Scheduling Algorithmic Research

Rami Abielmona
#1029817
94.571 (ELG 6171)
Wednesday April 5, 2000
Prof. T. W. Pearce

TABLE OF CONTENTS

List of Figures

List of Tables

List Of Acronyms

Executive Summary............ 1

1.0 Introduction............ 2

2.0 Project Scope............ 7

3.0 Scheduling Algorithmic Analysis Overview............ 9
3.1 Process State Transitions............ 9
3.2 Priority Systems............ 10
3.3 Scheduling Algorithm Contestants............ 12
3.3.1 First-Come First-Served (FCFS)............ 12
3.3.2 Round-Robin (RR)............ 13
3.3.3 Shortest Time to Completion First (STCF)............ 14
3.3.4 Multi-Level Feedback Queue (MLFQ)............ 15
3.3.5 Highest Response Ratio Next (HRRN)............ 16
3.4 Evaluation Characteristics............ 16

4.0 Implementation Analysis and Overview............ 18
4.1 Implementation Assumptions............ 18
4.2 Queue Implementation Discussion............ 19
4.3 Time Simulation/Interruption............ 21
4.4 Process Simulation............ 21
4.5 Scheduler Implementation............ 22
4.6 FCFS-specific Implementation Discussion............ 23
4.7 RR-specific Implementation Discussion............ 23
4.8 STCF-specific Implementation Discussion............ 24
4.9 Implementation Issues............ 24

5.0 Simulation Results............ 26
5.1 Results and Observations............ 26

6.0 Other Scheduling Algorithms............ 29
6.1 Multi-Level Feedback Queues (MLFQ)............ 29
6.2 Lottery Scheduling............ 29

7.0 Open Research Topics............ 30

8.0 Bibliography............ 31

List of Figures

Figure 1.1 Embedded OS Architecture..…...…... 5
Figure 3.1 Process State Transition Representations..........................………... 10
Figure 3.2 Diagram Representing FCFS Scheduling.............................……… 12
Figure 3.3 Diagram Representing RR Scheduling..................................…… 13
Figure 3.4 Diagram Representing STCF Scheduling............................…...… 14
Figure 3.5 Diagram Representing MLFQ Scheduling...............................…… 15
Figure 4.1 CPU-bound vs. I/O-bound processes...…… 19

List of Tables

Table 1.1 Types of scheduling… 3
Table 3.1 Brief description of FCFS variants 12
Table 3.2 Evaluation characteristic descriptions 16
Table 5.1 Tabulated characteristics for "cpumin.dat" input 26
Table 5.2 Tabulated characteristics for "cpumed.dat" input 26
Table 5.3 Tabulated characteristics for "cpumax.dat" input 27
Table 5.4 Tabulated characteristics for "cpumaxed.dat" input 27
Table 5.5 Tabulated characteristics averages for all input set tests 27
Table 5.6 Winner's circle 28

Rami Abielmona
#1029817

List of Acronyms

CPU	Control Processing Unit
FB	Foreground-Background
FCFS	First-Come First-Served
FIFO	First-In First-Out
HRRN	Highest Response Ratio Next
I/O	Input/Output
ISR	Interrupt Service Routine
LTS	Long-Term Scheduler
MLFQ	Multi-Level Feedback Queues
MTS	Medium-Term Scheduler
NPNP	Non-Priority Non-Preemptive
PCB	Process Control Block
PNP	Priority Non-Preemptive
PP	Priority Preemptive
OS	Operating System
RR	Round-Robin
SPN	Shortest Process Next
SRT	Shortest Remaining Time
STCF	Shortest Time to Completion First
STS	Short-Term Scheduler

Executive Summary

This paper provides an analytical overview of a myriad of scheduling algorithms. Three of these algorithms were designed and implemented in an object-oriented language (C++) and they are the First-Come First-Served (FCFS), the Round-Robin (RR), and the Shortest Time to Completion First (STCF) scheduling policies. According to various testing criteria, the most efficient algorithms turned out to be the FCFS and the RR algorithms, although the latter has to be utilized with quite a small time slice. The longest and most time-consuming implementation was that of the FCFS algorithm, because all the data structures and associated functionality had to be set up from scratch. The scheduler and time tick simulators also had to be correctly designed. This allowed for a rather easy and straightforward design of the remaining two algorithms. Research was also done on a couple of other scheduling algorithms, but their implementations never materialized because of their complexity. This analysis is directly related to CPU scheduling, embedded system operating system design and priority systems.

Index Terms -- scheduling algorithms, embedded systems, dynamic priority systems, preemptive scheduling, short-term scheduler, process queues.

1.0. Introduction

In a multiprogramming system, multiple processes are stored and maintained in main memory. Each process is defined by the portion of memory that contains the program that is being executed by the process (along with its associated data), and by the contents of the **Central Processing Unit (CPU)** registers used by the process. Each process also alternates between processor usage and **input/output (I/O)** event occurrences. The latter could be a wait for an I/O to be performed or for some other external event to occur. The processor will be executing one process at a time, but will switch to another process, if the latter is waiting for an event. The processor will thus be kept busy by executing a specific process, while all other system processes are waiting (either for the processor or for an event). The previous discussion briefly introduces the key to multiprogramming: **scheduling**. The scheduling problem and this project's scope can be efficiently summarized by the following question:

> "One CPU with a number of processes. Only one process can use the CPU at a time, and each process is specialized in one, and only one task. What's the best way to organized the processes (schedule them)?" [1]

The scheduling problem is a major task that is usually handled by the operating system. There are four types of scheduling involved (refer to Table 1.1) in a multitasking system, with each solving the scheduling problem for each area of operating system functionality. The **long-term scheduler (LTS)** sets global limits on the system. If the latter is timeshared, then the LTS sets a limit on the number of users on the system at any time. If the system is a batch one, then the LTS sets a limit on the number of (I/O)/(CPU)-bound processes on the system at any time. The **medium-term scheduler (MTS)** is used to swap out processes, when the LTS admits more users/processes than the system is built to handle. The **short-term scheduler (STS)** is used to decide which available process will be executed next by the processor. As we can see, the STS is the scheduler (disk (I/O) scheduling is beyond the scope of this project) that is of most concern in this research area [1,2].

From this point on, the reference to a "scheduler" will imply that it is a STS. If not, then the type of scheduler will be specifically named. The scheduler is the module of the operating system that decides the priorities of processes and their time on the

processor. The **dispatcher**, on the other hand, is the module that defines the execution mechanism. A scheduler works in co-operation with the interrupt system to perform its task:

- The scheduler assigns the CPU to perform computation on behalf of a particular process
- The CPU can be "borrowed" from its current process by an interrupt. This is under the control of the external devices, not the scheduler - although interrupts can be disabled for a short time if need be
- When a process requests an I/O transfer, it normally becomes ineligible to use the CPU until the transfer is complete

Long-term scheduling	**The decision** to admit new processes to the system **Required** because each process needs a portion of the available memory for its code and data **Executed** infrequently, only when a new job arrives
Medium-term scheduling	**The decision** to control the number of processes that are in main memory **Required** to control the temporary removal of a process from memory **Executed** more frequently than the long-term scheduler
Short-term scheduling	**The decision** of ready process assignment to the CPU **Required** because of I/O requests **Executed** every time an IO request is made, or an IO request completion is detected, thus has to be very simple, and with a minimum overhead
I/O scheduling	**The decision** to handle a process' I/O request by the I/O device **Required** because of I/O requests **Executed** on I/O device availability

Table 1.1. Types of scheduling

The scheduler is an integral component of any system. The systems being discussed fall into various categories. The eldest system is the foreground-background (FB) system, where a single task runs continuously in the background, while real-time events cause interrupts which are serviced in the foreground, and then control is given back to the background. This type of system is known as an interrupt-based or event-

driven system. Uniprocessing systems, on the other hand, consist of a single process (usually a simple loop that periodically performs a set of functions), which polls its inputs to see if there are any events waiting to be serviced. Multitasking systems are single computers that switch their attention between several sequential tasks. Finally, a timesharing system is one that is designed to support multiple, interactive users or terminals. The schedulers discussed in this project are designed for either multitasking or timesharing systems [3].

Each of the previously mentioned systems benefits from the presence of an operating system, in terms of reduced costs and increase reliability. An operating system has three distinct functions:

1. **Convenience**: Allows for a more convenient system
2. **Efficiency**: Allows for efficient allocation of system resources
3. **Evolution:** Allows for easy integration of new functionality

A real-time operating system, on the other hand, has the same basic functions as a mainframe operating system, but with additional requirements:

- **Determinism:** The characteristic of performing operations at a predetermined time or within a predetermined time interval
- **Responsiveness:** The characteristic of the delay between submission of process requests and first response to the request
- **User Control:** The characteristic of allowing the user more control than in typical non-real-time operation systems, such as priority classes and paging or process swapping
- **Reliability:** The characteristic at with which the system responds to a fault, considering that real events are being controlled [2]

The aforementioned functions and requirements all apply to embedded real-time operating systems. An embedded operating system is smaller and simpler than typical mainframe operating systems. The major building blocks of an embedded OS are the kernel, the executive, the real world interface and the application programs (refer to figure 1.1). The kernel provides the most important facilities and most frequently used operations in the operating system. The executive provides the system resources to the processes that require them. It controls all scheduling, mutual exclusion and

synchronization activities. The application programs are the tasks written by the programmer. And, finally, the real world interfacing is mainly the software which handles the hardware of the system [1].

