CL2006 - Operating Systems

LAB MANUAL

DEPARTMENT OF ELECTRICAL ENGINEERING,

FAST-NU, LAHORE

Created by: Hamza Yousuf

Date: January, 2019

Last Updated by: Hamza Yousuf

Date: January, 2021

Approved by: Head of Electrical Engineering Department

Date: February, 2022

Table of Contents

List of Equipment

Sr. No.	Description
1	Workstations (PCs)
2	Linux Ubuntu (Software)

EXPERIMENT 1
Introduction to OS and LINUX

OBJECTIVE:

- To get familiarized with the basics of operating systems
- Learn the basic commands used in Linux

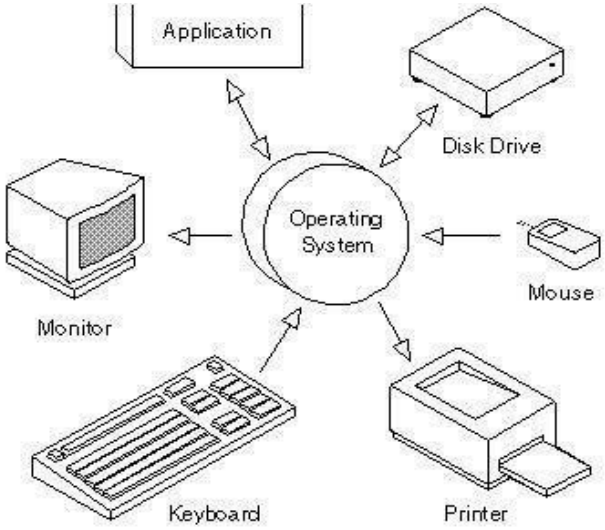
BACKGROUND:

On Computer Startup:

- Power-on self-test (POST) checks for errors
  - CPU
  - Memory
  - Basic input-output systems (BIOS)
- BIOS/firmware
  - Activates the computer's hard disk drives
- Bootstrap loader
  - First piece of the operating system
  - Has a single function to load the operating system into the memory

Operating System:

- What is Operating System?
  Supports computer's basic functions as shown in Figure 1.1
- What tasks an OS Perform?
  - Processor management
  - Memory management
  - Device management
  - Storage management
  - Application interface
  - User interface
- Types
  - Linux
  - Windows 8, Windows 7, Vista, XP
  - Mac



OS Basic Functions
Figure 1.1

What is LINUX?

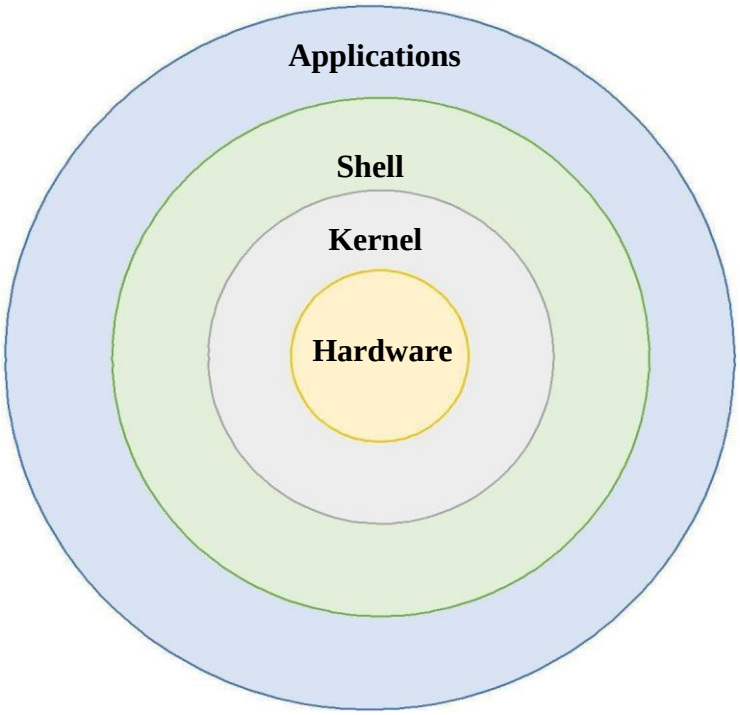
- A fully-networked 32/64-Bit Unix-like Operating System
  - Compilers Like C, C++
- Multi-user, Multitasking
- Coexists with other Operating Systems
- Includes the Source Code
- Open Source

Why is it Significant?

- Growing popularity
- Powerful
  - Runs on multiple hardware platforms
  - Users like its speed and stability
  - No requirement for latest hardware
  - It is free
  - Licensed under GPL (General Public License)

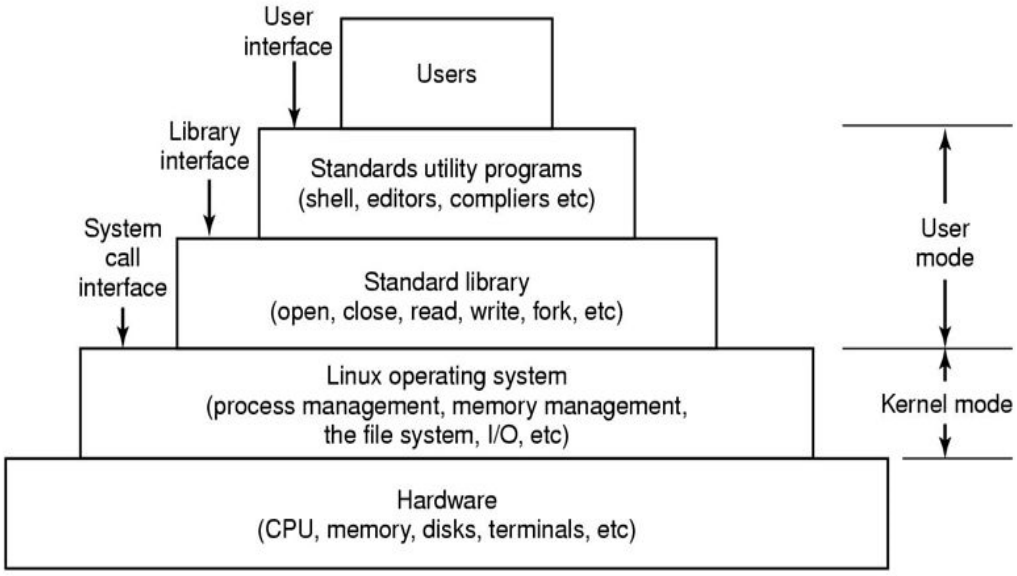
System Structure:

An operating system is a construct that allows the user application programs to interact with the system hardware. This is further divided in two layers i.e Kernel and Shell. Kernel deals with the Hardware and Shell deals with Applications as shown in Figure 1.2.



Structure of an Operating System
Figure 1.2

The Linux System:
Linux is the best-known and most-used open source operating system. As an operating system, Linux is software that sits underneath all of the other software on a computer, receiving requests from those programs and relaying these requests to the computer's hardware. Linux System Structure is shown in Figure 1.3.



Linux System Structure
Figure 1.3

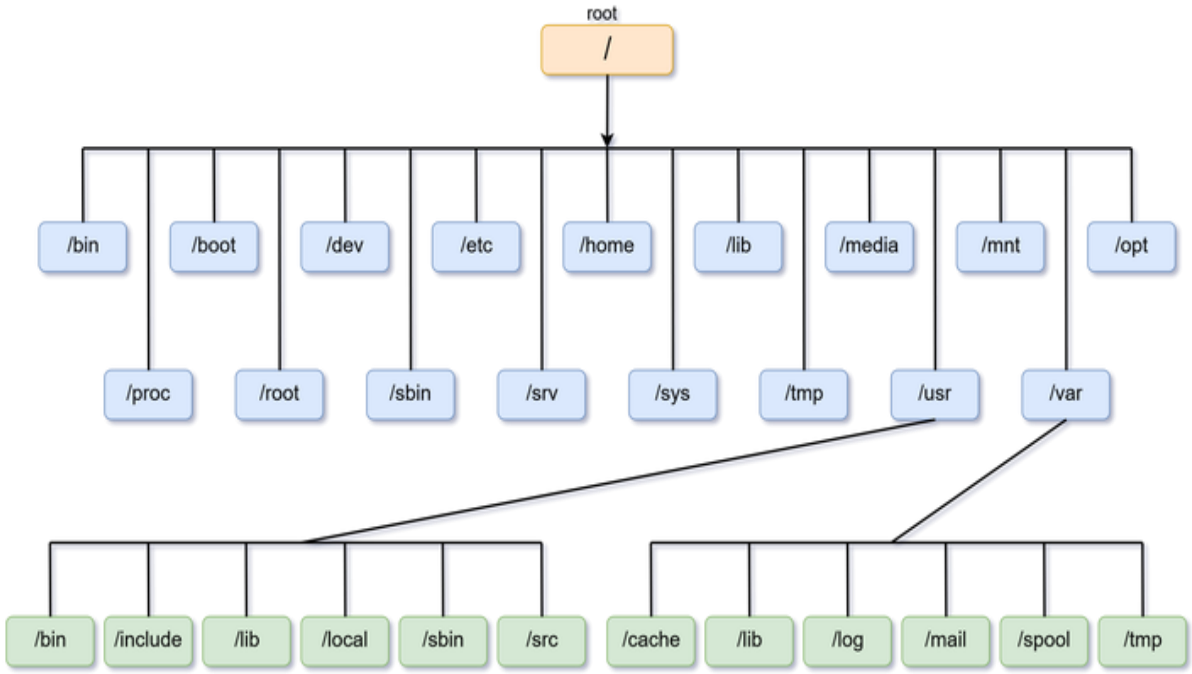
Linux Command Basics:

- To execute a command, type its name and arguments at the command line
- <command_name> <space> <options> <space> <arguments>

Editors:
Several Choices available:

- **`vi`** Standard UNIX editor
- **`the`** XEDIT like editor
- **`xedit`** X windows text editor
- **`emacs`** Extensible, Customizable Self-Documenting Display Editor
- **`pico`** Simple display-oriented text editor
- **`nedit`** X windows Motif text editor

The File system:
Unix uses a hierarchical file system structure, much like an upside-down tree, with root (/) at the base of the file system and all other directories spreading from there as can be seen from Figure 1.4. Linux differs from Windows in many ways. The comparison has been given in Table 1.5.



