**Operating Systems Laboratory**

**B.Tech. 5thSemester**



**Name : Subhendu Maji**

**Roll Number : 18ETCS002121**

**Department : Computer Science and Engineering**

**Faculty of Engineering & Technology**

**Ramaiah University of Applied Sciences**

**Ramaiah University of Applied Sciences**

Private University Established in Karnataka State by Act No. 15 of 2013

|  |  |
| --- | --- |
| Faculty | Engineering & Technology |
| Programme | B. Tech. in Computer Science and Engineering |
| Year/Semester | 2nd Year / 5th Semester |
| Name of the Laboratory | Operating Systems Laboratory |
| Laboratory Code | CSC306A |

List of Experiments

1. Introduction to operating system- compiling and booting of Linux kernel
2. Programs using process management system calls
3. Programs using file management system calls
4. Programs based on multithreaded programming
5. Programs for process scheduling algorithms
6. Solution to producer consumer problem using Mutex and Semaphore
7. Solutions to Dining philosopher problem using Semaphore
8. Programs for deadlock avoidance algorithm
9. Programs for memory management algorithms

**Index Sheet**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **No**  **.** | **Lab Experiment** | **Performing the**  **experiment**  **(7)** | **Document**  **(7)** | **Viva**  **(6)** | **Total**  **Marks**  **(20)** |
|  |  |  |  |  |  |
| 1 | Programs using process  management system calls |  |  |  |  |
| 2 | Programs using file management system calls |  |  |  |  |
| 3 | Programs based on multithreaded programming |  |  |  |  |
| 4 | Programs for process scheduling algorithms |  |  |  |  |
| 5 | Solution to Producer Consumer  Problem using Semaphore and  Mutex |  |  |  |  |
| 6 | Solution to Dining Philosopher problem using Semaphore |  |  |  |  |
| 7 | Programs for deadlock avoidance algorithm |  |  |  |  |
| 8 | Programs for memory  management algorithms |  |  |  |  |
| 9 | Lab Internal Test conducted along the lines of SEE and valued for 50 Marks and reduced for 20 Marks | | | |  |
|  | **Total Marks** | | | |  |

**Component 1 = Lab Internal Marks =**

**Signature of the Staff In-charge**

# Laboratory 1

Title of the Laboratory Exercise: Programs using process management system calls

1. Introduction and Purpose of Experiment

A system call is a programmatic way in which a [computer program](https://en.wikipedia.org/wiki/Computer_program) requests a service from the [kernel](https://en.wikipedia.org/wiki/Kernel_(computing)) of the [operating system](https://en.wikipedia.org/wiki/Operating_system) it is executed on. There are different types of system calls developed for various purposes. They are mainly classified as process management, file management, directory management. By solving the problems students will be able to apply process management system calls

Aim and Objectives

Aim

* + To develop programs involving process management system calls

Objectives

At the end of this lab, the student will be able to

* + Use different process management system calls
  + Apply different system calls wherever required
  + Create C programs using process management system calls

1. Experimental Procedure
2. Analyse the problem statement
3. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
4. Implement the algorithm in C language

iv. Compile the C program

v. Test the implemented program vi. Document the Results vii. Analyse and discuss the outcomes of your experiment

1. Questions

Implement the following operations in C

Create 5 processes and distinguish parent and child processes. And also display process ID and its parent ID of all the created processes using process management system calls

1. Calculations/Computations/Algorithms

STEP 1: Start.

STEP 2: print output : getpid() , getppid()

STEP 3: for i = 0 to 4

3.1 if fork() == 0

3.1.1 print output : getpid() , getppid()

3.1.2 exit(0)

STEP 4: Stop

1. Presentation of Results

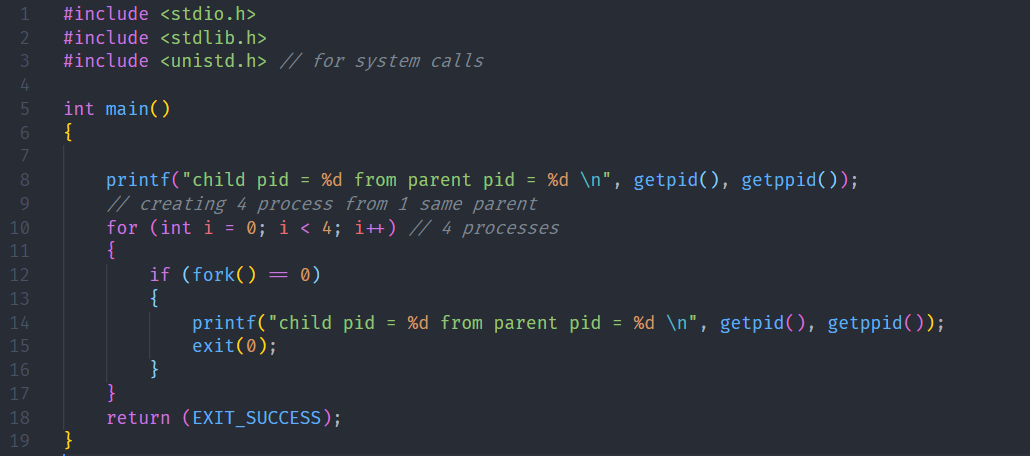


Figure Source Code

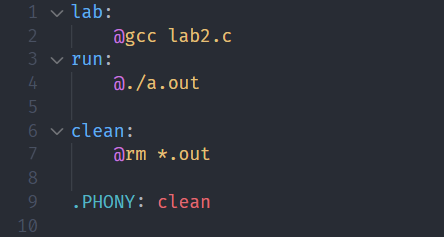


Figure makefile

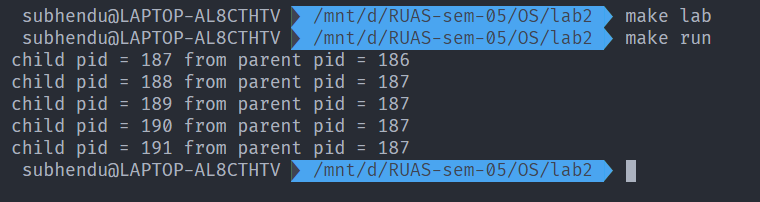


Figure execution

1. Analysis and Discussions

The fork() system call is a system call that is used to create process. This call creates a child process. The child process and the parent process, which called the fork() system call, execute the code that is present after the fork system called.

The parent and child processes do not share the same memory, but they can have multiple threads that share the same memory. The process run concurrently.

