NEAR EAST UNIVERSITY

GRADUATE SCHOOL OF APPLIED AND SOCIAL SCIENCES

ANALYSIS AND SIMULATION OF PROCESS SCHEDULING ALGORITHMS

Rami A H ALAJRAMI

MASTER THESIS

Department of Computer Engineering

Nicosia - 2006

Rami A H Alajrami : **Analysis and Simulation of Process Scheduling Algorithms**

Approval of the Graduate School of Applied and Social Sciences

Prof. Dr. Fakhraddin Mamedov
Director

We certify this thesis is satisfactory for the award of the Degree of Master of Science in Computer Engineering

Examining Committee in charge:

Assoc. Prof. Dr. Rahib Abiyev, Chairman , Computer Engineering Department, NEU

Assist. Prof. Dr. Firudin Muradov, Member , Computer Engineering Department, NEU

Assoc. Prof. Dr. Ilham Huseynov, Member, Computer Information System Department, NEU

Mr. Umit Ilhan , Member, Computer Engineering Department, NEU

Assist. Prof. Dr. Adil Amirjanov, Supervisor, Computer Engineering Department, NEU

ACKNOWLEDGMENTS

"Perhaps the most valuable result of all education is the ability to make yourself do the thing you have to do, when it ought to be done, whether you like it or not."
On second thought let it be only during your higher education not always, when education ends, you can be Normal again!

I would like to express my deepest thanks and gratitude to our Almighty God for enabling me to accomplish my goals; my Supervisor Assistant Professor Adil Amirjanov for all his devotion, time efforts ,and guidance; the soul of my deceased father, whose words and advices lightened my road and encouraged me to be someone to be proud of; my mother for her prayers, tolerance and patience that helped me to be who I am; my two sisters and brother for their continuous support love and belief in me, and finally, to those who helped me to accomplish this achievement.

ABSTRACT

Processor scheduling or CPU scheduling is the problem of determining when processors should be assigned and to which processes. The design of a scheduling discipline must consider many aims. Fairness, efficiency, response time, and throughput are the most important variables which scheduler should consider.

In this thesis a simulation tool is presented which is also a measurement system that analysis scheduling strategies with visualization which displays the scheduling decisions of any type of schedulers.

Using the measurement and visualization system, the performance of processes scheduler can be improved by comparison different methods of scheduling algorithms. This comparison shows the difference between seven methods of scheduling algorithms and chooses the efficient one for particular collection of user processes. By using this approach user can select and set up the particular efficient scheduling algorithm for particular collection of processes.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

OS: Operating System.

I/O: Inputs and Outputs.

CPU: Central Processing Unit.

SP: Stack Pointer.

PC: Program Counter.

PSW: Program Status Word.

PCB: Process control Block.

FCFS: First Come First Served.

FIFO: First In First Out.

SJF: Shortest Job First.

SRT: Shortest Remaining Time.

RR: Round Robin.

LJF: Longest Job First.

CDC: Control Data Corporation.

FB: Feed Back.

RM: Rate Monotonic.

EDF: Earliest Deadline First.

COTS: Commercially available Off-The-Shelf.

RTOS: Real Time Operating System

OO: Object Oriented.

HPC: High Performance Computing.

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Operating systems have been developed over the past 40 years for two main purposes:

- First, operating systems attempt to schedule computational activities to ensure good performance of the computing system,
- Second, they provide a convenient environment for the development and execution of programs.

Each computer allowed only one program to be executed at a time; this program had complete control of the system, and had access to all of the system's resources. Current-day computer systems allow multiple programs to be loaded into memory ad to be executed concurrently. This evolution required firmer control and more compartmentalization of the various programs; these needs resulted in the notion of a process, which is a program in execution.

The more complex the operating system, the more it is expected to do on behalf of its users. Although its main concern is the execution programs, it also needs to take care of various system tasks that are better left outside the kernel itself. All processes can potentially execute concurrently, with the CPU multiplexed among them. By switching the CPU between processes, the operating system can make the computer more productive.
A process migrates between the various scheduling queues throughout its lifetime. The operating system must for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler, It is important that the scheduler make a careful selection.

The objectives of the work presented within this thesis are to develop a simulation tool to improve the performance of process scheduler or manager, this tool presents seven different methods of scheduling algorithms, which enable the user to assess the difference between those methods.

This thesis is organized into five chapters. The first four chapters present background information about different types of operating systems, with detailed discussion about

scheduling algorithms, the final chapter describes the developed simulation tool for scheduling algorithms.

Chapter 1 is an introduction to operating system in general aspects; this chapter also discussed system components, operating system services, process definition, threads issues, context switch.

Chapter 2 is a detailed discussion about seven types of CPU scheduling algorithms which are First Come First Served (FCFS) scheduling, Shortest Job First (SJF) scheduling, Priority Scheduling, Round Robin scheduling (RR), Multilevel Queue scheduling, Multilevel Feedback scheduling, and Longest Job First scheduling.

Chapter 3 is based around the real time operating systems, with details about scheduling in real time systems, describing the methods of real time scheduling such as static and dynamic scheduling, commercially available off-the-shelf (COTS) and programming language choice are also discussed in this chapter.

Chapter 4 is a brief discussion about Linux operating system, which has an open source code that enable the user to change and improve the performance of the system according to his needs, this chapter also talked about scheduling in Linux operating system.

Chapter 5 presents the simulation tool that is developed by the author. In this chapter also a detailed description of the usage for this tool, with a solved example for each method of the seven previously well explained methods.

CHAPTER ONE

OPERATING SYSTEM'S ISSUES

1.1 Overview

The 1960's definition of an operating system is "the software that controls the hardware". However, today, due to microcode it is needed to have a better definition. It is common that operating system is the programs that make the hardware useable. In brief, an operating system is the set of programs that controls a computer. Some examples of operating systems are UNIX, Mach, MS-DOS, MS-Windows, Windows/NT, Chicago, OS/2, Mac OS, VMS, MVS, and VM.

The kernel responds to service calls from the processes and interrupts from the devices. The core of the operating system is the kernel, a control program that functions in privileged state (an execution context that allows all hardware instructions to be executed), reacting to interrupts from external devices and to service requests and traps from processes. Generally, the kernel is a permanent resident of the computer. It creates and terminates processes and responds to their request for service, according to figure 1.1 the rule of Operating System is described to typically between all the applications and the hardware devises.

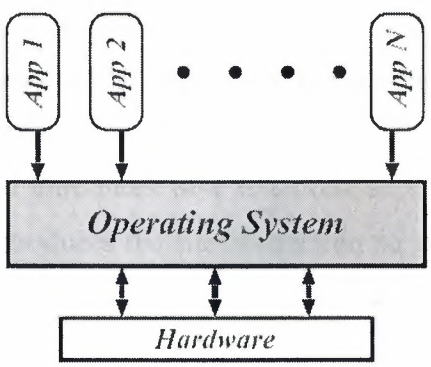


Figure 1.1 Operating System Rule

Operating Systems are resource managers. The main resource is computer hardware in the form of processors, storage, input/output devices, communication devices, and data. Some of the operating system functions are: implementing the user interface, sharing hardware among users, allowing users to share data among themselves, preventing users from interfering with one another, scheduling resources among users, facilitating input/output, recovering from errors, accounting for resource usage, facilitating parallel operations, organizing data for secure and rapid access, and handling network communications[1].

1.2 Objectives of Operating Systems

Modern Operating systems generally have following three major goals. Operating systems generally accomplish these goals by running processes in low privilege and providing service calls that invoke the operating system kernel in high-privilege state.

To hide details of hardware by creating abstraction which is software that hides lower level details and provides a set of higher-level functions. An operating system transforms the physical world of devices, instructions, memory, and time into virtual world that is the result of abstractions built by the operating system. There are several reasons for abstraction.

First, the code needed to control peripheral devices is not standardized. Operating systems provide subroutines called device drivers that perform operations on behalf of programs for example, input/output operations.
Second, the operating system introduces new functions as it abstracts the hardware. For instance, operating system introduces the file abstraction so that programs do not have to deal with disks.

Third, the operating system transforms the computer hardware into multiple virtual computers, each belonging to a different program. Each program that is running is called a

process. Each process views the hardware through the lens of abstraction. Fourth, the operating system can enforce security through abstraction.

To allocate resources to processes (Manage resources) an operating system controls how processes (the active agents) may access resources (passive entities).

Provide a pleasant and effective user interface the user interacts with the operating systems through the user interface and usually interested in the "look and feel" of the operating system. The most important components of the user interface are the command interpreter, the file system, on-line help, and application integration. The recent trend has been toward increasingly integrated graphical user interfaces that encompass the activities of multiple processes on networks of computers.

One can view Operating Systems from two points of views: Resource manager and extended machines. Form Resource manager point of view Operating Systems manage the different parts of the system efficiently and from extended machines point of view Operating Systems provide a virtual machine to users that is more convenient to use. The structurally Operating Systems can be design as a monolithic system, a hierarchy of layers, a virtual machine system, an exokernel, or using the client-server model [2]. The basic concepts of Operating Systems are processes, memory management, I/O management, the file systems, and security.

1.3 System Components

Even though, not all systems have the same structure many modern operating systems share the same goal of supporting the following types of system components

1.3.1 Process Management

The operating system manages many kinds of activities ranging from user programs to system programs like printer spooler, name servers, file server etc. Each of these activities is encapsulated in a process. A process includes the complete execution context (code, data, PC, registers, OS resources in use etc.).

It is important to note that a process is not a program. A process is only ONE instant of a program in execution. There are many processes can be running the same program. The five major activities of an operating system in regard to process management are:

- Creation and deletion of user and system processes.
- Suspension and resumption of processes.
- A mechanism for process synchronization.
- A mechanism for process communication.
- A mechanism for deadlock handling.

1.3.2 Main-Memory Management

Primary-Memory or Main-Memory is a large array of words or bytes. Each word or byte has its own address. Main-memory provides storage that can be access directly by the CPU. That is to say for a program to be executed, it must in the main memory.

The major activities of an operating in regard to memory-management are:

- Keep track of which part of memory are currently being used and by whom.
- Decide which process is loaded into memory when memory space becomes available.
- Allocate and de-allocate memory space as needed.

1.3.3 File Management

A file is a collected of related information defined by its creator. Computer can store files on the disk (secondary storage), which provide long term storage. Some examples of storage media are magnetic tape, magnetic disk and optical disk. Each of these media has its own properties like speed, capacity, data transfer rate, and access methods.

File systems normally organized into directories to ease their use. These directories may contain files and other directions.

The five main major activities of an operating system in regard to file management are:

1. The creation and deletion of files.
2. The creation and deletion of directions.
3. The support of primitives for manipulating files and directions.
4. The mapping of files onto secondary storage.
5. The back up of files on stable storage media.

1.3.4 I/O System Management

I/O subsystem hides the peculiarities of specific hardware devices from the user. Only the device driver knows the peculiarities of the specific device to which it is assigned, such devices are shown in details in the next figure 1.2, monitor, keyboard, IDE disk controller, parallel ports and serial ports are all connected to the PCI bus.

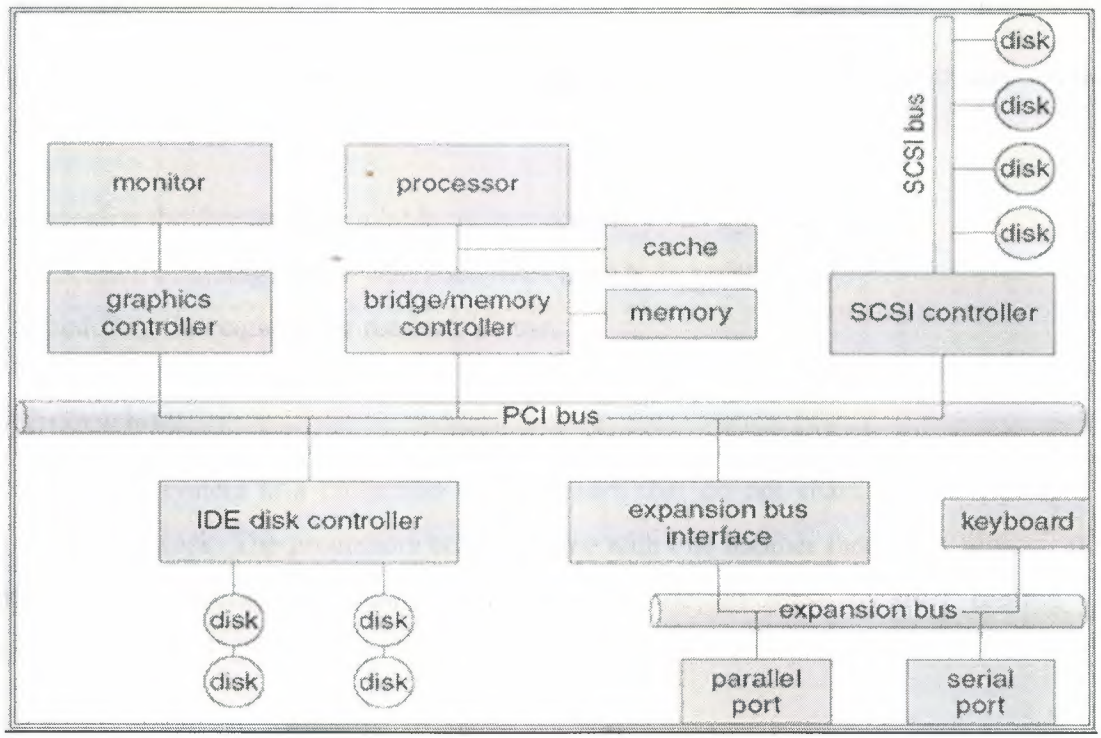


Figure 1.2 I/O Management and Connections

1.3.5 Secondary-Storage Management

Generally speaking, systems have several levels of storage, including primary storage, secondary storage and cache storage. Instructions and data must be placed in primary storage or cache to be referenced by a running program.

Because main memory is too small to accommodate all data and programs, and its data are lost when power is lost, the computer system must provide secondary storage to back up main memory. Secondary storage consists of tapes, disks, and other media designed to hold information that will eventually be accessed in primary storage (primary, secondary, cache) is ordinarily divided into bytes or words consisting of a fixed number of bytes. Each location in storage has an address; the set of all addresses available to a program is called an address space.

The three major activities of an operating system in regard to secondary storage management are:

1. Managing the free space available on the secondary-storage device.
2. Allocation of storage space when new files have to be written.
3. Scheduling the requests for memory access.

1.4 Networking

A distributed system is a collection of processors that do not share memory, peripheral devices, or a clock. The processors communicate with one another through communication lines called network, the communication-network design must consider routing and connection strategies, and the problems of contention and security.

1.5 Protection System

If a computer system has multiple users and allows the concurrent execution of multiple processes, then the various processes must be protected from one another's activities. Protection refers to mechanism for controlling the access of programs, processes, or users to the resources defined by computer systems.

1.6 Command Interpreter System

A command interpreter is an interface of the operating system with the user. The user gives commands with are executed by operating system (usually by turning them into system calls). The main function of a command interpreter is to get and execute the next user specified command.

Command-Interpreter is usually not part of the kernel, since multiple command interpreters (shell, in UNIX terminology) may be support by an operating system, and they do not really need to run in kernel mode. There are two main advantages to separating the command interpreter from the kernel.

1. If it is needed to change the way the command interpreter looks, i.e., to change the interface of command interpreter, it is possible to do that if the command interpreter is separate from the kernel. But it's not possible to change the code of the kernel so it's impossible to modify the interface.
2. If the command interpreter is a part of the kernel it is possible for a malicious process to gain access to certain part of the kernel that it showed not have to avoid this ugly scenario it is advantageous to have the command interpreter separate from kernel [3].

1.7 Operating Systems Services

Following are the five services provided by operating systems to the convenience of the users:

1.7.1 Program Execution

The purpose of a computer system is to allow the user to execute programs. So the operating system provides an environment where the user can conveniently run programs. The user does not have to worry about the memory allocation or multitasking or anything. These things are taken care of by the operating systems.

Running a program involves the allocating and de-allocating memory, CPU scheduling in case of multi-process. These functions cannot be given to the user-level programs. So user-

level programs cannot help the user to run programs independently without the help from operating systems.

1.7.2 I/O Operations

Each program requires an input and produces output. This involves the use of I/O. The operating systems hides the user the details of underlying hardware for the I/O. All the user sees is that the I/O has been performed without any details.

So the operating system by providing I/O makes it convenient for the users to run programs. For efficiently and protection users cannot control I/O so this service cannot be provided by user-level programs.

1.7.3 File System Manipulation

The output of a program may need to be written into new files or input taken from some files. The operating system provides this service. The user does not have to worry about secondary storage management. User gives a command for reading or writing to a file and sees his or her task accomplished. Thus operating system makes it easier for user programs to accomplish their task. This service involves secondary storage management. The speed of I/O that depends on secondary storage management is critical to the speed of many programs and hence the best relegated to the operating systems to manage it than giving individual users the control of it. It is not difficult for the user-level programs to provide these services but for above mentioned reasons it is best if this service s left with operating system.

1.7.4 Communications

There are instances where processes need to communicate with each other to exchange information.

It may be between processes running on the same computer or running on the different computers. By providing this service the operating system relieves the user of the worry of passing messages between processes.

In case where the messages need to be passed to processes on the other computers through a network it can be done by the user programs. The user program may be customized to the specifics of the hardware through which the message transits and provides the service interface to the operating system.

1.7.5 Error Detection

An error is one part of the system may cause malfunctioning of the complete system. To avoid such a situation the operating system constantly monitors the system for detecting the errors. This relieves the user of the worry of errors propagating to various part of the system and causing malfunctioning.

This service cannot allow to be handled by user programs because it involves monitoring and in cases altering area of memory or de-allocation of memory for a faulty process.

The CPU may be relinquished of a process that goes into an infinite loop. These tasks are too critical to be handled over to the user programs. A user program if given these privileges can interfere with the correct (normal) operation of the operating systems.

1.8 System Calls and System Programs

System calls provide an interface between the process and the operating system. System calls allow user-level processes to request some services from the operating system which process itself is not allowed to do.

In handling the trap, the operating system will enter in the kernel mode, where it has access to privileged instructions, and can perform the desired service on the behalf of user-level process; it is because of the critical nature of operations that the operating system itself does them every time they are needed.

For example, for I/O a process involves a system call telling the operating system to read or write particular area and this request is satisfied by the operating system.

System programs provide basic functioning to users so that they do not need to write their own environment for program development (editors, compilers) and program execution (shells). In some sense, they are bundles of useful system calls.

1.9 Layered Approach Design

In this case the system is easier to debug and modify, because changes affect only limited portions of the code, and programmer does not have to know the details of the other layers. Information is also kept only where it is needed and is accessible only in certain ways, so bugs affecting that data are limited to a specific module or layer.

1.10 Mechanisms and Policies

The policies what is to be done while the mechanism specifies how it is to be done? For instance, the timer construct for ensuring CPU protection is mechanism. On the other hand, the decision of how long the timer is set for a particular user is a policy decision; the separation of mechanism and policy is important to provide flexibility to a system.

If the interface between mechanism and policy is well defined, the change of policy may affect only a few parameters. On the other hand, if interface between these two is vague or not well defined, it might involve much deeper change to the system.

Once the policy has been decided it gives the programmer the choice of using his/her own implementation. Also, the underlying implementation may be changed for a more efficient one without much trouble if the mechanism and policy are well defined. Specifically, separating these two provides flexibility in a variety of ways.

First, the same mechanism can be used to implement a variety of policies, so changing the policy might not require the development of a new mechanism, but just a change in parameters for that mechanism, but just a change in parameters for that mechanism from a library of mechanisms.

Second, the mechanism can be changed for example, to increase its efficiency or to move to a new platform, without changing the overall policy [4].

1.11 Definition of Process

The notion of process is central to the understanding of operating systems. There are quite a few definitions presented in the literature, but no "perfect" definition has yet appeared.

1.11.1 Definition

The term "process" was first used by the designers of the MULTICS in 1960's. Since then, it was used somewhat interchangeably with 'task' or 'job'. The process has been given many definitions for instance

- A program in Execution.
- An asynchronous activity.
- The 'animated sprit' of a procedure in execution.
- The entity to which processors are assigned.
- The 'dispatch able' unit.

And many more definitions have given. As it is explained above that there is no universally agreed upon definition, but the definition "Program in Execution" seem to be most frequently used.

Now that it has been agreed upon the definition of process, the question is what the relation between process and program is. It is same beast with different name or when this beast is sleeping (not executing) it is called program and when it is executing becomes process. To be very precise, Process is not the same as program. In the following discussion the difference between process and program will be presented.

A process is more than a program code. A process is an 'active' entity as oppose to program which consider being a 'passive' entity. It is well known that a program is an algorithm expressed in some suitable notation, (e.g., programming language). Being a passive, a program is only a part of process. Process, on the other hand, includes:

- Current value of Program Counter (PC)
- Contents of the processors registers

- Value of the variables
- The process stacks (SP) which typically contains temporary data such as subroutine parameter, return address, and temporary variables.
- A data section that contains global variables.

A process is the unit of work in a system.

In Process model, all software on the computer is organized into a number of sequential processes. A process includes PC, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, the CPU switches back and forth among processes. (The rapid switching back and forth is called multiprogramming).

1.11.2 Process Resuming

The process state consist of everything necessary to resume the process execution if it is somehow put aside temporarily. The process state consists of at least following:

- Code for the program.
- Program's static data.
- Program's dynamic data.
- Program's procedure call stack.
- Contents of general purpose register.
- Contents of program counter (PC)
- Contents of program status word (PSW).
- Operating Systems resource in use [5].

1.11.3 Process Operations

1.11.3.1 Process Creation

In general-purpose systems, some way is needed to create processes as needed during operation. There are four principal events led to processes creation.

- System initialization.
- Execution of a process Creation System calls by a running process.
- A user request to create a new process.
- Initialization of a batch job.

Following are some reasons for creation of a process

- User logs on.
- User starts a program.
- Operating systems creates process to provide service, e.g., to manage printer.
- Some program starts another process, e.g., Netscape calls xv to display a picture.

1.11.3.2 Process Termination

A process terminates when it finishes executing its last statement. Its resources are returned to the system, it is purged from any system lists or tables, and its process control block (PCB) is erased i.e., the PCB's memory space is returned to a free memory pool. The new process terminates the existing process, usually due to following reasons:

- Normal Exist Most processes terminates because they have done their job. This call is exist in UNIX.
- Error Exist when process discovers a fatal error. For example, a user tries to compile a program that does not exist.
- Fatal Error An error caused by process due to a bug in program for example, executing an illegal instruction, referring non-existing memory or dividing by zero.
- Killed by another Process, a process executes a system call telling the Operating Systems to terminate some other process. In UNIX, this call is killed. In some systems when a process kills all processes it created are killed as well (UNIX does not work this way).

1.11.4 Process States

A process goes through a series of discrete process states.

- New State The process being created.
- Terminated State The process has finished execution.
- Blocked (waiting) State When a process blocks, it does so because logically it cannot continue, typically because it is waiting for input that is not yet available. Formally, a process is said to be blocked if it is waiting for some event to happen (such as an I/O completion) before it can proceed. In this state a process is unable to run until some external event happens.
- Running State a process is said to be running if it currently has the CPU that is, actually using the CPU at that particular instant.
- Ready State A process is said to be ready if it use a CPU if one were available. It is run-able but temporarily stopped to let another process run.

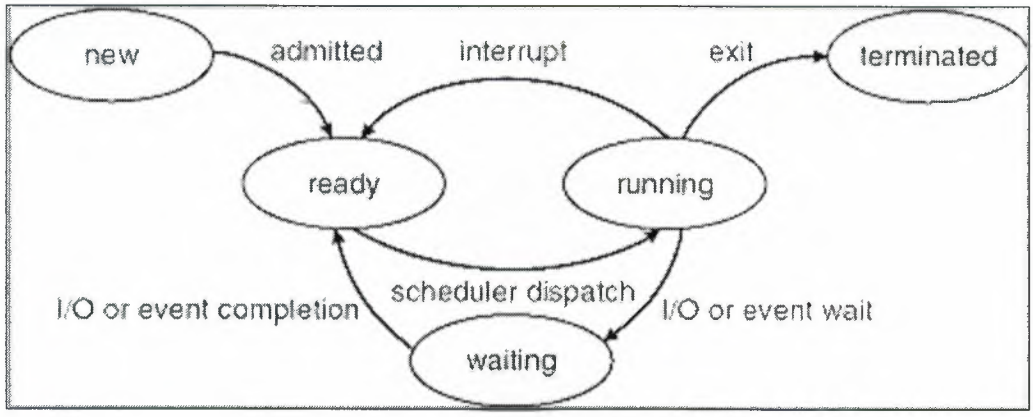


Figure 1.3 Process statuses through execution

Logically, the 'Running' and 'Ready' states are similar. In both cases the process is willing to run, only in the case of 'Ready' state, there is temporarily no CPU available for it. The 'Blocked' state is different from the 'Running' and 'Ready' states in that the process cannot run, even if the CPU is available.

1.11.5 Process Control Block

A process in an operating system is represented by a data structure known as a process control block (PCB) or process descriptor. The PCB contains important information about the specific process including

- The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- Unique identification of the process in order to track "which is which" information.
- A pointer to parent process.
- Similarly, a pointer to child process (if it exists).
- The priority of process (a part of CPU scheduling information).
- Pointers to locate memory of processes.
- A register save area.
- The processor it is running on.

The PCB (figure 1.4) is a certain store that allows the operating systems to locate key information about a process. Thus, the PCB is the data structure that defines a process to the operating systems [6].

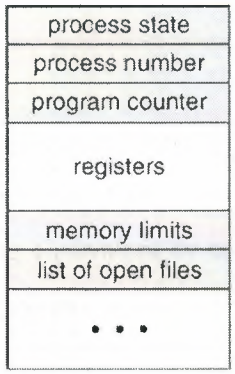


Figure 1.4 Process Control Block (PCB)

CHAPTER TWO

SCHEDULING ALGORITHMS OF PROCESS MANAGEMENT

2.1 Overview

The assignment of physical processors to processes allows processors to accomplish work. The problem of getting the time in which processor should be assigned and to which process is called processor scheduling or CPU scheduling.

When more than one process is run-able, the operating system must decide which one first. The part of the operating system concerned with this decision is called the scheduler, and algorithm it uses is called the scheduling algorithm.

Many objectives must be considered in the design of a scheduling discipline. In particular, a scheduler should consider fairness, efficiency, response time, turnaround time, throughput, etc., Some of these goals depends on the system one is using for example batch system, interactive system or real-time system, etc. but there are also some goals that are desirable in all systems.

- **Fairness**

  Fairness is important under all circumstances. A scheduler makes sure that each process gets its fair share of the CPU and no process can suffer indefinite postponement. Note that giving equivalent or equal time is not fair. Think of safety control and payroll at a nuclear plant.

- **Policy Enforcement**

  The scheduler has to make sure that system's policy is enforced. For example, if the local policy is safety then the safety control processes must be able to run whenever they want to, even if it means delay in payroll processes.