Linux File System
Figure 1.4

Windows	LINUX
• The directories in MS-DOS path are separated by '\' • File names are case insensitive. • Where DOS/Windows had various partitions and then directories under those partitions. • An executable is one with an extension of .exe, .com or .bat. • You can set attributes to make file read only, hidden	• Paths are separated by '/'. • File names are case sensitive. • There is only a single hierarchal directory structure (resembles a tree). Everything starts from the root directory, represented by '/', and then • expands into sub-directories. • Any file whose execute permission is turned on is executable • You can set permissions on a file

Table 1.5 (Comparison between Windows OS and Linux OS)

Special Files:

- **/home -** all users' home directories are stored here
- **/bin, /usr/bin -** system commands
- **/etc -** all sorts of configuration files
- **/var -** logs, spool directories etc
- **/dev -** device files
- **/proc -** special system files

Virtual Machine:

- What is virtual Machine?
  - VirtualBox and VMWare
- ISO files – Ubunto ISO file
- Ubunto installation on VirtualBox or VMware

Installation of Linux in Virtual Machine:

a. Install VMware on your Machines.
b. Get Latest ISO file of Ubuntu distribution according to your system architecture (32bit or 64bit) from following link http://www.ubuntu.com/download/desktop.
c. Install Ubuntu from this ISO image file as guest Operating system in VMware.

Some Commands for Beginners:

- Clear the console
  - **clear**
- Changing working Directory
  - **cd Desktop**
  - **cd Home**
- List all files in directory
  - **ls**
- Copy all files of a directory within the current work directory
  - **cp dir/***
- Copy a directory within the current work directory
  - **cp -a tmp/dir1**
- Look what these commands do
  - **cp -a dir1 dir2**
  - **cp filename1 filename2**
- To make archive of existing folder or files
  - **tar cvf archive_name.tar dirname/**
  - **tar cvf alldocs.tar *.txt**
- Extract from an existing tar archive
  - **tar xvf archive_name.tar**
- View an existing tar archive
  - **tar tvf archive_name.tar**

Some more Commands:

- **ls** show files in current position
- **cd** change directory
- **cp** copy file or directory
- **mv** move file or directory
- **rm** remove file or directory
- **pwd** show current position
- **mkdir** create directory
- **rmdir** remove directory
- **less, more, cat** display file contents
- **man** read the online manual page for a command
- **whatis** give brief description of a command
- **su** switch user
- **passwd** change password
- **useradd** create new user account
- **userdel** delete user account
- **mount** mount file system
- **umount** unmount file system
- **df** show disk space usage
- **shutdown** reboot or turn off machine

Post Lab Questions:

1. Provide details about the following commands?

- apt-get
- yum
- wget
- gzip tar
- rar

2. Find and Execute following commands in Linux Shell?

- show architecture of machine
- show CPU info
- show version of the kernel
- show system date
- set date and time
- show details of files and directory
- show hidden files
- show files and directory containing numbers
- create a directory called 'dir1'
- create two directories simultaneously
- show the path of work directory
- delete file called 'file1'
- remove a directory called 'dir1' and contents recursively
- delete directory called 'dir1'
- modify timestamp of a file or directory

EXPERIMENT 2

Creating, Compiling and Executing C/C++ programs using gcc/g++ Compilers and Make File

OBJECTIVE:

- Learn the use of g++ and gcc compilers to compile and execute C++ and C programs
- To get familiarized with the working of Make File for C/C++ programs

BACKGROUND:

Compiling C/C++ program using g++ and gcc:

For C++:

Command: g++ *source_files…* -o *output_file*

For C:

Command: gcc *source_files… -o output_file*

Source files need not be cpp or c files. They can be preprocessed files, assembly files, or object files.

The whole compilation file works in the following way:

Cpp/C file(s) → Preprocessed file(s) → Assembly File(s) Generation → Object file(s) Generation → Final Executable

Every c/cpp file has its own preprocessed file, assembly file, and object file.

1. For running only the preprocessor, we use -E option.

2. For running the compilation process till assembly file generation, we use –S option.

3. For running the compilation process till object file creation, we use –c option.

4. If no option is specified, the whole compilation process till the generation of executable will run.

A file generated using any option can be used to create the final executable. For example, let's suppose that we have two source files: math.cpp and main.cpp, and we create object files:

g++ main.cpp –c –o main.o

g++ math.cpp –c –o math.o

The object files created using above two commands can be used to generate the final executable.

g++ main.o math.o –o my_executable

The file named "my_executable" is the final exe file. There is specific extension for executable files in Linux.

Command Line Arguments:

Command line arguments are a way to pass data to the program. Command line arguments are passed to the main function. Suppose we want to pass two integer numbers to main function of an executable program called a.out. On the terminal write the following line:

./a.out 1 22

./a.out is the usual method of running an executable via the terminal. Here 1 and 22 are the numbers that we have passed as command line argument to the program. These arguments are passed to the main function. In order for the main function to be able to accept the arguments, we have to change the signature of main function as follows:

int main(int argc, char *arg[]);

- ➔ argc is the counter. It tells how many arguments have been passed.
- ➔ arg is the character pointer to our arguments.

argc in this case will not be equal to 2, but it will be equal to 3. This is because the name ./a.out is also passed as command line argument. At index 0 of arg, we have ./a.out; at index 1, we have 1; and at index 2, we have 22. Here 1 and 22 are in the form of character string, we have to convert them to integers by using a function atoi. Suppose we want to add the passed numbers and print the sum on the screen:

cout<< atoi(arg[1]) + atoi(arg[2]);

Compiler Process:

- Compiler Stage: All C++ language code in the .cpp file is converted into a lower-level language called Assembly language; making .s files.
- Assembler Stage: The assembly language code made by the previous stage is then converted into object code which are fragments of code which the computer understands directly. An object code file ends with .o.

- Linker Stage: The final stage in compiling a program involves linking the object code to code libraries which contain certain "built-in" functions, such as cout. This stage produces an executable program, which is named a.out by default.

Makefiles:

- Provide a way for separate compilation.
- Describe the dependencies among the project files.
- The *make* utility.

Using makefiles:

Naming:

- *makefile* or *Makefile* are standard
- other name can be also used

Running make:

make

make –f filename – if the name of your file is not “makefile” or “Makefile”

Sample makefile:

Makefiles main element is called a *rule*:

```
target : dependencies
TAB  commands     #shell commands
```

Example:

```
my_prog : eval.o main.o
g++ -o my_prog eval.o main.o        (-o to specify executable file name)
eval.o : eval.c eval.h
g++ -c eval.c                       (-c to compile only (no linking))
main.o : main.c eval.h
g++ -c main.c
```

Variables:

Figure 3.1 shows the use of variables while generating makefile.

The old way (no variables)	A new way (using variables)

	`C = g++`
	`OBJS = eval.o main.o`
	`HDRS = eval.h`

`my_prog : eval.o main.o`
`g++ -o my_prog eval.o main.o`
`eval.o : eval.c eval.h`
`g++ -c -g eval.c`
`main.o : main.c eval.h`
`g++ -c -g main.c`
	`my_prog : eval.o main.o`
	`$(C) -o my_prog $(OBJS)`
	`eval.o : eval.c`
	`$(C) -c -g eval.c`
	`main.o : main.c`
	`$(C) -c -g main.c`
	`$(OBJS) : $(HDRS)`

Use of Variables for generating makefile

Figure 3.1

Automatic variables:

Automatic variables are used to refer to specific part of rule components.

eval.o : eval.c eval.h

g++ -c eval.c

$@ - The name of the target of the rule (eval.o).

$< - The name of the first dependency (eval.c).

$^ - The names of all the dependencies (eval.c eval.h).

$? - The names of all dependencies that are newer than the target

make **options:**

-f filename - when the makefile name is not standard

-t - (touch) mark the targets as up to date

-q - (question) are the targets up to date, exits with 0 if true

-n - print the commands to execute but do not execute them

/ -t, -q, and -n, cannot be used together /

-s - silent mode

-k - keep going – compile all the prerequisites even if not able to link them !!

Conditionals (directives):

Possible conditionals are:

if ifeq ifneq ifdef ifndef

All of them should be closed with *endif*.

Complex conditionals may use *elif* and *else*.

Example:

```
libs_for_gcc = -lgnu

normal_libs =

ifeq ($(CC),gcc)

libs=$(libs_for_gcc)          #no tabs at the beginning

else

libs=$(normal_libs)           #no tabs at the beginning

endif
```

In-Lab Questions:

Question 1: Write a C or C++ program that accepts a file name as command line argument and prints the file's contents on console. If the file does not exist, print some error on the screen.