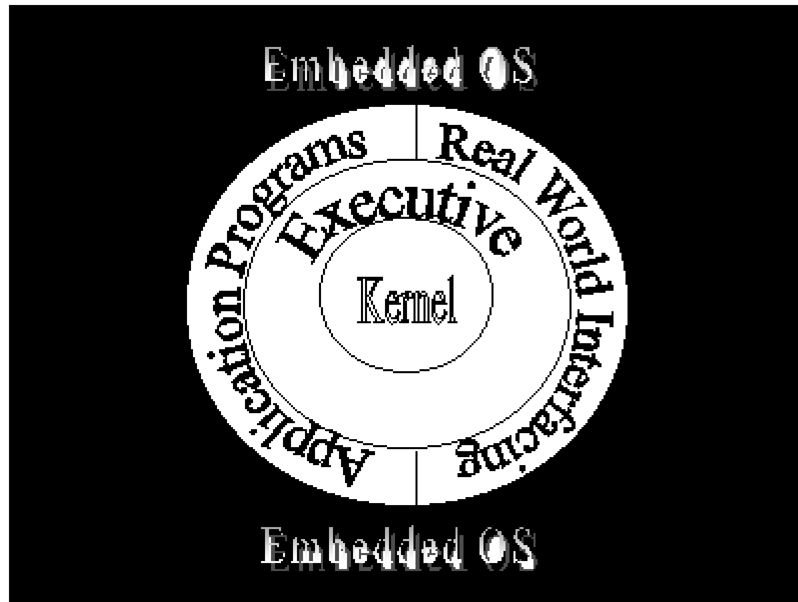

Figure 1.1. Embedded operating system architecture

Certain assumptions have been undertaken throughout the project. These will be expanded as the body of the report and simulation results are presented.

- A pool of runnable processes are contending for one CPU;
- The processes are independent and compete for resources;
- The job of the scheduler is to distribute the scarce resource of the CPU to the different processes "fairly" and in an optimal way;
- The OS is a multitasking, but not a multiprocessor, one;
- Only CPU scheduling is considered (the lowest level of scheduling);
- There are three scheduling states for each process (ready, running and blocked);
- There are two types of resources (preemptible and non-preemptible); and two types of processes (I/O-bound and CPU-bound).

A brief discussion about nomenclature is in order at this point of the introduction. The following terms have been used to represent a semi-independent program sub-division in execution: task, process and thread. A task is very similar to a process, in that it is a collection of one or more threads, and their associated system resources. A thread, on the other hand, is a dispatchable unit of work. The term process will be utilized in the remainder of the report to represent a program in execution, and encompasses a task, a thread, or even a "job". Also, it is quite worthwhile mentioning that the processes do exhibit true concurrent behaviour, since, after all, only one processor is present on the system, thus only one process can run at any one time. The processes do provide transparent concurrency, and thus are called "quasi-concurrent".

2.0. Project Scope

Scheduling topic discussions and related project area research are very integral design topics, when talking about real-time applications. The CPU scheduling algorithm that is used in the real-time system weighs heavily on the maximization of utilization and throughput, and on the minimization of waiting and turnaround times. The algorithms that are currently being used have been present for quite a long period of time. New algorithms have been recently developed and implemented. For that matter, it is quite an interesting research topic to analyze the progress that these algorithms have permitted the real-time computing world to enjoy.

This research project's goals are to analyze independent CPU scheduling algorithms, to compare them, to extrapolate the results to include hybrid models, as well as innovative CPU scheduling design techniques, and to conclude by attempting to expose the algorithms better-suited for certain applications. At the project proposal level of this research, it was stated that a major achievement would be the definition of a CPU scheduling algorithm after analysis of the already existing ones. This sub-goal has not materialized mainly due to time constraints and erroneous mathematical analysis. The algorithm that this designer had in mind did not tailor for a real-time application. It also could not be mathematically proven, and thus it is with great disappointment that it was decided to be left as an open research area.

The analysis of the various scheduling algorithms will be undertaken by implementation, simulation, characterization or simple algorithmic analysis techniques. Three crucial factors must be considered in the analysis of any scheduling policy: fairness, class distinction and efficiency. Fairness is present to allow all processes competing for the CPU to have an equal and fair chance of gaining the resource. Class distinction is present to allow the operating system to differentiate between the different classes of jobs vying for the CPU. Efficiency is present in order to maximize throughput and minimize response time, while staying within the fairness and class distinction boundaries. Along with these three factors, a scheduling policy can be analyzed by observing certain characteristics, which will be mentioned in the body of the report.

This project will be a means to a better understanding of the processor scheduling algorithms, as well as exposing all their fundamental concepts and theories. It will provide the designer/author with quite an extensive research area, in that the algorithms will have to be selectively chosen, because of time, effort and software constraints. The remainder of this paper is organized as follows. Section 3.0 introduces the scheduling algorithmic analysis by providing the reader with a brief background on related topics. Section 4.0 presents the implementation details associated with the design of the scheduling schemes. Section 5.0 presents the simulation results and observations. Section 6.0 provides a brief overview of two scheduling algorithms not implemented in this project. Section 7.0 discusses possible extensions and future research areas left open.

3.0. Scheduling Algorithmic Analysis Overview

The scheduling algorithms being discussed in this project were selected because of their affinity to short-term scheduling, as well as their application to real-time systems. Even though some are applicable for long-term scheduling, and some are not truly tailored for real-time systems, none the matter, they will be discussed because they help to provide us with a fair "playground". The policies being mentioned are: { **First-Come First-Served (FCFS)**, **Round-Robin (RR)**, **Shortest Time to Completion First (STCF)**, **Multi-Level Feedback Queue (MLFQ)**, and **Highest Response Ratio Next (HRRN)** }. The first three algorithms were chosen to be implemented, while the last three were picked to be theoretically analyzed. Along with these algorithms, a recent, innovative and specialized algorithm will be discussed later on in the report. The algorithm is called "lottery scheduling". The following sections will present in-depth looks at each of the "competing" policies, but first brief miscellaneous topic introductions are presented in order to prepare the reader for the actual descriptions of the algorithms.

3.1. Process State Transitions

The state of a process at any given time is comprised of the following minimal set: [Ready, Running, Blocked] (refer to figure 3.1).

- ***Ready:*** The process could be running, but another process has the CPU;
- ***Running:*** The CPU is currently executing the code belonging to this process;
- ***Blocked:*** Before the process can run, some external event must occur.

The external event could be a time-out (discussion of time slicing is saved for later on in the report), an interrupt or the I/O completion signal. As a process runs (eventually to completion), it goes through a series of state transitions, as shown in the figure 3.1. One other transition that was purposefully omitted is the "**preemption**" transition, which would be a line going from the 'Running' state to the 'Ready' state. This transition represents the occurrence of one of two events: either the currently running process used up its time-slice, or a higher priority process needs the CPU. Preemption allows for variations on the scheduling policies being discussed. Preemptive schemes suggest that a

running process may be forced to yield the CPU by an external event (higher priority process arrival, I/O interrupt occurrence placing a blocked process in the Ready state, or a periodic clock interrupt). Non-preemptive schemes, on the other hand, suggest that once a process starts executing, allow it to continue until it voluntarily yields the CPU (termination, I/O request or OS service call).

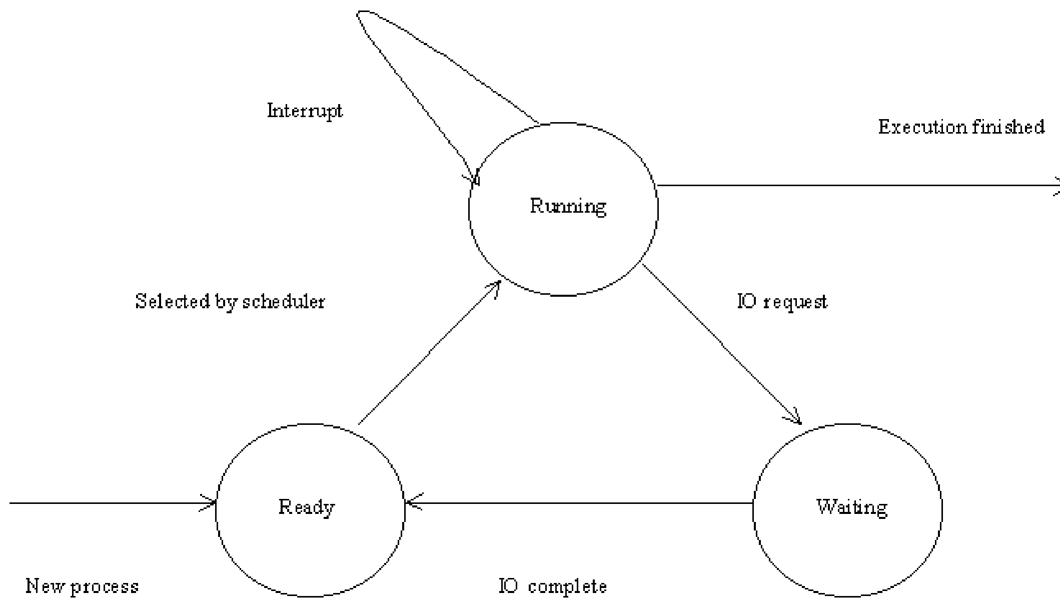

Figure 3.1. Process State Transition Representation

3.2. Priority Systems

The introduction of priority into scheduling schemes is of extreme importance. It provides a basic mechanism for dividing the system processes into high- and low-priority ones, and thus performs a rudimentary form of class distinction. In priority systems, each process is assigned a priority and the scheduler will always select the highest priority ready process. Priority queues replace the ready queue, and processes are dispatched, starting with the head of the highest priority queue. A problem with such a scenario is low-priority process starvation, in that if there is a steady stream of high-priority ready processes, the low-priority processes may not get any time on the processor. This situation is resolved by dynamically changing the priority of a process according to its age or execution history. Speaking of dynamics, there are three possible ways of assigning priorities to processes [1,2]:

1. *Statically or externally*:	Priority is assigned by some external system manager before process is scheduled
2. *Dynamically or internally*:	Priority is assigned according to an algorithm
3. *Hybrid*:	Priority is assigned by some combination of external and internal schemes

Along with priority comes a closely related topic called **timer interruption**. As soon as a process gets a hold of the CPU, a time interval begins to tick down. Once the interval expires, the process is forced to yield the CPU (if it has not already done so), before its CPU burst is complete. Unlike priority preemption, timer interruption is process-dependent and system-independent, and is used to guard against processes stuck in infinite loops, which will hang the system. For that matter, timer interruption is seen on almost all real-time operating systems, while priority preemption (which is a feature after all) is not.

