# LAB 7

**CPU Scheduling Algorithms**

## Introduction

Scheduling is the task of operating system in which processes are given access to system resources. Scheduling is a fundamental operating system function that determines which process run, when there are multiple processes in ready queue. CPU scheduling is important because it impacts resource utilization and other performance parameters.

## Scheduling Criteria

Different CPU scheduling algorithms have different properties and may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the different properties of the various algorithms.

***CPU Utilization*:** We want to keep the CPU as busy as possible. CPU utilization may range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

***Throughput:*** If the CPU is busy executing process, then work is being done. One calculate of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, throughput might be 10 processes per second.

***Turnaround time:***From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

***Waiting Time:***The CPU scheduling algorithm does not affect the amount of time during which a procedure executes or does I/O; it affects only the amount of time that a process spends waiting in the prepared queue. Waiting time is the sum of the periods spent waiting in the ready queue.

*Response time:* In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly easy, and can continue computing new results while earlier results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time that it takes to start responding, but not the time that it takes to output that response. The turnaround time is generally incomplete by the speed of output device.

## Scheduling Algorithms

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are implementing in this lab:

1. First-Come, First-Served (FCFS) Scheduling
2. Shortest-Job-First (SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling

### First Come First Serve (FCFS) Algorithm:

1. Start the process.
2. Declare the array size.
3. Get the number of elements to be inserted.
4. Select the process that first arrived in the ready queue
5. Make the average waiting the length of next process.
6. Start with the first process from its selection as above and let other process to be in queue.
7. Calculate the total number of burst time.
8. Display the values.
9. Stop the process.

### Shortest Job First (SJF) Algorithm:

1. Start the process.
2. Declare the array size.
3. Get the number of elements to be inserted.
4. Select the process which have shortest burst will execute first.
5. If two process have same burst length, then FCFS scheduling algorithm used.
6. Make the average waiting the length of next process.
7. Start with the first process from its selection as above and let other process to be in queue.
8. Calculate the total number of burst time.
9. Display the values.
10. Stop the process.

## Lab task

1. Perform the implementation and show the output of FCFS and SJF(Non-Preemptive), algorithms.

## Home Assignment

1. Write the algorithm of SJF (Preemptive) algorithm and also implement it and show the output.

# LAB # 8

**CPU Scheduling Algorithms**

## Scheduling Algorithms

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are implementing in this lab:

1. First-Come, First-Served (FCFS) Scheduling
2. Shortest-Job-First (SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling

### Priority Scheduling Algorithm:

1. Start the process.
2. Declare the array size.
3. Get the number of elements to be inserted.
4. Get the priority for each process and value
5. start with the higher priority process from it’s initial position let other process to be queue.
6. calculate the total number of burst time.
7. Display the values
8. Stop the process.

### Round Robin (RR) Algorithm:

1. Start the process.
2. Declare the array size.
3. Get the number of elements to be inserted.
4. Get the value.
5. Set the time sharing system with preemption.
6. Define quantum is defined from 10 to 100ms.
7. Declare the queue as a circular.
8. Make the CPU scheduler goes around the ready queue allocating CPU to each process for the time interval specified.
9. Make the CPU scheduler picks the first process and sets time to interrupt after quantum expired dispatches the process.
10. If the process has burst less than the time quantum than the process releases the CPU.

## Lab task

1. Perform the implementation and show the output Priority(Non-Preemptive) and Round Robin algorithms.

## Home Assignment

1. Write the algorithm of Priority (Preemptive) algorithm and also implement it and show the output.

9

NAMED PIPES

# OBJECTIVE

Study the features of Named Pipes.

# THEORY

While pipes are powerful inter-process communication they are not without drawbacks.

Firstly, pipes can only be used on processes that have a common ancestry like a parent and child process. And, secondly, they are not permanent. A process creates them and the termination of the process leads to their destruction.

