



School of Computer Engineering

**Laboratory Manual
for
CE3003
Microcontroller
Programming**

Laboratory #2

**Multithreaded C Programming
(on Raspberry Pi Embedded Platform)**

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

COMPUTER ENGINEERING COURSE

**SCHOOL OF COMPUTER ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY**

Learning Objectives

The laboratory #2 exercises have been designed to enable students to gain first-hand knowledge in writing multithreaded C program using the Pthreads library, and to understand how resource sharing must be carefully synchronized in a program when multiple threads can access the same resource. In addition, student will also gain exposure to the process of developing C program in a Linux environment running on an embedded platform (Raspberry Pi), and learn some basic Linux commands and programs (e.g. gcc, SSH, vi, nano.) commonly used for such purposes.

Equipment and accessories required

- i) One Raspberry Pi (RPI) board with SDCard installed with Raspbian.
- ii) One Win7 PC (installed with program PUTTY for SSH remote access.)
- iii) One Ethernet cable.
- iv) One Micro-USB cable (for supplying power to RPI board by the PC.)

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. Hence multithreaded programming is a useful technique for embedded OS based system that needs multi-tasking capability to improve its system performance.

Raspberry Pi (RPI) is an ARM (ARM11) processor based embedded board originally developed for teaching/learning computer programming purpose, but has since been widely used in many embedded applications. In addition, RPI runs embedded version of the Linux OS (in this Lab, we are using the Raspbian version).

From the exercises in this laboratory session, students will learn how to write multithreaded C program, understands the requirement of synchronization using Mutex and Semaphore, and be familiarized with program development in an embedded Linux environment.

1.1 System setup and configuration

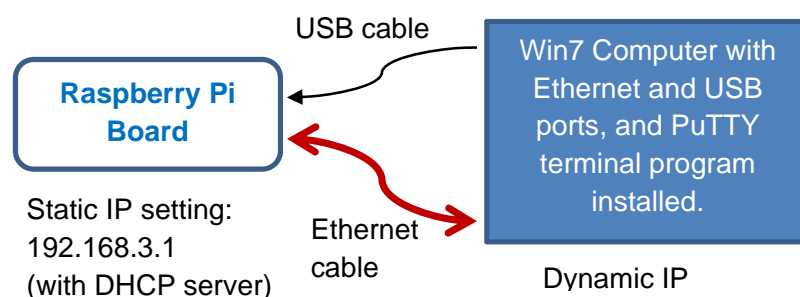


Figure 1: System setup and configuration

Figure 1 illustrates the basic setup of the system used in this laboratory #2 session. The RPI board is powered by the computer through a USB cable connection. It is also configured with static IP (192.168.3.1) and installed with a DHCP server. This enables a computer configured for dynamic IP to easily connect to the RPI board using Ethernet based network connection.

CE3003 Microcontroller Programming Laboratory #2



Figure 2: RPi in operation

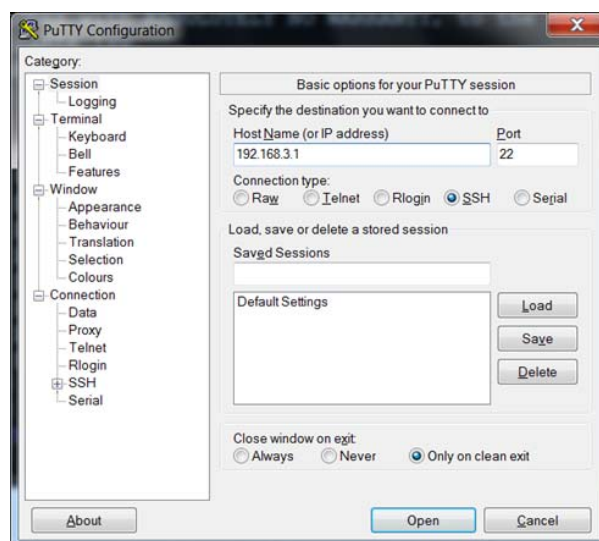
The RPi board runs Raspbian version of the Linux ported for the RPi embedded board. As such, it contains most of the software packaged found in Linux (or can be easily downloaded and installed through internet if needed), such as the GCC C compiler (of the GNU toolchain) that will be used in the laboratory 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. The software installed on the Win7 computer in the laboratory for such purpose is the PuTTY.