Either case of preemption (timer or priority) does not come for free. With each scheduling decision, a new process may be installed on the processor. The installation process is called a ***context switch***, referring to the switching of the processor context. As a running process is made ready, its CPU registers are all saved, while as a ready process is made running, its CPU registers are all loaded. The overhead of continuously changing from one process to another could hinder the overall performance of the system, and thus must be kept at a minimum. It is worth mentioning, that even an I/O request will force the running process to yield the CPU, but this is a required event, while priority preemption, for example, is an additional feature.

If you are wondering where the CPU register values could be stored, well, the answer is in the **Process Control Block (PCB)**. Each process has a PCB associated with it. The PCB is by far, the most important data structure in an operating system. It contains all of the information about a process that is needed by the OS. Examples of what is contained in a PCB are: process identification, process priority, process local stack, process control information and process register values [2].

Rami Abielmona
#1029817

3.3. Scheduling Algorithm Contestants

There are a number of variations of the scheduling algorithms mentioned in section 3.0. They cannot all be discussed but the most intriguing ones will be mentioned in this section. These algorithms represent the set on which analysis will be performed.

3.3.1. First-Come First-Served (FCFS)

This algorithm is the simplest, and yet quite effective, policy. Otherwise known as "**First-In First-Out (FIFO)** scheduling" or "cyclic scheduling", this approach consists of one ready queue -- *queues will be discussed later; their introduction here is meant to allow the reader to make their own deductions about their implementation before presenting the author's views and techniques* -- where new processes enter at its end. The head process is given the CPU until it completes its work, or it performs an I/O transfer. The resource (CPU) is then passed onto the next waiting process in the queue (refer to figure 3.2). Many variations of this scheme exist, including non-priority non-preemptive FCFS (NPNP-FCFS), priority non-preemptive FCFS (PNP-FCFS), and priority preemptive FCFS (PP-FCFS). Refer to table 3.1 for a brief description of each.

NPNP-FCFS	Simplest implementation of scheduling algorithms
	Mostly used in timeshared systems
PNP-FCFS	Next highest priority process is picked when CPU is yielded
	Once a process grabs the CPU, it keeps it until completion
	Rarely used in real-time systems
PP-FCFS	Next highest priority process is picked when CPU is yielded
	Currently running process could be forced to yield CPU
	Most popular FCFS algorithm implementation

Table 3.1. Brief description of FCFS variants

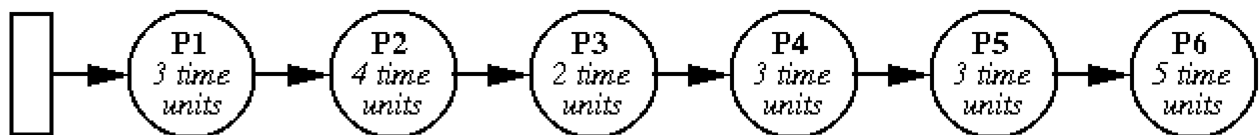

Figure 3.2. Diagram representing FCFS scheduling

FCFS scheduling presents a few problems, in that a process with a heavy computational load can monopolize the CPU, the system halts when a process gets stuck, and a new process always goes to the end of the ready queue, therefore, could end up waiting a long time before getting a crack at the CPU. All these issues are solved by the introduction of timer interruption. By limiting the time a process can run without a context switch, we effectively solve the major FCFS generated issues. This type of scheduling is called "Round-robin" scheduling [4].

3.3.2. Round-Robin (RR)

This algorithm is mostly used on timeshared systems, where the majority of users/processes have the same priority.

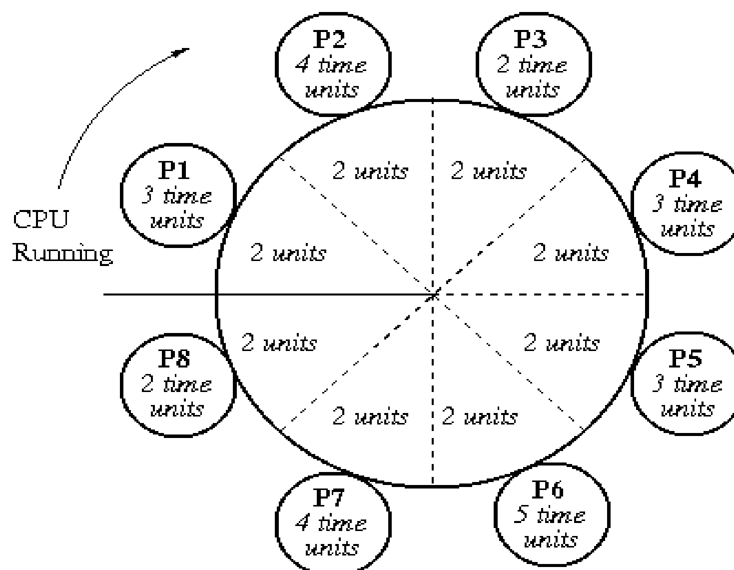

Figure 3.3. Diagram representing RR scheduling

Each process gets an equal share of the CPU time by running for one time slice, and then moving to the end of the ready queue, in a FIFO manner. The scheme allows multiple processes to execute on a "round-robin" basis (refer to figure 3.3). This scheme is basically an FCFS scheduler but with timer interruption priority. Its advantages lie in improved response time and a better use of shared resources. It does, on the other hand, leave some issues unresolved. Processes vary in importance, and thus a higher-priority

process needs more CPU time than a lower-priority process. Processes also do no always run at regular intervals (periodic vs. aperiodic, discussion to follow later), and may only run when certain real-time events occur. For this matter, RR scheduling is not popular with dynamic priority systems [4].

3.3.3. Shortest Time to Completion First (STCF)

This algorithm is used on timeshared systems to minimize response time. It has two variants: one with preemption and one without preemption. The "without preemption" variant is usually called the **Shortest Process Next (SPN)** policy, in which the process with the shortest expected processing time is selected next. The "with preemption" variant is usually called the **Shortest Remaining Time (SRT)** policy, in which the scheduler always chooses the process that has the shortest expected remaining processing time. In both variants, priorities are assigned in inverse order of time needed for completion of the entire process (refer to figure 3.4). They both also must predict the future, in that exponential averaging is used to estimate the processing time of each task. A process exceeding the resource estimation is aborted, while a process exceeding the time estimation is preempted (in SRT).

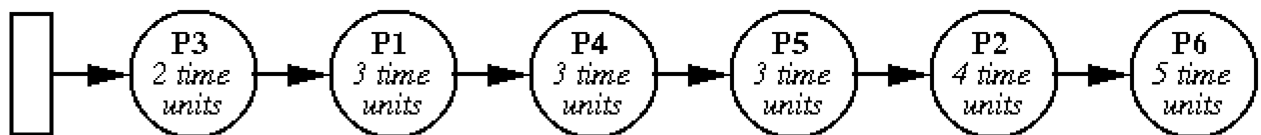

Figure 3.4. Diagram representing STCF scheduling

STCF is not biased in favor of long computational processes, as in FCFS scheduling, and does not require timer interruption, as in RR scheduling. It does on the other hand, need to store the actual value for elapsed service times in the PCB, thus greatly contributing to the malignant overhead. The SRT policy outputs better results than the SPN policy, because the former allows a short process to preempt a running long process [2,4].

3.3.4. Multi-Level Feedback Queue (MLFQ)

This algorithm is very popular in interactive systems. It resolves both efficiency and response time problems. It is also known as an "adaptive" algorithm, in that processes are always adapting to their previous execution history. This policy is mainly used if the remaining time of a process cannot be calculated for some reason, and thus turning its attention to the time spent executing. Its basic operation follows:

- A single queue is maintained for each priority level
- A new process is added at the end of the highest priority level
  - It is allotted a single time quantum when it reaches the front
- If the process uses up the time slice without blocking, then decrease its priority by one, and double its time slice for its next CPU burst
- If the process does not use up the time slice, then increase its priority by one, and half its time slice for its next CPU burst (refer to figure 3.5)

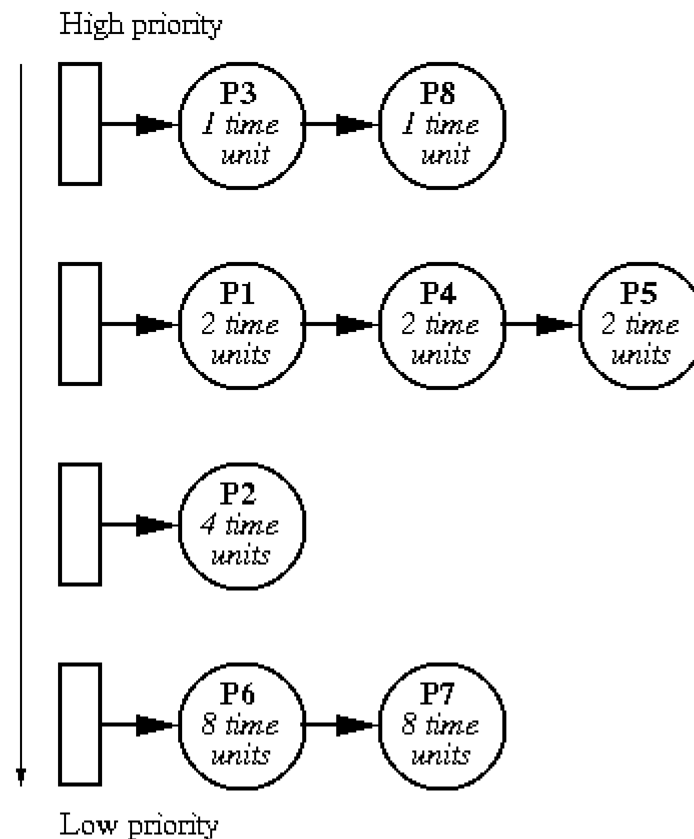

Figure 3.5. Diagram representing MLFQ scheduling

This policy may induce starvation of long processes. A solution to that problem is to move the process to a higher-priority queue after it spends a certain amount of time waiting for service in its current queue [1,4]. The UNIX scheduler is a derivation of the MLFQ scheduling algorithm. Basically, processes that have not recently used the CPU are given high-priorities, while processes that have are assigned lower-priorities. UNIX also allows its users to provide a "nice" value for each process in order to modify its priority.

