**CST3515 Advanced C programming Part 1**

**Please put your name and student number on this lab manual NOW!**

# LOGGING ON

For username, type: *guest*

For password, also type: *4loggingin*

Click on the Applications icon at the top of the screen and then scroll down to find the Terminal icon, click on the Terminal Icon and that should open a Terminal Window. If not type the word terminal in the application window in the top right hand corner of the screen.  
  
**SECTION 1: Revision: you should have done this in CST2522**

# PART1: Introduction to the Linux programming Environment PART1 is divided into 4 sections:

1. Editing Files
2. Writing C programs
3. Compiling C programs
4. Running C programs

**SECTION 1**: Editing Files

When UNIX was younger, **emacs** was a very powerful text editor found on all UNIX systems. It is worth getting to know. However, new and simpler editors are now common. One of those editors is called **gedit.** Use the *which* command to find out where *gedit* is located.

**SECTION 2**: Writing C programs  
Make a directory called **src** to hold the programs that you will make.

Type: **ls**

If a directory or file called **src** already exists then we need to remove it.

Type: ***rm –rf src***

Now make your own **src** directory

Type: ***mkdir src***

Go into the **src** directory

Type: ***cd src***

Use **gedit** to write your first programme..

Type: ***gedit firstprog.c***

Type the following into the file

*/\* This is a comment \*/*

*/\* My first program \*/*

*#include <stdio.h>*

*int main()*

*{*

*printf(“Hello! This is my first program\n”);*

*}*

Save firstprog.c and the exit **gedit.**

**SECTION 3:** Compiling C programs

Now we need to compile the program. To do this we need to use a C compiler.

**Gcc** is a C compiler that is available on most UNIX Systems.

Type: ***which gcc***  to find where it is located. Write down the result.

To compile your first program /user/bin/gcc

Type: ***gcc –o firstprog firstprog.c***

If there is an error, look carefully at your program and make sure what is on the sheet is EXACTLY what you have typed on the screen. If there are no reported errors, then continue.

Type: ***ls –al***

Write down the information on **firstprog.c** and **firstprog**.

-rw-r--r-- 1 mrworldend mrworldend 83 Nov 20 23:51 Myfirstprog.c

-rwxr-xr-x 1 mrworldend mrworldend 16048 Nov 20 23:56 firstprog

The first item of the result lists the different access rights for three groups: **user**, **group** and **other** so there are 9 fields with each group having 3 fields each. The access rights are: r = read access, w = write access and x = execute access.

Looking at the access rights what is the basic different between the source file, **firstprog.c** and the executable file, **firstprog.**

**SECTION 4**: Running programs

To run programs under UNIX you should put your program in a **bin** directory.

To find out where you need to put it you need to look at the PATH variable

Type: *printenv | more*

Write down the value to which your PATH variable has been set.

You should see that one of the directories on your PATH is:

**/home/guest/bin**

This is your very own **bin** directory. All your programs MUST be placed in **this** directory before the system can execute them. So let’s create it.

Type: ***cd*** to return to your home directory

Check first to see if a **bin** directory already exists.

Type: ***ls***

If a **bin** directory of file is present; remove it.

Type: ***rm –rf bin***

We now will make the bin directory:

Type: ***mkdir bin***

Move the executable into the **bin** directory.

From your **home directory**, type:

***mv src/firstprog bin/***

Now the system can find your file.

Type: ***which firstprog***

Write down the result

To run a program in UNIX you simply type the NAME of the program so

Type: ***firstprog***

Congratulations! You have run your first C program

# Writing a Hog Program

Go back to the **src** directory to write your second program by typing: ***cd src***

We want to write a program that attempts to take over the processor. This is called a **Hog** program. Let’s type in the second program

Type: ***gedit secondprog.c***

*/\* My second program counts items in blocks of 1 million \*/*

*#include <stdio.h>*

*int main()*

*{*

*int i = 0;*

*int nmillions;*

*for (;;) /\* Do forever \*/*

*{*

*if((i % 1000000) == 0)*

*{*

*nmillions = i / 1000000;*

*printf(“Now at %d million \n”, nmillions);*

*}*

*i++ ;*

*}*

*}*

Save this file and exit **gedit**.

Use **gcc** to compile your **secondprog** just as you did the first program

Move **secondprog** into your bin directory

From your **src** directory type:

***mv secondprog ../bin/***

Why do I have to use **../bin/** to move the file to the bin directory?

moves program to executable directory in the same folder

Before you do anything bring up another terminal window. To do this, in the current Terminal Window, ***click on File and then click Open.***

In the new terminal window:

Type: **top**

Write down the percentage of time the CPU is idle.

