A SIMPLE SCHEDULER GENERATOR TOOL

BY

YUNGAH PARK

Bachelor of Science

Pohang University of Science and Technology

Pohang, Korea

1992

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfilment of
the requirements for
the Degree of
MASTER OF SCIENCE
December 1997

A SIMPLE SCHEDULER GENERATOR TOOL

Thesis Approved:

Mansur Samadzadeh

Thesis Advisor

Blayne E. Mayfield

Jacques E. LaFrance

Wayne B. Powell

Dean of the Graduate College

PREFACE

The CPU scheduler is a basic component that supports multiprogramming in operating systems. Many scheduling algorithms have been introduced to improve the performance of systems in terms of processor utilization. The best scheduling algorithm for each system may be different based on the specific circumstances of that system. Object-oriented programming, which facilitates reusability and extendibility, has become quite popular for many computer applications. This thesis work involved the design and implementation of a simple scheduler generator tool. The scheduler generator tool simulated several scheduling algorithms by using object-oriented programming as the implementation language. The various components of the environment (i.e., the simulated operating system) used for CPU scheduling were developed as objects, and the scheduling algorithms were implemented using the techniques and characteristics of object-oriented programming. For a scheduling algorithm selected, the scheduler generator tool can compute performance parameters such as turnaround time, waiting time, and CPU utilization. The tool can be used for investigating the relative merits of scheduling algorithms.

ACKNOWLEDGMENTS

I would like to express special appreciation to my advisor Dr. Mansur H. Samadzadeh. He provided essential guidance and inspiration through my thesis work. Dr. Samadzadeh continued to spend endless hours reviewing my work and offering suggestions for further refinement.

I would like to thank my other committee members, Drs Blayne E. Mayfield and Jacques LaFrance. Their time and effort are greatly appreciated.

Finally, I would like to express my sincere thanks to my family for their continued support. They helped me throughout my MS program. I couldn't have done it without their continued love and support.

TABLE OF CONTENTS

Chapter

Chapter

LIST OF FIGURES

Figure

Figure

LIST OF TABLES

TABLE

CHAPTER I

INTRODUCTION

Multiprogramming and time sharing systems, which were introduced to improve the overall performance of computer systems, are the central themes of modern operating systems [Silberschatz and Galvin 94]. The basic objective of multiprogramming is to keep the CPU busy executing processes as much as possible. In multiprogramming, several programs are kept simultaneously in memory by switching the CPU among the processes, thus CPU utilization is increased. When a running process has to wait, the CPU is switched to another process and executes that process.

The part of an operating system that deals with the decision as to which process in the ready queue is to be executed next, is called the CPU scheduler [Tanenbaum 94]. The scheduler is one of the basic mechanisms to support multiprogramming together with virtual memory. To support multiprogramming, when the CPU is switched to another process, the scheduler must save the information of the old process and load the new process' information into CPU registers (i.e., a context switch must take place). Also, the scheduler changes the state of the running process to either ready or blocked, selects a new process that is ready, and changes the new process's state to running. The strategy which specifies the execution order of the processes in the ready queue, is the scheduling algorithm.

There are many scheduling algorithms implementing various properties and policies. Research into developing more efficient scheduling algorithms continues. The best scheduling algorithm for each computing system may be different based on the specific circumstances of that system. Thus the criteria used to determine the best algorithm for a given system depend on the characteristic of that system. For example, if the system is a real time system, the criteria may focus on guaranteeing that the processes complete within the defined time constraints [Nutt 92].

Several different methods can be used to evaluate scheduling algorithms [Silberschatz and Galvin 94]: deterministic modelling, queueing models, and simulation. Among them simulation is used more often because it generates a more accurate evaluation. In simulation, the components of a system would be represented as data structures. As the value of a variable representing the CPU clock is increased, the system state would be changed and the parameters indicating the performance of various scheduling algorithms would be computed.

Since the late 1980's, the object-oriented approach based on data abstraction has become quite popular in computer application areas such as database, graphics, and simulation [Ghezzi et al. 91]. This approach is a paradigm that views a software system as a collection of interacting objects which are composed of their states (i.e., attributes) and behaviors [Sommervile 96]. The attributes are represented as data variables and the behaviors are implemented by the operations within an object. By adding the mechanisms of data encapsulation, inheritance, and message passing to the idea of data abstraction, the object-oriented approach is completed [Budd 91].

In the object-oriented approach, objects are handled as independent entities. Changing and/or adding object attributes and object operations can be done at any time without affecting other objects. Objects that have been already developed can be reused in other system designs. Also, the attributes and operations of an object can be reused in a subclass or other objects through inheritance. So, the object-oriented approach facilitates reusability and extendibility of software. Furthermore, the understandability and maintainability of a system can be improved because the object-oriented approach provides a clear mapping between real-world objects and software objects [Sommervile 96].

Operations or services held within each object in a system make up the functionality of the system. The system functions through communication among objects implemented by calling services offered by other objects (rather than by using shared data). So this approach reduces the possibility of unexpected changes to shared data.

The main goal of this thesis work was to develop a simple scheduler generator tool for operating systems by using the object-oriented approach. Several different kinds of scheduling algorithms were simulated and evaluated using the tool. To simulate a scheduling algorithm, the various components of a typical operating system that are related to CPU scheduling were developed as objects. The algorithms were implemented by the techniques and characteristics of object-oriented programming.

The rest of this thesis report is organized as follows: Chapter II provides a literature review about process scheduling, scheduling algorithms, and object-oriented programming. Chapter III discusses design and implementation issues. Evaluation of the tool is included in Chapter IV. Finally, Chapter V contains the summary and future work.

CHAPTER II

LITERATURE REVIEW

2.1 Process Scheduling

A process, which is usually referenced to as a program in execution, is a widely used unit of work in modern operating systems [Silberschatz and Galvin 94]. A process can be executed when the resources required by the process are allocated to it [Tanenbaum 94]. A process may be *running* (using the CPU), *ready* (waiting for the CPU), or *blocked* (waiting for I/O completion) while it is in the system [Nutt 92].

In an operating system, each process is represented by a PCB (process control/context block) that contains all the relevant information about the process. The fields of a PCB may be different from system to system. Figure 1 gives the common fields of a PCB. When the CPU is switched from one process to another, the first process' PCB is saved so that it can be restarted later. To execute a process, various scheduling queues that the operating system uses to select a process (such as ready queue, job queue, and blocked queue) are required. These queues are important parts of schedulers and every process must migrate through them to get resources. For example, to get I/O service, processes must wait in a blocked queue, and, to use the CPU, processes must wait in a ready queue.

In scheduling, if the memory required by a selected process is available and the current number of processes is less than the maximum degree of multiprogramming, the process can be loaded from disk into main memory for execution. At this time, the selected process migrates from the job queue to the ready queue, and the state of this process becomes *ready*. The ready queue contains the *ready* processes that are kept in main memory and waiting to be dispatched to the CPU.

Process Number
Priority
Program Counter
Process State
Stack Pointer
Registers
Memory Allocation
Status of Open Files
Time Process Started
CPU Time Used
. . .

Figure 1. Some common PCB fields

The ready queue may be implemented in a number of ways depending on each scheduling algorithm's policy. The CPU scheduler selects a process from the ready queue to allocate to the CPU by executing the scheduling algorithm utilized. After the *running* process is executed for a certain amount of time, the process may be completed, placed in a blocked queue for I/O service, or returned to the ready queue to wait for further service [Lister and Eager 93]. A process waiting in a blocked queue would be returned to the ready queue for further CPU bursts after the completion of the I/O service. Figure 2 describes the general process scheduling model. Since each process typically consists of a sequence of CPU and I/O bursts, processes repeat the cycle as shown in Figure 2.

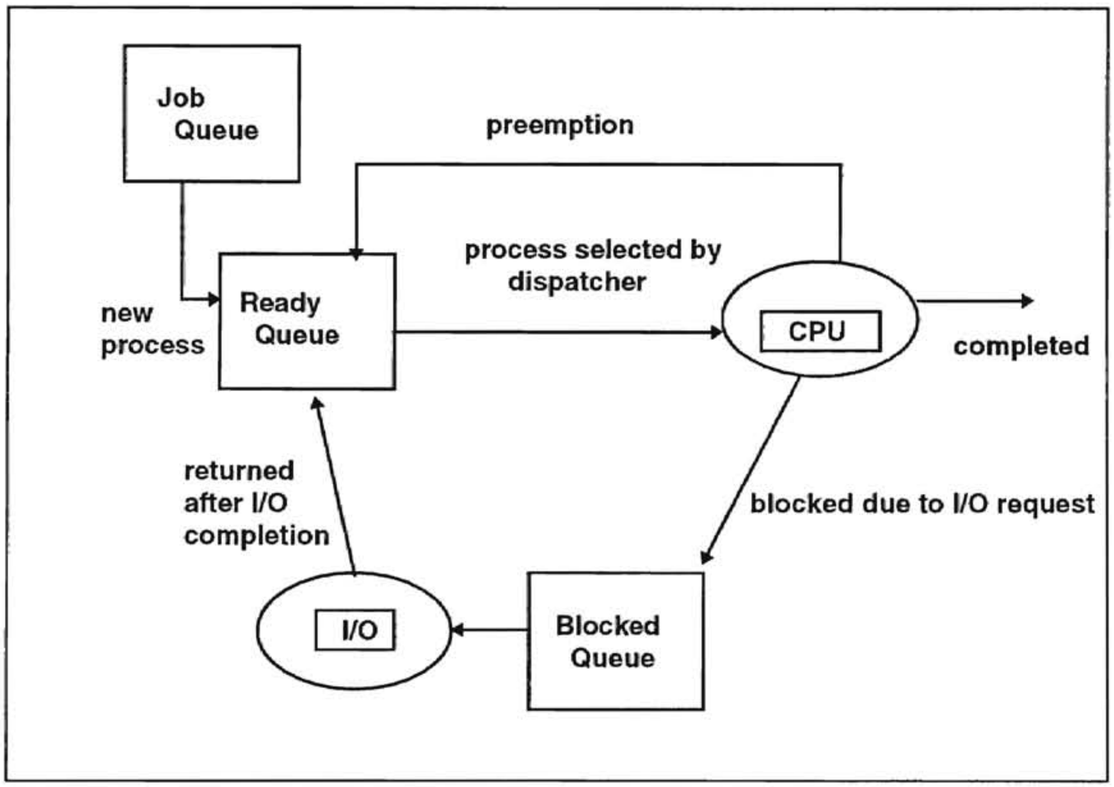

Figure 2. Process Scheduling Model

Some scheduling algorithms do not allow direct transitions from the CPU running state to the ready queue. These are called nonpreemptive scheduling algorithms, with the alternative being preemptive scheduling algorithms. In nonpreemptive scheduling algorithms, once the CPU is allocated to a process, the process can run continually until it voluntarily releases the CPU. The CPU is switched to another process only when the current running process is terminated or blocks itself.

Nonpreemptive scheduling algorithms are easy and inexpensive to implement because no extra hardware and methods are necessary (since the scheduler does not need to forcefully remove a running process from the CPU by a clock interrupt). Sometimes nonpreemptive scheduling algorithms are not suitable for interactive systems (such as time sharing systems) that focus on providing a fair share of the CPU to each process [Tanenbaum 94] [Silberschatz and Galvin 94] [Nutt 92]. On the other hand, preemptive scheduling can lead to race conditions and process synchronization problems when multiple processes access shared data. The reason being that interrupts can occur at any instant unpredictably. Sophisticated methods used by operating systems, such as semaphores and monitors, are needed to solve these problems [Tanenbaum 94].

2.2 Criteria for Scheduling Algorithms

When CPU scheduling algorithms are compared to determine which one is best for a system, the following performance factors are usually considered [Silberschatz and Galvin 94] [Tanenbaum 94].

- CPU Utilization: This factor indicates how busy the CPU is, with a range of 0 to 100 percent. The target is to maximize this value.
- Throughput: This factor indicates the number of processes that are completed per some uniform time interval. The target is to maximize the throughput.

- Waiting Time: This is the amount of time a process spends waiting to use the CPU in the ready queue. The target is to minimize this value.
- Turnaround Time: This is the amount of time it takes to complete a process from its arrival in the ready queue to its departure from the system. So this is the sum of the waiting time and the processing time of a process. The target is to minimize this value.
- Response Time: This is the amount of time that it takes to produce the first response for a process from its arrival in the ready queue. This is considered a more important criterion than turnaround time for interactive systems. The target is to minimize this value.

In general, it may be considered desirable to optimize the average value of each factor, however, the overall goals of the systems must be considered. For example for interactive systems, which require each process' equitable share of the CPU, it is more advantageous to minimize the maximum response time than to minimize the average response time [Tanenbaum 94] [Silberschatz and Galvin 94].

2.3 Scheduling Algorithms

There are many scheduling algorithms which implement various policies to decide the execution order of the processes in the ready queue. The following subsections describe several specific algorithms that are widely used.

2.3.1 First Come First Served (FCFS)

In the First Come First Served (FCFS) algorithm, the order of processes in the ready queue is assigned according to the time each process last requested the CPU. The process that requested the CPU first is executed first. This algorithm is easy to implement since a FIFO queue is used as the ready queue. An incoming process from the job queue to the ready queue is inserted at the tail of the ready queue, and the CPU is switched to the process at the head of the ready queue. When a long process is allocated to the CPU, other shorter processes must be wait for a relatively long time. So the FCFS algorithm sometimes does not satisfy criteria such as minimizing the average waiting time or the average turnaround time [Nutt 92] [Silberschatz and Galvin 94]. Also, the FCFS scheduling algorithm does not allow preemption of the CPU. As a result, this algorithms is rarely used for operating systems [Nutt 92].

2.3.2 Shortest Job Fist (SJF)

The process which has the shortest length for the next CPU burst is allocated to the CPU first in the Shortest Job First (SJF) algorithm. The ready queue is ordered according to the lengths of the next CPU bursts required by each process. If multiple processes have the same length, they are ordered FCFS. The SJF algorithm provides the optimal average waiting time and average turnaround time [Tanenbaum 94] [Silberschatz and Galvin 94].

Although SJF algorithm satisfies some criteria minimizing the average turnaround time, the average waiting time, and the average response time, it is in general difficult to know or estimate the length of the next CPU burst for interactive processes. The SJF algorithm is especially suitable for batch systems in which one can acquire the length of the CPU burst from job descriptions [Lister and Eager 93] [Tanembaum 94]. For interactive systems, the length of the next CPU burst for a process can be estimated using the previous behaviour of that process and exponential averaging [Silberschatz and Galvin 94].

2.3.3 Priority

In the priority scheduling algorithm, the ready queue is ordered by the processes' assigned priority; the process with the highest priority is allocated to the CPU first. If multiple processes are assigned the same priority, FCFS scheduling is used to break the tie. Priorities can be assigned internally by the operating system or externally by user identification to accomplish the performance goals of the system. Some measurable attributes such as time limits, the number of open file, and the memory requirements of the processes can be used for internally assigned priorities. Users (i.e., process owners) can also control the priorities based on the importance of each process, the social and political factors, and so on. The SJF algorithm is a special example of priority scheduling algorithms. In the SJF algorithm, the length of the next CPU burst is used by the scheduler to internally compute the priority of a process.

A modification of the SJF algorithm as a priority algorithm is to allow the CPU to be preempted. In the general preemptive case, when a new process with a higher priority than the running process enters the ready queue, the new process is allocated to the CPU (i.e., the CPU is preempted). In the case of the preemptive SJF algorithm, this preemption will occur if a new process with a shorter next CPU burst than the remaining CPU burst of the running process arrives. Another modification to the SJF algorithm, to prevent the low-priority processes from being delayed indefinitely or starving, is to use the aging technique [Silberschatz and Galvin 94].

2.3.4 Round Robin (RR)

The Round Robin (RR) scheduling is developed to provide fast response to requests in interactive systems and time sharing system. Since RR was used in CTSS (i.e., the earliest time sharing system), the RR algorithm including its several variations is one of the most widely used scheduling algorithms [Lister and Eager 93]. Each process is allocated to the CPU for a fixed time interval called the time quantum. After receiving one quantum of service, the CPU is preempted and switched to another process. If the running process has a current CPU burst that is less than one quantum, the CPU is switched to the next process in the ready queue.

The ready queue for the RR algorithm can be easily implemented by using a circular queue. The order of processes follows the FCFS rule. A clock interrupt (or a timer interrupt) of the operating system is used to preempt the CPU, and the interrupt interval is set to the time quantum size. It is important in the RR algorithm to define an appropriate length for the quantum. If the length of quantum is too long, the RR algorithm emulates the FCFS algorithm. On the other hand, if the length is too short, the execution time may be increased due to the overhead incurred as a result of frequent context switching. Some authors have discussed reasonable length for the time quantum. Tanenbaum claimed "a quantum around 100 msec is often a reasonable compromise," [Tanenbaum 94], and Silberschatz and Galvin mentioned "a rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum" [Silberschatz and Galvin 94].

2.3.5 Multilevel Queue

In a multilevel queue scheduling algorithm, the ready queue is partitioned into several subqueues which have their own policies. Each process is assigned to one of the subqueues according to the properties of the process. This algorithm is a combination of several scheduling algorithms. For scheduling between the subqueues of the ready queue, the preemptive priority scheduling algorithm is typically used. Each subqueue has its own scheduling algorithm because the goal of each queue may be different. For example, it is better to use a FCFS discipline or a nonpreemptive SJF algorithm for the subqueue containing batch processes than to use a RR algorithm with a small quantum size, since for batch processes we want to reduce turnaround time as opposed to response time. On the other hand, for the subqueue containing interactive processes, which require fast response times, a RR algorithm is usually used. In this situation, since the subqueue containing interactive processes has a higher priority than the batch process queue, the interactive queue is executed first. Batch processes can use the CPU only when there are no processes in the interactive process queue. When an interactive process joins the ready queue, the running batch process is preempted [Lister and Eager 93].

2.3.6 Multilevel Feedback Queue

The multilevel feedback queue algorithm, as a variation of the multilevel queue algorithm, does not assign a process to a subqueue permanently but rather allows the processes to move between the subqueues. Figure 3 illustrates a multilevel feedback queue that has n subqueues, numbered from 0 to n-1, according to the order of priority. Each queue i ($0 \leq i \leq n-1$) has a potentially different quantum size, these sizes generally increase with i (e.g., the quantum of queue 0 is less than that of queue 1). Sometimes queue n-1 has the FCFS algorithm. All new incoming processes to the ready queue start in queue 0. If a process in queue i is not completed within the quantum assigned to queue i, the process is moved to queue i+1. In the case of queue n-1, the process is returned to that queue again until it terminates [Krakowiak 88]. As s result, processes with long CPU bursts are executed in the lower priority subqueues [Nutt 92] [Silberschatz and Galvin 94].

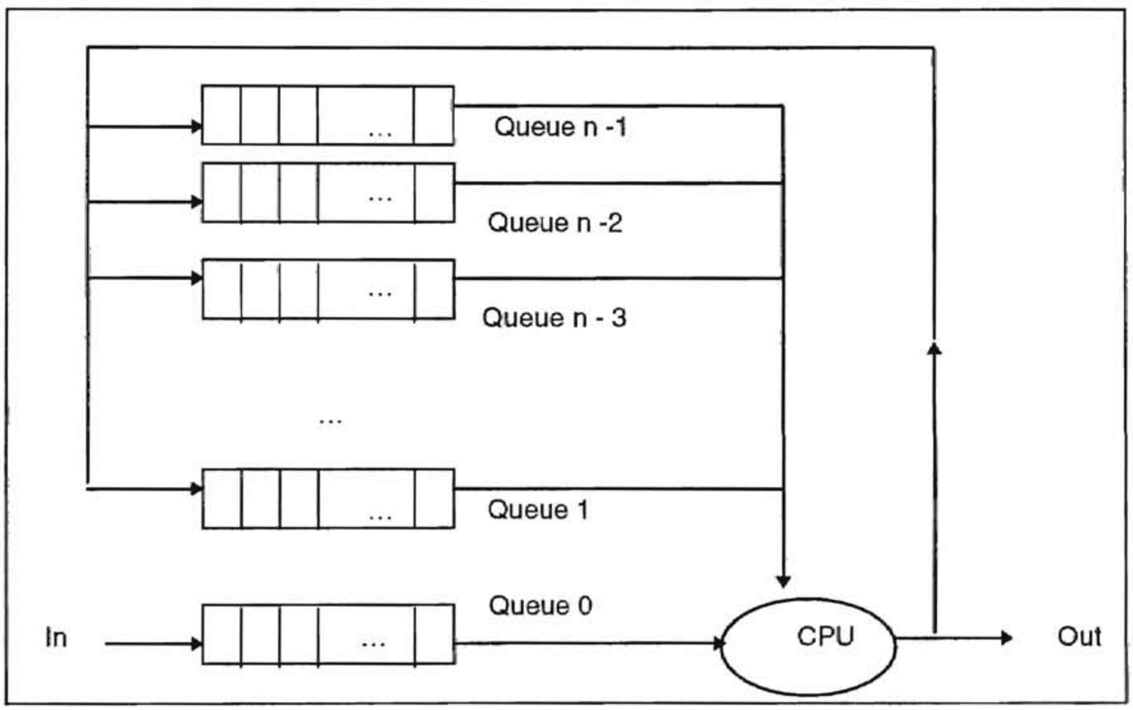

Figure 3. Multilevel feedback queue scheduling algorithm [Krakowiak 88]

2.4 Object-Oriented Programming

As the cost of computer hardware has been decreasing due to the revolutionary improvements in hardware technology during the last several decades, ever larger number of people can use computers. Computer users demand many software applications including large and complicated software systems. However, the software technology has difficulties in producing software at the appropriate time and also in maintaining the existing systems. So, the cost of software, especially large software systems, rises rapidly. The term 'software crisis' has been used to characterize this situation. The object-oriented programming approach is one of the proposed remedies for the software crisis [Florentin 91] [Ghezzi et al. 91] [Sommervile 96].