3.3.5. Highest Response Ratio Next (HRRN)

This algorithm implements the "aging priority" scheme, in that as a process waits, its priority is boosted until it eventually gets to run. The priority is calculated as follows:

$$\text{Priority} = (w + s) / s$$

Where :

w = time spent waiting for the processor
s = expected service time [2]

This policy is quite beneficial in that long processes will age, and thus will eventually be assigned a higher-priority than the shorter jobs (which already have a high-priority because of the small denominator value).

3.4. Evaluation Characteristics

The following table illustrates scheduling algorithm evaluation characteristics.

Characteristic	*Description*
CPU utilization	Keep it as high as possible
Throughput	Number of processes completed per unit time
Waiting time	Amount of time spent ready to run but not running
Response time	Amount of time between submission of request and first response to the request
Scheduler efficiency	Minimize the overhead
Turnaround time	Mean time from submission to completion of process

Table 3.2. Evaluation characteristic descriptions

Each scheduling policy produces different results when speaking of the aforementioned characteristics. These discussions will ultimately help us judge between the contestants in order to attempt to notice any clear-cut patterns. As well as providing us with a common testing base, table 3.2 provides us with some of the CPU scheduling goals. Another unmentioned goal is to better resource utilization. By designing and/or implementing an efficient (according to table 3.2) scheduling algorithm, the I/O requesting processes will be better served, thus increasing the resource utilization efficiency.

4.0. Implementation Analysis and Overview

As previously mentioned, only the FCFS, RR and STCF algorithms were chosen for implementation. These three policies span quite a variety of scheduling scheme research. First, the FCFS algorithm provides the designer with the challenge of setting up the data structures, as well as an object-oriented basis for the next two algorithms. The RR policy adds a timing constraint to the whole equation, while the STCF scheme inserts priority into the system. The only remaining major aspect not implemented would be any estimates used, for example, by the SPN algorithm (see open research area section). This section will discuss the implementation issues faced during this phase of the project.

4.1. Implementation Assumptions

The following assumptions were undertaken during the implementation phase. These assumptions not only simplified the design, but also hid the details that come with a real-time operating system design. They allowed the designer to concentrate on the major aspects of the scheduling algorithm, and left the extrapolations for future research work. *Firstly*, it was assumed that there are only two types of processes (in terms of resource usage): **CPU-bound** and **I/O-bound** processes. CPU-bound processes perform lots of computation and request a small amount of I/O transfers, while I/O-bound processes request a major amount of I/O transfers, while processing the I/O results in short CPU bursts (refer to figure 4.1). *Secondly*, it was assumed that there are also only two types of resources: preemptible and non-preemptible resources. Preemptible resources are obtained by the process, which use the resource and return it back to the resource pool (e.g. processor or I/O channel), while non-preemptible resources, once obtained by the process, are not returned until the latter is done with its use (e.g. file space). *Thirdly*, the processes that are simulated (more on this later) in the system are **aperiodic tasks**, in that they have deadlines (constraints) by which the task must finish or start, or they may have constraints on both start and finish time. Periodic tasks were considered at the beginning of the project implementation, but after further review, the design was switched to better simulate a real-time environment where tasks are mostly

aperiodic ones. *Fourthly*, the simulated processes were assigned priorities by the designer (thus, simulating a system manager which statically assigns priorities to processes on system entry). These priorities were effectively not used during the FCFS and RR simulations, but were utilized in the STCF simulations. Basically, the priorities were assigned with the inverse remaining time theory in mind. A hybrid of external and internal priority allocation was used, in that after the system manager was done, the internal dynamic priority-assignment algorithm took over. *Fifthly*, as the I/O "blocked" queue is concerned, only one was simulated. The decision to not allot a queue for each I/O device was made to simplify the design and worry more about scheduler/dispatcher functionality rather than cumbersome I/O queues. The basic structure is one I/O queue that stores PCB's of processes blocked on I/O calls. The processes in the blocked queue do not have to be accessing the same I/O device, although they may be, and any synchronization issues are left at/for the I/O scheduling level. *Lastly*, the following constraints were placed on the system:

- Only 10 processes can enter the system at the STS level;
- A process can block at most 10 times;
- A queue can hold at most 10 PCB's;
- The highest priority level is 10;
- The maximum amount of time a process can be in the system for is 255 time ticks.

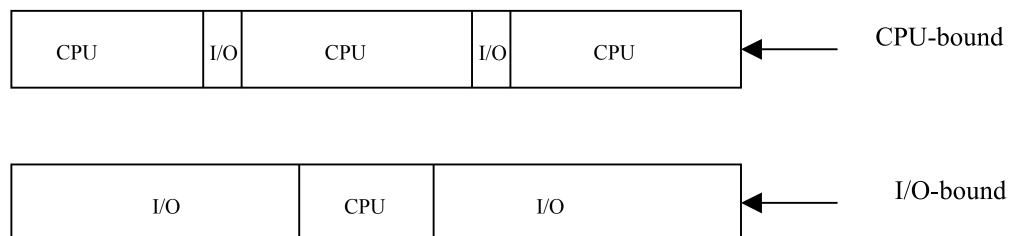

Figure 4.1 CPU-bound vs. I/O bound processes

4.2. Queue Implementation Discussion

The queue structure was implemented as a linked list of node structures. A template class definition was used, and instantiated four times, using PCB structures as

the queue element types. Four queues were utilized in the entire system to simulate the actual real-time environment. A "poolQ" represented the queue holding the pool of processes provided to the STS scope by the LTS. This queue, when called upon, fills up the "readyQ" in the STS scope. The two other queues are the "blockedQ" and the "runningQ". The former is used to hold all PCB's currently blocked on I/O requests, while the latter holds the PCB of the currently running process. Each queue element contained a pointer to a PCB and a pointer to the next node in the linked list.

```
/* Declaration of the template for the queue class */
template<class queue_element_type, int max_size=30>
class queue
        .
        .
        .
private:
        /* Data members */

                /* Declaration of the structure for each queue element (node) */
                struct q_node {
                                queue_element_type *item;    /* pointer to queue item */
                                q_node *next;       /* pointer to next item */
                            };
                q_node *front;
                q_node *rear;
                q_node *current;
                int num_items;
```

It is quite interesting to note that because of the template declaration of the queue, it was made easy to use this structure to hold any necessary linked list representation of a data structure. The "front" and "rear" pointers are used to easily add new processes to the end of the queue, and to easily dispatch a process from the front of the queue. The queue class has the typical public methods associated with a linked list: queue::queue; queue::~queue; queue::insert; queue::remove; queue::is_fifoQ_empty; queue::is_fifoQ_full; queue::getnumcount. Two additional methods were implemented and proved to be very useful during the implementation and debugging stages: queue::retrieve and queue:print. The former basically retrieves the pointer to the PCB at the head of the queue without removing it from the queue, and the latter prints out the process identification field of all the PCBs present in the queue (refer to Appendix A for

the complete source code). The queue data structure design was quite successful from the beginning, as a template declaration helped in numerous ways at the implementation stage, fortunately, previously foreseen at the design stage.

4.3. Time Simulation/Interruption

The time simulation and timer interruption mechanisms were implemented as a signal sent from the appropriate scheduler (FCFS, RR, or STCF). On reception of that signal, the "time_simulator()" function performed the following tasks:

- Updated in the PCB the remaining execution/blocking times of the running process;
- Updated in the PCB the blocking lengths of the blocked processes;
- Updated the process cycle graphs (more on this in the simulation section);
- Updated the priorities of the processes in the system (for STCF use only); and
- Updated the time slice global variable (for RR use only)

A time tick was picked to be about two seconds (the accuracy depends on the precision used in the sleep() function defined in "dos.h"). Thus, as a result, a time tick occurred every two seconds, and the time simulation mechanism was in place. As for the timer interruption, well, the update of the "time_ticks" global variables made sure that at each time tick, a timestamp is recorded (later to be used by the RR policy for time slicing). The time simulator functionality was picked so as to be generic, that is, the time simulator will service all implemented algorithms in the same way, as mentioned above. This design decision guarantees that all algorithms are serviced in exactly the same manner, just as in a real real-time operating system.

4.4. Process Simulation

The processes entering the system were input from a file ("cpu.dat"). They are read into the poolQ by the "fill_poolQ" function. Each declared process is assigned a PCB and is initialized using the inputs from the file. The priority field, as previously mentioned, is assigned according to the initial remaining processing time. The file structure is best defined by the introduction of an example process:

```
p1          defines a process with process id of 1
a3          defines a priority of 3
t20         defines the initial remaining execution time of 20
b5          defines a block at time 5 away from start of execution
c8          defines a block length of 8 for the previous block time
b18         defines a block at time 18 away from start of execution
c6          defines a block length of 6 for the previous block time
e           defines the end of the declaration of the process
```

The use of this format allows for ease of simulation of I/O-bound processes or CPU-bound processes, as will be seen in the next section. Just as a reminder that the most processes that could be defined, according to the project constraints, are ten.

4.5. Scheduler Implementation

The scheduler was implemented as a decoder block. The following pseudo-code briefly describes its functionality:

- If no process running
  - Pick a ready process to run;
- Else
  - If current running process is done
    - Remove running process from system
    - Pick a ready process to run
  - Else if current running process requests I/O
    - Suspend running process and pick a ready process to run

Also,

- If any blocked processes have completed their I/O
  - Unblock them by making them ready (and maybe even running)

The scheduler makes all scheduling decisions by calling three system functions: run(), suspend(), and resume(PCB*). The run() function basically performs a context switch, by readying the running process and running a ready process (if either exists). The suspend() function blocks the running process by placing it on the blocked queue. Finally, the resume(PCB*) function resumes the operation of a blocked process by placing it on the ready queue (or on the running queue if the latter is empty). Refer to Appendix A – schedule, run, suspend and resume declarations).