- **Efficiency**

  Scheduler should keep the system (or in particular CPU) busy cent percent of the time when possible. If the CPU and all the Input/Output devices can be kept running all the time, more work gets done per second than if some components are idle.

- **Response Time**

  A scheduler should minimize the response time for interactive user.

- **Turnaround**

  A scheduler should minimize the time batch users must wait for an output.

- **Throughput**

  A scheduler should maximize the number of jobs processed per unit time. A little thought will show that some of these goals are contradictory. It can be shown that any scheduling algorithm that favors some class of jobs hurts another class of jobs. The amount of CPU time available is finite, after all.

2.2 Preemptive Vs Non-preemptive Scheduling

The Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts:

2.2.1 Non-preemptive Scheduling

A scheduling discipline is non-preemptive if, once a process has been given the CPU; the CPU cannot be taken away from that process.

Following are some characteristics of non-preemptive scheduling

1. In non-preemptive system, short jobs are made to wait by longer jobs but the overall treatment of all processes is fair.
2. In non-preemptive system, response times are more predictable because incoming high priority jobs can not displace waiting jobs.

3. In non-preemptive scheduling, a scheduler executes jobs in the following two situations.

a. When a process switches from running state to the waiting state.

b. When a process terminates.

2.2.2 Preemptive Scheduling

A scheduling discipline is preemptive if, once a process has been given the CPU can taken away. The strategy of allowing processes that are logically run-able to be temporarily suspended is called Preemptive Scheduling and it is contrast to the "run to completion" method [7].

2.3 Scheduling Algorithms

CPU Scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

Following are some scheduling algorithms to be discussed

- FCFS Scheduling.
- SJF Scheduling. .
- Priority Scheduling.
- Round Robin Scheduling.
- Multilevel Queue Scheduling.
- Multilevel Feedback Queue Scheduling.
- Longest Job First Scheduling.

2.3.1 First-Come-First-Served (FCFS) Scheduling

By far the simplest CPU scheduling algorithm is the first-come, first served scheduling (FCFS) algorithm. With this scheme, the process that requests the CPU first is all allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue.

When a process enters the ready queue, its process control block is linked onto the tail of the queue. When the CPU is free it is allocated to the process at the head of the queue. The running process is the removed from the queue. The code for FCFS scheduling is simple to write and understand.

The average waiting time under the FCFS policy, however is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in unit.

If the processes arrive in the order P1, P2, P3, and are served in FCFS order, the next result will be determined and shown in the following Gantt chart:

Process	Burst Time
P_1	24
P_2	3
P_3	3

Table 2.1 Processes Burst Time for FCFS Scheduling Algorithm

Figure 2.1 First Come First Served Execution Progress

The waiting time is 0 units for process P1,24 units for process P2, and 27 units for P3, thus, the average waiting time is (0+24+27)/3 = 17 units.

In addition, consider the performance of FCFS scheduling in the dynamic situation. Assume there is a one CPU-bound process and many I/O –bound processes.
As the processes flow around the system, the following scenario may result. The CPU process will get the CPU and hold it. During this time, all the other processes will finish their I/O and move into the ready queue, waiting for the CPU.

While the processes wait in the ready queue, the input output devices are idle. Eventually, the CPU bound process finishes its CPU burst and moves to an I/O device.

All the I/O bound processes, which have very short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting the ready queue until the CPU-bound process is done.

There is a convoy-effect as all the other processes wait the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

The FCFS scheduling algorithm is none preemptive. Once the CPU has allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. the FCFS algorithm is particularly trouble some for time sharing system; where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

2.3.2 Shortest-Job-First (SJF) Scheduling

A different approach to CPU scheduling is the Shortest-Job-First (SJF) algorithm. This algorithm associates with each process the length of the latter's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie. Note that more appropriate term would be the shortest next CPU burst, because the scheduling is done by examining the length of the next CPU burst of a process, rather than its total length, as an example, consider the following set of processes, with the length of the CPU burst and arrival time given in units:

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

Table 2.2 Processes Burst Time for Non-preemptive SJF Scheduling Algorithm

The Gantt chart is:

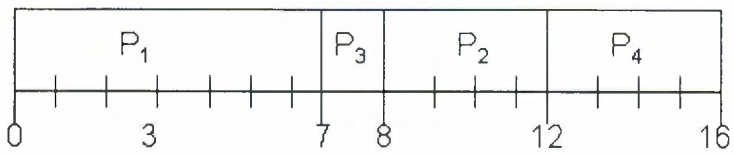


Figure2.2 Shortest Job First (Non-Preemptive) Execution Progress

- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. By moving short process before along one the waiting time of the short process decreases more than it increase the waiting time of the long process. Consequently the average waiting time decreases.

The real difficulty with the SJF algorithm is to know the length of the next CPU request. For long-term (job) scheduling in a batch system, the length of the process time limit could be used that a user specifies when the job is submitted.

Thus, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response. SJF scheduling is used frequently in long-term scheduling.

Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling.

There is no way to know the length of the next CPU burst. One approach is to try to approximate SJF scheduling. It is not possible to know the length of the next CPU burst; but it is possible to predict its value. The next CPU burst will be expected and it will be similar in length to the previous ones.

Thus by computing an approximation of the length of the next CPU burst, the process could be picked with the shortest predicted CPU burst.
The SJF algorithm may be either preemptive or non-preemptive. The choice arises when a new process arrives at the ready queue while a previous process is executing.

The new process may have a shorter next CPU burst than what is left of the currently executing process. A preemptive SJF will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.

Preemptive SJF scheduling is sometimes called shortest-remaining-time-first (SRT) scheduling, as an example; consider the following four processes, with the length of the CPU-burst and arrival time given in units:

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

Table 2.3 Processes Burst Time for Preemptive SJF Scheduling Algorithm

The Gantt chart is:

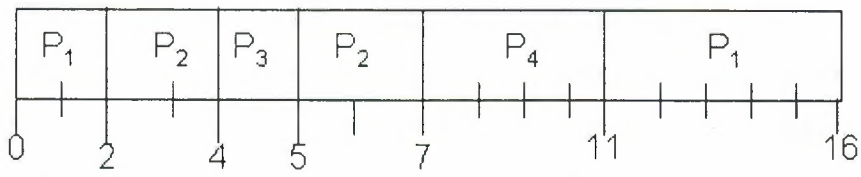


Figure 2.3 Shortest Job First (preemptive) Execution Progress

Process P1 is started at time 0, since it is the only process in the queue, process arrives at time 2. The remaining time for process P1 (5 units) is larger than the time required by process P2 (2 units), so process P1 is preempted, and process P2 is scheduled.

The average waiting time = ((11-2) + (3-2) + (4-4) + (7-5))/4 = 3

2.3.3 Priority Scheduling

The SJF algorithm is a special case of the general priority scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

An SJF algorithm is simply a priority algorithm where the priority (P) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa. Priorities are generally some fixed range of numbers, such as 0 to 7, or 0 to 4095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; other use low number for high priority. This difference can lead to confusion; in this text low numbers represent high priorities.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and ratio of average I/O burst to average CPU burst have been used in computing priorities.

External priorities are set by criteria that are external operating system, such as the importance of the process, the type and the amount of funds being paid for computer use, the department sponsoring the work, and other, often political factors.
Priority scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue, its priorities are compared with the priorities of the currently running process.

A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is the higher than the priority of the currently running process. A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithm is indefinite blocking or starvation.
A process that is ready to run but lacking the CPU can be considered blocked, waiting for the CPU. A priority scheduling algorithm can leave some low-priority processes waiting indefinitely for the CPU.

In the heavily loaded computer system, a steady stream of higher priority processes can prevent low priority process from ever getting the CPU. Generally one of two things will happen, either the process will eventually be run, or the computer system eventually crash and lose all unfinished low-priority processes.
A solution to the problem of indefinite blockage of low priority processes is aging; aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

2.3.4 Round Robin Scheduling

The round-robin (RR) scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a time quantum, or time slice, is defined. A time quantum is generally from 10 to 100 units. The ready queue is treated as a circular queue.

The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, the ready queue will be kept as a FIFO queue of the processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy, however, is often quite long.

Consider the following set of process that arrives at time 0, with the length of the CPU burst time given in units:

Process	Burst Time
P1	53
P2	17
P3	68
P4	24

Table 2.4 Processes Burst Time for RR Scheduling Algorithm

The Gantt chart is:

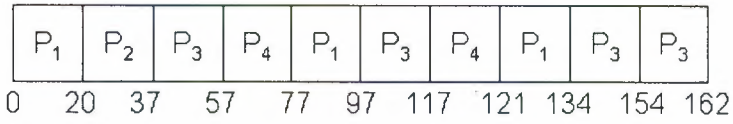


Figure2.4 Round Robin Execution Progress

Here a time quantum of 20 units is used, then process P1 gets the first 20 units. Since it requires another 33 units, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2; since process 2 does not need 20 units, it quits before its time, quantum expires. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The average waiting time is (302 / 4) = 75.5 units.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row. If a process CPU burst exceeds 1 time quantum, that process is preempted is put back in the ready queue. The RR scheduling algorithm is preemptive.
If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units. Each process must wait no longer that (n – 1) x q time units until its next time quantum.

The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is very large (infinite), the RR policy is the same as the FCFS policy.
If the time quantum is very small the RR approach is called processor sharing, and appears to the users as though each of n processes has its own processor running at 1/n the speed of the real processor. This approach was used in Control Date Corporation (CDC) hardware to implement 10 peripherals processors with only one set of hardware and 10 sets of registers. The hardware executes one instruction for one set of registers, then goes on to the next. This cycle continues, resulting in 10 slow processors rather than one fast one.

2.3.5 Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common division is made between foreground (interactive) processes and background (batch) processes, these two types of processes have different response-time requirements, and so might have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.

A multilevel queue-scheduling algorithm partitions the ready queue into several separate queues. The process is permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling between the queues, which is commonly implemented as a fixed- priority preemptive scheduling.
For example, the foreground queue may have absolute priority over the background queue.
The next figure shows an example of a multilevel queue scheduling algorithm with five queues:

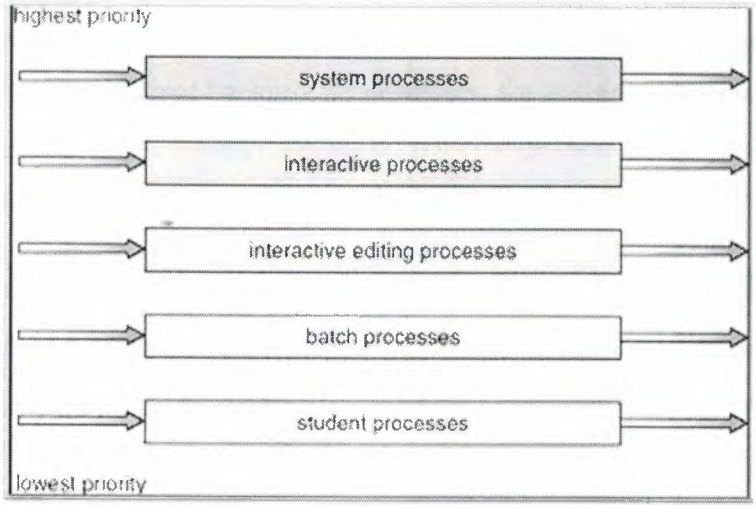


Figure 2.5 Multilevel Queue Scheduling

1. System process.
2. Interactive process.
3. Interactive editing process.
4. Batch process.
5. Student process

Each queue has absolute priority over lower- priority queues. No process in the batch queue, for example could run unless the queues for system process, interactive process, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time slice between the queues. Each queue gets certain portion of the CPU time, which it can then schedule among the various processes in its queues. For instance, in the foreground- background queue example, the foreground can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives twenty percents of the CPU to give to its processes in a FCFS manner.

2.3.6 Multilevel Feedback Queue Scheduling

Normally, in a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system .Processes do not move between queues. If there are separate queues foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but is inflexible.

Multilevel feedback queue scheduling, however, allows a process to move between queue; the idea is to separate processes with different CPU-burst characteristics. If a process uses so much of CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher priority queues. Similarly, a process that waits too long in a lower priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2, as shown in (Figure 2.7) the scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1; similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty.

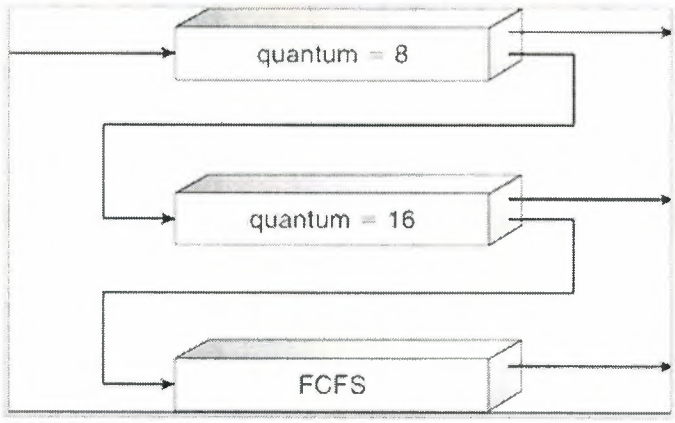


Figure 2.6 Multi-levels Feedback Queue scheduling

A process that arrives for queue 1 will preempt a process in queue 2; a process in queue 1 will in turn be preempted by a process arriving for queue 0.

A process entering the ready queue is put in queue 0; a process in queue 0 is given a time quantum of 8 units, if it does not finish within this time, it is moved to the tail of queue 1, if queue 0 is empty, the process at the head to queue 1 is given an quantum of 16 units, if it does not complete, It is preempted and is put into queue 2, processes in queue 2 are run on an FCFS basis, only when queue 0 and 1 are empty.

This scheduling algorithm gives high priority to any process with a CPU-burst of 8 units or less, such a process will quickly get the CPU, finish it's CPU-burst, and go off to its next I/O burst. Processes that need more than 8, but less than 24, units are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

In general, a multi-level feed back queue scheduler is defined by the following parameters:

- The number of queues.
- The scheduling algorithm of each queue.
- The method used to determine when to upgrade a process to higher-priority queue.
- The method used to determine when to demote a process to lower-priority queue.

- The method used to determine which queue a process will enter when that process needs service.

The definition of multi-level feedback queue scheduler makes it the most general CPU scheduling algorithm; it can be configured to match a specific system under design.

Unfortunately, it also requires sum means of selecting values for all the parameters to define the best scheduler, although a multi-level feed back queue is the most general scheme, it is also the most complex [8].

2.3.7 Longest Job First

In Shortest Job First algorithm, the CPU is assigned to the process that has the smallest burst time, In contrast, Longest Job First commits resources to longest jobs first, The LJF approach tends to maximize system utilization at the cost of turnaround time which is good in case of having huge number of processes.

In the event that two processes have the same length of necessary burst, then a First Come First Serve (FCFS) basis. For mostly long running jobs, longest job first (LJF) is beneficial, whilst shortest job first (SJF) is used with mostly short jobs. Hence, a single policy is not enough for an efficient resource management of systems [9]; consider the following set of processes, with the length of the CPU burst and arrival time given in units:

Process	Arrival Time	Burst time
P1	0.0	6
P2	2.0	4
P3	3.0	2
P4	4.0	1

Table 2.5 Processes Burst Time for LJF Scheduling Algorithm

The Gantt chart is:

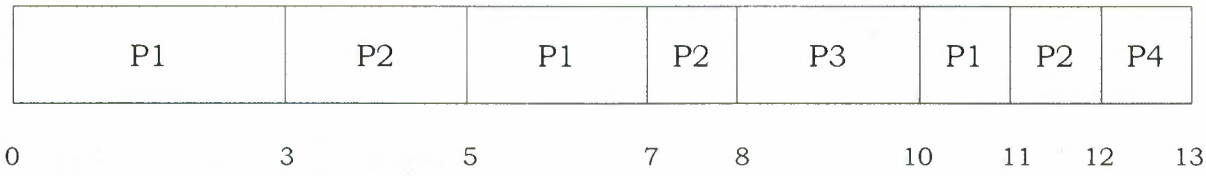


Figure 2.7 Longest Job First Execution Progress

The average waiting time = (5 + 6 + 5 + 8) / 4 = 6 units which is relatively long.

2.4 Comparison between Types of Scheduling Algorithms

To compare between the previous explained scheduling algorithms, the next example with same values of burst time for all processes will be executed using different types of scheduling algorithms, to notice the difference between turnaround times, thus the difference will be obvious[10].

Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

Table 2.6 Processes Burst Time for Scheduling Algorithms

The next figure shows the execution progress for different five types of scheduling algorithms.

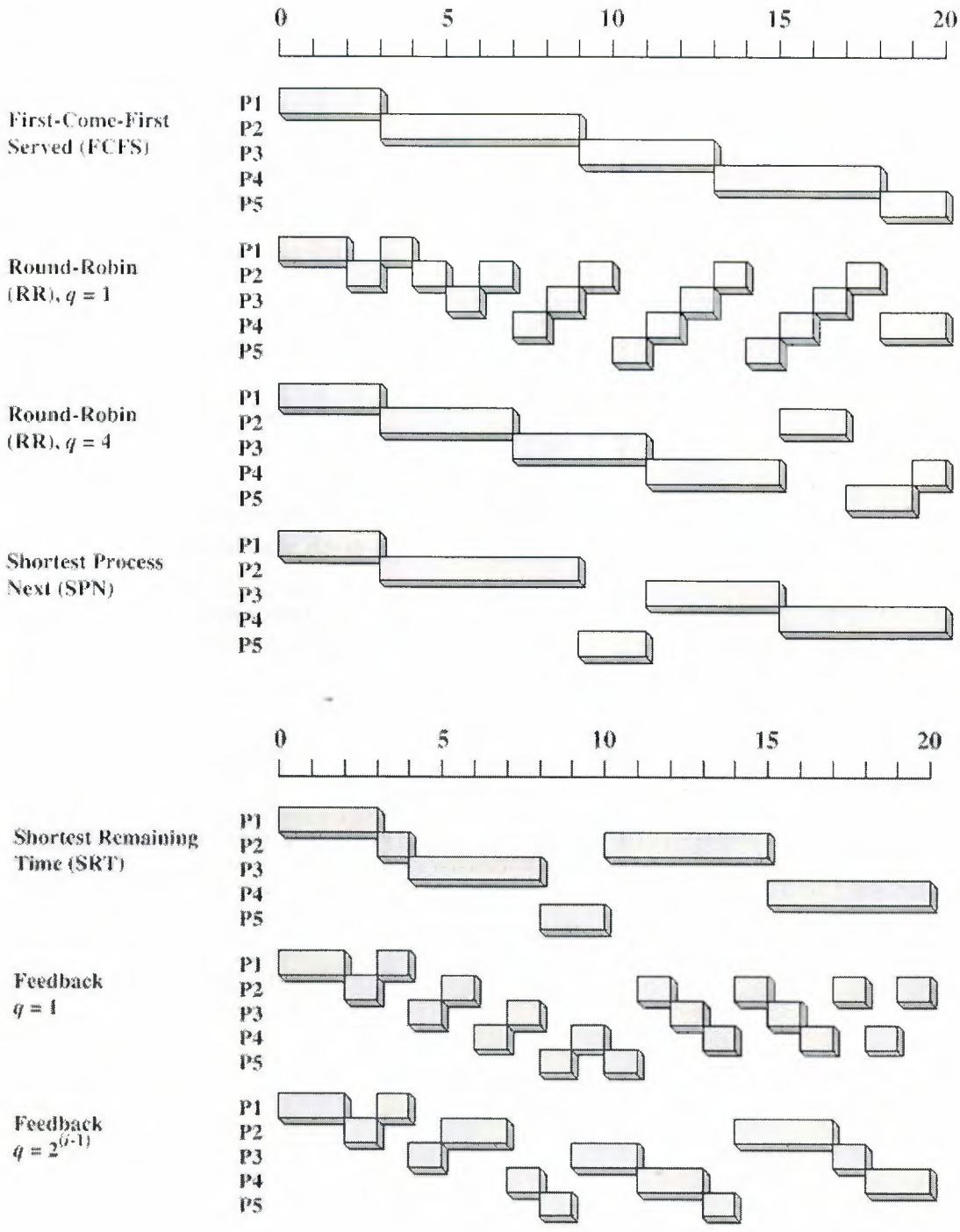


Figure 2.8 Scheduling Algorithms Comparison

	Process **Arrival Time** **Service Time**	1 0 3	2 2 6	3 4 4	4 6 5	5 8 2	**Mean**
FCFS	Turnaround Time Response Ratio	3 1.00	7 1.17	9 2.25	12 2.40	12 6.00	8.60 2.56
RR *q=1*	Turnaround Time Response Ratio	4 1.33	16 2.67	13 3.25	14 2.80	7 3.50	10.80 2.71
RR *q=4*	Turnaround Time Response Ratio	3 1.00	15 2.50	7 1.75	14 2.80	11 5.50	10.00 2.71
SJF	Turnaround Time Response Ratio	3 1.00	7 1.17	11 2.75	14 2.80	3 1.50	7.60 1.84
SRT	Turnaround Time Response Ratio	3 1.00	13 2.17	4 1.00	14 2.80	2 1.00	7.20 1.59
FB *q=1*	Turnaround Time Response Ratio	4 1.33	18 3.00	12 3.00	14 2.80	3 1.50	10.00 2.29
FB *q=2*	Turnaround Time Response Ratio	4 1.33	15 2.50	14 3.50	14 2.80	6 3.00	10.60 2.63

Table 2.7 Comparison between Scheduling Algorithms

The previous table gives the values of turnaround time, and response ratio for execution progress in different types of scheduling algorithms; it includes First Come First Served, Shortest Job First, Shortest Remaining Time First, Round Robin and Multi-level Feed back with two different quantum.

CHAPTER THREE

REAL TIME SYSTEMS SCHEDULING

3.1 Overview

Timeliness is the single most important aspect of a real-time system. These systems respond to a series of external inputs, which arrive in an unpredictable fashion. The real-time systems process these inputs, take appropriate decisions and also generate output necessary to control the peripherals connected to them. As defined by Donald Gilles "A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time in which the result is produced. If the timing constraints are not met, system failure is said to have occurred."

It is essential that the timing constraints of the system are guaranteed to be met. Guaranteeing timing behavior requires that the system be predictable.

The design of a real-time system must specify the timing requirements of the system and ensure that the system performance is both correct and timely. There are three types of time constraints:

- **Hard**

A late response is incorrect and implies a system failure. An example of such a system is of medical equipment monitoring vital functions of a human body, where a late response would be considered as a failure.

- **Soft**

Timeliness requirements are defined by using an average response time. If a single computation is late, it is not usually significant, although repeated late computation can result in system failures. An example of such a system includes airlines reservation systems.

- **Firm**

This is a combination of both hard and soft timeliness requirements. The computation has a shorter soft requirement and a longer hard requirement.

For example, a patient ventilator must mechanically ventilate the patient a certain amount in a given time period. A few seconds' delay in the initiation of breath is allowed, but not more than that.

One need is to distinguish between on-line systems, such as an airline reservation system, which operates in real-time but with much less severe timeliness constraints than, say a missile control system or a telephone switch. An interactive system with better response time is not a real-time system. These types of systems are often referred to as soft real time systems. In a soft real-time system (such as the airline reservation system) late data is still good data. However, for hard real-time systems, late data is bad data. Most real-time systems interface with and control hardware directly.

The software for such systems is mostly custom-developed. Real-time Applications can be either embedded applications or non-embedded (desktop) applications. Real-time systems often do not have standard peripherals associated with a desktop computer, namely the keyboard, mouse or conventional display monitors. In most instances, real-time systems have a customized version of these devices.
The following table compares some of the key features of real-time software systems with other conventional software systems [11].

Table 3.1 Key Feature of Real Time Systems

Feature	Sequential Programming	Concurrent Programming	Real-time Programming
Execution	Predetermined order	Multiple sequential programs executing in parallel	Usually composed of concurrent programs
Numeric Results	Independent of program execution speed	Generally dependent on program execution speed	Dependent on program execution speed
Examples	Accounting, payroll	UNIX operating system	Air flight Controller

3.2 Description of real-time scheduling

Real-time scheduling algorithms have been an active topic of research since the late 1960's. Real-time scheduling algorithms work either dynamically or statically. In static scheduling, all tasks are periodic and the periods are known. A periodic task is a task that repeats a request for the processor at a rate equal to its period.

A static scheduler runs ahead of time on a separate computer than the target system and specifies the order in which all tasks are to be executed. In a correct implementation of the scheduler, all hard real-time deadlines will be met or the algorithm returns no schedule.
As you might expect, unless all information about all tasks is known ahead of time, the system may not use a static scheduling algorithm.

Because static scheduling algorithms can be run off-line, the run time of the scheduler is generally not an issue. The scheduler is not taking processor time away from the tasks of the application. In practice, static scheduling algorithms have greater complexity.

With dynamic scheduling, both periodic and aperiodic tasks may be run. A dynamic scheduler is part of the system software for the computer and controls when and for how long tasks execute. A dynamic scheduling algorithm is sometimes referred to as a dispatching rule. The dynamic scheduling algorithms can further be divided into static priority dynamic and dynamic priority dynamic.

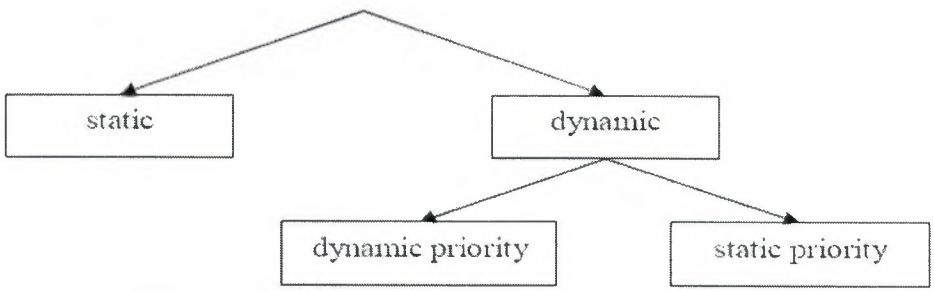


Figure 3.1 Classifications of Real-time Scheduling Algorithms

Static priority dynamic refers to the fact that at any point in the system's schedule, the priority of two tasks in relation to one another is fixed (it is sometimes referred to as fixed priority dynamic).

A static priority dynamic scheduler for two tasks A and B that at one point gives priority to A over B must always give priority to A over B. An example of a static priority dynamic scheduling algorithm is the Rate Monotonic algorithm (RM). An RM scheduler prioritizes the tasks by increasing period. The RM is the optimal static priority dynamic real-time scheduling algorithm for periodic task sets.

"Any periodic task set of any size will be able to meet all deadlines all of the time if the rate monotonic algorithm is used and the total (processor) utilization is not greater than 0.693". The application of the RM scheduling algorithm results have proved to be effective when used in combination with other techniques for scheduling periodic and periodic task loads as well.

A dynamic priority dynamic scheduler for two tasks A and B may at one point give preference to B and later give preference to A. Some examples of dynamic priority dynamic scheduling algorithms are the Earliest Deadline First (EDF) and First-In First-Out (FIFO). In dynamic scheduling practice, techniques that combine several simple heuristic algorithms may also be used.