To distinguish the parent and the child, the value returned by the fork system call is used. For the parent process, the pid (process id) of the child is returned. For the child, the value returned is 0. If the value returned is negative, that means that there was an error that occurred while creating the child process.

The general intuition behind this program is that the parent processes execute the operations while the child operations call fork().

1. Conclusions

A system call that creates a new process identical to the calling one – Makes a copy of text, data, stack, and heap – Starts executing on that new copy

Uses of fork() – To create a parallel program with multiple processes

1. Comments
   1. Limitations of Experiments

The fork() system call creates processes. Each process consumes lot of system resources, such as CPU, memory, I/O etc. if used without care, it can lead to system overloads and other catastrophic failure.

* 1. Limitations of Results

The child and parent processes do not share the same memory space.

* 1. Learning happened

The method to create processes using the fork memory call was learned.

# Laboratory 2

Title of the Laboratory Exercise: Programs using file management system calls

1. Introduction and Purpose of Experiment

A system call is a programmatic way in which a [computer program](https://en.wikipedia.org/wiki/Computer_program) requests a service from the [kernel](https://en.wikipedia.org/wiki/Kernel_(computing)) of the [operating system](https://en.wikipedia.org/wiki/Operating_system) it is executed on. There are different types of system calls developed for various purposes. They are mainly classified as process management, file management, directory management. By solving the problems students will be able to apply file management system calls

Aim and Objectives

Aim

* + To develop programs involving file management system calls

Objectives

At the end of this lab, the student will be able to

* + Use different file management system calls
  + Apply different system calls wherever required
  + Create C programs using file management system calls

1. Experimental Procedure

i. Analyse the problem statement ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code iii. Implement the algorithm in C language iv. Compile the C program

v. Test the implemented program vi. Document the Results vii. Analyse and discuss the outcomes of your experiment

1. Questions

Implement the following command in C

Implement copy command (cp) to copy a file content to other file using file management system calls

1. Calculations/Computations/Algorithms

STEP 1: Start

STEP 2: buff string of size 100

STEP 3: inFile in\_file.txt file descriptor

STEP 4: outFile out\_file.txt file descriptor

STEP 5: bytesRead 0, bytesWritten 0

STEP 6: while bytesRead = read(inFile) and not EOF do

6.1: bytesWritten write to outFile

STEP 7: if bytesWritten is greater than 0, then

7.1: display success message

STEP 8: Stop

1. Presentation of Results

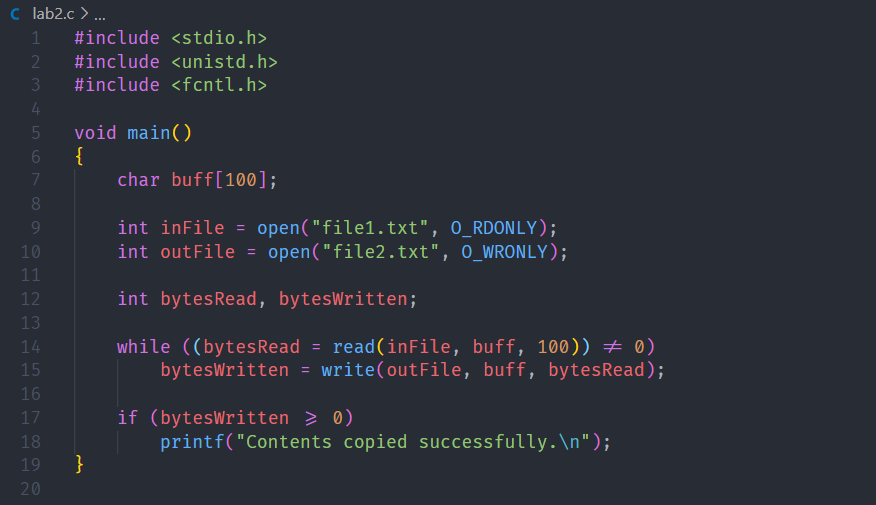


Figure Source Code

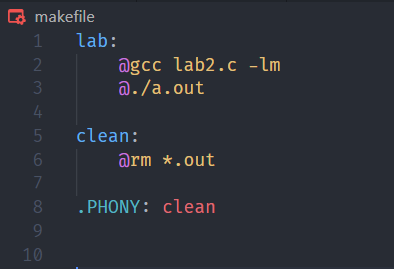


Figure Makefile

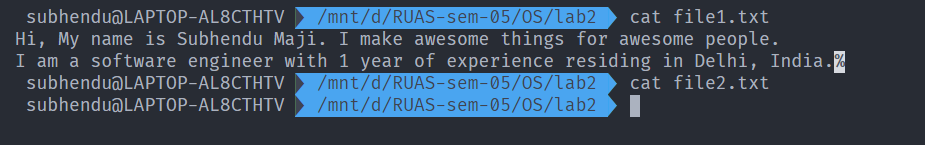


Figure Files before Execution

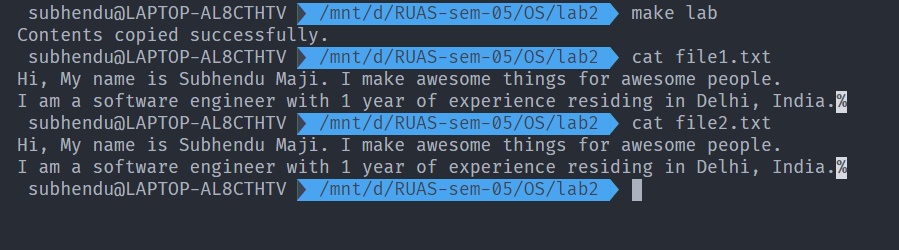


Figure Execution and file copied successfully.

1. Analysis and Discussions

In this experiment, System calls are used to copy the contents from one file to another file. To open a file, the open() system call was used. The open() system call takes the file path and the file mode as arguments. The function returns a file descriptor.

The read() system call is used to read a file that is open in read mode. The read() call takes the file descriptor to be read from, the buffer to store the data that is read and the amount of bytes to read. The function returns the number of bytes that was successfully read.

The write() system call is used to write data to a file that is open in write mode. The write() system call takes the output file descriptor, buffer containing the data and the number of bytes to write. The function returns the number of bytes that was successfully written to the file.

1. Conclusions

In this experiment, system calls were used to open, read and write to a file. These calls were used to make a mock ‘cp’ command. The program was implemented in C.

1. Comments

1. Limitations of Experiments

The read() system call reads from the file sequentially.

