ELE709
Laboratory Manual

Winter 2021 Edition

Y.C. Chen
Department of Electrical, Computer and Biomedical Engineering
Ryerson University

January 2021

ELE709
Real-Time Computer Control Systems
Laboratory Manual

Winter 2021 Edition

Y.C. Chen
Department of Electrical, Computer and Biomedical Engineering
Ryerson University

Contents

List of Figures

List of Tables

Introduction

Course Contents

This course deals with practical techniques for the specification, design and implementation of real-time computer control systems. Fundamental concepts in concurrent programming are also discussed in this course and reinforced through laboratory work.

Laboratory work involve writing concurrent programs in the C language using POSIX threads (pthreads).

Virtual Laboratory

As physical access to the Control System Laboratory in ENG413 is now impossible, the laboratory work will be carried out on a LINUX Virtual Machine (VM) instead. The VM as well as installation instructions can be found on the course web page.

System Requirements

In order to run the VM you will need the following:

1. A computer with at least 2GB of RAM.
2. A minimum of 40GB of free storage space on your computer (for installing the LINUX virtual machine, etc.) An external drive can be used if you run out of space on your internal drive.

3. The Oracle VirtualBox player, which is required to load and run the LINUX virtual machine. It can be also be downloaded through the course web page.

Rules and Regulations

The following rules and regulations are to be observed:

1. Students are required to work individually.
2. Laboratory work to be submitted are listed in the "What to Submit" section at the end of each experiment.
3. Submit all laboratory-related work directly to the teaching assistant for your lab section.
4. Written work to be submitted can be either hand-written or typed. Hand-written work must be neatly written and submitted as a **single** file in PDF format. You can take photos of your hand-written work and then convert them into a single PDF file using a scanner app on your smart phone.
5. All experiments must be completed within the allotted time. Late submissions are subjected to a penalty of 20% per day.

Experiment 1

C Review

Time to Complete: 1 session Marks: 5

Objectives

- To briefly review the C programming language.
- To become familiarized with the software development environment in the VM.

1.1 Pre-Lab Work

Go through the rest of this document, try out the provided program examples and answer the Exercises.

1.2 Getting Started

To begin, we will write a C program to print the words ‘Lab Number 1’ on the computer’s screen.

In C, a program for doing this simple task is

```
/* This is a comment. */

#include <stdio.h>

```

```
5.   int main()
6.   {
7.       printf("Lab Number 1\n");
8.   }
```

The line numbers above are for reference purpose only, they are not part of the program.

Line 1 is a comment. Any text enclosed by the pair '`/*`' and '`*/`' will be ignored by the compiler.

Line 3 instructs the C compiler to look for the header file `stdio.h` which contains definitions for some of the system functions. The angle brackets around `stdio.h` means it is a system header file so that the compiler can look it up in the default system directory. The programmer does not have to worry about supplying this header file.

Line 5 defines a function called `main()`. In C, `main` is a reserved word used for defining the main program. The keyword `int` is used to declare that the value returned by the function `main()` is an integer. We will look at how to define functions later on in more detail.

Lines 6 and 8 define the beginning and end of the function.

Line 7 calls the system function `printf()` to display the message 'Lab Number 1' on the screen. The sequence '`\n`' is C notation for the newline character, which when printed advances the cursor to the beginning of the next line.

Now, start up the VM and open up a terminal window by double clicking on the "`MATE Terminal`" icon. Go to the ELE709 directory and create a new directory "`lab1`" and make it your current directory, e.g.

```
cd
cd ele709
mkdir lab1
cd lab1
```

After that, create a file called 'test1.c' using an editor and enter the above program in it. Remember not to enter the line numbers. Save the file after you finished editing and compile the program as follow:

```
rcc test1.c
```

Do not forget the file extension '.c'. If you haven't made a mistake entering the program, the compiler should produce a file 'test1' in your directory. You can then execute the program by typing the name of the program: 'test1'. The message 'Lab Number 1' should now appear on the screen.

In general, a C program may have the following structure:

```
Include header files

Define constants

Define external functions

Define global variables

Define subroutines

Define main() program
```

Inclusion of the header files are done using the `#include` statement as shown in the above example. It is a common practice to place definitions for constants, declarations for global variables and external functions in a header file so that they can be included in a C program using an `#include` statement.

Constants can be defined using the `#define` statement. For example, the constants π and e can be defined as follows:

```
#define   PI   3.14159265358979
#define   E    2.71828182845905
```

Arithmetic expressions are also allowed in a `#define` statement. For example to define the fraction 2/3, we can use

```
#define   TWO_THIRD   (2.0/3.0)
```

It is important to remember that the `#define` statement simply defines a symbol (e.g. PI) to represent a particular string of characters (e.g. 3.14159265358979 for PI). Thereafter, the compiler will replace all occurrences of the symbol by the corresponding string. Thus the `#define` statement is equivalent to the `EQU` directive available in most assembler languages.

Exercise 1.1 A student uses the following `#define` statements in a program to define the constant one-half:

```
#define   ONE_THIRD   1.0/3.0
```

It is then used in the following statements:

```
double a;
a = 2.0/ONE_THIRD;
```

What do you think the value of `a` will be? Explain you answer.

1.3 Data Types and Operations

The basic data types in C are:

1. `char`: A single byte for storing one character.
2. `int`: An integer.
3. `float`: A single-precision floating-point number.
4. `double`: A double-precision floating-point number.

In addition, a number of *qualifiers* can be applied to the `int`, `float` or `double`:

1. `short int`: A short integer.
2. `long long int`: A long integer.
3. `unsigned int`: An unsigned integer.
4. `long double`: A long double-precision floating-point number.

The word `int` after the qualifiers are optional and can be omitted.

It is important to note that the number of bytes used to store the various types of data is not standard and is dependent on the C compiler. Let's now write a program to determine the size of the some common data types as implemented by the gcc C compiler.

```
#include <stdio.h>

int main()
```

```
{
    printf("The size of int is %d bytes\n", sizeof(int));
    printf("The size of short int is %d bytes\n", sizeof(short int));
    printf("The size of long long int is %d bytes\n", sizeof(long long int));
    printf("The size of float is %d bytes\n", sizeof(float));
    printf("The size of double is %d bytes\n", sizeof(double));
    printf("The size of long double is %d bytes\n", sizeof(long double));
}
```

The above program makes use of the system-provided C function`sizeof()` to determine the size of date types. The string '`%d`' in the `printf` statement is used to specify that the number to be printed is an integer. Some of the other format specifiers are summarized in Table 1.1

Table 1.1: Format Specifiers

Format Specifier	Data Type
`%d`	`int`
`%ld`	`long long int`
`%f`	`float`
`%lf`	`double`
`%c`	`char`
`%s`	character string

Exercise 1.2 Run the above program to determine the sizes of the following data types: `int`, `long long int`, `unsigned int` and `unsigned long long int`. Now, calculate the maximum and minimum values that can be represented using each of these data types. Present your answers as powers of 2.

The standard arithmetic operations can be performed in C using the operators in Table 1.2. In addition, C also supports the bitwise logical operations in Table 1.3. Note that while arithmetic operations can be performed on both integers and floating point numbers, bitwise logical operations can only be performed on integers.

