OPERATING SYSTEM LAB FILE

Submitted by

Vaibhav Srivastava

A41890819038

Computer Engineering (DIPLOMA) 2019-2022

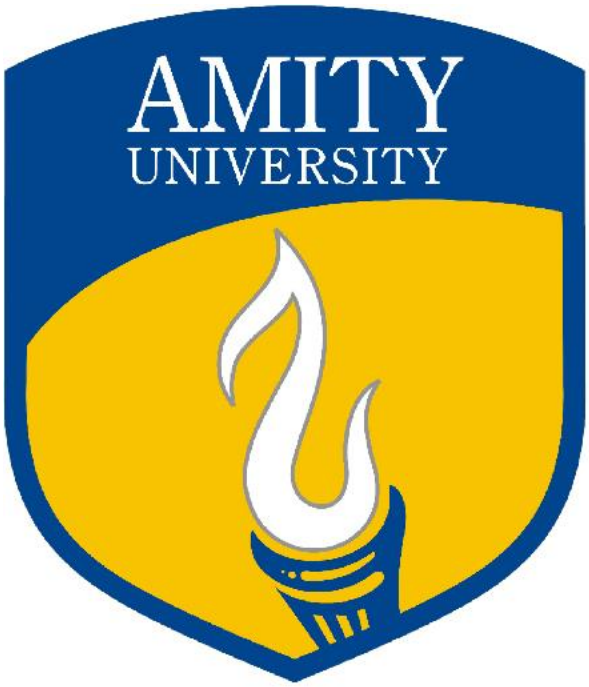
In Partial fulfilment toward the award

Of

DIPLOMA

In

Computer Engineering



DEPARTMENT OF COMPUTER ENGINEERING

AMITY UNIVERSITY GREATER NOIDA
GAUTAM BUDDH NAGAR

CONTENTS OF OPERATING SYSTEM LAB MANUAL

PREAMBLE

➢ What are virtual labs?

Virtual labs are simulated learning environments that allow students to complete laboratory experiments online and explore concepts and theories without stepping into a physical science lab.

Students can try out lab techniques for the first time and become more familiar with advanced lab equipment that might otherwise be inaccessible.

Through online compilers and Linux server, students can explore c/c++ coding and UNIX commands online.

Virtual lab software creates opportunities for alternative access to science education.

➢ Objectives:

1. To provide remote-access to Labs in various disciplines of Science and Engineering. These Virtual Labs would cater to students at the undergraduate level, post graduate level as well as to research scholars.

2. To enthuse students to conduct experiments by arousing their curiosity. This would help them in learning basic and advanced concepts through remote experimentation.

3. To provide a complete Learning Management System around the Virtual Labs where the students can avail the various tools for learning, including additional web-resources, video-lectures, animated demonstrations and self evaluation.

4. To share costly equipment and resources, which are otherwise available to limited number of users due to constraints on time and geographical distances.

➢ 6 ways to use virtual labs:

1. As a visual aid to teach complex concepts
2. To refresh students' knowledge before teaching new material
3. As a pre-lab exercise
4. To provide lab work to courses with no existing lab component
5. To facilitate online learning
6. As a post-lab exercise

➢ 6 challenges in offline education (and how virtual labs can help solve them):

Challenge #1: Limited lab access
Challenge #2: Limited time in the lab
Challenge #3: Low student motivation and engagement
Challenge #4: Teaching complex topics without visuals
Challenge #5: Making mistakes in high risk environments
Challenge #6: Unprepared students and knowledge gaps

➢ What hardware and software do I need?

AMITY virtual lab can be used on any laptop, Chromebook, or desktop computer that meets our minimum system requirements.
AMITY virtual lab simulations run directly in the browser, and no plugins or installations are required.