1.2 System start-up

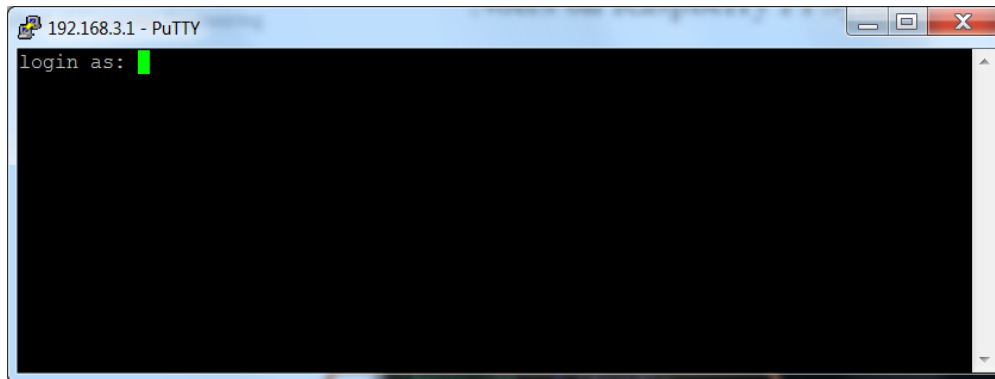
All RPi boards for this laboratory have been pre-configured as described above. Upon power-on (by connecting the USB cable between the computer and the board), the RPi board will automatically bootup and initialize using the appropriate settings, ready for the remote SSH connection. (The bootup process might take 30 seconds or so, and the board should be ready once all the LEDs, located on the left bottom corner of the board shown in Figure 2, turned on and blinking.)

On the Win7 computer, run the PuTTY program, which will initially present a Window as shown below. Key in the IP address of the RPi (192.168.3.1) in the field as shown, and click "Open".