Table 1.2: Arithmetic Operators

Operator	Arithmetic Operation	Example
`+`	addition	`a+b`
`-`	subtraction	`a-b`
`*`	multiplication	`a*b`
`/`	division	`a/b`

Table 1.3: Bitwise Logical Operators

Operator	Logical Operation	Example
`&`	bitwise AND	`a&b`
`|`	bitwise OR	`a|b`
`^`	bitwise exclusive OR	`a^b`
`<<`	logical shift left	`a<<4` (shift left by 4 bits)
`>>`	logical shift right	`a>>3` (shift right by 3 bits)
`~`	one's complement	`~a`

1.4 Variables and Control Structures

The names of variables in C are identified using alpha-numeric strings which must begin with a letter. The underscore character '`_`' can also be used. Upper and lower case characters are treated as different names. A traditional C practice is to identify constants using only upper case characters. A list of reserved keywords that cannot be used as names are given in Table 1.4.

A variable is defined by associating a name with a data type. For example, the following statement defines an integer variable '`count`':

```
int count;
```

More than one variables can be defined in a single statement:

```
int count, kount, qount;
```

Arrays can be defined as follows:

```
int theta[100]
```

The above statement defines an array called `theta` with 100 elements and each array element is of type `int`. Note that the array elements are indexed from 0. Thus, the elements of the array are `theta[0]` $\cdots$ `theta[99]`.

Let's write a program to compute the sum of an array.

Table 1.4: C Reserved Keywords

int	extern	else
char	register	for
float	typedef	do
double	static	while
struct	goto	switch
union	return	case
long	sizeof	default
short	break	entry
unsigned	continue	
auto	if	

```
#include <stdio.h>

int main()
{
   double sum, theta[100];
   int i;

   sum = 0.0;
   i = 0;
   while (i < 100) {
      sum = sum + theta[i];
      i = i + 1;
   }
   printf("The sum of the array is %lf\n", sum);
}
```

The above program uses a `while` loop. In general, a `while` loop has the following structure:

```
while (expression) {
  ...
  statements
  ...
}
```

In a `while` loop, `expression` is first evaluated. If it is non-zero, statements inside the loop (i.e. those enclosed with the brackets) are executed. `Expression` is then re-evaluated and the cycle continues until `expression` is evaluated to zero. Note that an expression which is true is evaluated to 1, and a false expression is evaluated to 0. Thus a simple way to create an infinite loop is:

```
while (1) {
    ...
    statements
    ...
}
```

Relational and equality operators are often used in expressions for decision making. The operators supported by C are summarized in Table 1.5. Logical operators are also

Table 1.5: Relational and Equality Operators

Operator	Description	Example
`<`	less than	`exp_1 < exp_2`
`>`	greater than	`exp_1 > exp_2`
`<=`	less than or equal to	`exp_1 <= exp_2`
`>=`	greater than or equal to	`exp_1 >= exp_2`
`==`	equal to	`exp_1 == exp_2`
`!=`	not equal to	`exp_1 != exp_2`

available in C for more complex decision making. These operators are summarized in Table 1.6. For example, we may use the following C statement to test whether a

Table 1.6: Logical Operators

Operator	Description	Example
`&&`	logical AND	`exp_1 && exp_2`
`||`	logical OR	`exp_1 || exp_2`

character `c` is ‘`a`’ or ‘`b`’:

```
if ( (c == 'a') || (c == 'b') ) {
    ...
}
```

Exercise 1.3 The previous program for computing the sum of an array may not produce a correct result since the elements of the array have not been properly initialized. Modify the program so that the array elements have values $1, 2, \ldots, 100$.

An alternate way to write the previous program is to replace the `while` loop with a `for` loop:

```
for (i = 0; i < 100; i = i + 1) {
   sum = sum + theta[i];
}
```

A `for` loop has the following structure:

```
for (exp_1; exp_2; exp_3) {
   ...
   statements
   ...
}
```

which is equivalent to the following `while` loop structure:

```
exp_1;
while (exp_2) {
   ...
   statements
   ...
   exp_3;
}
```

It is possible to break out of a `while` or `for` loop before the terminate condition is satisfied through the use of a `break` statement. For example,

```
while (exp_1) {              for (...; exp_1; ...) {
   ...                          ...
   statements                   statements
   ...                          ...
   if (exp_2) break;            if (exp_2) break;
   ...                          ...
}                            }
```

will cause a pre-mature exit from the `while` loop if `exp_2` is evaluated to a nonzero value even when `exp_1` is nonzero.

Decision making in a C program is often implemented using the `if-else` and `else-if` statements. Formally, the syntax of a `if-else` statement is:

```
if (exp_1) {
   statements to be executed if exp_1 is true.
}
else {
   statements to be executed if exp_1 is not true.
}
```

where the `else` part is optional. Multiple decisions can be made using a `if-else` and `else-if` construction:

```
if (exp_1) {
   statements to be executed if exp_1 is true.
}
else if (exp_2) {
   statements to be executed if exp_2 is true.
}
else if (exp_3) {
   statements to be executed if exp_3 is true.
}
...
else {
   statements to be executed if none of the above is true.
}
```

Note that the `else` part at the end is often used to handle the default case and can be omitted if no default action is required. However, it is a good programming practice to have a default case for error checking.

Another useful control structure is the `switch-case` statement. For example,

```
int selection;
...
switch (selection) {
   case 'a':
      statements_a;
      break;
   case 'b':
   case 'c':
      statements_bc;
      break;
   case 'd':
      statements_d;
      break;
   default:
      statements_e;
      break;
}
```

will execute `statements_a`, `statements_bc`, `statements_d` if the value of `selection` is equal to '`a`', '`b`' or '`c`', and '`d`' respectively. `Statements_e` will be executed if `selection` has other values. The `default` case is optional; no action will take place if none of the cases matches when the `default` is omitted. The `break` statement causes an immediate exit from the `switch` statement; otherwise, execution will continue to the next case.

1.5 Subroutines and Scope of Variables

Subroutines are known as *functions* in C. By defaults, C assumes that every function has a return value. The following example shows how to define and use a function `square()` for computing the square of a number.

```
#include <stdio.h>

double square(double x)
{
   double x_squared;

   x_squared = x*x;
   return x_squared;
}

int main()
{
   double a, a_squared;

15.      a = 2.5;
16.      a_squared = square(a);
17.
18.      printf("The square of %lf is %lf\n", a, a_squared);
19.   }
```

Lines 3 to 9 define the function `square()`. Line 3 defines the name and data type returned by the function. It also defines the parameters required by the function. In this case, the name of the function is `square` and its return value (i.e. the result) is of type `double`. There is only one parameter `x` (the number to be squared) and its data type is `double`. Lines 4 and 9 define the beginning and end of the function respectively. The body of a C function is made up of any valid C statements. However, the result obtained by the function must be returned by the `return` statement as indicated in Line 8. The `return` statement can be placed anywhere in a function. It is not necessary for it to be the last statement of a function.

By default, the parameters of a C functions are passed by their *values* not by their *address*. A called function has a local, temporary copy of the values of its parameters. Within a function, each of the parameters is in effect a *local* variable initialized with the value with which the function is called. As a result, changes made to a parameter inside a function have no effects on the corresponding variable outside the function. Let's look at the following program:

```
#include <stdio.h>

void square(double x, double x_squared)
{
   x_squared = x*x;
}

int main()
{
   double a, a_squared;

   a = 2.5;

   square(a, a_squared);

   printf("The square of %lf is %lf\n", a, a_squared);
}
```

The above program will not produce the desire result because only the *values* of the variables ‘a’ and ‘a_squared’ are passed to the function square. Changes in the variable x_squared have no effect on the value of a_squared in main() because its value is local to the function square().