Question 2: Write a C or C++ program that accepts a list of integers as command line arguments sorts the integers and print the sorted integers on the screen.

Question 3: Create the following classes in separate files (using .h and .cpp files)

Student, Teacher, Course.

A student has a list of courses that he is enrolled in.

A teacher has a list of courses that he is teaching.

A course has a list of students that are studying it, and a list of teachers that are teaching the course. Create some objects of all classes in main function and populate them with data.

Now compile all classes using makefile

Post-Lab Questions:

Problem 1: Write a C/C++ program that takes some integers as command line parameters, store them in an array and prints the sum and average of that array. Also note that you have to run the program for all possible error checks.

Problem 2: Write a C/C++ program that takes some integers in the form of series as command line parameters; store them in array than compute the missing element from that series and output that missing element to file.

Problem 3: Write a C/C++ program that reads file in which there are integers related to series and store them in array than compute the missing element from that series and output that missing element to file.

Problem 4: Create the following classes in separate files (using .h and .cpp files)

LetterCount, WordCount, LineCount.

LetterCount counts number of letters in a text file.

WordCount counts number of words in a text file.

LineCount counts number of lines in a text file.

Create some objects of all classes in main function and populate them with data.

Now compile all classes using makefile

EXPERIMENT 3
System Calls

OBJECTIVE:

- To understand about Linux system calls and their use in processes

BACKGROUND:

What are system calls?

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms, it is the method used by a process to request action by the operating system. Figure 4.1 shows some of the primarily used system calls.

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Some useful System Calls
Figure 4.1

Process Creation:

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination

I. fork()

- Has a return value
- Parent process => invokes fork() system call

- Continue execution from the next line after fork()
- Has its own copy of any data
- Return value is > 0 //it's the process id of the child process. This value is different from the Parents own process id.
- Child process => process created by fork() system call
- Duplicate/Copy of the parent process //LINUX
- Separate address space
- Same code segments as parent process
- Execute independently of parent process
- Continue execution from the next line right after fork()
- Has its own copy of any data
- Return value is 0

II. wait ()

- Used by the parent process
- Parent's execution is suspended
- Child remains its execution
- On termination of child, returns an exit status to the OS
- Exit status is then returned to the waiting parent process //retrieved by wait ()
- Parent process resumes execution
- #include <sys/wait.h>
- #include <sys/types.h>

III. exit()

- Process terminates its execution by calling the exit() system call
- It returns exit status, which is retrieved by the parent process using wait() command
- EXIT_SUCCESS // integer value = 0
- EXIT_FAILURE // integer value = 1
- OS reclaims resources allocated by the terminated process (dead process) Typically performs clean-up operations within the process space before returning control back to the OS
- Terminates the current process without any extra program clean-up
- Usually used by the child process to prevent from erroneously release of resources belonging to the parent process

IV. execlp() is a version of exec()

- **exec()**
- Runs an executable file
- Called by an already existing process //child process
- Replaces the previous executable //overlay
- Has an exist status but cannot return anything (if exec() is successful) to the program that made the call //parent process

- Return value is -1 if not successful
- Overlay => replacement of a block of stored instructions or data with another int execlp(char const *file_path, char const *arg0,);
- Arguments beginning at arg0 are pointers to arguments to be passed to the new process.
- The first argument arg0 should be the name of the executable file.
- Example
- execlp(/bin/ls , ls ,NULL) //lists contents of the directory

Header file used -> unistd.h

Information Maintenance:

i. sleep()

- Process goes into an inactive state for a time period
- Resume execution if
- Time interval has expired
- Signal/Interrupt is received
- Takes a time value as parameter (in seconds on Unix-like OS and in milliseconds on Windows OS)
- sleep(2) // sleep for 2 seconds in Unix
- Sleep(2*1000) // sleep for 2 seconds in Windows

ii. getpid() // returns the PID of the current process

- getppid() // returns the PID of the parent of the current process
- Header files to use
- #include <sys/types.h>
- #include <unistd.h>
- getppid() returns 0 if the current process has no parent

Example:

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
pid_t pid;
pid=fork();
if(pid==0){
printf("I'm in a child process \n\n");
}
else if (pid>0){
printf("I'm in the parent process \n\n");
}
else
{
printf("Error \n\n");
}
return 0;
}
```

In-lab problems:

1. Create a program that creates a child process. The child process prints "I am a child process" 100 times in a loop. Whereas the parent process prints "I am a parent process" 100 times in a loop.
2. Create a program named stat that takes an integer array as command line argument (deliminated by some character such as $). The program then creates 3 child processes each of which does exactly one task from the following.

    a) Adds them and print the result on the screen. (done by child 1)
    b) Shows the average on the screen. (done by child 2)
    c) Prints the maximum number on the screen. (done by child 3)

3. Invoke at least 4 commands from your programs, such as
   Cp, mkdir, rmdir, etc (The calling program must not be destroyed)

Post-Lab Questions:

Q1. Write a program which uses fork () system-call to create a child process. The child process prints the contents of the current directory and the parent process waits for the child process to terminate.

Q2. Write a program which prints its PID and uses fork () system call to create a child process. After fork () system call, both parent and child processes print what kind of process they are and their PID. Also the parent process prints its child's PID and the child process prints its parent's PID.

EXPERIMENT 4
Threads

OBJECTIVE:

- To understand and learn about threads and their implementation in programs

BACKGROUND:

Threads:

pthread_create(pthread_t* , NULL, void*, void*)

- First Parameter is pointer of thread ID it should be different for all threads.
- Second Parameter is used to change stack size of thread. Null means use default size.
- Third perimeter is address of function which we are going to use as thread.
- Forth parameter is argument to function.

pthread_join(i pthread_t , void**)

Pthread join is used in main program to wait for the end of a particular thread.

- First parameter is Thread ID of particular thread.
- Second Parameter is used to catch return value from thread.

Thread Library:

- POSIX Pthreads
- Two general stratergies for creating multiple threads.
  a) Asynchronous threading:
     - Parent and child threads run independently of each other
     - Typically little data sharing between threads
  b) Synchronous threading:
     - Parent thread waits for all of its children to terminate
     - Children threads run concurrently
     - Significant data sharing

Pthreads:

- **pthread.h**
- Each thread has a set of attributes, including stack size and scheduling information
- In a Pthreads program, separate threads begin execution in a specified function //runner()
- When a program begins
  - A single thread of control begins in main()
  - main() creates a second thread that begins control in the runner() function
  - Both threads share the global data

Example:

Design a multi-threaded program that performs the summation of a non-negative integer in a separate thread using the summation function:

$$sum = \sum_{i=0}^{N} \mathrm{i}$$

For example, if N were 5, this function would represent the summation of integers from 0 to 5, which is 15.

```
#include <pthread.h>
#include <stdio.h>
int sum;  //this data is shared by the thread(s)
//The thread will begin control in this funtion
void* runner(void *parameters)
{
int i, upper=atoi(parameters);
if(upper>0)
{
for(i=1;i<=upper;i++)
sum=sum+i;
}
pthread_exit(0);
}//End runner
int main(int argc, char*argv[])
{
//thread identifier
pthread_t threadID;
//set attributes for the thread
pthread_attr_t attributes;
//get the default attributes
  pthread_attr_init(&attributes);
//create the thread
pthread_create(&threadID, &attributes, runner,
argv[1]);
//now wait for the thread to exit
pthread_join(threadUD, NULL);
printf("sum=%d\n",sum);
}
```

Question 1:

Write a program which takes some positive integers (let's say **N** number of positive integers) as command line parameters, creates **N** synchronous threads, and send s the corresponding integer as parameter to the thread function fibonacciGenerator. The function returns the generated series to the main thread. The main thread will then print the thread number and the series generated by that thread. The output will be like:

Thread 1: 0 1 1 2 3 5 8 13

Example:

If you pass as command line argument the following numbers: 3 13 34 89

Then the program will create 4 threads. The first thread will find Fibonacci terms until 3 is generated, the second Fibonacci term will find Fibonacci terms until the term generated is 13 so on and so forth. All generated terms will be output on the screen by the main thread as follows:

Thread 0: 0, 1, 1, 2, 3

Thread 1: 0, 1, 1, 2, 3, 5, 8, 13

Thread 2: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Thread 3: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89

It is possible that the number passed to the thread is not a Fibonacci number. In this case the thread will generate numbers until the term generated is greater than the passed number. For example, if 7 is passed as parameter to a thread, then the thread will return the following series:

0, 1, 1, 2, 3, 5, 8

Question 2:

Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers. (The array of numbers must be passed as parameter to threads, and the thread must return the calculated value to main thread).

90 81 78 95 79 72 85

The main thread will print:

The average value is 82

The minimum value is 72

The maximum value is 95

EXPERIMENT 5
InterProcess Communication using Pipes

OBJECTIVE:

- Learn and Understand InterProcess Communication using implementation of Pipes

BACKGROUND:

Pipes:

Ordinary pipes allow two processes to communicate in standard producer consumer fashion: the producer writes to one end of the pipe (the **write-end)** and the consumer reads from the other end (the **read-end). As a result, ordinary** pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction.

- A pipe has a read end and a write end.
- Data written to the write end of a pipe can be read from the read end of the pipe.

Creating an Ordinary Pipe:

On UNIX and LINUX systems, ordinary pipes are constructed using the function

- int pipe(int fd[2]) – creates a pipe
- returns two file descriptors, fd[0], fd[1].
- Fd[0] is the read-end of the pipe.
- Fd[1] is the write-end of the pipe.
- Fd[0] is opened for reading
- Fd[1] is opened for writing. Pipe() sets 0 on success, -1 on failure and sets errno accordingly.
- The standard programming model is that after the pipe has been set up, two (or more) cooperative processes will be created using read() and write().
- Pipes opened with pipe() should be closed with close(int fd).

Example 1:

