Exercise Manual
For

CE3103
Embedded Programming

Practical Exercise #4
Multithreaded C Programming
(on Raspberry Pi Embedded Platform)

Venue: Hardware Laboratory 2
(Location: N4-01b-05)

COMPUTER ENGINEERING COURSE

**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY**

Learning Objectives

The following practical exercises are designed to enable students to gain first-hand experience in writing multithreaded C program using the Pthreads library as well as developing socket programming program for IPC between the Raspberry Pi (RPi) embedded platform and a networked computer. Furthermore, student will also learn how to configure the network to enable remote access over the network using SSH.

Equipment and accessories required

i)   One Raspberry Pi (RPi 3 or 4) board with SDCard installed with Raspbian.
ii) One Ubuntu based PC installed (for SSH remote access to RPi)
iii) One Ethernet cable.
iv) One Micro-USB adapter/cable (for supplying power to RPi board)

---

1.   Introduction

Multithreaded programming is a very efficient way to implement multi-tasking system due to its smaller resource overhead and simpler IPC, and Pthreads is the most commonly used library for C language programming as it is very portable across different OS platforms. As such multithreaded programming is a useful technique for embedded OS based system that needs multi-tasking capability to improve its system performance. On the other hand, server-client based design has been proven to be the most efficient way to connect multiple computers over network and it has been used in many real-world applications.
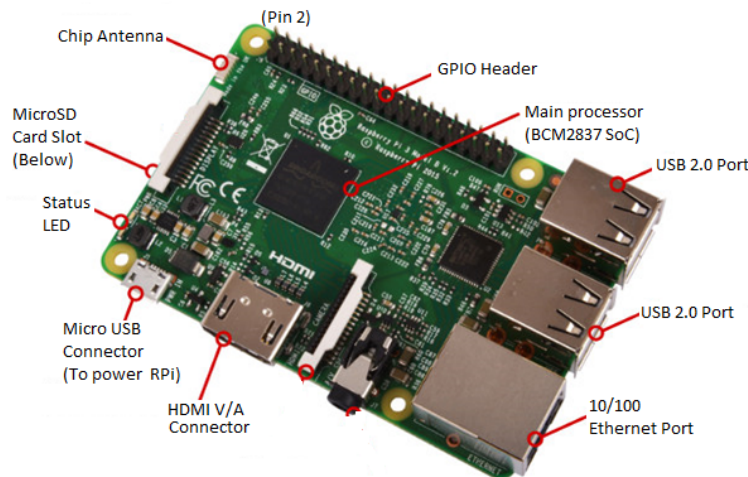

Figure 1: Raspberry Pi 3

Raspberry Pi (RPi) is an ARM processor (64-bit quad-core ARM Cortex A53 for the version of RPi 3 board we used here) based embedded board originally developed for teaching/learning computer programming purpose, but has since been widely used in many practical embedded applications. In addition, RPi runs embedded version of the Linux OS (in this Lab, we are using the Raspbian version. Q: How do you check the distribution version?).

In the following exercises, students will learn how to write multithreaded C program with server-client networked based application, and be familiarized with program development in an embedded Linux environment for an embedded platform.
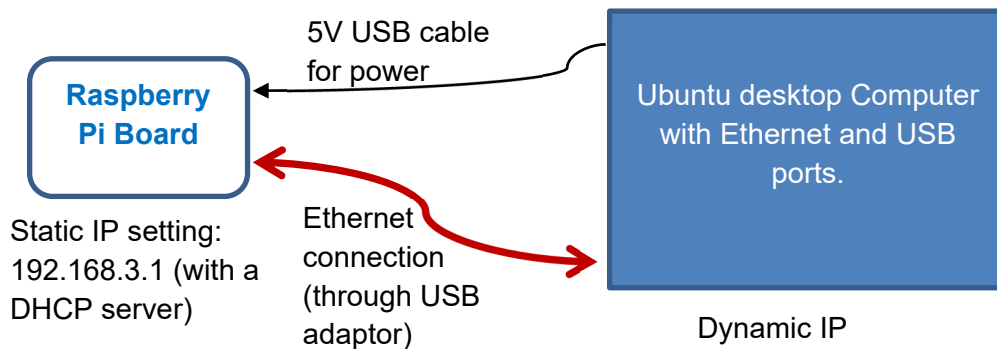
1.1 System setup and connection

Figure 2: System setup and configuration

Figure 2 illustrates the basic setup of the system to be used in this Exercise. The RPi board is powered by the desktop computer through a USB cable (from the computer's USB port). It should also be already configured with a static IP (192.168.3.1) and installed with a DHCP server. This setup will allow the desktop computer configured for dynamic IP to easily connect to the RPi board using Ethernet based network connection. (You can check the IP address assigned to the desktop computer by using the command `ifconfig`.)

