TAship & Laboratory Instruction

on

Operating Systems

for

B Tech CSE

IInd year, IIIrd semester

(Report)

Instructor

Arya Tanmay Gupta

M Tech CSE,

Indian Institute of Information Technology Vadodara, Gandhinagar

201861003@iiitvadodara.ac.in

Preface

This is the report of my experience as a Teaching Assistant cum Laboratory Instructor for B. Tech. CSE 3rd semester students at Indian Institute of Information Technology, Vadodara (IIITV). I instruct Operating Systems Laboratory to the students.

This document includes the activities that we, I and my students, have organized to make the learning experience enjoyable. It includes the experiments we worked on, the mistakes we did and the steps we took to rectify those errors which made us learn the subject even better by visualizing those problems from the errors that we did deliberately, sometimes unknowingly too.

I pursue the M Tech course at IIITV because of my passion for learning and teaching computer science and mathematics. I like to teach what I learn. I have already delivered various workshops on Java, Embedded Systems, Robotics and Artificial Intelligence. I was a Teaching Assistant at Ramanujan College (University of Delhi) during my fourth year B. Tech. Computer Science there. I used to instruct the students on C++ Programming to first year B Sc students, and then in the next semester, I taught them Java. This charge is an extension of what I dream of and realize, creating knowledge for me and others.

The errors that I did during the laboratory helped me elaborate some aspects of the Linux Operating System and related programming which otherwise would seem hard to accept in first go.

The reader

The reader might be a professor, one of my teachers, who might want to critically analyze my approach, the methods I followed, or the mistakes that I did during my tenure as a Teaching Assistant, or a Laboratory Instructor.

The reader might also be a Laboratory Instructor who might improve her quality as compared to my approach. She might already be better than me with respect to the concepts and teaching abilities, but on the other hand, she might also want to analyze my approach and try to add or subtract some procedures that I took.

I welcome all the readers and you are welcome to contact me freely for any comments, suggestions or critique.
The major procedures and approach to simulate various aspects of a general purpose Operating System that I demonstrated explicitly are given as separate headings in the major content. The related program codes are given in the Appendices. The majority of the codes that are in the report are programmed by the students in the laboratory itself; all credits are cited.

Acknowledgements

I must thank IIITV administration for all the support and giving me this opportunity. I thank my students, the B. Tech. CSE IInd year, to support me and attend my instructions throughout the semester, especially those who allowed me to add their programs to my Assistantship report. I thank Prof Prasenjit Majumder for his guidance and support.

Arya Tanmay Gupta

Major Contents of Instructions of Operating Systems Laboratory

1. Introduction to the course

2. Overview of OS, kernel, processes, Introduction to fork() and exec()

3. Repeat, last lab

4. CPU construction, kernel - fork() and exec(), file properties

5. Process Scheduling - FCFS, SJF

6. SRTF, Introduction to threads

7. Shell programming, Introduction to Critical Section, Peterson's Solution

8. File Access Control Mechanism - retrieval and modification

9. Peterson's Solution and Banker's Algorithm

10. Dining Philosopher's problem

11. Inter-process communication using shared memory

12. Contiguous memory allocation, first fit, best fit

13. Distributed Systems; Paging and Virtual Machines

14. Round Robin algorithm; Last class' revision

Appendices

A. Program Codes

B. Practical Examination

1. Introduction to the course

(August 6, 2018)

The students were given an Introduction to Operating Systems. I demonstrated them briefly how the hardware of a general purpose computing machine is organized, followed by discussing the functionality of a CPU, RAM.
I demonstrated them briefly the functionality provided by an operating system of modern times, the importance of the interface it provides. We discussed on what a process is, for an operating system, especially about the existence of a process on the hardware of the machine.
My main approach was to make them aware of the terminologies that they will be encountering in the coursework and the underlying concepts. We discussed how the concept of Operating Systems became significant. I also assured them that in due course, they will be able to appreciate the presence of this paper in their coursework more as they learn more.

2. Overview of OS, kernel, processes, Introduction to fork() and exec()

August 13, 2018

On this Monday, I demonstrated the students about the structure of an operating system. I started with the discussion about the history of computing machinery and the evolution of operating systems and the underlying hardware.
I then suddenly jumped on the current operating system on which we will be working on, Linux (also, Unix), explaining its attributes and general overview. We discussed the current hardware, and the layers of the kernel, Operating system, and application software. I demonstrated how a process comes to an existence inside a computer first from the start of the computer and then when we initiate some user process manually.
I demonstrated them the difference between the two major process types: user level and kernel level processes. I also demonstrated how kernel forks itself to handle a user process, and then it execs to join to a particular process and then returns back.

That might be a little deeper demonstration to be demonstrated at the start of an Operating Systems laboratory course, but I made sure that they do not get confused because of a lot of terminologies, so that when we discuss it again, they may be acquainted with the concepts, or otherwise also, while implementing various algorithms in future Instruction Labs, they have an idea about who (the OS) actually uses these algorithms and how does it utilizes them.

3. Repeat, last lab

August 20, 2018

Because of natural reasons (as they affirmed), only a few students came to this class. We planned for a remedial session for this class on Saturday of the same week, that is, August 25, 2018.

4. CPU construction, kernel - fork() and exec(), file properties

August 25, 2018

We discussed on how can we get information about a CPU's hardware using the file "***/proc/cpuinfo***". The program is given at Appendix A (1).
We discussed on fork() and exec(), their use and functionality - how and why does the kernel use fork and exec to enable other process to execute in its space. We can also use these concepts in our program. The code of fork() is given as Appendix A (2), and that of exec() is given as Appendix A (3). We also discussed the different ways exec, as execl, execlp, execle, execv, execvp, execvpe, can be implemented.
We also discussed shell programming. With reference to shell programming, we discussed on how to
(1) print data on terminal using echo
(2) create and use variables
(3) create a file
(4) write an if condition
(5) write file onto a file, instead of terminal

(6) search a file in a directory
We created a ***.c*** file using shell. We also compiled and run this c program using shell script. The code of this program is given as Appendix A (4).
We also discussed how to get stats of a file and modify its user permissions. The code is given at Appendix A (5). We discussed the stat structure in the ***sys/stat.h*** library, its variables and the amount of information about a file that we can obtain using stat structure.
I demonstrated them how to change ownership of a file, whose program is given as Appendix A (6).