3.2.1 Static Scheduling

This involves analyzing the tasks statically and determining their timing properties. These timing properties can be used to create a fixed scheduling table, according to which tasks will be dispatched for execution at run-time. Thus, the order of execution of tasks is fixed, and it is assumed that their execution times are also fixed.

3.2.1.1 Round Robin by Time Slicing

Round Robin by time slicing is one of the ways to achieve static scheduling. Round robin is one of the simplest and most widely used scheduling algorithms, in which a small unit of

time (called a time-slice) is defined. The CPU scheduler goes around a queue of ready-to-run processes and allocates a time slice to each such process.

3.2.1.2 Scheduling with priorities

Priority indicates the urgency or importance assigned to a task. The following are the two approaches followed when scheduling is based on priority.

3.2.1.3 Priority Based Execution

When the processor is idle, the ready task with the highest priority is chosen for execution; once chosen, a task is run to completion.

3.2.1.4 Preemptive Priority-based Execution

When the processor is idle, the ready task with the highest priority is chosen for execution; at any time, the execution of a task can be preempted if a task of higher priority becomes ready; Thus, at all times, the processor is idle or executing the ready task with the highest priority. This approach is followed in the SOS operating system.

3.2.2 Dynamic Scheduling

Dynamic scheduling of a real-time program requires a sequence of decisions to be taken during execution on the assignment of resources to transactions.

Each decision must be taken without prior knowledge of the needs of future tasks. Dynamic algorithms are needed for applications where the computing requirements may vary widely, making fixed priority scheduling difficult or inefficient. This kind of scheduling allows flexibility to alter scheduling based on:

a. Environment changes
b. Burst of task arrival
c. Partial system failure.

Dynamic scheduling has the following basic steps.

3.2.3 Feasibility checking

Feasibility checking is the process of determining whether the timing requirements of a set of tasks can be satisfied, usually under a given set of resource requirements and precedence constraints.

There are two approaches to feasibility-checking in dynamic real-time systems: Dynamic planning-based approach: Here the feasibility of a set of tasks is checked in terms of a scheduling policy such as 'earliest-deadline-first' or 'least-laxity-first'.

Dynamic best-effort approach: Tasks may be queued according to policies that take account of time constraints. No feasibility checking is done before the tasks are queued.

3.2.4 Schedule Construction

Schedule construction is the process of ordering the tasks to be executed and storing this in a form that can be used by the dispatching step.

3.2.5 Scheduling Overheads

Simple scheduling analysis usually ignores context switches and queue manipulations, but the time for this is often significant and cannot realistically be assumed to be negligible. The time taken should be considered when deciding on the scheduling algorithm [12].

3.2.6 Real-time Systems and COTS

Use of Commercially available Off-The-Shelf (COTS) software in the design of real-time embedded systems is prevalent these days as this helps the companies meet the aggressive time-to-market requirements within the given schedule and budget. As most of these COTS packages are proven in the industrial applications, the risk is also reduced significantly.

Selection of the right product (and vendor) is critical for the success of any COTS-based project. The financial stability of the vendor is definitely one of the key parameters to be looked into before deciding on a COTS package offered by the vendor.

In addition to this, the geographical spread, availability of local support for a long term, and responsiveness of the vendor in addressing the reported problems must be taken into account.

It may be worthwhile to conduct a detailed evaluation of the product to determine its applicability in meeting the requirements. This evaluation should also take into account the compatibility issues at a broader level with other software packages under consideration for the product under development. Having historical data on the integration of various packages is definitely helpful.

This data can normally be received from other projects that have already done such an integration exercise or a similar one. One more important aspect to be considered while making the selection of the RTOS is the availability of tools (for example, code coverage, memory leak detection, etc.).

The COTS package selection must show substantial savings in the costs and should be weighed against the risk of losing control (by using the COTS software). The evaluation process must take factors other than the financial and technical into account while making the selection of the COTS package.

3.2.7 Programming Language

The choice of programming language is very important for real-time embedded software. The following factors influence the choice of language:

- A language compiler should be available for the chosen RTOS and hardware architecture of the embedded system.

Compilers should be available on multiple Operating systems and micro processors. This is particularly important if the processor or the RTOS needs to be changed in future.

- The language should allow direct hardware control without sacrificing the advantages of a high-level language.

The language should provide memory management control such as dynamic and static memory allocation.

- Real-time systems are increasingly being designed using Object Oriented (OO) methodology and using a language that supports OO concepts is definitely helpful. The languages that are typically used for embedded systems are Assembly Language, C, C++, Ada and Java.

Choosing to write code in Assembler should be done on a case-by-case basis. While Code written in Assembler can be much faster, it is usually very processor-specific and less portable than a high-level language.
C is by far the most popular language and the language that maximizes application portability. C++ is used when real-time applications are developed using object-oriented methodology.

Members of the scientific community, for example, have millions of lines of existing FORTRAN code that implement proprietary numerical algorithms. Likewise, developers targeting military applications may require Ada.
The availability of languages for a development project is limited to the languages that have been ported to a specific RTOS environment, and in some cases, to languages that have been ported to a specific target single-board computer [13].

Some RTOS vendors offer their own versions of popular languages as they optimize the compiler for RTOS architecture. Microware, for example, has its own C compiler called Ultra C.

CHAPTER FOUR
SCHEDULING IN LINUX OS

4.1 Overview

This chapter describes what a process is and how the Linux kernel creates, manages and deletes the processes in the system. Processes carry out tasks within the operating system. A program is a set of machine code instructions and data stored in an executable image on disk and is, as such, a passive entity; a process can be thought of as a computer program in action. It is a dynamic entity, constantly changing as the machine code instructions are executed by the processor. As well as the program's instructions and data, the process also includes the program counter and all of the CPU's registers as well as the process stacks containing temporary data such as routine parameters, return addresses and saved variables.

The current executing program, or process, includes all of the current activity in the microprocessor. Linux is a multiprocessing operating system. Processes are separate tasks each with their own rights and responsibilities. If one process crashes it will not cause another process in the system to crash. Each individual process runs in its own virtual address space and is not capable of interacting with another process except through secure, kernel managed mechanisms.

During the lifetime of a process it will use many system resources. It will use the CPUs in the system to run its instructions and the system's physical memory to hold it and its data. It will open and use files within the file systems and may directly or indirectly use the physical devices in the system. Linux must keep track of the process itself and of the system resources that it has so that it can manage it and the other processes in the system fairly. It would not be fair to the other processes in the system if one process monopolized most of the system's physical memory or its CPUs.

The most precious resource in the system is the CPU, usually there is only one Linux is a multiprocessing operating system; its objective is to have a process running on each CPU in the system at all times, to maximize CPU utilization.

If there are more processes than CPUs (and there usually are), the rest of the processes must wait before a CPU becomes free until they can be run. Multiprocessing is a simple idea; a process is executed until it must wait, usually for some system resource; when it has this resource, it may run again.

In a uni-processing system, for example DOS, the CPU would simply sit idle and the waiting time would be wasted. In a multiprocessing system many processes are kept in memory at the same time. Whenever a process has to wait the operating system takes the CPU away from that process and gives it to another, more deserving process. It is the scheduler which chooses which is the most appropriate process to run next and Linux uses a number of scheduling strategies to ensure fairness. Linux supports a number of different executable file formats, ELF is one, Java is another and these must be managed transparently as must the processes use of the system's shared libraries [14], according to the next figure it shows the main components of the Linux kernel which shows how process scheduler intercommunicate with other pars of the Linux kernel.

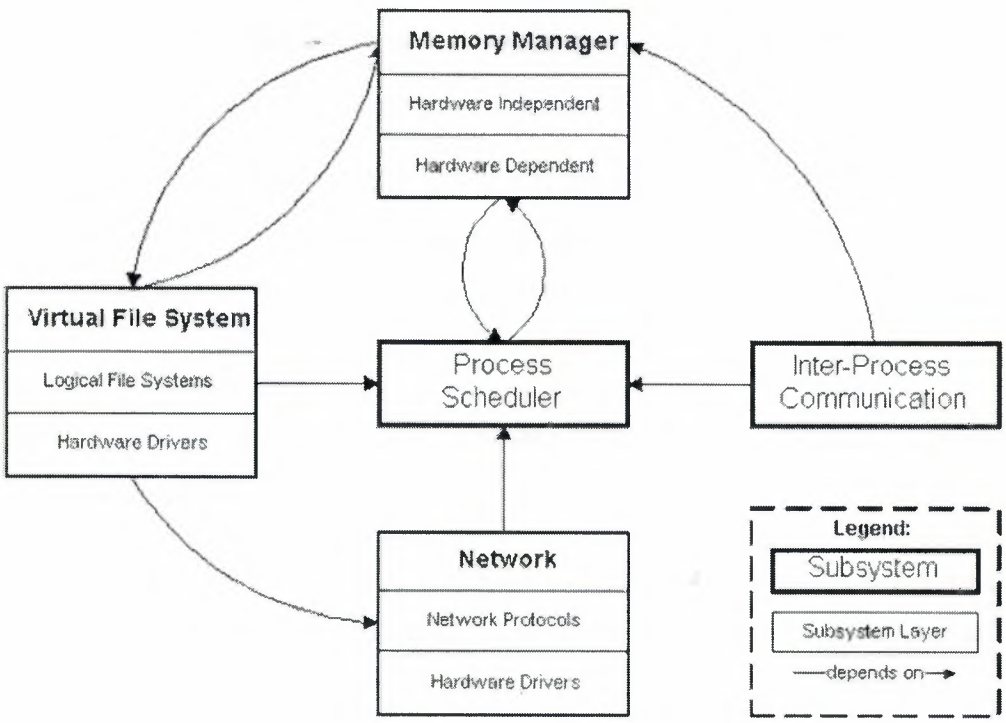


Figure 4.1 Linux Scheduler Overview.

4.2 Scheduling

All processes run partially in user mode and partially in system mode. How these modes are supported by the underlying hardware differs but generally there is a secure mechanism for getting from user mode into system mode and back again. User mode has far less privileges than system mode. Each time a process makes a system call it swaps from user mode to system mode and continues executing. At this point the kernel is executing on behalf of the process.

In Linux, processes do not preempt the current, running process; they cannot stop it from running so that they can run. Each process decides to relinquish the CPU that it is running on when it has to wait for some system event. For example, a process may have to wait for a character to be read from a file. This waiting happens within the system call, in system mode; the process used a library function to open and read the file and it, in turn made system calls to read bytes from the open file. In this case the waiting process will be suspended and another, more deserving process will be chosen to run. Processes are always making system calls and so may often need to wait. Even so, if a process executes until it waits then it still might use a disproportionate amount of CPU time and so Linux uses pre-emptive scheduling.

In this scheme, each process is allowed to run for a small amount of time, 200ms, and, when this time has expired another process is selected to run and the original process is made to wait for a little while until it can run again. This small amount of time is known as a time-slice.

It is the scheduler that must select the most deserving process to run out of all of the run-able processes in the system.

A run-able process is one which is waiting only for a CPU to run on. Linux uses a reasonably simple priority based scheduling algorithm to choose between the current processes in the system.

When it has chosen a new process to run it saves the state of the current process, the processor specific registers and other context being saved in the processes task_struct data structure. It then restores the state of the new process (again this is processor specific) to run and gives control of the system to that process. For the scheduler to fairly allocate CPU time between the run-able processes in the system it keeps information in the task_struct for each process:

4.2.1 Policy

This is the scheduling policy that will be applied to this process. There are two types of Linux process, normal and real time. Real time processes have a higher priority than all of the other processes. If there is a real time process ready to run, it will always run first. Real time processes may have two types of policy, round robin and first in first out. In round robin scheduling, each run-able real time process is run in turn and in first in, first out scheduling each run-able process is run in the order that it is in on the run queue and that order is never changed.

4.2.2 Priority

This is the priority that the scheduler will give to this process. It is also the amount of time (in jiffies) that this process will run for when it is allowed to run. It is possible to alter the priority of a process by means of system calls and the renice command.

4.2.3 Rt_priority

Linux supports real time processes and these are scheduled to have a higher priority than all of the other non-real time processes in system. This field allows the scheduler to give each real time process a relative priority. The priority of a real time processes can be altered using system calls.

4.2.4 Counter

This is the amount of time (in jiffies) that this process is allowed to run for. It is set to priority when the process is first run and is decremented each clock tick.

The scheduler is run from several places within the kernel. It is run after putting the current process onto a wait queue and it may also be run at the end of a system call, just before a process is returned to process mode from system mode. One reason that it might need to run is because the system timer has just set the current processes counter to zero.

4.3 Current process

The current process must be processed before another process can be selected to run.

If the scheduling policy of the current processes is round robin then it is put onto the back of the run queue.

If the task is INTERRUPTIBLE and it has received a signal since the last time it was scheduled then its state becomes RUNNING.

If the current process has timed out, then its state becomes RUNNING.

If the current process is RUNNING then it will remain in that state.

Processes that were neither RUNNING nor INTERRUPTIBLE are removed from the run queue. This means that they will not be considered for running when the scheduler looks for the most deserving process to run.

4.4 Process selection

The scheduler looks through the processes on the run queue looking for the most deserving process to run. If there are any real time processes (those with a real time scheduling policy) then those will get a higher weighting than ordinary processes. The weight for a normal process is its counter but for a real time process it is counter plus 1000. This means that if there are any run-able real time processes in the system then these will always be run before any normal run-able processes. The current process, which has consumed some of its time-slice (its counter has been decremented) is at a disadvantage if there are other processes with equal priority in the system; that is as it should be.

If several processes have the same priority, the one nearest the front of the run queue is chosen. The current process will get put onto the back of the run queue.

In a balanced system with many processes of the same priority, each one will run in turn. This is known as Round Robin scheduling. However, as processes wait for resources, their run order tends to get moved around.

4.5 Swap Processes

If the most deserving process to run is not the current process, then the current process must be suspended and the new one made to run. When a process is running it is using the registers and physical memory of the CPU and of the system. Each time it calls a routine it passes its arguments in registers and may stack saved values such as the address to return to in the calling routine. So, when the scheduler is running it is running in the context of the current process. It will be in a privileged mode, kernel mode, but it is still the current process that is running.

When that process comes to be suspended, all of its machine state, including the program counter (PC) and all of the processor's registers, must be saved in the processes task_struct data structure.

Then, all of the machine state for the new process must be loaded. This is a system dependent operation, no CPUs do this in quite the same way but there is usually some hardware assistance for this act.

This swapping of process context takes place at the end of the scheduler. The saved context for the previous process is, therefore, a snapshot of the hardware context of the system as it was for this process at the end of the scheduler. Equally, when the context of the new process is loaded, it too will be a snapshot of the way things were at the end of the scheduler, including this processes program counter and register contents [15].

If the previous process or the new current process uses virtual memory then the system's page table entries may need to be updated. Again, this action is architecture specific. Processors like the Alpha AXP, which use Translation Look-aside Tables or cached Page Table Entries, must flush those cached table entries that belonged to the previous process.

4.6 Linux Scheduling Goals

4.6.1 Linux's Target Market(s) And Their Effects on its Scheduler

An operating system's scheduling algorithm is largely determined by its target market, and vice-versa. Understanding an operating system's target market helps to explain its scheduling goals, and thus its scheduling algorithm. Linux was originally created by Linus Torvalds for use on his personal computer. However, despite its origins, Linux has become known as a server operating system. There are many reasons for this, not the least of which is the fact that most software designed to run on top of the Linux kernel is meant for users with a relatively high skill level or inherits design qualities targeting more skilled users.

This led to Linux's notoriously complex and unrefined graphical user interface options (compared to Apple rand Microsoft rope rating systems) and subsequent relegation to the server room. Linux's exposure in the server market guided its development along the lines of the one market that it initially succeeded in. Linux's prowess as a server operating system is nowadays perhaps matched only by a few operating systems such as Sun's Solaris and IBM's AIX.

However, cost and legal advantages are causing many companies to replace both of those operating systems with Linux as well; while Linux has made a name for itself in the server operating systems arena, many users and developers believe that it can also be a success on the desktop. In the last several years, there has been a push to optimize the Linux kernel for the desktop market. Perhaps the biggest step in that direction was the scheduler written by Ingo Molnar for the 2.6.x kernel series.

Molnar designed his scheduler with the desktop and the server market in mind, and as a result desktop performance is much improved in Linux distributions based on 2.6.x kernels. Targeting both the server and the desktop market imposes particularly heavy demands on the kernel's scheduler, and thus the Linux kernel's scheduler is an interesting case study in how to please two very different markets at the same time.

4.6.2 Efficiency

An important goal for the Linux scheduler is efficiency. This means that it must try to allow as much real work as possible to be done while staying within the restraints of other requirements. For example - since context switching is expensive, allowing tasks to run for longer periods of time increases efficiency.

Also, since the scheduler's code is run quite often, its own speed is an important factor in scheduling efficiency. The code making scheduling decisions should run as quickly and efficiently as possible. Efficiency suffers for the sake of other goals such as interactivity, because interactivity essentially means having more frequent context switches. However, once all other requirements have been met, overall efficiency is the most important goal for the scheduler.

4.6.3 Interactivity

Interactivity is an important goal for the Linux scheduler, especially given the growing effort to optimize Linux for desktop environments. Interactivity often flies in the face of efficiency, but it is very important nonetheless.

An example of interactivity might be a keystroke or mouse click. Such events usually require a quick response (i.e. the thread handling them should be allowed to execute very soon) because users will probably notice and be annoyed if they do not see some result from their action almost immediately. Users don't expect a quick response when, for example, they are compiling programs or rendering high-resolution images.

They are unlikely to notice if something like compiling the Linux kernel takes an extra twenty seconds. Schedulers used for interactive computing should be designed in such a way that they respond to user interaction within a certain time period. Ideally, this should be a time period that is imperceptible to users and thus gives the impression of an immediate response.

4.6.4 Fairness and Preventing Starvation

It is important for tasks to be treated with a certain degree of fairness, including the stipulation that no thread ever starves.

Starvation happens when a thread is not allowed to run for an unacceptably long period of time due to the prioritization of other threads over it. Starvation must not be allowed to happen, though certain threads should be allowed to have a considerably higher priority level than others based on user-defined values and/or heuristic indicators.

Somehow, threads that are approaching the starvation threshold (which is generally defined by a scheduler's implementers) must get a significant priority boost or one-time immediate preemption before they starve. Fairness does not mean that every thread should have the same degree of access to CPU time with the same priority, but it means that no thread should ever starve or be able to trick the scheduler into giving it a higher priority or more CPU time than it ought to have.

4.7 Linux Soft Real-Time Scheduling

The Linux scheduler supports soft real-time (RT) scheduling. This means that it can effectively schedule tasks that have strict timing requirements.

However, while the Linux 2.6.x kernel is usually capable of meeting very strict RT scheduling deadlines, it does not guarantee that deadlines will be met.

RT tasks are assigned special scheduling modes and the scheduler gives them priority over any other task on the system.

RT scheduling modes include a first-in-first-out (FIFO) mode which allows RT tasks to run to completion on a first-come-first served basis, and a round-robin scheduling mode that schedules RT tasks in a round-robin fashion while essentially ignoring non-RT tasks on the system.

4.8 Scheduling Performance Perspectives

In terms of schedulers, there is no single definition of performance that fits everyone's needs; that is, there is not a single performance goal for the Linux scheduler to strive for. The many definitions of good scheduling performance often lead to a give-and-take situation, such that improving performance in one sense hurts performance in another.
Some improvements to the Linux scheduler help performance all-around, but such improvements are getting more and harder to come by.
A good example of a give-and-take performance issue is desktop vs. server vs. high performance computing (HPC) performance.

The most important performance metric for desktop users is perceived performance that is, how fast does a machine seem to respond to requests such as mouse clicks and key presses. If a user is compiling a kernel in the background and typing in a word processor in the foreground, he or she is unlikely to notice if the kernel compile takes an extra minute because it is constantly interrupted by the word processor responding to keystrokes. What matters most to the users is that when he or she presses a key, the word processor inserts and displays the desired character as quickly as possible.

This entails a CPU making a context switch to the word processor's thread as soon as possible after the user presses a key. In order for this to happen, the currently running thread must either give up the processor before its time slice is up, or its time slice must be short enough that the delay between the time the keystroke happens and the time slice ends is imperceptible to the user.

Since context switching is expensive, context switches must be minimized while happening frequently enough to provide good perceived performance to interactive users (e.g. word processors).

Fewer context switches means better real efficiency, since more time is spent doing
Actual work and less is spent switching tasks. More context switches means the system is more responsive to user input.

On interactive desktop systems, the desired behavior is to have context switching happen often enough that user input seems to get an immediate response without happening so often that the machine becomes very inefficient.

Server systems generally focus less on perceived performance than desktop systems. They are relatively more concerned with actual performance; that is, reducing the overall amount of time it takes to complete a set of tasks.

Since users are normally willing to put up with a longer response delay (e.g. they are willing to wait longer for a web page to be transmitted over the network than they are for a keystroke to cause a character to appear in a word processing document), more of an emphasis is placed on overall efficiency via fewer context switches.

If three complex database queries on a database loaded into memory happen at the same time, it is most likely better to get them done faster overall than it is to do them inefficiently for the sake of returning results at the same time and thus lowering the average response time. People and applications submitting complex database queries generally have much lower response time expectations than people who are typing characters into a word processor.

However, if, for example, two massive files are requested from an FTP server, it would be unacceptable for the server to completely finish sending one file before beginning to send the other (the most extreme but perhaps overall most efficient case, potential I/O concerns aside). Thus server systems, while having lower response time requirements than desktop systems, are still expected to operate within some responsiveness expectations.
HPC systems generally require the least immediate response times as they tackle very large problems that can take days to solve. Given a set of tasks, overall efficiency is the imperative and this means that context switches for the sake of responsiveness must be minimized (or perhaps all but done away with?).

Response time expectations are generally the lowest for HPC applications, and thus they represent the true opposite of desktop computing performance ideals.

Servers tend to be somewhere in the middle, this comparison illustrates the point that there is no universal ideal for scheduler performance. A scheduler that seems superb to a desktop user might be a nightmare for someone running HPC applications.

The Linux scheduler strives to perform as well as possible in all types of situations, though it is impossible for it to perform ideally for everyone. Desktop users are constantly crying out for more tuning for their needs while at the same time HPC users are pushing for optimization towards their performance ideal [16].

CHAPTER FIVE
SIMULATION TOOL OF PROCESS MANAGER

5.1 Overview

Scheduling processes in any kind of the scheduling algorithms depends upon many factors such as burst time, turn-around time, arrival time, and priority; according to the used algorithm anyone of the previous factors.

CPU scheduling is the basis of multi-programmed operating systems; by switching the CPU among processes, the operating system can make computer more productive.
Scheduling is a fundamental operating-system function. Almost all computer resources are scheduled before use; the CPU is one of the primary computer resources, thus, its scheduling is central to operating-system design.

This chapter presents visualization to seven kinds of scheduling algorithms, First Come First Served (FCFS), Shortest Job First (SJF), Priority Scheduling, Round Robin (RR), Multi-Queue, Multi Level Feedback Queue, and Longest Job First.

5.2 Programming Language

It was considered to choose flexible programming language to implement this project, thus Visual C++ is one of the most suitable programming languages to be used for such task.

The Visual C++ user interface consists of an integrated set of windows, tools, menus, toolbars, directories, and other elements that allow to create, test, and refine application in one place. It is also possible to work on other types of documents within Visual C++, for example Microsoft Excel or Microsoft Word documents.

The user interface uses standard Windows interface functionality along with a few additional features to make development environment easy to use. The basic features that used most often are windows and document views, toolbars, menus, directories, and keyboard shortcuts.

Visual C++ is flexible that it is possible to be customized to suit preferences. Among other customizations, it's easy to establish a layout for the windows associated with a particular project workspace. It enables the user also to create custom toolbars, menus, and shortcut keys [17], the code of all scheduling algorithm with flow charts are shown in the appendices.

5.3 User Interface

The interface is designed to cover all of the user demands, this window consists of many options distributed all over the window, the first row of options contains seven options to choose one of the seven scheduling algorithms, the order of the scheduling algorithms as shown in figure 5.1 is ordered from left to right, First Come First Served, Shortest Job First, Priority Scheduling, Round Robin, Multi-Queue, Multi-Level Feed, and Longest Job First.

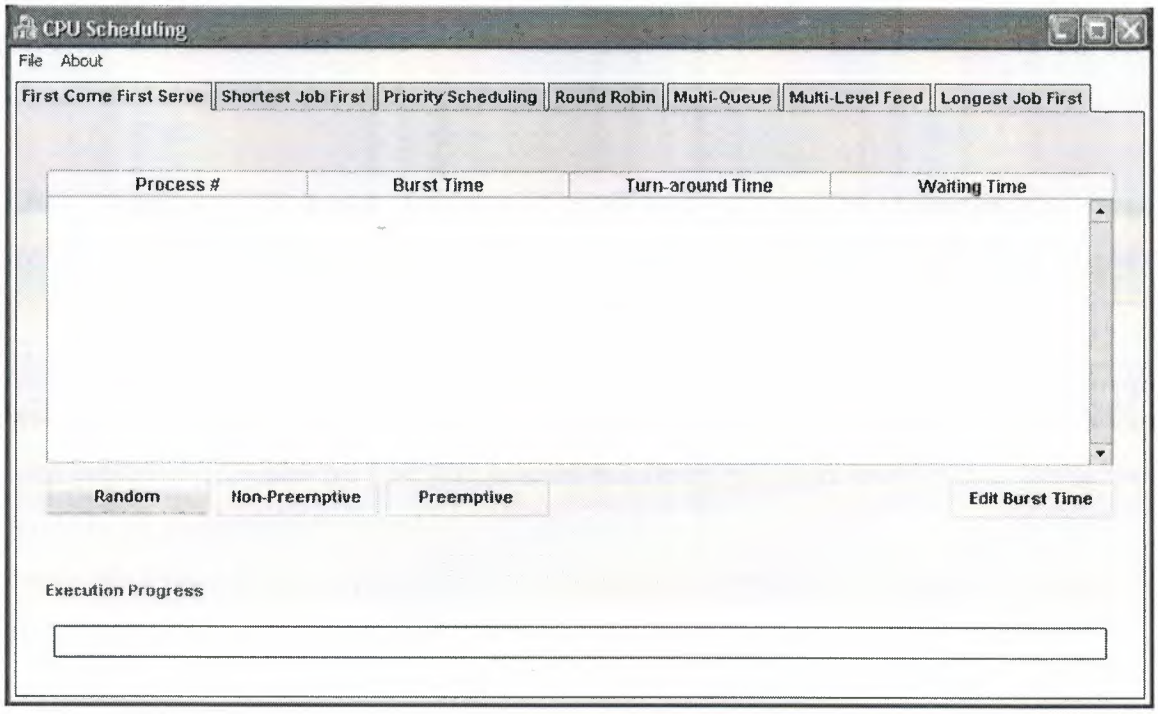


Figure 5.1 Scheduling Program Interface

By clicking on the button of the wanted scheduling algorithm's button, this type will become active thus corresponding buttons and variables will appear, the middle part of the

interface window includes four columns at least or more according the chosen algorithm, those columns specify the process number, burst time, Turn-around time, and waiting time. The bottom side of the window shows the execution bar which specifies the execution progress for each process until finish executing, all types of scheduling algorithms which are shown in this program will be explained in the next paragraphs with more details, specifications, and examples.