➢ Minimum System Requirements:

Processor: Dual core 2 GHz or higher
Memory: 4 GB or more
Graphic card: Intel HD 3000 / GeForce 6800 GT / Radeon X700 or higher
OS: Latest version of Windows (64-bit) or Mac OS or ChromeOS
Supported browsers: Latest versions of Firefox and Chrome
A stable internet Connection

➢ The Philosophy:

Physical distances and the lack of resources make us unable to perform experiments, especially when they involve sophisticated instruments.
Also, good teachers are always a scarce resource.
Web-based and video-based courses address the issue of teaching to some extent.
Conducting joint experiments by two participating institutions and also sharing costly resources has always been a challenge.
With the present day internet and computer technologies the above limitations can no more hamper students and researchers in enhancing their skills and knowledge.
Also, in a country such as ours, costly instruments and equipment need to be shared with fellow researchers to the extent possible.
Web enabled experiments can be designed for remote operation and viewing so as to enthuse the curiosity and innovation into students.
This would help in learning basic and advanced concepts through remote experimentation.
Today most equipment has a computer interface for control and data storage.
It is possible to design good experiments around some of this equipment which would enhance the learning of a student.
Internet-based experimentation further permits use of resources, knowledge, software, and data available on the web, apart from encouraging skillful experiments being simultaneously performed at points separated in space (and possibly, time).

➢ Salient Features:

1. Virtual Labs will provide to the students the result of an experiment by one of the following methods (or possibly a combination)

- ✓ Modeling the physical phenomenon by a set of equations and carrying out simulations to yield the result of the particular experiment. This can, at-the-best, provide an approximate version of the ‘real-world’ experiment.

- ✓ Providing measured data for virtual lab experiments corresponding to the data previously obtained by measurements on an actual system.

- ✓ Remotely triggering an experiment in an actual lab and providing the student the result of the experiment through the computer interface. This would entail carrying out the actual lab experiment remotely.

2. Virtual Labs will be made more effective and realistic by providing additional inputs to the students like accompanying audio and video streaming of an actual lab experiment and equipment.

➢ SUBJECT AREA OF MANUAL:

1. To understand how various operating system work
2. To understand some important algorithms of CPU scheduling.
3. To familiarize how certain applications can benefit from the choice of algorithms in operating systems.
4. To understand how the choice of scheduling can lead to efficient implementations of algorithms.

➢ ABOUT OPERATING SYSTEM VIRTUAL LAB:

A. To understand how various operating system work

B. To understand how UNIX commands works

C. To implement various scheduling problems in C language.

➢ HYPERLINK FOR AMITY VIRTUAL LAB:

http://202.12.103.135/vlab/

http://www.putty.org

PRAC-1	BASICS OF UNIX COMMANDS
	INTRODUCTION TO UNIX/LINUX

AIM:

To study about the basics of UNIX/ LINUX

UNIX:

It is a multi-user operating system. Developed at AT & T Bell Industries, USA in 1969.

Ken Thomson along with Dennis Ritchie developed it from MULTICS (Multiplexed Information and Computing Service) OS.

By1980, UNIX had been completely rewritten using C language.

LINUX:

It is similar to UNIX, which is created by Linus Torualds. All UNIX commands works in Linux. Linux is a open source software. The main feature of Linux is coexisting with other OS such as windows and UNIX.

STRUCTURE OF A LINUXSYSTEM:

It consists of three parts.

a) UNIX kernel
b) Shells
c) Tools and Applications

UNIX KERNEL:

Kernel is the core of the UNIX OS. It controls all tasks, schedule all Processes and carries out all the functions of OS.

Decides when one programs stops and another starts.

SHELL:

Shell is the command interpreter in the UNIX OS. It accepts command from the user and analyses and interprets them.

PRAC-2	BASICS OF UNIX COMMANDS
	EXPLORING UNIX COMMANDS

AIM:

To explore basic UNIX Commands like date, history, man, who, cut, cp, ls, mv, chmod, more, cd, mkdir, rmdir, pwd, banner, TTY, cat, touch, finger, ps, rm, lp, mail, sort

CONTENT:

Note: Syn->Syntax

a) date

–used to check the date and time

Syn:$date

Format	Purpose	Example	Result
+%m	To display only month	$date+%m	06
+%h	To display month name	$date+%h	June
+%d	To display day of month	$date+%d	O1
+%y	To display last two digits of years	$date+%y	09
+%H	To display hours	$date+%H	10
+%M	To display minutes	$date+%M	45
+%S	To display seconds	$date+%S	55

b) cal

–used to display the calendar

Syn:$cal 2 2009

c)echo

–used to print the message on the screen.

Syn:$echo “text”

d) ls

–used to list the files. Your files are kept in a directory.