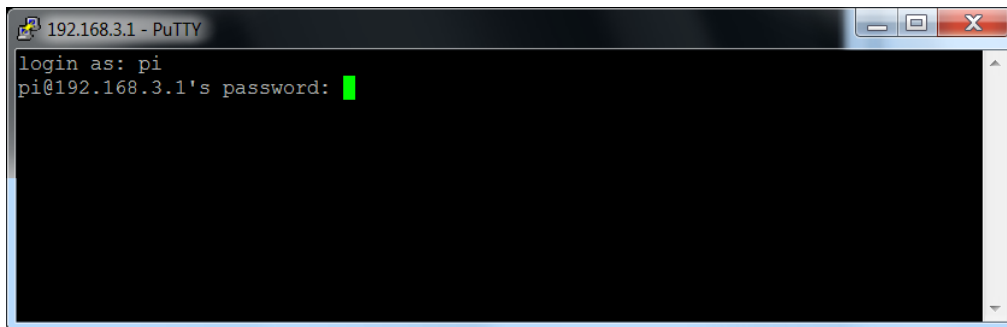


CE3003 Microcontroller Programming Laboratory #2

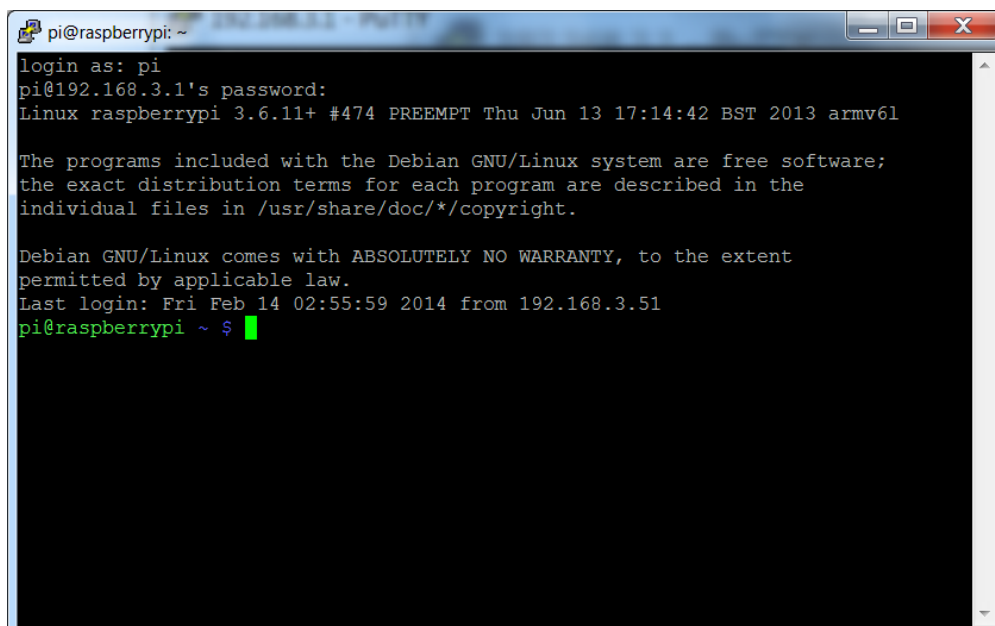
If the RPi board has bootup successfully, a command line console interface will be displayed with a login prompt.



The default login ID is “pi” and the password is “raspberrypi”.



If the remote login is successful, the console interface display on the PuTTY should be as follows:

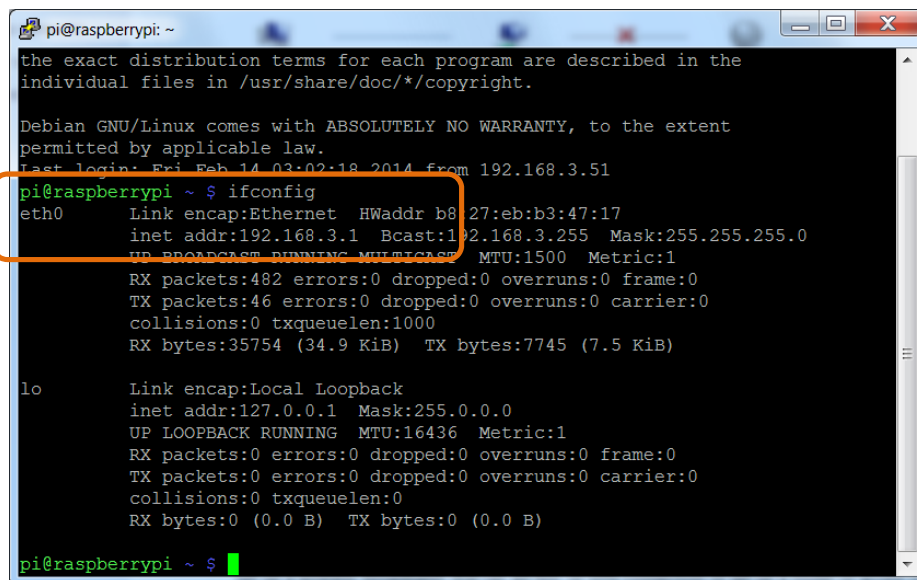


CE3003 Microcontroller Programming Laboratory #2

1.3 Basic Linux commands

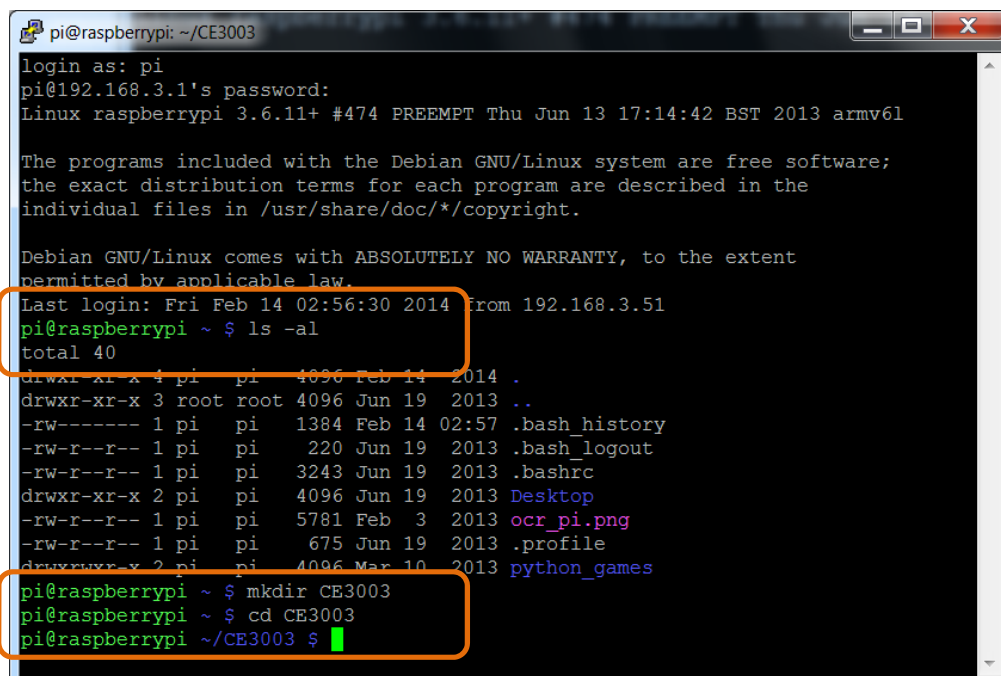
There are some commands that are commonly used when interacting with a Linux system through command line interface.

- The command “**ifconfig**” displays the details about the network interface configuration of the Linux system (i.e. the RPi board in this case).



```
pi@raspberrypi: ~  
the exact distribution terms for each program are described in the  
individual files in /usr/share/doc/*/copyright.  
  
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent  
permitted by applicable law.  
Last login: Fri Feb 14 02:02:18 2014 from 192.168.3.51  
pi@raspberrypi ~ $ ifconfig  
eth0      Link encap:Ethernet  HWaddr b8:27:eb:b3:47:17  
          inet addr:192.168.3.1  Bcast:192.168.3.255  Mask:255.255.255.0  
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1  
          RX packets:482 errors:0 dropped:0 overruns:0 frame:0  
          TX packets:46 errors:0 dropped:0 overruns:0 carrier:0  
          collisions:0 txqueuelen:1000  
          RX bytes:35754 (34.9 KiB)  TX bytes:7745 (7.5 KiB)  
  
lo        Link encap:Local Loopback  
          inet addr:127.0.0.1  Mask:255.0.0.0  
          UP LOOPBACK RUNNING  MTU:16436  Metric:1  
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0  
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0  
          collisions:0 txqueuelen:0  
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)  
  
pi@raspberrypi ~ $
```