C also allows parameters to be passed by their addresses. Thus, the previous program will work correctly if the result of the function square() is passed by its address.

```
 1.   #include <stdio.h>
 2.
 3.   void square(double x, double *x_squared)
 4.   {
 5.      *x_squared = x*x;
 6.   }
 7.
 8.   int main()
 9.   {
10.      double a, a_squared;
11.
12.      a = 2.5;
13.
14.      square(a, &a_squared);
15.
16.      printf("The square of %lf is %lf\n", a, a_squared);
17.   }
```

The above program makes use of two unary operators * and &. The unary operator & gives the address of its operand. Thus, in Line 14 the *address* (rather than *value*) of the variable a_squared is passed to the function square(). In order for this to work the function square() must be modified. This is done by declaring x_squared a pointer using the * operator in Line 3. The declaration ‘double *x_squared’ specifies that x_squared is a *pointer* to a data of type double. In addition, the unary operator * treats its operand as the address of a variable and accesses that address to obtain its contents. Therefore the effect of Line 5 is to replace the *content* of the address pointed to by x_squared (which is the same as the address of a_squared) with the result of x*x.

It is important to remember that *arrays are passed by their addresses in functions.* In fact, the name of an array can be regarded as a pointer whose value is the address of the first element of the array. The following program demonstrates how arrays are passed in functions:

```
#include <stdio.h>

double sum(double *data, int no_of_data)
{
   double s;
   int i;

   s = 0.0;
   for (i = 0; i < no_of_data; i = i + 1) {
      s = s + data[i];
   }
   return s;
}

int main()
{
   double theta[100], result;
   ...
   result = sum(theta, 100);
   ...
}
```

Variables defined within a function are only accessible inside that function, no other functions can have access to them. For example, the variables `s` and `i` in the function `sum()` are not accessible by the function `main()`. These variables are known as *local variables*. Since the scope of a local variable is limited to the function in which it is defined, the same name can be used in different functions to represent different variables. Local variables come into existence only when the function in which they are defined are called, they disappear as soon as the function is exited.

It is also possible to define variables that are accessible by *all* functions. These variables are known as *global variables*. Global variables are defined *externally* to all functions as follows:

```
#include <stdio.h>

int ga;
float gb[10];
float sub_1(...)
{
  ...
}

int sub_2(...)
{
   ...
}

int main()
{
   ...
}
```

The above example has two global variables `ga` and `gb`. These two variables are accessible by *any* of the functions `sub_1()`, `sub_2` and `main()`. Because global variables are accessible by all functions, they cannot appear as parameters of functions. Furthermore, global variables remain in existence permanently until the program terminates, rather than appearing and disappearing as functions are called and exited.

It may be tempting to declare all variables global because it appears to simplify communications between functions — there is no longer a need to pass parameters between functions. This is dangerous and undesirable since it leads to programs which are difficult to debug — changes in variables may be unexpected and hard to trace. Care must be taken to determine whether a particular variable should be made global or not.

1.6 Input and Output

We have already seen how a C program can perform output using the `printf()` function. The C library also provides the following functions for data input:

1. `scanf()`: for reading data according to the user's provided format.

2. `getchar()`: for reading a character.

The `scanf()` function can be used to read all kinds of data, e.g. `int`, `float`, `char`, etc. The data to be read in is controlled using the format specifiers in Table 1.1. Let's modify the program for computing the square of a number so that it now prompts the user for the number to be squared. We'll only show the `main()` function since it is the only one required to be modified.

```
 1.   int main()
 2.   {
 3.      double a, a_squared;
 4.
 5.      printf("Enter the number to be squared ");
 6.      scanf("%lf", &a);
 7.
 8.      a_squared = square(a);
 9.
10.      printf("The square of %lf is %lf\n", a, a_squared);
11.   }
```

Note that the `&` operator is used in Line 6 because all parameters are passed by addresses in the `scanf()` function.

Several values can be read in using a single `scanf()` statement. For example,

```
int a;
double b;
float c;
long d;
...
scanf("%d %lf %f %ld", &a, &b, &c, &d);
...
```

It is important to match the format specifiers with the types of the data being read in; otherwise erroneous values will be obtained.

The `getchar()` function is used to read in only one character. This function does not require any parameter and returns the ASCII code of the character read.

1.7 Structures

The C programming language provides a convenient way for organizing logically related items using *structures*. One can think of structures as generalizations of arrays. In an array, all the elements must be of the same data type. However, the elements of a structure can consist of any combination of basic or other user-defined data types.

As an example, suppose that we wish to create a record of ELE709 marks for a student “John Doe”. Assume that there are 3 experiments, 1 test and 1 final examination. Then, we can create the following structure to record the student number, and the marks for the experiments, test and final examination for him:

```
struct lab {
   double experiment1;
   double experiment2;
   double experiment3;
};

struct theory {
   double test;
   double final;
};

struct ele709_record {
   long ID_number;
   struct lab lab_mark;
   struct theory theory_mark;
};
```

The above C statements use the keyword `struct` to define a structure called `ele709_record` with 3 *members*: `ID_number`, `lab_mark` and `theory_mark`. The members `lab_mark` and `theory_mark` are themselves structures with 3 and 2 members respectively.

A member of a structure can be referred to using the *structure member operator* “.”. For example, we can use the following C statements to record the various marks for student John Doe.

```
struct ele709_record john_doe;

john_doe.ID_number = 12345678;
john_doe.lab_mark.experiment1 = 90.2;
john_doe.lab_mark.experiment2 = 70.5;
john_doe.lab_mark.experiment3 = 80.4;
john_doe.theory_mark.test = 82.3;
john_doe.theory_mark.final = 79.2;
```

The operations that can be performed on structures are restricted. Basically, the only operations that can be performed on a structure are accessing one of its members using the "." operator, and taking its address using the "&" operator. This means that structures can not be assigned to or copied *as a single unit*, and that they can not be passed to or returned from functions. However, it is important to note that *pointers* to structures do not have these restrictions.

The following C statements show an alternative way to record John Doe's marks using a *pointer* to the structure `john_doe`:

```
struct ele709_record john_doe;
struct ele709_record *p;

p = &john_doe;

p->ID_number = 12345678;
p->lab_mark.experiment1 = 90.2;
p->lab_mark.experiment2 = 70.5;
p->lab_mark.experiment3 = 80.4;
p->theory_mark.test = 82.3;
p->theory_mark.final = 79.2;
```

In addition to the predefined data types, the C language also provides a way for creating new data types using the `typedef` statement. For example, we can create a new data type called `ele709_record_t` for the structure `ele709_record` defined earlier as follow:

```
typedef struct ele709_record ele709_record_t;
```

After that, we can declare the records for student John Doe, and other students as follow:

```
ele709_record_t john_doe, jane_doe, tom_doe, mary_doe;
```

Exercise 1.4 Write a function to return the total ele709 mark for a student. The function must be of the following form: `double total_mark(struct ele709_record *p)`
where `p` is a pointer to an ele709 student record as defined earlier. Assume that the 3 labs have equal weightings and the various marks contribute towards the total mark in the following way: total lab mark (20%), theory test mark (30%) and theory final mark (50%). Test your function using the record of `john_doe` presented earlier. The correct answer should be 80.36. A possible psuedo code is given below.

```
include required header files ...
struct lab ....
struct theory ...
struct ele709_record ...

double total_mark(struct ele709_record *p)
{
    enter code for the function here ...
}

int main()
{
   struct ele709_record john_doe;
   enter john doe record here  ...
   john_doe_mark = total_mark(&john_doe);
   print out john_doe_mark
}
```

1.8 What to Submit

A report is not required. Archive the code for Exercise 1.4 and the completed answer sheet for this experiment into a .tar file (use your last name as the name of the tar file) and email it as an attachment to your TA at the end of the laboratory session. For example,

```
tar cvf lastname-lab1.tar ex1.4.c lab1ans.pdf
```

1.9 Reference

1. "The C Programming Language," Second Edition, B.W. Kernighan and D.M. Ritchie, Prentice Hall, 1988.