5.4 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, it must be considered the properties of the various algorithms.

Many criteria have been suggested for computing CPU scheduling algorithms. Which are used for comparison can make a substantial difference in the determination of the best algorithm. Criteria that are used include the following:

CPU utilization CPU should stay busy as much as possible. CPU utilization may range from 0 to 100 percent. In a real time 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 measure 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 creation 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 writing 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 process

executes or does I/O; it affects only the amount of time that process spends writing in the ready queue. Writing 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 creation.

Often, a process can produce some output fairly early and can be continue computing new results while previous results are begin output to the user. Thus, another measure is the time from the submissions of a request until the first response is produced.

The measure, called Response Time, is the amount of time it takes to start responding, but not the time that it takes to output that response. The turnaround time is generally limited by the speed of the output device [18].

5.5 The Processes Scheduling Tool

By using the scheduling program the user will be able to optimize the different types of scheduling algorithm, actually this programs contains seven types of scheduling algorithms, the explanation of the usage of this program will be specifies later in this chapter with all details about the outputs and the progress of the execution, the algorithms which are implemented in this program are:

5.5.1 First Come First Served Scheduling Algorithm

The user is able to enter the value of burst time either randomly or by edit burst time button, following the steps of the Flowchart (figure 5.2) to reach to the execution progress bar.

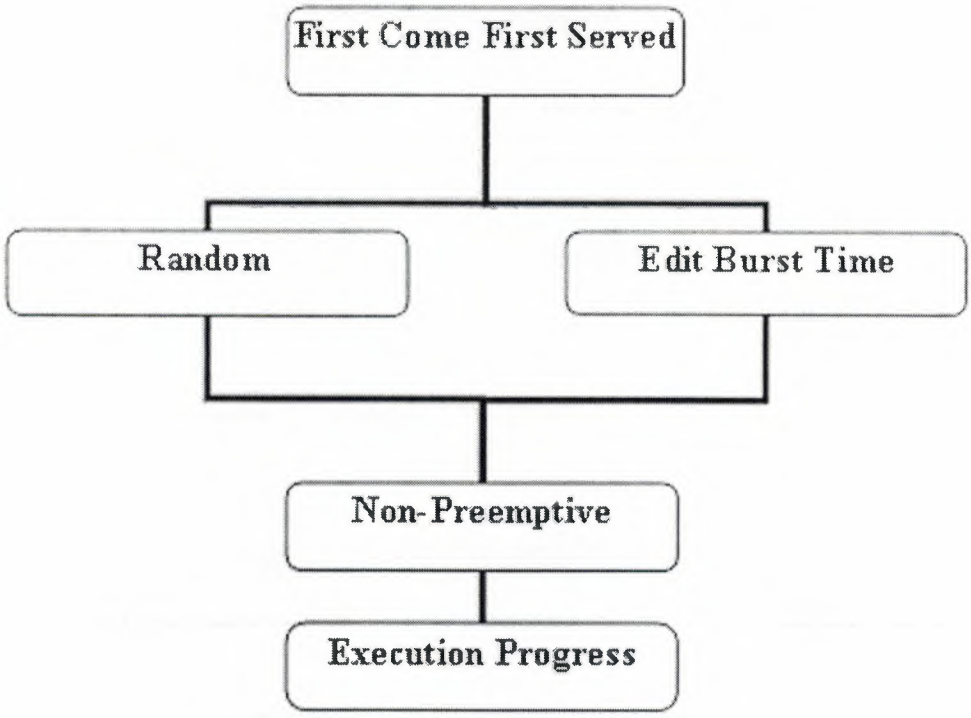


Figure 5.2 Flowchart of First Come First Served

As it is shown in the Flowchart the user will choose the First Come First Served scheduling algorithm, then it will be possible to make the program decide the values of burst time for the five processes randomly, or the user will edit these values by using edit burst time button, then to obtain the values of turn around time and average waiting time, and also to get the execution progress bar the user should press Non-preemptive button.

The next figure shows a typical interface of the program while it executes an example of FCFS scheduling algorithm.

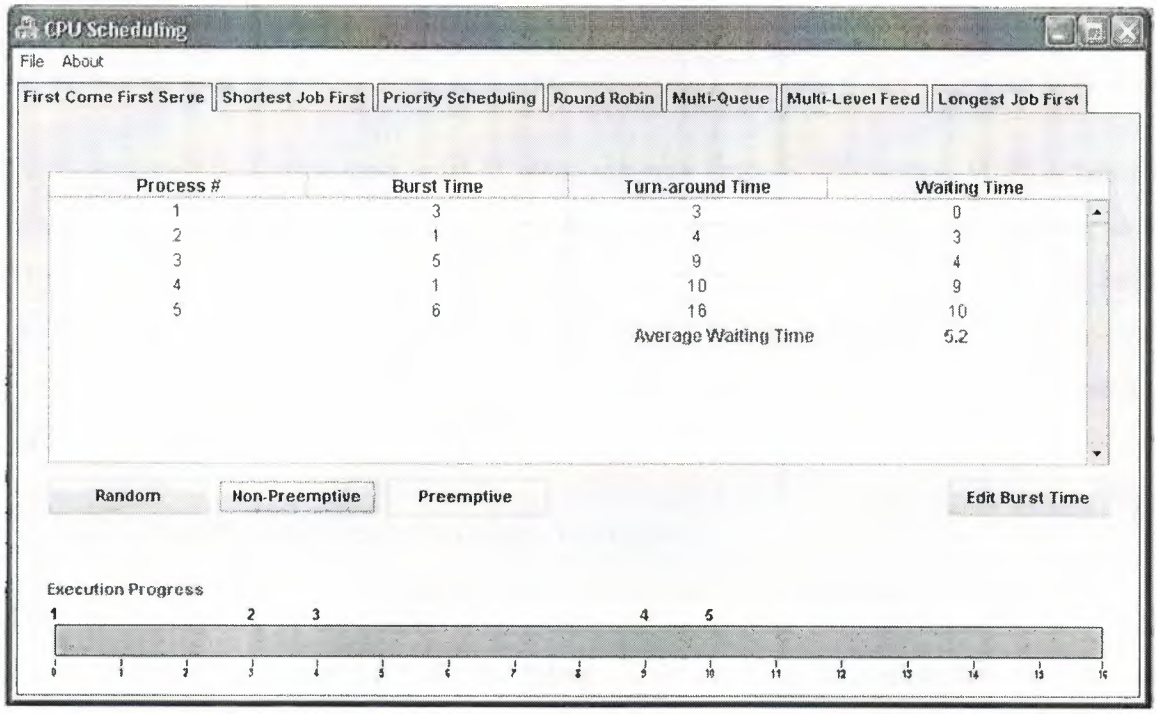


Figure 5.3 FCFS Scheduling Example

The previous figure implements an example of FCFS scheduling algorithm, the values of burst times of the processes could be chosen by edit burst time button then those values should be saved by save button.

The values in this example were chosen randomly by Random button those values are 3 units for P1, 1 unit for P2, 5 units for P3, 1 unit for P4, and 6 units for P5, process P1 will be assigned to the CPU at first that it has the first arrival time.

Then it will be followed by process P2, and then P3, P4, finally P5, all depends on the arrival time which specifies which process will be executed.

The average waiting time = (0 + 3 + 4 + 9 + 10) / 5 = 5.2 units.

The average turnaround time = (3 + 4 + 9 + 10 + 16) / 5 = 8.4 units.

The average response ratio = (3 + 4 + 9 + 10 + 16) / 16 = 2.625 units.

Processes are dispatched according to their arrival time on the ready queue. Being a non-preemptive discipline, once a process has a CPU, it runs to completion.

The FCFS scheduling is fair in the formal sense or human sense of fairness but it is unfair in the sense that long jobs make short jobs wait and unimportant jobs make important jobs wait.

FCFS is more predictable than most of other schemes since it offers time. FCFS scheme is not useful in scheduling interactive users because it cannot guarantee good response time. The code for FCFS scheduling is simple to write and understand. One of the major drawbacks of this scheme is that the average time is often quite long; the First-Come-First-Served algorithm is rarely used as a master scheme in modern operating systems but it is often embedded within other.

5.5.2 Shortest Job First Scheduling Algorithm

The SJF scheduling is especially appropriate for batch jobs for which the run times are known in advance. Since the SJF scheduling algorithm gives the minimum average time for a given set of processes, it is probably optimal, the SJF algorithm favors short jobs (or processors) at the expense of longer ones, the obvious problem with SJF scheme is that it requires precise knowledge of how long a job or process will run, and this information is not usually available.

The best SJF algorithm can do is to rely on user estimates of run times, in the production environment where the same jobs run regularly, it may be possible to provide reasonable estimate of run time, based on the past performance of the process. But in the development environment users rarely know how their program will execute.

Like FCFS, SJF is non preemptive therefore, it is not useful in timesharing environment in which reasonable response time must be guaranteed.

By following the next flowchart steps the user will be able to edit both the arrival times and burst times for all processes, also the burst times could be chosen randomly by the program.

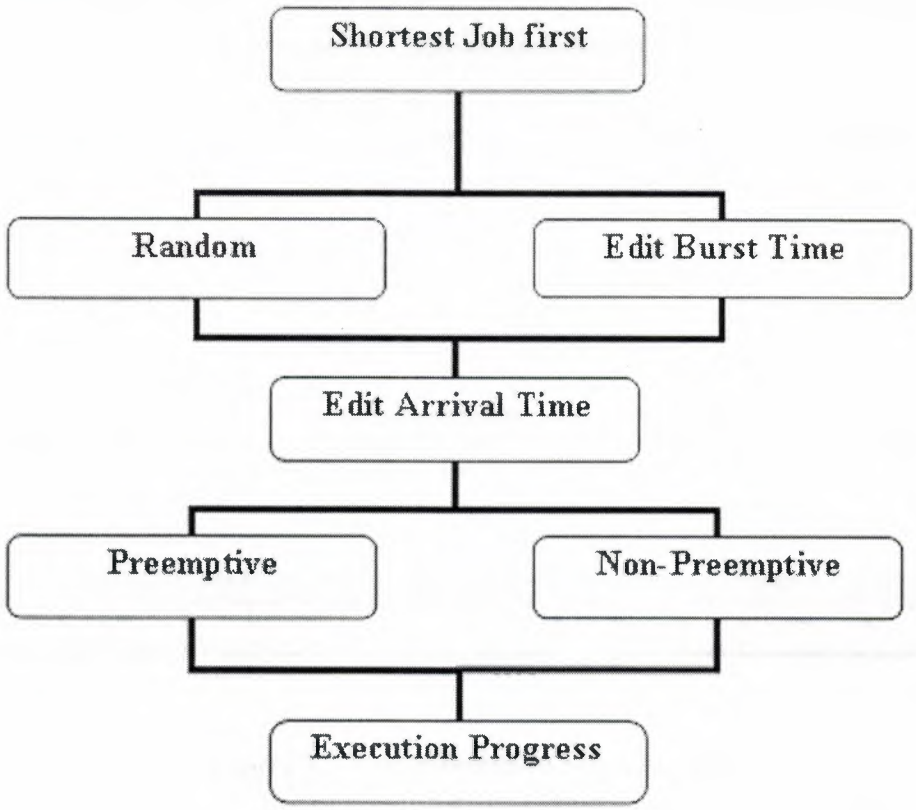


Figure 5.4 Shortest Job First Flowchart

After editing arrival times for processes, the choice will be to schedule the processes in preemptive or non-preemptive method, the result of this choice will give different values of turnaround times, waiting time, and also it will directly affect the shape of execution progress bar.

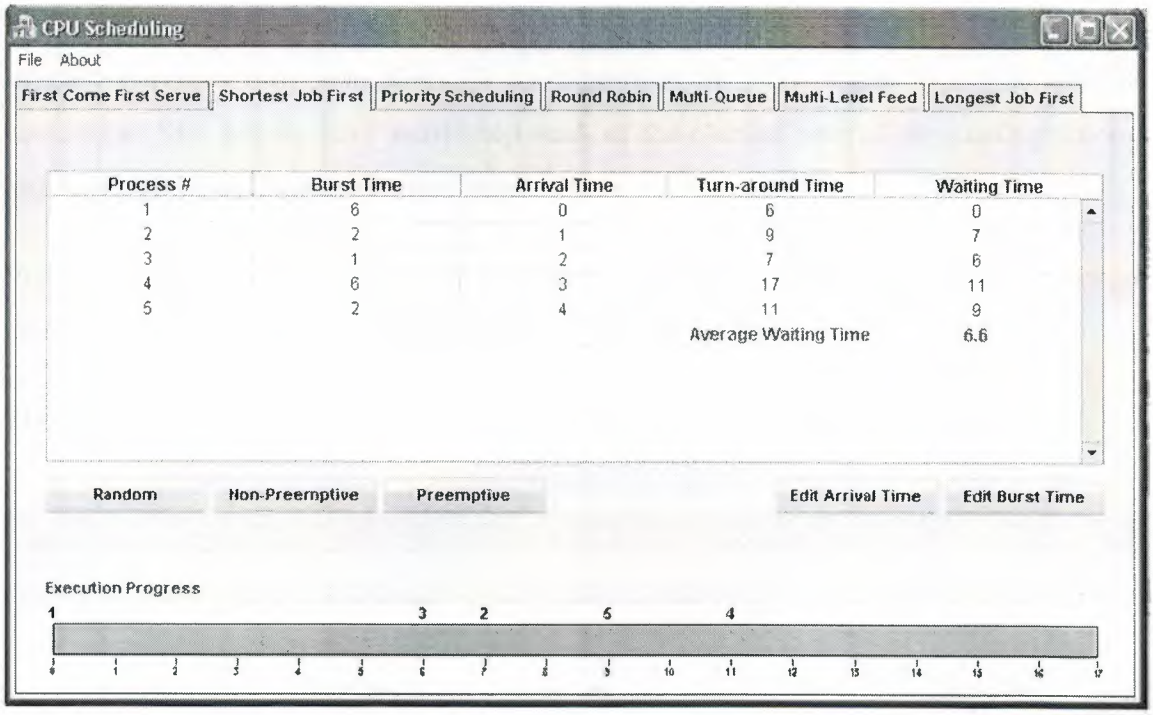


Figure 5.5 Non-preemptive SJF Interface

the values of burst times for the five processes as shown in figure 5.5 are 6 units for P1, 2 units for P2, 1 units for P3, 6 units for P4, and 2 units for P5.

First P1 will use the CPU because of its arrival time, then P3 that it has the shortest burst time, followed by P2 which has the next burst time, followed by P5 then P4 according to their burst times.

The average waiting time is (0 + 7 + 6 + 11 + 9)/ 5 = 6.6 units

The average turnaround time = (6 + 9 + 7 + 11 + 17) / 5 = 10 units.

The average response ratio = 50 / 17 = 2.94 units.

For Preemptive SJF the execution will be totally different, also preemptive SJF could be called Shortest Remaining Time First, The Shortest Remaining Time is the preemptive counter part of SJF and useful in time-sharing environment, in SRT scheduling, the process with the smallest estimated run-time to completion is run next, including new arrivals, but in SJF once a job begin executing, it run to completion.

In SJF scheme also, a running process may be preempted by a new arrival process with shortest estimated run-time, although the algorithm SRT has higher overhead than its counterpart SJF, but the SRT must keep track of the elapsed time of the running process and must handle occasional preemptions.

In this scheme, arrival of small processes will run almost immediately. However, longer jobs have even longer mean waiting time

The same values for burst times and arrival times will be used in the following figure

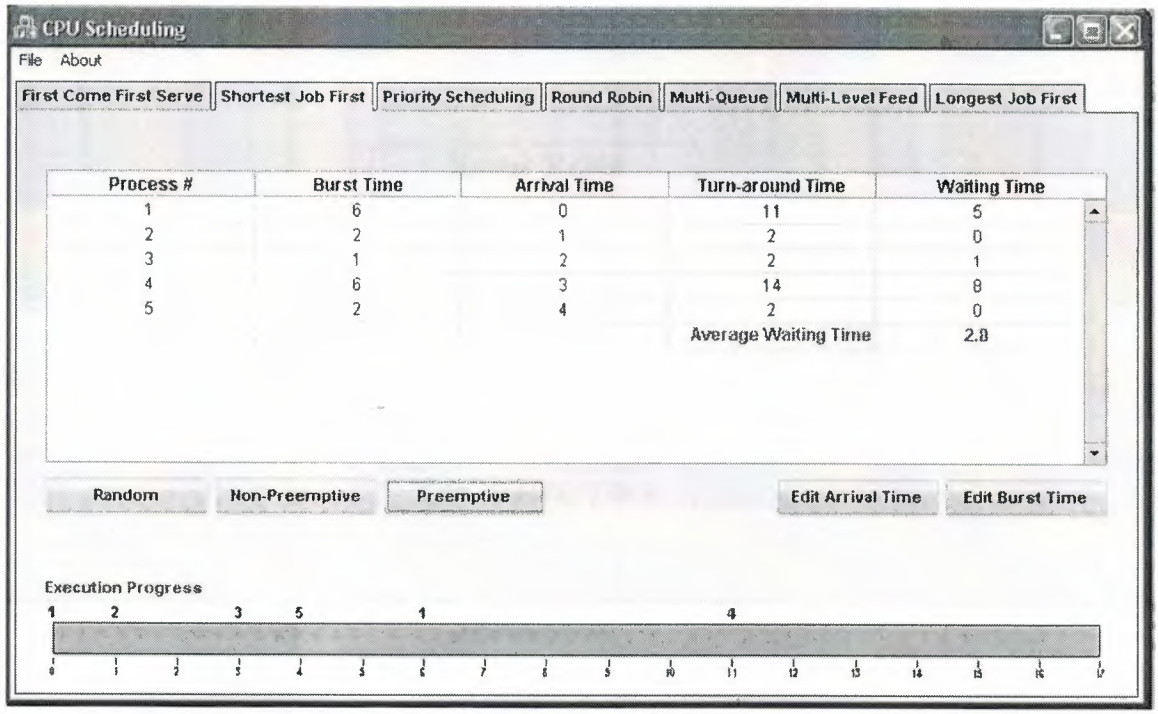


Figure 5.6 SJF Preemptive Interface

Process P1 takes the CPU at first as it has the shortest burst time and the first arrival, but it will be interrupted by P2 which arrives after P1 finished only 1 unit and still has 5 units in its burst time, which is longer than P2 burst time, after P2 finishes executing, the rest of processes will be executed according to their burst times, P3, P5, and then P4.

The average waiting time = (5 + 0 + 1 + 8 + 0) / 5 = 2.8 units.

The average turnaround time = (11 + 2 + 2 + 14 + 2) / 5 = 6.2 units.

The average response ratio = 31 / 17 = 1.823 units.

5.5.3 Round Robin Scheduling Algorithm

Round Robin Scheduling is preemptive therefore it is effective in time-sharing environments in which the system needs to guarantee reasonable response times for interactive users, the only interesting issue with round robin scheme is the length of the quantum. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and approximates FCFS; in any event, the average waiting time under round robin scheduling is often quite long.

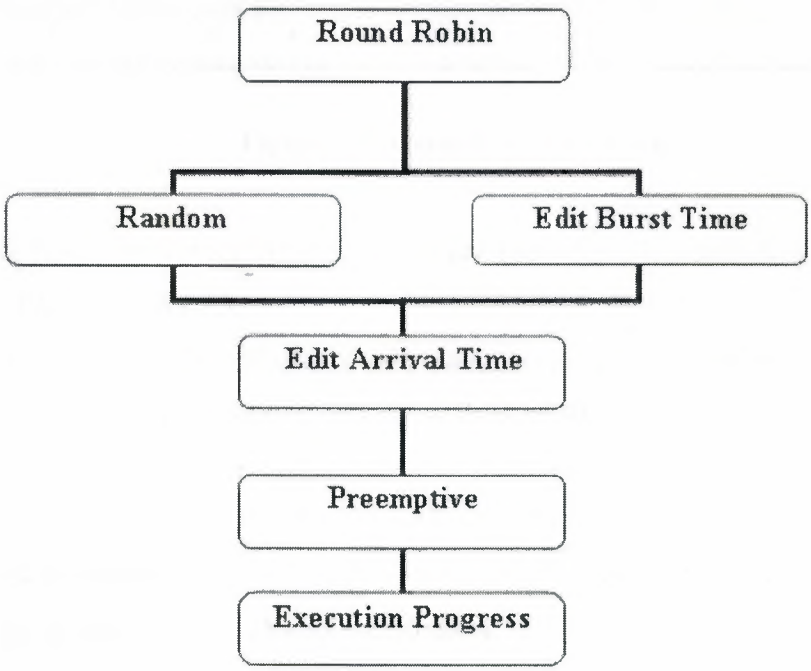


Figure 5.7 Round Robin Flowchart

Figure 5.7 specifies the steps which the user should follow to get the execution bar for the five processes scheduling using Round Robin scheduling algorithm.

Figure 5.8 Round Robin Interface

Burst times for the processes in this example are 1 unit for P1, 6 units for P2, 2 units for P3, 2 units for P4, 2 units for P5.

Each process will use the CPU for single quantum, process P1 will start according to the arrival time, and then all processes will finish its burst time sequentially.

The average waiting time = (0 + 6 + 3 + 3 +3) / 5 = 3 units.

The average turnaround time = (1 + 12 + 5 + 5 + 5) / 5 = 5.6 units.

The average response ratio = 28 / 13 = 2.15 units.

5.5.4 Priority Scheduling Algorithm

The basic idea is straightforward: each process is assigned a priority, and priority is allowed to run. Equal-Priority processes are scheduled in FCFS order. The shortest-Job-First (SJF) algorithm is a special case of general priority scheduling algorithm.

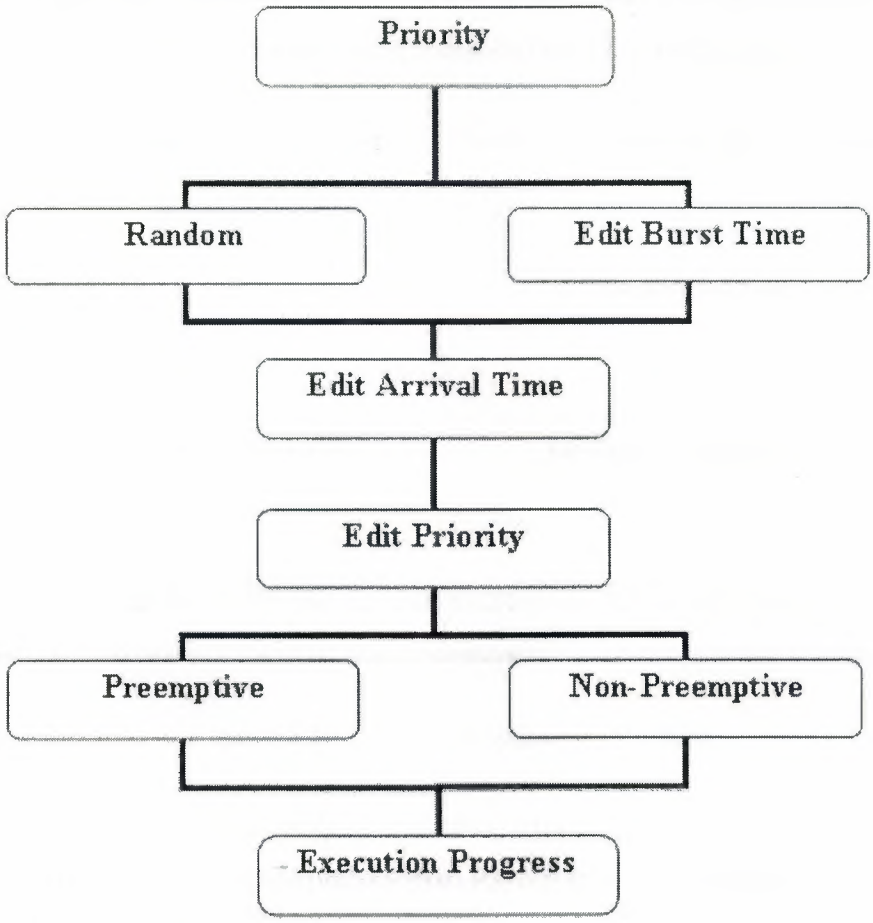


Figure 5.9 Priority Scheduling Flowchart

Both Priority and Shortest Job First have nearly the same flowchart; the only difference is the ability to change the priority for each process, which determines the arrangement of the processes execution.

Figure 5.10 shows an example of Non-preemptive Priority Scheduling, the value of the priority varies from 1 to 5, and the smallest value has the highest priority and vise versa.

The values of the burst times for the five processes are, 2 units for P1, 6 units for P3, 5 units for P3, 5 units for P4, and 2 units for P5.

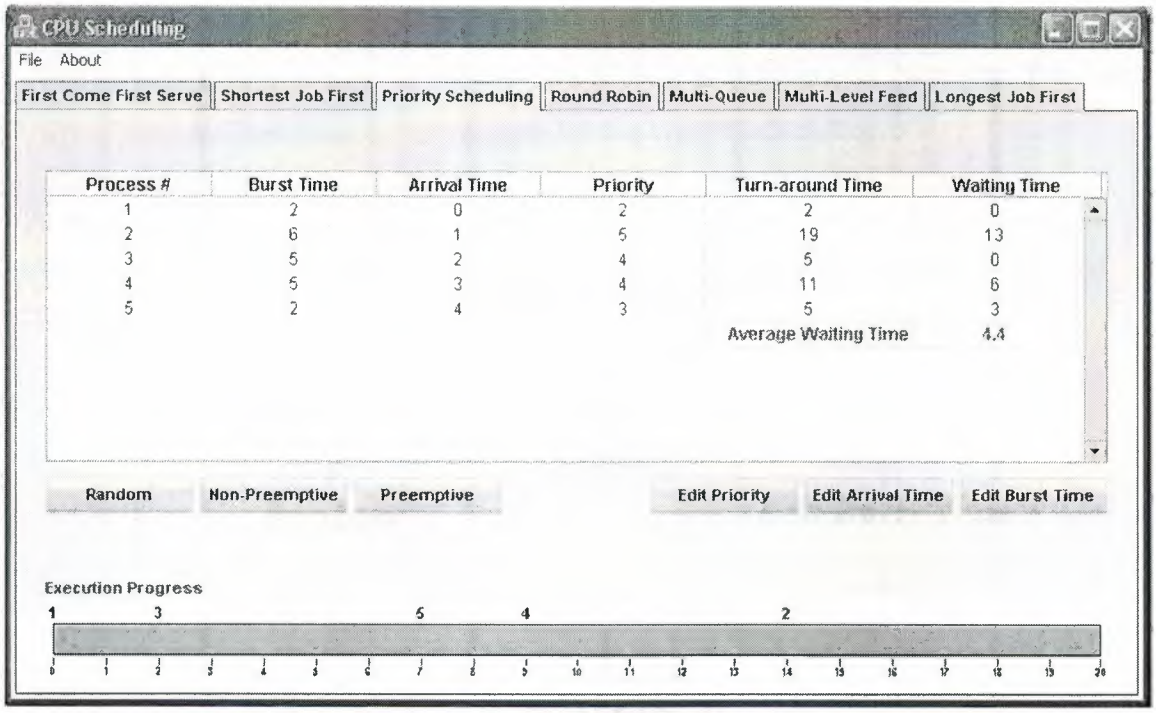


Figure 5.10 Priority Scheduling Interface

Process P1 has the highest priority and also arrives first, then process P3 arrives after, although process P5 has higher priority than P3, but P5 has not arrived yet.
Process P4 follows P5 that it has more priority; at the end Process P2 will use the CPU because it has the lowest priority.