5. Process Scheduling - FCFS, SJF

August 27, 2018

We discussed about how an operating system might think while entertaining (actually, allocating CPU to) a ready queue based on their arrival time. The processes that arrive first will be executed first. If more and more processes arrive, the scheduler will choose from among the left processes that are waiting for execution in the ready queue - the process which arrived first will be selected execution first.
I demonstrated them the basic algorithm of FCFS, followed by its implementation by the students in C programming language. They all successfully implemented it in the lab. The assumed input consisted of an **ARRAY** consisting of the arrival times of processes, and another **ARRAY** which consisted of the burst time at the corresponding array index.
The code is at Appendix A (7).
After implementing FCFS program, we discussed on SJF process scheduling algorithm. Only, I was unable to demonstrate them in the time left of how we should imagine about and implement the procedure that must be followed to implement such a complex algorithm.
I then promised them to email them the basic implementation. I designed a code in C in such a way that they must understand what the code is exactly doing. My approach was that besides getting just the correct output - Gantt chart, turnaround time, etc, they must be able to correlate the functions and instructions within with what they have learned in theory classes. I mailed

them the code the same evening. We decided to discuss the algorithm in the next Laboratory, after understanding the code that I send, and also using the same procedure to implement RR scheduling algorithm.

6. SRTF, Introduction to threads

September 10, 2018

The code that I mailed them was of another scheduling algorithm, SRT, close to SJF, but one step ahead in typicality to implement. The code is in Appendix A (8), which was provided to them 2 weeks before this date. On this day, I explained to them how the procedure that I developed worked, and I asked them to apply the same procedure to implement SJF and RR algorithms.

The program utilizes a struct, whose each object represents a block in the Gantt chart. It contains 3 variables: (1) int pno, to store the index of the process which might be executed as per the Gantt chart, (2) int st, to store the start time of execution with respect to a particular block in the Gantt chart, (3) int et, to store the end time of execution with respect to a particular block in the Gantt chart, and (4) struct process *next, to act as a pointer while designing the link list of Gantt chart blocks while simulating the actual Gantt chart.

getMax() is used to return the index of the process in the input table with the maximum burst time left. This is to detect whether the algorithm needs to proceed further or to exit.

getMin() returns the index of the process which has minimum remaining burst time.

getNextTQ() returns the time for which the process with minimum remaining time should run. It depends on the arrival instant of the next process, and the required burst of the process to be executed; it returns the value whichever is earlier.

add() is used to construct a linked list of struct process objects. This list will represent the Gantt chart.

getNextArrTime() returns the index of the next arriving process.

Until a process is left in the process table, the program will get the index of the next arriving process if there is no process executing or in the ready queue; it then takes the index of the

process with minimum remaining burst time; it computes how much time that should run; an object is created with the process index, starting time, end time; then the burst time of that process is reduced respectively and the current time instant is updated.
We discussed on the functioning of threads using the program given at Appendix A (9).

7. Shell programming, Introduction to Critical Section, Peterson’s Solution

September 17, 2018

We discussed shell programming in detail. We did an emphasized discussion on
(1) variables
(2) if condiions
(3) loops
(4) getting list of files in a directory
(5) read from user
(6) operations - binary mathematical operations
(7) functions
(8) return from functions
The code is given at Appendix A (9).

We also discussed critical section and Peterson’s solution’s algorithm.

8. File Access Control Mechanism - retrieval and modification

September 24, 2018

This day was quite adventurous for us because of electricity malfunction. Although, we discussed on the mathematics behind timestamp and datetime because it was necessary for the values related to time that were being returned by the program that we discussed following this. We finally discussed how to change last access and modification time of a file using a C program. The reference code is given at Appendix A (11).

9. Peterson’s Solution and Banker’s Algorithm

October 1, 2018

We discussed implementation of Peterson’s algorithm and Banker’s algorithm in C. I demonstrated them how I think we can approach to simulate peterson’s solution for 1 core, 2 processes’ system. The students then implemented the code successfully. I then showed them that while a process is executing its critical section, the other process will not execute its citical section, it will wait until its turn comes. The code is given as Appendix A (12).
I then demonstrated them how we can approach to simulate a system which uses Banker’s algorithm to avoid deadlock in operating system. All the time was consumed so we decided to code it and present it in the next lab. I then imposed a condition, those who will present me the code, correct or incorrect will get attendance. All the students agreed to this. In the next lab we discuss the issues which the students are facing in the implementation of Banker’s algorithm.

10. Dining Philosopher’s problem

October 8, 2018

We discussed about the Dining philosophers’ problem. I demonstrated how can we approach towards implementation of solution to this problem using semaphores. The students implemented this program successfully. The code is given as Appendix A (14).
From last week, I reviewed their programs on avoidance of deadlocks using banker’s algorithm. The code is given as Appendix A (13).

11. Inter-process communication using shared memory

October 22, 2018

There are two methods in which processes communicate in a general purpose operating system, via shared memory and signaling. I demonstrated the students how processes communicate using

shared memory. We discussed about how to proceed to construct a program to model two processes which may communicate using shared memory.
We took a buffer, data counter (representing amount of buffer filled), starting index buffer pointer and ending index buffer pointer. The producer process adds data to buffer if there is space left and increments the ending index buffer pointer. The consumer process reads data from buffer if it is not empty and increments the starting index buffer pointer.
We simulated the two processes by creating two threads using the ***pthread.h*** library.
The C program is given as Appendix A (15).