Object-oriented programming has its origins in Simula in 1967 [Kerr 91], but the object-oriented approach has become popular since Smalltalk was released in 1980 [Goldberg and Robson 89]. Nowadays, there are many object-oriented programming languages in use including C++, object-C, CLOS, ObjectLISP, and Object-Pascal. These languages were developed by adding object-oriented concepts to existing popular languages such as C, Lisp, and Pascal [Florentin 91]. Also, newly designed languages like Eiffel and Java have been introduced, and Ada, which is called an object-based language, is one of the programming languages that is widely used [Florentin 91] [Arnold and Gosling 96] [Meyer and Hucklesby 91].

The next five subsections discuss the common concepts and characteristics that all object-oriented programming languages should support.

2.4.1 Data Abstraction and Encapsulation

The state of an object, which is a static property of an object, is defined by its instance variables. The behaviour of an object, which is a dynamic property of an object, is defined by its methods [Budd 91]. The terms instance variable and method may have different meanings for different programming languages. In C++, the term 'data member' is used instead of instance variable, and 'member function' is used instead of method. The term 'member' is used as a general term that puts data member and member function together [Lippman 91]. Methods create new states and change the state by manipulating instance variables. As shown in Figure 4, instance variables are surrounded by methods.

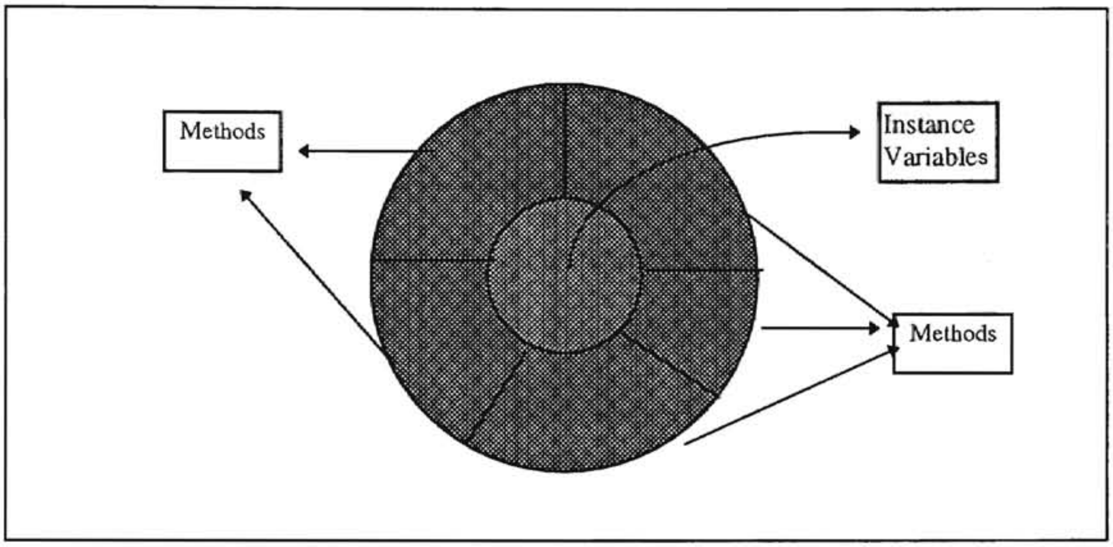

Figure 4. An object

The methods that hide and protect instance variables (i.e., the inner nucleus) from other objects are the only interface of an object to the outer world. This kind of packing is called data encapsulation which implements information hiding and provides modularity (i.e., data abstraction). So, the clients cannot access instance variables directly, and the clients do not have to know the details of the implementation of an object. The clients just know the interface (i.e., the object methods), and access an object's state only by using its interface. When the implementation of an object is changed, a client's program is not affected because of requiring only the change of the interfaces associated with the object.

In addition to independence, data can be protected from unexpected behaviors such as clients' errors by using encapsulation [Booch 91] [Budd 91] [Arnold and Gosling 96].

2.4.2 Class and Access Control

There are many objects of the same kind that share common characteristics. A class is a template that defines the instance variables and methods common to all objects of a certain kind. An object is created by instantiating a previously defined class, and many objects can be instantiated from one defined class. Since programmers can use the same class to create many objects, classes provide the necessary condition for reusability of the objects [Budd 91].

All declarations about members of a class may be classified according to the levels of constraints of accessibility from other classes; 'Public', 'Private', and 'Protected' are three categories into which members can be classified. Members of a class with public declaration can be accessed by all other classes. In the case of members with private declarations, the outer classes cannot access them directly. A member with protected declarations is only accessible to its subclasses [Booch 91].

2.4.3 Message Passing

There are many objects in a program, and a program is executed by each object interacting and communicating with other objects. Message passing is used for all interactions and communications among objects. If object A wants object B to do some work on object A's behalf, object A sends a message to object B. In response, object B selects the appropriate method to perform the request. The name of the method to perform is selector, which is used to find a matching method during the processing of message passing. Sometimes a receiving object needs more information. Such information is passed along with a message as parameters. So, In object-oriented programming, a message is composed of a receiver (i.e., the object to which the message is addressed), selector (i.e., the name of the method to perform), and the argument (i.e., any parameters needed by the method). The message passing paradigm has benefits in heterogeneous networking systems because it is not necessary for the sending object and the receiving object to be the same process or to exist on the same machine [Booch 91] [Budd 91].

2.4.4 Class Relationships and Inheritance

In general, there are three kinds of relationships among classes: *is_a* relationship, *has_a* (or *has_a_part*) relationship, and *associated* relationship. The *is_a* relationship is formed when one class is a special instance of another class, just as it is said that a circle *is_a* shape since a circle is a special instance of a shape. In other words, if class A *is_a* class B, it means that A is a specialized class of the more general class B. Specialized classes such as A and the circle are called subclasses or derived classes, and more generalized classes such as B and the shape are called superclasses or base classes. The *is_a* relationship supports generalization (i.e., a superclass can be extracted from its subclasses) and specialization (i.e., a subclass is formed from its superclass), which are abstraction techniques. A hierarchy of classes is based on this relationship [Booch 91] [Budd 91].

It may be said that a composite class, which consists of several subcomponents, has the *has_a* relationship with its subcomponent classes. For example, a complex number class *has_a* real number class since the complex number class consists of real numbers and imaginary numbers. The *has_a* relationship supports the aggregation technique which creates a composite class from subcomponent classes [Budd 91].

The last relationship is *associated,* which represents some semantic connection such as having the same purpose. A certain job is completed by interacting with the *associated* classes. This relationship is implemented by message passing techniques between the requester and provider classes [Booch 91].

When a subclass is defined from an existing superclass (i.e., subclassing) by an *is_a* relationship, the subclass may inherit the property (i.e., the instance variables and methods) of the superclass. In other words, the subclass may have the property of the superclass as well as its own property. Since the *is_a* relationship is transitive, a subclass can inherit a property from a superclass that is higher in the hierarchy. For example, if class car *is_a* class vehicle and class vehicle *is_a* class transporter, then class car *is_a* class transporter, so class car inherits the property of class transporter. Since inheritance generally enables software developers to reuse existing codes which are already developed and tested, the cost of software development may be reduced by using the object-oriented programming paradigm [Budd 91].

Sometimes some classes may inherit from more than one immediate superclass (i.e., multiple inheritance). The properties of these classes are combinations of properties from all relevant superclasses. Multiple inheritance can cause some problems. The problem that arises when the same member may be inherited from more than one superclasses, is one of such problems. Renaming of the instance variables and methods of the subclass is usually used as a solution to this problem.

Since multiple inheritance generally makes a program more complex, discussions about the necessity of this technique have continued. Actually, many object-oriented programming languages except for C++ and CLOS, do not support multiple inheritance.

The Java language, which extends the object model and removes the major complexities of C++ , does not support multiple inheritance either [Budd 91] [Arnold and Gosling 96].

2.4.5 Polymorphism, Overloading, and Overriding

In object-oriented programming, it is possible that a class has several variables and methods with the same name, which is unlike procedural programming languages. This mechanism is called polymorphism. Such methods are differentiated by their classes and parameters. Polymorphic variables that have no type associated with them can contain any type of data.

Sometimes several methods with the same name work for different classes and provide different behaviors. For example, the '+' method in the integer class operates addition between integers, but the '+' method in the complex number class operates complex number addition (i.e., real numbers are added among themselves and imaginary number are added among themselves). Overloading (i.e., ad hoc polymorphism) means that methods already defined in a class are used differently in other classes. When a new class is formed from the superclass, the new class can define its new method with the same name as the superclass's name. In this case, the subclass overrides the inherited methods and provides a specialized implementation for this new method [Budd 91].

CHAPTER III

DESIGN AND IMPLEMENTATION ISSUES

3.1 Implementation Platform and Environment

The tool was implemented on a Sequent Symmetry S/81 machine, a mainframe-class multiprocessor system with 24 80386 processors running at 20Mhz each. The operating system of this machine is DYNIX/ptx, a variant of the UNIX system. The object-oriented programming language ANSI C++ version 2.0.1 was used to implement the tool.

3.2 Objective

The purpose of this work was to develop a tool which simulates process scheduling by using the object-oriented approach. In this tool, six scheduling algorithms introduced in the literature review part of this thesis (Section 2.3) were simulated. To simulate the scheduling algorithms, objects which simulate various components of an operating system were developed. These could be reused for different kinds of scheduling algorithm and even for other scheduling algorithms not discussed in this work. This simulation was completed by using the techniques and characteristics of the object-oriented programming described in Section 2.4. The extended and complex scheduling algorithms were easily and compactly simulated by inheriting properties from basic scheduling algorithms such as FCFS.

By running the simulation, performance parameters such as CPU utilization, turnaround time, and response time were computed at regular time intervals. This scheduler generator tool can help users choose from among a number of candidate scheduling algorithms for a specific system.

3.3 Design and Implementation Issues

3.3.1 Overall Hierarchy of Scheduling Algorithms

Scheduling algorithms can generally be classified into priority/non-priority and preemptive/non-preemptive. Non-priority scheduling algorithms include FCFS and RR, and priority scheduling algorithms include SJF, multilevel queue, and multilevel feedback queue. Also, RR, multilevel queue, and multilevel feedback queue scheduling algorithms are preemptive. FCFS, SJF, and priority scheduling algorithms are non-preemptive. Figure 5 shows the overall classification of six scheduling algorithms.

Among the scheduling algorithms considered, the one that is both non-priority and non-preemptive (i.e., FCFS) can be the most basic object, since this algorithm is more general and simpler than the other algorithms. SJF, RR, and priority scheduling algorithms could be inherited from the FCFS class. Multilevel queue which is a variant of the RR algorithm could be inherited from RR, and Multilevel feedback queue which is a variant of the multilevel queue algorithm could be inherited from multilevel queue. Figure 6 gives the overall hierarchy of the six different scheduling algorithms considered in this study.

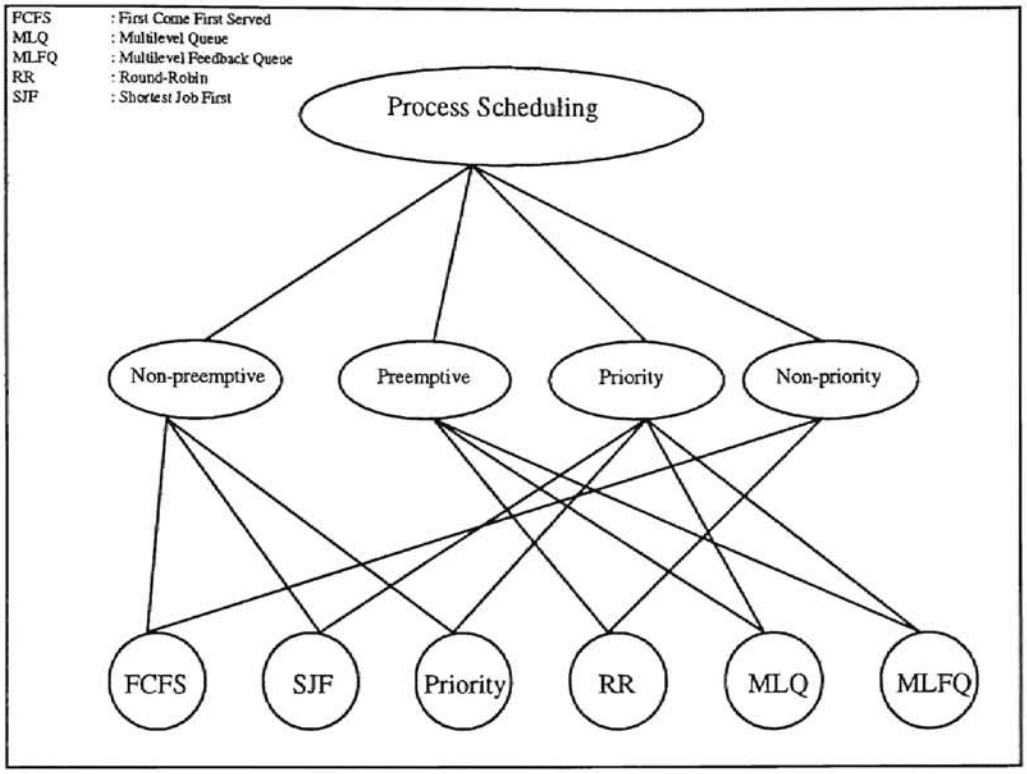

Figure 5. Overall Classification of Scheduling Algorithms

3.3.2 Components of Scheduling System

The process scheduling model illustrated in Figure 2 was used as the main procedure of the overall simulation. The operating system components for process scheduling were designed and implemented as objects. These objects include: clock, PCB, ready queue, job queue, blocked queue, memory manager, loader, dispatcher, and scheduler (see Figure 7).

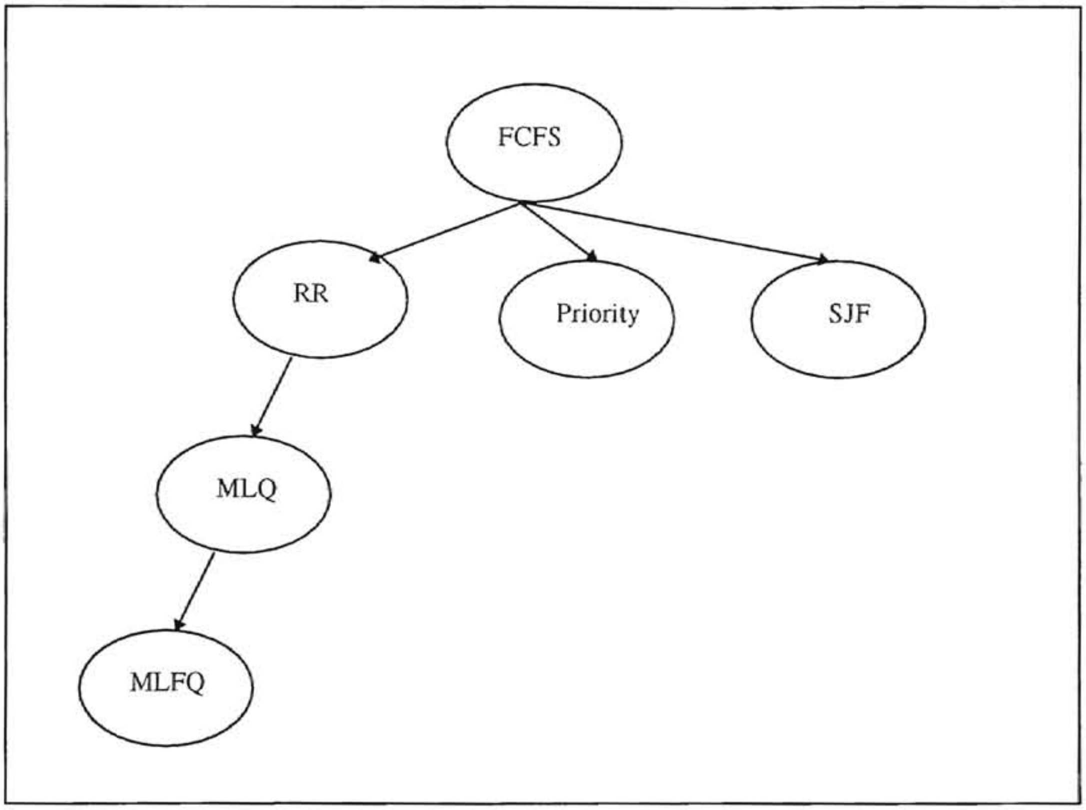

Figure 6. Hierarchy of Six Scheduling Algorithms

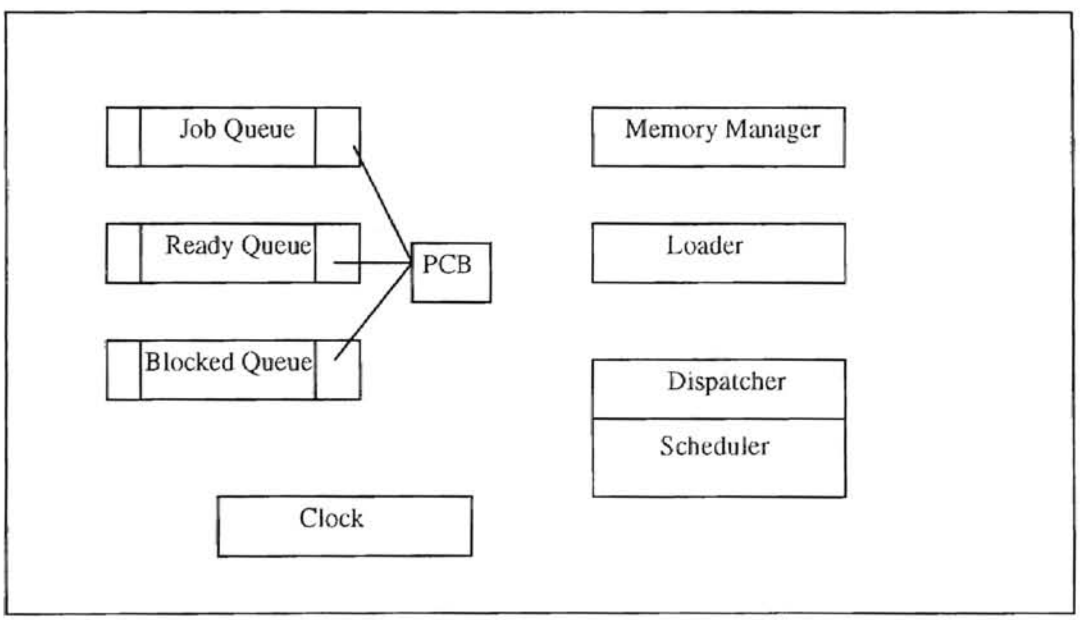

Figure 7. Components of Process Scheduling

The virtual CPU clock was simulated as a counter (i.e., data member 'value') which is increased by the CPU burst of the process currently in the running state. The CPU clock object was created by instantiating a class CLOCK which was previously defined. The class CLOCK also has a data member to store the collected statistics concerning system utilization. These statistics were collected and reported at every 500 clock units.

Each process in the main memory was represented by a PCB object. The object PCB was created by instatiating a class PCB or its subclasses such as class ExPCB and class EExPCB. Class PCB is the base class that contains the basic members necessary to implement the simplest scheduling algorithm. At least the following data member are included in class PCB: ID, size, priority, status, number of CPU bursts, burstoffset, current burst length, time the process entered the system, CPU execution time, and current I/O completion time. Most member functions were defined to access and update the data members.

To implement the multilevel queue scheduling algorithm, class ExPCB inherited from class PCB and was further defined by adding the extra data member, which indicated a current subqueue where a process was assigned. Class EExPCB was defined from class ExPCB with an extra data member indicating the number of turns spent in the current subqueue.

In the multilevel feedback queue scheduling algorithm, if the number of turns that a process has spent thus far in the current subqueue is greater than the number of turns assigned to the subqueue, the process moves into another low-level subqueue. As a result, while the object PCB as instantiated from class PCB was used in FCFS, SJF, priority, and

RR scheduling algorithms, the PCB from class ExPCB was used in the multilevel queue scheduling algorithm, and the PCB from class EExPCB was used in the multilvel feedback queue scheduling algorithm. Figure 8 gives the relations among the class PCB and its subclasses.