The average waiting time = (0 + 13 + 0 + 6 + 3) / 5 = 4.4 units.
The average turnaround time = (2 + 19 + 5 + 11 + 5) / 5 = 8.4 units.
The average response ratio = 42 / 20 = 2.1 units.

5.5.5 Multi-level Feedback Scheduling Algorithm

Multilevel feedback queue-scheduling algorithm allows a process to move between queues. It uses many ready queues and associates a different priority with each queue; the Algorithm chooses to process with highest priority from the occupied queue and run that process non-preemptively. If the process uses too much CPU time it will moved to a lower-priority queue.

Similarly, a process that wait too long in the lower-priority queue may be moved to a higher-priority queue may be moved to a highest-priority queue. Note that this form of aging prevents starvation, as an example for this type of scheduling.

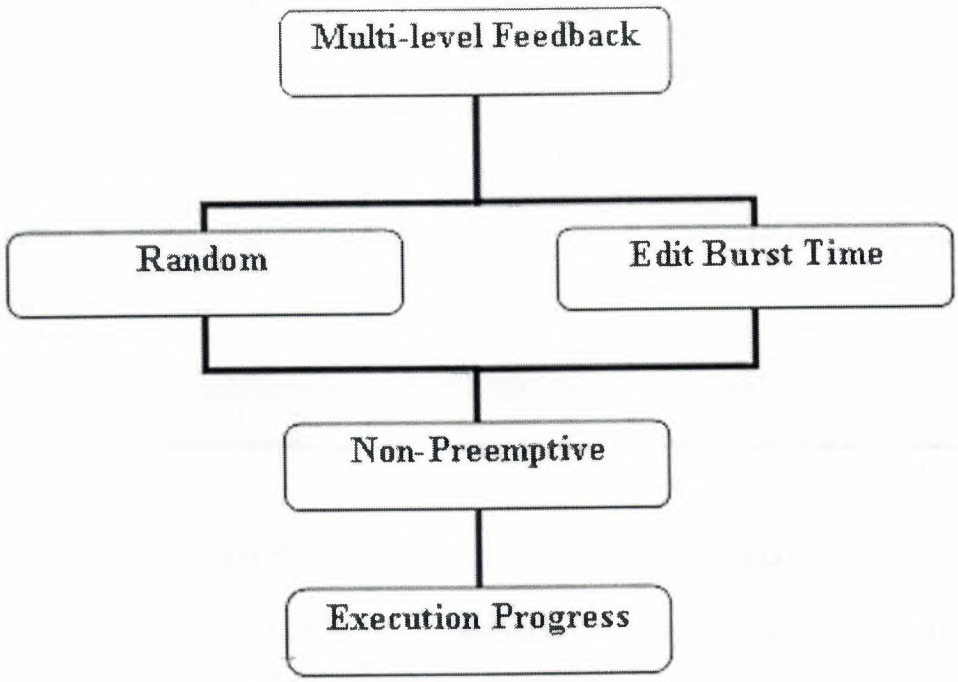


Figure 5.11 Multi-level Feedback Flowchart

The previous flowchart shows the steps which should be followed by the user to get the execution progress bar.

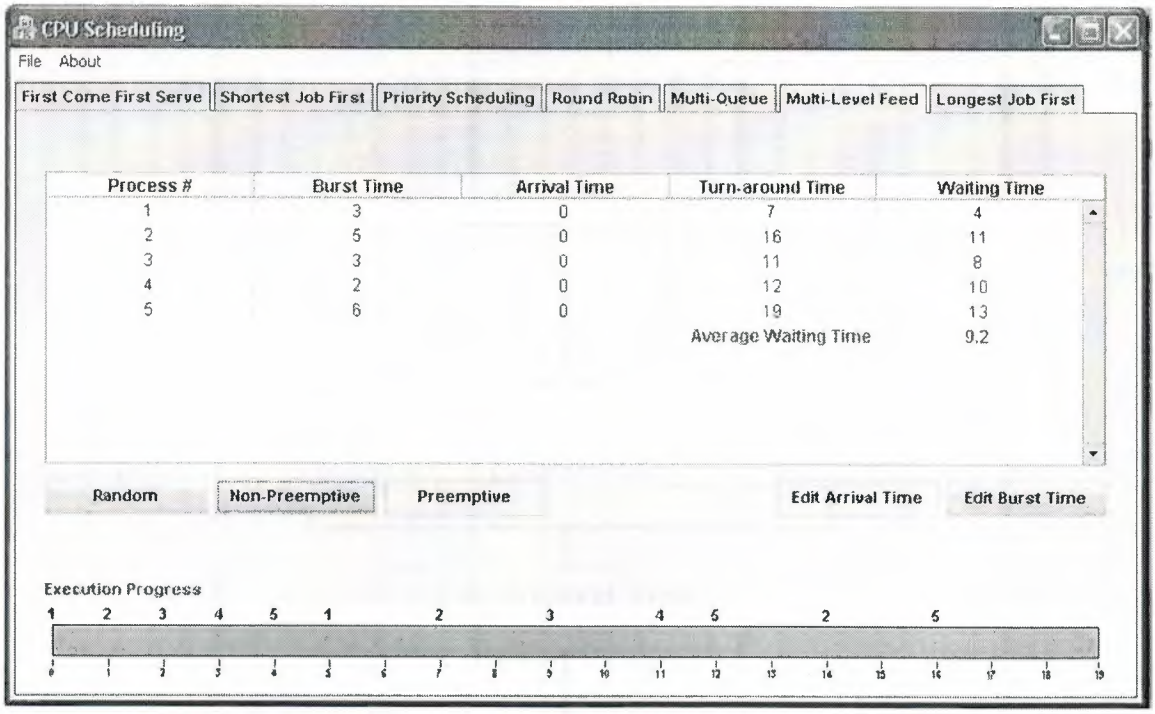


Figure 5.12 Multi-level Feedback Queue Interface

As it is shown in (Figure 5.12) the bust times values for the processes are, 3 units for P1, 5 units for P2, 3 units for P3, 2 units for P4, 6 units for P5.

All the processes will enter the first queue to finish only one unit for each process, then again all the processes will enter the second queue to finish 2 units for each process except process P4 which finishes execution.

The third queue will be executed by First Come First Served, for processes P2 and P5.

The average waiting time = (4 + 11 + 8 + 10 + 13) / 5 = 9.2 units.

The average turnaround time = (7 + 16 + 11 + 12 + 19) / 5 = 13 units.

The average response ratio = 65 / 19 = 3.42 units.

5.5.6 Largest Job First Scheduling Algorithm

Largest Job First attend to assign the longest burst time process next to the CPU, which allow long jobs to have the opportunity to be executed before short jobs, this type is suitable in systems where longjobs cant wait until short jobs finish execution.

Preemptive LJF allows longer time remaining to preempt long process which guarantees all processes to have chance to finish even part of its burst time.

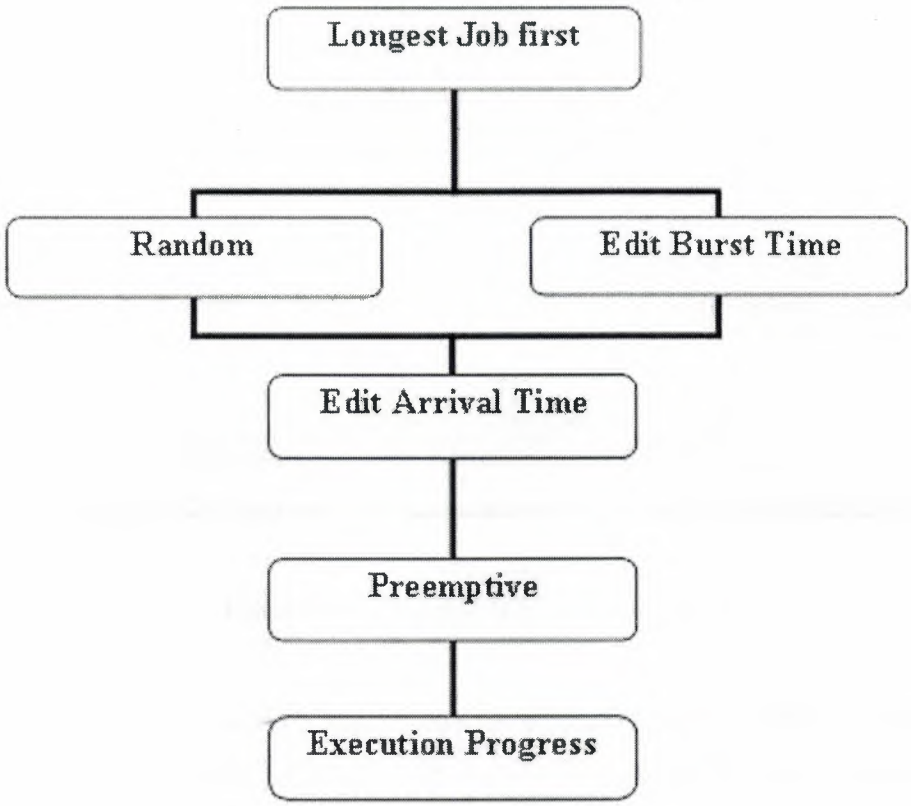


Figure 5.13 Longest Job First Flowchart

The previous Flowchart represent the steps to execute processes by Longest Job First scheduling algorithm, LJF has only preemptive way to enable shorter processes to preempt long ones, which is usually used with systems that has large number of processes, consider the following queue of processes which will be executed by preemptive Longest Job First scheduling algorithm.

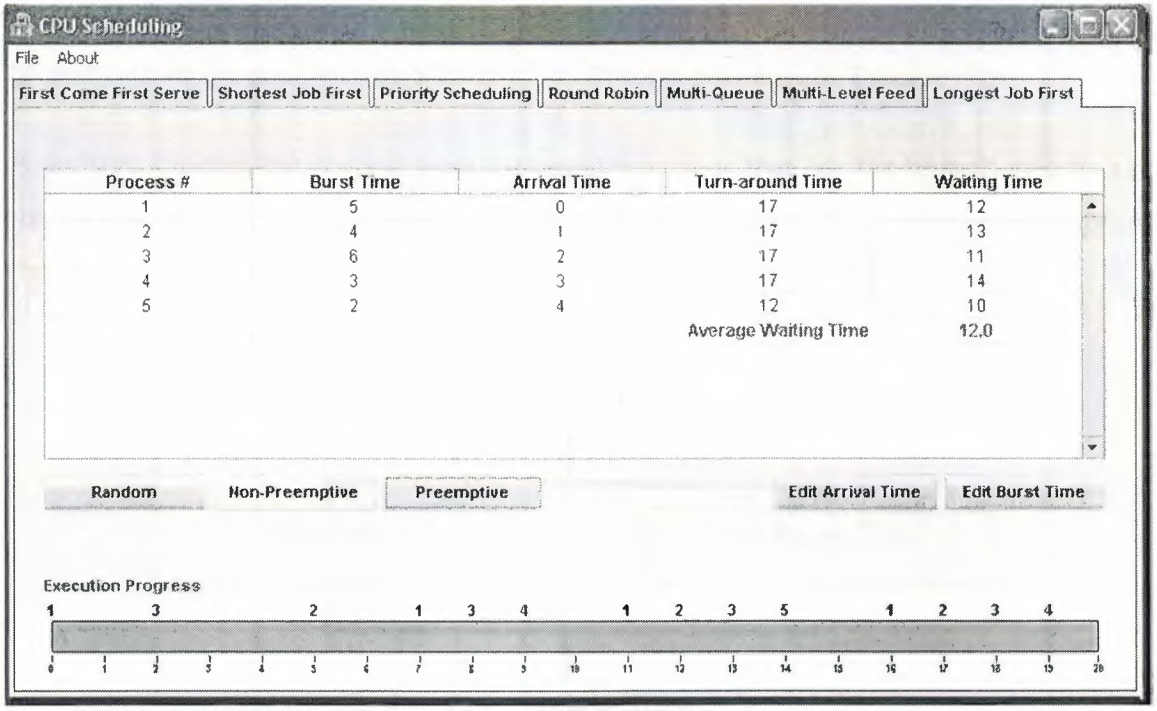


Figure 5.14 Longest Job First Interface

The burst time's values in the previous figure for all processes are 5 units for process P1, 4 units for process P2, 6 units for process P3, 3 units for process P4, and 2 units for process P5.

Process P1 will be assigned to the CPU first, it will be preempted by P3 which has the longest remaining time, after process P3 executes 3 units it is interrupted by P2, which executes 2 units, then all processes will continue executing in the same way.

The average waiting time = (12 + 13 + 11 + 14 + 10) / 5 = 12 units which is relatively long.

The average turnaround time = (17 + 17 + 17 + 17 + 12) / 5 = 16 units.

The average response ratio = 80 / 20 = 4 units.

5.5.7 Multi-level Queue Scheduling Algorithm

A multilevel queue scheduling algorithm partitions the ready queue in several separate queues, for instance; in a multilevel queue scheduling processes are permanently assigned

to one queue; the processes are permanently assigned to one another, based on some property of the process, such as Memory size, Process priority, and Process type

Algorithm chooses the process from the occupied queue that has the highest priority and run that process either preemptively or non-preemptively [19]. The next flowchart explains the steps of running processes using Multi-Level scheduling algorithm.

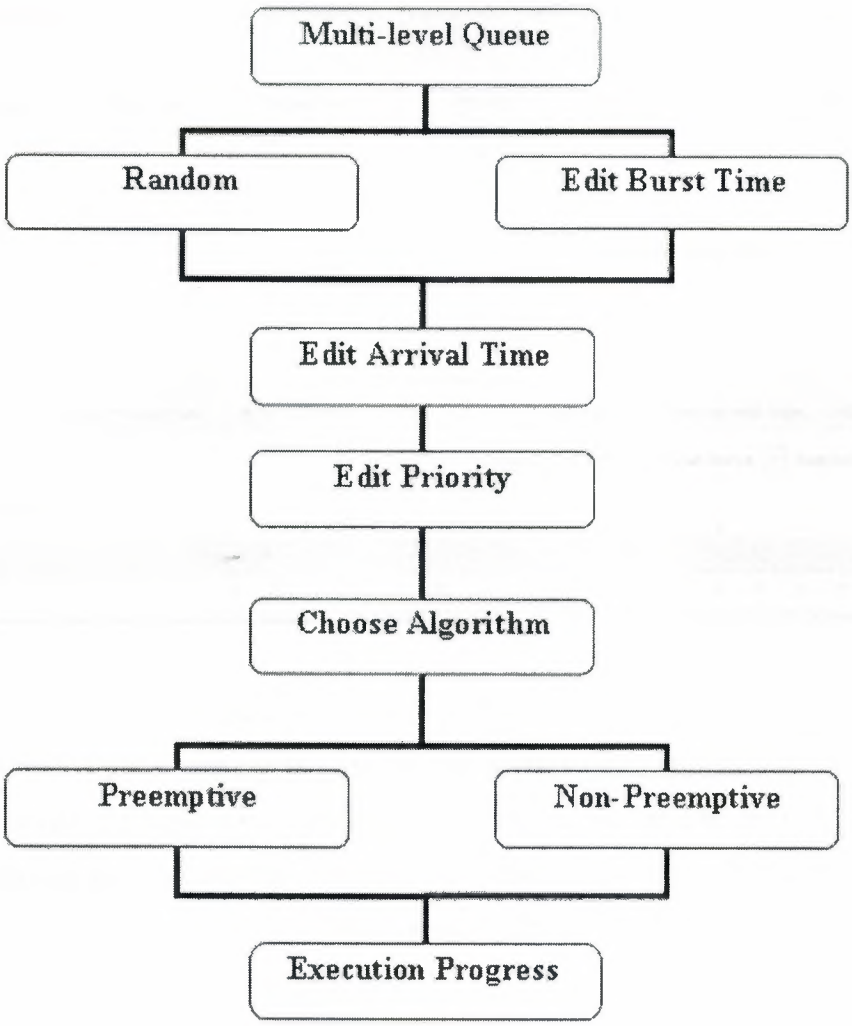


Figure 5.15 Multi-level queue Flowchart

Choose algorithm is the step where the program runs another scheduling algorithm in case of equal priorities; this option includes three algorithms First Come First Served, Shortest Job First, and Round Robin.

Consider the following queue of processes with burst time's values as follows:
6 units for process P1, 5 units for process P2, 3 units for P3, 6 units for process P4, and 6 units for processes P5, in this example the algorithm which is used in case of equal priorities is First Come First Served.

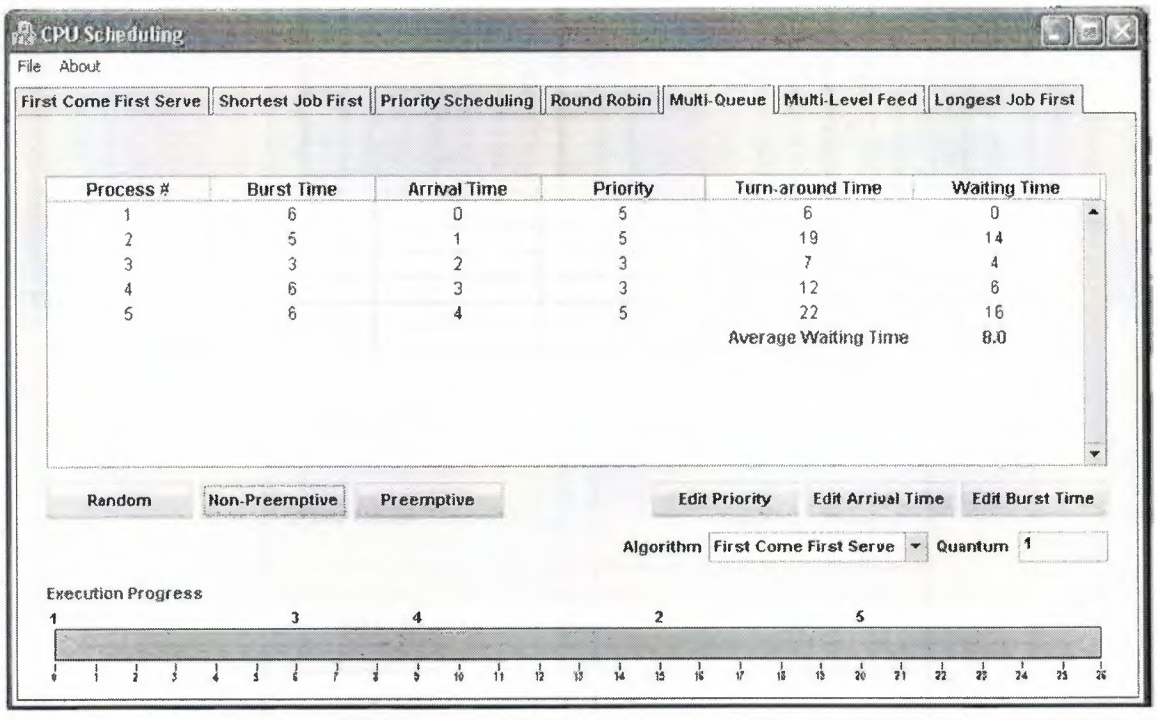


Figure 5.16 Multi-level Queue Interface

Process P1 will start executing because of the arrival time, then it will be followed by process P3 which the highest priority, process P4 will be assigned to the CPU after as it has the next priority and the next arrival time also, then process P2 because it has the next arrival time and the next priority, finally process P5 will be assigned that it has the last arrival time and the least priority.

The average waiting time = (0 + 14 + 4 + 6 + 16) / 5 = 8 units.
The average turnaround time = (6 + 19 + 7 + 12 + 22) / 5 = 13.2 units.
The average response ratio = 66 / 26 = 2.53 units.

5.6 Comparison between Scheduling Algorithms Using Simulation Tool

By applying the seven scheduling algorithms on two sets of processes, it is obvious that according to the process variables the best scheduling algorithm may differ.

Table 5.1 Set of Examined Processes

Process	Arrival Time	Burst Time
P1	0	5
P2	1	5
P3	2	3
P4	3	6
P5	4	2

Type of Scheduling Algorithm	Average Waiting Time
First Come First Served	9.4
Shortest Job First	7.4
Priority Scheduling	7.4
Round Robin	10
Multilevel Queue	6.6
Multi level Feedback	11.8
Longest Job First	12.8

Table 5.2 Different Responses for Scheduling Algorithms

Table 5.3 Set of Examined Processes

Process	Arrival Time	Burst Time
P1	0	2
P2	1	1
P3	2	3
P4	3	6
P5	4	5

Table 5.4 Different Responses for Scheduling Algorithms

Type of Scheduling Algorithm	Average Waiting Time
First Come First Served	4.6
Shortest Job First	4.4
Priority Scheduling	2.6
Round Robin	3.2
Multilevel Queue	4.0
Multi level Feedback	6.2
Longest Job First	7.4

Previous tables shows that in the first examined set of processes, Multilevel queue has got the best response, although in the second set of examined processes priority scheduling algorithm got the best average waiting time.

CONCLUSION

CPU scheduling is the basis of multi-programmed operating systems, by switching the CPU among processes; the operating system can make the computer more productive, choosing the best CPU scheduling algorithm for a particular system is a difficult issue, because there are many scheduling algorithms and each with its own parameters.

In this work, an approach was presented to improve the process manager in operating systems, it is well known that each system is preferred to have high CPU utilization under the constraint that response time is relatively low, also the throughput should be high such that the turnaround time is (on average) linearly proportional to total execution time.

In this approach, an evaluation for each scheduling algorithm was presented under consideration, by using a simulation method to determine the performance by imitating the scheduling algorithm on a representative sample of processes, and computing the resulting performance.

This simulation tool include seven methods of scheduling algorithms, the benefit of such approach is to compare easily between those methods and to find the suitable scheduling method for any type of systems, by finding all the results of a method such as response time, average turnaround time, and average waiting time, it will be easy to decide which method of scheduling algorithms is the most suitable and effective for the particular collection of regular processes.

Based on the findings and experimental results in this thesis, the following problems are suggested for further research:

- Make simulation for more methods of scheduling algorithms.
- Merge types of scheduling algorithms may result of better results.
- Connect the simulation tool directly to the system which enables the user to choose the method of scheduling and to change it in case of bad response.
- To develop an optimization algorithm to choose the most suitable scheduling criteria according to the input values (burst time, turnaround time, arrival time

REFERENCES

[1] Dietel, H. M., "Operating Systems", 2nd ed. Addison-Wesley, Reading, MA, 1992.

[2] Finkel, R.A., "An Operating Systems Vade Mecum", 2nd ed. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[3] Goscinski, Andrzej, "Distributed Operating Systems : the logical design".

[4] Hartley, Stephen. J. "Operating Systems Programming".

[5] Krakowiak, S, "Principles of Operating Systems", MIT Press, 1988.

[6] Lane, M.G. and Mooney, J.D, "A Practical Approach to Operating Systems", Boyd and Fraser, 1988.

[7] Silberschatz, A. and Galvin, P.B., "Operating System Concepts", 4th ed. Addison-Wesley, Reading, MA, 1994.

[8] D. G. Feitelson. A survey in Scheduling in Multi-programmed Parallel Systems. Research report rc 19790 (87657), IBM T.J. Watson Research Center, Yorktown Heights, NY, 1995.

[9] D. G. Feitelson and B. Nitzberg. Job Characteristics of a Production Parallel ScientificWorkload on the NASA Ames iPSC/860. In D. G. Feitelson and L. Rudolph, editor, Proc. of 1st Workshop on Job Scheduling Strategies for Parallel Processing, volume 949 of Lecture Notes in Computer Science, pages 337–360. Springer, 1995.

[10] "Operating Systems", William Stallings,Prentice Hall, 4th ed. 2001, Chapter 4

[11] Mathai Joseph, Real-time Systems: Specification, Verification and Analysis, Prentice Hall International, London, 1996

[12] Bran Selic, Turning Clockwise: Using UML in the Real-Time Domain, Communications of the ACM, October 1999.

[13] Robert Rosenberg, Lessons Learnt Using COTS in Real-Time Embedded Systems, Presented to the Joint Avionics and Weapons Systems Support Software and Simulation Conference, June 1998, http://www.sabresys.com/whitepapers/cots.html

[14] The Linux Kernel Copyright I 1996-1999 David A Rusling david.rusling@arm.com REVIEW, Version 0.8-3 Understanding the Linux 2.6.8.1 CPU Scheduler By Josh Aas c 2005 Silicon Graphics, Inc. (SGI) 17th February 2005

[15] Understanding the Linux 2.6.8.1 CPU Scheduler By Josh Aas c 2005 Silicon Graphics, Inc. (SGI) 17th February 2005.

[16] Dinkel, W., Niehaus, D., Frisbie, M., Woltersdorf, J. .KURT-Linux User Manual.; on 7/28/2003, obtained from: http://www.ittc.ku.edu/kurt/.

[17] Visual C++ Issues: http://msdn.microsoft.com/library/default.asp?url=/library/enus/vcug98/html/_asug_overview.3a_.user_interface_basics.asp

[18] D. G. Feitelson. Metrics for Parallel Job Scheduling and their Convergence. In D. G. Feitelson and L. Rudolph, editor, Proc. of 7th Workshop on Job Scheduling Strategies for Parallel Processing, volume 2221 of Lecture Notes in Computer Science, pages 190–208. Springer, 2001.