12. Contiguous memory allocation, first fit, best fit

October 29, 2018

Memory can be utilized by operating system in two ways, from the perspective of today's lab-work, in placing processes in RAM, and in placing files in some permanent storage. I shall call both of these scenarios commonly as entities being placed in a storage device. We simulated contiguous allocation for entities in storage devices, and thus simulated this aspect of an operating system. This procedure is adopted both in case of placement of processes in RAM and of files in a permanent storage. The major tasks are as follows.
(1) Simulation of a storage device: allocating random bits to an array of size *n*. *1* stands for a file entity; *0* stands for presence of no data at that location. We were aware that this is not be the correct way how these resources are organized, but for ease in implementation of algorithms that we were going to implement, we chose this method.
(2) Identifying the starting and ending indices of all holes, saving that to a data structure.
(3) Given a file size of *m*, placing that file as per the FIRST_FIT procedure.
(4) Given a file size of *m*, placing that file as per the BEST_FIT procedure.
The program in C is given as Appendix A (16).

13. Distributed Systems; Paging and Virtual Machines

(November 5, 2018)

We discussed how a typical distributed system is constructed (both hardware and software overview) and (a) what are the challenges that must be taken care of while programming on such systems, (b) how are distributed systems utilized nowadays.
We discussed on how paging is proceeded in a typical operating system. We took the example of JVM for efficient paging implementation.
We then discussed how Java Virtual Machine (JVM) implements paging. Each program that we run on JVM is a process for JVM. JVM lies as a layer between those programs that are picked from the bytecode. No java program communicates directly with the parent operating system. For those processes, JVM is the operating system. They send all the requests to JVM and JVM then communicates with the underlying operating system for accomplishments of those requests. JVM divides each program into pages and extensively implements this concept. The Object Oriented Programming System (OOPS) makes it even easier to slice the program into justifiable pieces, pages.
For the Operating System, JVM is a process. So from the perspective of the underlying Operating System, each program running on JVM is a thread. This also leads to the fact that java programs cannot utilize the memory addresses (or any other hardware resource) through direct addressing.
That's why we call JVM a "virtual machine". All virtual machine follow similar procedures.

14. Round Robin algorithm; Last class' revision

(November 12, 2018)

We discussed the previous laboratory for those students who were not in the Institute because of the Diwali week.
We discussed the Round Robin scheduling algorithm. The students programmed for the algorithm. The program is given as Appendix A (17).

Appendices

Appendix A
Program Codes

(1) Getting Hardware Information

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

struct CpuInfo {
      char vendor_id[50];
      int family;
      char model[50];
      float freq;
      char cache[20];
};

int main(){
      printf(__VERSION__);
      printf("ATG");
      struct CpuInfo info = {"", 0, "", 0.0, ""};

      FILE *file = fopen("/proc/cpuinfo", "rb");
      if( file  == NULL ) {
            printf("ERRORE! Impossibile aprire il file relativo alla CPU.");
      }
      else {
                  fread(&info, sizeof(struct CpuInfo), 1, file);
                  printf("%s\n%d\n%s\n%.2f\n%s\n", info.vendor_id,
info.family, info.model, info.freq, info.cache);
            }
}
```

(2) Fork()

Credits: Ms Vaide Vaishnav

201751059

Program 1.

```
#include<stdio.h>
#include<unistd.h>
int main()
{
      printf("%d\n",getppid()); // process ID, which OS allocated
      fork();
      int x=getppid();
      if(x%2==0)
      {
            printf("%d,ABC\n",x);
      }
      else
      {
            printf("%d,MASS BUNK",x);
      }
return 0;
}
```

Program 2.

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

int main()
{
      pid_t P=getppid();
      printf("%d\n",P);
      fork();
      fork();
      if(P==getppid())
      {
            printf("Parent ID=%d\n",getppid());
      }
      else
      {
            printf("Child ID=%d\n",getppid());
      }
} //only one out of if and else should run, but both are running, after fork
the given process divides into 2, one will remain parent and the other is
child. both of the process(i.e. child and parent) both will have have the same
if-else code.
```

(3) Exec()

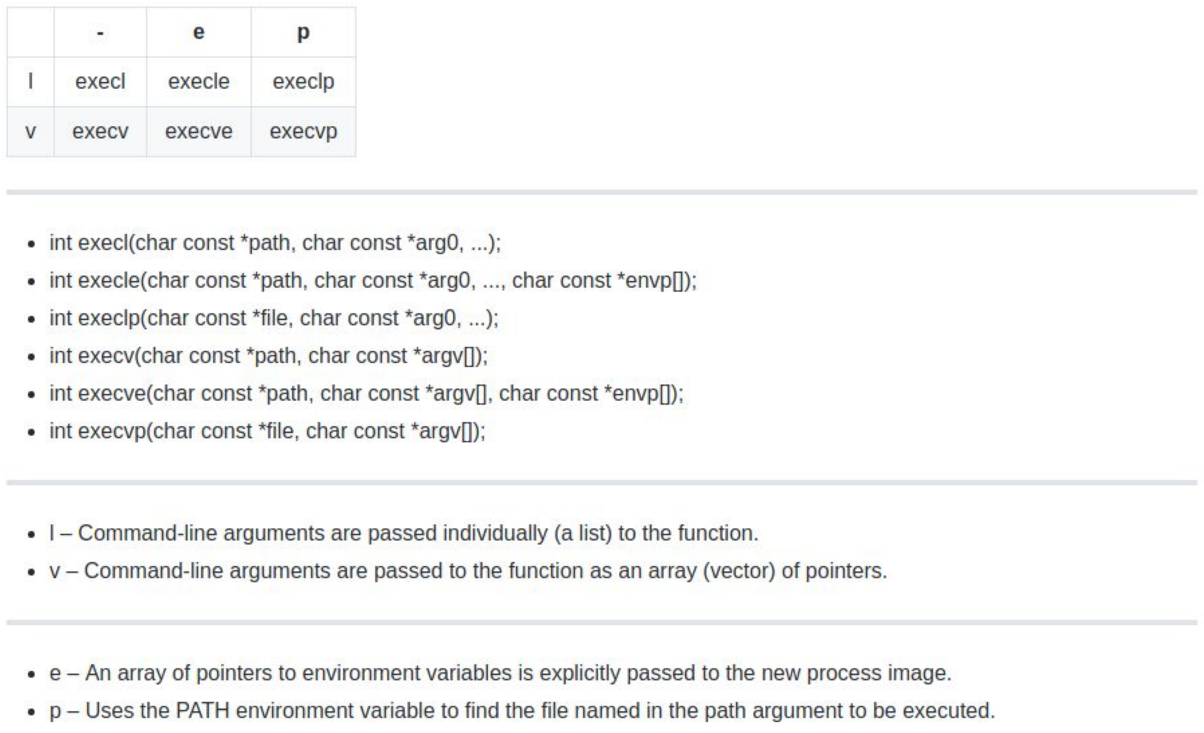
```
#include<stdio.h>

int main(){
      printf("MASSBUNK");
}

#include<stdio.h>
#include<unistd.h>

int main(){
      printf("FULLATTENDENCE\n\n");
      char*args[]={"./new",NULL};
      execvp(args[0],args);
      printf("FULLATTENDENCE");
}

//execl,execlp,execle,execv,execvp,execvpe
```