- The command “**ls -al**” lists the content details of the current directory that the user is in (known as pwd – present work directory.)



```
pi@raspberrypi: ~/CE3003  
login as: pi  
pi@192.168.3.1's password:  
Linux raspberrypi 3.6.11+ #474 PREEMPT Thu Jun 13 17:14:42 BST 2013 armv6l  
  
The programs included with the Debian GNU/Linux system are free software;  
the exact distribution terms for each program are described in the  
individual files in /usr/share/doc/*/copyright.  
  
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent  
permitted by applicable law.  
Last login: Fri Feb 14 02:56:30 2014 from 192.168.3.51  
pi@raspberrypi ~ $ ls -al  
total 40  
drwxr-xr-x 4 pi pi 4096 Feb 14 2014 .  
drwxr-xr-x 3 root root 4096 Jun 19 2013 ..  
-rw----- 1 pi pi 1384 Feb 14 02:57 .bash_history  
-rw-r--r-- 1 pi pi 220 Jun 19 2013 .bash_logout  
-rw-r--r-- 1 pi pi 3243 Jun 19 2013 .bashrc  
drwxr-xr-x 2 pi pi 4096 Jun 19 2013 Desktop  
-rw-r--r-- 1 pi pi 5781 Feb 3 2013 ocr_pi.png  
-rw-r--r-- 1 pi pi 675 Jun 19 2013 .profile  
drwxr-xr-x 2 pi pi 4096 Mar 10 2013 python_games  
pi@raspberrypi ~ $ mkdir CE3003  
pi@raspberrypi ~ $ cd CE3003  
pi@raspberrypi ~/CE3003 $
```

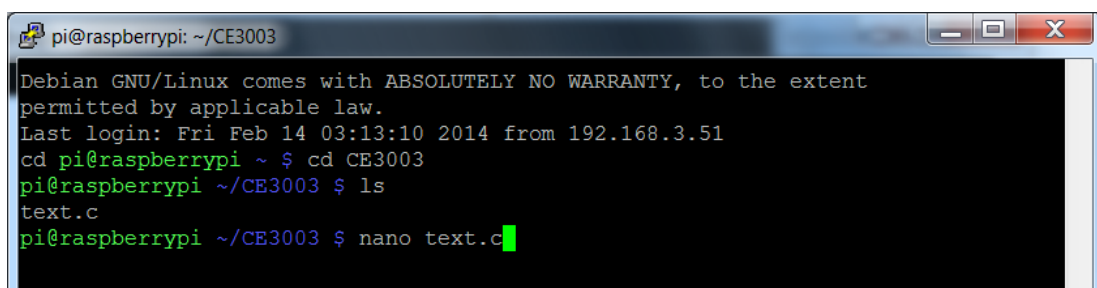
CE3003 Microcontroller Programming Laboratory #2

- To create a subdirectory, use the command “**mkdir**”. The previous screenshot shows how a “CE3003” subdirectory is created in the pwd.
- The “**cd**” command is used to change directory, such as from the pwd to the newly created directory for the previous example.

1.4 Text Editor - Nano

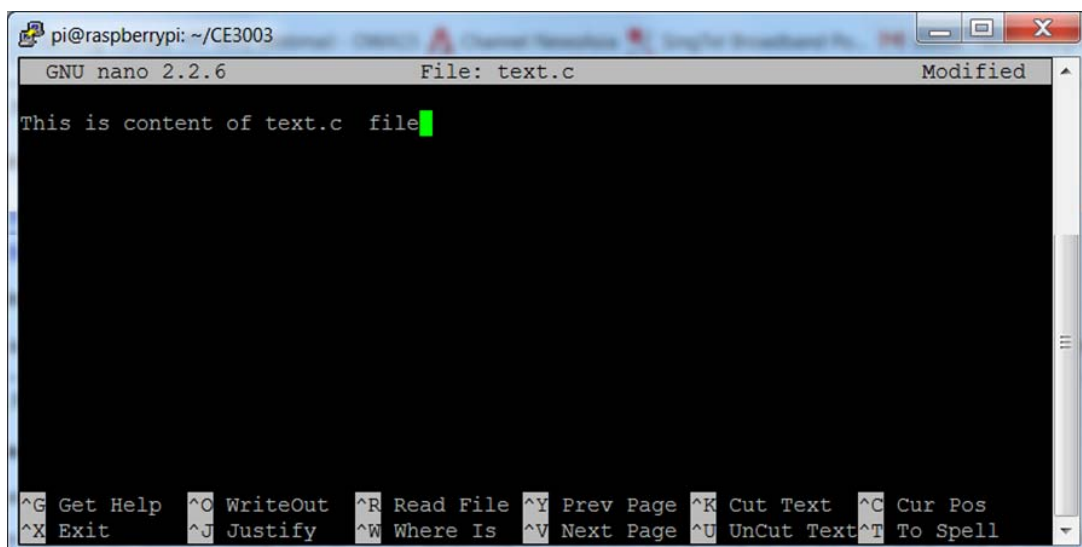
Text editor is a program to allow user to create text file, such as the program source code. There are two common text editor programs frequently used when in the text console mode to create/open/edit text file. These are the user friendly GNU **nano** text editor and the more versatile **vi** editor.