Experiment 2

Time and Clock

Time to Complete: 1 session Marks: 5

Objectives

- To learn how to do time measurement.
- To learn about the system clock and its resolution.
- To demonstrate that time cannot be determined with absolute certainty.

2.1 Pre-Lab Work

1. Logon to D2L, go to Content/Documentation on POSIX Functions, and familiarize yourself with the following POSIX functions by going over their manual pages: `clock_gettime()`, `clock_getres()` and `nanosleep()`.
2. Go over the rest of this document and prepare all the required programs before going to the Laboratory.

2.2 Lab Work

A real-time system must satisfy, in addition to its functional requirements, timing requirements. In order to write software applications for real-time systems one must understand the concept of time measurement, timers and their limitations.

It is important to realize that time cannot be measured with absolute certainty on a digital computer because of the following reasons:

1. Time is measured using something called *clock ticks* on a computer. The value of a clock tick defines the *resolution* or accuracy of the timing information.

2. Timing information is stored using a finite number of bits.

3. Timing information is obtained by making a function call. This will have an effect on the time that is being measured, especially on a multitasking operating system.

In the following sections, we will demonstrate how to measure time and the uncertainty involved using a simple program partly described by the following pseudo code:

```
Get the current time and store it in start_time.
Sleep for 20 msec.
Get the current time and store it in stop_time.
Determine elapsed_time = stop_time - start_time.
```

As will be seen later, the value of `elapsed_time` is not exactly 20 msec.

2.3 POSIX Time Measurement

POSIX stands for Portable Operating System Interface for UNIX. Programs conforming to the POSIX standard are portable across computers that support this standard. All the LINUX workstations on the Department's network support the POSIX standard.

The following program (available as `lab2a.c` on the course web page) is a POSIX implementation of the above pseudo code. This program uses the function `clock_gettime()` to determine time from the real-time clock `CLOCK_REALTIME`. The timing information returned is specified using the structure `timespec` which has 2 members, `tv_sec` and `tv_nsec`, representing the time elapsed since the Epoch (i.e. January 1, 1970 at 00:00h). It also uses the function `clock_getres()` to determine the resolution of the clock, and the function `nanosleep()` to sleep for a user-specified amount of time. The resolution of the `CLOCK_REALTIME` varies between 1/50 to 1/250 sec, depending on the system. Some POSIX implementations support clocks with higher resolution.

```
#include <stdio.h>
#include <time.h>
```

```
int main()
{
   int i, itr;
   struct timespec start_time, stop_time, sleep_time, res;
   double tmp, elapsed_time[10];

   clock_getres(CLOCK_REALTIME, &res);
   printf("The clock resolution of CLOCK %d is %d sec and %d nsec\n",
         CLOCK_REALTIME, res.tv_sec, res.tv_nsec);

   sleep_time.tv_sec = 0;
   sleep_time.tv_nsec = 20000000;
   printf("Sleep time requested is %ld sec  %ld nsec\n",
         sleep_time.tv_sec, sleep_time.tv_nsec);

   itr = 10;
   for(i = 0; i < itr; i++)
   {
      clock_gettime(CLOCK_REALTIME, &start_time);
      nanosleep(&sleep_time,NULL);
      clock_gettime(CLOCK_REALTIME, &stop_time);

      tmp = (stop_time.tv_sec - start_time.tv_sec)*1e9;
      tmp  += (stop_time.tv_nsec - start_time.tv_nsec);
      elapsed_time[i] = tmp;
   }

   for(i = 0; i < itr; i++)
   {
      printf("Iteration %2d ... Slept for %lf nsec\n", i, elapsed_time[i]);
}

   return 0;
}
```

Exercise 2.1 Compile the program `lab2a.c` using `rcc` and execute it. Are there any difference between the requested sleep time (20 msec) and the "actual" sleep time? What do you think caused the difference?

2.4 Extension

Exercise 2.2 Write a C program to *estimate* how fast the computer in the Control Systems Laboratory (ENG413) performs the four basic mathematical operations (+, −, ×, ÷). Assume that the operations are of the form: `c = a <op> b`, where `<op>` is one of the four basic mathematical operations and `a`, `b` and `c` are *variables* of type `double`. The pseudo code for such a program may look like the following:

1. Set the number of iterations: `maxitr = 500000000`.
2. Prompt user for the numbers a and b, and the mathematical operation to be timed.
3. Determine and store the current time in `TIME1`.
4. Repeat the selected mathematical operation `maxitr` times. For example, for the "+" operation, use
   `for (i = 0; i < maxitr; i++) c = a + b;`
5. Determine and store the current time in `TIME2`.
6. Calculate the elapsed time for performing the selected mathematical operation `maxitr` times using: `ELAPSED_TIME = TIME2 - TIME1`.
7. Calculate the time per iteration using `ELAPSED_TIME/maxitr`.
8. Print the time per iteration in nanoseconds.
9. Return to Step 2.

Implement the above pseudo code using the POSIX clock function `clock_gettime`. You may use `lab2a.c` as a template. Use `a` = 2.3 and `b` = 4.5 to test your program. Run the programs and record the results. Note that the above pseudo code assumes that the time required to execute the `for`-loop is negligible and is therefore not included in the calculation of `ELAPSED_TIME`. Develop and test your program using the VM first. After that remotely login to one of the workstations in ENG413, upload your program to the EE network, recompile and run it to obtain the timing results again for `lab2ans.pdf`. Instructions on remote access to ENG413 can be found on D2L.

2.5 What to Submit

A report is not required. Archive the code for Exercise 2.2 and the completed answer sheet for this experiment into a .tar file (use your last name as the name of the tar file) and email it as an attachment to your TA at the end of the laboratory session. For example,

```
tar cvf lastname-lab2.tar ex2.2.c lab2ans.pdf
```

Experiment 3

POSIX Threads and Concurrent Programming

Time to Complete: 2 sessions Marks: 20

Objectives

- To introduce concurrent programming with POSIX threads (Pthreads).
- To see the effect of time-sharing on the execution time of a program.
- To apply concurrent programming in a practical application.

3.1 Pre-Lab Work

1. Familiarize yourself with the following Pthreads functions by reading their manual pages available on the course web page: `pthread_create()`, `pthread_exit()` and `pthread_join()`.
2. Go over the rest of this document and prepare all the required programs before going to the Laboratory.

3.2 Lab Work

The programs for this Experiment can be developed and tested on the VM. However, the final timing results should be obtained by running your programs remotely on a workstation in ENG413 in order to have results consistent with those in the marking scheme.