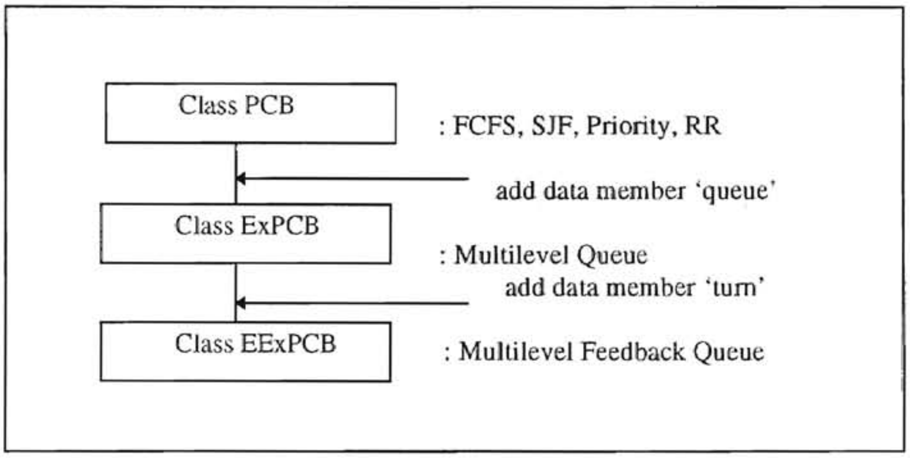

Figure 8. Inheritance of class PCB and its subclasses

Class Queue was defined to implement the FIFO queue. The blocked queue and job queue objects used in all of the scheduling algorithms, as well as the basic ready queue objects used in the FCFS and RR scheduling algorithms, were created by class Queue. PCB objects were used for elements of class Queue. The FIFO queue was constructed based on class PCB by including a pointer to another PCB. The data member 'top' indicates the header of a queue, 'end' indicates the tail of the queue, and 'num' indicates the number of processes in the queue. The main operations of class Queue are enqueue, dequeue, and remove. Figure 9 gives the data structure and operations of class Queue. In this figure, T may be class PCB or its subclasses according to the algorithm.

Since the blocked and job queues were implemented as FIFO queues, their types are the same in different kinds of algorithms, but the ready queue type may be different according to each scheduling algorithm under consideration. Class SortedQueue defines a queue whose elements are arranged in ascending order. Class SortedQueue inherited class Queue's data member and function member except for the 'Enqueue' member function. The function Enqueue of SortedQueue overrides the super class's Enqueue function. So the Enqueue of class Queue must be defined as a virtual function. The ready queues used in the priority and SJF algorithms were created from class SortedQueue.

```
class Queue {
protected :
        T *top;
        T *end;
        int num;
public:
        Queue();
        virtual void Enqueue(T *Node);
        T *dequeue(void);
        T *Head(void) { return top; }
        T *Tail(void) { return end; }
        void print(void);
        int GetNumProcess() { return num;}
        void change_num(int i) { num=num+i; }
        T *remove_pcb(int id);
};
```

Figure 9. Definition of Class Queue

In the multilevel queue and multilevel feedback queue scheduling algorithms, the ready queue was divided into several subqueues. In this simulation, each subqueue had RR scheduling algorithm with different quantum sizes (the highest priority subqueue, which is the lowest numbered subqueue, has the smallest quantum size). For scheduling between the subqueues, the preemptive priority policy was used. So each subqueue contained the relevant information about its own quantum size. By adding the extra data member indicating the quantum size, class SubQueue was defined from class Queue. While a process is assigned to its subqueue permanently in the multilevel queue algorithm, in the multilevel feedback queue algorithm, a process can move to a lower-level subqueue after spending the assigned number of turns in the current subqueue. So class ExSubQueue which adds the 'turn' data member to class SubQueue was defined. Figure 10 shows the organization of class Queue and its subclasses.

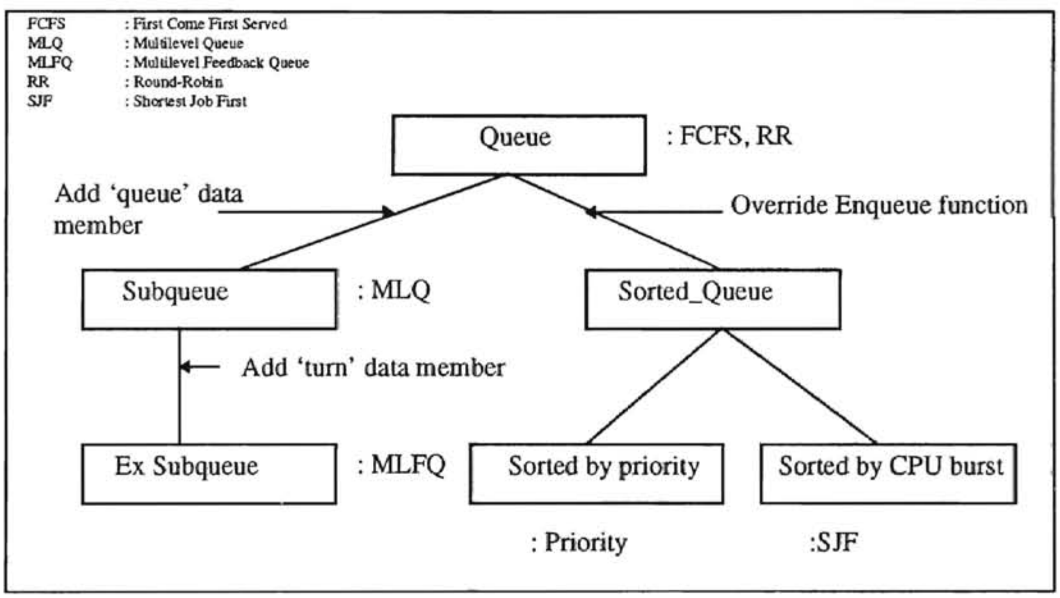

Figure 10. Organization of class Queue and its subclasses

For this simulation memory was simulated as a counter. The user can specify the maximum and minimum number of allocable units. At default, 512 allocable units are specified as the upper bound and 12 units as the lower bound. Class Memory defines all information and functions to manage the simulated memory. Memory manager is responsible for checking, acquiring, releasing, and reporting statistics about the simulated memory. The same type of memory object was used in different kinds of scheduling algorithm. Figure 11 presents the definition of class Memory. The data member 'pcbcount' indicates the total number of processes in main memory, and this number should be less than the maximum degree of multiprogramming.

```
class Memory {
protected:
        int availmemory;
        int minmemory;
        int pcbcount;
public:
        Memory();
        Memory(int n);
        rvalue checksize();
        Boolean acquire(int job_size);
        void release(int jsize);
        void print(void);
        int getpcb() { return pcbcount; }
        void compute_pcbcount(int i ) { pcbcount = pcbcount + i; }
};
```

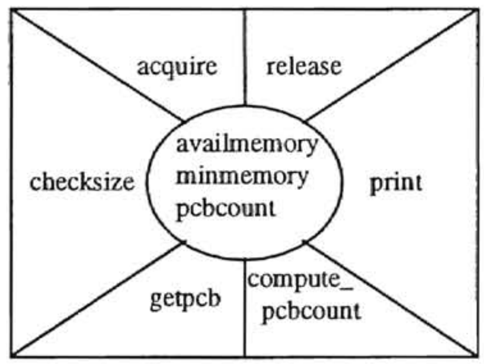

Figure 11. Definition of class Memory

The object loader, which is responsible for loading processes into main memory from the job queue and the disk until memory is full, was created from class Loader. Processes are in the form of <process ID> < process size> <process priority> <burst 1 ...

burst n> in an input data file. The value 0 for process size indicates that these are no new processes arriving at that time. If enough memory is available, the loader creates a PCB and inserts it into the ready queue. Otherwise, the loader creates a PCB and inserts it into the job queue. The processes in the job queue wait to be loaded to main memory. The loader stops loading processes when there is not enough memory or there are no new process arrivals. In this simulation, when the loader load a process, the process in the job queue have priority over new arrivals. The definition of class Loader is presented in Figure 12.

```
class Loader {
protected:
        FILE *inputfile;
public:
        loader();
        void LoadJob(Queue &jqueue, Memory &m, RQTYPE *rqueue);
        virtual void GoToReadyQueue(T *cur, RQTYPE *rq);
        rvalue Status(Queue Jqueue);
};
```

Figure 12. Definition of class Loader

Since the loader assigns a process to a subqueue permanently based on the priority of the process in the multilevel queue scheduling algorithm, extra action is required when the process goes to the ready queue. A subclass of class Loader, Class ExLoader, has its own GotoReadyQueue function that overrides the super classs's corresponding function. In the multilevel feedback queue scheduling algorithm, every process start at the highest level subqueue. So class Loader was used for its loader. Figure 13 shows the definition of class ExLoader.

```
class ExLoader: public loader {
        public:
                void GoToReadyQueue(T *cur, RQTYPE *rqueue);
};
```

Figure 13. Definition of class ExLoader

The process scheduler dispatches a process and maintains the process after the execution. The dispatcher, which is a part of the scheduler, dispatches the process at the head of the ready queue. In non-preemptive scheduling, once a process is running, the following actions can occur:

1. Process requests I/O: the scheduler lets the process go to the blocked queue and stay there until its I/O is completed (blocked member function of class scheduler). Ten time units is specified as default for I/O service time. A user can redefine the service time. When a process completes its I/O and is ready to run again, it is placed on the ready queue (unblocked member function).

2. Process terminates: the process's memory is released by the memory manager and the PCB is destroyed, and the statistics related to the process are reported (terminate member function).

Class Dispatcher and class Scheduler were used to create object dispatcher and object scheduler for non-preemptive scheduling. Figure 14 presents the definition of class Scheduler.

For the preemptive scheduling algorithm, some activities and information were added for implementing CPU preemption to class Dispatcher (class RR_Dispather). Since mutilevel queue and multilevel feedback queue are variations of the RR scheduling algorithm, the dispatcher and scheduler inherited from RR's functions. For the multilevel queue and multilevel feedback queue scheduling algorithms, some actions for dispatching the process at the head of the highest priority non-empty subqueue was added to class RR_Dispatcher. Figure 15 shows the relations among class Dispatcher and its subclasses.

```
class Scheduler {
        protected:
                FILE *memoryfile;
                FILE *jobdonefile;
                int jobdonecount;
        public:
                Scheduler();
                void update_burst(T *cur);
                void blocked(Queue &bq, RQTYPE &rq, CLOCK &c1);
                rvalue unblocked(RQTYPE *rq, Queue &bq, CLOCK c1 );
                void report(Queue jq, RQTYPE &rq, Queue bq, CLOCK c1,
                Memory m1);
                void terminate(T *cur, Memory &m1);
                void close_file();
                virtual void GoToReadyQueue(T *cur, RQTYPE *rq);
        };
```

Figure 14. Definition of class Scheduler

When the CPU is preempted, the scheduler appends the preempted process to the ready queue. For this action, the function UpdateQueue() was added to class RR_Scheduler. In multilevel queue scheduling, since the ready queue consists of several subqueues, the scheduler appends the process to its own assigned subqueue when the CPU is preempted and the I/O request is completed. So the GoToReadyQueue function in class ML_Scheduler overrides the superclass's definition of that function. To implement the multilevel feedback queue algorithm, appropriate actions for the movement of processes between the subqueues were added to class ML_Scheduler. The UpdateQueue and GoToReadyQueue member functions in class MLFQ_Scheduler overrides the superclass's definition. Figure 16 gives the organization of the scheduler for the six scheduling algorithms.

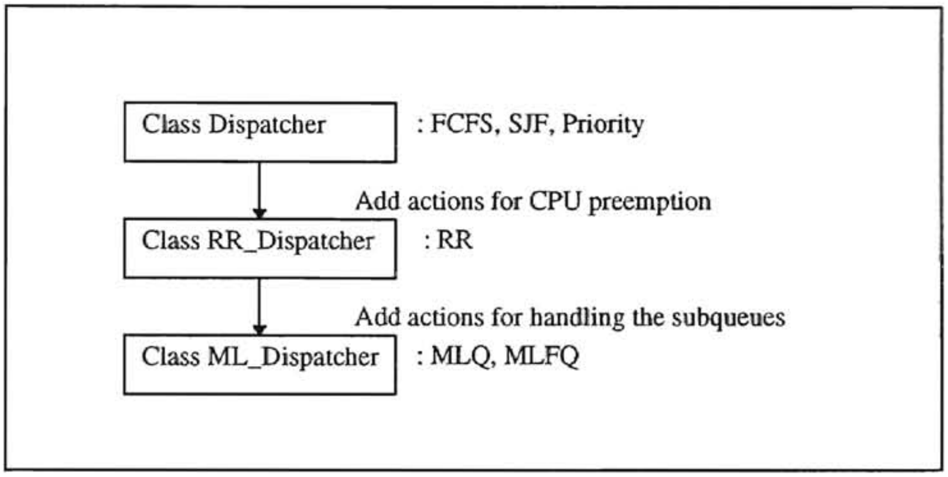

Figure 15. Organization of dispatcher

In this simulation, each scheduling algorithm itself was developed as a complex object created from six subcomponents. Class FCFS, which implements the FCFS scheduling algorithm, was defined in Figure 17. The 'system' member function is the main program to drive the sumulation and the overall loop that accesses the memory manager, loader, clock, queues, dispatcher, and scheduler.

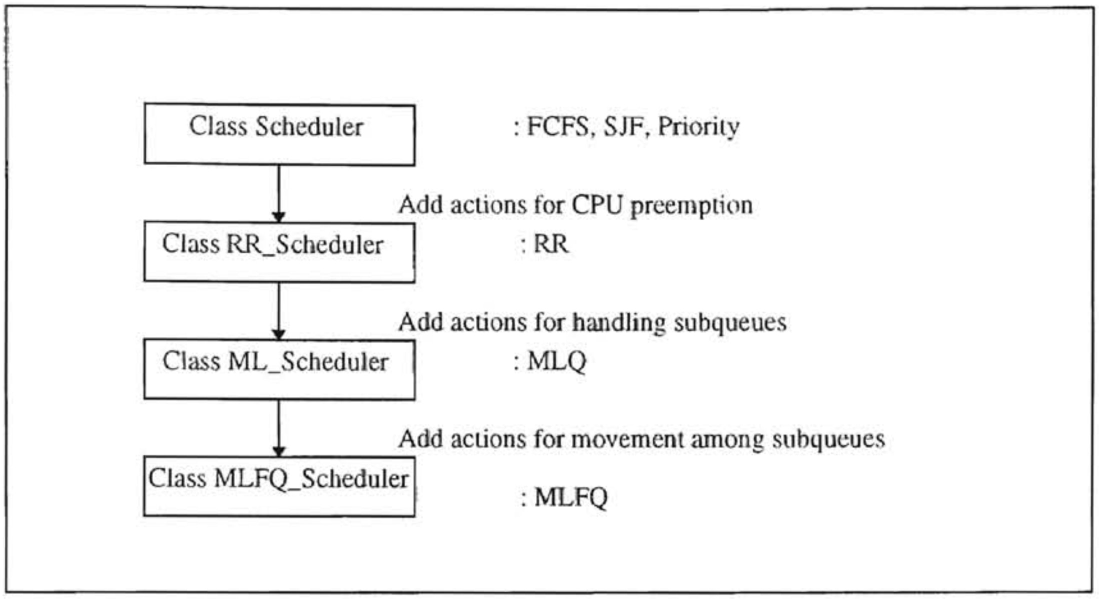

Figure 16. Organization of Scheduler

```
class fcfs {
protected:
        CLOCK c1;
        Memory m1;
        Queue JobQueue;
        Queue blockedQueue;
        Loader l1;
        Scheduler sch;
        Dispatcher d1;
        Queue readyQueue;
public:
        void system(loader &l1, scheduler &sch, Dispatcher &d1,
        Queue *readyQueue);
        virtual void CPU(T *cur, RQTYPE *readyQueue);

};
```

Figure 17. Definition of FCFS Scheduling Objects

The SJF and priority algorithms have the same procedure and components as FCFS scheduling but the type of the ready queue is a sorted queue. They are defined in Figure 18.

Preemptive scheduling (that includes RR, multilevel queue, and multilevel feedback queue scheduling) has the following extra procedure: if the quantum is used up, place the process in the ready queue. Figures 19 to 21 give the definition of the RR, multilevel queue, and multilevel feedback queue scheduling. They satisfy the relations among the six scheduling algorithms as illustrated in Figure 6.

```
class sjf: public fcfs {
        protected:
                Sorted_Queue readyQueue;
        public:
                void system();
};

void sjf::system()
{
        fcfs::system(l1,sch,d1, &readyQueue);
}
```

Figure 18. Definition of SJF Scheduling Object

```
class rr: public fcfs {
        protected:
                RR_Dispather d1;
                RR_scheduler sch;
        public:
                void system();
                virtual void CPU(T *cur, RQTYPE *readyQueue);
};
```

Figure 19. Definition of RR Scheduling Object

```
class mlqueue: public rr {
        private:
                Exloader l1;
                ml_Dispatcher d1;
                ML_scheduler sch;
                SubQueue Sq[10];
        public:
                virtual void CPU(T *cur, RQTYPE *rq);
                void system();
        };
```

Figure 20. Definition of MLQ Scheduling Object

```
class mlfq :public mlqueue {
        protected:
                loader l1;
                mlfq_Dispatcher d1;
                MLFQ_scheduler sch;
                ExSubQueue Sq[10];
        public:
                void system();
                virtual void CPU(T *cur, RQTYPE *rq);
};
```

Figure 21. Definition of MLFQ Scheduling Object

3.3.3 Communication among Objects

Several objects were developed in the simulation. Execution of the program was carried out by each object interacting and communicating with other objects. The system interacts with the object loader by calling loader.status(JobQueue) to check if there is a process on disk, and by calling loader.LoadJob(JobQueue, Memory, ReadyQueue) to load the process. Loader communicates with the job queue by calling JobQueue.Head() to get the head of the job queue, by calling JobQueue.Enqueue() to place a process in the job queue, and by calling JobQueue.remove_pcb() to remove the process from the job queue, with the memory by calling Memory.checksize() to check if there is enough memory to load, and by calling Memory.acquire() to allocate the memory requested, and with the ready queue by calling ReadyQueue.Enqueue() to place a process in the ready queue.

The system communicates with the dispatcher object by calling dispatcher.Dispatch(PCB, ReadyQueue, clock), and the dispatcher interacts with the ready queue to remove the terminating process from the ready queue (ReadyQueue.Dequeue()), and with the clock to compute the current virtual clock (clock.ComputerClock()).

The system interacts with the scheduler by calling scheduler.update_burst(PCB), scheduler.blocked(BlockedQueue, ReadyQueue, clock), scheduler.terminate(PCB, Memroy), scheduler.unblocked(ReadyQueue, BlockedQueue), and scheduler.report(JobQueue, ReadyQueue, BlockedQueue, clock, Memory). The scheduler communicates the ready queue, the blocked queue, the job queue, PCB, memory, and clock to maintain the ready queue and the blocked queue (blocked, unblocked), to report the statistics about system performance (report), to release memory when the process is terminated (terminate), and to update the process information (update_burst). Figure 22 describes the communication among objects.