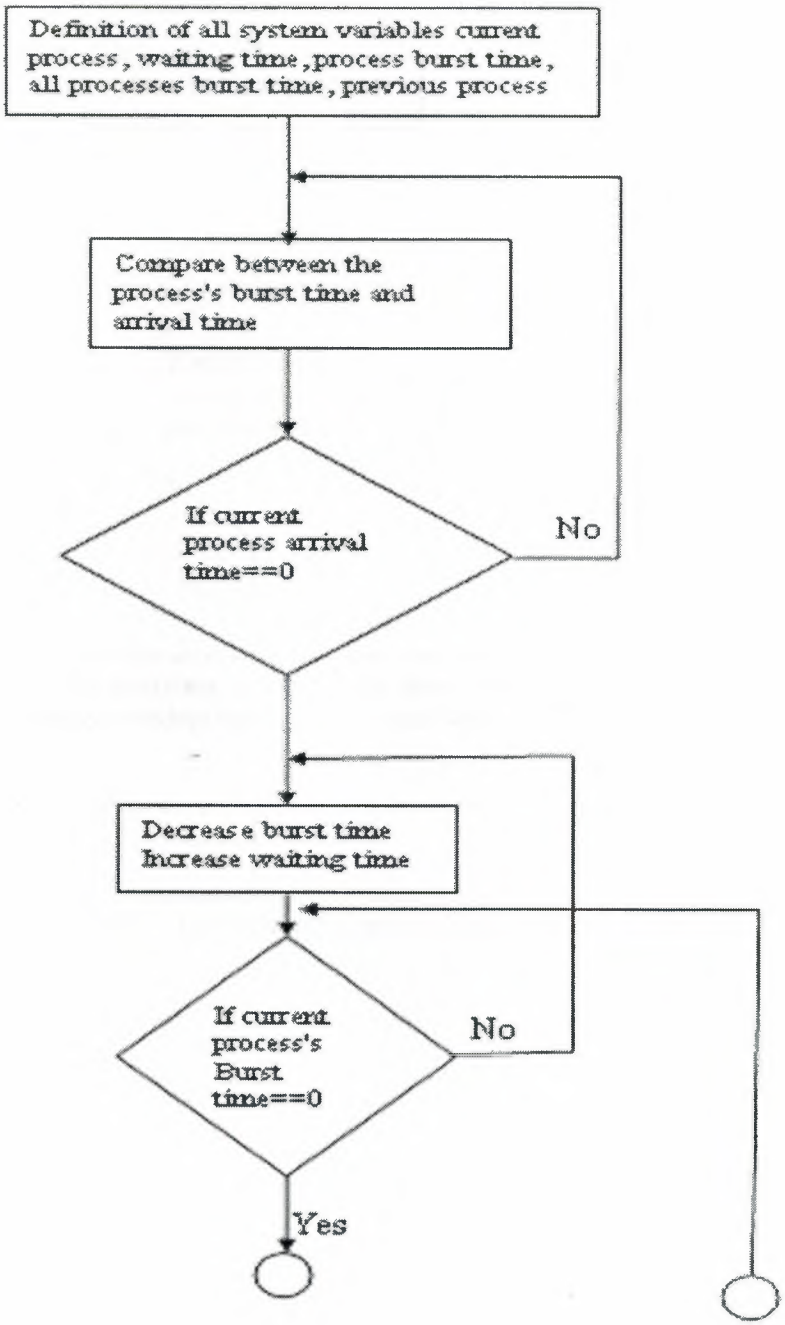
[19] Singhal, Mukesh and Shivaratri, "Advanced Concepts in Operating Systems".

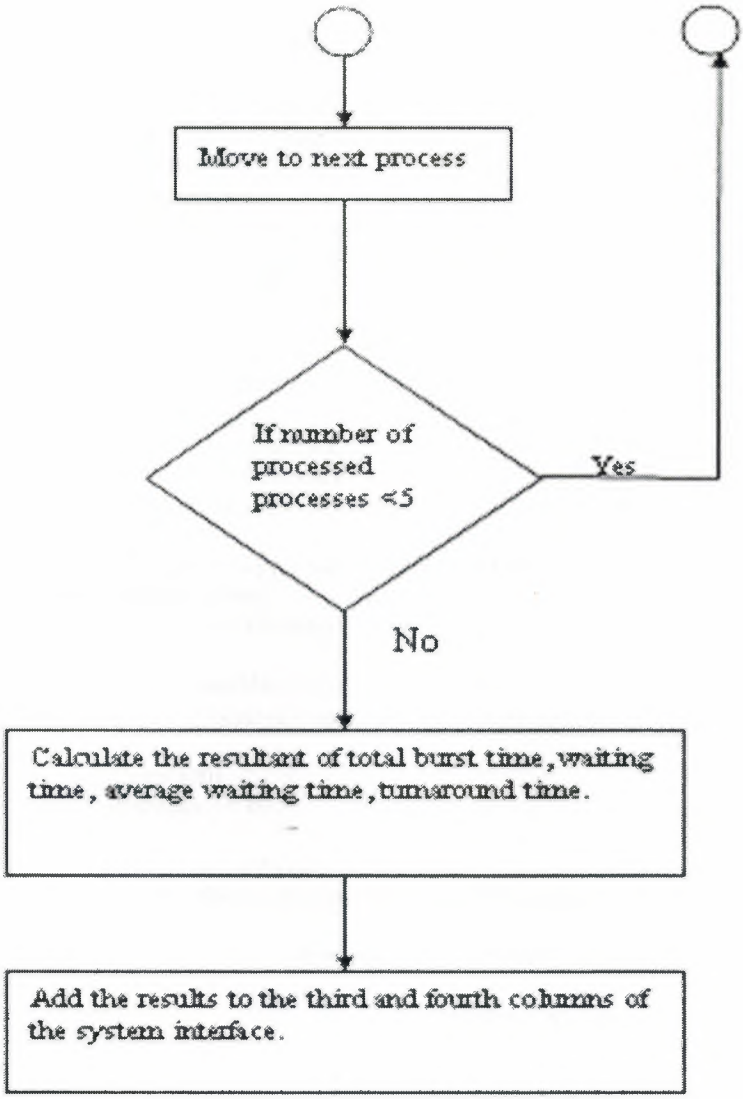
APPENDICES
Scheduling Algorithms Code

First Come First Serve (FCFS) Algorithm

```
Void CAlgorithm1Dlg::OnNonPreemptive (void)
{
        PROCESS_PERIODS1 pp[5];
        memcpy(pp, m_pp, sizeof(pp));
        CHAR szNumber[10];
        m_ctlAnimation.SetRange(0, m_nTotalBurstTime);
        m_ctlAnimation.SetPos(0);
        int nCurProcess = 0;
        m_pp[0].nWaitingTime = 0;
        int nWaitingTime = 0;
        m_ctlAnimation.ClearMarks();
        m_ctlAnimation.AddMark(1, 0);
        for(int nIndex = 0; nIndex < m_nTotalBurstTime; nIndex++)
        {
                Sleep(300);
                m_pp[nCurProcess].nBurstTime--;
                nWaitingTime++;
                m_ctlAnimation.SetPos(m_ctlAnimation.m_nPos + 1);
                if(m_pp[nCurProcess].nBurstTime == 0)
                {
                        nCurProcess++;
                        if(nCurProcess < 5)
                        {
                                m_pp[nCurProcess].nWaitingTime = m_pp[nCurProcess-
1].nWaitingTime + nWaitingTime;
                                m_ctlAnimation.AddMark(nCurProcess+1, m_ctlAnimation.m_nPos);
                        }
                        nWaitingTime = 0;
                }
        }
        int nTotalWaitingTime = 0;
        for(int nIndex = 0; nIndex < 5; nIndex++)
        {
                nTotalWaitingTime += m_pp[nIndex].nWaitingTime;
                m_pp[nIndex].nTurnaroundTime = m_pp[nIndex].nWaitingTime + pp[nIndex].nBurstTime;
                sprintf(szNumber, "%d", m_pp[nIndex].nWaitingTime);
                m_lstNumbers.SetItemText(nIndex, 3, szNumber);
                sprintf(szNumber, "%d", m_pp[nIndex].nTurnaroundTime);
                m_lstNumbers.SetItemText(nIndex, 2, szNumber);
        }
        double fAverageWaitingTime = double(nTotalWaitingTime) / 5.0;
        sprintf(szNumber, "%0.1f", fAverageWaitingTime);
        m_lstNumbers.SetItemText(5, 3, szNumber);
        memcpy(m_pp, pp, sizeof(pp));
        m_lstNumbers.Invalidate()
```

FCFS Code Flow Chart:



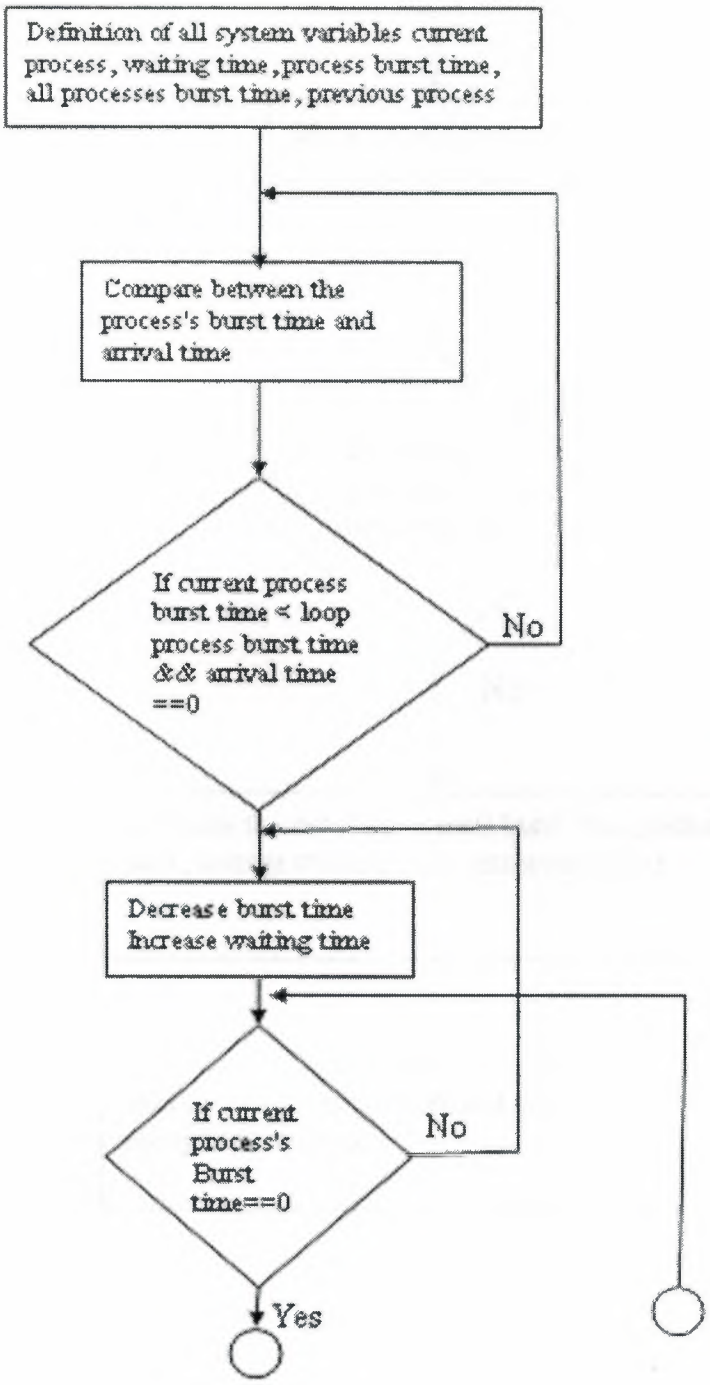

Move to next process
If number of processed processes <5
Yes
No
Calculate the resultant of total burst time, waiting time, average waiting time, turnaround time.
Add the results to the third and fourth columns of the system interface.

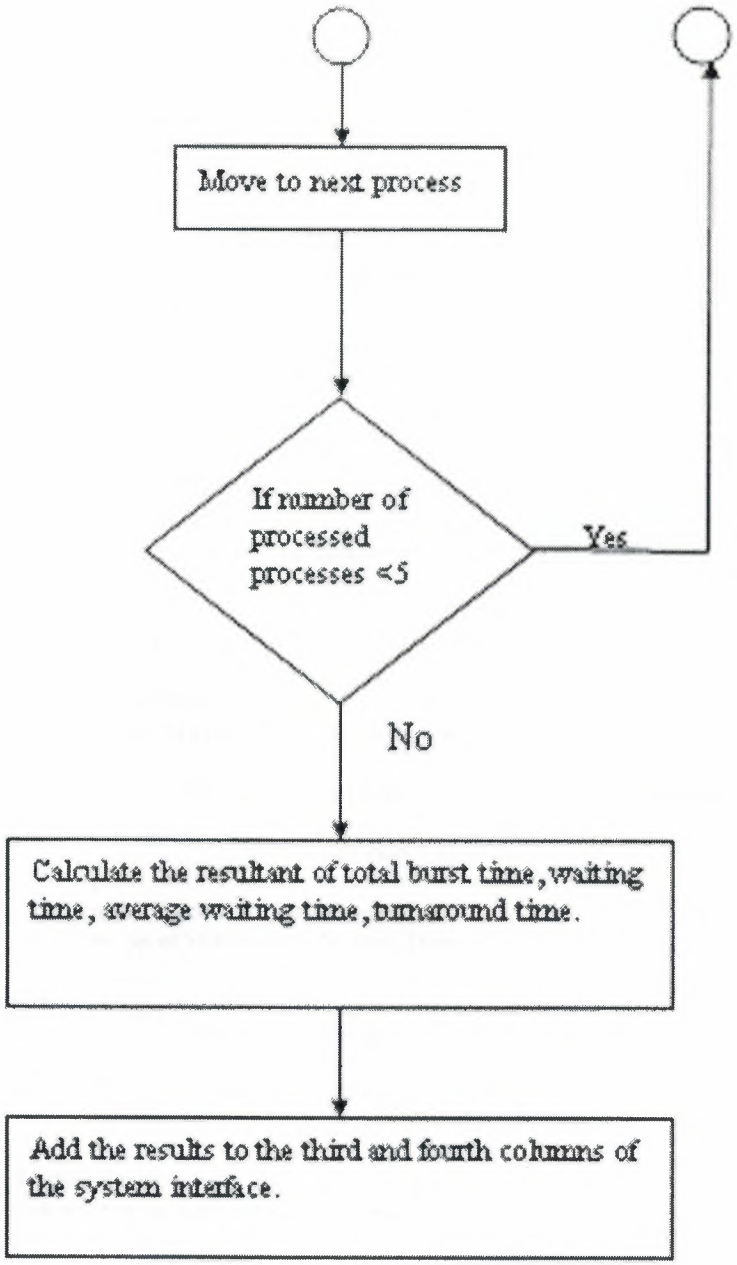
Shortest Job First (SJF) Algorithm

```
void CAlgorithm2Dlg::OnNonPreemptive(void)
{
	PROCESS_PERIODS2 pp[5];
	memcpy(pp, m_pp, sizeof(pp));
	int nArrivalTime = 0;
	CHAR szNumber[10];
	int nTotalBurstTime = m_nTotalBurstTime;
	m_ctlAnimation.SetRange(0, m_nTotalBurstTime);
	m_ctlAnimation.SetPos(0);
	int nCurProcess = 0;
	int nPrevProcess;
	m_pp[0].nWaitingTime = 0;
	int nWaitingTime = 0;
	m_ctlAnimation.ClearMarks();
	m_ctlAnimation.ClearIdleMarks();
	for(int nLoop = 0; nLoop < 5; nLoop++)
	{
		if(m_pp[nLoop].nArrivalTime == 0 && m_pp[nLoop].nBurstTime <
m_pp[nCurProcess].nBurstTime)
			nCurProcess = nLoop;
	}
	m_ctlAnimation.AddMark(nCurProcess+1, 0);
	for(int nIndex = 0; nIndex < m_nTotalBurstTime; nIndex++)
	{
		Sleep(300);
		if(nIndex == 0)
		{
			nCurProcess = 0;
			for(int nLoop = 0; nLoop < 5; nLoop++)
			{
				if(m_pp[nLoop].nArrivalTime == 0 && m_pp[nLoop].nBurstTime <
m_pp[nCurProcess].nBurstTime)
					nCurProcess = nLoop;
			}
			nPrevProcess = nCurProcess;
			m_pp[nCurProcess].nWaitingTime = 0;
		}
		m_pp[nCurProcess].nBurstTime--;
		nWaitingTime++;
		m_ctlAnimation.SetPos(m_ctlAnimation.m_nPos + 1);
		if(m_pp[nCurProcess].nBurstTime == 0 && nIndex < m_nTotalBurstTime - 1)
		{
			m_pp[nCurProcess].bProcessed = TRUE;
			nCurProcess = 0;
			BOOL bFound = FALSE;
			for(int nLoop = 0; nLoop < 5; nLoop++)
			{
				if(m_pp[nLoop].bProcessed == FALSE && m_pp[nLoop].nArrivalTime
<= nIndex + 1)
				{
					nCurProcess = nLoop;
```

```
                                bFound = TRUE;
                                        break;
                                }
                        }
                        if(!bFound)
                        {
                                for(; nIndex < m_nTotalBurstTime; nIndex++)
                                {
                                        int nLoop;
                                        for(nLoop = 0; nLoop < 5; nLoop++)
                                        {
                                                if(m_pp[nLoop].bProcessed == FALSE &&
m_pp[nLoop].nArrivalTime <= nIndex + 1)
                                                {
                                                        nCurProcess = nLoop;
                                                        break;
                                                }
                                        }
                                        if(nCurProcess == nLoop)
                                                break;
                                        nWaitingTime++;
                                        m_ctlAnimation.SetIdleMark(m_ctlAnimation.m_nPos);
                                        m_ctlAnimation.SetPos(m_ctlAnimation.m_nPos + 1);
                                        m_nTotalBurstTime++;
                                        m_ctlAnimation.SetRange(0, m_nTotalBurstTime);
                                        m_ctlAnimation.SetGrid(m_nTotalBurstTime);
                                }
                        }
                        for(int nLoop = 0; nLoop < 5; nLoop++)
                        {
                                if(m_pp[nLoop].bProcessed == FALSE && m_pp[nLoop].nArrivalTime
<= nIndex + 1 &&
                                  m_pp[nLoop].nBurstTime < m_pp[nCurProcess].nBurstTime)
                                        nCurProcess = nLoop;
                        }
                        m_pp[nCurProcess].nWaitingTime = m_pp[nPrevProcess].nWaitingTime +
nWaitingTime;
                        m_ctlAnimation.AddMark(nCurProcess+1, m_ctlAnimation.m_nPos);
                        nPrevProcess = nCurProcess;
                        nWaitingTime = 0;
                }
        }
        int nTotalWaitingTime = 0;
        for(int nIndex = 0; nIndex < 5; nIndex++)
        {
                nTotalWaitingTime += m_pp[nIndex].nWaitingTime;
                m_pp[nIndex].nTurnaroundTime = pp[nIndex].nBurstTime + m_pp[nIndex].nWaitingTime;
                sprintf(szNumber, "%d", m_pp[nIndex].nWaitingTime);
                m_lstNumbers.SetItemText(nIndex, 4, szNumber);
                sprintf(szNumber, "%d", m_pp[nIndex].nTurnaroundTime);
                m_lstNumbers.SetItemText(nIndex, 3, szNumber);
        }
        double fAverageWaitingTime = double(nTotalWaitingTime) / 5.0;
        sprintf(szNumber, "%0.1f", fAverageWaitingTime);
        m_lstNumbers.SetItemText(5, 4, szNumber); }
```

SJF Code Flow Chart:



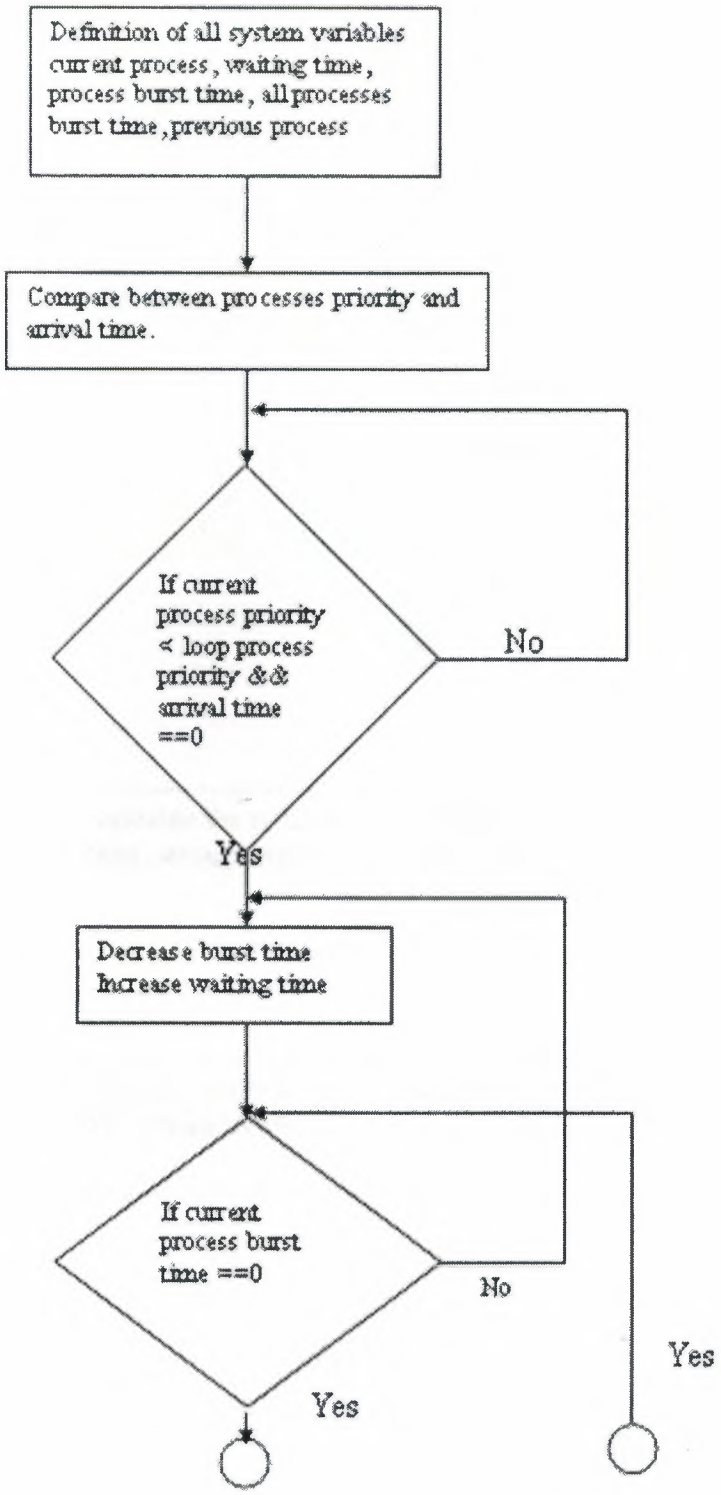

Move to next process
If number of processed processes <5
Yes
No
Calculate the resultant of total burst time, waiting time, average waiting time, turnaround time.
Add the results to the third and fourth columns of the system interface.

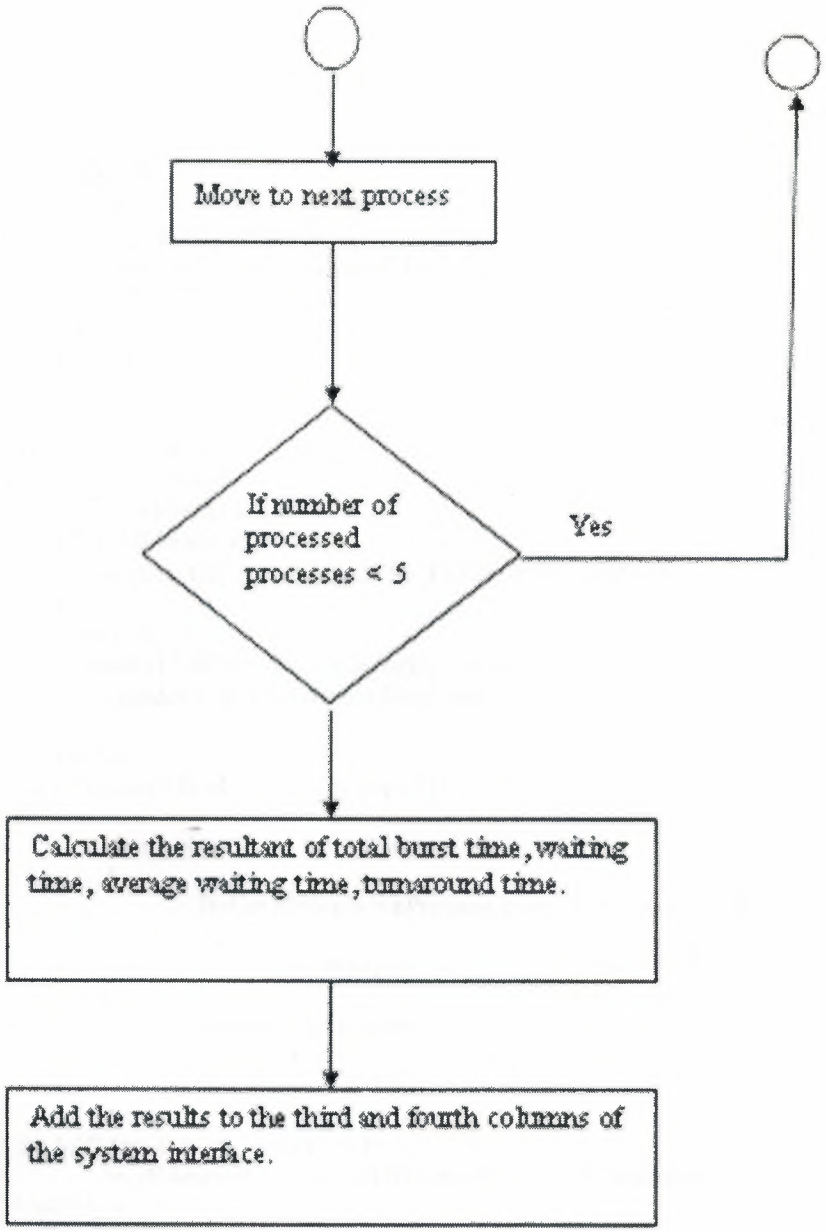
Priority Scheduling Algorithm

```
void CAlgorithm3Dlg::OnNonPreemptive(void)
{
	PROCESS_PERIODS3 pp[5];
	memcpy(pp, m_pp, sizeof(pp));
	int nArrivalTime = 0;
	CHAR szNumber[10];
	int nTotalBurstTime = m_nTotalBurstTime;
	m_ctlAnimation.SetRange(0, m_nTotalBurstTime);
	m_ctlAnimation.SetPos(0);
	int nCurProcess = 0;
	int nPrevProcess;
	m_pp[0].nWaitingTime = 0;
	int nWaitingTime = 0;
	m_ctlAnimation.ClearMarks();
	m_ctlAnimation.ClearIdleMarks();
	m_ctlAnimation.AddMark(1, 0);
	for(int nIndex = 0; nIndex < m_nTotalBurstTime; nIndex++)
	{
		Sleep(300);
		if(nIndex == 0)
		{
			nCurProcess = 0;
			for(int nLoop = 0; nLoop < 5; nLoop++)
			{
				if(m_pp[nLoop].nArrivalTime == 0 && m_pp[nLoop].nPriority <
m_pp[nCurProcess].nPriority)
					nCurProcess = nLoop;
			}
			nPrevProcess = nCurProcess;
			m_pp[nCurProcess].nWaitingTime = 0;
		}
		m_pp[nCurProcess].nBurstTime--;
		for(int nLoop = 0; nLoop < 5; nLoop++)
		{
			if(m_pp[nLoop].bProcessed == FALSE)
				m_pp[nLoop].nTurnaroundTime++;
		}
		nWaitingTime++;
		m_ctlAnimation.SetPos(m_ctlAnimation.m_nPos + 1);
		if(m_pp[nCurProcess].nBurstTime == 0 && nIndex < m_nTotalBurstTime - 1)
		{
			m_pp[nCurProcess].bProcessed = TRUE;
			nCurProcess = 0;
			BOOL bFound = FALSE;
			for(int nLoop = 0; nLoop < 5; nLoop++)
			{
				if(m_pp[nLoop].bProcessed == FALSE && m_pp[nLoop].nArrivalTime
<= nIndex + 1)
				{
					nCurProcess = nLoop;
					bFound = TRUE;
					break;
				}
```

```
                        }
                        if(!bFound)
                        {
                                for(; nIndex < m_nTotalBurstTime; nIndex++)
                                {
                                        int nLoop;
                                        for(nLoop = 0; nLoop < 5; nLoop++)
                                        {
                                                if(m_pp[nLoop].bProcessed == FALSE &&
m_pp[nLoop].nArrivalTime <= nIndex + 1)
                                                {
                                                        nCurProcess = nLoop;
                                                        break;
                                                }
                                        }
                                        if(nCurProcess == nLoop)
                                                break;
                                        nWaitingTime++;
                                        m_ctlAnimation.SetIdleMark(m_ctlAnimation.m_nPos);
                                        m_ctlAnimation.SetPos(m_ctlAnimation.m_nPos + 1);
                                        m_nTotalBurstTime++;
                                        m_ctlAnimation.SetRange(0, m_nTotalBurstTime);
                                        m_ctlAnimation.SetGrid(m_nTotalBurstTime);
                                }
                        }
                        for(int nLoop = 0; nLoop < 5; nLoop++)
                        {
                                if(m_pp[nLoop].bProcessed == FALSE && m_pp[nLoop].nArrivalTime
<= nIndex + 1 &&
                                  m_pp[nLoop].nPriority < m_pp[nCurProcess].nPriority)
                                        nCurProcess = nLoop;
                        }
                        m_pp[nCurProcess].nWaitingTime = m_pp[nPrevProcess].nWaitingTime +
nWaitingTime;
                        m_ctlAnimation.AddMark(nCurProcess+1, m_ctlAnimation.m_nPos);
                        nPrevProcess = nCurProcess;
                        nWaitingTime = 0;
                }
        }
        int nTotalWaitingTime = 0;
        for(int nIndex = 0; nIndex < 5; nIndex++)
        {
                m_pp[nIndex].nTurnaroundTime -= m_pp[nIndex].nArrivalTime;
                m_pp[nIndex].nWaitingTime = m_pp[nIndex].nTurnaroundTime - pp[nIndex].nBurstTime;
                nTotalWaitingTime += m_pp[nIndex].nWaitingTime;
                sprintf(szNumber, "%d", m_pp[nIndex].nWaitingTime);
                m_lstNumbers.SetItemText(nIndex, 5, szNumber);
                sprintf(szNumber, "%d", m_pp[nIndex].nTurnaroundTime);
                m_lstNumbers.SetItemText(nIndex, 4, szNumber);
        }
        double fAverageWaitingTime = double(nTotalWaitingTime) / 5.0;
        sprintf(szNumber, "%0.1f", fAverageWaitingTime);
        m_lstNumbers.SetItemText(5, 5, szNumber);
        memcpy(m_pp, pp, sizeof(pp));
        m_nTotalBurstTime = nTotalBurstTime;
        m_lstNumbers.Invalidate();}
```