- The following screenshot show how the **nano** editor program is used to open/create a file, which allows keying in of the texts directly.



```
pi@raspberrypi: ~/CE3003
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Feb 14 03:13:10 2014 from 192.168.3.51
cd pi@raspberrypi ~ $ cd CE3003
pi@raspberrypi ~/CE3003 $ ls
text.c
pi@raspberrypi ~/CE3003 $ nano text.c
```

- All commands within the nano programs are issued using the combination of the keyboard's 'Ctrl' key with another alphabet, as shown in the various commands listed at the bottom of the screen within the editor. For example, to exit the editor (and save the edited file), use 'Ctrl-X' which is indicated as '^X' on the command lines.



```
pi@raspberrypi: ~/CE3003
GNU nano 2.2.6 File: text.c Modified
This is content of text.c file
^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos
^X Exit ^J Justify ^W Where Is ^V Next Page ^U UnCut Text ^T To Spell
```

You will most likely use this text editor for the exercises in this laboratory session, unless you are already familiar with using the vi editor, and would find it more flexible than nano.

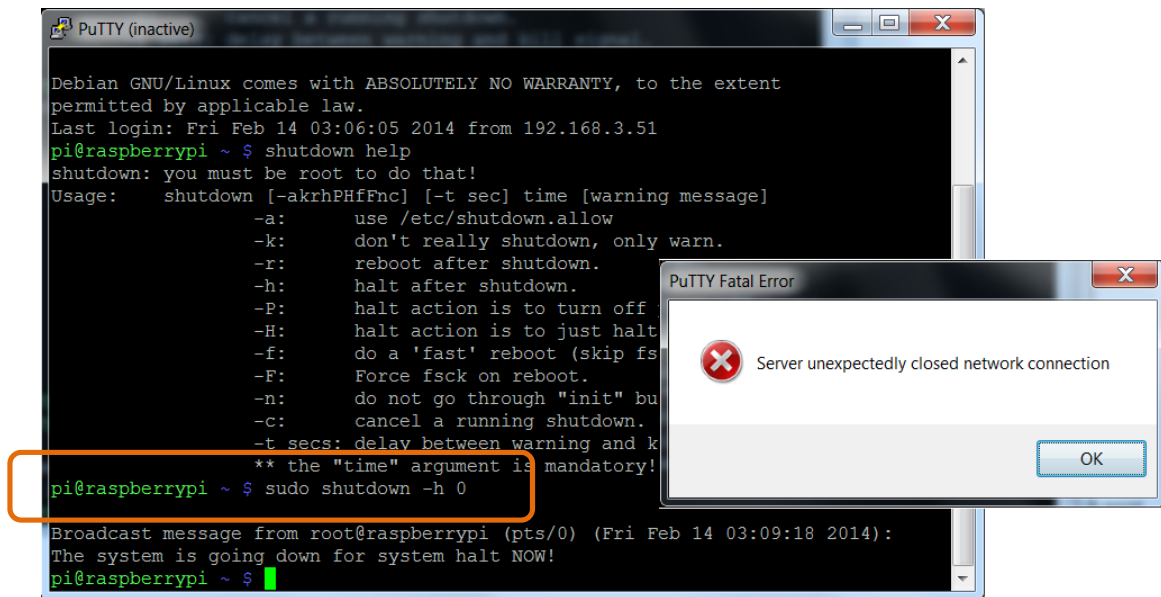
CE3003 Microcontroller Programming Laboratory #2

1.5 System Shutdown

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 RPi, a program called “**sudo**” can be used to temporarily allow a normal user to issue system level commands.