2. Limitations of Results

The use of system calls to implement the ‘cp’ command was learned.

3. Recommendations

readv() and writev() can be used, which read and write from multiple buffers at once.

# Laboratory 3

Title of the Laboratory Exercise: Programs based on multithreaded programming

1. Introduction and Purpose of Experiment

Multithreading is the ability of a processor or a single core in a multi-core processor to execute multiple [threads](https://en.wikipedia.org/wiki/Thread_(computer_science)) concurrently, supported by the [operating system.](https://en.wikipedia.org/wiki/Operating_system) By solving students will be able to manipulate multiple threads in a program.

1. Aim and Objectives

Aim

* + - To develop programs using multiple threads.

Objectives

At the end of this lab, the student will be able to

* + - Identify multiple tasks
    - Use threads constructs for creating threads
    - Apply threads for different/multiple tasks

1. Experimental Procedure

i. Analyse the problem statement ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code iii. Implement the algorithm in C language iv. Compile the C program

v. Test the implemented program vi. Document the Results vii. Analyse and discuss the outcomes of your experiment

1. Questions

Create multithreaded programs to implement the following

* + 1. Display “Hello World” message by 3 different threads
    2. Display thread IDs by each thread
    3. Create three threads;

 Thread1 add marks out of 10 from subject1 to subject5 of student 1, Thread2 adds from Subject1 to subject 5 of student 2 and Thread3 takes the sum from the Thread1 and Thread2 and decides who scored more marks. Display the total marks of the student (highest score) in parent process.

Instructions: Use a global variable to keep track of the sum from Thread1 and Thread2 and update it. Thread3 needs to wait until both Thread1 and Thread2 are done. Finally, parent process needs to wait until Thread3 is done. Get the return value from Thread3, and print the return value.

1. Calculations/Computations/Algorithms

**Algorithm of a.**

STEP 1: Start

STEP 2: define NUM\_THREADS ← 3

STEP 3: ret\_code ← 0, malloc threads array

STEP 4: for i=0 to NUM\_THREADS, do

4.1: ret\_code ← create thread that executes callback function (pthread\_create)

4.2: if ret\_code is positive, then display error message and exit.

STEP 5: Stop

void\* callback(void\* args);

STEP 1: Start

STEP 2: display “hello world”

STEP 3: Stop

**Algorithm of b.**

STEP 1: Start

STEP 2: define NUM\_THREADS ← 3

STEP 3: ret\_code ← 0, malloc threads array

STEP 4: for i=0 to NUM\_THREADS, do

4.1: ret\_code ← create thread that executes callback function (pthread\_create)

4.2: if ret\_code is positive, then display error message and exit.

STEP 5: Stop

void\* callback(void\* args);

STEP 1: Start

STEP 2: display “hello world : id = ” pthread\_self()

STEP 3: Stop

**Algorithm of c.**

STEP 1: Start

STEP 2: define student1 marks = [], student2 marks = []

STEP 3: sums = malloc (2\* sizeof(int))

STEP 4: create thread1, thread2, thread3

STEP 5: pthread\_create (&thread1 , NULL, sum, student1)

STEP 6: pthread\_create (&thread2 , NULL, sum, student2)

STEP 7: pthread\_join (&thread1 ,&sums[0])

STEP 8: pthread\_join (&thread2 ,&sums[1])

STEP 9: pthread\_create(&thread3 ,NULL, compare, sums)

STEP 10: pthread\_join (thread3 ,result)

STEP 11: print result

STEP 12: Stop

Void \***sum** (void \*arg)

STEP 1: define \*arr

STEP 2: for i = 0 to 5

2.1 res = res+arr[i]

STEP 3: return res

Void \***compare**(void \*arg)

STEP 1: define totals

STEP 2: if totals[0] >= totals[1]

2.1 return totals[0]

STEP 3: else

3.1 return totals[1]