	-	e	p
l	execl	execle	execlp
v	execv	execve	execvp

- int execl(char const *path, char const *arg0, ...);
- int execle(char const *path, char const *arg0, ..., char const *envp[]);
- int execlp(char const *file, char const *arg0, ...);
- int execv(char const *path, char const *argv[]);
- int execve(char const *path, char const *argv[], char const *envp[]);
- int execvp(char const *file, char const *argv[]);

- l – Command-line arguments are passed individually (a list) to the function.
- v – Command-line arguments are passed to the function as an array (vector) of pointers.

- e – An array of pointers to environment variables is explicitly passed to the new process image.
- p – Uses the PATH environment variable to find the file named in the path argument to be executed.

(4) Shell Programming - create a C file using shell, write data to it, compile and exectute it

```
#author=Arya T Gupta
echo "welcome to my first batch program"
echo "i am going to execute a c program using a batch program"
file="prog.c"
echo going to check this file = $file
if [ -f $file ]
then
    rm $file
    echo file was already present and has been removed
fi
touch $file
echo file has been created.
echo '#include<stdio.h>' >> $file
echo 'written #include<stdio.h> to file'
echo 'int main(){' >> $file
echo 'written int main(){ to file'
echo '    printf("\nprogram has been run through shell\n");' >> $file
echo 'written     printf("\nprogram has been run through shell\n"); to file'
echo '}' >> $file
echo 'written } to file'
echo 'now compiling file'
gcc $file
echo 'now executing file'
file="a.out"
./$file
```

(5) Accessing stats of a file using C program, changing user permissions

```
#include <stdio.h>         // For printf()
#include <sys/stat.h> // For struct stat and stat()

int main(){
      struct stat stResult;

      if(stat("a.out", &stResult) == 0)
      {
            printf("Filesize: %i", stResult.st_size);
      }
      else
      {
            printf("Couldn't get file attributes...");
      }
}
```

The stat structure in the ***sys/stat.h*** library.

```
struct stat {
  dev_t     st_dev;     /* ID of device containing file */
  ino_t     st_ino;     /* inode number */
  mode_t    st_mode;    /* protection */
  nlink_t   st_nlink;   /* number of hard links */
  uid_t     st_uid;     /* user ID of owner */
  gid_t     st_gid;     /* group ID of owner */
  dev_t     st_rdev;    /* device ID (if special file) */
  off_t     st_size;    /* total size, in bytes */
  blksize_t st_blksize; /* blocksize for file system I/O */
  blkcnt_t  st_blocks;  /* number of blocks allocated */
  time_t    st_atime;   /* time of last access */
  time_t    st_mtime;   /* time of last modification */
  time_t    st_ctime;   /* time of last status change */
};
```

(6) Changing user permissions of a file

```
#include <stdio.h>         // For printf()
#include <sys/stat.h> // For struct stat and stat()

int main(){
      struct stat fn;

      stat("a.out", &fn);
      printf("Ownwer ID: %d", fn.st_uid);
      chown("a.out", 25, 0);
      stat("a.out", &fn);
      printf("\nOwner ID: %d\n", fn.st_uid);
}
```

(7) FCFS implementation in C

Credits: Ms Preeti Yadav
201751039

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

```
int main()
{
      int n=5;
      int at[n]={0,1,2,3,4};
      int bt[n]={2,3,4,5,6};
      int pgc[n],tgc[n],rt[n];

      int tq=0;
      for(int i=0;i<n;i++)
      {
            if(at[i]<=tq)
            {
                  pgc[i]=i;
                  tgc[i]=tq;
                  tq+=at[i];
            }
            else
            {
                  tq++;i--;
            }
      }
      printf("PGC: ");
      for(int i=0;i<n;i++)
      {
            printf("%d ",pgc[i]);
      }
      printf("\nTGC: ");
      int i,j;
      for(i=0;i<n;i++)
      {
            printf("%d ",tgc[i]);
      }


      for(j=0;j<n-1;j++)
```

```
	{
		rt[j]=tgc[j+1]-tgc[j];
	}
	rt[j]=bt[j]+tgc[j-1];
	tgc[j]=rt[j];
	printf("\nResponse Time: ");
	for(int i=0;i<n;i++)
	{
		printf("%d ",rt[i]);
	}
}
```