- The following screenshot shows the response from the system when the shutdown (halt) command: “**sudo shutdown -h 0**” is issued.



```

PuTTY (inactive)
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Feb 14 03:06:05 2014 from 192.168.3.51
pi@raspberrypi ~ $ shutdown help
shutdown: you must be root to do that!
Usage:      shutdown [-akrhPHfnc] [-t sec] time [warning message]
            -a:      use /etc/shutdown.allow
            -k:      don't really shutdown, only warn.
            -r:      reboot after shutdown.
            -h:      halt after shutdown.
            -P:      halt action is to turn off
            -H:      halt action is to just halt
            -f:      do a 'fast' reboot (skip fs
            -F:      Force fsck on reboot.
            -n:      do not go through "init" bu
            -c:      cancel a running shutdown.
            -t secs: delay between warning and k
            ** the "time" argument is mandatory!
pi@raspberrypi ~ $ sudo shutdown -h 0
Broadcast message from root@raspberrypi (pts/0) (Fri Feb 14 03:09:18 2014):
The system is going down for system halt NOW!
pi@raspberrypi ~ $
  
```

- Once the system is halted, the SSH connection is also terminated, causing the pop-up error message.
- A new PuTTY/SSH session is hence needed in order to re-connect to the RPi board after it is re-powered.

2. Multithreaded Program using Pthreads library

The first exercise is to let the students practice creating a C program file using nano or vi editor. A sample C program **pthread1.c** is given (available in NTULearn, and also shown in the Appendix), 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 Exercise A – Creating a simple two threads C program

Run the PuTTY program to establish a SSH session with the RPi board.

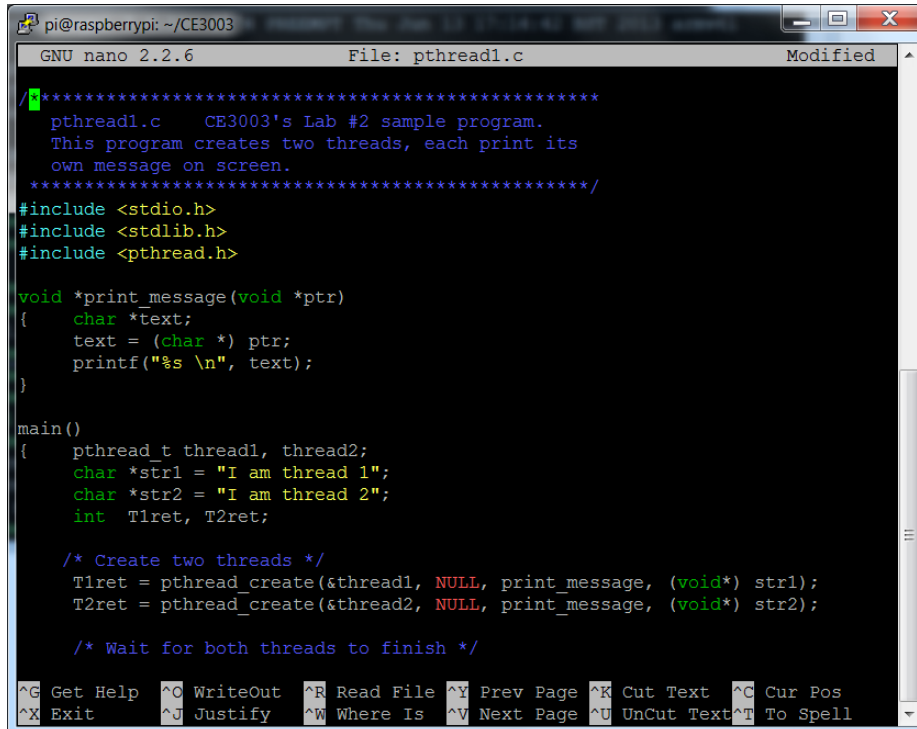
- Create a CE3003 subdirectory under the default pwd. Change to this subdirectory and use nano text editor (or vi test editor) to create a new text file named “pthread1.c”.