Priority Code Flow chart:



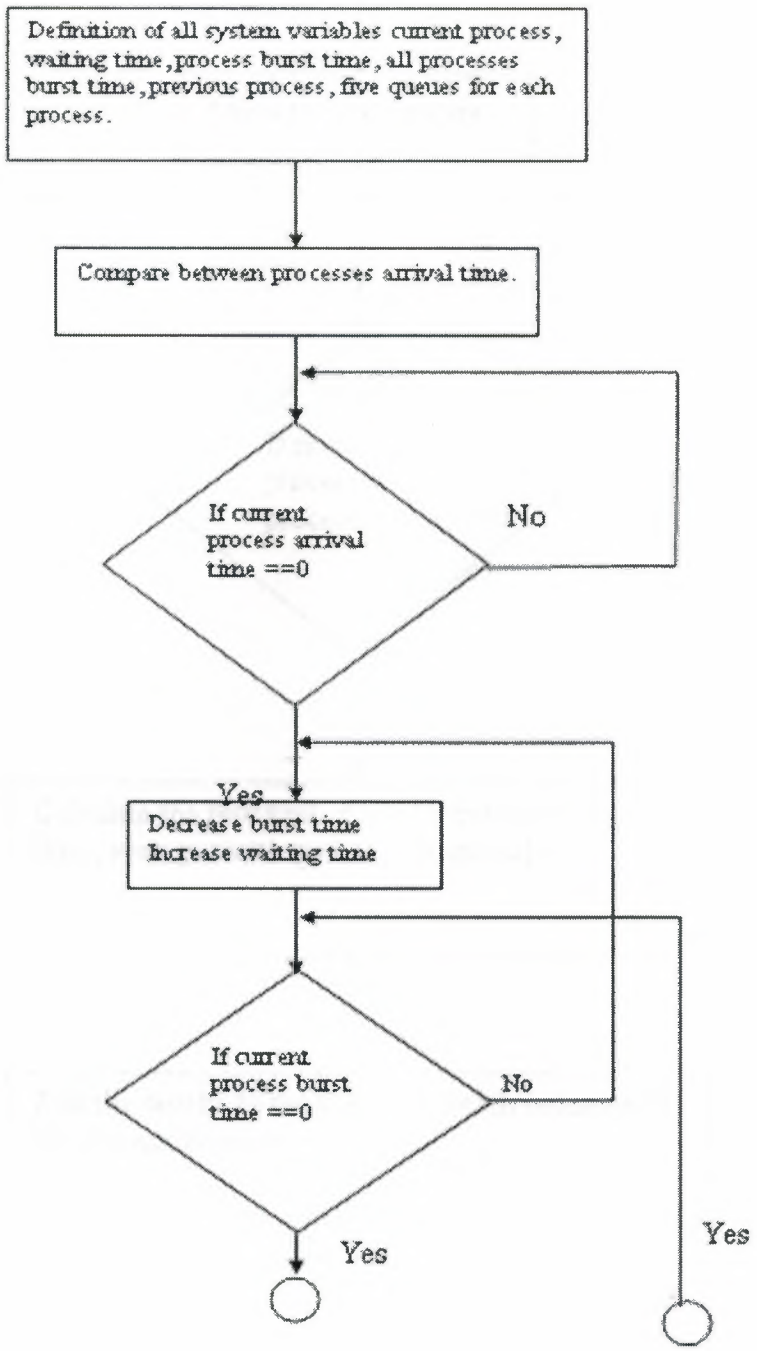

Move to next process
If number of processed processes < 5
Yes
Calculate the resultant of total burst time, waiting time, average waiting time, turnaround time.
Add the results to the third and fourth columns of the system interface.

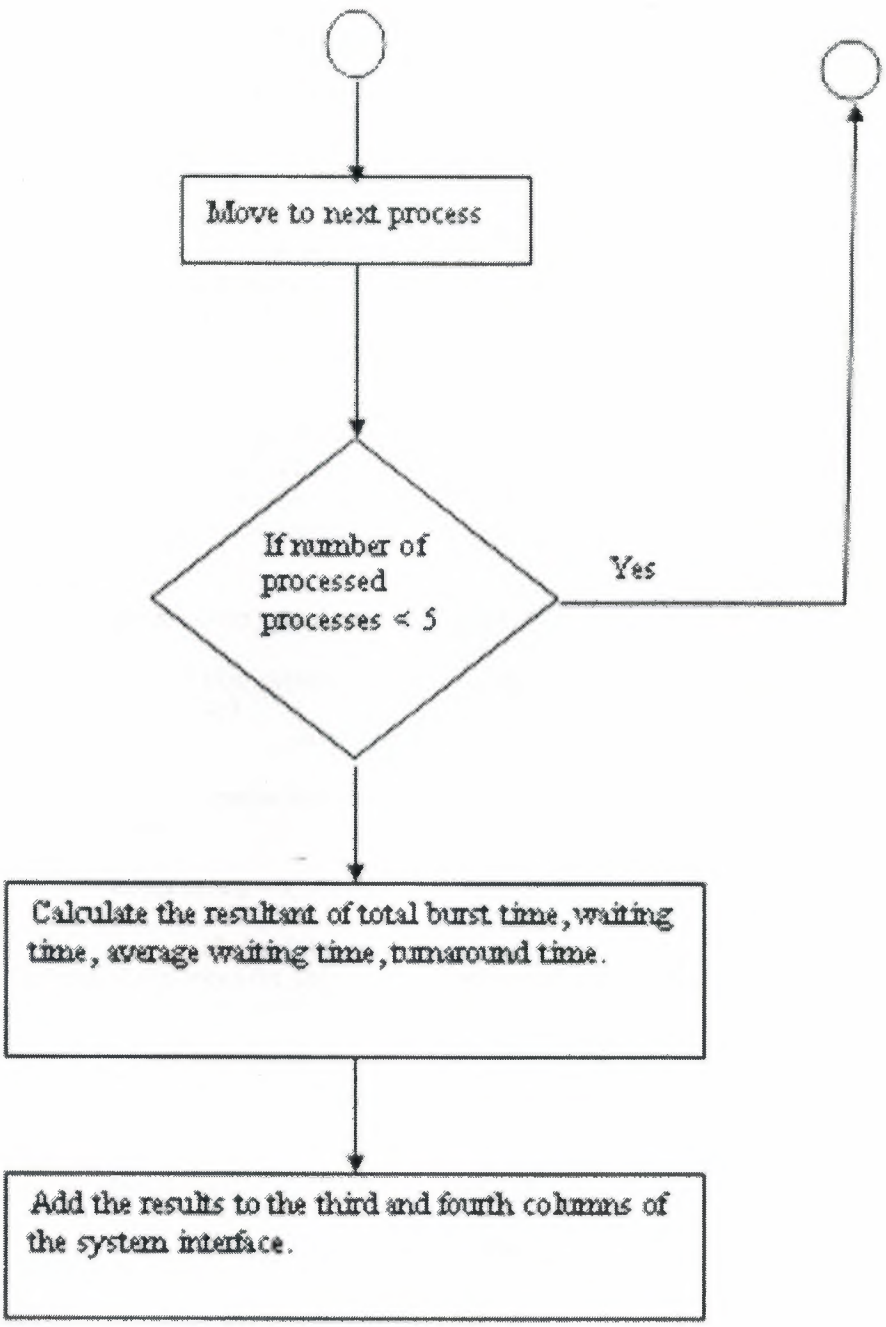
Round Robin Algorithm

```
void CAlgorithm4Dlg::OnNonPreemptive(void)
{
	PROCESS_PERIODS4 pp[5];
	int nArrivalTime = 0;
	CHAR szNumber[10];
	int nTotalBurstTime = m_nTotalBurstTime;
	CopyMemory(pp, m_pp, sizeof(pp));
	m_ctlAnimation.SetRange(0, m_nTotalBurstTime);
	m_ctlAnimation.SetPos(0);
	int nCurProcess = 0;
	m_pp[0].nWaitingTime = 0;
	int nWaitingTime = 0;
	BOOL bFound = TRUE;
	BOOL bAddMark = FALSE;
	m_ctlAnimation.ClearMarks();
	m_ctlAnimation.ClearIdleMarks();
	m_ctlAnimation.AddMark(1, 0);
	qsort((void*)m_pp, (size_t)5, sizeof(PROCESS_PERIODS4), compare4);
	int aQueue[5];
	int nProcessCount = 0;
	aQueue[0] = aQueue[1] = aQueue[2] = aQueue[3] = aQueue[4] = -1;
	for(int nIndex = 0; nIndex < m_nTotalBurstTime; nIndex++)
	{
		Sleep(100);
		for(int nLoop = 0; nLoop < 5; nLoop++)
		{
			if(m_pp[nLoop].nArrivalTime == nIndex)
			{
				for(int nProcess = nProcessCount - 1; nProcess >= 0; nProcess--)
				{
					aQueue[nProcess + 1] = aQueue[nProcess];
				}
				aQueue[0] = nLoop;
				nProcessCount++;
				bAddMark = TRUE;
			}
		if(bAddMark && aQueue[0] != -1)
			m_ctlAnimation.AddMark(aQueue[0]+1, m_ctlAnimation.m_nPos);
		bAddMark = FALSE;
		for(int nLoop = 0; nLoop < 5; nLoop++)
		{
			if(m_pp[nLoop].bProcessed == FALSE)
				m_pp[nLoop].nTurnaroundTime++;
		}
		if(aQueue[0] == -1)
		{
			m_ctlAnimation.SetIdleMark(m_ctlAnimation.m_nPos);
			m_ctlAnimation.SetPos(m_ctlAnimation.m_nPos + 1);
			m_nTotalBurstTime++;
			m_ctlAnimation.SetRange(0, m_nTotalBurstTime);
			m_ctlAnimation.SetGrid(m_nTotalBurstTime);
```

```
bAddMark = TRUE;                }
                else
                {
                        m_pp[aQueue[nCurProcess]].nBurstTime--;
                        int nBurstTime = m_pp[aQueue[nCurProcess]].nBurstTime;
                        int nProcessNo = aQueue[nCurProcess];
                        for(int nLoop = 0; nLoop < nProcessCount; nLoop++)
                                aQueue[nLoop] = aQueue[nLoop+1];
                        if(nBurstTime == 0)
                        {
                                aQueue[nProcessCount-1] = -1;
                                m_pp[nProcessNo].bProcessed = TRUE;
                                if(aQueue[0] != -1)
                                        bAddMark = TRUE;
                                nProcessCount--;
                        }
                        else
                        {
                                aQueue[nProcessCount-1] = nProcessNo;
                                if(aQueue[0] != nProcessNo)
                                        bAddMark = TRUE;
                        }
                        m_ctlAnimation.SetPos(m_ctlAnimation.m_nPos + 1);
                }
        }
        int nTotalWaitingTime = 0;
        for(int nIndex = 0; nIndex < 5; nIndex++)
        {
                m_pp[nIndex].nTurnaroundTime -= m_pp[nIndex].nArrivalTime;
                m_pp[nIndex].nWaitingTime = m_pp[nIndex].nTurnaroundTime - pp[nIndex].nBurstTime;
                nTotalWaitingTime += m_pp[nIndex].nWaitingTime;
                sprintf(szNumber, "%d", m_pp[nIndex].nWaitingTime);
                m_lstNumbers.SetItemText(nIndex, 4, szNumber);
                sprintf(szNumber, "%d", m_pp[nIndex].nTurnaroundTime);
                m_lstNumbers.SetItemText(nIndex, 3, szNumber);
        }
        double fAverageWaitingTime = double(nTotalWaitingTime) / 5.0;
        sprintf(szNumber, "%0.1f", fAverageWaitingTime);
        m_lstNumbers.SetItemText(5, 4, szNumber);
        memcpy(m_pp, pp, sizeof(pp));
        m_nTotalBurstTime = nTotalBurstTime;
        m_lstNumbers.Invalidate();
}
```

Round Robin Code Flow chart:




Move to next process
If number of processed processes < 5
Yes
Calculate the resultant of total burst time, waiting time, average waiting time, turnaround time.
Add the results to the third and fourth columns of the system interface.

Multi Level Queue Algorithm

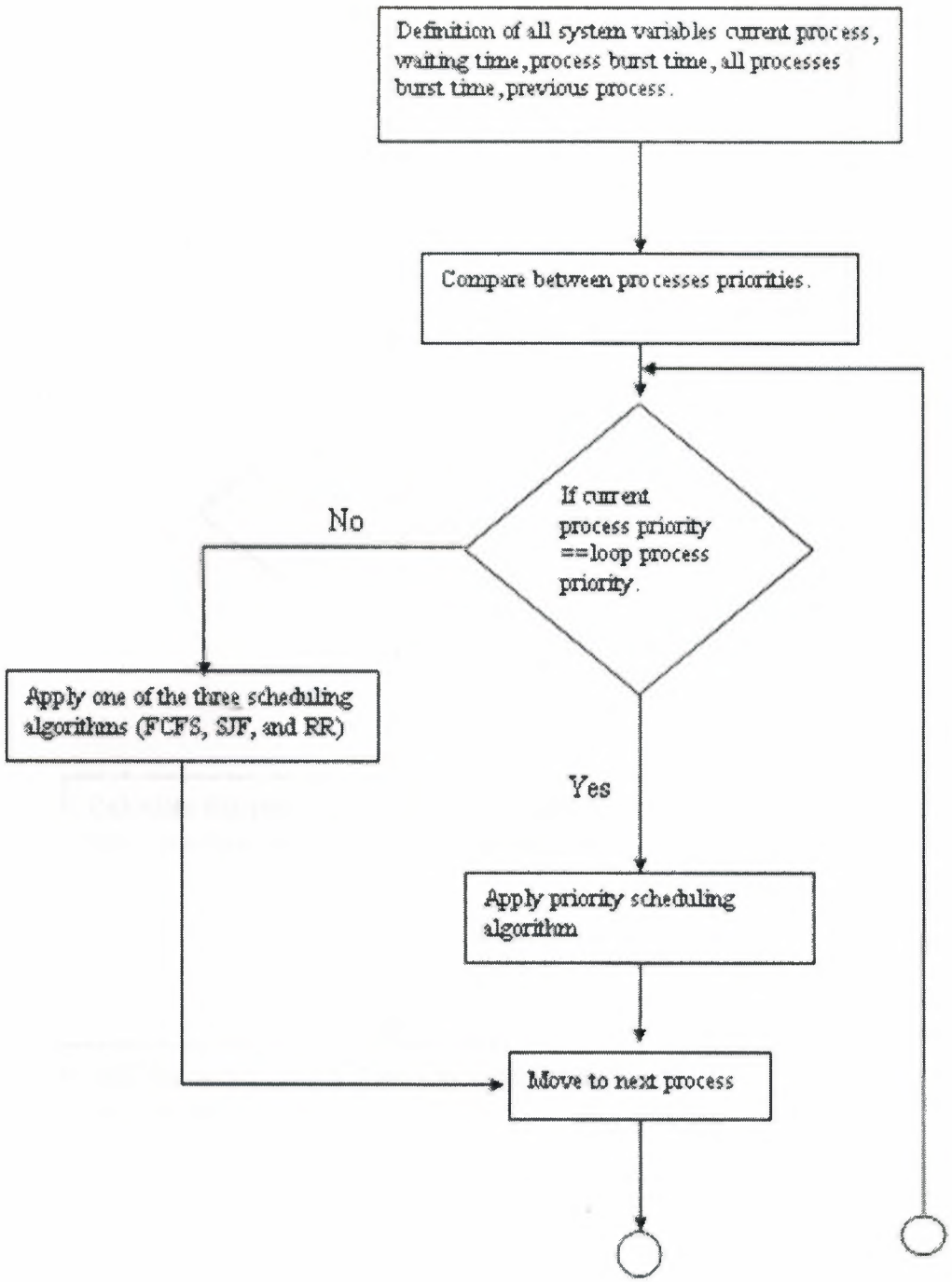
```
void CAlgorithm5Dlg::OnNonPreemptive(void)
{
	PROCESS_PERIODS5 pp[5];
	memcpy(pp, m_pp, sizeof(pp));
	int nArrivalTime = 0;
	CHAR szNumber[10];
	int nTotalBurstTime = m_nTotalBurstTime;
	m_ctlAnimation.SetRange(0, m_nTotalBurstTime);
	m_ctlAnimation.SetPos(0);
	int nCurProcess = 0;
	int nPrevProcess;
	m_pp[0].nWaitingTime = 0;
	int nWaitingTime = 0;
	BOOL bAddMark = FALSE;
	m_ctlAnimation.ClearMarks();
	m_ctlAnimation.ClearIdleMarks();
	switch(m_cboAlgorithm.GetCurSel())
	{
	case 0:
		for(int nLoop = 0; nLoop < 5; nLoop++)
		{
			if(m_pp[nLoop].nArrivalTime == 0 && m_pp[nLoop].nPriority <
m_pp[nCurProcess].nPriority)
				nCurProcess = nLoop;
		}
		m_ctlAnimation.AddMark(nCurProcess+1, 0);
		break;
	case 1:
		for(int nLoop = 0; nLoop < 5; nLoop++)
		{
			if(m_pp[nLoop].nArrivalTime == 0 && (m_pp[nLoop].nPriority <
m_pp[nCurProcess].nPriority || (m_pp[nLoop].nPriority == m_pp[nCurProcess].nPriority &&
m_pp[nLoop].nBurstTime < m_pp[nCurProcess].nBurstTime)))
				nCurProcess = nLoop;
		}
		m_ctlAnimation.AddMark(nCurProcess+1, 0);
		break;
	case 2:
		for(int nLoop = 0; nLoop < 5; nLoop++)
		{
			if(m_pp[nLoop].nArrivalTime == 0 && m_pp[nLoop].nPriority <
m_pp[nCurProcess].nPriority)
				nCurProcess = nLoop;
		}
		break;
	}
	int nInitialProcess = nCurProcess;
	for(int nIndex = 0; nIndex < m_nTotalBurstTime; nIndex++)
	{
		Sleep(300);
		if(nIndex == 0)
		{
```

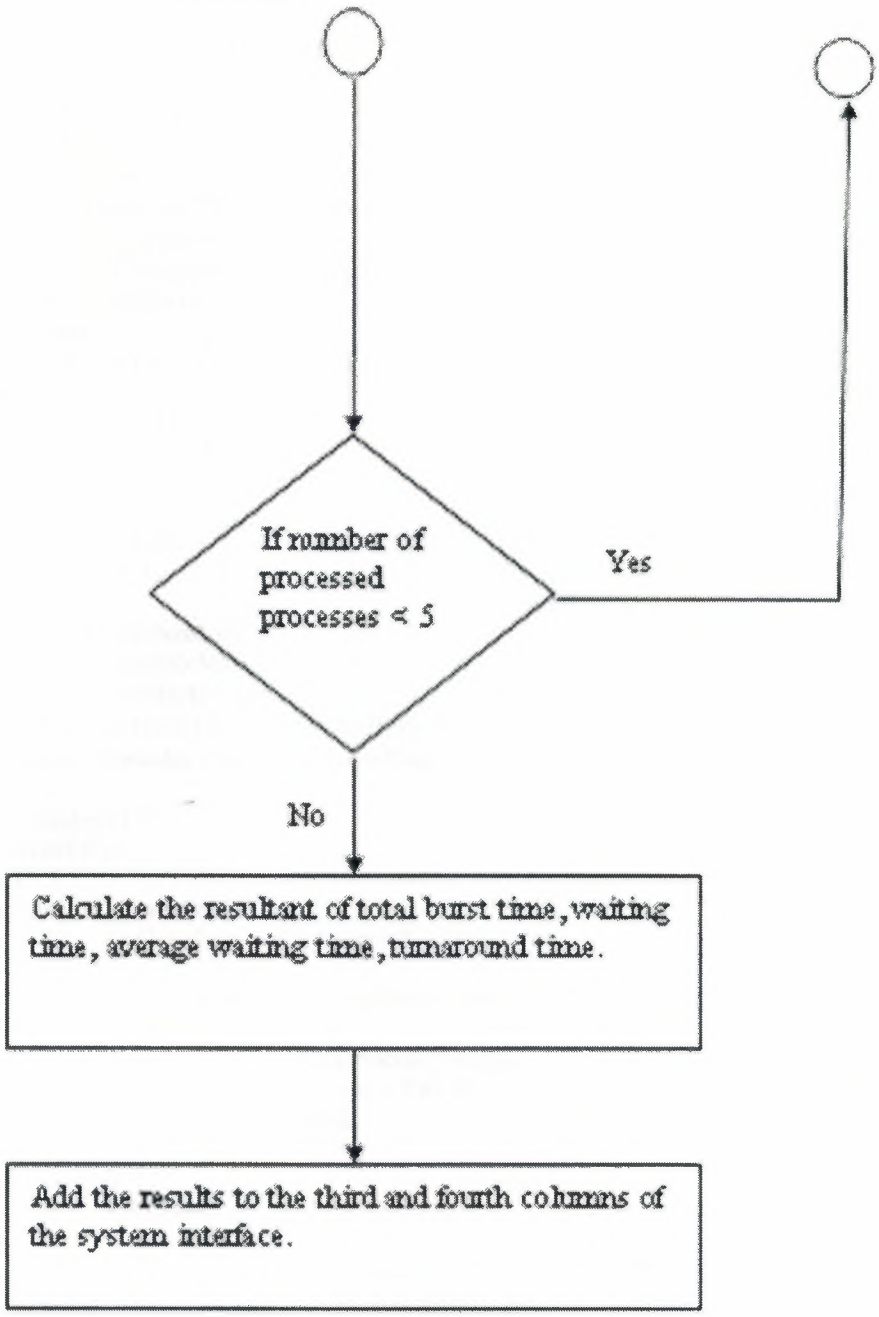
```
nCurProcess = 0;
                        switch(m_cboAlgorithm.GetCurSel())
                        {
                        case 0:
                                for(int nLoop = 0; nLoop < 5; nLoop++)
                                {
                                        if(m_pp[nLoop].nArrivalTime == 0 && m_pp[nLoop].nPriority
< m_pp[nCurProcess].nPriority)
                                                nCurProcess = nLoop;
                                }
                                break;
                        case 1:
                                for(int nLoop = 0; nLoop < 5; nLoop++)
                                {
                                        if(m_pp[nLoop].nArrivalTime == 0 && (m_pp[nLoop].nPriority
< m_pp[nCurProcess].nPriority || (m_pp[nLoop].nPriority == m_pp[nCurProcess].nPriority &&
m_pp[nLoop].nBurstTime < m_pp[nCurProcess].nBurstTime)))
                                                nCurProcess = nLoop;
                                }
                                break;
                        case 2:
                                break;
                        }
                        nPrevProcess = nCurProcess;
                        m_pp[nCurProcess].nWaitingTime = 0;
                }
                if(m_cboAlgorithm.GetCurSel() == 2)
                {
                        nPrevProcess = nCurProcess;
                        for(int nLoop = 0; nLoop < 5; nLoop++)
                        {
                                if(m_pp[nLoop].bProcessed == FALSE && nLoop != nCurProcess &&
m_pp[nLoop].nArrivalTime <= nIndex && (m_pp[nLoop].nPriority < m_pp[nCurProcess].nPriority ||
(m_pp[nLoop].nPriority == m_pp[nCurProcess].nPriority && m_pp[nLoop].nQuantumCount <
m_pp[nCurProcess].nQuantumCount)))
                                {
                                        bAddMark = TRUE;
                                        nCurProcess = nLoop;
                                }
                        }
                        if(nPrevProcess == nCurProcess)
                        {
                                nCurProcess = nInitialProcess;
                                bAddMark = TRUE;
                        }
                }
                if(bAddMark)
                {
                        m_ctlAnimation.AddMark(nCurProcess+1, m_ctlAnimation.m_nPos);
                        bAddMark = FALSE;
                }
                m_pp[nCurProcess].nQuantumCount++;
                m_pp[nCurProcess].nBurstTime--;
                for(int nLoop = 0; nLoop < 5; nLoop++)
                {
```

```
if(m_pp[nLoop].bProcessed == FALSE)
                        m_pp[nLoop].nTurnaroundTime++;                    }
            nWaitingTime++;
            m_ctlAnimation.SetPos(m_ctlAnimation.m_nPos + 1);
            if(m_pp[nCurProcess].nBurstTime == 0 && nIndex < m_nTotalBurstTime - 1)
            {
                m_pp[nCurProcess].bProcessed = TRUE;
                nCurProcess = 0;
                BOOL bFound = FALSE;
                for(int nLoop = 0; nLoop < 5; nLoop++)
                {
                    if(m_pp[nLoop].bProcessed == FALSE && m_pp[nLoop].nArrivalTime
<= nIndex + 1)
                    {
                        nCurProcess = nLoop;
                        nInitialProcess = nLoop;
                        bFound = TRUE;
                        break;
                    }
                }
                if(!bFound)
                {
                    for(; nIndex < m_nTotalBurstTime; nIndex++)
                    {
                        int nLoop;
                        for(nLoop = 0; nLoop < 5; nLoop++)
                        {
                            if(m_pp[nLoop].bProcessed == FALSE &&
m_pp[nLoop].nArrivalTime <= nIndex + 1)
                            {
                                nCurProcess = nLoop;
                                nInitialProcess = nLoop;
                                break;
                            }
                        }
                        if(nCurProcess == nLoop)
                            break;
                        nWaitingTime++;
                        m_ctlAnimation.SetIdleMark(m_ctlAnimation.m_nPos);
                        m_ctlAnimation.SetPos(m_ctlAnimation.m_nPos + 1);
                        m_nTotalBurstTime++;
                        m_ctlAnimation.SetRange(0, m_nTotalBurstTime);
                        m_ctlAnimation.SetGrid(m_nTotalBurstTime);
                    }
                }
                switch(m_cboAlgorithm.GetCurSel())
                {
                case 0:
                    for(int nLoop = 0; nLoop < 5; nLoop++)
                    {
                        if(m_pp[nLoop].bProcessed == FALSE &&
m_pp[nLoop].nArrivalTime <= nIndex + 1 &&
                        m_pp[nLoop].nPriority < m_pp[nCurProcess].nPriority)
                            nCurProcess = nLoop;
                    }
```

```
m_ctlAnimation.AddMark(nCurProcess+1, m_ctlAnimation.m_nPos);
                                        break;
                                case 1:
                                        for(int nLoop = 0; nLoop < 5; nLoop++)
                                        {
                                                if(m_pp[nLoop].bProcessed == FALSE &&
m_pp[nLoop].nArrivalTime <= nIndex + 1 &&
                                                (m_pp[nLoop].nPriority < m_pp[nCurProcess].nPriority ||
(m_pp[nLoop].nPriority == m_pp[nCurProcess].nPriority && m_pp[nLoop].nBurstTime <
m_pp[nCurProcess].nBurstTime)))
                                                        nCurProcess = nLoop;
                                        }
                                        m_ctlAnimation.AddMark(nCurProcess+1, m_ctlAnimation.m_nPos);
                                        break;
                                case 2:
                                        break;
                                }
                        }
                }
                int nTotalWaitingTime = 0;
                for(int nIndex = 0; nIndex < 5; nIndex++)
                {
                        m_pp[nIndex].nTurnaroundTime -= m_pp[nIndex].nArrivalTime;
                        m_pp[nIndex].nWaitingTime = m_pp[nIndex].nTurnaroundTime - pp[nIndex].nBurstTime;
                        nTotalWaitingTime += m_pp[nIndex].nWaitingTime;
                        sprintf(szNumber, "%d", m_pp[nIndex].nWaitingTime);
                        m_lstNumbers.SetItemText(nIndex, 5, szNumber);
                        sprintf(szNumber, "%d", m_pp[nIndex].nTurnaroundTime);
                        m_lstNumbers.SetItemText(nIndex, 4, szNumber);
                }
                double fAverageWaitingTime = double(nTotalWaitingTime) / 5.0;
                sprintf(szNumber, "%0.1f", fAverageWaitingTime);
                m_lstNumbers.SetItemText(5, 5, szNumber);
                memcpy(m_pp, pp, sizeof(pp));
                m_nTotalBurstTime = nTotalBurstTime;
                m_lstNumbers.Invalidate();
}
```