1. Presentation of Results

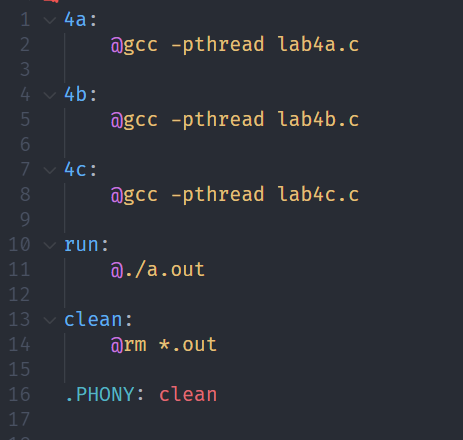


Figure makefile

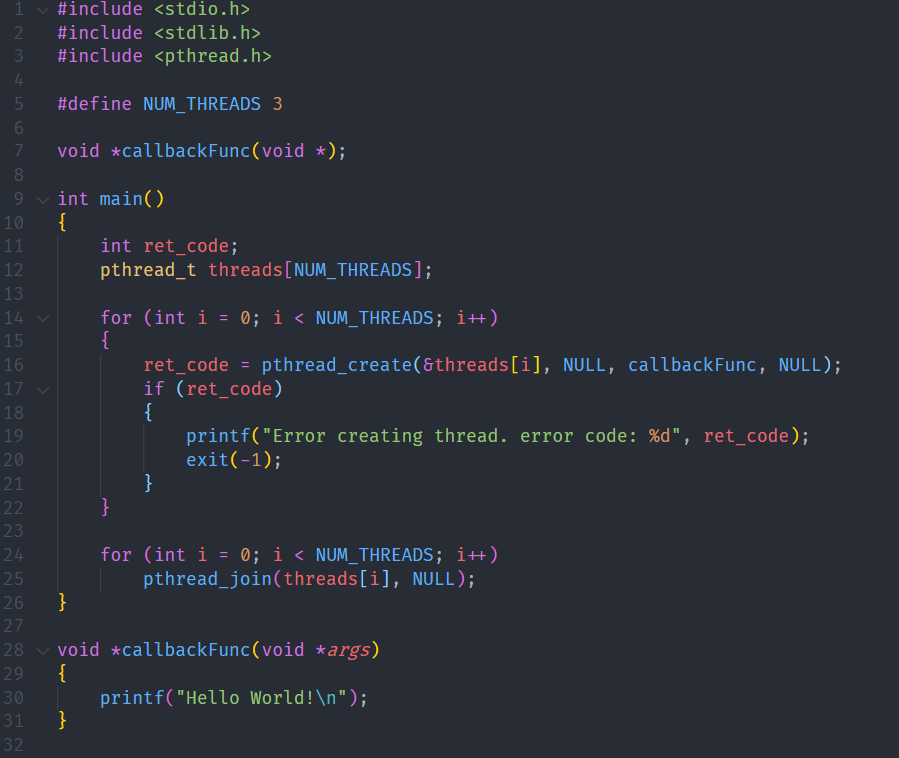


Figure Source Code of lab4a

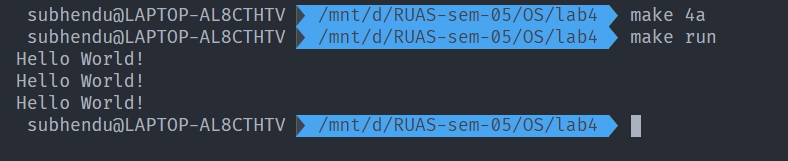


Figure Execution of a

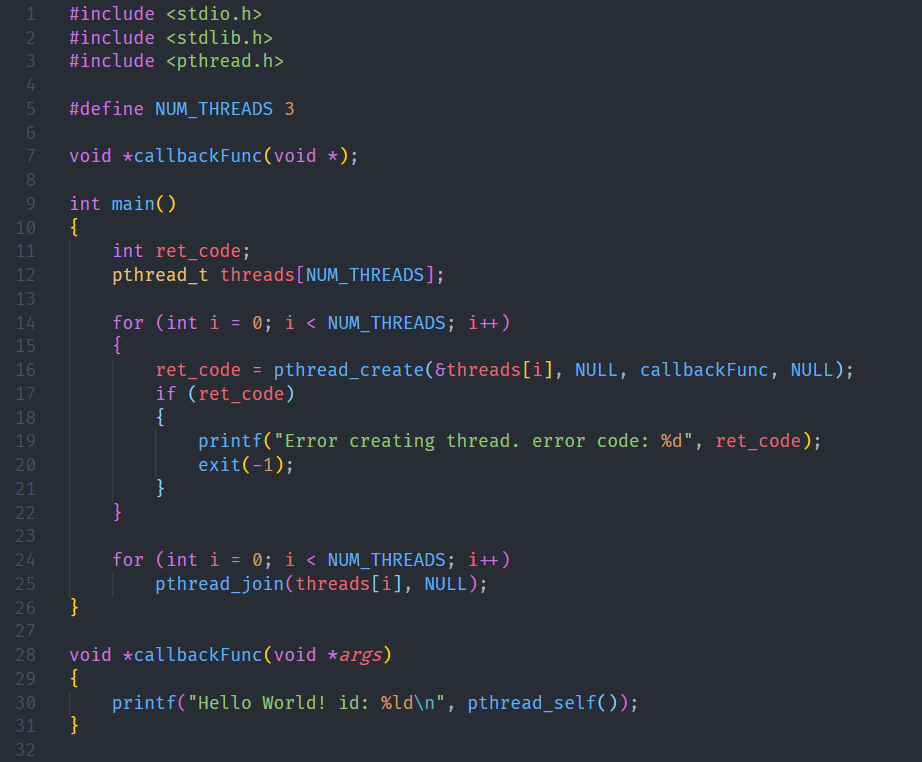


Figure Source code of lab4b

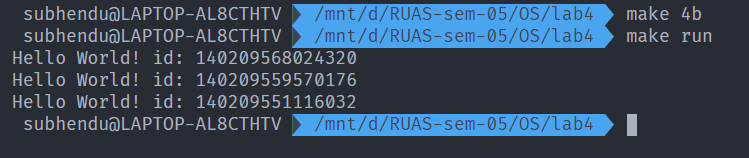


Figure execution of b

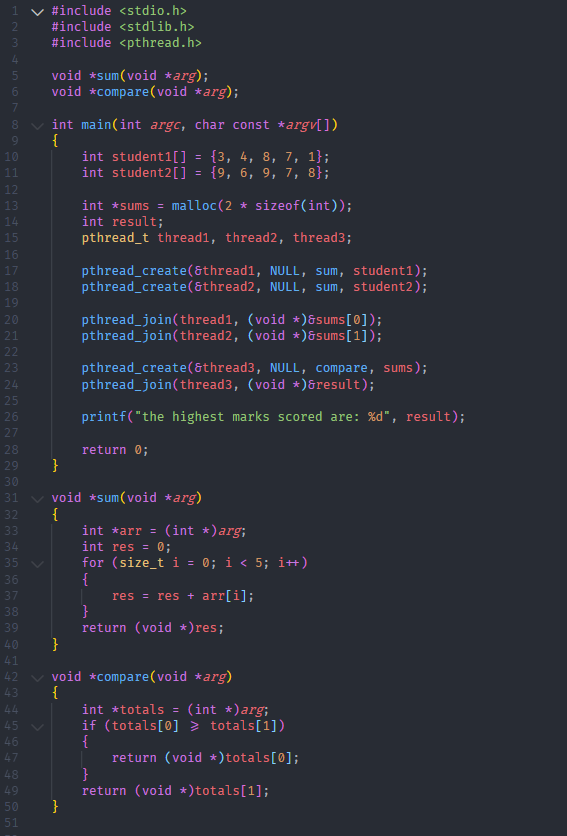


Figure sourcce code of lab4c

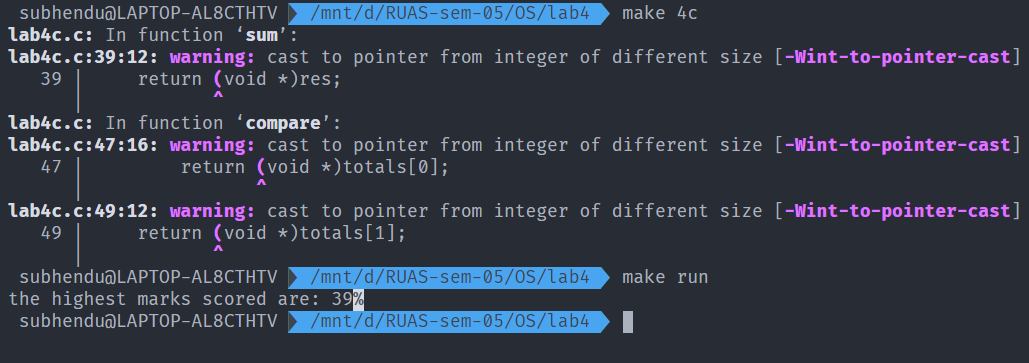


Figure execution of c

1. Analysis and Discussions

A thread is the smallest sequence of programming instructions that can be managed independently by the scheduler, which is typically part of the operating system. The implementation of a thread and a process differs from operating system to operating system, but in most cases, a thread is a component of a process. Multiple threads can exist in one single process, executing concurrently and sharing resources such as memory. This is unlike processes, since different processes do not share the same memory.