(8) SRT Implementation

```
#include<stdio.h>

struct process{
	//st=starting time ; et=ending time; pno = process no
	int pno;
	int st;
	int et;
	struct process *next;
};
struct process head,tail;

int getMax(int arrive[],int burst[],int t, int n){
	int maxIndex;
	int max=-1;
	for(int i=0;i<n;i++){
		if(max<burst[i] && burst[i]!=0){
			max=burst[i];
			maxIndex=i;
		}
	}
	if(max>-1)return maxIndex;
```

```
        return -1;
}

int getMin(int arrive[],int burst[],int t,int n){
        int minIndex;
        int min=-1;
        for(int i=0;i<n;i++){
                if(t>=arrive[i] && burst[i]!=0){
                        if(min==-1){
                                min=burst[i];
                                minIndex=i;
                        }
                        else if(min>burst[i]){
                                min=burst[i];
                                minIndex=i;
                        }
                }
        }
        if(min>-1)return minIndex;
        return -1;
}

int getnextTQ(int arrive[],int burst[],int t,int n,int minIndex){
        int min=burst[minIndex];
        for(int i=0;i<n;i++){
                if(burst[i]!=0 && minIndex!=i){
                        if(min>arrive[i]-t && arrive[i]>t)
                                min=arrive[i]-t;
                }
        }
        return min;
}

void add(int pno,int st,int et){
        //struct process ob={.pno=pno,.st=st,.et=et,.next=NULL};
        printf("pno %d st %d et %d\n",pno,st,et);
```

```
	/*
	 * MAKE A LINKED LIST USING THESE VALUES AND THE STRUCT
	 *
	 */



	/*
	if(head==NULL)
		head=tail=ob;
	else{
		tail->next=ob;
		tail=ob;
	}
	* */
}

int getNextArrTime(int t,int arrive[],int burst[],int n){
	for(int i=0;i<n;i++){
		if(burst[i]!=0){
			return arrive[i];
		}
	}
	return t;
}

int main(){
	int n=4;
	int arrive[]={0,3,9,10};
	int burst[]={4,1,2,17};
	int t=0;


				printf("arrive array\n");
				for(int i=0;i<n;i++)printf("%d ",arrive[i]);
				printf("\n");
				printf("burst array\n");
```

```
                        for(int i=0;i<n;i++)printf("%d ",burst[i]);
                        printf("\n");
                        printf("time=%d\n",t);

            while((getMax(arrive,burst,t,n))>=0){
                        printf("\n\n");

                int minIndex=getMin(arrive,burst,t,n);

                        printf("minIndex %d\n",minIndex);

                if(minIndex==-1){//no process in ready queue
                    t=getNextArrTime(t,arrive,burst,n);
                    continue;
                }
                int nextTQ=getnextTQ(arrive,burst,t,n,minIndex);

                        printf("nextTQ %d\n",nextTQ);

                add(minIndex,t,t+nextTQ);
                t=t+nextTQ;
                arrive[minIndex]=t;
                burst[minIndex]-=nextTQ;
                        printf("arrive array\n");
                        for(int i=0;i<n;i++)printf("%d ",arrive[i]);
                        printf("\n");
                        printf("burst array\n");
                        for(int i=0;i<n;i++)printf("%d ",burst[i]);
                        printf("\n");
                        printf("time=%d\n",t);
            }
        }
```

(9) Pthreads

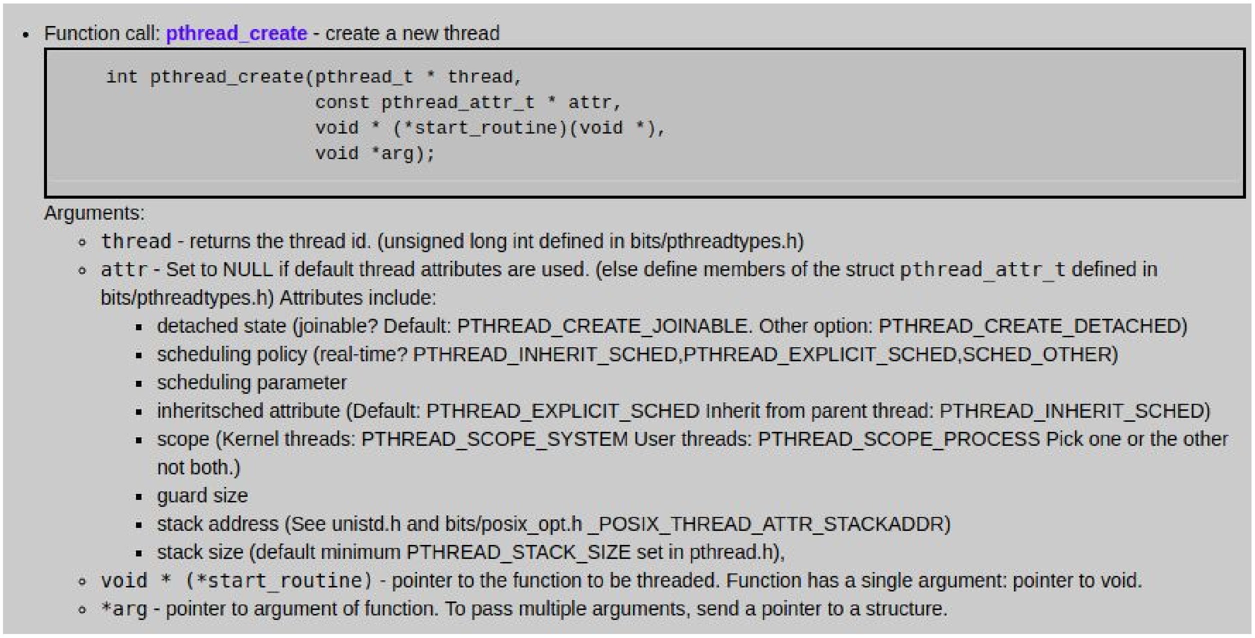
```
gcc untitled.c -lpthread

#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{
        //sleep(1);
        for(int i=0;i<10;i++)
        printf("Printing GeeksQuiz from Thread \n");
        return NULL;
}

int main()
{
        pthread_t thread_id;
        printf("Before Thread\n");
        pthread_create(&thread_id, NULL, myThreadFun, NULL);
        //pthread_join(thread_id, NULL);
        for(int i=0;i<10;i++)
                printf("After Thread\n");
        pthread_exit(0);
        //sleep(10);
}
```

Attributes of pthread_create().

