads are at least sharing the memory space, which stores the code of the `vi` editor (which is a system program), but each thread has its own data section (i.e. each user is editing his/her own text document).

Every process must have at least one ‘thread of control’ or it will not do anything. You can create more than one ‘thread of control’ in one process by invoking the `pthread_create()` function as seen in Section 2.5.

3.2 Pre-class exercise (1 mark)

Download the C program called `thread.c` from Moodle. This program creates two threads by calling the `pthread_create()`, and the two threads will run concurrently.

The `main()` just creates two threads of execution and then waits for them to terminate. The functions `pthread1()` and `pthread2()` (for the two threads) are actually of the same code. In this simple example, we just use several nested for loops to take up time on the CPU. You could certainly modify the two functions so that the two threads do some more useful work.

Compile this program using the following command line:

```
$ gcc thread.c -o thread -lpthread
```

Run the `thread` program for 20 times, each time observing the behaviour. Since the threads execute concurrently, you should see both threads start to run immediately (they will both print their start-up messages at the same time). You may find, over a number of runs, that on some occasions Thread 1 terminates first, and on other occasions Thread 2 terminates first, depending on which thread happens to get the CPU first.

3.3 Practical Tasks

3.3.1 Task 5: Race Condition (2 marks)

Now that we can create concurrently executing threads, we can easily demonstrate *race condition*, since each thread can access a program’s global variables.

Download another C program called `race.c` from the Moodle site. Note the global variable `theValue` which is initialised to the value 50. The two threads will both try to modify this variable simultaneously.

The function `main()` generates two threads of execution: `pthread1()` and `pthread2()`. When both threads terminate, the program then displays the final value of the global `theValue` and terminates.

`pthread1()` attempts to `increment` the global variable by making a local copy, adding one to the local copy, and then saving the local copy back to the global variable. `pthread2()` attempts to *decrement* the global variable using a similar procedure. In both cases, a random delay of one or zero seconds is introduced between each of the load, increment/decrement, and save steps.This provides us with enough random timing to have thread switches occur at unpredictable times during the transactions, and hence produce an unpredictable output.

Compile this program using the following command line:

```
$ gcc race.c -o race -lpthread
```

Run the `race` program for 10 times. In each case, the output will be 49, 50 or 51, but it is not possible to predict the output on any given run. If the two threads were able to gain exclusive access to the global variable before they modified it, the output would always be 50.

Is it possible to get a value less than 49 or more than 51?

3.3.2 Task 6: Thread Synchronisation (2 marks)

The `Pthread` library offers the `pthread_mutex_t` data type, which is much like a binary semaphore and therefore somewhat of limited utility in the solution of synchronisation problems.

Fortunately, POSIX gives you the more general-purpose semaphore in the `sem_t` data type. In this last task, you will work with these two mechanisms for thread synchronisation, as you will implement a solution to avoid the race condition.

You will use two different types of synchronisation mechanisms: (1) Pthreads **mutex locks** and (2) POSIX unnamed **semaphores**.

Use your best resources to investigate how to work with both these mechanisms. You might want to refer to the manual pages of the following library methods:

```
pthread_mutex_init
pthread_mutex_lock
pthread_mutex_unlock
sem_init(3)
sem_wait(3)
sem_post(3)
```

Write the answers to the following questions:

(a) Describe succinctly the difference between **mutex** and **semaphore**.

(b) Use a couple of sentences to describe each of the six methods listed above.

First, use `mutex` to solve the race problem in Task 5 (Section 3.3.1) by enforcing *mutual exclusion*. Modify the code so that every time you run the program, the output is always 50.

Now, use `semaphore` to solve the same problem.

Do you see any difference between these two implementations? Which one is preferable and why? Document your explanation.