```
int pdes[2];
pipe(pdes);
if(fork()==0)
{//child
close(pdes[1]);
read(pdes[0]); //read from parent
}
else
{
close(pdes[0]);
write(pdes[1]); //write to child
}
```

Example 2:

```
#include <sys/wait.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
int main(int argc, char* argv[])
{
int pfd[2];
pid_t cpid;
char buf;
if (pipe(pfd)==-1)
{perror("pipe");
exit(EXIT_FAILURE);}
cpid=fork();
if (cpid==-1)
{perror("fork");
exit(EXIT_FAILURE);}
if (cpid==0)
{
close(pfd[1]);
while(read(pfd[0],&buf,1)>0)
{printf(buf);}
close(pfd[0]);
exit(EXIT_SUCCESS);
}
else
{close(pfd[0]);
write(pfd[1],argv[1],strlen(argv[1]));
close(pfd[1]);
wait(NULL);
exit(EXIT_SUCCESS);
}
}
```

When pipe() System Call Fails:

The pipe() system call fails for many reasons, including the following:

- At least two slots are not empty in the FDT – too many files or pipes are open in the processes.
- Buffer space not available in the kernel.

In-lab Questions:

Q1. Design a program using ordinary pipes in which one process sends a string message to a second process, and the second process reverses the case of each character in the message and sends it back to the first process. For example, if the first process sends the message Hi There, the second process will return hI tHERE. This will require using two pipes, one for sending the original message from the first to the second process, and the other for sending the modified message from the second back to the first process.

Q2. Design a file-copying program named FileCopy using ordinary pipes. This program will be passed two parameters: the first is the name of the file to be copied, and the second is the name of the copied file. The program will then create an ordinary pipe and write the contents of the file to be copied to the pipe. The child process will read this file from the pipe and write it to the destination file. For example, if we invoke the program as follows: FileCopy input.txt copy.txt the file input. txt will be written to the pipe. The child process will read the contents of this file and write it to the destination file copy.txt.

EXPERIMENT 6

InterProcess Communication using Shared Memory

OBJECTIVE:

- Learn and Understand InterProcess Communication using implementation of Shared Memory

BACKGROUND:

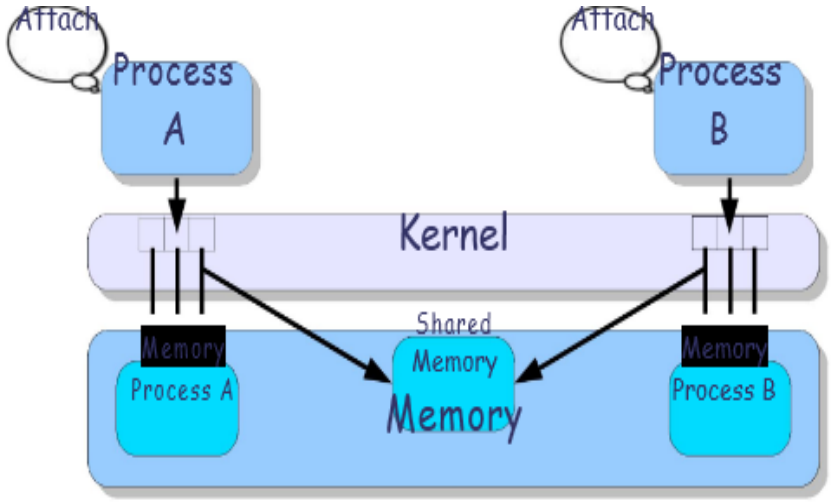
Shared Memory:

InterProcess Communication through shared memory is a concept where two or more process can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by anther process.

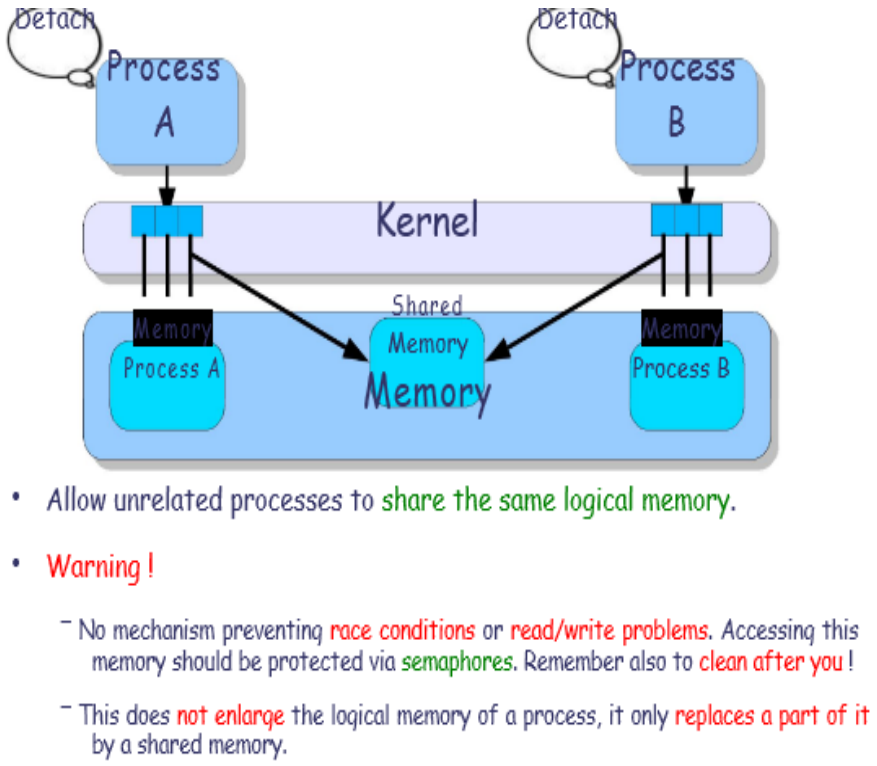
The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.

- Server reads from the input file.
- The server writes this data in a message using either a pipe, fifo or message queue.
- The client reads the data from the IPC channel, again requiring the data to be copied from kernel's IPC buffer to the client's buffer.
- Finally, the data is copied from the client's buffer.

A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file. Figure 7.1 and 7.2 are a pictorial representation of above concept.



Shared Memory Attachment
Figure 7.1



Shared Memory Detachment
Figure 7.2

int shmget(key_t key, int size, int flags);
Allocates a shared memory segment.

- key is the key associated with the shared memory segment you want.
- size is the size in bytes of the shared memory segment you want allocated. Memory gets allocated in pages, so chances are you'll probably get a little more memory than you wanted.
- flags indicate how you want the segment created and its access permissions. The general rule is just to use 0666 | IPC_CREAT | IPC_EXCL if the caller is making a new segment. If the caller wants to use an existing share region, simply pass 0 in the flag.

shmget() will fail if:
1. Size specified is greater than the size of the previously existing segment. Size specified is less than the system imposed minimum, or greater than the system imposed maximum.
2. No shared memory segment was found matching *key*, and IPC_CREAT was not specified.
3. The kernel was unable to allocate enough memory to satisfy the request.
4. IPC_CREAT and IPC_EXCL were specified, and a shared memory segment corresponding to *key* already exists.

Return Values:
Upon successful completion, **shmget**() returns the positive integer identifier of a shared memory segment. Otherwise, -1 is returned

void *shmat(int shmid, const void *shmaddr, int shmflg);
Maps a shared memory segment onto your process's address space.

- shmid is the id as returned by shmget() of the shared memory segment you wish to attach.
- Addr is the address where you want to attach the shared memory. For simplicity we will pass NULL. NULL means that kernel itself will decide where to attach it to address space of the process.

shmat() will fail if:
1. No shared memory segment was found corresponding to the given id.

Return Values:
Upon success, **shmat**() returns the address where the segment is attached; otherwise, -1 is returned and *errno* is set to indicate the error.

int shmdt(void *addr);

This system call is used to detach a shared memory region from the process's address space.

- Addr is the address of the shared memory

shmdt() will fail if:

1. The address passed to it does not correspond to a shared region.

Return Values:

Upon success, **shmdt**() returns 0; otherwise, -1 is returned and *errno* is set to indicate the error.
How to Delete Shared Memory Region:

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
shmctl() performs the control operation specified by *cmd* on the System V shared memory segment whose identifier is given in *shmid*.

For Deletion we will use **IPC_RMID** flag.

IPC_RMID marks the segment to be destroyed. The segment will actually be destroyed only after the last process detaches it (The caller must be the owner or creator of the segment, or be privileged). The *buf* argument is ignored.

Return Values:
For IPC_RMID operation, 0 is returned on success; else -1 is returned.
Shmctl(shmid, IPC_RMID, NULL);

In-Lab Questions:

1. Create a private shared memory in C/C++. The process then creates a child and waits for the child to write the file's contents to shared memory. The parent then reads the shared memory and changes the case of each character and removes all integers from the data. The child reads it back and writes the changed data back to the same file. (The file name is passed as command line argument).
2. Create a C++/C program that creates a shared memory and waits for the other process to write n data of n number of Students. The process that created the shared memory then writes the data of students to the file.

EXPERIMENT 7
InterProcess Communication using Sockets

OBJECTIVE:

- Learn and Understand InterProcess Communication using socket programming

BACKGROUND:

Sockets:

A socket is defined as an endpoint for communication. A pair of processes communicating over a network employs a pair of sockets—one for each process. A socket is identified by an IP address concatenated with a port number. In general, sockets use a client–server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection. Servers implementing specific services (such as telnet, FTP, and HTTP) listen to well-known ports (a telnet server listens to port 23; an FTP server listens to port 21; and a web, or HTTP, server listens to port 80). All ports below 1024 are considered well known; we can use them to implement standard services.

- Communication points on the same or different computers to exchange data
- Allows communication between two different processes on the same of different machines
- Always has an address (IP and Port)
- A UNIX socket is used in a client-server application framework
- Server is a process that performs some functions on request from a client
- Just like a file (open, close, read, write)

IP address (Internet Prortocol) & Socket:

- IP address
  - Identify hosts connected to the internet
  - Written in a dotted-decimal notation of the form N1.N2.N3.N4 where each Ni is a decimal number between 0 and 255
- Socket
  - To identify a particular process running on a host
  - An integer number
  - Port numbers smaller than 1024 i.e. 0-1023 are well-known ports // port 80 for http (standard service)
  - We can use port numbers from 1024 to 65535
- Works like telephone extension
  - Main phone number computer IP address
  - Extension numbers set of port numbers

Loopback IP:

- The IP address 127.0.0.1 is a special IP address known as the loopback. When a computer refers to IP address 127.0.0.1, it is referring to itself.
- This mechanism allows a client and server on the same host to communicate using the TCP/IP protocol.

Types of Socket:

- Stream sockets (SOCK_STREAM)
  - TCP (Transmission Control Protocol)
  - Message delivery is guaranteed
  - Message order retains
  - Sender receives error message on failure
- Datagram sockets (SOCK_DGRAM)
  - UDP (User Datagram Protocol)
  - Delivery not guaranteed
  - Connection less (build message with destination information and sent it out)

Functions used in Socket Programming:

socket()	Endpoint for communication
bind()	Assign a unique telephone number
listen()	Wait for a caller
connect()	Dial a number
accept()	Receive a call
send(), recv()	Talk
close()	Hang up

- **socket( ) … get the file descriptor**
  - int sd=socket(int domain, int type, int protocol);
    - domain AF_INET, PF_INET
    - type SOCK_STREAM, SOCK_DGRAM
    - protocol set to 0 for appropriate protocol selection, IPPROTO_TCP, IPPROTO_UDP
    - return socket descriptor on success and -1 on error
  - Example
    - int T_s=socket(AF_INET, SOCK_STREAM, 0);

- **bind( ) … what port am I on?**
  - Associate a socket id with an address to which other process can connect
  - int status=bind(int sd, struct sockaddr* addrptr, int size);
    - status 0 on success and -1 on error
    - sd socket file descriptor created and return by socket()

- addrptr pointer to struct sockaddr type parameter, contains current socket IP and port
- size size of addrptr

- **connect( ) … request for connection**
  - int status=connect(int sd, struct sockaddr *serv_addr, int addrlen)
    - status error -1
    - sd socket file descriptor
    - serv_addr is a pointer to struct sockaddr that contains destination IP address and port
    - addrlen size of serv_addr

- **listen( )**
  - Waits for incoming connections
  - int status=listen(int sd, int backlog);
    - sd socket on which the server is listening
    - backlog maximum number of connections pending in a queue
    - status return -1 on error

- **accept( )**
  - Blocking system call
  - Waits for an incoming request and when received, creates a socket for it
  - int sid=accept(int sd, struct sockaddr *cli_addr, int *addrlen)
    - sid socket file descriptor for communication
    - sd socket file descriptor used for listening
    - addr poitner to struct sockaddr containing client address IP and Port
    - addrlen size of struct sockaddr

- **send( )**
  - int sb=send(int sd, const char *msg, int len, int flags);
    - sb return number of bytes send of -1 for error
    - sd socket file descriptor
    - msg is a pointer to data buffer
    - len number of bytes we want to send
    - flags set it to 0 for default

- **recv( )**
  - int rb=recv(inst sd, char *buf, int len, int flags);
    - rb number of bytes received or -1 on error. 0 if connection is closed at other side
    - sd socket file descriptor
    - buf is a pointer to data buffer
    - len receive up to len bytes in buffer pointer
    - flag set it to 0 for default

- **close( )**
  - Close connection on given socket and frees the socket descriptor
  - int close(int sd);

struct sockaddr:

- Generic
  - Holds socket address information for many types of sockets