Syn:$lsls–s

e)lp

–used to take printouts

Syn:$lp filename

f)man

–used to provide manual help on every UNIX commands.

Syn:$man unix command

$man cat

g)who & **whoami**

–it displays data about all users who have logged into the system currently. The next command displays about current user only.

Syn:$who

$whoami

h) uptime

–tells you how long the computer has been running since its last reboot or power-off.

Syn:$uptime

FILE MANIPULATION COMMANDS

a) **cat**–this create, view and concatenate files.

Creation:
Syn:$cat>filename

Viewing:
Syn:$cat filename

Add text to an existing file:
Syn:$cat>>filename

Concatenate:
Syn:$catfile1file2>file3
$catfile1file2>>file3 (no over writing of file3)

b) **rm**–deletes a file from the file system
Syn:$rm filename

c) **touch**–used to create a blank file.
Syn:$touch file names

d) **cp**–copies the files or directories
Syn:$cpsource file destination file
Eg:$cp student stud

e) **mv**–to rename the file or directorysyn:$mv old file new file
Eg:$mv–i student student list(-i prompt when overwrite)

f) **cut**–it cuts or pickup a given number of character or fields of the file.
Syn:$cut<option><filename>
Eg: $cut –c filename
$cut–c1-10emp
$cut–f 3,6emp
$ cut –f 3-6 emp
-c cutting columns
-f cutting fields

g) **chmod**–used to change the permissions of a file or directory.
Syn:$ch mod
Category operation permission file Where, Category–is the user type
Operation–is used to assign or remove permission
Permission–is the type of permission
File–are used to assign or remove permission all

Examples:
$chmodu-wx student
Removes write and execute permission for users
$ch modu+rw,g+rwstudent
Assigns read and write permission for users and groups
$chmodg=rwx student
Assigns absolute permission for groups of all read, write and execute permissions

h) **wc**–it counts the number of lines, words, character in a specified file(s) with the options as –l,-w,-c

Category	Operation	Permission
u– users g–group o– others	+assign -remove =assign absolutely	r– read w– write x-execute

Syn: $wc –l filename
$wc –w filename
$wc–c filename

i) **banner:** command in Linux is used to print the ASCII character string in large letter to standard output.
Syn:
banner text

PRAC-3	CPU SCHEDULING ALGORITHMS
	PRIORITY

AIM:

To write a C program for implementation of Priority scheduling algorithms.

ALGORITHM:

Step 1: Inside the structure declare the variables.
Step 2: Declare the variable i,j as integer, totwtime and totttime is equal to zero.
Step 3: Get the value of „n‟ assign p and allocate the memory.
Step 4: Inside the for loop get the value of burst time and priority.
Step 5: Assign wtime as zero .
Step 6: Check p[i].pri is greater than p[j].pri .
Step 7: Calculate the total of burst time and waiting time and assign as turnaround time.
Step 8: Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<stdio.h>
#include<stdlib.h>
typedef struct
{
int pno;
int  pri;
int  pri;
int btime;
int wtime;
}sp;
int main()
{
int i,j,n;
int tbm=0,totwtime=0,totttime=0;
sp *p,t;
printf("\n PRIORITY SCHEDULING.\n");
printf("\n enter the no of process. ..\n");
scanf("%d",&n);
p=(sp*)malloc(sizeof(sp));
printf("enter the burst time and priority:\n");
for(i=0;i<n;i++)
{
printf("process%d:",i+1);
scanf("%d%d",&p[i].btime,&p[i].pri);
 p[i].pno=i+1;
```