4.6. FCFS-specific Implementation Discussion

The FCFS algorithm follows the scheduler functionality (or maybe it's the other way around), because the FCFS policy was the first one implemented. Basically, the poolQ is filled up with the system processes, and then readied (by calling "get_tasks" with a source queue of poolQ and a destination queue of readyQ). Then, a while loop is entered, which only exits upon the successful check that all three queues (running, ready and blocked) are completely empty, thus guaranteeing that all processes ran to completion and proceeded to exit the system. The body of this popular while loop is very simple. The scheduler is called, and then control is given to the time simulator defined above. The process is repeated until the task set has been completely executed. At the end of the algorithm execution, the statistical results (evaluation characteristics) are calculated and output both to the screen and out to a file in memory ("results.dat"). The statistics include the following: total CPU time, total wait time, total ready time, total time, throughput, average waiting time, average turnaround time and CPU utilization.

4.7. RR-specific Implementation Discussion

The RR algorithm performs the same calls and has the same structure as the FCFS algorithm. That is, the same while loop functionality was implemented for this policy. The only difference is that the scheduler is now called with a 'R' flag, and is given a time slice period. This specific policy follows the same algorithmic procedure as the FCFS policy, **but if** it was found that the runningQ is not empty, **and** the current running process is neither done or blocking, **then** the time slice of the current running process is checked. If the process has exceeded its given time slice, and still is attempting to run, while there are other ready to run processes, then it is preempted, and a new process is installed onto the processor. To accomplish this successfully, it was needed to reset the "time_ticks" global variable every time a new ready to run process was about to be installed on the processor. Again, at the end of the execution of the last remaining process, the statistical outputs are printed to the screen and to the same file.

4.8. STCF-specific Implementation Discussion

The STCF algorithm has the same basic structure as do the previous two implemented algorithms. The difference comes into the scheduler code itself. The scheduler, once it detects that the STCF algorithm is being utilized, performs one extra functionality, which effectively simulates the priority queue: it sorts the ready queue in ascending priority (low numbers = high priorities) every time a scheduling decision is about to be made. The only time that it does not perform the priority sorting is when the RR scheduling decision is made, since the latter involves time slices, which are of no concern here. There are two variants to the STCF algorithm: one without preemption, and one with preemption. The former basically schedules a process, according to its remaining time, every time tick, but does not worry about preempting a process, once it has been handed the CPU, while the latter performs the same functionality as the former, but does preempt a process, if it finds that there is a ready process with a higher priority than the currently running one. Careful design had to be made in order to ensure that preemption occurred when it should (refer to Appendix A for the scheduler declaration).

4.9. Implementation Issues

The following sub-section describes various implementation issues encountered during the project. The major obstacle that was overcome was in the declaration of the PCB struct (refer to Appendix A, cpu.h). As we can see, the structure is filled with various control fields used by the scheduler. One of those fields, the process_cycle field was originally defined as a "char *process_cycle". Every time, the update_graphs() function would execute, the process_cycle field of every PCB on every queue received the same value. The update_graphs() function fills up process_cycle with an execution summary of the whole process, using a "-" to represent a time tick spent running, a "/" to represent a time tick spent ready to run but not running, and a "|" to represent a time tick spent blocked for I/O. The problem was debugged and was found to be a stray pointer issue. The problem was temporarily fixed by including a physical array in every PCB struct. It took 4 man-hours for this fix alone. Although the process cycle graphs are not a

necessity in this project, it was thought to be a great feature for the future ease of readability of resulting outputs of simulations.

Other implementation issues were resolved by normal bug fixes or logical thought (and "re-thought"). The other major obstacles were mainly in the design phase, and not in the implementation phase. The design phase consisted of correctly running each scheduling algorithm, using the same base scheduler, and the exact same time simulator.

It is quite interesting to note the following for future discussions. There are certain times (e.g. system exceptions), when the scheduler needs to be completely by-passed. The reason being that a delay is incurred between the time that the interrupt mechanism readies a task for a process switch, and the actual time the scheduler performs the switch. This delay cannot be tolerated in certain situations, and thus the scheduler is directly by-passed, and execution control is passed to a special **Interrupt Service Routine (ISR)** [1].

5.0. Simulation Results

The simulation phase of the project was performed on four separate input files: “cpumin.dat”, “cpumed.dat”, “cpumax.dat”, and “cpumaxed.dat” Their results are stored in the corresponding files: “resultsmin.dat”, resultsmed.dat”, “resultsmax.dat” and “resultsmaxed.dat”. The different input files represent different task sets. The first one (cpumin.dat) consists of two CPU-only processes. The second one (cpumed.dat) consists of four processes: one CPU-only (p3), one CPU-bound (p2), one I/O bound (p4) and one in between (p1). The third input set (cpumax.dat) consists of five processes: two CPU-only (p3 and p4), one CPU-bound (p2), and two I/O bound (p1 and p5). Notice also that p4 is a long job while p3 is quite a short one. The last input set (cpumaxed.dat) consists of seven processes: two CPU-only (p3 and p4), two I/O bound (p6 and p7), and three in between (p1, p2 and p5).

5.1. Results and Observations

Simulations were run for all four files and the outputs were recorded in the aforementioned four resultant files. The results are tabulated below (refer to Appendix B for a complete listing of each input and its corresponding output file):

cpumin.dat	FCFS()	RR(3)	RR(6)	STCF(NO)	STCF(YES)
Throughput	0.2	0.2	0.2	0.2	0.2
Avg. Waiting Time	0	0	0	0	0
Avg. Turnaround Time	7	8.5	7	8	7.5
CPU Utilization	100	100	100	100	100

Table 5.1. Tabulated characteristics for "cpumin.dat" input

cpumed.dat	FCFS()	RR(3)	RR(6)	STCF(NO)	STCF(YES)
Throughput	0.056	0.056	0.056	0.055	0.055
Avg. Waiting Time	9.75	9.75	9.75	9.75	9.75
Avg. Turnaround Time	41.25	43	42.25	37.5	37
CPU Utilization	83.33	83.33	83.33	82.19	82.19

Table 5.2. Tabulated characteristics for "cpumed.dat" input

cpumax.dat	FCFS()	RR(3)	RR(6)	STCF(NO)	STCF(YES)
Throughput	0.060	0.067	0.065	0.065	0.067
Avg. Waiting Time	7.2	7.2	7.2	7.2	7.2
Avg. Turnaround Time	48.6	51	50.4	43	39.8
CPU Utilization	90.36	100	97.40	97.40	100

Table 5.3. Tabulated characteristics for "cpumax.dat" input

cpumaxed.dat	FCFS()	RR(3)	RR(6)	STCF(NO)	STCF(YES)
Throughput	0.051	0.052	0.050	0.050	0.050
Avg. Waiting Time	8.14	8.14	8.14	8.14	8.14
Avg. Turnaround Time	82.29	94	92.14	69.71	65.86
CPU Utilization	97.83	100	97.12	97.12	96.43

Table 5.4. Tabulated characteristics for "cpumaxed.dat" input

Averages	FCFS()	RR(3)	RR(6)	STCF(NO)	STCF(YES)
Throughput	0.092	0.094	0.093	0.093	0.093
Avg. Waiting Time	6.27	6.27	6.27	6.27	6.27
Avg. Turnaround Time	44.79	49.13	47.95	39.55	37.54
CPU Utilization	92.88	95.83	94.46	94.18	94.66

Table 5.5. Tabulated characteristics averages for all input set tests

We may conclude from the previous tabulations that the algorithm with the best throughput is RR(3), the round-robin algorithm with a time slice of 3 time units. All the algorithms had the exact same average waiting time. The contestant with the best (minimum) average turnaround time is STCF(YES), the shortest time to completion first with preemption algorithm. And finally, the policy with the maximum CPU utilization is RR(3), the round-robin algorithm with a time slice of 3 time units. The following table ranks each algorithm according to the evaluation characteristics, and assigns it a winning value (5 for best down to 1 for worst). As we can see, there is a tie between the FCFS and the RR(3) algorithm, so to the winners go the riches: their values are bolded!

Winner	**FCFS()**	**RR(3)**	RR(6)	STCF(NO)	STCF(YES)
Throughput	**4**	**5**	3	3	3
Avg. Waiting Time	**5**	**5**	5	5	5
Avg. Turnaround Time	**3**	**1**	2	4	5
CPU Utilization	**4**	**5**	3	3	2
Totals	**16**	**16**	13	15	15

Table 5.6. Winner's circle

The previous winning declaration is only made to add a bit of spice to this report. It does not, by any means, prove that the FCFS and RR(3) algorithms are better than the rest, but it does provide some feedback as to which algorithm gives better results for certain applications. For example, this designer would be inclined to use the round-robin algorithm with a small time slice the next time a scheduling algorithm decision has to be made, knowing that the number of tasks in the task set is quite short.

It is also quite interesting to observe the task order by which the processes completed for each algorithm. For example, let us take "cpumaxed.dat" as our input, and check out its corresponding output "resultsmaxed.dat" (refer to Appendix B). During the FCFS algorithm, the task order was 3-4-2-1-7-5-6, which is understandable because processes 3 and 4 do not block, and thus run to completion as soon as they have a hold of the CPU (and p3 came in before p4, thus completes first). Now, process 2 only blocks once, and is very short, and thus is expected to come out next. Between processes 1, 7 and 5, the one with the least I/O calls is 1 (thus is expected to come out first), and the one with the most I/O calls is 5 (so is expected to come out last). Finally, process 6 is the last one to complete because it blocks the most. Similar analysis were carried out for the remaining algorithms (the cycle graphs do provide a great insight in terms of why a process got blocked and for that long, for example). The following was deduced by looking at the results:

- FCFS: simple, but short jobs get stuck behind long jobs;
- RR: better for short jobs, and poor when jobs are about same length;
- STCF: optimal for both short jobs, and jobs with about the same length.

6.0. Other Scheduling Algorithms

The analysis of the remaining scheduling algorithms will be briefly be done theoretically, and will not involve any implementation or simulation details. These algorithms require quite a lot of design and implementation work, and thus are just presented here to complete the project research area, and to set up the open research topic discussion.

