School of Computing and Information Systems

COMP30023: Computer Systems

Lab Week 10

1 Introduction

This lab will provide an introduction to threads using pthreads.

2 Creating a Thread

The `main()` function of a C program runs on its own thread (commonly called the 'main' thread)

We can create additional, independent threads by using the `pthread_create` function provided by `pthread.h`.

The listing `thread1.c` on the LMS creates such a thread. The thread runs the function `say_hello()` upon creation.

1. Compile and run `thread1.c`. Note the use of the **-lpthread** option to explicitly link the pthread library Command: `$ gcc thread1.c -o thread1 -lpthread`

   `$ ./thread1`

2. Notice how the second thread said hello before the first thread? This is because the `pthread_join` function will wait for the thread specified in the function call to finish executing before proceeding with the current thread.

   In this scenario the main thread 'waited' for the other thread to 'join' it before proceeding.

3. Can you guess what might happen if we did not call the `pthread_join` function? Comment that line in the code, compile and rerun to observe the output.

   Ask your laboratory demonstrator if you are unable to understand the behaviour you observe.

3 Threads and Race Conditions

Race conditions, where the final result of a computation depends on the order in which threads happened to run, may occur when several threads access a shared resource.

1. The code `thread2.c` given on the LMS has two threads accessing the common global variable `count`. This code has a race condition.

   Run the code several times and observe that the output changes each time.

2. We can solve race conditions such as these by defining a section of code that can only be executed by one thread at a time (called a '**critical section**').

   We can use a **mutex** to define a critical section. The methods `pthread_mutex_lock(&lock)` and `pthread_mutex_unlock(&lock)` can be used to define a critical section, where `lock` is a global variable that is of type `pthread_mutex_t`.

3. The definition, initialisation, and destroying of the mutex have been written for you in `thread2.c`. Determine the critical section that would prevent the race condition and use the function calls to lock and

unlock the mutex to fix the race condition.

```
pthread_mutex_lock(&lock)
/* Code in Critical Section */
pthread_mutex_unlock(&lock)
```

4 Spatial Locality

The principle of spatial locality says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses CPU caches take advantage of this by reading in cache lines from physical memory and reading from the CPU cache is order of magnitudes faster than actually reading from Physical RAM[1].
In the appendix, titled `fast.c` and `slow.c`, you will find one program that makes good use of spatial locality, while the other does not. Compare the performance of both programs using `time` to measure the runtime for each program.

1. Compile both programs.
   `gcc -o slow slow.c; gcc -o fast fast.c`
2. Time the runtime for both programs.
   `time ./slow; time ./fast`
3. Notice the difference in runtime.

Extra question: A multi-threaded web server

In `thread3.c`, we present a simple HTTP web server. The server currently makes use of two threads, a welcome thread which is dedicated to accepting new connections and a main thread which waits for and processes requests from clients.

In the function `handle_request`, the server reads the request from the client and sends the appropriate response. To simulate some realistic actions that a web server might perform (e.g. verify user credentials, perform reads and writes to a database to track users' access), a 3 second sleep statement is added. Additionally, a constraint is placed on the time frame that the server has to process requests. If a connection that has been accepted is not processed within 1 second, then the server will return a 503 response to the client to prevent a large number of unprocessed queued connections.

1. Run the server by compiling and executing the command `$ ./thread3 <interface> <port>`.
2. Now open two terminal windows and issue the command `$ curl <interface>:<port>/thread3.c` in both windows. If this action is performed quickly, you should see the contents of `thread3.c` in one window and a 503 message in the other. If you have specified your VM IP and a port that is within the range of 8000-10000, you should also be able to observe the same behaviour by visiting `http://<vm-ip>:<port>` with two tabs on your web browser.
3. As you can imagine, this behaviour is not ideal. You task is to modify `thread3.c` in a way that allows multiple requests to be processed simultaneously through threading, such that both clients will receive the contents of files that are being served. You should not reduce the request processing time by removing the sleep statement nor relax the time constraint.
4. Optional extension for those wanting a challenge (warning: this will likely be very time consuming): Web servers do not generally spawn a new thread for every new connection. If there are lots of connections, then there will be significant overhead relating to the creation and destruction of threads. One approach is to have worker threads in a thread pool which wait for and performs tasks (similar to what is described in section 2.2 of the textbook). Rewrite the web server to fit this model.

[1]https://gist.github.com/jboner/2841832

A thread1.c

```
/************************************

 Demo for pthread commands
 compile: gcc threadX.c -o threadX -lpthread

*************************************/

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

void *say_hello(void *param);  /* the work_function */

int main(int args, char **argv)
{
    pthread_t tid; /* thread identifier */

    /* create the thread */
    pthread_create(&tid, NULL, say_hello, NULL);

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

    printf("Hello from first thread\n");
    return 0;
}

void *say_hello(void *param)
{
    printf("Hello from second thread\n");
    return NULL;
}
```

B thread2.c

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

#define ITERATIONS 1000000

void *runner(void *param);   /* thread doing the work */

int count = 0;
pthread_mutex_t lock;

int main(int argc, char **argv)
{
    pthread_t tid1, tid2;
    int value;

    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("\n mutex init failed\n");
        exit(1);
    }

    if(pthread_create(&tid1, NULL, runner, NULL))
    {
      printf("\n Error creating thread 1");
      exit(1);
    }

    if(pthread_create(&tid2, NULL, runner, NULL))
    {
      printf("\n Error creating thread 2");
      exit(1);
    }

    if(pthread_join(tid1, NULL)) /* wait for the thread 1 to finish */
    {
      printf("\n Error joining thread");
      exit(1);
    }

    if(pthread_join(tid2, NULL))        /* wait for the thread 2 to finish */
    {
      printf("\n Error joining thread");
      exit(1);
    }

    if (count != 2 * ITERATIONS)
        printf("\n ** ERROR ** count is [%d], should be %d\n", count, 2*ITERATIONS);
    else
        printf("\n OK! count is [%d]\n", count);

    pthread_exit(NULL);
    pthread_mutex_destroy(&lock);

}

void *runner(void *param)   /* thread doing the work */
{

    int i, temp;
    for(i = 0; i < ITERATIONS; i++)
    {
        temp = count;     /* copy the global count locally */
        temp = temp + 1;    /* increment the local copy */
        count = temp;     /* store the local value into the global count */
    }

}
```

C fast.c

```
#define BIG 10000
int big[BIG][BIG];

int main(void) {
    for (int i = 0; i < BIG; i++) {
        for (int j = 0; j < BIG; j++) {
            big[i][j] = 0;
        }
    }
    return 0;
}
```

D slow.c

```c
#define BIG 10000
int big[BIG][BIG];

int main(void) {
    for (int i = 0; i < BIG; i++) {
        for (int j = 0; j < BIG; j++) {
            big[j][i] = 0;
        }
    }
    return 0;
}
```