3.2.1 Elementary Thread Programming

Obtain the following example program (lab3.c) from the course web page and save it in a file:

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

struct thread_info {
   int maxitr;
   double exec_time;
};

typedef struct thread_info thread_info_t;

void *func(void *arg)
{
   struct timespec time_1, time_2;
   double exec_time;
   thread_info_t *info;
   int i, maxitr;
   volatile double a, b, c;

   info = (thread_info_t *)arg;
   maxitr = info->maxitr;

   b = 2.3; c = 4.5;

   exec_time = 0.0;

   clock_gettime(CLOCK_REALTIME, &time_1);

   for (i = 0; i < maxitr ; i++) {
         a = b*b + b*c + c*c;
   }

   clock_gettime(CLOCK_REALTIME, &time_2);

   exec_time = (time_2.tv_sec - time_1.tv_sec);
   exec_time = exec_time + (time_2.tv_nsec - time_1.tv_nsec)/1.0e9;

   info->exec_time = exec_time;

   pthread_exit(NULL);

}

int main(void)
{
   pthread_t thread1;
   thread_info_t info1;
   double maxitr;

   maxitr = 5.0e8;

   info1.maxitr = (int)maxitr;

   if (pthread_create(&thread1, NULL, &func, &info1) != 0) {
           printf("Error in creating thread 1\n");
           exit(1);
   }

   pthread_join(thread1, NULL);
   printf("Exec time for thread1 = %lf sec\n", info1.exec_time);

   pthread_exit(NULL);
}
```

The above program demonstrates how a POSIX thread (Pthread) can be created, and how a thread function and its arguments can be passed to a thread. Let's now look at this program in more details:

Lines 1-4 instruct the compiler to include the header files required by this program.

Lines 6-11 define the data type `thread_info_t` which is used to declare the data structure `info1` on Line 48. The data structure `info1` is used to pass data to and from the thread function `func()`.

Lines 13-43 define the thread function `func()` to be executed by the Pthread `thread1` (more on this later). Basically, what `func()` does is to determine the execution time (in seconds) of the `for`-loop in Lines 30-32 using the `clock_gettime()` function. The number of iterations, `maxitr`, is obtained through the thread info data structure `info1`, and the execution time is passed back to the main program `main()` through the same data structure. Line 41 calls the Pthread function `pthread_exit()` to terminate the thread.

Lines 45-64 define the main program `main()`. Line 47 declares a Pthread `thread1` using the pre-defined data type `pthread_t` (defined in the header file `pthreads.h`). Lines 51-53 initialize the number of iterations `maxitr` and store it in the thread info data structure `info1`. Line 55 calls the POSIX library function `pthread_create()` to create a new Pthread `thread1` to be executed concurrently with the calling thread. The first argument of `pthread_create()` is a pointer to the new thread (in our case, `thread1`). The second argument of `pthread_create()` is used to pass *attributes* to the new thread (in our case, it is not used and is set to `NULL`). The third argument is a pointer to the function that is to be executed by the new thread, i.e. the thread function (in our case, it is `func()`). The fourth and last argument is a pointer to the arguments of the thread function (in our case, it is a pointer to the arguments of the function `func()`, i.e. `info1`).

After calling `pthread_create()`, the main program simply waits until the thread function finishes executing by calling the Pthread function `pthread_join()` in Line 60 and then prints out the execution time obtained by the thread function. If the `pthread_join()` call is removed, then the `printf()` function in Line 61 will be called immediately and the execution time printed may be incorrect since the thread function may still be executing.

Finally, it is important to note that there is no synchronization between the creating thread's return from `pthread_create()` and the scheduling of the new thread. In other words, the new thread may start executing before `pthread_create()` returns. The new thread may even run to completion and terminate before `pthread_create()` returns.

Exercise 3.1 Compile and execute the example program (lab3.c) and record the execution time. Run it a few times to see if there is any significant difference in the execution times.

Exercise 3.2 Modify the example program (lab3.c) to estimate the execution time for the four basic arithmetic operations ($+$, $-$, $\times$, $\div$) as in Experiment 2 as follow:

- The program should use the POSIX function `clock_gettime()` to get timing information.
- The program should use 4 Pthreads, with one Pthread for each of the arithmetic operations,
- Each of the 4 Pthreads should call the *same* thread function. The data structure shown in Figure 3.1 should be used to pass information (the numbers: `a`, `b`; the operation to be performed: `op`; and the number of iterations: `maxitr`) to the thread function, and to return results (the number: `c = a <op> b`; and the execution time per iteration in nanoseconds: `exec_time`) to `main()` .
- The 4 Pthreads should run *sequentially*, i.e. one after another.

Run the modified program 3 times and record the results in Table 3.1. Are the execution time obtained similar to those obtained in Lab 2?

```
struct thread_info {
   double a, b;         // The numbers a and b
   char op;             // The math iteration to be performed: +,-,*,/
   int maxitr;          // The number of iterations to be performed
   double c;            // The result: c = a <op> b
   double exec_time;    // The execution time per iteration in nsec
};
```

Figure 3.1:

Exercise 3.3 Modify the program in Exercise 3.2 so that the 4 Pthreads now run *concurrently*. Run the modified program 3 times and record the results in Table 3.1. Are the execution time obtained similar to those obtained in Exercise 3.2. Explain why (or why not). (Hints: All the workstations in ENG413 have quad-core processors.)

Table 3.1: Execution Time per Operation

Exercise 3.2					Exercise 3.3				
	$+$	$-$	$\times$	$\div$		$+$	$-$	$\times$	$\div$
1					1				
2					2				
3					3				

3.2.2 Effect of Time Sharing

Obtain the following program (load.c) from the course web page and save it in a file:

```
#include <stdio.h>
#include <math.h>
#include <sys/mman.h>
#include <pthread.h>
#include <malloc.h>
#include <stdlib.h>

#define NTHREADS 3

#define SIZE 1000000

void *func(void *arg)
{
   int i, j, rcode;
   double *a;
   for (j = 0; ; j++) {
    a = (double *)calloc(SIZE, sizeof(double));
      for (i = 0; i < SIZE; i++) {
         a[i] = sqrt((double) i);
      }
      free(a);
   }

}

int main(int argc, char *argv[])
{
   pthread_t loads[NTHREADS];
   int no_of_threads;
   int i;

   no_of_threads = NTHREADS;

   printf("Created %d threads to simulate load on the system.\n", no_of_threads);

   for (i = 0; i < no_of_threads; i++) {
      if (pthread_create(&loads[i], NULL, &func, &i) != 0) {
         printf("Error creating thread %d\n", i);
         exit(-1);
      }
      else {
         printf("Created Thread %d\n", i);
      }
   }

   printf("\nRunning ...\n");
   printf("Use Ctrl-C to stop.\n");