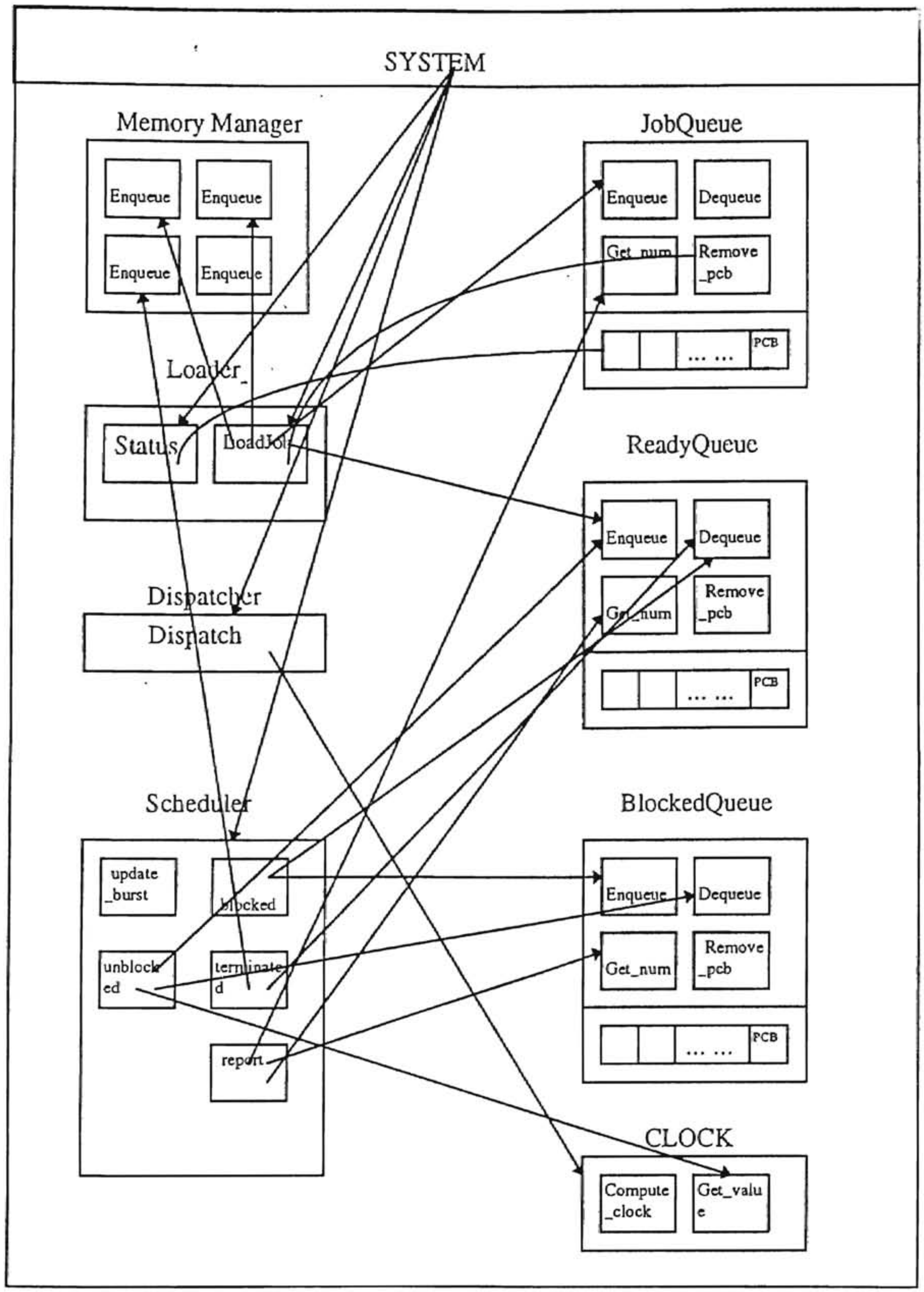

Figure 22 Communication Among Objects

CHAPTER IV

EVALUATION OF THE TOOL

4.1 Input file and Hardware specification

The secondary store and disk was simulated as an input file where process requests resided in this simulation. A process request was formed of: Process ID as the first parameter, amount of memory units requested as the second parameter, process priority as the third parameter, and the given CPU bursts as the remaining parameters. A process size of 0 indicated that there was no incoming process at that time. Figure 23 gives the format and an example of process requests. Appendix C contains a sample input file used to test the simulation.

<Process ID>	<Memory Size>	<Priority>	< Burst 1>	<Burst 2>	....	<Burst n>
4	4	3	50	163	....	17
5	71	2	51	53	....	57
0	0					
0	0					
....	...					

Figure 23. Format and Example of Process Requests

For the process of evaluation, 512 allocable memory units was chosen as the upper bound, and 12 units as the lower bound. A period of 10 virtual time units was used per I/O request. Quantum sizes of 30 were used in RR scheduling, and the ready queue was divided into 4 subqueues in multilevel queue scheduling. In multilevel feedback queue scheduling, the ready queue was divided into four subqueues with residency rules as specified in TABLE I.

TABLE I. RESIDENCY RULES IN MULTILEVEL FEEDBACK QUEUE

	Subqueue 1	Subqueue 2	Subqueue 3	Subqueue 4
# of turns	3	5	6	----
quantum size	20	30	50	80

4.2 Output

When the simulation of each scheduling algorithm was finished, two output files named jobstat and memstat were created by the tool. When a process terminated, the following statistics about the process was written to the jobstat file: <Process ID> <Time process entered the system> <Time process is leaving the system> <Execution time> <Turnaround time>. The execution time of each process was computed by adding its CPU running time to its I/O service time. Figure 25 shows a segment of the jobstat, and Appendix E shows a sample jobstat file.

Every 500 time units, the following information relating to the system utilization and status was written to a file called memstat: <allocated memory units> <free memory units> <number of processes in job queue> <number of processes in ready queue>

<number of processes in blocked queue> <number processes delivered>. Figure 25 shows the part of the memtat file. Appendix D shows the whole memstat file.

```
ID:  8 Entered:  220  Left: 2250 Execution:  341 TAT:  2030
ID:  4 Entered:    0  Left: 3063 Execution:  371 TAT:  3063
ID:  7 Entered:  220  Left: 3212 Execution:  400 TAT:  2992
ID: 10 Entered: 3212  Left: 4125 Execution:  512 TAT:  3102
ID: 13 Entered: 2250  Left: 4823 Execution:  726 TAT:  2237
                          ... ... ...
```

Figure 24. Part of a sample jobstat file

Stat. Time	Allocated Mem.	Free Mem.	Job_q	Blocked_Q	Read_Q	Jobs Done
515	510	2	1	0	10	0
1000	510	2	1	0	10	0
1522	510	2	1	1	9	0
2022	510	2	1	0	10	0
2517	505	7	2	0	10	1
3016	505	7	2	0	10	1
... ...						

Figure 25. Part of a sample memstat file

CHAPTER V

SUMMARY AND FUTURE WORK

5.1 Summary

Chapter I introduced the overall concepts of process scheduling and object-oriented programming. It also addressed the importance and necessity of process scheduling and popularity of the object-oriented approach. Chapter I ended by presenting the purpose and outlining the organization of this thesis.

In Chapter II, the general process scheduling model used for this simulation was described. Chapter II also presented several common system utilization factors, and six widely-used scheduling algorithms. It also discussed the advantages and problems of each of the six scheduling algorithms. The origin of object-oriented programming was briefly addressed in this chapter. The chapter ended by discussing the common concepts and characteristics that all object-oriented programming languages should support.

Chapter III presented the implementation platform and the design/implementation issues of the simultaion. The overall hierarchy of six scheduling algorithms, and the development of the various components of the scheduling system (i.e., loader, clock, memory, scheduler, dispatcher, PCB, ready queue, blocked queue, and job queue) were discussed in Chapter III. This chapter included a discussion about the relations and communications among the components of the system. The development of each scheduling algorithm as an object, and the relation among such objects was also discussed in Chapter III.

Chapter IV presented the input file and other specifications including memory size, quantum length, degree of multiprogramming, number of subqueues, and residency rules that were used to test the tool. This chapter also described two output files and the performance factors obtained from each execution of the simulation.

The simple scheduler generator tool, which was simulated on Sequent S/81 running DYNIX/ptx using C++ version 2.0.1, could serve as an object-oriented prototyping environment for conventional and innovative process scheduling algorithms. Extended and complex objects with their own properties and operations were easily created by inheriting from the existing objects with the most basic and common properties and operations. This tool can be used to choose from among a number of scheduling algorithms in a given system environments.

5.2 Future Work

Real-time scheduling, distributed scheduling, and multiprocessor scheduling are the difficult problems of process scheduling. In this tool, multiprocessor, distributed, and real-time scheduling were not be included. As an area of future work, these could be implemented by adding new complex objects and updating the features of some existing objects.

This tool was developed using C++ version 2.0.1 under a flavor of the UNIX environment (i.e., DYNIX/ptx). This version does not support the "template", which is a keyword for polymorphic variables. It can be argued that if templates were used, the program would be more legible.

REFERENCES

[Arnold and Gosling 96] K. Arnold and J. Gosling, *The Java™ Programming Language*, Addison-Welsey Publishing Company, Inc., Reading, MA, 1996.

[Booch 91] G. Booch, *Object Oriented Design with Applications*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1991.

[Budd 91] T. Budd, *An Introuction to Object-Oriented Programming*, Addison-Welsey Publishing Company, Inc., Reading, MA, 1991.

[Florentin 91] J. Florentin, "Object-Oriented Techniques: Now and the Future", In *Object-Oriented Programming Systems Tools and Applications* (pp. 1-6), J. J. Florentin (Ed.), Chapman & Hall, Inc., London, UK, 1991.

[Ghezzi et al. 91] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

[Goldberg and Robinson 89] A. Goldberg and D. Robinson , *Smalltalk-80: The language*, Addison-Welsey Publishing Company, Inc., Reading, MA, 1989.

[Kerr 91] R. Kerr, "Simula - Ancient and Modern", In *Object-Oriented Programming Systems Tools and Applications* (pp. 125-134), J. J. Florentin (Ed.), Chapman & Hall, Inc., London, UK, 1991.

[Krakowiak 88] S. Krakowiak, *Principles of Operating Systems*, The MIT Press, Cambridge, MA, 1988.

[Lippman 91] S. B. Lippman, *C++ Primer*, Second Edition, Addison-Welsey Publishing Company, Inc., Reading, MA, 1991.

[Lister and Eager 93] A. M. Lister and R. D. Eager, *Fundamentals of Operating Systems*, Fifth Edition, Spring-Verlag, Inc., London, UK, 1993.

[Meyer and Hucklesby 91] B. Meyer and P. Hucklesby, "Eiffel: An Introduction", In *Object-Oriented Programming Systems Tools and Applications* (pp. 125-134), J. J. Florentin (Ed.), Chapman & Hall, Inc., London, UK, 1991.

[Nutt 92] G. J. Nutt, *Centralized and Distributed Operating Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992.

[Silberschatz and Galvin 94] A. Silberschatz and P. B. Galvin, *Operating System Concepts*, Fourth Edition, Addison-Welsey Publishing Company, Inc., Reading, MA, 1994.

[Sommerville 96] I. Sommerville, *Software Engineering,* Fifth Edition, Addison-Welsey Publishing Company, Inc., Workingham, England, 1996.

[Tanenbaum 92] A. S. Tanenbaum, *Modern Operating Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992.

APPENDIXES

APPENDIX A:

GLOSSARY

Aging:	The gradual increasing of the priority of the processes that are waiting in the ready queue.
Batch Process:	A process whose user cannot interact with it when the process is executing.
CTSS:	Compatible Time Sharing System. It was an experimental time sharing system designed at MIT and implemented on an IBM 7090.
Information Hiding:	The principle that users do not need to know the details of implementation of software components but need to know the essential details of how to initialize and access a component.
Instance:	A specific example of a defined class.
Instance Variable:	The data associated with each instance of a class. In C++, these are called data members.
Interactive Process:	A process whose user can make on-line interactions with it. The user gives instructions to the operating system and the program, and receives a response.
Member:	A general term used for both a data member and a function member in C++.
Method:	A procedure or function associated with a class. In C++, these are called function members.
MLQ:	Multilevel Queue
MLFQ:	Multilevel Feedback Queue

Multiprogramming:	Multiprogramming allows processes to share memory and CPU. Several programs can run on the same machine virtually at the same time in a multiprogrammed system.
PCB:	The Process Control or Context Block of a process contains the information associated with that process.
Polymorphism:	A property that indicates the instance variables and methods have more than one form.
Process:	A program in execution and a sequential unit of computation.
Resource:	A resource denotes any abstract machine environment object that is required by a process for execution.
Subclass:	A class that inherits from another class.
Superclass:	A class from which other classes inherit attributes.
Time Sharing:	A logical extension of multiprogramming that switches the CPU among processes so frequently that the users can interact with each process.

APPENDIX B:

TRADEMARK INFORMATION

Ada:	A registered trademark of the U.S. Government (Ada Joint Program Office).
DYNIX/ptx:	A registered trademark of Sequent Computer Systems, Inc.
Eiffel:	A registered trademark of Interactive Software Engineering, Inc.
Java:	A registered trademark of Sun Microsystems, Inc.
NCD:	A registered trademark of Network Computing Devices, Inc.
Sequent Symmetry S/81:	A registered trademark of Sequent Computer Systems, Inc.
UNIX:	A registered trademark of AT&T.

APPENDIX C:

INPUT FILE

This is a sample input file used to test the scheduler generator tool. The process requests are in the form of <process ID> <process size> <process priority> <burst 1> ... <burst n>, where the bursts are the periods of uninterrupted CPU activity. The value 0 for process ID and size indicates that there is no incoming process at that time.

ID	size	priority				CPU bursts				
1	20	0	40	44	53	40	63	163	56	
2	71	2	147	51	346	56	44	63	15	56
3	17	3	6	145	64	16	461	112		
4	4	1	50	163	111	17				
5	71	2	51	53	115	440	156	57		
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
6	62	3	66	163	14	30				
7	67	2	14	44	190	54	15	11	12	
8	67	0	51	56	31	63	36	54		
9	64	1	64	15	40	157	151	66	163	151
10	72	1	141	145	143	53				
11	67	2	440	146	56	52	141	89		
12	70	1	12	141	564	40	17	11	78	
13	62	2	27	156	44	143	54	16	54	162
14	64	3	164	145	14	546	64	40		
15	60	0	54	43	12	41	45	71	12	
16	64	2	50	13	52	15	51	40		
17	20	1	157	43	89	16	15	44	54	
18	10	2	15	16	56	15	4	15	190	
19	10	3	14	16	14	67				
20	13	0	15	124	49	156				
21	103	2	56	190	44	54	189	25	4	
22	10	1	15	56	55	56	4	78	41	
23	14	3	15	17	4	16	14			
24	14	0	90	65	47	55	15	190	48	
25	31	1	15	4	16	16				
26	109	1	15	54	15	15	54	120		
27	31	0	14	44	16	56	50	16	16	67
28	31	1	44	17	90	16	44	15	17	55
29	10	2	15	54	54	44	15			
30	30	0	44	16	62	54				

31	105	0	14	14	62	56	45	44	54	
32	10	1	15	14	54	14	14	16	15	44
33	10	2	16	54	4	14	278			
34	31	3	56	45	123	15	54	15		
35	30	0	44	15	54	27				
36	32	1	54	84	54	44	15	15		
37	30	2	67	56	15	34	56	56	44	
38	10	3	16	92	56	14	56	4	56	
39	14	1	14	44	15	54	44	14	15	
40	32	3	4	54	4	54	15	54	189	90
41	31	1	14	12	74	44	15	17	1	
42	105	0	16	15	4	54	49	44		
0	0									
0	0									
0	0									
43	185	1	4	14	15	90	15	150		
44	10	3	14	16	15	54	4			
45	25	2	57	4	44	15	5	85		
46	13	0	44	25	4	15	56	44	15	54
47	32	1	54	79	128	16	54	4	44	
48	32	1	56	97	43	14	55	16	59	
49	14	2	14	14	45	12	14	56	15	
0	0									
0	0									
50	30	0	44	56	15	12				
51	10	0	15	15	56	15	16	15	16	
52	112	1	15	180	54	14	44			
53	31	2	56	14	14	16	4	87	16	
54	14	3	4	54	56	94	16	12	15	
55	10	0	48	16	14	16	54	54	16	16
56	30	1	56	56	54	16	9	54		
57	30	1	4	900	54	4	15	200	4	89
58	14	2	4	15	44	190	44	15	55	14
59	10	2	15	44	15	14	44	4	14	14
60	30	3	55	15	16	12				
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
61	112	3	15	5	16	56	56	14		
62	105	0	14	46	15					
63	30	0	90	78	16	12	14	56	56	
64	56	1	78	54	54	180	44	16	44	96
65	14	2	14	14	15	17	44			
66	31	2	54	12	56	15	16	174		
67	32	3	12	45	56	15	85			
68	10	3	16	15	89	15	16	12		
69	10	1	4	56	14	16	55	15	18	
70	31	1	44	4	56	56				
71	10	2	14	15	14	15	44			
0	0									
72	31	2	15	15	4	67	4	56		
73	14	2	16	56	14	4	4	15		
74	10	3	15	14	14	189	63	42	15	
75	14	3	4	55	44	15	55	14	14	4
76	32	0	56	14	55	55	56	44	15	
77	10	0	84	44	184	56	54	314		
78	31	1	55	14	14	54				
79	31	1	56	14	55	53	1	4	14	
80	30	0	54	44	55	4	54	54		
81	30	1	44	16	19	55	15	16	16	4
82	31	2	55	44	14					
83	14	1	56	14	16	14	14	54		
84	32	3	56	4	55	4	17			
85	10	1	15	55	44	15	55	14		
86	105	0	55	56	4	44	55			
87	10	2	16	56	44	14	15	55		
88	10	1	15	56	16	16	16	55		
89	10	2	15	56	14	55	15			

90	30	3	47	44	54	4	54	16		
91	105	2	14	14	14	56	55	47	56	
92	10	0	15	67	54	44	56	54	12	14
93	30	3	55	15	29	54	44	16		
94	105	1	56	4	56	4	55	44	55	
95	79	3	120	15	55	56	124			
96	31	2	55	16	63	55	16	12	4	
97	12	3	4	52	23	14	16	14		
98	30	3	56	15	15	44	15			
99	10	3	17	54	14	54				
100	10	0	44	56	44	16	16			
101	31	0	55	56	15	54	14			
102	28	1	54	56	16	4	14	55		
103	31	0	54	4	40	12	54	56	44	
104	14	2	15	17	55	16				
105	14	0	16	55	16	14	56	55	16	
106	14	3	56	15	15	16				
107	10	0	64	12	89	14	54	16		
108	14	0	44	4	16	55	14			
109	14	1	16	14	14	54	56	64		
110	10	2	74	56	56	15	4	14		
111	31	3	56	16	15	4	55	24	55	
112	30	3	15	66	15	94	14	54	45	
113	15	2	49	55	16	15	16			
114	14	1	4	56	54	49	16	14	56	
115	105	0	54	15	44	15				
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									
116	32	2	16	45	55	15	190	14	56	
117	14	2	16	56	15	56	55	44	55	56
118	31	2	4	15	14	44	15	55		
119	54	2	14	55	49	55	40	56	4	
120	10	2	14	15	4	11	4			
0	0									
0	0									
0	0									
0	0									
121	31	1	55	4	56	4	55	14	54	
122	10	1	16	56	57	44	63	47	54	56
123	10	1	15	56	16	14	15	55		
124	10	1	16	55	56	56	16	44	67	14
125	25	1	56	4	35	4	56	15	15	
126	112	0	14	17	15	54	56			
127	32	2	56	57	15	13	87	14	44	
128	30	1	44	56	15	44	56	1	4	
129	105	0	54	16	56	48	56	54	14	
130	112	0	44	15	14	44	54			
131	32	0	56	15	81	54	1	44	17	
0	0									
132	31	1	14	44	54	4	54	56		
133	10	0	14	16	56	15				

134	11	2	33	80	56					
0	0									
135	32	2	56	16	81	67	56	56	16	
136	13	3	16	12	23	4	44	14	14	44
137	14	2	16	54	54	56	14	81		
138	14	3	56	44	16	54	33	87	56	
0	0									
0	0									
0	0									
139	13	3	14	17	45	4	56	56	16	
140	14	3	180	4	56	14	15	4	89	
141	32	1	56	54	55	56	44			
142	9	2	81	56	44	56	44	56	54	
0	0									
0	0									
0	0									
143	10	1	14	54	56	72	19	15		
144	98	2	79	44	56	54	2	14		
145	30	0	30	16	14	25	4	14		
146	30	0	44	16	16	28	6			
147	31	1	54	4	90					
148	14	0	4	55	12	15				
149	105	0	15	14	14	15	44			
150	31	2	4	15	89	54	16	54	56	
151	28	3	56	56	16	14	1	20	12	4
152	14	1	54	18	15	5	22	44	19	44
153	14	0	16	54	14	56	44			
154	32	2	56	54	67	56	26	189	56	15
155	10	2	14	4	89	30	29	44	56	
156	14	0	16	16	56	44				
157	78	1	12	34	14	54	56	67		
158	32	2	44	16	78	54	4	15	28	
159	30	3	54	56	40	190	4	54	44	
160	3	1	42	54	15	17	56	56	16	12
0	0									
161	10	2	4	4	178	10	17	46	6	6
162	32	2	56	16	15	4	78	50	14	
163	14	1	44	12	57	44	56	56	52	
164	14	3	16	122	44	15	54	16	54	
165	25	3	47	129	55	56	56	56	57	
166	32	2	56	15	60	56	44	15	55	
167	49	0	58	14	56					
168	32	1	56	16	87	39				
0	0									
0	0									
0	0									
0	0									
0	0									
169	105	1	14	66	12	189	70	68		
170	12	1	4	4	72	72	15	12		
171	167	0	44	14	14	14				
172	30	0	12	32	4					
0	0									
173	31	3	72	30	54	16	15	14	2	
174	30	2	87	55	4	18	45	5	4	
0	0									
175	31	2	98	4	6	15	14	4	14	16
0	0									
176	30	3	54	45	4	12	54	120	16	
177	10	3	14	92	230	44	14	14	14	
178	30	1	54	56	16	44	56			
179	14	3	16	14	20	67	22	190	588	
0	0									
0	0									
180	165	0	55	56	4	15				
181	10	0	14	4	12					
182	105	3	14	15	4	279	30	56	16	56
183	14	2	54	120	32	14	4	278	14	
184	31	0	4	56	34	36	44	54		
0	0									
0	0									
0	0									
0	0									
0	0									
0	0									

0	0									
185	45	1	97	56	56	56	60	42	90	
186	32	1	12	14	18	14	44	4		
187	89	1	92	14	44	46	56			
188	14	2	16	14	72	90	4	50	15	
189	30	1	26	56	1	52	90	14	5	56
190	30	3	44	4	14	67	76	14	127	
191	112	1	68	56	56	44	56	56	56	16
192	16	0	45	60	4	71	44	21	280	
193	105	1	75	62	88	15	27	16	4	
194	3	2	64	6	16	56	15	56	28	
195	14	2	44	66	56	67	5	51		
196	3	1	70	4	99	44	72			
197	10	2	16	88	5	320	14	4	74	10
198	45	3	123	54	56	16	90	15	76	
199	105	0	44	14	55	1	19	56	55	
0	0									
0	0									
0	0									
0	0									
200	14	2	4	56	280	54	56	89	4	
201	32	2	16	15	15	14	4			
202	105	3	55	14	15	10	16	47	4	46
203	31	3	4	57	35	14	56	4	14	
204	30	1	54	56	44	16	15	44	17	14
205	30	1	44	15	55	20	90	90	55	1
206	5	0	22	34	24	16	17	56	44	56
207	31	0	54	4	14	54	55	4	56	
208	14	1	16	63	34	32	26	44	56	51
209	10	2	14	34	23	16	56	56	4	17
210	5	1	36	54	15	47	34	40	89	42
211	90	3	45	44	54					
212	32	3	56	55	46	89	50	25	15	
213	14	2	44	15	56	56	52	16	14	
214	18	1	4	44	120	56	28	16		
0	0									
0	0									
0	0									
215	112	2	14	4	57	29	60	289	62	26
216	30	1	44	16	15	56	16			
217	30	2	55	14	12	56	16	1		
218	5	2	70	29	72	29	15	44	74	14
219	10	2	16	56	80	14	44	76	55	
220	31	3	56	1	39	2	79	14	14	57
221	5	2	4	290	44	230	56	15		
222	5	0	10	4	54	1	12			
223	56	3	78	14	4	90	14	16		
224	10	3	7	6	59	5	20	4		
225	25	1	60	89	78	4	24	90	15	
226	105	3	54	54	16	26	97	56	54	
227	10	3	1	30	14	54	79	55	32	
228	25	2	12	34	14	16	14	56	43	34
0	0									
229	30	3	55	56	15	44	16	54	15	18
230	5	0	36	56	35	57	16	81	14	56
231	30	3	44	40	15	4	118			
0	0									
232	28	1	90	42	15	57	16			
0	0									
233	5	2	44	16	55	15	14			
234	5	1	46	56	38	17	16	56	50	
0	0									
235	14	0	54	4	56	56	54	4	52	

APPENDIX D:

MEMSTAT OUTPUT FILE

This appendix gives the system specification that produces the "memstat" output file, and shows a sample "memstat" file to which information related to system utilization was written at every 500 time units during the executing of the simulation. The fields of this file are: <current clock> <allocated memory units> <free memory units> <number of processes in the job queue> <number of processes in the ready queue> <number of processes in the blocked queue> and <number of processes delivered>.