```
p[i].wtime=0;
}
for(i=0;i<n-1;i++)
for(j=i+1;j<n;j++)
{
if(p[i].pri>p[j].pri)
{
t=p[i];
p[i]=p[j];
p[j]=t;
}
}
printf("\n process\tbursttime\twaiting time\tturnaround time\n");
for(i=0;i<n;i++)
{
totwtime+=p[i].wtime=tbm;
tbm+=p[i].btime;
printf("\n%d\t\t%d",p[i].pno,p[i].btime);
printf("\t\t%d\t\t%d",p[i].wtime,p[i].wtime+p[i].btime);
}
totttime=tbm+totwtime;
printf("\n total waiting time:%d",totwtime);
printf("\n average waiting time:%f",(float)totwtime/n);
printf("\n total turnaround time:%d",totttime);
printf("\n avg turnaround time:%f",(float)totttime/n);
}
```

PRAC-4	CPU SCHEDULING ALGORITHMS
	ROUND ROBIN SCHEDULING

AIM:

To write a C program for implementation of Round Robin scheduling algorithms.

ALGORITHM:

Step 1: Inside the structure declare the variables.
Step 2: Declare the variable i,j as integer, totwtime and totttime is equal to zero.
Step 3: Get the value of „n‟ assign p and allocate the memory.
Step 4: Inside the for loop get the value of burst time and priority and read the time quantum.
Step 5: Assign wtime as zero.
Step 6: Check p[i].pri is greater than p[j].pri .
Step 7: Calculate the total of burst time and waiting time and assign as turnaround time.
Step 8: Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
struct rr
{
int pno,btime,sbtime,wtime,lst;
}p[10];
int main()
{
int pp=-1,ts,flag,count,ptm=0,i,n,twt=0,totttime=0;
printf("\n round robin scheduling.....");
printf("enter no of processes:");
scanf("%d",&n);
printf("enter the time slice:");
scanf("%d",&ts);
printf("enter the burst time");
for(i=0;i<n;i++)
{
printf("\n process%d\t",i+1);
scanf("%d",&p[i].btime);
p[i].wtime=p[i].lst=0;
p[i].pno=i+1;
p[i].sbtime=p[i].btime;
 }
```

```
printf("scheduling ...\n");
do
{
flag=0;
for(i=0;i<n;i++)
{
count=p[i].btime;
if(count>0)
{
flag=-1;
count=(count>=ts)?ts:count;
printf("\n process %d",p[i].pno);
printf("from%d",ptm);
ptm+=count;
printf("to%d",ptm);
p[i].btime-=count;
if(pp!=i)
{
pp=i;
p[i].wtime+=ptm-p[i].lst-count;
p[i].lst=ptm;
}
}
```

PRAC-5	CPU SCHEDULING ALGORITHMS
	FIRST CUM FIRST SERVE (FCFS)

AIM:

To write a C program for implementation of FCFS scheduling algorithm.

ALGORITHM:

Step 1: Inside the structure declare the variables.
Step 2: Declare the variable i,j as integer,totwtime and totttime is equal to zero.
Step 3: Get the value of „n‟ assign pid as I and get the value of p[i].btime.
Step 4: Assign p[0] wtime as zero and tot time as btime and inside the loop calculate wait time and turnaround time.
Step 5: Calculate total wait time and total turnaround time by dividing by total number of process.
Step 6: Print total wait time and total turnaround time.
Step 7: Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
struct fcfs
{
int pid;
int btime;
int wtime;
int ttime;
}
p[10];
int main()
{
int i,n;
int towtwtime=0,totttime=0;
printf("\n fcfs scheduling...\n");
printf("enter the no of process");
scanf("%d",&n);
for(i=0;i<n;i++)
{
p[i].pid=1;
printf("\n burst time of the process”);
scanf("%d",&p[i].btime);
}
```

```
p[0].wtime=0;
p[0].ttime=p[0].btime;
totttime+=p[i].ttime;
for(i=0;i<n;i++)
{
p[i].wtime=p[i-1].wtime+p[i-1].btim
p[i].ttime=p[i].wtime+p[i].btime;
totttime+=p[i].ttime;
towtwtime+=p[i].wtime;
}
for(i=0;i<n;i++)
{{
printf("\n waiting time for process”);
printf("\n turn around time for process”);
printf("\n");
}}
printf("\n total waiting time :%d", totwtime );
printf("\n average waiting time :%f",(float)totwtime/n);
printf("\n total turn around time :%d",totttime);
printf("\n average turn around time: :%f",(float)totttime/n);
}
```

PRAC-6	PRODUCER CONSUMER PROBLEM USING SEMAPHORES

AIM:

To write a C-program to implement the producer – consumer problem using semaphores.

ALGORITHM:

Step 1: Start the program.
Step 2: Declare the required variables.
Step 3: Initialize the buffer size and get maximum item you want to produce.
Step 4: Get the option, which you want to do either producer, consumer or exit from the operation.
Step 5: If you select the producer, check the buffer size if it is full the producer should not produce the item or otherwise produce the item and increase the value buffer size.
Step 6: If you select the consumer, check the buffer size if it is empty the consumer should not consume the item or otherwise consume the item and decrease the value of buffer size.
Step 7: If you select exit come out of the program.
Step 8: Stop the program.

PROGRAM:

```
#include<stdio.h>

{
int n;
void producer();
void consumer();
int wait(int);
int signal(int);
printf("\n1.PRODUCER\n2.CONSUMER\n3.EXIT\n");
while(1) {
printf("\nENTER YOUR CHOICE\n");
scanf("%d",&n);
switch(n)
{ case 1:
if((mutex==1)&&(empty!=0))
producer();
else
printf("BUFFER IS FULL");
break;
```

```
case 2:
if((mutex==1)&&(full!=0))
consumer();
else
printf("BUFFER IS EMPTY");
break;
case 3:
exit(0);
break;
}
}
}
int wait(int s) {
return(--s); }
int signal(int s) {
return(++s); }
void producer() {
mutex=wait(mutex);
full=signal(full);
empty=wait(empty);
x++;
printf("\nproducer produces the item%d",x);
mutex=signal(mutex); }
void consumer() {
mutex=wait(mutex);
full=wait(full);
empty=signal(empty);
printf("\n consumer consumes item%d",x);
x--;
mutex=signal(mutex); }
```

PRAC-7	DINING PHILOSOPHER PROBLEM USING SEMAPHORES

AIM:

To write a C-program to implement the dining philosopher problem using semaphores.

The Dining Philosopher Problem:

The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

Semaphore Solution to Dining Philosopher –
Each philosopher is represented by the following pseudocode:

PSEUDOCODE:

```
process P[i]
 while true do
   {  THINK;
      PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
      EAT;
      PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
   }
```

There are three states of the philosopher: THINKING, HUNGRY, and EATING. Here there are two semaphores: Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or putdown at the same time. The array is used to control the behavior of each philosopher. But, semaphores can result in deadlock due to programming errors.

PROGRAM:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define N 5
#define THINKING 2
#define HUNGRY 1
```

```
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };

sem_t mutex;
sem_t S[N];

void test(int phnum)
{
    if (state[phnum] == HUNGRY
            && state[LEFT] != EATING
            && state[RIGHT] != EATING) {
            // state that eating
            state[phnum] = EATING;

            sleep(2);

            printf("Philosopher %d takes fork %d and %d\n",
                                    phnum + 1, LEFT + 1, phnum + 1);

            printf("Philosopher %d is Eating\n", phnum + 1);

            // sem_post(&S[phnum]) has no effect
            // during takefork
            // used to wake up hungry philosophers
            // during putfork
            sem_post(&S[phnum]);
    }
}

// take up chopsticks
void take_fork(int phnum)
{

    sem_wait(&mutex);

    // state that hungry
    state[phnum] = HUNGRY;

    printf("Philosopher %d is Hungry\n", phnum + 1);

    // eat if neighbours are not eating
    test(phnum);
```

```
    sem_post(&mutex);

    // if unable to eat wait to be signalled
    sem_wait(&S[phnum]);

    sleep(1);
}

// put down chopsticks
void put_fork(int phnum)
{

    sem_wait(&mutex);

    // state that thinking
    state[phnum] = THINKING;

    printf("Philosopher %d putting fork %d and %d down\n",
            phnum + 1, LEFT + 1, phnum + 1);
    printf("Philosopher %d is thinking\n", phnum + 1);

    test(LEFT);
    test(RIGHT);

    sem_post(&mutex);
}

void* philospher(void* num)
{

    while (1) {

            int* i = num;

            sleep(1);

            take_fork(*i);

            sleep(0);

            put_fork(*i);
    }
}

int main()
{
```

```
    int i;
    pthread_t thread_id[N];

    // initialize the semaphores
    sem_init(&mutex, 0, 1);

    for (i = 0; i < N; i++)

            sem_init(&S[i], 0, 0);

    for (i = 0; i < N; i++) {

            // create philosopher processes
            pthread_create(&thread_id[i], NULL,
                                    philospher, &phil[i]);

            printf("Philosopher %d is thinking\n", i + 1);
    }

    for (i = 0; i < N; i++)

            pthread_join(thread_id[i], NULL);
}
```