6.1. Multi-Level Feedback Queues (MLFQ)

The idea of using the past to predict the future is quite central in computer science: if a process was I/O bound in the past, it is most likely to also be in the future. These are called adaptive policies, because the policy is modified based on its past behaviour (as previously mentioned in this report). The result of an MLFQ-based system is that it approximates an STCF-based system: CPU-bound processes drop to the bottom of the queues, while the I/O-bound processes remain near the top. The only remaining drawback is that processes with long execution times may get starved [4].

6.2. Lottery Scheduling

In this algorithm, each process in the set receives a number of "lottery tickets", and on each time slice, a winner is randomly picked to run. The tickets are assigned by giving a big number of them to short running jobs, and a small number of them to long running jobs. To avoid starvation, every process at least gets one ticket. This type of scheduling is quite effective in terms of graceful behaviour with a dynamic load. The addition or deletion of a process effects the rest of the processes proportionally, independent of the number of tickets that a particular process has in its possession [6].

7.0. Open Research Topics

The major research topic that can greatly supplement this project deals with an innovative approach to CPU scheduling. By analyzing existing algorithms, the author has developed quite a deep understanding for the design methodologies, and implementation techniques that would provide fairness, class distinction and efficiency to the real-time application in question. This research will extend the author's knowledge, by allowing for a deeper understanding of scheduling algorithmic research. It is one thing to implement already existing algorithms, but it is a totally different matter when we speak of a newly formed scheduling policy. It is a great disappointment to the author, that the algorithm in mind did not pan out, but it is left as an open research question for future (not necessarily real-time related) design work.

Another major research question left open by this project is the design of the true SPN or SRT algorithms, in terms of future predictions and estimations. These estimations will be stored in the PCB, and compared to the actual values calculated at each CPU burst. The next prediction will be a better one because of the previous error calculations. In certain compilers, the user is asked to submit an estimate for the execution time of the process. All these methods could be used in the future design of SRN or SRT-based scheduler implementations.

Other research questions left open include the theoretical analysis (queuing theory) of the MLFQ, HRRN and lottery scheduling algorithms. It would be interesting to find out which one would have the best CPU utilization and compare it to the ones discussed in this project. Also, it would be worthy to note that extensions to this project should be done, by the simple relaxation of certain project constraints. For example, an implementation with multiple I/O blocked queues should follow, as well as, an implementation that would encompass the scheduling of both periodic as well as aperiodic processes. Finally, it would be a great challenge to research the scheduling design approaches of trying to schedule soft and hard real-time tasks in the same system, where a **soft task** is one that has an associated deadline that is desirable but not mandatory to fulfill, while a **hard task** is one that must meet its deadline, or undesirable damage will have to be sustained by the system.

8.0. Bibliography

1) Cooling, J.E. Software Design for Real-Time Systems. Chapman & Hall, London, UK: 1995.

2) Stallings, William. Operating Systems: Internals and Design Principles. Upper Saddle River, NJ: Prentice Hall, 1998.

3) Savitzky, Stephen. Real-Time Microprocessor Systems. Van Nostrand Reinhold Company, N.Y.: 1985.

4) http://www.cs.wisc.edu/~bart/537/lecturenotes/s11.html - viewed on 03/24/2000

5) Undergraduate Operating System Course Notes (Ottawa University, 1998)

6) Personal Knowledge or Previous Reading

Appendix A-
Source Code

Rami Abielmona
#1029817

// FILE: Queue.h

```
#define QUIT_SIGNAL -1
#define SUCCESS 1
#define FAILURE 0


/* Declaration of the template for the queue class */
template<class queue_element_type, int max_size=30>
class queue
{
        public:
        /* Member functions */
    /* Constructor to create an empty queue */
    queue()
            {
                num_items = 0;
             front = NULL;
             rear = NULL;
            }

    /* Destructor */
    ~queue()
            {
             cout << "Your wish is my command! Cleaning up..." << endl;

                if (num_items == 0)
             {
                cout << "Nothing to delete..." << endl;
             }
             else
             {
      current = front;
                        while (current != NULL)
                   {
                     front = front->next;
                     cout << "Deleting " << current->item->process_id << endl;
       current = NULL;
                     delete current;
                     current = front;
                        } /* end of while */
             } /* end of else */
     cout << "Done....CYA!" << endl;
            }

    /* Node insertion into queue */
    int insert(const queue_element_type *pcb)
            {
     if ( is_fifoQ_full() == SUCCESS )
     {
         cout << "fifoQ is full!" << endl;
         return FAILURE;
     }

                if (num_items == 0)
             {
                rear = new q_node;
              if (rear == NULL)
                return FAILURE;
              else
      {
       rear->next = NULL;
       front = rear;
      }
             }
             else
             {
                rear->next = new q_node;
              if (rear->next == NULL)
```

Rami Abielmona
#1029817

```
                return FAILURE;
             else
     {
                rear = rear->next;
       rear->next = NULL;
     }
           }

           rear->item = (queue_element_type*)pcb;
//    cout << "Inserted " << rear->item->process_id << endl;
           num_items++;
           return SUCCESS;
          } /* end of insert */


  /* Node removal from queue */
  int remove(queue_element_type *&x)
          {
    if ( is_fifoQ_empty() == SUCCESS)
    {
       cout << "fifoQ is empty!" << endl;
      return FAILURE;
    }

//       cout << "Deleting first element of fifoQ...." << endl;
    current = front;
    if ( num_items > 1 )
                     front = front->next;
    x = current->item;
//    cout << "Returned " << x->process_id << endl;
//    cout << "Deleting node..." << endl;
    current = NULL;
    delete current;
    num_items--;

           return SUCCESS;
          } /* end of remove */

  /* Header node retrieval without removal */
  int retrieve(queue_element_type *&x)
          {
    if ( is_fifoQ_empty() == SUCCESS)
    {
       cout << "fifoQ is empty!" << endl;
      return FAILURE;
    }

       cout << "Retrieving first element of fifoQ...." << endl;
    x = front->item;
    cout << "Returned " << x->process_id << endl;

           return SUCCESS;
          } /* end of retrieve */

          /* Print out the FIFO Q */
  int print()
  {
                if (num_items == 0)
    {
       cout << "Nothing to print in the fifoQ!" << endl;
      return FAILURE;
    }
          cout << "The data element(s) are:" << endl;

    if ( front == rear )
    {
        cout << front->item->process_id << endl;
        return SUCCESS;
    }
```

Rami Abielmona
#1029817

```
    else
    {
                    for(current = front ;
            current != NULL;
            current = current->next)
                            cout << current->item->process_id << endl;
                    return SUCCESS;
    }
            } /* end of print */

   /* Empty queue test */
   int is_fifoQ_empty()
   {
         if ( num_items != 0 )
         return FAILURE;
     else return SUCCESS;
   }

   /* Full queue test */
   int is_fifoQ_full()
   {
         if ( num_items < max_size )
         return FAILURE;
     else return SUCCESS;
   }

   /* Retrieve number of nodes in queue */
   int getnumcount()
            {
              return num_items;
            }

   private:
         /* Data members */

            /* Declaration of the structure for each queue element (node) */
            struct q_node {
          queue_element_type *item;     /* pointer to queue item */
      q_node *next;        /* pointer to next item */
     };

   q_node *front;
   q_node *rear;
   q_node *current;
   int num_items;

};
```

// FILE: Queue.cpp

```
#include <iostream.h>
#include <stddef.h>
#include "queue.h"

void main()
{
         void print_menu();

  queue<int, 5> fifoQ;
  char selection;
  int data_element,x;

         print_menu();
  cin >> selection;
  while ( selection != 'Q' && selection != 'q' )
  {
            switch(selection)
              {
                case 'P': case 'p':
```

Rami Abielmona
#1029817

```
                                        cout << "Printing fifoQ...." << endl;
          fifoQ.print();
                         break;
                       case 'I': case 'i':
          cout << "The data element is:" << endl;
          cin >> data_element;
                            cout << "Inserting at head of fifoQ..." << endl;
                         if ( fifoQ.insert(data_element) == FAILURE )
                   cout << "Error inserting node " << data_element << endl;
                            break;
                       case 'D': case 'd':
                         if ( fifoQ.remove(x) == FAILURE )
                            cout << "Error deleting node" << endl;
                         break;
                       case 'R': case 'r':
                         if ( fifoQ.retrieve(x) == FAILURE )
                            cout << "Error retrieving node" << endl;
                         break;
        case 'O': case 'o':
          cout << "The number of items in the fifoQ is:" << endl;
          cout << fifoQ.getnumcount() << endl;
          break;
        case 'F': case 'f':
          if ( fifoQ.is_fifoQ_full() == 1 )
             cout << "Yes." << endl;
          else cout << "No." << endl;
          break;
        case 'E': case 'e':
          if ( fifoQ.is_fifoQ_empty() == 1 )
             cout << "Yes." << endl;
          else cout << "No." << endl;
          break;
                       default:
                            cout << "Unknown selection! Try again..." << endl;
                     } /* end of switch */
    print_menu();
    cin >> selection;
         } /* end of while */
  /* destructor execution here */
         return;
}


void print_menu()
{
         cout << endl;
         cout << "Select one of these options:" << endl;
  cout << "P) Print the fifoQ" << endl;
  cout << "I) Insert an element into the fifoQ" << endl;
  cout << "D) Delete the first element of the fifoQ" << endl;
  cout << "R) Retrieve the first element of the fifoQ without deleting it" << endl;
  cout << "O) Get number of items in the fifoQ" << endl;
  cout << "F) Is the fifoQ full?" << endl;
  cout << "E) Is the fifoQ empty?" << endl;
  cout << "Q) Quit" << endl;
  return;
}
```

// FILE: Cpu.h

```
#define MAXBLOCKS   10
#define MAXPOWER    5
#define MAXPROCESS  10
#define YES                    1
#define NO                     0

enum state { Running, Ready, Waiting };
```

```
struct PCB {
                    int         process_id;
                    int         priority;
    char  process_cycle[255];
    int   timeleft;
    int   waiting_time;
    int   cpu_time;
    int   ready_time;
    int   num_of_block;
    int   cur_block;
    int   blocking_times[MAXBLOCKS];
    int   blocking_lengths[MAXBLOCKS];
        state status;
  };

void fill_poolQ();
void FCFS();
void RR(int timeslice);
void STCF(int preemption);
```