Specification:

- Scheduling Algorithm : FCFS
- Memory Size : 512 units (mininum :12 units)
- Degree of Multiprogramming : 15
- I/O Service Time : 10

memstat file

Stat. Time	Allocated Mem.	Free Mem.	Job_Q	Blocked_Q	Ready_Q	Jobs Done
528	183	329	0	1	4	0
1207	510	2	1	1	9	0
1515	510	2	1	1	9	0
2084	510	2	1	1	9	0
2641	510	2	1	1	9	0
3098	506	6	1	1	8	1
3522	506	6	1	1	8	1
4101	509	3	6	1	8	2
4611	509	3	6	1	8	2
5024	510	2	4	1	8	4
5508	500	12	4	1	8	5
6110	509	3	3	1	9	6
6531	512	0	1	1	10	8
7039	509	3	0	1	10	9
7602	506	6	1	1	13	10
8499	504	8	2	1	14	11

8513	504	8	2	1	14	11
9003	494	18	2	0	14	12
9813	494	18	4	1	14	13
10071	481	31	4	0	14	14
10539	511	1	3	1	14	14
11157	449	63	3	0	14	17
11539	480	32	2	1	14	19
12004	491	21	1	1	14	22
12507	491	21	1	1	14	22
13027	487	25	1	0	14	25
13545	504	8	2	1	13	26
14043	492	20	3	1	14	29
14504	496	16	4	1	14	30
15011	486	26	4	1	13	31
15552	495	17	3	1	14	33
16040	386	126	3	0	14	34
16515	498	14	2	1	14	37
17184	498	14	4	1	14	39
17551	482	30	2	1	14	42
18052	448	64	2	1	13	44
19208	478	34	5	1	14	45
19302	478	34	5	1	14	45
19538	496	16	5	1	14	46
20033	391	121	5	0	14	49
20504	503	9	4	1	14	49
21178	486	26	3	1	14	53
21685	512	0	2	1	14	54
22001	502	10	2	0	14	55
22553	481	31	2	0	14	59
23012	496	16	1	0	14	64
23543	510	2	3	1	14	65
24006	500	12	3	0	14	66
24544	449	63	1	1	14	67
25022	376	136	1	1	14	69
25505	450	62	1	1	14	70
26051	369	143	1	0	14	73
26505	368	144	1	0	14	75
27033	490	22	3	1	14	78
27501	465	47	3	1	14	81
28015	512	0	1	1	14	83
28550	502	10	1	1	13	84
29017	504	8	3	1	13	87
29543	506	6	3	1	14	90
30021	508	4	1	1	14	92
30529	366	146	1	1	14	96
31007	371	141	1	1	14	98
31509	347	165	1	0	13	101
32005	490	22	0	0	12	103
32514	449	63	0	1	9	105
33005	472	40	8	0	14	111
33527	498	14	5	1	14	113
34001	498	14	5	2	13	113
34516	422	90	4	1	14	115
35067	422	90	4	1	14	115
35514	465	47	2	1	14	117
36002	468	44	2	1	14	119
36502	454	58	2	1	14	121
37049	487	25	1	1	14	124
37603	483	29	0	1	13	128
38064	491	21	1	1	14	130
38552	491	21	1	1	14	130
39004	460	52	1	0	14	131
39544	473	39	1	1	14	132
40010	486	26	2	1	14	134
40516	456	56	1	1	14	138
41045	490	22	2	1	14	143
41516	503	9	0	1	13	146
42041	511	1	8	1	14	147
42503	511	1	8	1	14	147
43179	511	1	6	1	14	149
43518	438	74	5	1	14	150
44007	366	146	3	1	14	152
44502	288	224	3	0	14	153
45006	452	60	1	1	14	156
45555	438	74	1	1	13	160

46033	475	37	1	1	14	161
46505	446	66	1	1	13	163
47001	422	90	3	1	14	165
47515	425	87	3	0	14	167
48002	438	74	3	0	14	168
48505	457	55	4	1	14	170
49013	477	35	3	1	14	172
49536	494	18	4	1	14	175
50034	471	41	4	1	14	177
50706	450	62	4	1	14	178
51384	436	76	4	0	14	179
51739	481	31	4	1	14	179
52064	450	62	5	1	13	180
52509	481	31	6	1	14	182
53065	436	76	6	0	14	183
53517	505	7	5	1	14	185
54091	475	37	5	0	14	186
54512	481	31	5	1	14	187
55027	488	24	4	1	14	190
55565	506	6	5	1	14	192
56017	488	24	5	1	14	193
56528	488	24	5	1	14	193
57014	481	31	6	1	14	195
57542	471	41	6	1	14	197
58018	471	41	6	1	14	197
58606	471	41	6	1	14	197
59038	504	8	4	1	14	201
59564	504	8	3	1	14	203
60006	505	7	4	1	13	209
60574	422	90	4	1	12	211
61018	457	55	4	1	14	212
61531	457	55	4	1	14	212
62011	457	55	4	1	14	212
62504	480	32	4	1	14	214
63046	379	133	4	0	14	217
63578	469	43	2	1	11	221
64006	354	158	2	0	10	223
64507	471	41	1	0	9	225
65066	403	109	1	1	4	229
65531	382	130	0	0	3	232

====================== Final Values =====================================

65813	0	512	0	0	0	235

APENDIX E:

JOBSTAT OUTPUT FILE

This appendix shows a sample "jobstat'' file that is automatically created when the simulation is finished. When a process is terminated, the following information is written to this file: <Process ID> <Time the process entered the system> <Time the process is leaving the system> <Execution time> <Turnaround time>.

jobstat file

```
ID:   4  Entered:      0  Left:   2658  Execution:   341  TAT:  2658
ID:   3  Entered:      0  Left:   3741  Execution:   798  TAT:  3741
ID:   5  Entered:      0  Left:   4668  Execution:   872  TAT:  4668
ID:   6  Entered:    861  Left:   4698  Execution:   273  TAT:  3837
ID:   1  Entered:      0  Left:   5080  Execution:   459  TAT:  5080
ID:   2  Entered:      0  Left:   5720  Execution:   778  TAT:  5720
ID:   8  Entered:    861  Left:   6175  Execution:   291  TAT:  5314
ID:  11  Entered:    861  Left:   6330  Execution:   924  TAT:  5469
ID:   7  Entered:    861  Left:   6587  Execution:   340  TAT:  5726
ID:  10  Entered:   4668  Left:   7264  Execution:   482  TAT:  2596
ID:   9  Entered:    861  Left:   7753  Execution:   807  TAT:  6892
ID:  19  Entered:   5720  Left:   9003  Execution:   111  TAT:  3283
ID:  17  Entered:   3741  Left:   9057  Execution:   418  TAT:  5316
ID:  20  Entered:   6330  Left:  10071  Execution:   344  TAT:  3741
ID:  18  Entered:   5080  Left:  10773  Execution:   311  TAT:  5693
ID:  25  Entered:   7264  Left:  10995  Execution:    51  TAT:  3731
ID:  13  Entered:   4698  Left:  11157  Execution:   656  TAT:  6459
ID:  14  Entered:   6175  Left:  11248  Execution:   973  TAT:  5073
ID:  12  Entered:   5720  Left:  11485  Execution:   863  TAT:  5765
ID:  23  Entered:   7264  Left:  11611  Execution:    66  TAT:  4347
ID:  16  Entered:   6587  Left:  11782  Execution:   221  TAT:  5195
ID:  15  Entered:   6330  Left:  11942  Execution:   278  TAT:  5612
ID:  30  Entered:  10071  Left:  12767  Execution:   176  TAT:  2696
ID:  29  Entered:   9003  Left:  12972  Execution:   182  TAT:  3969
ID:  22  Entered:   7264  Left:  13027  Execution:   305  TAT:  5763
ID:  24  Entered:   7264  Left:  13131  Execution:   510  TAT:  5867
ID:  35  Entered:  11157  Left:  13708  Execution:   140  TAT:  2551
ID:  27  Entered:   7753  Left:  13875  Execution:   279  TAT:  6122
ID:  28  Entered:   7753  Left:  13930  Execution:   298  TAT:  6177
ID:  33  Entered:  10995  Left:  14436  Execution:   366  TAT:  3441
ID:  32  Entered:   9057  Left:  14955  Execution:   186  TAT:  5898
ID:  34  Entered:  10995  Left:  15114  Execution:   308  TAT:  4119
ID:  36  Entered:  11248  Left:  15205  Execution:   266  TAT:  3957
ID:  26  Entered:  11942  Left:  16040  Execution:   273  TAT:  4098
ID:  21  Entered:  11485  Left:  16088  Execution:   562  TAT:  4603
ID:  37  Entered:  11611  Left:  16147  Execution:   328  TAT:  4536
ID:  38  Entered:  11782  Left:  16444  Execution:   294  TAT:  4662
ID:  44  Entered:  13764  Left:  16817  Execution:   103  TAT:  3053
```

ID:	39	Entered:	12767	Left:	16995	Execution:	200	TAT:	4228
ID:	41	Entered:	13131	Left:	17214	Execution:	177	TAT:	4083
ID:	45	Entered:	13764	Left:	17375	Execution:	210	TAT:	3611
ID:	50	Entered:	15205	Left:	17441	Execution:	127	TAT:	2236
ID:	40	Entered:	12972	Left:	17649	Execution:	464	TAT:	4677
ID:	47	Entered:	13930	Left:	17996	Execution:	379	TAT:	4066
ID:	49	Entered:	14436	Left:	18278	Execution:	170	TAT:	3842
ID:	46	Entered:	13875	Left:	19400	Execution:	257	TAT:	5525
ID:	51	Entered:	15085	Left:	19627	Execution:	148	TAT:	4542
ID:	48	Entered:	15114	Left:	19806	Execution:	340	TAT:	4692
ID:	42	Entered:	16088	Left:	20033	Execution:	182	TAT:	3945
ID:	31	Entered:	16040	Left:	20612	Execution:	289	TAT:	4572
ID:	60	Entered:	17649	Left:	20683	Execution:	98	TAT:	3034
ID:	53	Entered:	16147	Left:	20699	Execution:	207	TAT:	4552
ID:	54	Entered:	16444	Left:	20883	Execution:	251	TAT:	4439
ID:	56	Entered:	17214	Left:	21295	Execution:	245	TAT:	4081
ID:	55	Entered:	16817	Left:	22001	Execution:	234	TAT:	5184
ID:	65	Entered:	18308	Left:	22059	Execution:	104	TAT:	3751
ID:	58	Entered:	16995	Left:	22167	Execution:	381	TAT:	5172
ID:	67	Entered:	19806	Left:	22464	Execution:	213	TAT:	2658
ID:	57	Entered:	17375	Left:	22553	Execution:	1270	TAT:	5178
ID:	59	Entered:	17441	Left:	22581	Execution:	164	TAT:	5140
ID:	66	Entered:	19400	Left:	22755	Execution:	327	TAT:	3355
ID:	52	Entered:	20033	Left:	22855	Execution:	307	TAT:	2822
ID:	70	Entered:	20699	Left:	22946	Execution:	160	TAT:	2247
ID:	68	Entered:	19627	Left:	23012	Execution:	163	TAT:	3385
ID:	63	Entered:	18308	Left:	23068	Execution:	322	TAT:	4760
ID:	71	Entered:	20883	Left:	24006	Execution:	102	TAT:	3123
ID:	61	Entered:	20612	Left:	24048	Execution:	162	TAT:	3436
ID:	62	Entered:	22855	Left:	24658	Execution:	75	TAT:	1803
ID:	69	Entered:	20683	Left:	24880	Execution:	178	TAT:	4197
ID:	78	Entered:	22581	Left:	25203	Execution:	137	TAT:	2622
ID:	73	Entered:	22074	Left:	25637	Execution:	109	TAT:	3563
ID:	82	Entered:	24048	Left:	25784	Execution:	113	TAT:	1736
ID:	64	Entered:	21295	Left:	26051	Execution:	566	TAT:	4756
ID:	72	Entered:	22464	Left:	26191	Execution:	161	TAT:	3727
ID:	77	Entered:	22553	Left:	26505	Execution:	736	TAT:	3952
ID:	74	Entered:	22167	Left:	26705	Execution:	352	TAT:	4538
ID:	83	Entered:	23012	Left:	26876	Execution:	168	TAT:	3864
ID:	80	Entered:	23068	Left:	26930	Execution:	265	TAT:	3862
ID:	79	Entered:	22755	Left:	27065	Execution:	197	TAT:	4310
ID:	75	Entered:	22167	Left:	27159	Execution:	205	TAT:	4992
ID:	76	Entered:	22946	Left:	27189	Execution:	295	TAT:	4243
ID:	84	Entered:	24658	Left:	27554	Execution:	136	TAT:	2896
ID:	86	Entered:	25203	Left:	27931	Execution:	214	TAT:	2728
ID:	85	Entered:	24880	Left:	28218	Execution:	198	TAT:	3338
ID:	81	Entered:	24048	Left:	28577	Execution:	185	TAT:	4529
ID:	89	Entered:	26051	Left:	28696	Execution:	155	TAT:	2645
ID:	87	Entered:	25637	Left:	28860	Execution:	200	TAT:	3223
ID:	88	Entered:	25784	Left:	29140	Execution:	174	TAT:	3356
ID:	99	Entered:	27189	Left:	29349	Execution:	139	TAT:	2160
ID:	90	Entered:	26191	Left:	29440	Execution:	219	TAT:	3249
ID:	98	Entered:	27159	Left:	29758	Execution:	145	TAT:	2599
ID:	93	Entered:	26876	Left:	29884	Execution:	213	TAT:	3008
ID:	91	Entered:	26505	Left:	30077	Execution:	256	TAT:	3572
ID:	97	Entered:	27065	Left:	30155	Execution:	123	TAT:	3090
ID:	95	Entered:	27554	Left:	30334	Execution:	370	TAT:	2780
ID:	96	Entered:	26930	Left:	30474	Execution:	221	TAT:	3544
ID:	92	Entered:	26705	Left:	30852	Execution:	316	TAT:	4147
ID:	104	Entered:	28860	Left:	30902	Execution:	103	TAT:	2042
ID:	100	Entered:	28577	Left:	31081	Execution:	176	TAT:	2504
ID:	106	Entered:	29349	Left:	31261	Execution:	102	TAT:	1912
ID:	94	Entered:	27931	Left:	31509	Execution:	274	TAT:	3578
ID:	102	Entered:	28577	Left:	31618	Execution:	199	TAT:	3041
ID:	108	Entered:	29440	Left:	32005	Execution:	133	TAT:	2565
ID:	101	Entered:	29758	Left:	32166	Execution:	194	TAT:	2408
ID:	107	Entered:	29440	Left:	32400	Execution:	249	TAT:	2960
ID:	105	Entered:	29140	Left:	32534	Execution:	228	TAT:	3394
ID:	109	Entered:	30077	Left:	32787	Execution:	218	TAT:	2710
ID:	115	Entered:	31081	Left:	32802	Execution:	128	TAT:	1721
ID:	110	Entered:	30155	Left:	32831	Execution:	219	TAT:	2676
ID:	113	Entered:	30852	Left:	32907	Execution:	151	TAT:	2055
ID:	103	Entered:	29884	Left:	33005	Execution:	264	TAT:	3121
ID:	111	Entered:	30334	Left:	33412	Execution:	225	TAT:	3078
ID:	112	Entered:	30474	Left:	33471	Execution:	303	TAT:	2997

ID:	114	Entered:	30902	Left:	34057	Execution:	249	TAT:	3155
ID:	43	Entered:	31618	Left:	34260	Execution:	288	TAT:	2642
ID:	133	Entered:	32961	Left:	35285	Execution:	101	TAT:	2324
ID:	120	Entered:	32787	Left:	35388	Execution:	48	TAT:	2601
ID:	118	Entered:	32534	Left:	35707	Execution:	147	TAT:	3173
ID:	134	Entered:	34057	Left:	35946	Execution:	169	TAT:	1889
ID:	116	Entered:	32534	Left:	36073	Execution:	391	TAT:	3539
ID:	123	Entered:	32863	Left:	36189	Execution:	171	TAT:	3326
ID:	119	Entered:	32802	Left:	36581	Execution:	273	TAT:	3779
ID:	121	Entered:	32863	Left:	36705	Execution:	242	TAT:	3842
ID:	117	Entered:	32534	Left:	36938	Execution:	353	TAT:	4404
ID:	125	Entered:	33005	Left:	37065	Execution:	185	TAT:	4060
ID:	126	Entered:	34260	Left:	37241	Execution:	156	TAT:	2981
ID:	122	Entered:	32863	Left:	37351	Execution:	393	TAT:	4488
ID:	124	Entered:	32863	Left:	37423	Execution:	324	TAT:	4560
ID:	127	Entered:	33412	Left:	37728	Execution:	286	TAT:	4316
ID:	128	Entered:	33471	Left:	37786	Execution:	220	TAT:	4315
ID:	132	Entered:	35388	Left:	39004	Execution:	226	TAT:	3616
ID:	131	Entered:	35285	Left:	39378	Execution:	268	TAT:	4093
ID:	137	Entered:	36073	Left:	39683	Execution:	275	TAT:	3610
ID:	135	Entered:	35720	Left:	39913	Execution:	348	TAT:	4193
ID:	141	Entered:	37065	Left:	40086	Execution:	265	TAT:	3021
ID:	130	Entered:	37241	Left:	40212	Execution:	171	TAT:	2971
ID:	138	Entered:	36189	Left:	40316	Execution:	346	TAT:	4127
ID:	146	Entered:	37786	Left:	40461	Execution:	110	TAT:	2675
ID:	147	Entered:	39004	Left:	40660	Execution:	148	TAT:	1656
ID:	129	Entered:	36581	Left:	40674	Execution:	298	TAT:	4093
ID:	136	Entered:	35946	Left:	40718	Execution:	171	TAT:	4772
ID:	139	Entered:	36882	Left:	40900	Execution:	208	TAT:	4018
ID:	140	Entered:	36938	Left:	40989	Execution:	362	TAT:	4051
ID:	143	Entered:	37728	Left:	41060	Execution:	230	TAT:	3332
ID:	145	Entered:	37728	Left:	41130	Execution:	103	TAT:	3402
ID:	142	Entered:	37351	Left:	41392	Execution:	391	TAT:	4041
ID:	148	Entered:	39378	Left:	41683	Execution:	86	TAT:	2305
ID:	156	Entered:	40718	Left:	42714	Execution:	132	TAT:	1996
ID:	153	Entered:	40316	Left:	42818	Execution:	184	TAT:	2502
ID:	149	Entered:	40674	Left:	43474	Execution:	102	TAT:	2800
ID:	144	Entered:	40212	Left:	43532	Execution:	249	TAT:	3320
ID:	150	Entered:	39683	Left:	43660	Execution:	288	TAT:	3977
ID:	157	Entered:	40900	Left:	44502	Execution:	237	TAT:	3602
ID:	151	Entered:	39913	Left:	44562	Execution:	179	TAT:	4649
ID:	152	Entered:	40086	Left:	44748	Execution:	221	TAT:	4662
ID:	155	Entered:	40660	Left:	44804	Execution:	266	TAT:	4144
ID:	158	Entered:	41060	Left:	45238	Execution:	239	TAT:	4178
ID:	154	Entered:	40461	Left:	45253	Execution:	519	TAT:	4792
ID:	159	Entered:	41130	Left:	45297	Execution:	442	TAT:	4167
ID:	170	Entered:	41842	Left:	45495	Execution:	179	TAT:	3653
ID:	160	Entered:	40989	Left:	45612	Execution:	268	TAT:	4623
ID:	167	Entered:	44502	Left:	46309	Execution:	128	TAT:	1807
ID:	161	Entered:	41698	Left:	46505	Execution:	271	TAT:	4807
ID:	172	Entered:	44804	Left:	46521	Execution:	48	TAT:	1717
ID:	168	Entered:	44562	Left:	46909	Execution:	198	TAT:	2347
ID:	163	Entered:	42714	Left:	47386	Execution:	321	TAT:	4672
ID:	164	Entered:	42818	Left:	47515	Execution:	321	TAT:	4697
ID:	162	Entered:	43474	Left:	48002	Execution:	233	TAT:	4528
ID:	165	Entered:	43532	Left:	48067	Execution:	456	TAT:	4535
ID:	166	Entered:	43660	Left:	48134	Execution:	301	TAT:	4474
ID:	169	Entered:	44748	Left:	48573	Execution:	419	TAT:	3825
ID:	181	Entered:	46655	Left:	48597	Execution:	30	TAT:	1942
ID:	178	Entered:	46309	Left:	49337	Execution:	226	TAT:	3028
ID:	173	Entered:	45253	Left:	49367	Execution:	203	TAT:	4114
ID:	174	Entered:	45253	Left:	49371	Execution:	218	TAT:	4118
ID:	176	Entered:	45555	Left:	49624	Execution:	305	TAT:	4069
ID:	177	Entered:	45612	Left:	49844	Execution:	422	TAT:	4232
ID:	175	Entered:	45343	Left:	50106	Execution:	171	TAT:	4763
ID:	179	Entered:	46505	Left:	51384	Execution:	917	TAT:	4879
ID:	184	Entered:	47386	Left:	51904	Execution:	228	TAT:	4518
ID:	183	Entered:	46909	Left:	52082	Execution:	516	TAT:	5173
ID:	186	Entered:	48002	Left:	52433	Execution:	106	TAT:	4431
ID:	185	Entered:	47918	Left:	53065	Execution:	457	TAT:	5147
ID:	188	Entered:	48067	Left:	53159	Execution:	261	TAT:	5092
ID:	187	Entered:	49371	Left:	53417	Execution:	252	TAT:	4046
ID:	190	Entered:	48597	Left:	54091	Execution:	346	TAT:	5494
ID:	189	Entered:	48134	Left:	54476	Execution:	300	TAT:	6342
ID:	196	Entered:	49844	Left:	54708	Execution:	289	TAT:	4864