As mentioned above, threads can be used to execute code within a program concurrently. But care should be taken to avoid race conditions, i.e., a situation that occurs when two threads access a resource simultaneously, which leaves the resource in an unstable state.

In C, threads are created and used using POSIX threads, or more specifically, the pthread library. This library includes code that allows the developer to spawn and run posix threads. The function used to create a thread is called pthread\_create(). This thread has the following prototype:

pthread\_create(pthread\_t newThread, pthread\_attr\_t threadArgs, void\* (\*callback)(void\*), void\* args)

The function takes in four arguments. The first, a variable of type pthread\_t that defines the thread, the second, a variable of type pthread\_attr\_t that specifies thread arguments, third, a callback to execute, and fourth, the arguments to be passed to the callback.

To terminate a thread and merge the termination with the main thread, the function pthread\_join() is used. It has the following definition:

pthread\_join(pthread\_t thread, void\*\* return\_status)

the function takes in two arguments. One, the thread to terminate, and the second, a pointer to store the return status or the value returned by the callback which the thread executes. This function makes the main thread wait till the child thread finishes execution.

1. Conclusions

In this experiment, the ways to create a thread, pass arguments to it, obtain the return status etc, was learned. The program that demonstrated these functionalities was written in C using the POSIX thread library.

1. Comments
   1. Limitations of Experiments

The kernel doesn’t know about the existence of threads in a process. So regardless of whether a process has 1000 threads or 1 thread, both of them get the same time slice. Moreover, if a thread uses blocking system calls, the whole process is blocked in the kernel.

* 1. Limitations of Results

The code has not been optimized to handle race conditions and other unfavourable scenarios that occur due to the use of threads.

* 1. Learning happened

The ways to create, run and manage threads in C was learned. The disadvantages were highlighted and the advantages were clearly seen.

# Laboratory 4

Title of the Laboratory Exercise: Programs for process scheduling algorithms

1. Introduction and Purpose of Experiment

A Process Scheduler schedules different processes to CPU based on particular scheduling algorithms. There are various scheduling algorithms present in each group of operating system. By solving these problems students will be able use different scheduling algorithms as part of their implementation

1. Aim and Objectives

Aim

* + To develop programs to implement scheduling algorithms

Objectives

At the end of this lab, the student will be able to

* + Distinguish different scheduling algorithms
  + Apply the logic of scheduling algorithms wherever required  Create C programs to simulate scheduling algorithms

1. Experimental Procedure

i. Analyse the problem statement ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code iii. Implement the algorithm in C language iv. Compile the C program

v. Test the implemented program vi. Document the Results vii. Analyse and discuss the outcomes of your experiment

1. Questions

Write program to simulate the following process scheduling algorithms. Calculate average waiting time and average turnaround time for processes under each scheduling algorithm.

Instructions: Assume all the processes arrive at the same time. For FCFS and SJF scheduling algorithms, read the number of processes/jobs in the system and their CPU burst times. For round robin scheduling algorithm, read the number of processes in the system, their CPU burst times and the size of the time slice. For priority scheduling algorithm, read the number of processes in the system, their CPU burst times and the priorities.

* 1. First Come First Served
  2. Shortest Job First
  3. Priority
  4. Round Robin

1. Calculations/Computations/Algorithms

**Algorithm of First Come First Served (FCFS):**

STEP 1: Start

STEP 2: Input the processes along with their burst time (bt).

STEP 3: Find waiting time (wt) for all processes.

STEP 4: As first process that comes need not to wait so waiting time for process 1 will be 0 i.e. wt[0] = 0.

STEP 5: Find waiting time for all other processes i.e. for

process i ->

wt[i] = bt[i-1] + wt[i-1] .

STEP 6: Find turnaround time = waiting\_time + burst\_time

for all processes.

STEP 7: Find average waiting time =

total\_waiting\_time / no\_of\_processes.

STEP 8: Similarly, find average turn\_around\_time =

total\_turn\_around\_time / no\_of\_processes.

STEP 9: Stop

**Algorithm of Shortest Job First (SJF):**

STEP 1: Sort all the process according to the arrival time.

STEP 2: Then select that process which has minimum arrival time and minimum Burst time.

STEP 3: After completion of process make a pool of process which after till the completion of previous process and select that process among the pool which is having minimum Burst time.

**Algorithm of Priority Scheduling:**

STEP 1: First input the processes with their burst time and priority.

STEP 2: Sort the processes, burst time and priority according to the priority.

STEP 3: Now simply apply FCFS algorithm.

**Algorithm of Shortest Job First (SJF):**

STEP 1: Start

STEP 2: Create an array rem\_bt[] to keep track of remaining burst time of processes. This array is initially a copy of bt[] (burst times array)

STEP 3: Create another array wt[] to store waiting times processes. Initialize this array as 0.

STEP 4: Initialize time: t = 0

STEP 5: Keep traversing the all processes while all processes are not done. Do following for i 'th process if it is not done yet.

4.1 If rem\_bt[i] > quantum

4.1.1 t = t + quantum

4.1.2 bt\_rem[i] -= quantum;

4.2 Else // Last cycle for this process

4.2.1 t = t + bt\_rem[i];

4.2.2 wt[i] = t - bt[i]

4.2.3 bt\_rem[i] = 0; // This process is over

STEP 6: Stop.