Now in the first window, type **secondprog** to run the program.

Look at **top** and write down the percentage of time the computer is now idle. If your computer has one processor or CPU, then the idle time should go to zero or be very small. If your computer uses more than one CPU, then the idle time will not go to zero. Explain why this is so. If you have more than one CPU, open a new window and run a second version of the Hog program. Then check the idle state again. Repeat until the idle time is zero. Explain how such a technique can be used to work out how many CPUs a computer is using.

<undone>

Write down the process id or PID of **secondprog.** If you have started more than one, then get the PID of each one. Exit the **top** program by typing *q* for quit

## Killing your program

The **Hog** program is in fact a very dangerous program and must be killed immediately. Open a third window. The **kill** command allows you to kill your programs. But you must specify the PID of the process you want to kill, and you must OWN the process.

Type: ***kill (followed by the PID of secondprog)***

Again, if you have you have more than one instance of **secondprog**, please kill all of them.

Try to kill the **init** program by typing **kill 1**. What does the system say about doing that?

**PART2: Using System calls**

**SECTION 1:** Using System Calls

UNIX allows you to call system services directly using **system calls**.

You can call these routines directly.

Make sure that you are in your **src** directory

Copy your **secondprog.c** into another file called **thirdprog.c**

Type: ***cp secondprog.c thirdprog.c***

Use **gedit** to change **thirdprog.c** to look like:

*/\* My third program counts items in blocks of 1 million*

*without trying to kill the processor \*/*

*#include <stdio.h>*

*#include <unistd.h>*

*int main()*

*{*

*int i = 0;*

*int nmillions;*

*for (;;) /\* Do forever \*/*

*{*

*if((i % 1000000) == 0)*

*{*

*nmillions = i / 1000000;*

*printf(“Now at %d million \n”, nmillions);*

*sleep(1); /\* add this line:* ***Note*** *the number one, 1, is placed between the brackets \*/*

*}*

*i++ ;*

*}*

*}*

Save the file and exit **gedit**.

Compile and run **thirdprog** using ***gcc*.** Remember to move it to your **bin** directory.

What does the **sleep** system call actually do?

Remove the **thirdprog** program using the **top** and **kill** commands as we did for **secondprog**.

# PART3: Investigating Program Interaction PART3 is divided into 3 sections:

1. Times and Timings
2. Measuring the effect of the **Hog** program
3. The Hog Program and Security

**SECTION 1:** Times and Timings

We want to investigate the effect of the **Hog** Program on other programs. Does the **Hog** program slow up other programs? If so, we want to quantify the damage. However, in order to do that, we need to be able to accurately time how much time a computer takes to do a given task. And in order to do this, we must know how to get the current time from a computer system. In Linux, there is a call named **gettimeofday w**hich is used to find out the current time.

To time how long a task takes, we need to find the difference between two timer readings. We therefore create a routine called **diff\_time\_func** that finds the difference between two timer structures. The function is given below:

int diff\_time\_func(struct timeval \*t1, struct timeval \*t2)

{

int diff\_time;

diff\_time = (int)(t1->tv\_sec - t2->tv\_sec) \* 1000000 + (int)(t1->tv\_usec - t2->tv\_usec);

if(diff\_time < 0)

printf("diff\_time is negative\n");

return diff\_time;

}

Answer the following questions:

a) This program uses the concept of **pointers**. What are pointers?

A variable that holds an adress in memory.

b) The program also casts various time elements into integers. Why does it do this?

To allow for mathematical operations to be used on them

c) What does having a negative value of **diff\_time** actually mean?

It is to represent that the second time structure is greater than the first/

So, we can now write a program that can time a task. In keeping with our counting theme, we want to measure how long the computer takes to count to 100,000,000. So let us use **gedit**.