```
struct_sockaddr{
        unsigned short sa_family; //address family AF_xxx
        unsigned short sa_data[14]; //14 bytes of protocol addr
}
```

struct sockaddr_in:

- IPV4 specific

```
struct_sockaddr_in{
        short int sin_family; //set to AF_INET
        unsigned short int sin_port; //port number
        struct in_addr sin_addr; //internet address
        unsigned char sin_zero[8]; //set to all zeros
}
```

Client-Server Model:

Client-Server mechanism has been shown in a generic model in Figure 8.1 and Figure 8.2 shows the model for the implementation here in specific regard to IPC.

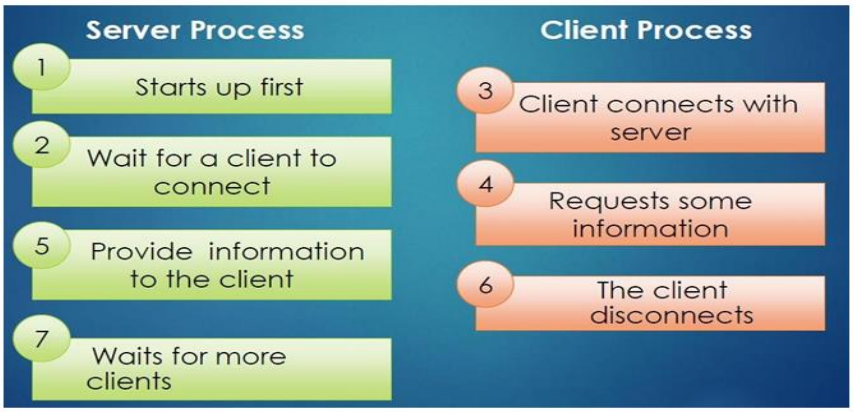


Figure 8.1 (General client-server model)

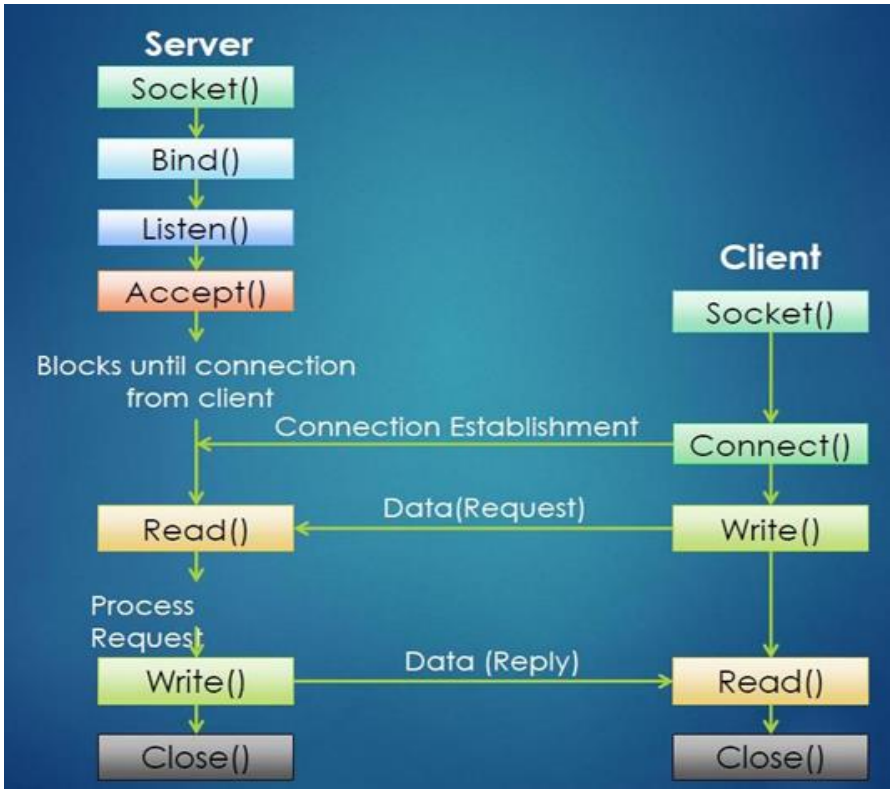


Figure 8.2 (TCP client-server model)

Commands for IP inquiry:

- ipconfig // windows
- ifconfig // linux
- hostname –I //linux
- ip addr show //linux
- Network tools

Header Files:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
```

- Server runs first

Server:

```
int main(){
int sd;
char* msg="connected";
struct sockaddr_in my_addr, client_addr;
my_addr.sin_family=AF_INET;
my_addr.sin_port=htons(4999);
my_addr.sin_addr.s_addr=inet_addr("127.0.0.1");
sd=socket(AF_INET,SOCK_STREAM,0);
bind(sd,(struct sockaddr *) &my_addr,sizeof(struct
sockaddr_in));
listen(sd,5);
int i=0;
while(i<1)
{
int size=sizeof(struct sockaddr);
int new_sd=accept(sd,(struct sockaddr*)&client_addr,
&size);
send(new_sd, msg,100,0);
++i;
}
return 0;
}
```

Client:

```
int main()
{
int sd;
char msg[100];
struct sockaddr_in server_addr;
server_addr.sin_family=AF_INET;
server_addr.sin_port=htons(4999);
server_addr.sin_addr.s_addr=inet_addr("127.0.0.1");
sd=socket(AF_INET,SOCK_STREAM,0);
connect(sd,(struct sockaddr*)
&server_addr,sizeof(struct sockaddr));
recv(sd,msg,100,0);
printf("%s\n",msg);
return 0;
}
```

EXPERIMENT 8
Named Pipes

OBJECTIVE:

- Learn and execute InterProcess Communication using implementation of named pipes

BACKGROUND:

Named Pipes:

- It is an extension to the traditional pipe concept on Unix. A traditional pipe is "unnamed" and lasts only as long as the process.
- A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.
- Usually a named pipe appears as a file, and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.
- A FIFO special file is entered into the filesystem by calling mkfifo() in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.
- Reading from or writing to a named pipe occurs just like traditional file reading and writing; except that the data for named pipe is never written to or read from a file in hard disk but memory.