1. Presentation of Results

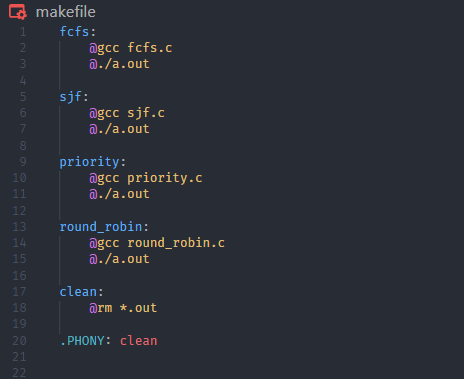


Figure makefile



Figure FCFS source code

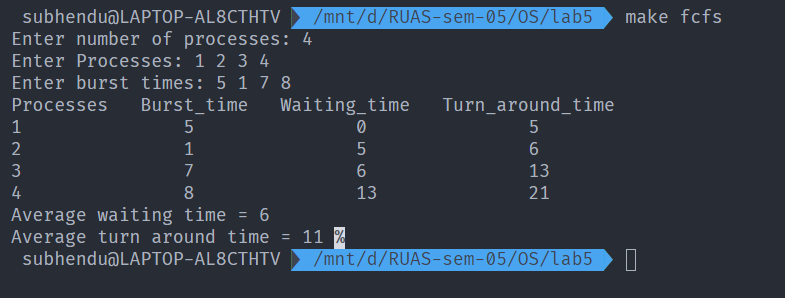


Figure Execution of FCFS

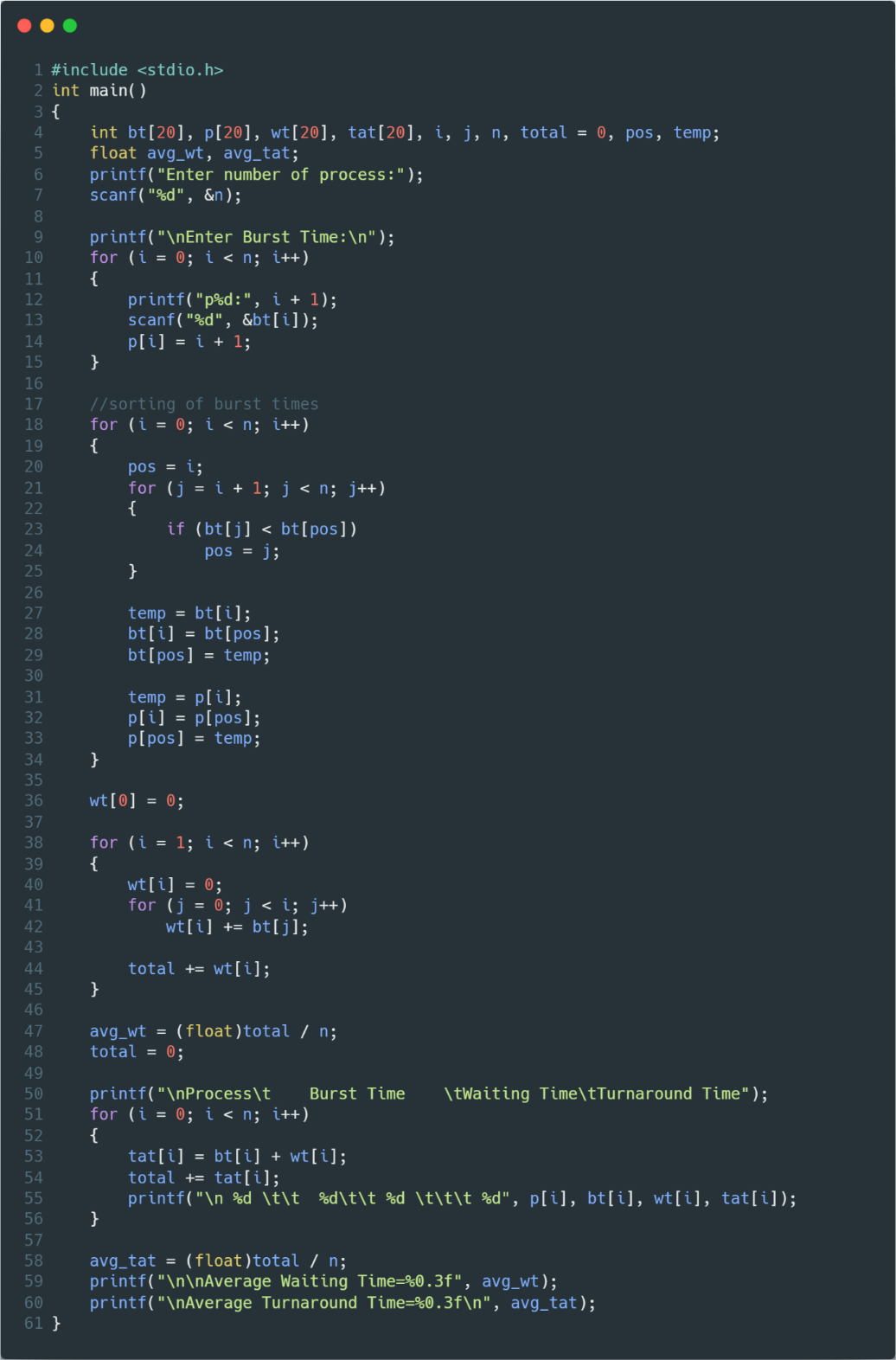


Figure SJF Source Code

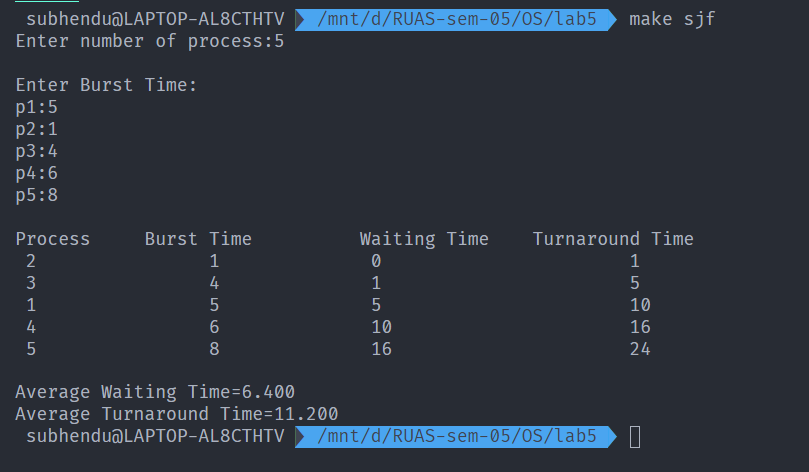


Figure Execution of SJF

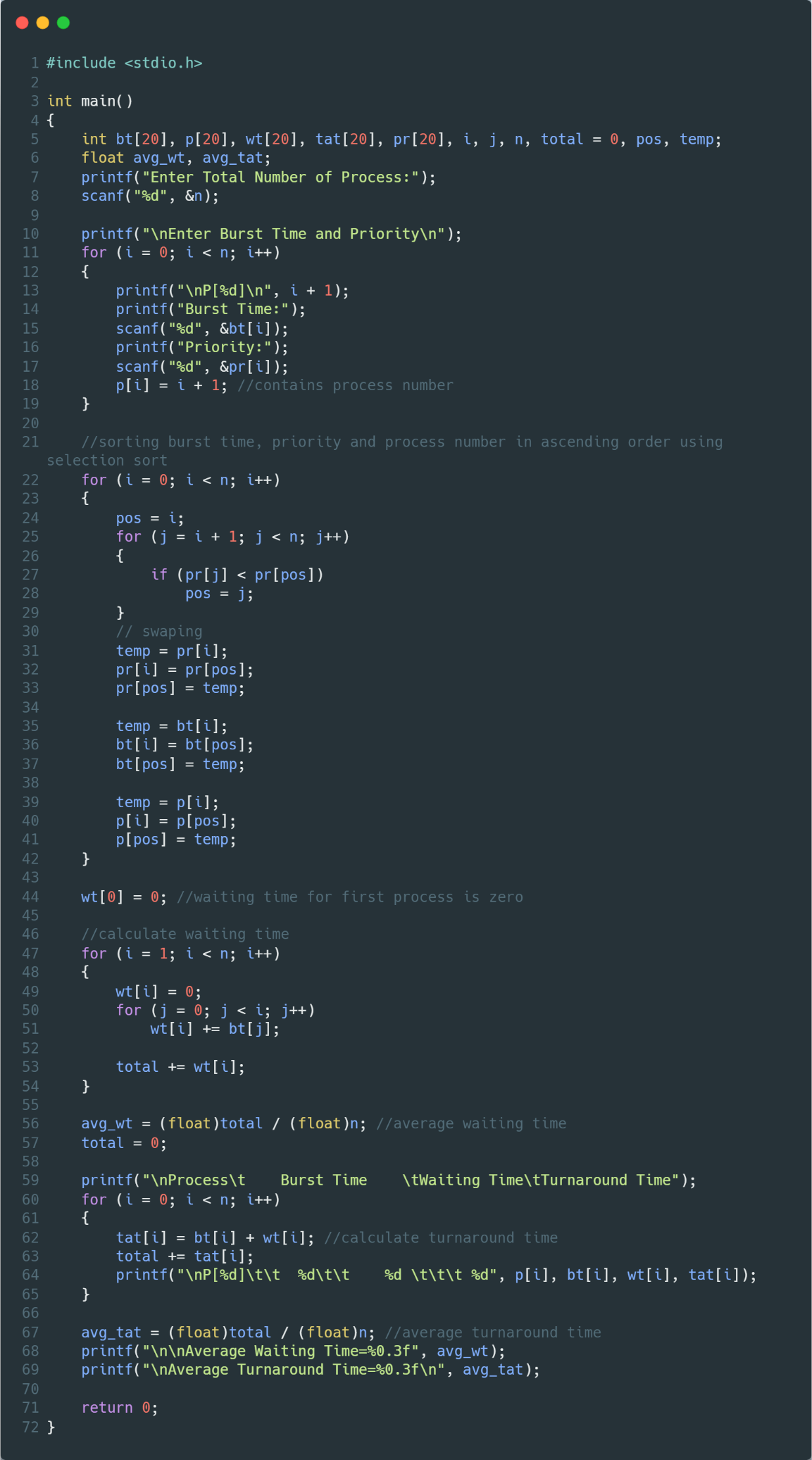


Figure Priority Scheduling Source Code

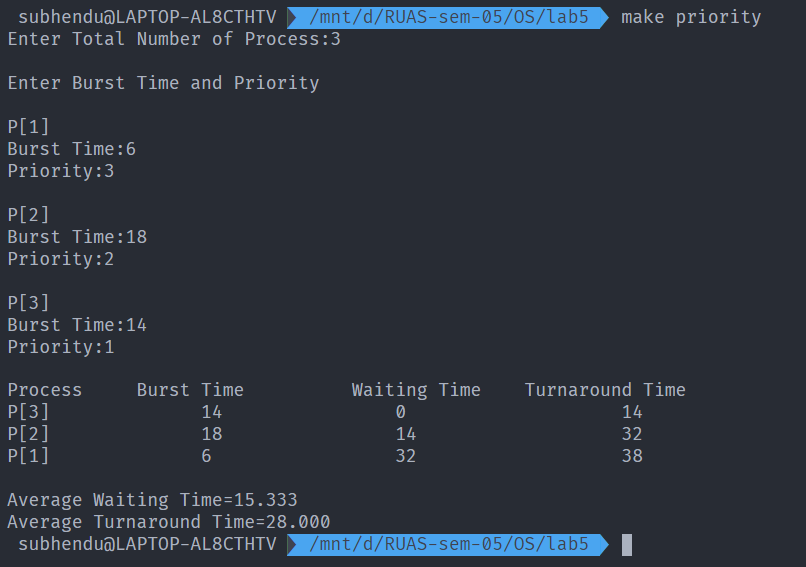


Figure Execution of Priority Scheduling



Figure Round Robin Source Code

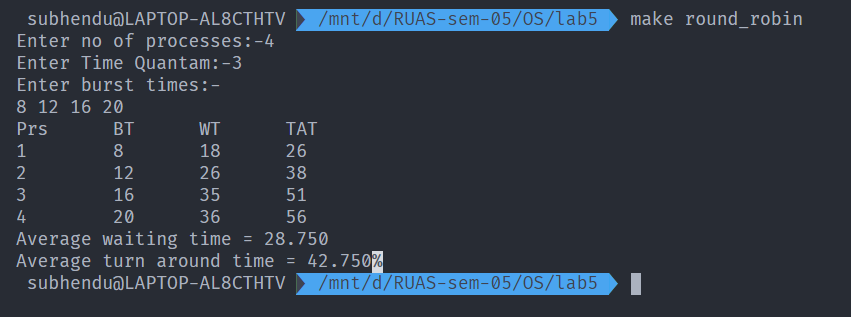


Figure Execution of Round Robin

1. Analysis and Discussions
   * **First Come First Served**

Given n processes with their burst times, the task is to find average waiting time and average turnaround time using FCFS scheduling algorithm.  
First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue.  
In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed.

* + **Shortest Job First**

Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non-pre-emptive algorithm.