To overcome these deficiencies a variation on these pipes has been implemented known as NAMED PIPES. These named pipes basically function in the same way as pipes: acting as one way communication channels. But unlike pipes, a named pipe is a permanent fixture. UNIX treats it just likes a file, giving it a size, owner and access permission. It can be opened, closed or deleted like any other file.

Now let’s see how we can use this named pipe.

Do a cat < testpipe &. This results in the cat program being made into a back ground process. Why we are making it a background process? Well, had we not, the cat would have kept waiting to read something. But since the pipe is empty, the program would hang. By making it a background process we ensure that the program does not hang (try it).

But this process is unable to read from the named pipe testpipe. Why? Because in the first place this pipe has not even been opened. A named pipe can only be opened if there is someone waiting at both its ends. In this case cat tries to read from it, but there is no corresponding process trying to write to it. As a result pipe is not opened at all.

So what we need here is some way of writing to the pipe. And that is what a ls –l>testpipe at the prompt, will do. The moment we give this command at the prompt, the pipe is opened in the background process for a read and in this (the ls) process for a write. The output of the ls –l is now outputted to the pipe instead of to the screen. But since, at the read end of the pipe, a cat is being performed we see the output on the screen.

UNIX, we know, is based on a client-server relationship. On the server machine we have the OS and all other files. This is the main machine on which all processing is done. All other machines connected to this main machine are known as nodes or clients. They are merely a screen and keyboard connection. All commands or programs specified at these nodes are actually run on the server.

Probably one of the most common uses of named pipes is for communicating between a client and server. We keep a process that has opened a named pipe running in the background. This process waits for some other process to write to the named pipe. Once something is written, the message is printed on screen.

Basically, there will be two programs that run hand in hand. One that waits for something to be written to a name pipe and one that can be used to write something to the named pipe. Of course, the write program has to be aware of named pipe it can write to. And the program that keeps the named pipe ready and waiting to receive information has to be active. The read program will have to be run on the server while the write program can be run by any client. This is what client-server communication is all about. For reference consider Program 1.

In Program 1, we opened the same named pipe we had created at the command line. The file is being opened in the read write mode. In the never ending for loop, a read( ) is activated. The moment a message is read from the pipe it is printed by the printf( ) function.

(Run the Program 1 and verify it what it is doing)

Now we need a program that will write messages to the pipe. For reference consider Program 2.

Program 2, when it is run, has to be passed a message or messages. It is this message that will be read and printed by the receiving program.

(Run the Program 2 and verify it what it is doing)

# Example programs

## Program 1

#include<fcntl.h>

#include<stdio.h>

#define MSGSIZ 63

main( )

{

int fd;

char msgbuf[MSGSIZ+1];

if((fd=open(“testfile”,O\_RDWR))<0)

perror(“pipe open failed”);

for(;;)

{

if(read(fd,msgbuf,MSGSIZ+1)>0)

printf(“message received:%s \n”,msgbuf);

}

}

## Program 2

#include<fcntl.h>

#include<stdio.h>

#include<errno.h>

#define MSGSIZ 63

main(argc,argv)

int argc;

char \*argv[ ];

{

int fd,j,nwrite;

char msgbuf[MSGSIZ];

if(argc<2)

{

printf(“Usage:<filename><message><message>…\n”);

}

if((fd=open(“testfile”,O\_WRONLY))<0)

perror(“fifo open failed”);

for(j=1;j<argc;j++)

{

strcpy(msgbuf,argv[j]);

if((nwrite=write(fd,msgbuf,MSGSIZ+1))<=0)

perror(“message write failed”);

}

}

## Program 3

#include<fcntl.h>

#include<stdio.h>

#define MSGSIZ 63

main( )

{

int fd;

char msgbuf[MSGSIZ+1];

if(mknod(“myfifo”,010666,0)<0)

perror(“myfifo failed”);

if((fd=open(“myfifo”,O\_RDWR))<0)

perror(“fifo open failed”);

for(;;)

{

if(read(fd,msgbuf,MSGSIZ+1)>0)

printf(“message received:%s \n”,msgbuf);

}

}

# Exercises

1. What did you learn after running the above Program 1 and Program 2?
2. What did you learn after running the Program 3?