   pthread_exit(NULL);
}
```

The above program creates 3 threads that execute the thread function `func()` indefinitely. It is simply used to simulate additional load on the CPU.

Exercise 3.4 Compile and run the program load.c. Now, open another terminal window and run the programs in Exercises 3.2 and 3.3 again and record the execution times in Table 3.2. Compare the results obtained with those from Exercises 3.2 and 3.3. Explain any discrepancies. (Hints: All the workstations in ENG413 have quad-core processors.)

Table 3.2: Execution Time per Operation (With Load)

Exercise 3.2 (With Load)					Exercise 3.3 (With Load)				
	$+$	$-$	$\times$	$\div$		$+$	$-$	$\times$	$\div$
1					1				
2					2				
3					3				

3.3 An Application of Concurrent Programming

Consider a $r_A \times c_A$ matrix A and a $r_B \times c_B$ matrix B, with $c_A = r_B$. Let $C = A \times B$, and C_{ij} be the entry in the ith row and jth column of C, then

$$C_{ij} = \sum_{k=1}^{c_A} A_{ik} B_{kj}. \tag{3.1}$$

As can be seen from the above equation, the computation of C_{ij} depends only on the matrices A and B. Hence, the entries of the matrix C can be computed independently of each other. In other words, they can be computed concurrently. This will speed up the computation significantly in a multiprocessor system, especially if the sizes of the matrices involved are very large.

Exercise 3.5 Write a program to perform multiplication of 2 matrices *concurrently* as described above. The program

- Must use a function `compute_C_ij()` that implements Eq. (3.1) to calculate C_{ij}.
- Must use $r_A \times c_B$ Pthreads to compute the product matrix C concurrently, with one Pthread responsible for computing one entry C_{ij} of the matrix C. Furthermore, *all* Pthreads should use the *same* thread function `compute_C_ij()` to perform the calculation.
- May declare the matrices A, B and C as global variables to simplify programming.

Test your program with a 18×16 matrix A such that $A_{ij} = i + j$, $(i = 1, \cdots, 18;$ $j = 1, \cdots, 16)$ and a 16×18 matrix B such that $B_{ij} = i + 2j$, $(i = 1, \cdots, 16;$ $j = 1, \cdots, 18)$. As a check, verify that the resulting matrix C has the following diagonal elements: 1936, 2440, 3008, 3640, 4336, 5096, 5920, 6808, 7760, 8776, 9856, 11000, 12208, 13480, 14816, 16216, 17680 and 19208.

3.4 What to Submit

A report is not required; however, the following are to be submitted:

- **Week 1:** The program code for Exercises 3.2 and 3.3, and the answer sheet `lab3ans1.pdf`.
- **Week 2:** The program code for Exercises 3.5 and the answer sheet `lab3ans2.pdf`.

Archive the program code and the completed answer sheet for this experiment into a .tar file (use your last name as the name of the tar file) and email it as an attachment to your TA at the end of the laboratory session. For example,

```
tar cvf lastname-lab3week1.tar ex3.2.c ex3.3.c lab3ans1.pdf
tar cvf lastname-lab3week2.tar ex3.5.c lab3ans2.pdf
```

Experiment 4

Resource Sharing and Coordination

Time to Complete: 1 session Marks: 10

Objectives

- To understand how to share a common resource.
- To understand how a common resource can be protected using a mutex.
- To understand how to coordinate tasks sharing a resource using a condition variable.

4.1 Pre-Lab Work

1. Logon to D2L, go to Content/Documentation on POSIX Functions, and familiarize yourself with the following POSIX functions by going over their manual pages: `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `pthread_cond_wait()`, and `pthread_cond_signal()`.
2. Go over the rest of this document and prepare all the required programs before coming to the Laboratory.

4.2 Lab Work

This Lab can be done entirely on the VM.

4.2.1 Resource Sharing

This part of the Lab demonstrates how a common resource can be shared among different threads. In this case, three threads will attempt to cooperatively write a character string repeatedly to the screen.

The character string to be written onto the screen is “0123456789”. Each thread will take turn and write one character of this string to the computer screen. A global variable `string_index` (see the example program below) is used to keep track of current character to be written. After all 10 characters have been printed, the variable `string_index` is reset to 0 to start over again. Therefore, if the program is written correctly, the following will appear on the computer screen:

```
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
..........
```

As a first attempt, consider the following program (lab4.c, available through the course web page):

```
#include <stdio.h>       /* for fprintf() */
#include <stdlib.h>      /* for exit() */
#include <unistd.h>      /* for sleep() */
#include <pthread.h>     /* for pthreads functions */

#define NTHREADS 3

int string_index = 0;

char string_to_print[] = "0123456789";

void *func(void *arg)
{
   int tmp_index;

   sleep(1);   /* sleep 1 sec to allow all threads to start */

   while (1) {
      tmp_index = string_index;

      if (!(tmp_index+1 > sizeof(string_to_print))) {
         printf("%c", string_to_print[tmp_index]);
         usleep(1);   /* sleep 1 usec */
      }


      string_index = tmp_index + 1;

      if (tmp_index+1 > sizeof(string_to_print)) {
         printf("\n");
         string_index = 0;   /* wrap around */
      }
   }
}


int main(void)
{
   pthread_t threads[NTHREADS];
   int k;

   for (k = 0; k < NTHREADS; k++) {
      printf("Starting Thread %d\n", k+1);
      if (pthread_create(&threads[k], NULL, &func, NULL) != 0) {
            printf("Error creating thread %d\n", k+1);
            exit(-1);
      }
   }

   sleep(20);   /* sleep 20 secs to allow time for the threads to run */
                /* before terminating them with pthread_cancel()      */
   for (k = 0; k < NTHREADS; k++) {
      pthread_cancel(threads[k]);
   }

   pthread_exit(NULL);
}
```

Exercise 4.1

(a) Compile and execute the example program (lab4.c). Study the program and its results to explain why it didn't work correctly.

(b) Modify the example program (lab4.c) to use a mutex so that the correct results are produced.

4.2.2 Resource Sharing and Coordination

In many applications, there are situations in which one thread needs to notify another thread on a change in the status of a shared resource protected by a mutex. This can be done with the help of a condition variable.

In this part of the Lab, you are required to modify the example program (lab4.c) to coordinate 2 threads (A and B) to write the character string '0123456789' to the screen as described below.

Exercise 4.2 Modify the example program (lab4.c) to use *a mutex and a condition variable* to print the character string '0123456789' using 2 threads A and B, with Thread A printing characters '0', '2', '4', '6', '8', and Thread B printing characters '1', '3', '5', '7', '9'. You must use 2 *separate* thread functions for the threads, and modify the format control string `"%c"` in the `printf()` statement in the thread function of the example program to `"A%c "` for Thread A, and `"B%c "` for Thread B. If the program works correctly, the string "`A0 A1 A2 A3 A4 B5 B6 B7 B8 B9`" will appear repeatedly on the screen. Your program must use *one and only one* mutex and condition variable respectively.

4.3 What to Submit

A report is not required. Archive the code for Exercise 4.2 and the completed answer sheet for this experiment into a .tar file (use your last name as the name of the tar file) and email it as an attachment to your TA at the end of the laboratory session. For example,