ID:	182	Entered:	48573	Left:	54885	Execution:	470	TAT:	6312
ID:	195	Entered:	49624	Left:	54973	Execution:	289	TAT:	5349
ID:	192	Entered:	49337	Left:	55361	Execution:	525	TAT:	6024
ID:	194	Entered:	49367	Left:	55403	Execution:	241	TAT:	6036
ID:	201	Entered:	52082	Left:	55905	Execution:	64	TAT:	3823
ID:	198	Entered:	51384	Left:	56791	Execution:	430	TAT:	5407
ID:	197	Entered:	50106	Left:	56816	Execution:	531	TAT:	6710
ID:	200	Entered:	52068	Left:	57041	Execution:	543	TAT:	4973
ID:	203	Entered:	52433	Left:	57321	Execution:	184	TAT:	4888
ID:	204	Entered:	53065	Left:	58620	Execution:	260	TAT:	5555
ID:	205	Entered:	53159	Left:	58725	Execution:	370	TAT:	5566
ID:	207	Entered:	54476	Left:	58849	Execution:	241	TAT:	4373
ID:	191	Entered:	53417	Left:	58982	Execution:	408	TAT:	5565
ID:	193	Entered:	54885	Left:	59186	Execution:	287	TAT:	4301
ID:	206	Entered:	54091	Left:	59343	Execution:	269	TAT:	5252
ID:	212	Entered:	55403	Left:	59579	Execution:	336	TAT:	4176
ID:	208	Entered:	54708	Left:	59674	Execution:	322	TAT:	4966
ID:	216	Entered:	56924	Left:	59858	Execution:	147	TAT:	2934
ID:	209	Entered:	54973	Left:	59939	Execution:	220	TAT:	4966
ID:	213	Entered:	55905	Left:	59954	Execution:	253	TAT:	4049
ID:	210	Entered:	55361	Left:	59996	Execution:	357	TAT:	4635
ID:	214	Entered:	56791	Left:	60043	Execution:	268	TAT:	3252
ID:	211	Entered:	58725	Left:	60284	Execution:	143	TAT:	1559
ID:	217	Entered:	57041	Left:	60575	Execution:	154	TAT:	3534
ID:	218	Entered:	57321	Left:	62117	Execution:	347	TAT:	4796
ID:	222	Entered:	59579	Left:	62412	Execution:	81	TAT:	2833
ID:	219	Entered:	58620	Left:	62559	Execution:	341	TAT:	3939
ID:	221	Entered:	59343	Left:	62830	Execution:	639	TAT:	3487
ID:	199	Entered:	58982	Left:	63046	Execution:	244	TAT:	4064
ID:	224	Entered:	59674	Left:	63050	Execution:	101	TAT:	3376
ID:	223	Entered:	59858	Left:	63172	Execution:	216	TAT:	3314
ID:	231	Entered:	60575	Left:	63322	Execution:	221	TAT:	2747
ID:	220	Entered:	58849	Left:	63488	Execution:	262	TAT:	4639
ID:	202	Entered:	59186	Left:	63766	Execution:	207	TAT:	4580
ID:	227	Entered:	59939	Left:	64006	Execution:	265	TAT:	4067
ID:	225	Entered:	59954	Left:	64021	Execution:	360	TAT:	4067
ID:	228	Entered:	60043	Left:	64507	Execution:	223	TAT:	4464
ID:	232	Entered:	62207	Left:	64540	Execution:	220	TAT:	2333
ID:	229	Entered:	60574	Left:	64558	Execution:	273	TAT:	3984
ID:	230	Entered:	60574	Left:	64614	Execution:	351	TAT:	4040
ID:	233	Entered:	62468	Left:	64714	Execution:	144	TAT:	2246
ID:	171	Entered:	64021	Left:	65084	Execution:	86	TAT:	1063
ID:	234	Entered:	62559	Left:	65423	Execution:	279	TAT:	2864
ID:	235	Entered:	62909	Left:	65531	Execution:	280	TAT:	2622
ID:	226	Entered:	63322	Left:	65702	Execution:	357	TAT:	2380
ID:	215	Entered:	63046	Left:	65784	Execution:	541	TAT:	2738
ID:	180	Entered:	65084	Left:	65813	Execution:	130	TAT:	729

APPENDIX F:

PROGRAM LISTING

```
/////////////////////////////////////////////////////////////////////////////
//
//                                const.h
//
// This head file defines the program constants and the default hardware
// specification.
//
/////////////////////////////////////////////////////////////////////////////

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <malloc.h>

#define     FINAL    1
#define     NONFINAL 0
#define     MAXBURSTCOUNT 100          // maximum # of CPU burst fixed as 100
const int degree_MP = 15;              // default degree of multiprogramming
const int maxmemory = 512;             // default maximum allocatable memory 2
const int IOTIME = 10;                 // default I/O service time is 10
enum status { START,READY,RUNNING,BLOCKED,TERMINATED };     // process states
enum Boolean { FALSE, TRUE };
enum rvalue {MOREJOBS, NOMOREJOBS, MOREMEMORY, NOMEMORY, NOBLOCKEDJOBS,
CANNOTBLOCKED, UNBLOCKED};             // function return value


/////////////////////////////////////////////////////////////////////////////
//
//                          clock.h
//
// This is the header file to implement the CPU virtual clock. The clock is
// simulated as a counter ('value' data member). 'old' data member is to store
// the last time collected the statistics concerning the system performance.
// They are collected and reported at every 500 clock units. Class clock is
// used in every scheduling algorithm with same type. Clock object are called
// from dispatcher to compute CPU clock and from scheduler to get the current
// CPU clock.
//
/////////////////////////////////////////////////////////////////////////////

class CLOCK {
private:
      long value;                      // the CPU virtual clock value
      long old;                        // clock value for last time collected
                                       // statistics
public:
      CLOCK() { value = 0, old=0 ;}    // constructor: initialize the values
      long get_value() { return value;}       // get clock value
      long get_old() { return old; }          // get value of old
```

```
	void compute_clock(long burst) { value=value+burst; } // compute clock
	void assign_clock(long currenttime) { value = currenttime; }
	void compute_old(long burst) { old=old+burst; }   // compute old value

};


////////////////////////////////////////////////////////////////////////////////
//
//                                  pcb.h
//
// This is the header file to implement object Process Control Block (PCB)
// This object represents a process which stays in main memory and store the
// information about a process such as ID, size, priority, status, bursts,
// current burst length and so on. Queue could easily be constructed based on
// PCB objects by including the pointer to another PCB. Class PCB and its
// subclasses are defined in this file.
//
////////////////////////////////////////////////////////////////////////////////

// object PCB which represents a process is created by instantiating a class
// PCB. class PCB is the base class which contains the basic members which are
// necessary to implement the most simple scheduling algorithm such as FCFS.

class PCB {
	friend class Queue;
	friend class Sorted_Queue;
	friend class sub_queue;
// data members
private:
	int id;                          // process ID
	int priority;                    // priority of process
	status state;                    // process status
	int size;                        // size of process
	int bursts[MAXBURSTCOUNT];       // bursts
	int burstcount;                  // # of bursts
	int burstoffset;                 // index of current burst
	int currentburst;                // length of current burst
	long arrivaltime;                // time when a process was loaded
	long iocomptime;                 // time when I/O completion will occur
	long exectime;                   // total run time
	PCB *next;                       // pointer to another PCB

// member functions, most member functions are defined to access and update
// the data members
public:
	PCB(char *jstr);                 // constructor
	PCB();                           // constructor
	PCB(const PCB &);                // constructor
	~PCB(void);                      // destructor
	int get_id() { return id; }
	status get_state() {return state; }
	int get_size() {return size; }
	int& get_bursts(int boffset) { return bursts[boffset]; }
	int get_burstcount() { return burstcount; }
	int get_burstoffset() { return burstoffset; }
	long get_arrivaltime() { return arrivaltime; }
	long get_iocomptime() { return iocomptime; }
	long get_exectime() { return exectime;}
	int get_priority() { return priority; }
	void print();                    // print the PCB
	void release();                  // destory the PCB
	void change_state(status st) { state=st; }
	void inc_burstoffset() { burstoffset++; }
	void comp_arrivaltime(long curtime) { arrivaltime=curtime; }
	void comp_iocomptime(long curtime) { iocomptime=curtime; }
```

```
        void comp_exectime(long curtime) {exectime += curtime; }
        PCB *get_next() { return next;}
        int get_currentburst() { return currentburst; }
        void comp_currentburst() { currentburst = bursts[burstoffset]; }
        void currentburst_makezero() { currentburst = 0;}
        void update_burst(int quantum) { currentburst -= quantum;}
};

// class ExPCB is one of subclass of class PCB. This class is defined by
// adding data member 'queue' which indicates the current subqueue where
// a process was assigned to implement multilevel queue scheduling algorithm

class ExPCB : public PCB {
        friend class Queue;
        friend class Sorted_Queue;
protected:
                int queue;                      // indicate the current subqueue
                ExPCB *next;                    // pointer to another PCB object
public:
                ExPCB(char *jstr): PCB(jstr) {  queue=0; }
                ExPCB(const PCB &) { queue= 0; }
                int get_queue() { return(queue); }
                void comp_queue() { queue++; }
                void queue_makezero(int n) {queue = n; }
                ExPCB *get_next() { return next;}

};

// class EExPCB inherited from class ExPCB is defined by adding the data
// member 'turn' to store the number of turns which a process spent in the
// current subqueue to implement multilevel feedback queue scheduling
// algorithm.

class EExPCB : public ExPCB {
        friend class Queue;
        friend class Sorted_Queue;
protected:
        int turn;                       // # of turns which a process spent in a
                                        // subqueue
        EExPCB *next;                   // pointer to another PCB objects
public:
        EExPCB(char *jstr): ExPCB(jstr) { turn=0; }
        void comp_turn() { turn++; }
        void comp_queue();
        void turn_makezero() {turn = 0; }
        EExPCB *get_next() { return next; }
        int get_turn()  { return turn; }

};


////////////////////////////////////////////////////////////////////////////////
//
//                              queue.h
//
// This is a header file to implement the queues (the ready queue, the job
// queue, the blocked queue) which are used in process scheduling. Class Queue
// and its subclasses are defined in this file.
//
////////////////////////////////////////////////////////////////////////////////


// Class Queue was defined to implement FIFO queue. The ready queue of FCFS
// and RR scheduling, the job queue, and blocked queue are created from class
// Queue. FIFO queue is easily constructed based on class PCB by including the
// pointer to another PCB.
```

```
class Queue {
protected :
      T *top;                          // header of the queue
      T *end;                          // tail of the queue
      int num;                         // number of processes in the queue
public:
      Queue();                         // constructor, initialize data members
      virtual void Enqueue(T *Node);   // enqueue operation of the queue
      T *dequeue(void);                // dequeue operation of the queue
      T *remove_pcb(int id);           // remove a process from the queue
      T *Head(void) { return top; }    // return the header
      T *Tail(void) { return end; }    // return the tail
      void print(void);          // print the elements of the queue
      int GetNumProcess() { return num;}    // return # of processes in a
queue
      void change_num(int i) { num=num+i; } // increase the # of processes
};


// This class is to implement a subqueue of multilevel queue scheduling
// algorithm. This is inherited by class Queue. The 'quantum' data member is
// added since each subqueue is scheduled by RR scheduling with the different
// quantum size in multilevel queue scheduling.

class sub_queue : public Queue
protected:
      int quantum;                     // each subqueue has its own quantum
public:
      void put_values(int m,int n ){ quantum =m; } // assign quantum size
      int get_quantum() { return(this->quantum); } // return quantum size
};

// This class is defined as a subclass of class subqueue by adding the 'turn'
// data member to class subqueue for multilevel feedback queue scheduling. In
// multilevel feedback queue, residency rule is assigned to each subqueue.
// When a process used up amount assigned to a subqueue (amount = turn*quantum
// size), the process moves the lower-level subqueue.

class Exsub_queue : public sub_queue {
protected:
      int turn;                        // turn assigned to a subqueue
public:
      void put_values(int m,int n ) {  quantum =m; turn=n ;}
      int get_turn() { return(this->turn); } // return value of turn
};



//////////////////////////////////////////////////////////////////////////////
//
//                              sortedqueue.h
//
// This file implements the sorted queue in ascending order. Sorted queue is
// used as the ready queue in the SJF and priority algorithm. This queue is
// inherited from class Queue but it has its own enqueue member function which
// overrides the parent's enqueue function.
//
//////////////////////////////////////////////////////////////////////////////

#define BURSTSIZE    2
#define PRIORITY     1

class Sorted_Queue : public Exsub_queue {
private:
      int    bywhat;
public:
      void assignbywhat(int n) { bywhat = n; }
```

```
        void Enqueue(T *cur);
};


//////////////////////////////////////////////////////////////////////////////
//
//                                  memory.h
//
// This header file is for the simulated memory. The main memory is simulated
// as a counter which decreases when a process acquires memory and increases
// when a process releases memory. At default, 512 allocable units are
// specified as an upper bound and 12 units as a lower bound.
//
//////////////////////////////////////////////////////////////////////////////

// This class defines the information and functions to manage the simulated
// memory. Memory manager is responsible for checking, acquiring, releasing,
// and reporting statistics about the memory. This class is used in different
// kinds of scheduling algorithm with same type. The total number of processes
// should be less than the degree of multiprogramming.

class Memory {
        friend class loader;
protected:
         int availmemory;              // maximum allocable units of memory
         int minmemory;                // minimum allocable units of memory
         int pcbcount;                 // total # of processes in main memory
public:
        Memory();                      // constructor, values are assigned by
                                       // default
        Memory(int n);                 // constructor, values are assigned by
                                       // user
        rvalue checksize();            // check if there is enough memory to
                                       // load a process
        Boolean acquire(int job_size); // acquire the memory when load a
                                       // process
        void release(int jsize);       // release the memory when a process
                                       // terminates
        void print(FILE *memoryfile);  // print the information about memory
        int getpcb(){ return pcbcount; } // return total # of processes in
system
        void compute_pcbcount(int i ) { pcbcount = pcbcount + i; }
                                       // compute the total number of
processes
};


//////////////////////////////////////////////////////////////////////////////
//
//                                  loader.h
//
// This is a header file to implement object loader which is responsible for
// loading processes into main memory. Loader moves all available jobs from
// the disk and the job queue to the ready queue. Loader stops when there is
// no incoming process and memory is full. Disk is simulated as input file.
// The processes in the job queue have a higher priority than any new arrival.
//
//////////////////////////////////////////////////////////////////////////////

// This class creates the object loader of all scheduling except multilevel
// queue. GoToReadyQueue function is defined as virtual function since a
// subclass's GoToReadyQueue overrides it.

class loader {
        friend class Memory;
```

```
        friend class scheduler;
protected:
        FILE *inputfile;                // simulated disk
public:
        loader();                       // constructor
        void LoadJob(Queue &jqueue, Memory &m, RQTYPE *rqueue, CLOCK cl);
                                        // load a process
        virtual void GoToReadyQueue(T *cur, RQTYPE *rq);
                                        // enter a process into the ready queue
        rvalue Status(Queue Jqueue);    // check status of disk and the job
                                        // queue
};


//////////////////////////////////////////////////////////////////////////////
//
//                              exloader.h
//
// This class inherited from class loader is defined by having the its own
// GotoReadyQueue function. This function includes the extra actions to assign
// a process into its subqueue permanently by priority of the process in
// multilevel queue scheduling.
//
//////////////////////////////////////////////////////////////////////////////

class Exloader: public loader {
        public:
                void GoToReadyQueue(T *cur, RQTYPE *rqueue);
                                // enter a process into its own subqueue

};


//////////////////////////////////////////////////////////////////////////////
//
//                              scheduler.h
//
// This is a header file to implement scheduler which dispatches a process to
// the CPU and maintains the process after execution.
//
//////////////////////////////////////////////////////////////////////////////

// This class is to implement dispatcher (a part of scheduler), which  removes
// the process from the ready queue and gives it to the CPU. This class
// creates the object dispatcher of FCFS, SJF, priority scheduling.

class Dispatcher {
public:
        T* Dispatcher::Dispatch(T *CurrentPCB, RQTYPE &Rqueue, CLOCK &cl);
                                // dispatch the process
};

// This is to implement object scheduler of non-preemptive scheduling
// including FCFS,SJF, priority scheduling. This class could be a prototyping
// of extended and complex scheduler. This class places a process on the
// blocked queue when the process request I/O. After I/O service, a process is
// moved to the ready queue by scheduler. When a process terminates, process's
// memory is released and the process is destroyed. Also the information about
// the process is reported to the jobdone file. The statistics about system
// performance are collected and reported to a memory file at every 500 clock
// units

class scheduler {
protected:
        FILE *memoryfile;
        // output file to contains the information about the system
```

```
      FILE *jobdonefile;  // output file to contains the information about the
                          // terminating process
      int  jobdonecount;              // total number of processes terminated
public:
      scheduler();                    // constructor
      void update_burst(T *cur);      // update the information of current
PCB
      void blocked(Queue &bq, RQTYPE &rq, CLOCK &cl);
                                      // place to the blocked queue
      rvalue unblocked(RQTYPE *rq, Queue &bq, CLOCK c1 );
                                      // place to the ready queue
      void report(int option, Queue jq, RQTYPE &rq, Queue bq, CLOCK c1,
Memory ml);                           // print information to the output file
      void terminate(T *cur, Memory &ml, CLOCK cl);
                                      // actions when a process leaves the
                                      // system
      void close_file();              // closes the output files
      virtual void GoToReadyQueue(T *cur, RQTYPE *rq);
                                      // enter the process to the ready queue
};


///////////////////////////////////////////////////////////////////////////////
//
//                            pcb.C
//
// This source file contains the member functions of class PCB and its
// subclasses.
//
///////////////////////////////////////////////////////////////////////////////

#include "pcb.h"

///////////////////////////////////////////////////////////////////////////////
/
//    PCB           : Constructor of class PCB
//     Purpose      : This function is used to construct an object PCB. It
//                    initialize some data member as 0 and assign the ID,
// size, priority, CPU bursts, and current burst by the input string. A
// process is in one line with the form of <ID> <size> <priority> <burst 1 ...
// burst n>. This is called from loader.LoadJob() to create a new PCB.
///////////////////////////////////////////////////////////////////////////////

PCB::PCB(char *jstr)
{
      char *tmp;

      tmp = new char[81];
      id = atoi(strtok(jstr," "));    // get the process ID
      state=START;                    // initialize the status to START
      size = atoi(strtok(NULL," "));  // get the size of a process
      priority = atoi(strtok(NULL," "));      // get the priority of a process
      burstcount = 0;
      while((tmp= strtok(NULL, " \t\n")) != NULL )  // get the CPU bursts
            bursts[burstcount++]=atoi(tmp);        // get the count of burst
      iocomptime =0;                  // initialize iocomptime to 0
      exectime = 0;                   // initialize exectime to 0
      currentburst=bursts[0];         // put the first CPU burst as current
                                      // burst
      burstoffset = 0;                // initialize burstoffset to 0
      next=NULL;                      // initialize next pointer to NULL
}

///////////////////////////////////////////////////////////////////////////////
//
//    ~PCB          : Destructor of class PCB
```

```
//      Purpose         : This function is used to destroy a PCB when the process
//                        terminates.
////////////////////////////////////////////////////////////////////////////////

PCB::~PCB(void)
{
        cout << "destorying "<<endl;
}
////////////////////////////////////////////////////////////////////////////////
//      release         : Member Function of class PCB
//      Puepose         : This function is used to destroy a PCB when the process
//                        terminates.
////////////////////////////////////////////////////////////////////////////////

void PCB::release()
{
        if (this->state == TERMINATED )
                delete(this);
}

////////////////////////////////////////////////////////////////////////////////
//      Print           : Member Function of class PCB
//      Purpose         : This function is used to print the information of a
//                        process when the process terminates.
////////////////////////////////////////////////////////////////////////////////

void PCB::print(void)
{
        cout<< "id :" <<id << " state :" << state <<endl;
        cout<< "size :" <<size << " burstcount :" << burstcount <<endl;
        cout<< "burstoffset: "<<burstoffset<<endl;
        cout<< " arrival times :" << arrivaltime <<endl;
        cout<< "currentburst :" <<currentburst<<endl;
        cout<< " ==================================="<<endl;
}

////////////////////////////////////////////////////////////////////////////////
//      CompQueue       : Member Function of class EExPCB
//      Purpose:        : This function is to used to update the subqueue for next
//                        execution when the process used the amount assigned to
//                        the current subqueue. The turn for next subqueue is
//                        assigned to 0. This is called by
//                        MLFQ_schduler.update_queue().
////////////////////////////////////////////////////////////////////////////////

void EExPCB::comp_queue()
{
                queue++;                        // update the subqueue which a process
                                                // will stay
                turn = 0;                       // initialize to 0
}



////////////////////////////////////////////////////////////////////////////////
//
//                                      queue.C
//
// This file contains the source programs about member functions of class
// Queue.
//
////////////////////////////////////////////////////////////////////////////////

#include "queue.h"
```

```
///////////////////////////////////////////////////////////////////////////////
//     Queue          : Constructor of class Queue
//     Purpose        : This is use to construct the object FIFO queue. It
//                      initializes data members.
///////////////////////////////////////////////////////////////////////////////

Queue::Queue(void)
{
       top =end = NULL;
       num=0;
}

///////////////////////////////////////////////////////////////////////////////
//     Enqueue        : Member Function of class Queue
//     Purpose        : This is use to append the PCB at the tail of the FIFO
//                      queue. It is called from loader and scheduler.
///////////////////////////////////////////////////////////////////////////////

void Queue::Enqueue(T *Node)
{
       Node->next = NULL;

       if (this->top == NULL )            // queue is empty
              this->top = this->end= Node;

       else
       {
              (this->end)->next = Node; // append at the tail
              this->end=Node;
       }
       this->num++;

}


///////////////////////////////////////////////////////////////////////////////
//     dequeue        : Member Function of class Queue
//     Purpose        : This is used to removes the process at header from
//                      queue. It returns a pointer to the PCB removed. It is
//                      called from loader and scheduler.
///////////////////////////////////////////////////////////////////////////////

T *Queue::dequeue(void)
{
       T *tmp;

       if (top != NULL )                  // queue is not empty
       {
              tmp=top;
              top=top->next;              // remove the process at header
              if(top == NULL )            // queue become empty
                     end=NULL;
              num--;                      // decrease the number
              tmp->next=NULL;
              return(tmp);
       }
       else
              return(NULL);
}

///////////////////////////////////////////////////////////////////////////////
//     print          : Member Function of class Queue
//     Purpose        : This is used to print all PCB in the queue. It traces
//                      whole queue.
///////////////////////////////////////////////////////////////////////////////

void Queue::print(void)
```

```
{
	T *tmp;

	cout<< "Queue::print :";
	cout<< num<<endl;
	tmp=top;
	while(tmp != NULL)
	{					// traverse whole queue
		tmp->print();			// print the PCB
		tmp=tmp->next;			// go to next PCB
	}
	cout<<"\n ------------------"<<endl;
}

////////////////////////////////////////////////////////////////////////////////
//	RemovePCB	: Member Function of class Queue
//	Purpose		: This is used to remove the PCB which has same ID as
//			  input. It traverse the queue until it finds a PCB which
//			  has same ID as input. It then removes this PCB and
//			  update the queue. It returns the PCB removed.
////////////////////////////////////////////////////////////////////////////////

T *Queue::remove_pcb(int id)
{
	T *cur;
	T *prev = NULL;

	cur=top;
	while(cur->id != id )
	{	// traverse the queue until finds the PCB that has input ID
		prev = cur;
		cur= cur->next;
	}
	if (prev == NULL )			// header has same ID
	{
		top = top->next;
		if ( top == NULL )
			end =NULL;
	}
	else
	{
		prev->next=cur->next;		// remove the PCB
		if (prev->next == NULL )	// update the queue
			end=prev;
	}
	num--;
	return(cur);				// return the PCB removed
}


////////////////////////////////////////////////////////////////////////////////
//
//				sortedqueue.C
//
// This is a souce file to contain the member function of class Sorted_Queue.
//
////////////////////////////////////////////////////////////////////////////////

#include "sortedqueue.h"

////////////////////////////////////////////////////////////////////////////////
//
//	Enqueue		: Member Function of class sorted_queue
//	Purpose		: This function is used to implement the enqueue operation
//			  of ordered queue. When a process is inserted to the
//			  queue, processes are sorted by priority or the CPU
```

```
//                    burst. Until finds the process which has the lower-level
//                    priority or the larger CPU burst than the inserting
//                    process, it traces queue.
//
/////////////////////////////////////////////////////////////////////////////

void Sorted_Queue::Enqueue(T *cur)
{
        T *tmp;
        int flag;

        cur->next = NULL;
        if (top == NULL )
        {
                top=end=cur;
                num++;
        }
        else
        {
                if (bywhat == PRIORITY)                    // queue is sorted by
                                                           // priority
                {
                        tmp=top;
                        flag =1;
                        while ( (tmp->priority <= cur->priority ) && (tmp !=
                        NULL))
                        {
                                flag=0;
                                if (tmp->next ==  NULL )   // insert at the tail
                                {
                                        tmp->next = cur;
                                        end = cur;
                                        num++;
                                        break;
                                }
                                else if ( (tmp->next)->priority <= cur->priority)
                                        tmp=tmp->next;     // go to the next process
                                else
                                {                          // insert at appropriate
                                                           // position
                                        cur->next=tmp->next;
                                        tmp->next=cur;
                                        num++;
                                        break;
                                }
                        }
                        if (flag == 1 )
                        {                                  // insert at header
                                cur->next =  top;
                                top= cur;
                                num++;
                        }
                }
                else if (bywhat == BURSTSIZE)
                {
                                                           // sorted by the length of
                                                           // current CPU burst
                        tmp=top;
                        flag =1;
                        while ( (tmp->currentburst <= cur->currentburst ) && (tmp
                        != NULL ))
                        {
                                flag=0;
                                if (tmp->next ==  NULL )
                                {                          // insert at tail
                                        tmp->next = cur;
                                        end = cur;
                                        num++;
```

```
                        break;
                    }
                    else if ( (tmp->next)->currentburst <= cur->
                    currentburst)
                        tmp=tmp->next;          // go to next process
                    else
                    {                           // insert at appropriate
                                                // position
                        cur->next=tmp->next;
                        tmp->next=cur;
                        num++;
                        break;
                    }
                }
                if (flag == 1 )                 // insert at header
                {
                    cur->next =  top;
                    top= cur;
                    num++;
                }
            }
        }
}


/////////////////////////////////////////////////////////////////////////////
//
//                                  memory.C
//
// This is source program to contain the member functions of class memory.
//
/////////////////////////////////////////////////////////////////////////////

#include "memory.h"

/////////////////////////////////////////////////////////////////////////////
//
//      Memory          : Constructor of class Memory without argument
//      Purpose         : This is used to create object memory. The values are
//                        given as default.
/////////////////////////////////////////////////////////////////////////////

Memory::Memory()
{
        availmemory = maxmemory;
        minmemory = 12;
        pcbcount = 0;
}

/////////////////////////////////////////////////////////////////////////////
//
//      Memory          : Constructor of class Memory with argument
//      Purpose         : This is used to create object memory. The values are
//                        given by user.
/////////////////////////////////////////////////////////////////////////////

Memory::Memory(int n)
{
        if ( n > maxmemory ) {
                cerr<< " Memory size max = 512"<<endl;
                exit(0);
        }
        availmemory = n;
        minmemory = 12;
}
```

```
/////////////////////////////////////////////////////////////////////////////
//
//      acquire        : Member Function of class Memory
//      Purpose        : This is used to acquire the memory to a process when the
//                         process is loaded. It returns FALSE when there is no
//                         enough memory to load the process, Otherwise returns
//                         TRUE. It is called from loader.
/////////////////////////////////////////////////////////////////////////////

Boolean Memory::acquire(int job_size)
{
        if (job_size > availmemory)
                return(FALSE);
        availmemory-=job_size;              // acquire memory for a process
        return(TRUE);
}

/////////////////////////////////////////////////////////////////////////////
//      checksize      : Member Function of class Memory
//      Purpose        : This is used to check if there is minimum memory to
//                         execute the system and total number of processes is less
//                         than the degree of multiprogramming. It is called from
//                         loader.
/////////////////////////////////////////////////////////////////////////////

rvalue Memory::checksize()
{

        if ( (availmemory > minmemory) && (pcbcount < degree_MP ))
                return(MOREMEMORY);
        else
                return(NOMEMORY);
}

/////////////////////////////////////////////////////////////////////////////
//      release        : Member function of class Memory
//      Purpose        : This is used to release the memory when the process
//                         terminates. It is called from scheduler.
/////////////////////////////////////////////////////////////////////////////

void Memory::release(int jsize)
{
        availmemory+=jsize;
}

/////////////////////////////////////////////////////////////////////////////
//      print          : Member Function of class memory
//      Purpose        : This is used to print the current allocable memory and
//                         allocated memory to memory file.
/////////////////////////////////////////////////////////////////////////////

void Memory::print(FILE *memoryfile)
{
        // print the allocated memory and free memory to memory file
        fprintf(memoryfile, " %12d %12d  ", maxmemory-availmemory,
availmemory);
}


/////////////////////////////////////////////////////////////////////////////
//
//                              loader.C
//
// This is a source program to contain the member functions of class loader
// and its subclass.
//
/////////////////////////////////////////////////////////////////////////////
```

```
#include "loader.h"

////////////////////////////////////////////////////////////////////////////////
//      loader          : Constructor of class loader
//      Purpose         : This is used to create object loader. When the loader is
//                        created, the input file which simulates disk is opened.
////////////////////////////////////////////////////////////////////////////////

loader::loader()
{
        if( (inputfile =fopen("in.data", "r")) == NULL)
                cerr<<"Error file open"<<endl;
}

////////////////////////////////////////////////////////////////////////////////
//      GoToReadyQueue          : (sub) Member function of class loader
//      Purpose                 : This is used to place the process on the ready
//                                queue. It is defined as virtual since the
//                                subclass's function should override it. It is
//                                called from LoadJob member function.
////////////////////////////////////////////////////////////////////////////////

void loader::GoToReadyQueue(T *cur, RQTYPE *rq)
{
        rq->Enqueue(cur);
}

////////////////////////////////////////////////////////////////////////////////
//      LoadJob         : Member function of class loader
//      Purpose         : This is used to load all available process from the disk
//                        and the job queue into the ready queue. It stops when
//                        there is no incoming process and memory is full. It is