LAB # 10

INTERPROCESS COMMUNICATION

# OBJECTIVE

To develop an application using Inter Process communication with pipes.

# THEORY

**pipe**() creates a pipe, a unidirectional data channel that can be used for inter-process communication.  The array *pipefd* is used to return two file descriptors referring to the ends of the pipe.  *pipefd[0]* refers to the read end of the pipe.  *pipefd[1]* refers to the write end of the pipe.  Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. On success, zero is returned.  On error, -1 is returned, and [*errno*](http://man7.org/linux/man-pages/man3/errno.3.html) is set

**Program 1**

#include<stdio.h>

#include<unistd.h>

#include<sys/ipc.h>

#include<sys/uio.h>

#include<sys/types.h>

#include<fcntl.h>

main()

{

int pid,pfd[2],n,a,b,c;

if(pipe(pfd)==-1)

{

printf("\nError in pipe connection\n");

exit(1);

}

pid=fork();

if(pid>0)

{

printf("\nParent Process");

printf("\n\n\tFibonacci Series");

printf("\nEnter the limit for the series:");

scanf("%d",&n);

close(pfd[0]);

write(pfd[1],&n,sizeof(n));

close(pfd[1]);

exit(0);

}

else

{

close(pfd[1]);

read(pfd[0],&n,sizeof(n));

printf("\nChild Process");

a=0;

b=1;

close(pfd[0]);

printf("\nFibonacci Series is:");

printf("\n\n%d\n%d",a,b);

while(n>2)

{

c=a+b;

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

a=b;

b=c;

n--;

}

}

}

# Exercise

1. What did you learn after running the Program 1?

LAB # 11

SEMaPHORES

# OBJECTIVE

Study the features of Semaphores.

# THEORY

Assume that two processes were accessing the same file. The one that got the time slice first would initialize a variable before working on the file. When the other process is given the time slice it would first check the value of this variable realize it has been initialized by some other process and refrain from accessing the file. When the first process finished it would reinitialize this variable. As a result the second process which has been waiting for just this to happen will get access to the file.

This variable in UNIX terms is known as semaphore and is basically an integer which acts as a counter. Its value depends on the number of resources there are to share. For example if we had one file which is to be shared then the semaphore can have a value 0 or 1. A 0 initialized semaphore signifies that the resource in this case the file is in use and therefore all other processes would have to wait. The moment the process that has access to the file finishes it sets this semaphore value to 1. Thereby allowing one other process access.

**Creating a Semaphore**

Now that we have a basic idea of semaphores and why they are used lets get to the hows, i.e. how to create and implement them.

To start with do a ipcs –s at the prompt. This will tell us if there are any semaphores existing. Details about any semaphore will be under the following column headings – key, semid, owner, perms, nsms and status. For reference consider Program 1 and Program 2.

Run the Program 1 that creates a semaphore:

semget( ) returns an value which is the identifier of the semaphore. If this value is a –1 it means an error.

And that’s what this program returned. So where’s the problem? Why because we have not specified how many semaphores we want.

UNIX does not just create one semaphore, what it allows us instead is the ability to create a set of semaphores. And in this set there should be a minimum of at least 1 sub-semaphore. The second parameter we pass the semget( ) function defines how many sub-semaphores there will be in this set of semaphores. Unfortunately we have assigned a 0 to this variable nsem which is below the minimum required. As a result an error is returned.

Do a ipcs –s to verify that this semaphore has not been created.

**Who is using the Resource**

A semaphore is basically used by different processes to synchronize access to a resource. Of course what value the semaphore should have to determine access or not has to be specified by us. For example may be we have designed our system in such a way that when a semaphore is deciphered as having a value of 1 it means that a certain resource is in use by another process. And a value of 2 means that a resource is free and that our process can now access to it.