Input/output Redirection:

To understand input/output redirection, we need to understand named pipes and dup2 system call first.

Dup2:

Dup2 system call is used to make alias of a file descriptor, for example:

Int fd=open("file.txt", O_WRONLY);

dup2(fd, 1);

cout << "This data will not be printed to the screen, but to the file", 50);

The above system call will first close file descriptor 1 using close system call. Then, file descriptor fd's data will be copied to file descriptor 1. So, every write that is made using fd=1 will then be written to file.txt

Similarly,

```
Int fd=open("file.txt", O_RDONLY);

dup2(fd, 0);

char buffer[10];

cin>>buffer;
```

Cin will not get input from keyboard, but from file.txt in above piece of code**. (Sample code is given)**

On shell, we make the input of one command as the output of another command by using | symbol.

ls | sort

ls displays directory contents and sort simply sorts the input data. In above command, we are passing the output of ls command as input to sort command. The output shown on the screen will be generated by sort command. Behind the scenes, it is done by named pipes and dup2.

To redirect the output of a command to a file, we can use > symbol, such as

Ls > 1.txt

The above command will redirect the output generated by ls command to file 1.txt

In-lab Questions

Question 1:

Create 2 independent programs that perform communication using named pipes. One program will be the server program that will wait for client to send some data via a named pipe. The data sent by the client is as follows:

Operator operand1 operand2

The operands can be +, -, *, /. The server will then apply the operator on the operands and return the result to client via named pipe. The client will then print the result on the screen

For example, if the client passed the following to the server: + 4 10, then the server will calculate 4+10 and return 14 to client via the pipe. The client will then print it.

Question 2:

Implement a program that executes the command:

ls | sort

This will help: execlp("sort","sort",NULL);

EXPERIMENT 9
Semaphores using Shared Memory

OBJECTIVE:

- Learn the concept of semaphores
- Understand the use of Shared Memory
- Learn to use semaphores for synchronization in InterProcess Communication

BACKGROUND:

Race Condition:

A situation that several tasks access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access take place.

- Example:
  - Suppose that the value of the variable counter = 5.
  - Process 1 and process 2 execute the statements “counter++” and “counter--” concurrently.
  - Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6! Table 10.1 shows what happens in this case …

"counter++" is implemented as
register1=counter
register1=register1 + 1
counter=register1

"counter--" is implemented as
register2=counter
register2=register2 - 1
counter=register2

Process1	register1	**Process2**	register2	**counter**
register1 = counter	5	…	…	5
register1 = register1 + 1	6	…	…	5
…	…	register2 = counter	5	5
…	…	register2 = register2 - 1	4	5
counter = register1	6	…	…	6
…	…	counter = register2	4	4

Table 10.1 (Race Condition)

Shared memory synchronization:

- There are two essential needs for synchronization between multiple processes executing on shared memory

- Establishing an order between two events
  - E.g. in the server and client case, we want to make sure the server finishes writing before the client reads
- Mutually exclusive access to a certain resource
  - Such as a data structure, a file, etc
  - E.g. Two people deposit to the same account "deposit +=100". We want to make sure that the increment happens one at a time. Why? (Let us look draw a time line showing possible interleaving of events)

➢ A semaphore can be used for both purposes
➢ An ordinary while loop (busy wait loop) is not safe for ensuring mutual exclusion
- Two processes may both think they have successfully set the lock and, so, have the exclusive access
  - Again, we can draw a time line showing possible interleaving of events that may lead to failed mutual exclusion
- A semaphore is guaranteed to be able to have the correct view of the locking status. Table 10.2 shows solution for problem in Table 10.1.

The concept of semaphores:

- Semaphores may be binary (0/10), or counting
- Every semaphore variable, s, it is initialized to some positive value
  - 1 for a binary semaphore
  - N > 1for a counting semaphore

Binary semaphore:

- A binary semaphore, s, is used for mutual exclusion and wake up sync
  1== unlocked
  0==locked
- s, is associated with two operations:
  - P(s)
    – Tests s; if positive, resets s to 0 and proceed; otherwise, put the executing process to the back of a waiting queue for s
  - V(s)
    – Set s to 1 and wake up a process in the waiting queue for s

Counting Semaphore:

- A counting semaphore, s, is used for producer/consumer sync
  n== the count of available resources
  0==no resource (locking consumers out)
- s, is associated with two operations:
  - P(s)

– Tests s; if positive, decrements s and proceed
– otherwise, put the executing process to the back of a waiting queue for s

- V(s)
  – Increments s; wakes up a process, if any, in the waiting queue for s

Process 1	register1	**Process2**	register2	**counter**
sem_wait(&semA);		…		5
…		sem_wait(&semA);		5
register1 = counter	5	/*blocked*/	…	5
register1 = register1 + 1	6	/*blocked*/	…	5
counter = register1	6	/*blocked*/	…	6
sem_post(&semA)		/*blocked*/	…	6
	…	register2 = counter	6	6
	…	register2 = register2 – 1	5	6
	…	counter = register2	5	5
		sem_post(&semA)		

Table 10.2 (Solution for Race Condition)

Critical Sections:

- We like to think of locking a concurrent data structure
- In current practice, however, locks (incl. binary semaphores) are typically used to lock a segment of program statements (or instructions)
- Such a program segment is called a critical section
  - A critical section is a program segment that may modify shared data structures
  - It should be executed by one process at any given time
- With a binary semaphore
  - If multiple processes are locked out of a critical section
  - As soon as the critical section is unlocked, only one process is allowed in
  - The other processes remain locked out
- Implementation of semaphores is fair to processes
  - A first-come-first-serve queue

Unix Semaphores:

- There are actually at least two implementations
- UNIX System V has an old implementation
  - Analogous to shared memory system calls
  - Calls to semget(), semat(), semctl(), etc
  - Not as easy to use as POSIX implementation
- We will use POSIX implementation in this course

POSIX Semaphore System Calls:

- #include <semaphore.h>
- POSIX semaphores come in two forms: named semaphores and unnamed semaphores.

Using Unnamed Semaphore:

- Unnamed semaphores are also called memory- based semaphores
  - Named semaphores are "file-based"
- An unnamed semaphore does not have a name
  - It is placed in a region of memory that is shared between multiple threads (a thread-shared semaphore) or processes (a process-shared semaphore).
- A process-shared semaphore must be placed in a shared memory region

System Calls:

- Before being used, an unnamed semaphore must be initialized using *sem_init(3)*. It can then be operated on using *sem_post(3)* and *sem_wait(3)*
- When the semaphore is no longer required, and before the memory in which it is located is deallocated, the semaphore should be destroyed using ***sem_destroy(3)***
- Compile using -lrt

Recall that shared memory segments must be removed before program exits

- "An unnamed semaphore should be destroyed with sem_destroy() before the memory in which it is located is deallocated."
- "Failure to do this can result in resource leaks on some implementations."

int sem_init(sem_t *sem, int pshared, unsigned int value);

- #include <semaphore.h>
- **sem_init**() initializes the unnamed semaphore at the address pointed to by *sem*. The *value* argument specifies the initial value for the semaphore.
- If pshared has the value 0, then the semaphore is shared between the threads of a process.
- If pshared is nonzero, then the semaphore is shared between processes, and should be located in a region of shared memory.

int sem_wait(sem_t *sem);

- sem_wait() decrements (locks) the semaphore pointed to by *sem*
- If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately.

- If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

int sem_post(sem_t *sem);

- sem_post() increments (unlocks) the semaphore pointed to by *sem*.
- If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a ***sem_wait****(3)* call will be woken up

int sem_destroy(sem_t *sem);

- Destroys the unnamed semaphore at the address pointed to by *sem*. Only a semaphore that has been initialized by *sem_init(3)* should be destroyed using sem_destroy().
- Destroying a semaphore that other processes or threads are currently blocked on (in ***sem_wait****(3)* produces undefined behavior.
- Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized using ***sem_init****(3)*.