* Shortest Job first has the advantage of having a minimum average waiting time among all scheduling algorithms.
* It is a Greedy Algorithm.
* It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.
* It is practically infeasible as Operating System may not know burst time and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available.
  + **Priority**

Priority scheduling is one of the most common scheduling algorithms in batch systems. Each process is assigned a priority. Process with the highest priority is to be executed first and so on.  
Processes with the same priority are executed on first come first served basis. Priority can be decided based on memory requirements, time requirements or any other resource requirement

* + **Round Robin**

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way.

* It is simple, easy to implement, and starvation-free as all processes get fair share of CPU.
* One of the most commonly used technique in CPU scheduling as a core.
* It is pre-emptive as processes are assigned CPU only for a fixed slice of time at most.
* The disadvantage of it is more overhead of context switching.

1. Conclusions

In FCFS,

It is simple and easy to understand

In SJF,

* Shortest jobs are favoured.
* It is provably optimal; in that it gives the minimum average waiting time for a given set of processes.

In Round Robin,

* Every process gets an equal share of the CPU.
* RR is cyclic in nature, so there is no starvation

In Priority,

This Scheduling provides a good mechanism where the relative importance of each process maybe precisely defined.

1. Comments
   1. Limitations of Experiments

**FCFS limitation**

In First Come First Serve scheduling the average waiting time is not optimal. ...

The First Come First Serve scheduling is Non-pre-emptive in nature. ...

As the First come First Serve scheduling is Non-pre-emptive, it does not understand the priority of processes.

**SJF Limitations**

SJF may cause starvation, if shorter processes keep coming. This problem is solved by aging. It cannot be implemented at the level of short-term CPU scheduling.

**Round Robin limitation**

If slicing time of OS is low, the processor output will be reduced. This method spends more time on context switching. Its performance heavily depends on time quantum. Priorities cannot be set for the processes

**Priority scheduling limitation**

The problem occurs when the operating system gives a particular task a very low priority, so it sits in the queue for a larger amount of time, not being dealt with by the CPU. If this process is something the user needs, there could be a very long wait, this process is known as “Starvation” or “Infinite Blocking”.

* 1. Learning happened

Implemented scheduling algorithms in C.

# Laboratory 6

Title of the Laboratory Exercise: Solution to Producer Consumer Problem using Semaphore and Mutex 1. Introduction and Purpose of Experiment

In multitasking systems, simultaneous use of critical section by multiple processes leads to data inconsistency and several other concurrency issues. By solving this problem students will be able to use Semaphore and Mutex for synchronisation purpose in concurrent programs. 2. Aim and Objectives

Aim

* To implement producer consumer problem using Semaphore and Mutex

Objectives

At the end of this lab, the student will be able to

* Use semaphore and Mutex
* Apply semaphore and Mutex in the required context
* Develop multithreaded programs with Semaphores and Mutex

1. Experimental Procedure

i. Analyse the problem statement ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code iii. Implement the algorithm in C language iv. Compile the C program

v. Test the implemented program vi. Document the Results vii. Analyse and discuss the outcomes of your experiment

1. Questions

Implement producer consumer problem by using the following

* + 1. Semaphore
    2. Mutex

1. Calculations/Computations/Algorithms

1. Presentation of Results

1. Analysis and Discussions

1. Conclusions

1. Comments

* 1. Limitations of Experiments

* 1. Limitations of Results

* 1. Learning happened

* 1. Recommendations

# Laboratory 7

Programs for deadlock avoidance algorithm

1. Introduction and Purpose of Experiment

Deadlocks can be avoided if certain information is available in advance. By solving these problems students will become familiar to avoid deadlock in advance with the available resource information

1. Aim and Objectives

Aim

* + - To develop Bankers algorithm for multiple resources for deadlock avoidance

Objectives

At the end of this lab, the student will be able to

* + - Verify a problem to check that whether deadlock will happen or not for the given resources
    - Implement the bankers algorithm for multiple resources

1. Experimental Procedure

i. Analyse the problem statement ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code iii. Implement the algorithm in C language iv. Compile the C program

v. Test the implemented program vi. Document the Results vii. Analyse and discuss the outcomes of your experiment

1. Questions

Implement a Bankers algorithm for deadlock avoidance

1. Calculations/Computations/Algorithms

1. Presentation of Results

1. Analysis and Discussions

1. Conclusions

1. Comments

* 1. Limitations of Experiments

* 1. Limitations of Results

* 1. Learning happened

* 1. Recommendations

# Laboratory 8

Programs for memory management algorithms

1. Introduction and Purpose of Experiment

In a multiprogramming system, the user part of memory must be further subdivided to accommodate multiple processes. This task of subdivision is carried out dynamically done by the operating system known as memory management. By solving these problems students will become familiar with the implementations of memory management algorithms in dynamic memory partitioning scheme of operating system.

1. Aim and Objectives

Aim

* + - To develop a simulator for memory management algorithms

Objectives

At the end of this lab, the student will be able to

* + - Apply memory management algorithms wherever required  Develop simulators for the algorithms

1. Experimental Procedure
   * + - Analyse the problem statement
       - Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
       - Implement the algorithm in C language
       - Compile the C program
       - Test the implemented program
       - Document the Results
       - Analyse and discuss the outcomes of your experiment

1. Questions

Implement a simulator for the memory management algorithms with the provision of compaction and garbage collection

* + 1. First fit
    2. Best fit
    3. Worst fit

1. Calculations/Computations/Algorithms

1. Presentation of Results

1. Analysis and Discussions

1. Conclusions

1. Comments

* 1. Limitations of Experiments

* 1. Limitations of Results

* 1. Learning happened

* 1. Recommendations

# Laboratory 9

Solution to Dining Philosopher problem using Semaphore

1. Introduction and Purpose of Experiment

In multitasking systems, simultaneous use of critical section by multiple processes leads to data inconsistency and several other concurrency issues. By solving this problem students will be able to use semaphore for synchronisation purpose in concurrent programs. 2. Aim and Objectives

Aim

* To develop concurrent programs using semaphores

Objectives

At the end of this lab, the student will be able to

* Use semaphore
* Apply appropriate semaphores in different contexts
* Develop concurrent programs using semaphores

1. Experimental Procedure

i. Analyse the problem statement ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code iii. Implement the algorithm in C language iv. Compile the C program

v. Test the implemented program vi. Document the Results vii. Analyse and discuss the outcomes of your experiment

1. Question

Implement the Dining Philosopher problem using POSIX threads

1. Calculations/Computations/Algorithms
2. Presentation of Results
3. Analysis and Discussions
4. Conclusions
5. Comments
   1. Limitations of Experiments
   2. Limitations of Results
   3. Learning happened
   4. Recommendations