Using the semctl( ) function we can find out which process has set the value of semaphore. Avalue of GETPID passed to the semctl( ) will result in it passing the PID of the process that has set the value of the semaphore. For reference consider Program 3 and Program 4.

Note: kill the semaphore first and then run Program 3 twice.

# Example programs

## Program 1

#include<sys/types.h>

#include<sys/ipc.h>

main( )

{

int semid,key,nsem;

key=(key\_t) 0\*20;

nsem=0;

semid=semget(key,nsem,IPC\_CREAT | 0666);

printf(“Created semaphore with ID:%d \n”,semid);

}

## Program 2

#include<sys/types.h>

#include<sys/ipc.h>

main ( )

{

int semid,key,flag,nsem;

key=(key\_t) 0\*20;

flag=IPC\_CREAT | 0666;

nsem=1;

semid=semget(key,nsem,flag);

printf(“Created Semaphore with ID:%d \n”,semid);

}

## Program 3

#include <sys/types.h>

#include<sys/ipc.h>

#include<sys/sem.h>

#include<errno.h>

main( )

{

int semid,retval;

semid=semget(0\*20,1,0666 | IPC\_CREAT);

retval = semctl(semid,0,GETPID,0);

printf(“PID returned by semctl is %d and actual PID is %d\n”, retval,getpid( ));

}

## Program 4

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/sem.h>

#include<errno.h>

main( )

{

int semid,retval;

semid=semget(0\*20,1,0666| IPC\_CREAT);

retval=semctl(semid,0,GETPID,0);

printf(“PID returned by semctl is %d and actual PID is %d\n”,retval,getpid( ));

retval=semctl(semid,0,SETVAL,1);

printf(“PID returned by semctl is %d and set value is %d\n”,retval,SETVAL);

}

# Exercises

1. What was the mistake in the Program 1, and how it is rectified in Program 2?
2. Do ipcs –s at the prompt to see a listing of the semaphores and an entry for the semaphore will be displayed.
3. Run the Programs 3 and 4, and find out what is the problem in 3 and how it is rectified in 4.

LAB # 11

ON FILES

# OBJECTIVE

Study the features of ON Files.

# THEORY

To start with, let’s see how running two processes which have access to the same file could result in a mess up of data. Run Program 1 and Program 2 one after the other, both times in the background. To run both the following programs do this at the prompt:

$<first>&

$<second>&

Once both these processes are in memory the time slice will be alternated between them. As a result, since both processes have read write access to the same file, the data they will write be all jumbled up.

(Run the Program 2 and verify the result)

# Example programs

## Program 1

#include<fcntl.h>

#include<unistd.h>

main( )

{

int fd, i;

fd = open(“locktest”,O\_APPEND | O\_CREAT| O\_RDWR,0777);

for(i=0;i<=2000;i++)

write(fd, “A”,1);

}

## Program 2

#include<fcntl.h>

#include<unistd.h>

main( )

{

int fd,i;

fd = open(“locktest”,O\_APPEND | O\_CREAT | O\_RDWR,0777);

for(i=0 ;i<=2000;i++)

write(fd, “B”,1);

}

## Program 3

#include<fcntl.h>

#include<unistd.h>

main( )

{

int fd,i;

fd = open(“locktest”,O\_APPEND | O\_CREAT | O\_RDWR,0777);

lockf(fd,F\_LOCK,0);

for(i=0;i<=2000++)

write(fd “A”,1);

}

## Program 4

#include<fcntl.h>

#include<unistd.h>

main( )

{

int fd,i;

fd = open(“locktest”,O\_APPEND | O\_CREAT | O\_RDWR,0777);

lockf(fd,F\_LOCK,0);

for(i=0;i<=2000++)

write(fd “B”,1);

}

# Exercises

1. Write few lines about what did you learn after running Programs 3 and Program 4.