```
tar cvf lastname-lab4.tar ex4.2.c lab4ans.pdf
```

Experiment 5

Task Synchronization and Communication

Time to Complete: 2 sessions Marks: 20

Objectives

- To understand how to do task synchronization with semaphores.
- To understand how to perform data communication and task synchronization using message queues.

5.1 Pre-Lab Work

1. Logon to D2L, go to Content/Documentation on POSIX Functions, and familiarize yourself with the following POSIX functions by going over their manual pages: `mq_open()`, `mq_send()`; `mq_receive()`, `mq_getattr()`, `mq_close()`, `mq_unlink()`, `perror()`, `sem_init()`, `sem_post()` and `sem_wait()`.
2. Go over the rest of this document and prepare all the required programs before coming to the Laboratory.

5.2 Lab Work

This Lab can be done entirely on the VM.

Non-Interlocked, One-Way Data Communication

The following program (available as lab5.c through the course web page) demonstrates the "Non-Interlocked, One-Way Data Communication" scheme as shown in Figure 5.1.

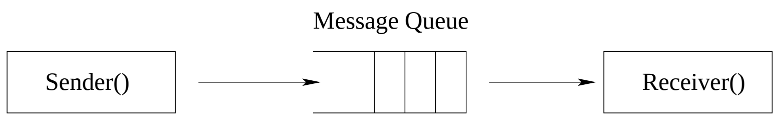

Figure 5.1: Non-Interlocked, One-Way Data Communication

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <mqueue.h>
#include <errno.h>

struct Msg {char str[20]; unsigned int priority; };
int errno;

const char *qname = "/myque";  // Define the name of the MQ, use your
                               // last name as part of the name to create
                               // a unique name.

void *Receiver(void *arg)  // This thread receives msgs from the MQ
{
   mqd_t rx;
   struct mq_attr my_attrs;
   size_t n;
   int nMsgs;
   char inBuf[50];
   unsigned int priority;

   sleep(2);

   rx = mq_open(qname, O_RDONLY);  // Open the MQ for reading only (O_RDONLY)

   if (rx < 0 ) perror("Receiver mq_open:");  // Error checking

   mq_getattr(rx, &my_attrs);     // Determine no. of msgs currently in the MQ
   nMsgs = my_attrs.mq_curmsgs;

   printf("Receiver found %d messages\n", nMsgs);

   // Receive all messages in the MQ by calling mq_receive

   while (nMsgs-- > 0) {
      n = mq_receive(rx, inBuf, sizeof(inBuf), &priority);
      inBuf[n] = '\0';
      if (n < 0) perror("mq_receive");  // Error checking
      printf("Receiver got <%s>, %d bytes at priority %d\n", inBuf, n, priority);
   }

   mq_close(rx);   // Close the MQ
   pthread_exit(NULL);
}

void *Sender(void *arg)  // This thread sends msgs to the MQ
{
   int retcode, i;
   mqd_t tx;
   struct Msg myMsg;
   unsigned int msg_priority[5] = {3, 1, 5, 8, 4};  // Set up message priorities

   sleep(2);

   tx = mq_open(qname, O_WRONLY);  // Open the MQ for writing only (O_WRONLY)

   if (tx < 0) perror("Sender mq_open:");  // Error checking
   // Send 5 messages to the MQ by calling mq_send

   for (i = 1; i <= 5; i++) {
      sprintf(myMsg.str, "Test msg %d", i);  // Set up message to be sent
      myMsg.priority = msg_priority[i-1];    // Set up priority of this msg
      retcode = mq_send(tx, myMsg.str, (size_t)strlen(myMsg.str), myMsg.priority);
      if (retcode < 0) perror("mq_send");  // Error checking
   }

   mq_close(tx);   // Close the MQ
   pthread_exit(NULL);
}

int main(void *arg)
{
   mqd_t trx;
   mode_t mode;
   int oflags;
   struct mq_attr my_attrs;
   pthread_t Sender_thread, Receiver_thread;

   my_attrs.mq_maxmsg = 10;      // Set max no of msg in queue to 10 msgs
   my_attrs.mq_msgsize = 50;     // Set the max msg size to 50 bytes

   oflags = O_CREAT | O_RDWR ;   // Set oflags to create queue (O_CREAT)
                                 //  and for read & write (O_RDWR)

   mode = S_IRUSR | S_IWUSR;     // Set mode to allow other threads
                                 //  (created by same user) to use the queue

   trx = mq_open(qname, oflags, mode, &my_attrs);  // Create the MQ

   if (trx < 0) perror("Main mq_open:");  // Error checking

   // Create a thread to send messages to the MQ

   printf("Creating Sender thread\n");
```

```
    if (pthread_create(&Sender_thread, NULL, &Sender, NULL) != 0) {
        printf("Error creating Sender thread.\n");
        exit(-1);
    }

    // Cretae a thread to receive the messages from the MQ

    printf("Creating Receiver thread\n");
    if (pthread_create(&Receiver_thread, NULL, &Receiver, NULL) != 0) {
        printf("Error creating Receiver thread.\n");
        exit(-1);
    }

    pthread_join(Sender_thread, NULL);
    pthread_join(Receiver_thread, NULL);

    mq_unlink(qname);     // Destroy the MQ
    pthread_exit(NULL);
}
```

The above program has 3 threads: the `main()` thread, the `Sender()` thread and the `Receiver()` thread. The `main()` thread is used to create the message queue, the `Sender()` and `Receiver()` threads. The message queue is created by calling the POSIX function `mq_open()`. Detailed description of this and other related POSIX message queue functions can be found at the ELE709 Blackboard site under Course Documents/OpenGroup Single Unix Specifications.

The `Sender()` thread is used to send 5 messages with different priorities to the message queue. To do this, the queue is first opened as write-only using `mq_open()`. The messages are sent to the queue using `mq_send()`. The queue is then closed after all messages have been sent by calling `mq_close()`. It is important to use `mq_close()` to close a message queue when a thread has finished using it.

The `Receiver()` thread reads the messages from the message queue. To do this, the queue is first opened as read-only using `mq_open()`. The number of messages in the queue is then determined using `mq_getattr()`. The messages in the queue are read by calling `mq_receive()`, and then the queue is closed using `mq_close()`.

Exercise 5.1 A student compiled and ran the program `lab5.c` many times. For most of the time the following expected result was obtained:

```
Creating Sender thread
Creating Receiver thread
Receiver found 5 messages
Receiver got <Test msg 4>, 10 bytes at priority 8
Receiver got <Test msg 3>, 10 bytes at priority 5
Receiver got <Test msg 5>, 10 bytes at priority 4
Receiver got <Test msg 1>, 10 bytes at priority 3
Receiver got <Test msg 2>, 10 bytes at priority 1
```

However, on occasion the following result was obtained:

```
Creating Sender thread
Creating Receiver thread
Receiver found 0 messages
```

Examine the code in `lab5.c` and explain why the expected result is not always produced. Now, without using any semaphores or condition variables, modify the program so that the expected result is always obtained.

Interlocked, One-Way Data Communication

Figure 5.2 shows the “Interlocked, One-Way Data Communication” scheme described in Section 7.7.2 of the reference book.

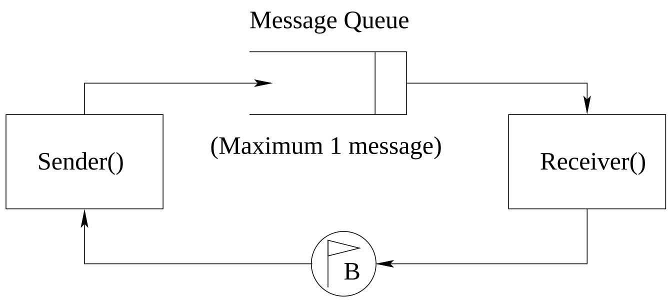

Figure 5.2: Interlocked, One-Way Data Communication

Exercise 5.2 Using the code in lab5.c and Section 7.7.2 of the reference book as a guide, develop a program to demonstrate the “Interlocked, One-Way Data Communication” scheme. Demonstrate your program by using 10 messages of the form “`This is message 1`”, “`This is message 2`”, etc. Set the priority level of Message 1 to 1, Message 2 to 2, etc. The `Sender()` should print out a message before sending it as follow:

```
Sender:  Message sent = This is message 1.  Priority = 1
```

and the `Receiver()` should also print out the received message as follow:

```
Receiver:  Received message = This is message 1.  Priority = 1
```

Interlocked, Two-Way Data Communication

Figure 5.3 shows the "Interlocked, Two-Way Data Communication" scheme described in Section 7.7.3 of the reference book.