//                        called from system.
////////////////////////////////////////////////////////////////////////////////

void loader::LoadJob(Queue &jqueue, Memory &m, RQTYPE *rqueue, CLOCK c1)
{
        T *currentPCB;
        T *readyPCB;
        T *newPCB;
        char buf[80];
        int jid,jsize;

        // traverse the job queue for all available processes candidates
        currentPCB=jqueue.Head();
        while( (currentPCB != NULL) && (m.checksize() == MOREMEMORY))
        {                               // finds the available processes
                if(m.acquire(currentPCB->get_size()) == TRUE)
                {                       // load the process from the job queue
                        readyPCB = currentPCB;
                        currentPCB=currentPCB->get_next();
                        readyPCB =jqueue.remove_pcb(readyPCB->get_id());
                        //rqueue.Enqueue(readyPCB);
                        readyPCB->comp_arrivaltime(c1.get_value());
                        readyPCB->change_state(READY);
                        this->GoToReadyQueue(readyPCB, rqueue);
                        m.compute_pcbcount(1);
                }
                else
                        currentPCB=currentPCB->get_next();
        }

        // load the process from input file
        while(m.checksize() == MOREMEMORY)
        {
                if(!fgets(buf,80,inputfile))
                        break;
```

```
            cout<<buf<<endl;
            sscanf(buf, "%d %d", &jid, &jsize);
            if (jid == 46 )
                  cout<<"ggggg\n";
            if (jsize == 0)
                  break;
            if(m.acquire(jsize) == TRUE )
            {     // load the process from disk to the ready queue
                  readyPCB = new T(buf);
                  cout<<"readyqueue ======"<<endl;
                  cout<<readyPCB->get_state()<<" : ";
                  readyPCB->change_state(READY);
                  cout<<readyPCB->get_state()<<"\n";
                  //rqueue.Enqueue(readyPCB);
                  readyPCB->comp_arrivaltime(cl.get_value());
                  this->GoToReadyQueue(readyPCB, rqueue);
                  m.compute_pcbcount(1);
            }
            else
            {     // place to the job queue to wait loading
                  newPCB = new T(buf);
                  jqueue.Enqueue(newPCB);
            }
      }
}

//////////////////////////////////////////////////////////////////////////////
//    Status          : Member Function of class loader
//    Purpose         : This is used to check if there is new arrival. It there
//                      is no arrival, it returns NOMOREJOBS. Otherwise, it
//                      returns MOREJOBS. It is called from the system to stop
//                      the simulation.
//////////////////////////////////////////////////////////////////////////////

rvalue loader::Status(Queue jqueue)
{
      if (feof(inputfile) && (jqueue.Head() == NULL))
            return(NOMOREJOBS);
      else
            return(MOREJOBS);
}

/////////////////////////////////////////////////////////////////////////////
//    GoToReadyQueue        : Member function of class ExLoader
//    Purpose               : This is used to place a process to its assigned
//                            subqueue by the priority of the process.
/////////////////////////////////////////////////////////////////////////////

void Exloader::GoToReadyQueue(T *cur, RQTYPE *rqueue)
{
      int which_queue;

      which_queue=cur->get_priority(); // assign the subqueue by priority
      //cur->queue_makezero(which_queue);
      rqueue[which_queue].Enqueue(cur);// enter a process to its subqueue
}


//////////////////////////////////////////////////////////////////////////////
//
//                            fcfsoj.h
//
// class fcfs implements First-Come, First Served scheduling algorithm.
// FCFS scheduling algorithm is a non-priority and non-preemptive algorithm.
// class fcfs would be a superclass of other objects which implement
// scheduling algorithm. This class is a composite class which consists of
// several subcomponents classes. It has the "has_a" relationship with
```

```
// subcomponets. The subcomponets classes communicate with each other.
//
/////////////////////////////////////////////////////////////////////////////

typedef class loader LOADERTYPE;
typedef class scheduler SCHEDTYPE;
typedef class Dispatcher DISPATYPE;

class fcfs {
	protected:
		CLOCK c1;		// object clock
		Memory m1;		// object memory
		Queue JobQueue;		// object job queue
		Queue blockedQueue;	// object blocked queue
		LOADERTYPE l1;		// object loader
		SCHEDTYPE sch;		// object scheduler
		DISPATYPE d1;		// object dispatcher
		RQTYPE readyQueue;	// object ready queue
	public:
		void call_system();
		void system(loader &l1, scheduler &sch, Dispatcher &d1, RQTYPE
		*readyQueue);
		virtual RQTYPE *get_ready() { return &readyQueue; }
		void report(RQTYPE &readyQueue);
		void timer_lock(int &noreadyflag, int &unblockflag);
		virtual void CPU(T *cur, RQTYPE *readyQueue);
		virtual T *choose_next(RQTYPE *rq) { return rq->Head(); }
		virtual void LOAD(Queue &JobQueue, Memory &m1, RQTYPE
		*readyQueue);

};


/////////////////////////////////////////////////////////////////////////////
//
//				fcfsoj.C
//
// This is a source code to contain the member functions of class fcfs.
// system member function is the main function and other functions are the
// sub functions called from system.
//
/////////////////////////////////////////////////////////////////////////////

#include "fcfsoj.h"

/////////////////////////////////////////////////////////////////////////////
//	call_system : Member Function
//	Purpose		: This is used to call the system in main.
/////////////////////////////////////////////////////////////////////////////

void fcfs::call_system()
{
	system(l1,sch,d1,&readyQueue);
}

/////////////////////////////////////////////////////////////////////////////
//	LOAD	: (sub) Member Function
//	Purpose : This is used to call its own loader for loading the process.
/////////////////////////////////////////////////////////////////////////////

void fcfs::LOAD(Queue &JobQueue, Memory &m1, RQTYPE *readyQueue, CLOCK c1)
{
	l1.LoadJob(JobQueue,m1,readyQueue,c1 );
}
```

```
//////////////////////////////////////////////////////////////////////////////
//     system        : Member Function
//     Purpose       : This simulates the system. It is a overall loop that
//                       accesses the memory manager, the loader, and the
//                       scheduler. The main procedures to execute the process
//                       scheduling simulation are:
//                            - loads the available processes from input file
//                            - dispatch the process stayed at header of the ready
//                              queue to the CPU and execute the CPU
//                            - places the process which requests I/O on the
//                              blocked queue.
//                            - terminates the process which executed the last CPU
//                              burst.
//                            - moves all processes whose I/O request have been
//                               completed to the ready queue for later execution
//                            - reports the informations about the system every
//                              500 time units
//                            - check the input and the job queue to finish the
//                              simulation
//                       Above procedures are continue until there is no new
//                       arrival process.
//////////////////////////////////////////////////////////////////////////////

void fcfs::system(loader &l1, scheduler &sch, Dispatcher &d1, RQTYPE
*readyQueue)
{

        int donesimulation = FALSE;
        int unblockflag = FALSE;
        int noreadyflag = FALSE;

        T *cur;

        while(donesimulation == FALSE )
        {
                if((l1.Status(JobQueue) == MOREJOBS) && (m1.checksize() ==
                MOREMEMORY))
                // load all available processes into the ready queue
                this->LOAD(JobQueue,m1,readyQueue,c1);
                //(this->l1).LoadJob(JobQueue,m1,readyQueue);
                // select the process which will be dispatched
                if ( (cur =choose_next(readyQueue)) != NULL )
                {
                        if(cur->get_currentburst() == 0 )
                                sch.update_burst(cur);
                                                // update the variable of current PCB

                        this->CPU(cur, readyQueue);      // simulate the CPU
                }
                else
                        noreadyflag= TRUE;
                // unblock all processes that have completed their I/O
                if (sch.unblocked(readyQueue, blockedQueue,c1 ) ==CANNOTBLOCKED )
                        unblockflag=TRUE;
                // when all processes stayed in the blocked queue
                timer_lock(noreadyflag,unblockflag);
                report(*readyQueue);          // output statistics every 500 time
                                              // units test for end of simulation
                if( (l1.Status(JobQueue) == NOMOREJOBS) && (m1.getpcb() == 0 ))
                        donesimulation = TRUE;
        }
        sch.report(FINAL,JobQueue, *readyQueue, blockedQueue,c1,m1);
        sch.close_file();

}
```

```
////////////////////////////////////////////////////////////////////////////////
//      CPU             : (sub) Member Function
//      Purpose         : This is a subfunction called by system to use the
//                        system's own dispatcher and scheduler.
////////////////////////////////////////////////////////////////////////////////

void fcfs::CPU(T *cur, RQTYPE *readyQueue)
{

        d1.Dispatch(cur,*readyQueue,c1);
        if(cur->get_state() == BLOCKED)
                sch.blocked(blockedQueue,*readyQueue,c1);
        else if(cur->get_state() == TERMINATED)
                sch.terminate(cur,m1,c1);

}

////////////////////////////////////////////////////////////////////////////////
//      report          : (sub) Member Function
//      Purpose         : This is a subfunction called by system to print the
//                        information every 500 clock units.
////////////////////////////////////////////////////////////////////////////////

void fcfs::report(RQTYPE &readyQueue)
{
        if(c1.get_value() > (c1.get_old()+500))
        {
                sch.report(NONFINAL,JobQueue, readyQueue, blockedQueue,c1,m1);
                c1.compute_old(500);
        }
}

////////////////////////////////////////////////////////////////////////////////
//      timer_lock      : (sub) Member Function
//      Purpose         : This is to execute I/O when all processes in the memory
//                         stay in the blocked queue for unblocking the processes.
////////////////////////////////////////////////////////////////////////////////

void fcfs::timer_lock(int &noreadyflag, int &unblockflag)
{
        T *tmp;

        if( (noreadyflag== TRUE) && (unblockflag==TRUE) )
        {
                tmp=blockedQueue.Head();
                c1.assign_clock(tmp->get_iocomptime()); // increase the clock
                unblockflag=FALSE;
                noreadyflag=FALSE;
        }
}

////////////////////////////////////////////////////////////////////////////////
//
//                                  sjfoj.C
// This file implements the SJF scheduling algorithm. SJF scheduling algorithm
// is simulated by reusing the class FCFS except the ready queue.
//
////////////////////////////////////////////////////////////////////////////////

class sjf: public fcfs {
        protected:
                Sorted_Queue readyQueue;
        public:
                virtual void system();
};
```

```
/////////////////////////////////////////////////////////////////////////////
//
//      system          : Member Function
//      Purpose         : This function is used to call the fcfs's system.
/////////////////////////////////////////////////////////////////////////////
//

void sjf::system()
{
        readyQueue.assignbywhat(BURSTSIZE);
        fcfs::system(l1,sch,d1, &readyQueue);
}



/////////////////////////////////////////////////////////////////////////////
//
//                                  priority.C
//
// This file implements the priority scheduling algorithm. Priority scheduling
// algorithm is simulated by reusing the class FCFS except the ready queue.
//
/////////////////////////////////////////////////////////////////////////////

class priority: public fcfs {
        private:
                Sorted_Queue readyQueue;
        public:
                void system();
};

/////////////////////////////////////////////////////////////////////////////
//
//      system          : Member Function of class priority
//      Purpose         : This function is used to call the fcfs's system.
/////////////////////////////////////////////////////////////////////////////
//

void priority::system()
{
        readyQueue.assignbywhat(PRIORITY);
        fcfs::system(l1,sch,d1, &readyQueue);
}



/////////////////////////////////////////////////////////////////////////////
////                                rrscheduler.h
//
// This is a header file to implement dispatcher and scheduler of RR
// scheduling algorithm. They are inherited from class dispatcher and class
// scheduler but they  have extra actions to implement preemptive scheduling.
//
///////////////////////////////////////////////////////////////////////

// class RR_scheduler inherited from class scheduler has some extra members to
// implement preemptive schedulings

class RR_scheduler : public scheduler {
        protected:
                int quantum;                            // quantum size
        public:
                RR_scheduler();                         // constructor
                int get_quantum() { return(quantum); } // return quantum number
                void update_queue(T *cur, Queue &rq);
                        // append to the ready queue after expiring the quantum
};
```

```
// class RR_Dispatcher inherited from class dispatcher has its own Dispatch
// function modified from parent's Dispatch and two function related to update
// the variables related to the process and clock.

class RR_Dispatcher: public Dispatcher {
	public:
		virtual void update_value(CLOCK &c1, T *currentPCB, int quantum);
		virtual void update_turn( T *currentPCB, int quantum) {
		cout<<"ddd\n"; }
		T *Dispatch(T *currentPCB, RQTYPE &rqueue, CLOCK &c1, int
		quantum);
};


///////////////////////////////////////////////////////////////////////////////
//
//                                  rrscheduler.C
//
// This file contains the source code to implement of dispatcher and scheduler
// of RR scheduling
//
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
//	RR_scheduler : Constructor
//	Purpose      : This file is used to create the object RR_scheduler. It
//	               gets the quantum size by the user.
///////////////////////////////////////////////////////////////////////////////

RR_scheduler::RR_scheduler()
{
	cout<<"Put the quantum";
	cin>>quantum;                          // get the quantum size
}

///////////////////////////////////////////////////////////////////////////////
//	update_queue : Member Function of RR_scheduler
//	Purpose      : This file is used to append a process to the ready queue
//	               when the CPU is preempted.
///////////////////////////////////////////////////////////////////////////////

void RR_scheduler::update_queue(T *cur, Queue &rq)
{
	cur=rq.dequeue();               // remove the process from the header
	rq.Enqueue(cur);                // append the process into the tail
}

///////////////////////////////////////////////////////////////////////////////
//	update_value : Member Function of RR_dispatcher
//	Purpose      : This function is used to update clock value, current
//	               burst and state of the process when the CPU is
//	               preempted.
///////////////////////////////////////////////////////////////////////////////

void RR_Dispather::update_value(CLOCK &c1, T *currentPCB, int quantum)
{
	c1.compute_clock(quantum);              // update the clock
	currentPCB->update_burst(quantum);      // update the current burst
	currentPCB->change_state(READY);        // update the state of process
}

///////////////////////////////////////////////////////////////////////////////
////	Dispatch     : Member Function of RR_dispatcher
//	Purpose      : This function contains some extra actions for CPU
//	               preemption. When the current CPU burst is not greater
```

```
//                      than the quantum, it uses the Dispatch function of its
//                      parent.
////////////////////////////////////////////////////////////////////////////////

T *RR_Dispather:: Dispatch(T *currentPCB, RQTYPE &rqueue, CLOCK &c1, int
quantum)
{
        int offset;

        currentPCB->change_state(RUNNING);
        offset=currentPCB->get_burstoffset();

        if ( currentPCB->get_currentburst() > quantum )
        {       // the CPU is preemted
                this->update_value(c1,currentPCB,quantum);
                return(currentPCB);
        }
        else
        {
                this->update_turn(currentPCB,quantum);
                currentPCB=Dispatcher::Dispatch(currentPCB, rqueue,c1);
                return(currentPCB);
        }
}

////////////////////////////////////////////////////////////////////////////////
//
//                                      rroj.C
//
// This file is used to implement the Round-Robin scheduling algorithm. The
// class rr inherited from class fcfs has its own dispatcher and scheduler for
// CPU preemption.
//
////////////////////////////////////////////////////////////////////////////////

class rr: public fcfs {
        protected:
                RR_Dispather d1;                        // object dispatcher
                RR_scheduler sch;                       // object scheduler
        public:
                void system();
                virtual void CPU(T *cur, RQTYPE *readyQueue);
};

////////////////////////////////////////////////////////////////////////////////
////    system          : Member Function of class rr
//      Purpose         : This function is used to call class fcfs's system
////////////////////////////////////////////////////////////////////////////////
//

void rr::system()
{
        fcfs::system(l1,sch,d1,&readyQueue);
}

////////////////////////////////////////////////////////////////////////////////
////    CPU             : Member Function of class rr
//      Purpose         : This is sub member function called from the class fcfs's
//                        system. Only this part is different as the system of
//                        parent class.
////////////////////////////////////////////////////////////////////////////////
//
void rr::CPU(T *cur, RQTYPE *readyQueue)
{

        cur=d1.Dispatch(cur,*readyQueue,c1,sch.get_quantum());
                // use its own dispatcher for the CPU preemption
```

```
        if(cur->get_state() == READY)            // when the CPU is preempted
                sch.update_queue(cur,*readyQueue);
                                                 // append to the ready queue
        if(cur->get_state() == BLOCKED)          // when the process requests I/O
                sch.blocked(blockedQueue,*readyQueue,c1);
                                                 // place on the ready queue
        else if(cur->get_state() == TERMINATED) // when the process terminates
                sch.terminate(cur,ml);
}

/////////////////////////////////////////////////////////////////////////////
//
//                              ML_scheduler.h
//
// This file is a header file to implement the dispatcher and scheduler of
// multilevel queue scheduling algorithm. In multilevel scheduling, the ready
// queue is divided into several subqueues. The priority scheduling is used
// among the subqueue and each subqueue is scheduled by the RR scheduling with
// different quantum size. Non-empty lowest numbered subqueue has the highest
// prority.
//
/////////////////////////////////////////////////////////////////////////

// class ML_scheduler has the extra data member to indicate the number of
// subqueues and different GoToReadyQueue member function for the process to
// place on the ready queue.

class ML_scheduler : public RR_scheduler {
        protected:
                int maxsubqueue;            // # of subqueues
        public:
                ML_scheduler();             // constructor
                int get_maxsubqueue() { return(maxsubqueue); }
                void GoToReadyQueue(T *cur, RQTYPE *rq);
};

// class ml_Dispatcher inherited class RR_dispatcher has extra member function
// used to select the process which is in header of non-empty highest priority
// queue.

class ml_Dispatcher : public RR_Dispather
{
        public:
                T *findnext(RQTYPE *sq, ML_scheduler ml);
                                        // select the highest priority process
};

/////////////////////////////////////////////////////////////////////////////
//
//                              ML_scheduler.C
// This file contains the source code to implement class ML_scheduler and
// ML_dispatcher.
//
/////////////////////////////////////////////////////////////////////////////

#include        "ML_scheduler.h"

/////////////////////////////////////////////////////////////////////////////
//      ML_scheduler : Constructor of class ML_scheduler
//      Purpose      : This function is used to create the scheduler of the
//                     mltilevel queue scheduling. It gets the # of subqueues
//                     from the user.
/////////////////////////////////////////////////////////////////////////////

ML_scheduler::ML_scheduler()
{
        cout<<"Enter the numner of sub queues";// get the # of subqueues
```

```
        cin>>maxsubqueue;
}

//////////////////////////////////////////////////////////////////////////////
////    GoToReadyQueue        : Member Function of class ML_scheduler
//      Purpose               : This function is used to place a process on the
//                              subqueue where the process is assigned permanently
//                              according to the priority.
//////////////////////////////////////////////////////////////////////////////

void ML_scheduler::GoToReadyQueue(T *cur, RQTYPE *rq)
{
        rq[cur->get_queue()].Enqueue(cur);      // go to assigned subqueue
}


//////////////////////////////////////////////////////////////////////////////
//      findnext        : Member Function of ML_scheduler
//      Purpose         : This function is used to search the process which is the
//                        oldest at the non-empty highest priority subqueue.
//////////////////////////////////////////////////////////////////////////////

T *ml_Dispatcher::findnext(RQTYPE *sq, ML_scheduler mlq)
{
        int i;

        for(i=0; i <mlq.get_maxsubqueue(); i ++)
        {       // search the highest priority process
                if(sq[i].Head() != NULL )
                        return(sq[i].Head());
        }
        return(NULL);
}


//////////////////////////////////////////////////////////////////////////////
//
//                                      mloj.C
//
// This file is used to implement object multilevel queue scheduling. The
// class mlqueue inherited from class rr has its own loader, dispatcher,
// scheduler and ready queue which consists of several subqueue and some
// member functions which override the parent's functions.
//
//////////////////////////////////////////////////////////////////////////////

class mlqueue: public rr {
        private:
                Exloader l1;                    // loader
                ml_Dispatcher d1;               // dispatcher
                ML_scheduler sch;               // scheduler
                RQTYPE Sq[10];                  // ready queue
        public:
                T *choose_next(RQTYPE *rq);
                virtual void CPU(T *cur, RQTYPE *rq);
                RQTYPE *get_ready() { return Sq; }
                void setup();                   // give the quantum to subqueues
                void system();
                virtual void LOAD(Queue &JobQueue, Memory &m1, RQTYPE
                *readyQueue);
};

//////////////////////////////////////////////////////////////////////////////
//      system          : Member Function of class mlqueue
//      Purpose         : This function is used to call class fcfs's system.
//////////////////////////////////////////////////////////////////////////////
```

```
void mlqueue::system()
{
	fcfs::system(l1,sch, d1, Sq);
}

//////////////////////////////////////////////////////////////////////////////
//	LOAD		: Member Function of class mlqueue
//	Purpose		: This function is used to call its own loader extended
//			  from parent's
//////////////////////////////////////////////////////////////////////////////

void mlqueue::LOAD(Queue &JobQueue, Memory &m1, RQTYPE *readyQueue)
{
	l1.LoadJob(JobQueue,m1,readyQueue);
}

//////////////////////////////////////////////////////////////////////////////
//	choose_next	: Member Function of class mlqueue
//	Purpose		: This function is used to call its own dispatcher to
//			  select the highest priority process.
//////////////////////////////////////////////////////////////////////////////

T *mlqueue::choose_next(RQTYPE *rq)
{
	T *cur;

	cur = d1.findnext(rq, sch);	// get the highest priority process
	return(cur);
}

//////////////////////////////////////////////////////////////////////////////
//	CPU		: Member Function of class mlqueue
//	Purpose		: This function is called from the system of class fcfs's
//			  to implement the CPU of multilevel queue scheduling.
//////////////////////////////////////////////////////////////////////////////

void mlqueue::CPU(T *cur, RQTYPE *rq)
{
	int which_queue;

	which_queue = cur->get_queue();
	d1.Dispatch(cur, rq[which_queue],c1,rq[which_queue].get_quantum());
	if (cur->get_state() == READY)
		sch.update_queue(cur,rq[cur->get_queue()]);
	else if(cur->get_state() == BLOCKED)
		sch.blocked(blockedQueue,rq[cur->get_queue()],c1);
	else if(cur->get_state() == TERMINATED)
		sch.terminate(cur,m1);
}

//////////////////////////////////////////////////////////////////////////////
//	setup		: Member Function of class mlqueue
//	Purpose		: This function is used to assigns the quantum to each
//			  subqueue.
//////////////////////////////////////////////////////////////////////////////

void mlqueue::setup()
{
	Sq[0].put_values(20,0);
	Sq[1].put_values(30,0);
	Sq[2].put_values(50,0);
	Sq[3].put_values(80,0);
}
```

```
////////////////////////////////////////////////////////////////////////////
//
//                                MLFQ_scheduler.h
//
// This is a header file to define object dispatcher and scheduler of
// multilevel feedback queue scheduling. They have some extra actions and
// variables to allow the movements of the processes among the subqueues.
//
////////////////////////////////////////////////////////////////////////////

// This class has its own member functions which override its parent's since
// queue and turn variables of the PCB are updated after dispatching to the
// CPU

class mlfq_Dispatcher : public ml_Dispatcher
{
      public:
             void update_value(CLOCK &cl, T *cur, int quantum);
             void update_turn(T *cur, int quantum);
};

// class MLFQ_scheduler adds the some extra actions to its parent for
// movements of the processes among subqueues.

class MLFQ_scheduler : public ML_scheduler {
      public:
             void update_queue(T *cur,RQTYPE *Rq);
             void GoToReadyQueue(T *cur, RQTYPE *rq);
};



////////////////////////////////////////////////////////////////////////////
//
//                                MLFQ_scheduler.C
//
// This file contains the source code about member functions of class
// MLFQ_scheduler and MLFQ_dispatcher.
//
////////////////////////////////////////////////////////////////////////////

#include      "MLFQ_scheduler.h

////////////////////////////////////////////////////////////////////////////
//      update_queue : Member Function of class MLFQ_scheduler
//      Purpose      : This function is used to move the process from current
//                     subqueue to another subqueue which has lower priority
//                     when the process used up # of turns assigned to the
//                     current subqueue.
////////////////////////////////////////////////////////////////////////////

void MLFQ_scheduler::update_queue(T *cur, RQTYPE *Rq)
{
      int whichqueue;

      whichqueue=cur->get_queue();       // get the current subqueue
      Rq[whichqueue].dequeue();          // remove from the ready queue

      if ((cur->get_turn() == Rq[whichqueue].get_turn()) && (cur->get_queue()
      != 3) )      // when used up # of turns assigned to the current queue
             cur->comp_queue();          // get the subqueue where the process
                                         // will stay
      Rq[cur->get_queue()].Enqueue(cur);      // place on the subqueue
obtained
}
```

```
//////////////////////////////////////////////////////////////////////////////
//     GoToReadyQueue       : Member Function of MLFQ_scheduler
//     Purpose              : This function adds the extra actions when the
//                            process place on the ready queue after I/O is
//                            completed. If the process is blocked after staying
//                            in the lowest priority subqueue, the process goes
//                            to the highest priority subqueue after finishing
//                            the I/O (aging).
//////////////////////////////////////////////////////////////////////////////

void MLFQ_scheduler::GoToReadyQueue(T *cur, RQTYPE *rq)
{
      if ( cur->get_queue() == 3)              // aging
      {
            cur->queue_makezero(0);
            cur->turn_makezero();
      }
      rq[cur->get_queue()].Enqueue(cur);
}


//////////////////////////////////////////////////////////////////////////////
//     update_value : Member Function of class mlfq_Dispatcher
//     Purpose      : This function is used to update the 'turn' variable of
//                    the PCB
//////////////////////////////////////////////////////////////////////////////

void mlfq_Dispatcher::update_value(CLOCK &c1, T *cur, int quantum)
{
      c1.compute_clock(quantum);
      cur->update_burst(quantum);
      cur->comp_turn();
      cur->change_state(READY);
}

//////////////////////////////////////////////////////////////////////////////
//     update_turn  : Member Function of class mlfq_Dispatcher
//     Purpose      : This function is used to reset the turn variable of the
//                    PCB when the current CPU burst is less than the quantum.
//////////////////////////////////////////////////////////////////////////////

void mlfq_Dispatcher::update_turn(T *cur, int quantum)
{
      if(cur->get_currentburst() < quantum )
            cur->turn_makezero();
}

//////////////////////////////////////////////////////////////////////////////
//
//                                        mlfqoj.C
//
// This file is used to implement multilevel feedback queue scheduling
// algorithm. The class mlfq inherited from class mlqueue has its own loader,
// dispatcher, scheduler, and ready queue.
//
//////////////////////////////////////////////////////////////////////////////

class mlfq :public mlqueue {
      protected:
            loader l1;                  // loader
            mlfq_Dispatcher d1;         // dispatcher
            MLFQ_scheduler sch;         // scheduler
            RQTYPE Sq[10];              // ready queue : Exsubqueue type
      public:
            void setup();
            void system();
```

```
            virtual void LOAD(Queue &JobQueue, Memory &m1, RQTYPE
            *readyQueue);
            virtual void CPU(T *cur, RQTYPE *rq);
};

//////////////////////////////////////////////////////////////////////////////
//    LOAD             : Member Function of class mlfq
//    Purpose          : This is used to call class mlfq's own loader.
//////////////////////////////////////////////////////////////////////////////

void mlfq::LOAD(Queue &JobQueue, Memory &m1, RQTYPE *readyQueue)
{
      l1.LoadJob(JobQueue,m1,readyQueue);
}

//////////////////////////////////////////////////////////////////////////////
//    setup            : Member Function of class mlfq
//    Purpose          : This is used to assign the residency rule of each queue.
//////////////////////////////////////////////////////////////////////////////

void mlfq::setup()
{
      Sq[0].put_values(20,3);
      Sq[1].put_values(30,5 );
      Sq[2].put_values(50,6);
      Sq[3].put_values(80,-1);
}

//////////////////////////////////////////////////////////////////////////////
//    CPU              : Member Function of class mlfq
//    Purpose          : This function contains the different parts of parent's
//                       system.
//////////////////////////////////////////////////////////////////////////////

void mlfq::CPU(T *cur, RQTYPE *rq)
{
      int which_queue;

      which_queue = cur->get_queue();
      d1.Dispatch(cur, rq[which_queue],c1, rq[which_queue].get_quantum());
      if (cur->get_state() == READY)
            sch.update_queue(cur,rq);
      else if(cur->get_state() == BLOCKED)
            sch.blocked(blockedQueue,rq[cur->get_queue()],c1);
      else if(cur->get_state() == TERMINATED)
            sch.terminate(cur,m1);
}

//////////////////////////////////////////////////////////////////////////////
//    system           : Member Function of class mlfq
//    Purpose          : This is used to reuse system of class fcfs.
//////////////////////////////////////////////////////////////////////////////

void mlfq::system()
{
      fcfs::system(l1,sch, d1,Sq);
}
```

VITA

Yungah Park

Candidate for the Degree of

Master of Science

Thesis: A SIMPLE SCHEDULER GENERATOR TOOL

Major Field: Computer Science

Biographical:

Personal Data: Born in Bonghwa, Korea, October 18, 1968, daughter of Mr. Jongman Park and Mrs. Hyunju Hwang Park.

Education: Received Bachelor of Science in Computer Science from Pohang University of Science and Technology, Pohang, Korea, in August 1992; completed the requirements for the Master of Science Degree at the Computer Science Department at Oklahoma State University in December 1997.

Experience: Employed by Oklahoma State University, Computer Science Department as a teaching assistant, January 1997 to August 1997.

Professional Membership: Korean-American Scientists and Engineers Association.