CE3003 Microcontroller Programming Laboratory #2

```

pi@raspberrypi ~ $ cd CE3003
pi@raspberrypi ~/CE3003 $ ls
text.c
pi@raspberrypi ~/CE3003 $ nano pthread1.c
pi@raspberrypi ~/CE3003 $ ls
pthread1.c  text.c
pi@raspberrypi ~/CE3003 $
  
```

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



```

GNU nano 2.2.6      File: pthread1.c      Modified
/*
*****
pthread1.c      CE3003's Lab #2 sample program.
This program creates two threads, each print its
own message on screen.
*****
*/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

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

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);

    /* Wait for both threads to finish */
  
```

- Save the file after complete the entering of the program codes.
- Compile the program using the gcc command:


```
gcc -o pthread1 pthread1.c -lpthread
```
- If there is no error, a new executable file **pthread1** will be created in the subdirectory. (Otherwise check through 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
```

(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 in.)
- The program should run successfully with the appropriate messages shown on screen.


```
pi@raspberrypi ~/CE3003 $ gcc -o pthread1 pthread1.c -lpthread
pi@raspberrypi ~/CE3003 $ ls
pthread1 pthread1.c text.c
pi@raspberrypi ~/CE3003 $ ./pthread1
I am thread 2
I am thread 1
T1 & T2 return: 0, 0
pi@raspberrypi ~/CE3003 $
```

This completes the typical procedure used to create 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.

3. Threads Synchronization

One advantage of a multithreaded program is that data can be shared easily since all threads shared 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.

3.1 Exercise B - Race Condition

- Write 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 TArete, 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.**

4. Critical Section

The problem observed in the program pthread2 is due to the lack of synchronization between the threads that access the shared data. It leads to a situation known as Race condition. One technique that can be used to solve this type of problem is through the use of critical section.

4.1 Exercise C - Mutex

Critical section is the part of a program where only one thread can enter at any one time. One way to achieve this mutual exclusion 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 hence 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.

4.2 Exercise D - Semaphore

Another way to control the access to a critical section of the program code is to use the semaphore object. A semaphore is essentially a counter that is decremented by a thread that enters the critical section successfully. The critical section is locked once the value of semaphore reaches 0. It will be unlocked when a thread exits the critical section, which causes an increment to the semaphore value. Semaphore can perform the mutual exclusion effect of a mutex by having it initialized to a value of 1. (The decrement and increment of the semaphore value are described as ‘wait’ and ‘post’ respectively.)

The following lists the necessary declarations and some of control functions needed to implement a semaphore for a multithreaded program.

- A semaphore header file is needed: `semaphore.h`
- To declare a semaphore object(e.g. **semM**): `sem_t semM`
- To initialize the semaphore value (e.g. set to 1): `sem_init(&semM,0,1)`
- To decrement the semaphore value(or wait if not possible): `sem_wait(&semM)`
- To increment the semaphore value: `sem_post(&semM)`

Copy your “pthread2.c” to a new file “sem1.c”. Use a semaphore in “sem1.c” to resolve the race condition problem observed in “pthread2.c”. Compile and run the program to check that it achieves the objective intended.

Appendix A: Sample two-thread C program

a) pthread1.c

```
/******  
//  pthread1.c    CE3003's Lab #2 sample program.  
//  This program creates 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);  
}  
  
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));  
  
    /* Wait for both threads to finish */  
    pthread_join( thread1, NULL);  
    pthread_join( thread2, NULL);  
  
    printf("T1 & T2 return: %d, %d\n",T1ret, T2ret);  
}
```