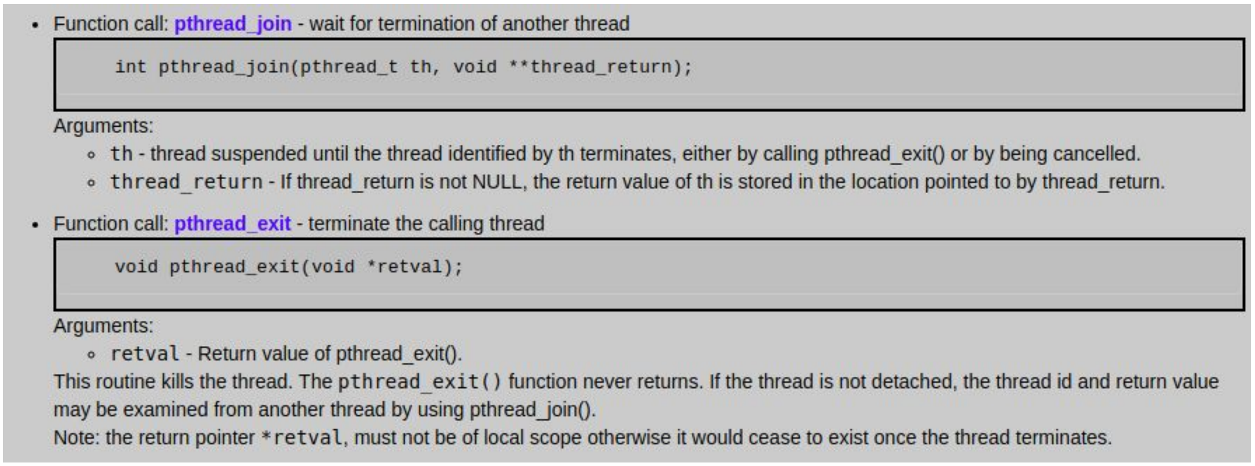


- Function call: pthread_create - create a new thread

```
    int pthread_create(pthread_t * thread,
                       const pthread_attr_t * attr,
                       void * (*start_routine)(void *),
                       void *arg);
```

Arguments:
- `thread` - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)
- `attr` - Set to NULL if default thread attributes are used. (else define members of the struct `pthread_attr_t` defined in bits/pthreadtypes.h) Attributes include:
  - detached state (joinable? Default: PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED)
  - scheduling policy (real-time? PTHREAD_INHERIT_SCHED,PTHREAD_EXPLICIT_SCHED,SCHED_OTHER)
  - scheduling parameter
  - inheritsched attribute (Default: PTHREAD_EXPLICIT_SCHED Inherit from parent thread: PTHREAD_INHERIT_SCHED)
  - scope (Kernel threads: PTHREAD_SCOPE_SYSTEM User threads: PTHREAD_SCOPE_PROCESS Pick one or the other not both.)
  - guard size
  - stack address (See unistd.h and bits/posix_opt.h _POSIX_THREAD_ATTR_STACKADDR)
  - stack size (default minimum PTHREAD_STACK_SIZE set in pthread.h),
- `void * (*start_routine)` - pointer to the function to be threaded. Function has a single argument: pointer to void.
- `*arg` - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

Other methods in pthread.



- Function call: pthread_join - wait for termination of another thread

```
    int pthread_join(pthread_t th, void **thread_return);
```

Arguments:
- `th` - thread suspended until the thread identified by th terminates, either by calling pthread_exit() or by being cancelled.
- `thread_return` - If thread_return is not NULL, the return value of th is stored in the location pointed to by thread_return.

- Function call: pthread_exit - terminate the calling thread

```
    void pthread_exit(void *retval);
```

Arguments:
- `retval` - Return value of pthread_exit().

This routine kills the thread. The `pthread_exit()` function never returns. If the thread is not detached, the thread id and return value may be examined from another thread by using pthread_join().
Note: the return pointer `*retval`, must not be of local scope otherwise it would cease to exist once the thread terminates.

(10) Shell Programming

(1)

```
limit=10
#for a in {0,1,2,3,4,5}
#for a in {0..10}
#for a in {0..10..2}
#for a in 1 2 3 4 5
for ((a=0;a<$limit;a++))
```

```
do
   echo "welcome $a"
done
```

(2)

```
for((;;))
do
      echo "Infinite loop, hit Ctrl+c to stop"
done
```

(3)

```
read -p "Enter a number to check:" n
echo "Entered number is $n"
prime=1
for((i=2;i<$((n/2));i++))
do
      r=$(($n % $i))
      if (( $r == 0 ))
      then
            prime=0
            break
      fi
done
if (( $prime == 0 ))
then
      echo "$n is NOT a prime number"
else
      echo "$n is a prime number!"
fi
```

(4)

```
search_dir=/home/arya
for entry in "$search_dir"/* ; do
      echo $(basename "$entry")
      if [ $(basename "$entry") == "untitled.c" ]
      then
            echo
            echo
            echo
            echo Output starts hence
            gcc $(basename "$entry") -lpthread -o atg
            ./atg
            break
      fi
done
```

(5)

```
func(){
      echo "parameters were $@"
      echo "First value $1"
      echo "Second value $2"
      r=$(($1 ** 4))
      echo "first value treated $r"
      return $(($r+1))
}
func 2 3
z=$?
echo "Next Value $z"
```

(11) Modify last access and modification time of a file

```
#include <stdio.h>         // For printf()
#include <time.h>       //tm struct
#include <utime.h>      //utime struct
#include <sys/stat.h> // For struct stat and stat()

int main(){
      struct stat stResult;
```

```
    if(stat("untitled.c", &stResult) == 0)
    {
        printf("Last time access: %d\n",stResult.st_mtime);
        //printf("Microseconds%i", stResult.st_atime);
        ;
        struct tm t;
        t.tm_year=1996-1900;
        t.tm_mon=3;//april
        t.tm_mday=9;
        t.tm_hour=18;
        t.tm_min=0;
        t.tm_sec=0;
        t.tm_isdst = -1;        // Is DST on? 1 = yes, 0 = no, -1 =
unknown
        printf("This is going to be updated%d\n",mktime(&t));


        struct utimbuf new_time;
        new_time.actime=stResult.st_atime; //actime for access time
        new_time.modtime=mktime(&t);  //modtime for modification time

        utime("untitled.c",&new_time);
        printf("Now changed.\nNowPrinting.\n");

        stat("untitled.c", &stResult);
            printf("Last time access: %d\n",stResult.st_mtime);
    }
    else
    {
        printf("Couldn't get file attributes...");
    }
}
```