The RPi board runs Raspbian version of the Linux ported for the RPi embedded board. As such, it contains most of the software packaged found in Linux (or can be easily downloaded and installed through internet if needed). Examples are the GCC C compiler (of the GNU toolchain) and the SSH client program that is used in the laboratory exercises here.

As the RPi board will be used in a 'headless' mode (i.e. without monitor/keyboard/mouse connected directly to the board), remote command line login using SSH (Secure Shell) will be used on the computer to communicate with the RPi board.

In order to enable remote access of the RPi (over the network from the Ubuntu computer), the RPi board needs to be properly configured with appropriate network settings, and for convenience, also running a DHCP server. It is likely that the RPi board (more correctly, the Raspbian OS on the SDCard) that you have would have been configured properly for remote SSH access.

1.2 System connection

Connect the RPi board to the desktop Ubuntu computer using the Ethernet cable (through a USB adaptor) provided. Power up the RPi board through the Desktop USB port.

On the Ubuntu computer, open a terminal and issue the following ssh command to connect to the RPi board (check with the Lab assistant if the IP doesn't seem to be correct):

`ssh pi@192.168.3.1`

The default password is "**`raspberry`**".

Aside: Shutting down and restarting the RPi board

To shutdown (i.e. halt) or restart the RPi, use the command "**`shutdown –h 0`**" or "**`shutdown –r 0`**". (The '0' is for delay of 0 seconds, meaning immediately.)

However, only a system administrator (the "root" user in Linux system) with privileged access to the system is allowed to execute such type of commands. In Linux, a program called "**sudo**" can be used to temporarily allow a normal user to issue system level commands such as below:

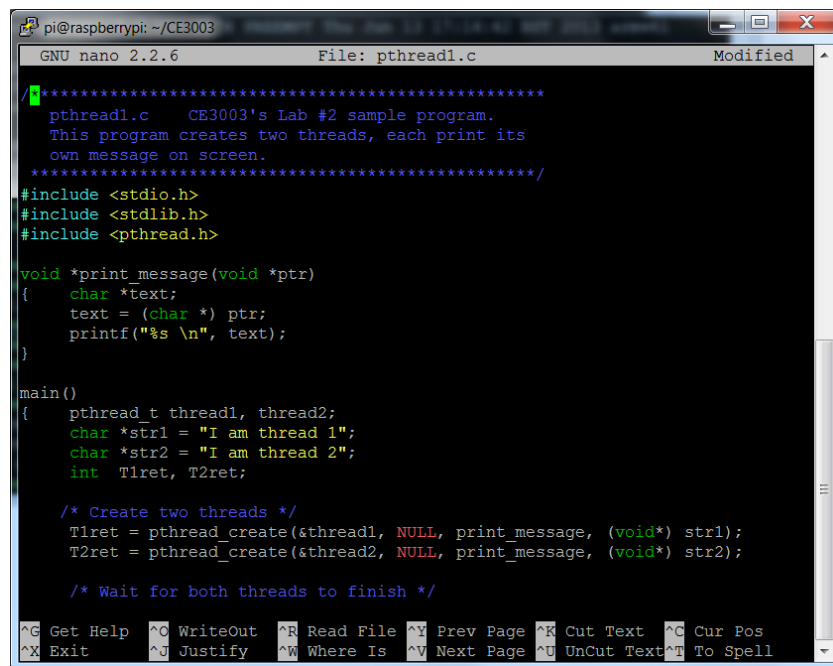<div align="center">

sudo shutdown –h 0

</div>

2. Practice Exercises - Multithreaded Programs using Pthreads library

The first exercise is to let the students practice creating a C program file (using the nano editor). A sample C program pthread1.c is shown in Appendix A, which shows the various steps used to create a two-thread C program using the Pthreads library. Students will then learn how to compile the C program using the GCC C compiler. The successfully compiled program will then be executed on the RPi board, displaying messages to indicate the program is behaving as expected.

2.1   Practice A – Creating a simple two threads C program

Establish a SSH session with the RPi board as before.

➢ Create a working directory ce3103/'labgroup' under the default pwd (which may have already been done during Exercise #1). Change to this subdirectory and use the nano (or vi) text editor to create a new text file named "pthread1.c".

➢ Key in the program codes as given in the sample C program (See Appendix A)

➢ Save the file after completing the entering of the program codes.

➢ Compile the program using the gcc command (or using a Makefile):

<div align="center">

gcc -o pthread1 pthread1.c -lpthread

</div>

If there is(are) compilation error(s), this is a good chance to practice code debugging.