// FILE: Cpu.cpp

```
#include <fstream.h>    /* cout, file output operations */
#include <stdio.h>      /* file input operations */
#include <stdlib.h>     /* atoi() */
#include <ctype.h>      /* isdigit() */
#include <string.h>     /* strcat(), strlen() */
#include <dos.h>        /* sleep() */
#include <signal.h>     /* raise(), signal() */
#include "queue.h"      /* template queue class definition */
#include "cpu.h"        /* PCB struct */

/* Typedefs */
typedef void (*fptr)(int);
typedef queue<PCB, MAXPROCESS> fifoQ;

/* Global variables */
fifoQ poolQ, readyQ, runningQ, blockedQ;
int total_cpu_time = 0, total_waiting_time = 0, total_ready_time = 0;
int total_ticks = 0;

/* Function prototypes */
void schedule(char, int);
void run();
void suspend();
void resume(PCB *);
void print_process_info(PCB *);
void update_graphs();
void update_priorities();
void print_results();
void fprint_cycle(PCB *);
void get_tasks(fifoQ&, fifoQ&);
void pri_get_tasks(fifoQ&, fifoQ&, int);
void priority_sort(fifoQ&);
int get_input(char buffer[MAXPOWER]);

void main()
{
        /* Fill the poolQ with the task set */
        fill_poolQ();

  /* Clear out results.dat */
  ofstream fout("results.dat");
  if (!fout)
        cout << "Cannot open results.dat for truncation!" << endl;
  fout.close();

  cout << "Testing FCFS..." << endl;
```

```
   /* Open results.dat in append mode */
   ofstream foutapp("results.dat",ios::app);
   if (!foutapp)
           cout << "Cannot open results.dat for appending!" << endl;
   foutapp << "Testing FCFS..." << endl << endl;
   foutapp.close();

   FCFS();

   cout << "Testing RR(3)..." << endl;
   /* Open results.dat in append mode */
   foutapp.open("results.dat",ios::app);
   foutapp << endl << "Testing RR(3)..." << endl << endl;
   foutapp.close();

   RR(3);

   cout << "Testing RR(6)..." << endl;
   /* Open results.dat in append mode */
           foutapp.open("results.dat",ios::app);
   foutapp << endl << "Testing RR(6)..." << endl << endl;
   foutapp.close();

   RR(6);

   cout << "Testing STCF() without preemption..." << endl;
   /* Open results.dat in append mode */
           foutapp.open("results.dat",ios::app);
   foutapp << endl << "Testing STCF() without preemption..." << endl << endl;
   foutapp.close();

   STCF(NO);

   cout << "Testing STCF() with preemption..." << endl;
   /* Open results.dat in append mode */
           foutapp.open("results.dat",ios::app);
   foutapp << endl << "Testing STCF() with preemption..." << endl << endl;
   foutapp.close();

   STCF(YES);

   cout << "DONE ALL ALGORITHMS!" << endl;
   /* Open results.dat in append mode */
           foutapp.open("results.dat",ios::app);
   foutapp << endl << "DONE ALL ALGORITHMS!" << endl;
   foutapp.close();

   cout << "Cleaning up blockedQ, runningQ, readyQ and poolQ" << endl;
}

void time_simulator()
{
   PCB *cur_process, *blocked_header;
   int i,blocked_count;

   /* reinstall signal handler */
   signal(SIGUSR1, (fptr)time_simulator);

   /* Update running process' timing information */
   /* Decrement computation time as well as */
   /* any blocking times */
           if (runningQ.is_fifoQ_empty() == FAILURE)
   {
             runningQ.retrieve(cur_process);
     if (cur_process->timeleft > 0)
                     cur_process->timeleft--;
     for(i=0; i < cur_process->num_of_block; i++)
     {
           if (cur_process->blocking_times[i] > 0)
```

```
                    cur_process->blocking_times[i]--;
      } /* end of for loop (updated blocking times) */
            cout << "Running " << cur_process->process_id << endl;
            print_process_info(cur_process);
   } /* end of if statement (updated running process info) */



   /* Update timing infomation on all blocked processes */
   /* Basically, decrement the lengths field */
          if (blockedQ.is_fifoQ_empty() == FAILURE)
   {
      blocked_count = blockedQ.getnumcount();
      for(i=0; i < blocked_count; i++)
      {
         blockedQ.remove(blocked_header);
         if (blocked_header->blocking_lengths[blocked_header->cur_block] > 0)
                              blocked_header->blocking_lengths[blocked_header->cur_block]--;
         blockedQ.insert(blocked_header);
      } /* end of for loop (updated blocking lengths) */
   }

   cout << "In readyQ" << endl;
   readyQ.print();
   cout << "In runningQ" << endl;
   runningQ.print();
   cout << "In blockedQ" << endl;
   blockedQ.print();

   /* Update the process cycle graphs for all processes in the system */
   update_graphs();

   /* Update the process' priorities */
   update_priorities();

   /* Increment total_ticks because a time tick occured */
   total_ticks++;
   cout << "TIME TICK!" << endl;
}

void STCF(int preemption)
{
          cout << "In STCF()..." << endl;

   /* Drain poolQ and re-queue input task set into it */
   fill_poolQ();

   /* Retrieve the tasks from the pool Q into the ready Q */
   get_tasks(poolQ,readyQ);

   /* While at least one queue is not empty, schedule the task set */
          while ( (readyQ.is_fifoQ_empty() == FAILURE) ||
                       (blockedQ.is_fifoQ_empty() == FAILURE) ||
        (runningQ.is_fifoQ_empty() == FAILURE) )
   {
          /* Call the scheduler for STCF scheduling */
      if (preemption == YES)
                    schedule('S',0);
      else if (preemption == NO)
                    schedule('N',0);

      /* cast to appropriate type */
            signal(SIGUSR1, (fptr)time_simulator);
      /* Call the time_simulator which will update PCB timings */
            raise(SIGUSR1);
      /* Simulate a time tick (currently 2 sec) */
            sleep(2);
          } /* end of while loop */
```