(12) Peterson’s algorithm C implementation

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

int turn=0;
int i=0,j=1;
bool flag[2]={0};

void critical_section(int p)
{
      printf("Process %d critical section.\n",p);
      printf("Process %d critical section END.\n",p);
}

void remaining_section(int p){
      printf("Process %d remaining section.\n",p);
}

void *process0(void *args){
      do{
            turn=j;
            flag[i]=1;

            while(turn==j && flag[j]==1)
            {printf("Process 0 waiting\n");}

            critical_section(i);

            flag[i]=false;

            remaining_section(i);
      }while(true);
}


void *process1(void *args){
	do{
		turn=i;
		flag[j]=1;

		while(turn==i && flag[i]==1)
		{printf("Process 1 waiting\n");}

		critical_section(j);

		flag[j]=false;

		remaining_section(j);
	}while(true);
}

int main()
{
	pthread_t id1, id2;
	pthread_create(&id1,NULL,process0,NULL);
	pthread_create(&id2,NULL,process1,NULL);
	pthread_exit(0);
}
```

(13) Banker's Algorithm

Credits: Ms Akshita Agarwalla
201751066

```
#include<stdio.h>
int alloc[3];

int Allocate(int t[], int ps[][4], int pr[][4], int left[]){
	int i,j;
	for(i=0; i<3; i++){
		int pri=1;
		for(j=0; j<4; j++){
			if(pr[i][j]>left[j]){
```

```
                pri=0;
            }
        }
        int sum=0;
        for(j=0; j<4; j++)
            sum+=pr[i][j];
        if(pri && sum){
            for(j=0; j<4; j++){
                left[j]-=pr[i][j];
                pr[i][j]=0;
            }
            return i;
        }
    }
    return -1;

}

void Deallocate(int i, int t[], int ps[][4], int pr[][4], int left[]){
    int j;
    for(j=0; j<4; j++){
        left[j] += ps[i][j];
        ps[i][j]=0;
    }
}

void bankers(int t[], int ps[][4], int pr[][4]){
    int i,j,left[4];
    for(j=0; j<4; j++){
        left[j]=t[j];
        for(i=0; i<3; i++){
            left[j]-=ps[i][j];
        }
    }
    while(1){
        int allocated = 0;
```

```
            int a;
            do{
                  a = Allocate(t,ps,pr,left);
                  if(a>=0){
                        allocated = 1;
                        alloc[a]=1;
                  }
            }while(a >= 0);
            for(i=0; i<3; i++){
                  if(alloc[i]){
                        alloc[i]=0;
                        Deallocate(i,t,ps,pr,left);
                  }
            }
            if(allocated==0){
                  printf("DEADLOCK\n");
                  return;
            }
            int psi=1;
            for(i=0; i<3; i++){
                  for(j=0; j<4; j++){
                        if(ps[i][j] != 0){
                              psi=0;
                        }
                  }
            }
            if(psi){
                  printf("NO DEADLOCK\n");
                  return;
            }
      }
}

int main(){
      int t[4] = {10,10,10,10};
      int ps[3][4] = {{2,3,2,3},{1,2,1,2},{5,5,5,4}};
```

```
	int pr[3][4] = {{3,1,3,4},{0,0,0,1},{2,2,2,3}};
	for(int i=0; i<3; i++)
		alloc[i]=0;
	bankers(t,ps,pr);
	return 0;
}
```

(14) Dining Philosopher’s problem using semaphore

Credits: Mr Shubham Bindal
201751054

```
#include<stdio.h>
#define n 5

int  lock[n] ={0};
int semaphore = n;
int P[n]={0};

void signal(semaphore)
{
	semaphore+=1;
}

void wait(semaphore)
{
	while(semaphore<=0)
	{}
	semaphore-=1;
}

int main()
{
	int i;
	while(1)
	{
		int free=0;
```

```
for(i=0;i<n;i++)
{
      if(P[i]==1){
            free+=1;
      }
}
if(free==n)
      {break;}

for (i=0;i<n;i++)
{
      if(P[i]==0)
      {
            if(!lock[i] && !lock[(i-1+n)%n])
            {
                  wait(semaphore);
                  wait(semaphore);
                  P[i]=1;
                  lock[i]=1;
                  lock[(i-1+n)%n]=1;
                  printf("process P[%d] is starting\n",i);
            }
            else
            {
                  printf("process P[%d] is waiting\n",i);
            }
      }
}

for (i=0;i<n;i++)
{
      if(P[i]==1)
      {
            if(lock[i] && lock[(i-1+n)%n])
            {
                  signal(semaphore);
```

```
                              signal(semaphore);
                              lock[i]=0;
                              lock[(i-1+n)%n]=0;
                              printf("process P[%d] has released the
resources\n",i);
                        }
                  }
            }
      }
}
```

(15) Inter-process communication using shared memory

Credits: Ms Kushboo Garg
201751065

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

# define s 6
int buff[s];
int data=0;
int si=0,ei=0;


void *producer(void*args){
    int i,j;
    char command[26];
    char a='a';
    for (i=0;i<26;i++){
       command[i]=a;
       a++;
    }

    i=0;
    while(i<26){
       if(data<s){
             buff[ei]=command[i];
             printf("p buff[%d]= %c i=%d command[i]=%c
data=%d\n",ei,buff[ei],i,command[i],data);
             i++;
             ei=(ei+1)%s;
             data=data+1;
       }
       else
             printf("p waiting\n");
       sleep(1);
    }
}

void *consumer(void*args){
    while(1){
       printf("c data=%d",data);
       if(data>0){
             printf("c buff[si]=%c\n",buff[si]);
             data=data-1;
             if(buff[si]=='z'){
                   printf("terminated");
                   break;
                   }
             si=(si+1)%s;
       }
       sleep(2);
    }
}
int main(){
    pid_t P=getpid();
    pthread_t tid1,tid2;
    pthread_create(&tid1,NULL,producer,NULL);
    pthread_create(&tid2,NULL,consumer,NULL);
    pthread_exit(0);
    return 0;
}
```