Examples:

- Example1

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t mutex;

void* thread(void* arg)
{
//wait
sem_wait(&mutex);
printf("\nEntered..\n");
//critical section
 sleep(4);
//signal
printf("\nJust Exiting...\n");
sem_post(&mutex);
}
int main()
{
sem_init(&mutex, 0, 1);
pthread_t t1,t2;
pthread_create(&t1,NULL,thread,NULL);
sleep(2);
```

```
pthread_create(&t2,NULL,thread,NULL);
pthread_join(t1,NULL);
pthread_join(t2,NULL);
sem_destroy(&mutex);
return 0;
}
```

- Example2

```
#include <sys/wait.h>
#include <stdlib.h>
#include <assert.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
int main(int argc, char **argv)
{
int i, nloop=10, *ptr;
sem_t mutex;
int shmid2,shmid1;
int SHMSIZE=1024;
sem_t *p_mutex;
if((shmid2 = shmget(IPC_PRIVATE,SHMSIZE,0666))<0)
{
perror("shmget");
exit(1);
}
p_mutex = (sem_t*) shmat(shmid2,NULL,0);
if(p_mutex==(sem_t *)-1)
{
perror("mutex shmat fails");
exit(0);
}
if(p_mutex == (sem_t*)-1)
{
perror("semaphore initialization");
exit(0);
}
if (fork()==0)
{
sem_wait(p_mutex);
for (i=0;i<nloop;i++)
printf("child: %d\n",(*ptr)++);
sem_destroy(p_mutex);
shmctl(shmid2,IPC_RMID,(struct shmid_ds*)0);
shmctl(shmid1,IPC_RMID,(struct shmid_ds*)0);
exit(0)};
```

```
sem_post(p_mutex);
for (i=0;i<nloop;i++)
printf("parent: %d\n",(*ptr)++);
exit(0);
return 0;
}
```

EXPERIMENT 10
Memory Mapped Files

OBJECTIVE:

- Learn the concept of Memory Mapped Files

BACKGROUND:

Memory Mapped Files:

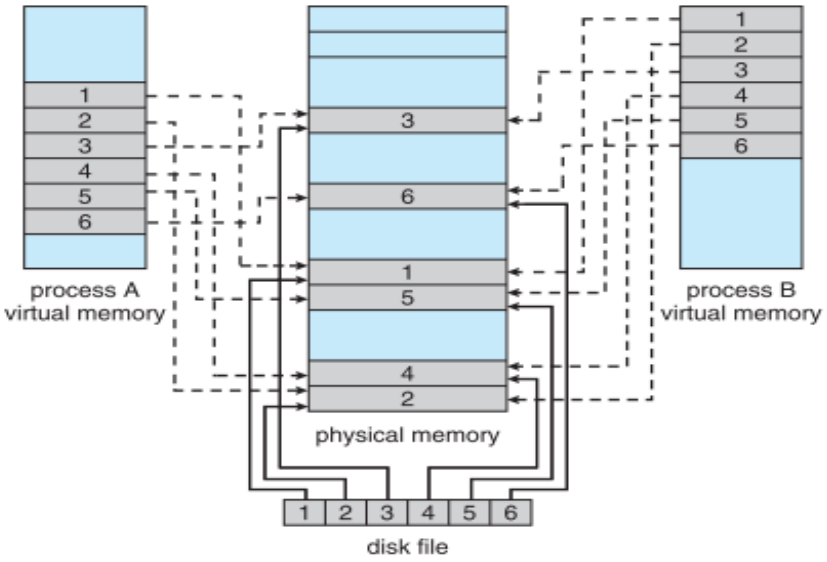


Figure 11.1 (Memory Mapping)

Memory mapping a file is accomplished by mapping a disk block to a page (or pages) in memory. Subsequent reads and writes to the file are handled as routine memory accesses. Manipulating files through memory rather than incurring the overhead of using the read() and write() system calls simplifies and speeds up file access and usage. Figure 11.1 shows memory mapping for two processes A and B.

Creating a Memory Map:

void *mmap(void *addr, size_t len, int prot, int flags, int fields, off_t off);

- **addr:** This is the address we want the file mapped into. The best way to use this is to set it to **(caddr_t)0 or NULL** and let the OS choose it for you. If you tell it to use an address the OS doesn't like (for instance, if it's not a multiple of the virtual memory page size), it'll give you an error.
- **len:** This parameter is the length of the data we want to map into memory. This can be any length you want. (Aside: if len not a multiple of the virtual memory page size, you will get a block size that is rounded up to that size. The extra bytes will be 0, and any changes you make to them will not modify the file.)

- **prot:** The "protection" argument allows you to specify what kind of access this process has to the memory mapped region. PROT_READ, PROT_WRITE, and PROT_EXEC, for read, write, and execute permissions, respectively. The value specified here must be equivalent to the mode specified in the **open()** system call that is used to get the file descriptor.
- **flags:** You'll want to set it to MAP_SHARED if you're planning to share your changes to the file with other processes, or MAP_PRIVATE otherwise. If you set it to the latter, your process will get a copy of the mapped region, so any changes you make to it will not be reflected in the original file—thus, other processes will not be able to see them.
- **fields:** This is where you put that file descriptor you opened earlier.
- **off:** This is the offset in the file that you want to start mapping from. A restriction: this *must* be a multiple of the virtual memory page size. This page size can be obtained with a call to **getpagesize()**.

Return value:

- **mmap()** returns a pointer to the mapped area. On error, the value **MAP_FAILED** is returned.

Munmap:

- int munmap(void *addr, size_t len);

- On success, munmap() returns 0. On failure, it returns -1.

Header File to be included:

#include <sys/mman.h>

For reading the memory map area:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <errno.h>
int main(int argc, char*argv[]){
int fd;
char* data;
if((fd = open("m.c",O_RDONLY)) == -1)
{
perror("open");
exit(1);
}
data = mmap(NULL, getpagesize(), PROT_READ,MAP_SHARED, fd,0);
printf("File contains: %s\n", data);
return 0;}
```

For writing in the memory mapped area:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <time.h>
int main(int argc, char*argv[])
{
int fd;
void* file_memory;
fd = open(argv[1], O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
/*creates the memory mapping */
file_memory = mmap(NULL,256,PROT_WRITE,MAP_SHARED,fd,0);
close(fd);
printf("%s", (char*) file_memory);
/*release the memory*/
munmap(file_memory,256);
return 0;
}
```

In-Lab Question:

Write C/C++ code for a program that takes as command line argument the file name and the substring to be found in file. Your program will make a memory map of the file and find the number of times the substring has occurred in the file. Create 2 threads for searching. First thread will search for substring in the first half, and the second thread will search for string in the second half of map. Whenever, a thread finds the string it increments the count of some shared variable “count”. Since count is being shared by both threads, you must synchronize the access using semaphore. After both threads have terminated, the main thread will print the count on the screen.

Example:

If the data in the file is “We went shopping on Sunday. There was hustle and bustle in the market. We also went shopping on Saturday.”, and the substring is “went shopping”; then your program must output 2.

EXPERIMENT 11
File Allocation Strategies

OBJECTIVE:

- Learn to simulate the following file allocation strategies
  a) Sequential
  b) Linked
  c) Indexed

BACKGROUND:

A file is a collection of data, usually stored on disk. As a logical entity, a file enables to divide data into meaningful groups. As a physical entity, a file should be considered in terms of its organization. The term "file organization" refers to the way in which data is stored in a file and, consequently, the method(s) by which it can be accessed.

Sequential File Allocation:

In this file organization, the records of the file are stored one after another both physically and logically. That is, record with sequence number 16 is located just after 15^{th} record. A record of a sequential file can only be accessed by reading all previous records.

Linked File Allocation:

With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and the last blocks of the file. Each block contains a pointer to the next block.

Indexed File Allocation:

Indexed file allocation strategy brings all the pointers together into one location: an index block. Each file has its own index block, which is an array of disk-block addresses. The i^{th} entry in the index block points to the i^{th} block of the file. The directory contains the address of the index block. To find and read the i^{th} block, the pointer in the i^{th} index-block entry is used.

Example:

Taking an example of Sequential file allocation we see through its simulation how records of file are to be stored one after the other. Each record in spread over different blocks which are all there in a sequence.

```
#include<stdio.h>
struct fileTable
{
    char name[20];
    int sb, nob;
}ft[30];

void main()
{
    int i, j, n;
    char s[20];
    printf("Enter no of files :");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("\nEnter file name %d :",i+1);
        scanf("%s",ft[i].name);
        printf("Enter starting block of file %d :",i+1);
        scanf("%d",&ft[i].sb);
        printf("Enter no of blocks in file %d :",i+1);
        scanf("%d",&ft[i].nob);
    }
    printf("\nEnter the file name to be searched -- ");
    scanf("%s",s);

    for(i=0;i<n;i++)
    if(strcmp(s, ft[i].name)==0)
        break;
    if(i==n)
        printf("\nFile Not Found");
    else
    {
      printf("\nFILE NAME   START BLOCK   NO OF BLOCKS   BLOCKS
OCCUPIED\n");
       printf("\n%s\t\t%d\t\t%d\t",ft[i].name,ft[i].sb,ft[i].nob);
      for(j=0;j<ft[i].nob;j++)
      printf("%d, ",ft[i].sb+j);
    }
}
```

Output:

Output for above code will be as follows

FILE NAME	START BLOCK	NO OF BLOCKS	BLOCKS OCCUPIED
B	102	4	102, 103, 104, 105

In-lab Questions

Question 1:

Write a program to implement simulation for Linked File Allocation. You can use given struct where output of this simulation will be as given below:

```
struct fileTable
{ char name[20];
int nob;
struct block *sb;
}ft[30];

struct block
{ int bno;
  struct block *next;
};
```

Output:

FILE NAME	NO OF BLOCKS	BLOCKS OCCUPIED
G	5	88 → 77→ 66→ 55→ 44

Question 2:

Write a program to implement simulation for Indexed File Allocation. Where struct and output of this simulation will be as given below:

```
struct fileTable
{ char name[20];
int nob, blocks[30];
}ft[30];
```

Output:

FILE NAME	NO OF BLOCKS	BLOCKS OCCUPIED
G	5	88, 77, 66, 55, 44

EXPERIMENT 12
File Organization Techniques

OBJECTIVE:

- Learn to simulate the following file organization techniques
  a) Single level directory
  b) Two level directory
  c) Hierarchical

BACKGROUND:

The directory structure is the organization of files into a hierarchy of folders. In a single-level directory system, all the files are placed in one directory. There is a root directory which has all files. It has a simple architecture and there are no sub directories. Advantage of single level directory system is that it is easy to find a file in the directory. In the two-level directory system, each user has own user file directory (UFD). The system maintains a master block that has one entry for each user. This master block contains the addresses of the directory of the users. When a user job starts or a user logs in, the system's master file directory (MFD) is searched. When a user refers to a particular file, only his own UFD is searched. This effectively solves the name collision problem and isolates users from one another. Hierarchical directory structure allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name. A directory (or subdirectory) contains a set of files or subdirectories.

Example: (Single Level Directory Organization)

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
struct
{
   char dname[10],fname[10][10];
   int fcnt;
}dir;

void main()
{  int i,ch;
  char f[30];
  dir.fcnt = 0;
  printf("\nEnter name of directory --    ");
  scanf("%s", dir.dname);
```

```
while(1)
  {
     printf("\n\n1. Create File\t2. Delete File\t3. Search File \n 4. Display Files\t5. Exit\nEnter your
choice -- ");
     scanf("%d",&ch);
     switch(ch)
     {
        case 1: printf("\nEnter the name of the file --   ");
        scanf("%s",dir.fname[dir.fcnt]);
        dir.fcnt++;
        break;
        case 2: printf("\nEnter the name of the file -- ");
        scanf("%s",f);
        for(i=0;i<dir.fcnt;i++)
        {
           if(strcmp(f, dir.fname[i])==0)
           {
              printf("File %s is deleted ",f);
              strcpy(dir.fname[i],dir.fname[dir.fcnt-1]);
              break;
            }
        }
        if(i==dir.fcnt)
        printf("File %s not found",f);
        else
        dir.fcnt--;
        break;
        case 3: printf("\nEnter the name of the file -- ");
        scanf("%s",f);
        for(i=0;i<dir.fcnt;i++)
        {
           if(strcmp(f, dir.fname[i])==0)
           {
              printf("File %s is found ", f);
              break;
            }
        }
        if(i==dir.fcnt)
        printf("File %s not found",f);
        break;
        case 4:  if(dir.fcnt==0)
        printf("\nDirectory Empty");
        else
        {
           printf("\nThe Files are --  ");
           for(i=0;i<dir.fcnt;i++)
           printf("\t%s",dir.fname[i]);
         }
         break;
         default: exit(0); } }  }
```

In-lab Questions

Question 1:

Write a program to implement simulation for Two Level Directory Organization.

Hint: You can use an array of struct variables.

Question 2:

Write a program to implement simulation for Hierarchical Directory Organization.

Hint: Pointer variables of struct may be used as there will be a tree like structure.

EXPERIMENT 13

The Readers and Writers Problem (Part A)

Problem Statement:

Synchronization has always been a problem, you can find multiple solutions on synchronization problem. You can observe a lot of systems around us where multiple processes try to access same database at the same time. Take airline reservation system or hotel reservation system for an example.

Your task is to develop a solution for The Readers and Writers Problem (A problem where multiple Processes wish to read or write. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers.).

EXPERIMENT 14

The Readers and Writers Problem (Part B)

Problem Statement:

Start with the solution developed for Experiment 13, write a program to implement that solution by creating a dummy database. After implementation, your task is to reflect your thoughts on the merits/demerits of the solution and suggest steps for improvement where required.

Appendix A: Lab Evaluation Criteria

Labs with projects

1. Experiments and their report 50%
    a. Experiment 60%
    b. Lab report 40%
2. Quizzes (3-4) 15%
3. Final evaluation 35-%
    a. Project Implementation 60%
    b. Project report and quiz 40%

Labs without projects

1. Experiments and their report 50%
    a. Experiment 60%
    b. Lab report 40%
2. Quizzes (3-4) 20%
3. Final Evaluation 30%
    a. Experiment. 60%
    b. Lab report, pre and post experiment quiz 40%

Notice:

Copying and plagiarism of lab reports is a serious academic misconduct. First instance of copying may entail ZERO in that experiment. Second instance of copying may be reported to DC. This may result in awarding FAIL in the lab course.

Appendix B: Safety around Electricity

In all the Electrical Engineering (EE) labs, with an aim to prevent any unforeseen accidents during conduct of lab experiments, following preventive measures and safe practices shall be adopted:

- Remember that the voltage of the electricity and the available electrical current in EE labs has enough power to cause death/injury by electrocution. It is around 50V/10 mA that the "cannot let go" level is reached. "The key to survival is to decrease our exposure to energized circuits."
- If a person touches an energized bare wire or faulty equipment while grounded, electricity will instantly pass through the body to the ground, causing a harmful, potentially fatal, shock.
- Each circuit must be protected by a fuse or circuit breaker that will blow or "trip" when its safe carrying capacity is surpassed. If a fuse blows or circuit breaker trips repeatedly while in normal use (not overloaded), check for shorts and other faults in the line or devices. Do not resume use until the trouble is fixed.
- It is hazardous to overload electrical circuits by using extension cords and multi-plug outlets. Use extension cords only when necessary and make sure they are heavy enough for the job. Avoid creating an "octopus" by inserting several plugs into a multi-plug outlet connected to a single wall outlet. Extension cords should ONLY be used on a temporary basis in situations where fixed wiring is not feasible.
- Dimmed lights, reduced output from heaters and poor monitor pictures are all symptoms of an overloaded circuit. Keep the total load at any one time safely below maximum capacity.
- If wires are exposed, they may cause a shock to a person who comes into contact with them. Cords should not be hung on nails, run over or wrapped around objects, knotted or twisted. This may break the wire or insulation. Short circuits are usually caused by bare wires touching due to breakdown of insulation. Electrical tape or any other kind of tape is not adequate for insulation!
- Electrical cords should be examined visually before use for external defects such as: Fraying (worn out) and exposed wiring, loose parts, deformed or missing parts, damage to outer jacket or insulation, evidence of internal

damage such as pinched or crushed outer jacket. If any defects are found the electric cords should be removed from service immediately.

- Pull the plug not the cord. Pulling the cord could break a wire, causing a short circuit.
- Plug your heavy current consuming or any other large appliances into an outlet that is not shared with other appliances. Do not tamper with fuses as this is a potential fire hazard. Do not overload circuits as this may cause the wires to heat and ignite insulation or other combustibles.
- Keep lab equipment properly cleaned and maintained.
- Ensure lamps are free from contact with flammable material. Always use lights bulbs with the recommended wattage for your lamp and equipment.
- Be aware of the odor of burning plastic or wire.
- ALWAYS follow the manufacturer recommendations when using or installing new lab equipment. Wiring installations should always be made by a licensed electrician or other qualified person. All electrical lab equipment should have the label of a testing laboratory.
- Be aware of missing ground prong and outlet cover, pinched wires, damaged casings on electrical outlets.
- Inform Lab engineer / Lab assistant of any failure of safety preventive measures and safe practices as soon you notice it. Be alert and proceed with caution at all times in the laboratory.
- Conduct yourself in a responsible manner at all times in the EE Labs.
- Follow all written and verbal instructions carefully. If you do not understand a direction or part of a procedure, ASK YOUR LAB ENGINEER / LAB ASSISTANT BEFORE PROCEEDING WITH THE ACTIVITY.
- Never work alone in the laboratory. No student may work in EE Labs without the presence of the Lab engineer / Lab assistant.
- Perform only those experiments authorized by your teacher. Carefully follow all instructions, both written and oral. Unauthorized experiments are not allowed.

- Be prepared for your work in the EE Labs. Read all procedures thoroughly before entering the laboratory. Never fool around in the laboratory. Horseplay, practical jokes, and pranks are dangerous and prohibited.
- Always work in a well-ventilated area.
- Observe good housekeeping practices. Work areas should be kept clean and tidy at all times.
- Experiments must be personally monitored at all times. Do not wander around the room, distract other students, startle other students or interfere with the laboratory experiments of others.
- Dress properly during a laboratory activity. Long hair, dangling jewelry, and loose or baggy clothing are a hazard in the laboratory. Long hair must be tied back, and dangling jewelry and baggy clothing must be secured. Shoes must completely cover the foot.
- Know the locations and operating procedures of all safety equipment including fire extinguisher. Know what to do if there is a fire during a lab period; “Turn off equipment, if possible and exit EE lab immediately.”

Appendix C: Guide lines on Preparing Lab Reports

Each student will maintain a lab notebook for this lab course. You will write a report for each experiment you perform in your notebook.

OS Lab Report Format

The format of the report will be as given below:

1. **Introduction**: Introduce the new commands being used, and their significance.
2. **Objective**: What are the learning goals of the experiment?
3. **Design**: If applicable, draw the flow chart for the program. How do the new constructs facilitate achievement of the Objective; if possible, a comparison in terms of efficacy and computational tractability with the alternate constructs?
4. **Issues**: The bugs encountered and the way they were removed.
5. **Conclusions**: What conclusions can be drawn from experiment?
6. Answers to post lab questions (if any).