Multilevel Queue Code Flow chart:




If number of processed processes < 5
Yes
No
Calculate the resultant of total burst time, waiting time, average waiting time, turnaround time.
Add the results to the third and fourth columns of the system interface.

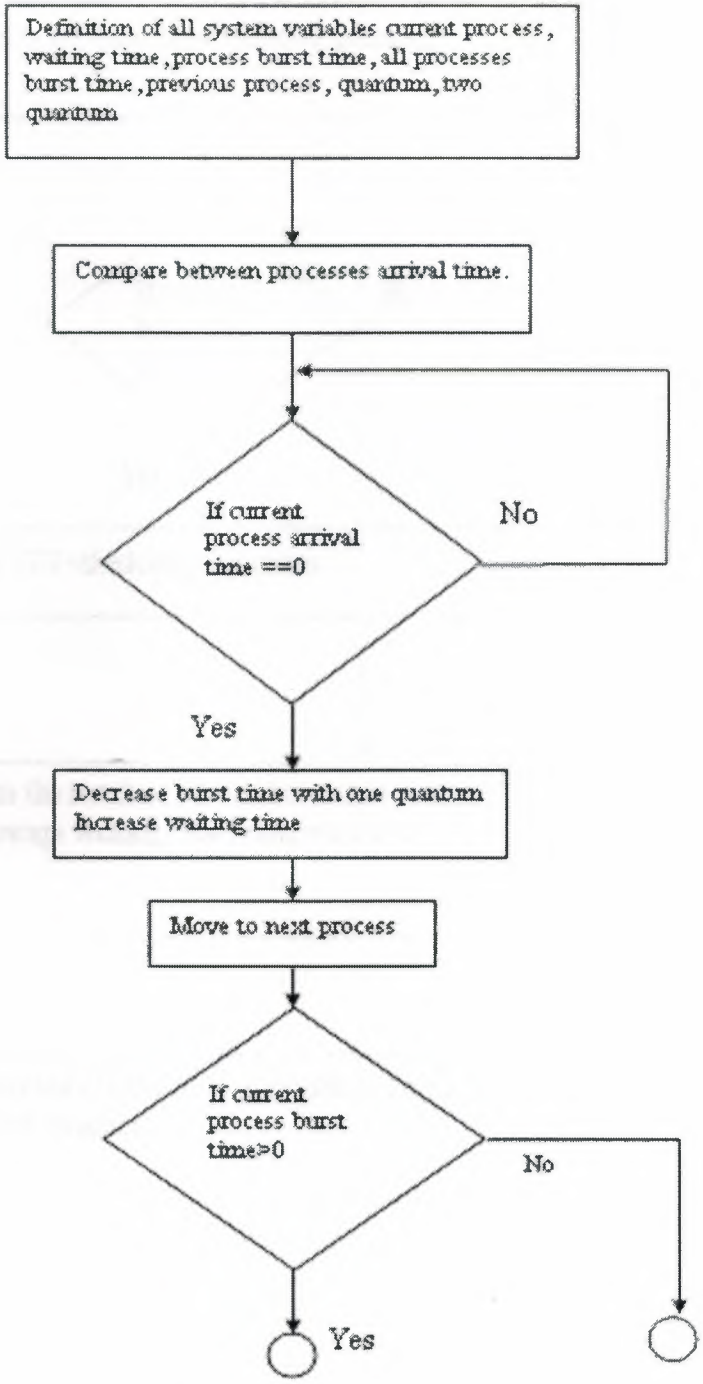
Multi-level Feedback Algorithm

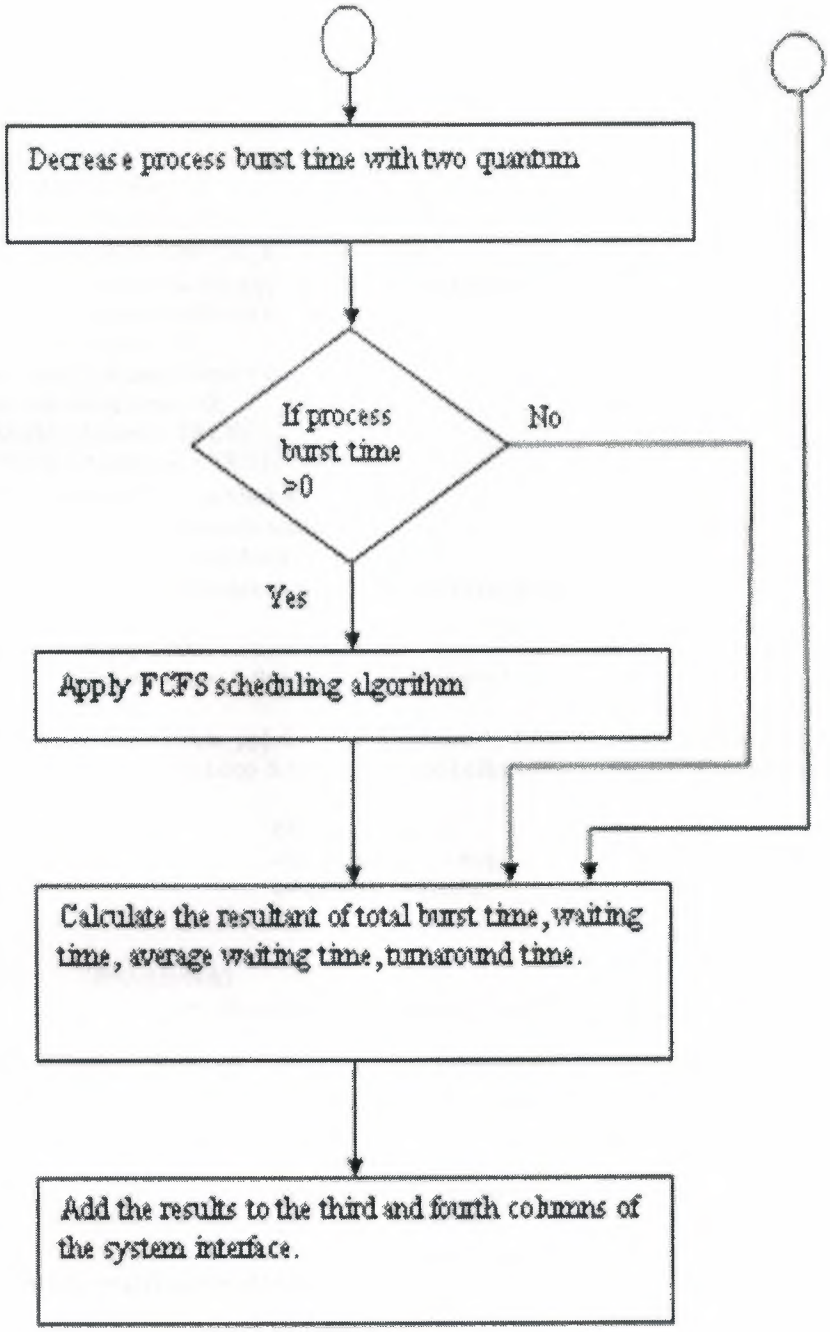
```
void CAlgorithm6Dlg::OnNonPreemptive(void)
{
	PROCESS_PERIODS6 pp[5];
	int nArrivalTime = 0;
	CHAR szNumber[10];
	int nTotalBurstTime = m_nTotalBurstTime;
	CopyMemory(pp, m_pp, sizeof(pp));
	m_ctlAnimation.SetRange(0, m_nTotalBurstTime);
	m_ctlAnimation.SetPos(0);
	int nCurProcess = 0;
	m_pp[0].nWaitingTime = 0;
	int nWaitingTime = 0;
	BOOL bFound = FALSE;
	BOOL bAddMark = FALSE;
	int nRound = 0;
	int nStep = 1;
	BOOL bFCFS = FALSE;
	int nMaxQuantum = 1;
	int nQuantum;
	m_ctlAnimation.ClearMarks();
	m_ctlAnimation.ClearIdleMarks();
	m_ctlAnimation.AddMark(1, 0);
	qsort((void*)m_pp, (size_t)5, sizeof(PROCESS_PERIODS6), compare6);
	for(int nIndex = 0; nIndex < m_nTotalBurstTime; nIndex++)
	{
		Sleep(300);
		if(bFCFS)
		{
			bFound = FALSE;
			for(int nLoop = 0; nLoop < 5; nLoop++)
			{
				if(m_pp[nLoop].bProcessed == FALSE)
				{
					nCurProcess = nLoop;
					bFound = TRUE;
					break;
				}
			}
			if(bFound)
			{
				nQuantum = m_pp[nCurProcess].nBurstTime;
				m_pp[nCurProcess].nBurstTime-=nQuantum;
				m_pp[nCurProcess].nQuantum+=nQuantum;
				nIndex += (nQuantum - 1);
				for(int nLoop = 0; nLoop < 5; nLoop++)
				{
					if(m_pp[nLoop].bProcessed == FALSE)
						m_pp[nLoop].nTurnaroundTime+=nQuantum;
				}
				if(m_pp[nCurProcess].nBurstTime == 0)
				{
```

```
                                m_pp[nCurProcess].bProcessed = TRUE;
                        }
                        m_ctlAnimation.AddMark(nCurProcess+1, m_ctlAnimation.m_nPos);
                        m_ctlAnimation.SetPos(m_ctlAnimation.m_nPos + nQuantum);
                        continue;
                }
        }
        else
        {
                bFound = FALSE;
                for(int nLoop = 0; nLoop < 5; nLoop++)
                {
                        if(m_pp[nLoop].bProcessed == FALSE && nRound ==
m_pp[nLoop].nQuantum)
                        {
                                nCurProcess = nLoop;
                                bFound = TRUE;
                                break;
                        }
                }
                if(bFound)
                {
                        if(nMaxQuantum > m_pp[nCurProcess].nBurstTime)
                                nQuantum = m_pp[nCurProcess].nBurstTime;
                        else
                                nQuantum = nMaxQuantum;
                        m_pp[nCurProcess].nBurstTime-=nQuantum;
                        m_pp[nCurProcess].nQuantum++;
                        nIndex += (nQuantum - 1);
                        for(int nLoop = 0; nLoop < 5; nLoop++)
                        {
                                if(m_pp[nLoop].bProcessed == FALSE)
                                        m_pp[nLoop].nTurnaroundTime+=nQuantum;
                        }
                        if(m_pp[nCurProcess].nBurstTime == 0)
                        {
                                m_pp[nCurProcess].bProcessed = TRUE;
                        }
                        m_ctlAnimation.AddMark(nCurProcess+1, m_ctlAnimation.m_nPos);
                        m_ctlAnimation.SetPos(m_ctlAnimation.m_nPos + nQuantum);
                }
        }
        if(nStep == 5)
        {
                nStep = 1;
                nRound++;
                nMaxQuantum++;
                if(nRound > 1)
                {
                        bFCFS = TRUE;
                }
        }
        else
        {
                nStep++;
        }
```

```
        }
        int nTotalWaitingTime = 0;
        for(int nIndex = 0; nIndex < 5; nIndex++)
        {
                m_pp[nIndex].nTurnaroundTime -= m_pp[nIndex].nArrivalTime;
                m_pp[nIndex].nWaitingTime = m_pp[nIndex].nTurnaroundTime - pp[nIndex].nBurstTime;
                nTotalWaitingTime += m_pp[nIndex].nWaitingTime;
                sprintf(szNumber, "%d", m_pp[nIndex].nWaitingTime);
                m_lstNumbers.SetItemText(nIndex, 4, szNumber);
                sprintf(szNumber, "%d", m_pp[nIndex].nTurnaroundTime);
                m_lstNumbers.SetItemText(nIndex, 3, szNumber);
        }
        double fAverageWaitingTime = double(nTotalWaitingTime) / 5.0;
        sprintf(szNumber, "%0.1f", fAverageWaitingTime);
        m_lstNumbers.SetItemText(5, 4, szNumber);
        memcpy(m_pp, pp, sizeof(pp));
        m_nTotalBurstTime = nTotalBurstTime;
        m_lstNumbers.Invalidate();
}
```

Multilevel Feedback Queue Code Flow chart:



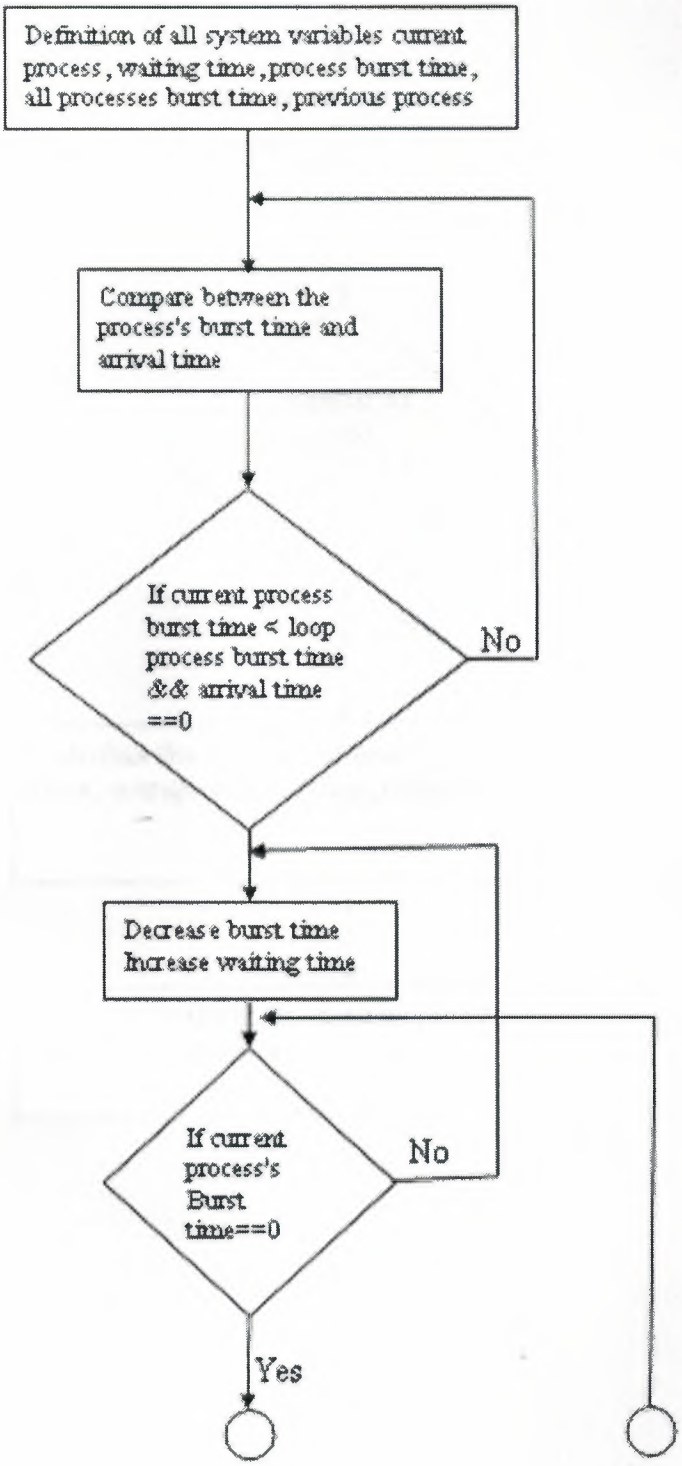

Decrease process burst time with two quantum
If process burst time >0
No
Yes
Apply FCFS scheduling algorithm
Calculate the resultant of total burst time, waiting time, average waiting time, turnaround time.
Add the results to the third and fourth columns of the system interface.

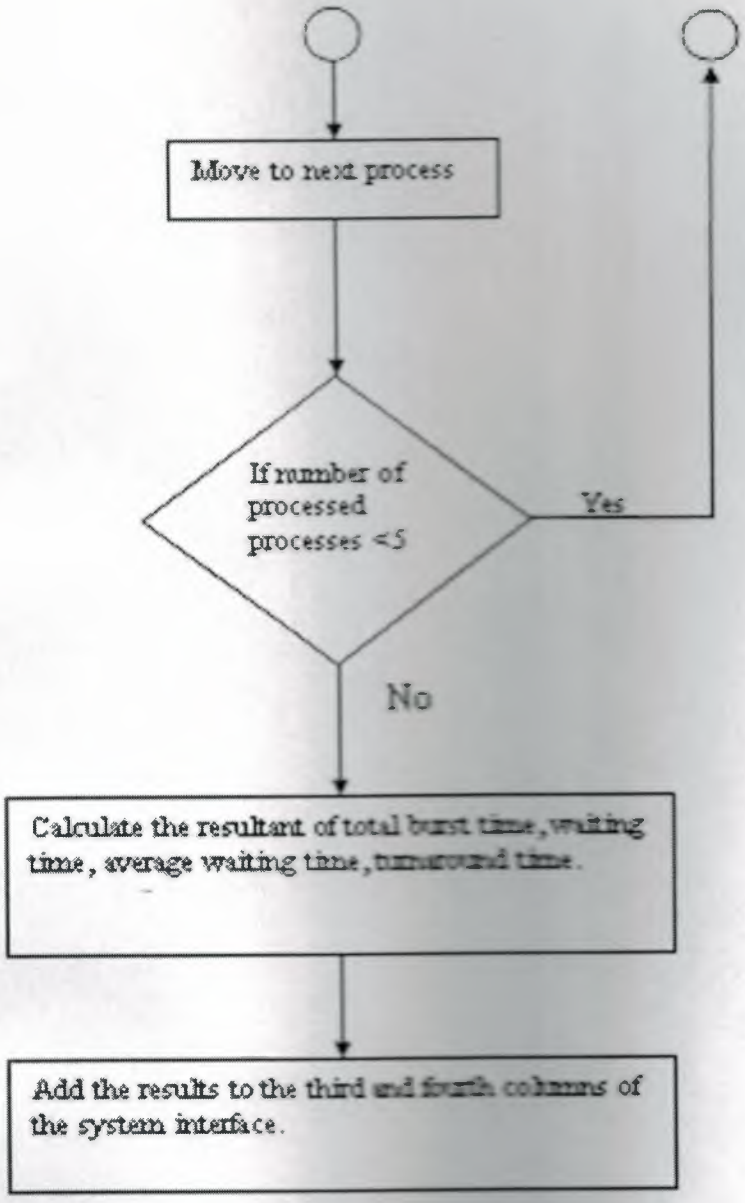
Longest Job First Algorithm

```
void CAlgorithm7Dlg::OnPrimitive(void)
{
	PROCESS_PERIODS7 pp[5];
	memcpy(pp, m_pp, sizeof(pp));
	int nArrivalTime = 0;
	CHAR szNumber[10];
	int nTotalBurstTime = m_nTotalBurstTime;
	m_ctlAnimation.SetRange(0, m_nTotalBurstTime);
	m_ctlAnimation.SetPos(0);
	int nCurProcess = 0;
	m_pp[0].nWaitingTime = 0;
	int nWaitingTime = 0;
	BOOL bFound = TRUE;
	BOOL bAddMark = TRUE;
	m_ctlAnimation.ClearMarks();
	m_ctlAnimation.ClearIdleMarks();
	//m_ctlAnimation.AddMark(1, 0);
	for(int nIndex = 0; nIndex < m_nTotalBurstTime; nIndex++)
	{
		Sleep(300);
		for(int nLoop = 0; nLoop < 5; nLoop++)
		{
			if(m_pp[nLoop].bProcessed == FALSE && m_pp[nLoop].nArrivalTime <=
nIndex && nCurProcess != nLoop && m_pp[nLoop].nBurstTime > m_pp[nCurProcess].nBurstTime)
			{
				bFound = TRUE;
				nCurProcess = nLoop;
				bAddMark = TRUE;
			}
		}
		if(bAddMark)
			m_ctlAnimation.AddMark(nCurProcess+1, m_ctlAnimation.m_nPos);
		if(!bFound)
		{
			for(; nIndex < m_nTotalBurstTime; nIndex++)
			{
				int nLoop;
				for(nLoop = 0; nLoop < 5; nLoop++)
				{
					if(m_pp[nLoop].bProcessed == FALSE &&
m_pp[nLoop].nArrivalTime <= nIndex + 1)
					{
						nCurProcess = nLoop;
						break;
					}
				}
				m_ctlAnimation.SetIdleMark(m_ctlAnimation.m_nPos);
				m_ctlAnimation.SetPos(m_ctlAnimation.m_nPos + 1);
				m_nTotalBurstTime++;
				m_ctlAnimation.SetRange(0, m_nTotalBurstTime);
				m_ctlAnimation.SetGrid(m_nTotalBurstTime);
				if(nCurProcess == nLoop)
				{
```

```
                                        bFound = TRUE;
                                        bAddMark = TRUE;
                                        break;
                                }
                        }
                }
                else
                {
                        m_pp[nCurProcess].nBurstTime--;
                        m_ctlAnimation.SetPos(m_ctlAnimation.m_nPos + 1);
                        for(int nLoop = 0; nLoop < 5; nLoop++)
                        {
                                if(m_pp[nLoop].bProcessed == FALSE)
                                        m_pp[nLoop].nTurnaroundTime++;
                        }
                        if(m_pp[nCurProcess].nBurstTime == 0 && nIndex < m_nTotalBurstTime - 1)
                        {
                                bFound = FALSE;
                                m_pp[nCurProcess].bProcessed = TRUE;
                                for(int nLoop = 0; nLoop < 5; nLoop++)
                                {
                                        if(m_pp[nLoop].bProcessed == FALSE &&
m_pp[nLoop].nArrivalTime <= nIndex + 1)
                                        {
                                                nCurProcess = nLoop;
                                                bFound = TRUE;
                                                bAddMark = TRUE;
                                                break;
                                        }
                                }
                        }
                        else
                        {
                                bAddMark = FALSE;
                        }
                }int nTotalWaitingTime = 0;
        for(int nIndex = 0; nIndex < 5; nIndex++)
        {
                m_pp[nIndex].nTurnaroundTime -= m_pp[nIndex].nArrivalTime;
                m_pp[nIndex].nWaitingTime = m_pp[nIndex].nTurnaroundTime - pp[nIndex].nBurstTime;
                nTotalWaitingTime += m_pp[nIndex].nWaitingTime;
                sprintf(szNumber, "%d", m_pp[nIndex].nWaitingTime);
                m_lstNumbers.SetItemText(nIndex, 4, szNumber);
                sprintf(szNumber, "%d", m_pp[nIndex].nTurnaroundTime);
                m_lstNumbers.SetItemText(nIndex, 3, szNumber);
        }
        double fAverageWaitingTime = double(nTotalWaitingTime) / 5.0;
        sprintf(szNumber, "%0.1f", fAverageWaitingTime);
        m_lstNumbers.SetItemText(5, 4, szNumber);
        memcpy(m_pp, pp, sizeof(pp));
        m_nTotalBurstTime = nTotalBurstTime;
        m_lstNumbers.Invalidate();
}
```

LJF Code Flow Chart:




Move to next process
If number of processed processes <5
Yes
No
Calculate the resultant of total burst time, waiting time, average waiting time, turnaround time.
Add the results to the third and fourth columns of the system interface.