(16) Contiguous memory allocation, first fit, best fit

Credits: Mr Ajit Kumar Sancheti
201751067

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

void BEST_FIT(int si[],int ei[],int ei_index,int size)
{
      int empty_space,t=0;
      int start_index,end_index,i;
      empty_space=1000;
      for(i=1;i<ei_index;i++)
      {
            if(ei[i]-si[i]>=size-1)
            {
                  t=1;
                  //    start_index=si[i];
                        //end_index=ei[i];


                  if(empty_space>ei[i]-si[i]+1-size)
                  {
                        empty_space=ei[i]-si[i]+1-size;
                        start_index=si[i];
                        end_index=ei[i];
                  }
            }
      }
      if(t==1)
            printf("\nFOR BEST FIT SIZE OF HOLE IS %d , STARTING INDEX %d ,
ENDING INDEX %d\n",end_index-start_index+1,start_index,end_index);
```

```
	else
		printf("Not fit to anywhere!!  Best case not possible !!");
}

void CHECK_FILE_SIZE(int si[],int ei[],int ei_index)
{
	int size,i;
	printf("\n ENTER THE SIZE OF THE FILE YOU WANT TO FIT\n");
	scanf("%d",&size);
	int fitt = 0;
	for(i=0;i<ei_index;i++)
	{
		if(ei[i]-si[i]>=size-1)
			{
				printf("THE FILE OF GIVEN SIZE CAN BE FITTED BETWEEN
INDEXES %d AND %d\n",si[i],ei[i]);
				fitt=1;
				break;
			}
	}
	if(fitt==0)
	{
		printf("\nTHE FILE OF GIVEN SIZE CANNOT BE FITTED\n");
	}

	BEST_FIT(si,ei,ei_index,size);
}

void HOLES(int arr[],int n,int count_0,int count)
{

	int ei_index=0,i;
	int si_index=0;
	int ei[count_0];
	int si[count_0];
	count=0;
	while(count!=n)
	{
		if(arr[count]==0)
			{
				si[si_index]=count;
				si_index++;
				while(arr[count]==0)
					{
						count++;
					}
				ei[ei_index]=count-1;
				ei_index++;

			}
		else
		{
			count++;
		}
	}

	printf("\nSTARTING INDEXES :\n");
	for(i=0;i<si_index;i++)
	{
		printf(" %d \n",si[i]);
	}

	printf("\nENDING INDEXES :\n");
	for(i=0;i<ei_index;i++)
	{
		printf(" %d \n",ei[i]);
	}

	CHECK_FILE_SIZE(si,ei,ei_index);

}
```

```
int main()
{

	int n,i;
	srand(time(NULL));
	printf("\nENTER THE NUMBER OF BITS \n");
	scanf("%d",&n);
	int arr[n],count=0;
	int count_0=0;
	for(i=0;i<n;i++)
	{
		arr[i]=rand()%2;
	}
	printf("\nRANDOMLY GENERATED BITS ARE :\n");
	for(i=0;i<n;i++)
	{
		printf(" %d ",arr[i]);
	}

	while(count!=n)
	{
		if(arr[count]==0)
			{
				count_0++;
				while(arr[count]==0)
					{
						count++;
					}
			}
		else
		{
			count++;
		}
	}
```

```
      HOLES(arr,n,count_0,count);
      return 0;

}
```

(17) Round Robin algorithm

Credits: Mr Kartikey Lakhotia
201751020

```
#include<stdio.h>

int main()
{

  int count,j,n,time,remain,flag=0,time_quantum;
  int wait_time=0,turnaround_time=0;
  remain=n;
  time_quantum=2;
  n=4;
  int at[]={0,1,2,3};
  int bt[]={2,3,5,6};
  int rt[]={2,3,5,6};
  printf("\n\nProcess\t|Turnaround Time|Waiting Time\n\n");
  for(time=0,count=0;remain!=0;)
  {
    if(rt[count]<=time_quantum && rt[count]>0)
    {
      time+=rt[count];
      rt[count]=0;
      flag=1;
    }
    else if(rt[count]>0)
    {
      rt[count]-=time_quantum;
      time+=time_quantum;
    }
    if(rt[count]==0 && flag==1)
    {
      remain--;

printf("P[%d]\t|\t%d\t|\t%d\n",count+1,time-at[count],time-at[count]-bt[count]
);
      wait_time+=time-at[count]-bt[count];
      turnaround_time+=time-at[count];
      flag=0;
    }
    if(count==n-1)
      count=0;
    else if(at[count+1]<=time)
      count++;
    else
      count=0;
  }
  printf("\nAverage Waiting Time= %f\n",wait_time*1.0/n);
  printf("Avg Turnaround Time = %f",turnaround_time*1.0/n);

  return 0;
}
```

Appendix B

Practical Examination

(November 19, 2018)

I. Practical (2 hours)

(60 marks)

Each student has to select one question from the pool of 4 questions on the basis of this formula (((roll number - 1) % 5) + 1).

1. Simulation of inter-process communication using shared memory - the producer-consumer simulation.
2. Simulation of solution to dining philosophers' problem using semaphore.
3. Simulation of Banker's algorithm for allocation of resources to 5 processes and 4 resources.
4. Simulation of Shortest remaining Time First (SRTF) Scheduling Algorithm for 5 processes.
5. Simulation of Round Robin (RR) Scheduling Algorithm for 5 processes.

The programs will take input from files. The programs must be developed in such a way that the output should come even if the input values are changed.

II. Viva

(40 marks)

Viva included questions related to the syllabus theory (1) and experiments conducted in the Operating Systems laboratory (3). Each correct answer will give them credit of 10 marks.

The MM for the practical was 100. (60 for practical experiment and 40 for viva), which was sent to Prof. Majumder after the practical examination.