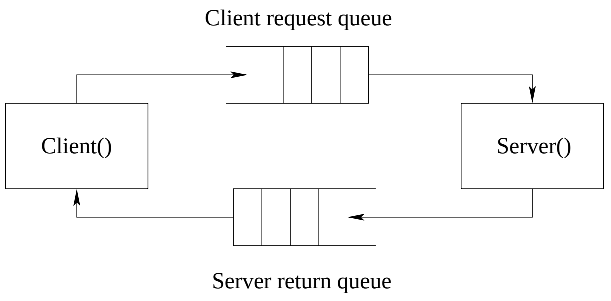

Figure 5.3: Interlocked, Two-Way Data Communication

Exercise 5.3 Using the code in lab5.c and Sections 7.7.3 and 15.3 of the reference book as a guide, develop a program to demonstrate the "Interlocked, Two-Way Data Communication" scheme as follow:

(a) Write a C program client.c which prompts the user for integer inputs n. The integer inputs are then passed on to the "Client request queue".

(b) Write a C program server.c which reads the "Client request queue" and computes the $n-$th prime number (using the integer inputs n received from the "Client request queue"), and returns the results to the "Server return queue". The `compute_prime()` function in Chapter 4 of the book "Advanced Linux Programming" can be modified and used in server.c.

(c) The client.c program should have 2 pthreads, one for prompting the user for the integer inputs and passing the inputs to the "Client request queue", and another to print the results returned by the "Server return queue".

(d) Both the client and server programs should terminate when the user enter the number "0" as the input.

(e) Semaphores or conditional variables are not needed and should not be used.

Sample pseudo code for client.c and server.c are given below. To avoid potential problem, be sure to create the 2 message queues using unique names. Verify the programs by opening 2 terminal windows. Run the server program in one of the windows first, and then run the client program in the other window. Test the programs with the following numbers: 10, 20, 30, 6000, 5000. The correct results should be 29, 71, 113, 59359 and 48611 respectively. (Hints: The C functions `sprintf()` and `atoi()` can be used to convert integers to strings and vice versa.)

Pseudo code for `server.c`:

```
main():
create and open "client request queue"
create and open "server return queue"
while (1) {
   while "client request queue" is not empty {
      read "client request queue" for n
      convert n from string to integer
      if (n == 0) {
         close both queues
         destroy both queues
         pthread_exit
      }
      result = compute_prime(n)
      convert result from integer to string
      send result to "server return queue"
   }
}
```

Pseudo code for `client.c`:

```
main():
   create "client prompt" thread for prompting user’s inputs
   create "client print" thread for printing server’s results
   join both threads
   pthread_exit

client prompt thread():
   open "client request queue"
   stop_printing = 0
   while (1) {
     prompt user for an integer n
     convert n from integer to string
     send n to "client request queue"
     if (n == 0) {
        close "client request queue"
        stop_printing = 1
        pthread_exit
      }
   }

client print thread():
   open "server return queue"
   while (1) {
      while "server return queue" is not empty {
         print result from queue
      }

      if (stop_printing == 1) {
         close "server return queue"
         pthread_exit
      }
   }
```

5.3 What to Submit

A report is not required. Archive the code for Exercises 5.2 and 5.3, and the completed answer sheet for this experiment into a .tar file (use your last name as the name of the tar file) and email it as an attachment to your TA at the end of the laboratory session. For example,

```
tar cvf lastname-lab5.tar ex5.2.c ex5.3.c lab5ans.pdf
```

Appendix A

Answer Sheets

Print out the answer sheets, answer the questions and submit a copy to the TA by the end of the corresponding laboratory sessions.

ELE709 - Real-Time Computer Control Systems
Lab 1 - C Review

Name:

---

1. **Exercise 1.1:**
   The value of `a` is ____________.

2. **Exercise 1.2:** Express your answers in power of 2.

Data Type	Size (bytes)	Minimum Value	Maximum Value
`int`			
`long long int`			
`unsigned int`			
`unsigned long long int`			

3. **Exercise 1.3:**
   The sum of the array is ____________.

ELE709 - Real-Time Computer Control Systems
Lab 2 - Time and Clocks

Name:

---

1. What do you think caused the difference between the requested sleep time and the "actual" sleep time when you run `lab2a.c`?

2. Run the required program for Exercise 2.2 on a workstation in ENG413 and record the execution time per operation for the four basic mathematical operations in the table below:

Execution Time per Operation			
$+$	$-$	$\times$	$\div$

ELE709 - Real-Time Computer Control Systems
Lab 3 - POSIX Threads and Concurrent Programming (Week 1)

Name:

Timing information should be obtained by running the required programs for the Exercises on a workstation in ENG413.

1. **Exercise 3.2**
   (a) Record the required execution time in Table A.1 below.
   (b) Should these execution time be similar to those obtained in Lab 2? Explain.

2. **Exercise 3.3**
   (a) Record the required execution time in Table A.1 below.
   (b) Compare the results obtained for Exercises 3.2 and 3.3. Are the results similar. Explain why.

Table A.1: Execution Time per Iteration

Exercise 3.2					Exercise 3.3				
	$+$	$-$	$\times$	$\div$		$+$	$-$	$\times$	$\div$
1					1				
2					2				
3					3				

ELE709 - Real-Time Computer Control Systems
Lab 3 - POSIX Threads and Concurrent Programming (Week 2)

Name:

Timing information should be obtained by running the required programs for the Exercises on a workstation in ENG413

1. **Exercise 3.4**

   (a) Repeat Exercises 3.2 and 3.3 with the `load` program running concurrently. Record the results in Table A.2 below.

   (b) Are the timing results similar to those obtained when the `load` program *wasn't* running concurrently? Explain why (or why not).

2. **Exercise 3.5** Demonstrate your concurrent matrix multiplication program to the TA.

Table A.2: Execution Time per Iteration (With Load)

Exercise 3.2 (With Load)					Exercise 3.3 (With Load)				
	$+$	$-$	$\times$	$\div$		$+$	$-$	$\times$	$\div$
1					1				
2					2				
3					3				

ELE709 - Real-Time Computer Control Systems
Lab 4 - Resource Sharing and Coordination

Name:

---

1. **Exercise 4.1:**
   (a) Explain why the example program `lab4.c` didn't work correctly?

   (b) A mutex was suggested as a way to make the example program work correctly. What is the purpose for using the mutex in this case? In particular, what is being protected by this mutex?

2. **Exercise 4.2:** Explain the logic of your program for this exercise. In particular, what are the predicates (associated with the condition variable) for Thread A and Thread B?

ELE709 - Real-Time Computer Control Systems
Lab 5 - Task Synchronization and Communication

Name:

1. **Exercise 5.1:**
   (a) Explain why the example program `lab5.c` didn't always produce the correct results?

   (b) How did you modify it so that the correct results are always produced.

2. **Exercise 5.2:** What is the purpose of the binary semaphore in Figure 5.2?