Type**: gedit fourthprog.c**

/\* need to include the timer stuff \*/

#include <sys/types.h>

#include <sys/time.h>

#include <stdio.h>

#include <unistd.h>

/\* We need to put our diff\_time routine \*/

int diff\_time\_func(struct timeval \*t1, struct timeval \*t2)

{

int diff\_time;

diff\_time = (int)(t1->tv\_sec - t2->tv\_sec) \* 1000000 + (int)(t1->tv\_usec - t2->tv\_usec);

if(diff\_time < 0)

printf("diff\_time is negative\n");

return diff\_time;

}

/\* our main program finds out the time the computer takes to count up to 100 million

\*/

int main()

{

struct timeval start, stop;

int i = 0;

int nmillions = 0;

int time\_diff = 0;

gettimeofday(&start, 0); /\* start time \*/

for(nmillions = 0; nmillions < 1000; i++)

{

do {

i++;

}while((i % 1000000) != 0);

nmillions++;

printf("now at %d million\n", nmillions);

}

/\* we need to measure the time again \*/

gettimeofday(&stop, 0);

time\_diff = diff\_time\_func(&stop, &start);

printf("Time taken:%d microseconds\n", time\_diff);

}

Save the file and exit **gedit**.

Compile and run **fourthprog** using ***gcc*.** Remember to move it to your **bin** directory. Check that it does count really count to 100 million and then write down the time taken. We however have a slight problem. The printf statement that tells us which million we have reached slows down program considerably because it is making an I/O request in order to print the sentence on the screen. So we need to remove the printf function to find out the true time it takes to count up to 100,000,000.

So we need to edit the fourthprog again, so:

Type: **gedit fourthprog.c**

# Change the line with the printf from:

printf("now at %d million\n", nmillions);

to:

// printf("now at %d million\n", nmillions);

# Save and exit gedit. Recompile and rerun fourthprog using gcc. Remember to move it to the bin directory. Write down the new time taken. What does the // actually do? If you computer is has more than one CPU then you might see little change. Explain why that is so.

**SECTION 2:** Measuring the Effect of the **Hog** program

In order to accomplish this goal, we first have to measure the system with the **Hog** program. So run fourthprog ten times. Each time take the value of the time take find the average value of the ten times.

Now open another terminal window and run the **Hog** program. Leave it running. Now run the fourthprog ten times in the previous window. Note down each reading and find the average.

Now open yet another window and run the **Hog** program again. So there should 2 versions of the **Hog** program now running. Now run the fourthprog ten times in the first window. Note each reading and find the average value.

It is important that you run enough **Hog** programs to really affect the CPU cycles in the system. So, if you have a Dual core then run 3 **Hog** programs, if you have a Quad-Core run 5 **Hog** programs. For each additional Hog repeat the exercise above.

**Putting it all together:**

Draw the following graph: On the x-axis you have the number of instances of the Hogg program that were running at the same time. If you followed the instructions above you should have 0 – 3 or 5 as your values. On the Y-axis we have the time taken to run the fourthprog. So we should be able to get some points on the graph corresponding to the readings from the experiments you did.

See if you can use the graph (using extrapolation if necessary) to answer the following questions:

What number of **Hog** programs will cause the execution time of the fourthprog to double?

It takes between 3 and 4 instances to double the time to run hog program

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number of instances of hog program | 1 | 2 | 3 | 4 |
| Average speed | 2474973 | 2856449 | 3112733 | 6219492 |
|  | 2474937 | 2929478 | 3850769 | 6221615 |
|  | 2474946 | 2916570 | 4089718 | 6654601 |
|  | 2470780 | 2899939 | 4028108 | 6410957 |
|  | 2473230 | 2860827 | 4116874 | 5745244 |
|  | 2479416 | 2871226 | 4034873 | 6483614 |
|  | 2474465 | 2859714 | 4033955 | 6461859 |
|  | 2477785 | 2885989 | 4040453 | 6724697 |
|  | 2475591 | 2913684 | 4030587 | 6740923 |
|  | 2474776 | 2877527 | 4026014 | 5932644 |
| Average time | 2250082 | 2624673 | 3578553 | 5781423 |

**SECTION 3:** The **Hog** program and Security

As we have previously said, the **Hog** program is dangerous and so represents a security threat to your computer system. But what type of security is it.

Explore the following security threats; write down in detail what they do:

a) virus

A malware program capable of replicating/ reproducing itself, harming files or other programs on computer.

b) Trojan Horse

A program that appears desirable but contains something harmful.

c) Logical Bomb

Malware that is hidden until triggered when certain specific conditions are met.

d) worm

Malware subset of trojan hours that con propagate/self-replicate from one computer to another.

Which of the 4 threats does running several versions of the Hog program represent and why?

# End of LAB 1

**CLEANUP**

**Please delete all the files and then delete all the directories you have created during this Lab**

***Type: cd*** to go to your home directory

***Type: rm –rf src bin (explain what this does)***

**Please log out and take your logbook to the instructor who must sign it before you leave.**