➢ Once the code is successfully compiled, a new executable file **pthread1** will be created in the subdirectory. (Otherwise, check your program code to find and fix the error(s) indicated by the compiler output message(s), and re-compile the program.)

➢ Run the program by using the command: `./pthread1`

The program should run successfully with the appropriate messages shown on screen.

(Aside: the '.' in the above command mean 'this directory', which is to tell the system that the program should be found in the present directory that the user is currently in.)

This completes the typical procedure used to create and execute a program in a Linux environment.

Before you proceed to next exercise, study the program code carefully to understand the structure and functions calls used in a Pthreads based multi-thread C program.

Explore further:
Comment off the two **pthread_join()** *statements in the program. Compile and run the program, and observe whether it still behaves as expected.*

2.2   Practice B - Race Condition and Threads Synchronization

One advantage of a multithreaded program is that data can be shared easily since all threads share the same memory. However, when multiple threads update a shared data, it may lead to misbehaving behaviour, as will be demonstrated in the following exercise.

➢ Code a C program, "pthread2.c" that contains the following function.

```
int  g_var1 = 0;                        // global variable

void *inc_gv()
{ int i,j;
  for (i=0;i<10;i++)
       { g_var1++;                       // increment the global variable
         for (j=0; j<5000000;j+);    // delay loop
         printf(" %d",g_var1)         // print the value
         fflush(stdout);
       }
}
```

➢ Add the **main()** function that spawns two threads that both call the above function. The following is code snippet of the **main()**.

```
main()
{ pthread_t TA, TB;
  int TAret, TBret;
        :
        :
  printf("\n pthread2 completed \n");
}
```

➢ Compile and run the program. Observe the output values shown on the terminal. **Look through the code and understanding the cause of the problem before attempting the next exercise.**

2.3   Practice C - Critical Section and Mutex

The problem observed in the pthread2 program is due to the lack of synchronization between the threads that access the shared data, which leads to a situation known as Race condition. One technique that can be used to resolve this type of problem is through the use of critical section. which is the part of a program where only one thread can enter at any one time.

One way to achieve this atomic access is through the use of a Mutex object. Mutex is used to delimit (i.e. define) the boundary of the critical section. It excludes other threads from entering the critical section if a thread had earlier entered the section and acquired(locked) the Mutex. The Mutex will only be released (unlocked) after the thread exits the critical section.

The following are the declaration and some control functions needed when using a Mutex in a multithreaded program.

- To declare a mutex object (e.g. `mutexA`):   `pthread_mutex_t mutexA`

- To initialize the mutex object: `pthread_mutex_init(&mutexA, NULL)`

- To lock the mutex object: `pthread_mutex_lock(&mutexA)`

- To release the mutex object: `pthread_mutex_unlock(&mutexA)`

➢ Copy your "pthread2.c" to a new file "mutex1.c".

➢ Add a mutex object to the program code in "mutex1.c", together with the necessary mutex control functions, to remove the race condition problem observed in "pthread2.c".

➢ Compile and run the program to confirm that the issue is indeed resolved in your new program.

3. Exercise

Now you should be familiar with how to create, compile and execute multithread programs in a Linux environment. Use the Server-Client codes given in the lecture note as reference, develop two programs with functionalities as follows:

i)   A client program running on the Ubuntu computer and access the server on the RPi over the network. The client program will request the user to key in a number, which is then sent to the server. The client will also display the message it receives from the server.

ii)   A server program running on the RPi board. The server program should spawn a thread to response to the client by multiply the number received by 5, and send it back to the client.

Challenge

➢ Modify your server program such that it can spawn up to three threads to serve up to three clients simultaneously.

➢ Modify your client program such that it can fork up to three child processes, each requesting connection to the server by sending different numbers.

You may want to consider using a producer-consumer based semaphore scheme to manage/limit the number of threads spawned by the server during operation.

Appendix A: Sample two-thread C program

a) `pthread1.c`

```c
//*************************************************************
//   pthread1.c    CE3103's Exercise #2 sample program.
//   This program will create two threads, each print its
//   own message onto the screen.
//*************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *print_message(void *ptr)
{    char *text;
     text = (char *) ptr;
     printf("%s \n", text)
     return NULL;
}


main()
{    pthread_t thread1, thread2;
     char *str1 = "I am thread 1";
     char *str2 = "I am thread 2"
     int  T1ret, T2ret;

     /* Create two threads */
     T1ret = pthread_create(&thread1, NULL, print_message, (void*) str1);
     T2ret = pthread_create(thread2, NULL, print_message, (void*) str2))

     /* main() thread now waits for both threads to finish */
     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("T1 & T2 return: %d, %d\n",T1ret, T2ret);
     return 0
}
```