Rami Abielmona
#1029817

```
   /* Output statistics of algorithm to screen */
   print_results();

            cout << "Leaving STCF" << endl;
} /* end of STCF() */

void FCFS()
{
            cout << "In FCFS()..." << endl;

   /* Retrieve the tasks from the pool Q into the ready Q */
   get_tasks(poolQ, readyQ);

   /* While at least one queue is not empty, schedule the task set */
            while ( (readyQ.is_fifoQ_empty() == FAILURE) ||
                        (blockedQ.is_fifoQ_empty() == FAILURE) ||
         (runningQ.is_fifoQ_empty() == FAILURE) )
   {
            /* Call the scheduler for FCFS scheduling (0 timeslice) */
      schedule('F',0);

               /* cast to appropriate type */
               signal(SIGUSR1, (fptr)time_simulator);
      /* Call the time_simulator which will update PCB timings */
               raise(SIGUSR1);
      /* Simulate a time tick (currently 2 sec) */
               sleep(2);
            } /* end of while loop */

   /* Output statistics of algorithm to screen */
   print_results();

            cout << "Leaving FCFS()..." << endl;
} /* end of FCFS() */

void RR(int timeslice)
{
            cout << "In RR(int)..." << "with a time slice of " << timeslice << endl;

   /* Drain poolQ and re-queue input task set into it */
   fill_poolQ();

   /* Retrieve the tasks from the pool Q into the ready Q */
   get_tasks(poolQ, readyQ);

   /* While at least one queue is not empty, schedule the task set */
            while ( (readyQ.is_fifoQ_empty() == FAILURE) ||
                        (blockedQ.is_fifoQ_empty() == FAILURE) ||
         (runningQ.is_fifoQ_empty() == FAILURE) )
   {
            /* Call the scheduler for RR scheduling (with timeslice) */
      schedule('R',timeslice);

      /* cast to appropriate type */
               signal(SIGUSR1, (fptr)time_simulator);
      /* Call the time_simulator which will update PCB timings */
               raise(SIGUSR1);
      /* Simulate a time tick (currently 2 sec) */
               sleep(2);
            } /* end of while loop */

   /* Output statistics of algorithm to screen */
   print_results();

            cout << "Leaving RR(" << timeslice << ")..." << endl;
} /* end of RR(int) */

void schedule(char algorithm, int timeslice)
{
```

```
        PCB *cur_process, *blocked_header, *ready_process;
  int i, blocked_count, string_length;

        cout << "SCHEDULING................................" << endl;


  if (runningQ.is_fifoQ_empty() == SUCCESS)
  {
    total_ticks = 0;
    /* If using STCF, then sort readyQ before pick the highest */
    /* priority process in the ready Q to run */
    if ( (algorithm == 'S') || (algorithm == 'N') )
        priority_sort(readyQ);
                run();      /* If no process running, run the next one */
  }
  else
  {
          runningQ.retrieve(cur_process);
          if (cur_process->timeleft == 0)
          {
            runningQ.remove(cur_process);  /* Process is done executing */
            cout << "Removing " << cur_process->process_id << " from the system" << endl;

                        /* Print out process cycle graph */
      string_length = strlen(cur_process->process_cycle);
      cout << "The string length is: " << string_length << endl;
      cout << cur_process->process_cycle << endl;

      /* Print out to file process cycle graph */
      fprint_cycle(cur_process);

                        /* Update the timing global variables with this process' values */
      total_cpu_time += cur_process->cpu_time;
      total_waiting_time += cur_process->waiting_time;
      total_ready_time += cur_process->ready_time;

      /* Run the next available process in the ready Q */
      /* and reset total_ticks global var for RR */
      total_ticks = 0;
            /* If using STCF, then sort readyQ before pick the highest */
          /* priority process in the ready Q to run */
            if ( (algorithm == 'S') || (algorithm == 'N') )
                priority_sort(readyQ);
                run();
          }
          else   /* Process is not done executing */
          {
                  if ( (cur_process->num_of_block > 0) &&
                          (cur_process->blocking_times[cur_process->cur_block] == 0) &&
              (cur_process->num_of_block != cur_process->cur_block) )
                  {
                                /* Block the process and run the next available process from */
        /* the ready Q */
                        suspend();
        total_ticks = 0;
                  /* If using STCF, then sort readyQ before pick the highest */
                /* priority process in the ready Q to run */
                    if ( (algorithm == 'S') || (algorithm == 'N') )
                        priority_sort(readyQ);
                        run();
                  }  /* end of if statement (process block check) */
                  else if ( (algorithm == 'R') &&
                              (total_ticks % timeslice == 0) &&
                          (total_ticks > 0) )
                  {
                    cout << "PREEMPT!!! TIME SLICE UP!!" << endl;
        total_ticks = 0;        /* Reset global var */
                                run();      /* Time slice is up, so process switch! */
                  } /* end of else if (RR time slice check) */
```

```
                        else if (algorithm == 'S')
       {
           /* Only STCF with preemption gets to preempt a runnning process */
         /* if its priority is lower than the highest priority ready process */
                        priority_sort(readyQ);
                                            readyQ.retrieve(ready_process);
//          cout << "Ready process " << ready_process->process_id << endl;
//          cout << "has PRIORITY " << ready_process->priority << endl;
//          cout << "Running process " << cur_process->process_id << endl;
//          cout << "has PRIORITY " << cur_process->priority << endl;
         if (ready_process->priority < cur_process->priority)
           run();
       }  /* end of else if statement (Check for STCF) */
              }  /* end of if statement (process done check) */
   }  /* end of if statement (empty running Q check) */

   if (blockedQ.is_fifoQ_empty() == FAILURE)
   {
      blocked_count = blockedQ.getnumcount();
      /* Check all blocked processes to see if they completed their I/O */
      for(i=0; i < blocked_count; i++)
      {
         blockedQ.remove(blocked_header);
         if ( (blocked_header->blocking_times[blocked_header->cur_block] == 0) &&
                                             (blocked_header->blocking_lengths[blocked_header->cur_block] == 0) )
                                  /* process completed I/O, so resume its execution */
                                                          resume(blocked_header);
         else
                        blockedQ.insert(blocked_header);
      }  /* end of for loop (went through blocked Q) */
   }  /* end of if statement (resumption checks) */

   cout << "Leaving SCHEDULER...." << endl;

} /* end of scheduler */

void run()
{
          PCB *to_be_run_process, *preempted_process;

   /* Preempt running process and make it ready */
   if (runningQ.is_fifoQ_empty() == FAILURE)
   {
          runningQ.remove(preempted_process);
             cout << "Removed " << preempted_process->process_id << " from the runningQ" << endl;
                        print_process_info(preempted_process);
      readyQ.insert(preempted_process);
      cout << " on the readyQ" << endl;
      preempted_process->status = Ready;
   }

   /* Make a ready process running */
   if (readyQ.is_fifoQ_empty() == FAILURE)
   {
                        readyQ.remove(to_be_run_process);
             cout << "Removed " << to_be_run_process->process_id << " from the readyQ" << endl;
                        print_process_info(to_be_run_process);
             runningQ.insert(to_be_run_process);
      cout << " on the runningQ" << endl;
      to_be_run_process->status = Running;
   }
}  /* end of run() */

void suspend()
{
          PCB *cur_process;

   /* Make a running process wait for I/O event (block it) */
          runningQ.remove(cur_process);
```

```
   blockedQ.insert(cur_process);
   cur_process->status = Waiting;
}  /* end of suspend() */

void resume(PCB *blocked_process)
{
   /* Resume a blocked process (I/O complete) by making it ready */
   /* Unless the runningQ is empty, then just run the process */
   if (runningQ.is_fifoQ_empty() == SUCCESS)
   {
                    runningQ.insert(blocked_process);
      blocked_process->status = Running;
   }
   else
   {
                    readyQ.insert(blocked_process);
                      blocked_process->status = Ready;
   }
   blocked_process->cur_block++;
}  /* end of resume(PCB*) */

void fill_poolQ()
{
   FILE *fp;
   char buffer[MAXPOWER];
   int input, queue_count, i;
          PCB *pcb;

   /* Open cpu.dat for reading */
   fp = fopen("cpu.dat", "r");
   if ( fp == NULL )
   {
          cout << "Cannot find cpu.dat!" << endl;
     return;
   }

   /* If a poolQ already exists (thus, not empty), then */
   /* empty it */
   if (poolQ.is_fifoQ_empty() == FAILURE)
   {
          queue_count = poolQ.getnumcount();
     for(i=0; i < queue_count; i++)
          poolQ.remove(pcb);
   }

   /* While !NULL, set up each PCB read from cpu.dat */
   while (fgets(buffer, sizeof(buffer), fp))
   {
     /* Convert to integer */
     input = buffer[1] - 48;

          switch (buffer[0])
     {
          case 'p':
                    /* Set up a new PCB and initialize its fields */
                    pcb = new PCB;
           cout << buffer[0] << endl;
           cout << buffer[1] << endl;
           pcb->process_id = input;
           pcb->status = Ready;   /* Processes enter as Ready */
           pcb->num_of_block = 0;
           pcb->cur_block = 0;
           pcb->waiting_time = 0;
           pcb->cpu_time = 0;
           pcb->ready_time = 0;
           for(i=0; i < 255; i++)
                    pcb->process_cycle[i] = '\0';
           break;
           /* What is the process' priority? */
```

```
    case 'a':
                pcb->priority = get_input(buffer);
        case 't':
                /* How long does the process execute for? */
          pcb->timeleft = get_input(buffer);
          break;
        case 'b':
                /* When does the process block? */
          pcb->blocking_times[pcb->num_of_block] = get_input(buffer);
          break;
    case 'c':
                /* For each block, how long does the process block for? */
               pcb->blocking_lengths[pcb->num_of_block] = get_input(buffer);
          pcb->num_of_block ++;
          break;
    case 'e':
                /* Insert process into poolQ */
                        cout << "Attempting to insert " << pcb->process_id << endl;
                poolQ.insert(pcb);
          break;
    }  /* end of switch statement (on buffer[0]) */
  }  /* end of while statement */
  fclose(fp);
}  /* end of fill_poolQ() */

void get_tasks(fifoQ& sourceQ, fifoQ& destinationQ)
{
        PCB *pcb;
  int count,i;

  /* Copy processes from sourceQ into the destinationQ */
  count = sourceQ.getnumcount();
  for(i=count ; i>0; i--)
  {
        sourceQ.remove(pcb);
    destinationQ.insert(pcb);
    sourceQ.insert(pcb);
  }
  cout << "Finished getting tasks from poolQ" << endl;
}  /* end of get_tasks(fifoQ&, fifoQ&) */

void pri_get_tasks(fifoQ& sourceQ, fifoQ& destinationQ, int Qpriority)
{
        PCB *pcb;
  int count,i;

  /* Copy processes from sourceQ into the destinationQ */
  /* only if they have Qpriority as well */
  count = sourceQ.getnumcount();
  for(i=count ; i>0; i--)
  {
        sourceQ.remove(pcb);
    if (pcb->priority == Qpriority)
            destinationQ.insert(pcb);
    sourceQ.insert(pcb);
  }
  cout << "Finished getting tasks from poolQ" << endl;
}  /* end of pri_get_tasks(fifoQ&, fifoQ&, int) */

void priority_sort(fifoQ& queue)
{
        PCB* pcb[MAXPROCESS];
  PCB *temp_process;
  int num_processes, i, j;

  /* Dequeue all PCBs on the queue */
  num_processes = queue.getnumcount();
        for(i=0; i < num_processes; i++)
        queue.remove(pcb[i]);
```

```
   /* Swap PCB's in the array if they are misplaced */
   for(i=0; i < num_processes; i++)
   {
            for(j=i+1; j < num_processes; j++)
      {
            if(pcb[i]->priority > pcb[j]->priority)
         {
            temp_process = pcb[i];
           pcb[i] = pcb[j];
           pcb[j] = temp_process;
         }  /* end of if statement */
      }  /* end of j for loop */
   }  /* end of i for loop */

   /* Enqueue all PCBs back on the queue sorted by priority */
   for(i=0; i < num_processes; i++)
            queue.insert(pcb[i]);
}  /* end of priority_sort(fifoQ&) */

int get_input(char buffer[MAXPOWER])
{
   const int base = 10;
   const char newline = '\n';
            int integer_value = 0, i = 0, digit;


   /* Get the integer value of the passed-in buffer */
   while (buffer[i] != newline)
   {
            if ( isdigit(buffer[i]) )
      {
            digit = int (buffer[i]) - int ('0');
         integer_value = base * integer_value + digit;
      }
      i++;
   }  /* end of while loop */

   return integer_value;
}  /* end of get_input(char[]) */


void print_process_info(PCB *process)
{
                          int blocks = process->num_of_block, i;

      cout << "Its computation time left is " << process->timeleft << endl;

      if (blocks == 0)
            cout << "Does not block" << endl;
      else
      {
                 for (i=0; i < blocks; i++)
                 {
                                         cout << "It blocks at " << process->blocking_times[i] << " for " <<
                                         process->blocking_lengths[i] << " time units" << endl;
                                       }  /* end of for loop (goes through number of blocks) */
      }  /* end of if statement */
}  /* end of print_process_info(PCB*) */


void update_graphs()
{
            PCB *running_process = NULL, *blocked_process = NULL, *ready_process = NULL;
   const char *running = "-", *blocked = "|", *ready = "/";
   int i, queue_count;

   /* Used to update the stats (cpu, waiting and ready) times */
   /* for all processes in the system */
```

Rami Abielmona
#1029817

```
        if (runningQ.is_fifoQ_empty() == FAILURE)
  {
        runningQ.remove(running_process);
//      cout << "Before concatRu: " << running_process->process_cycle << endl;
              strcat(running_process->process_cycle, running);
//      cout << "After concatRu: " << running_process->process_cycle << endl;
              running_process->cpu_time++;
    runningQ.insert(running_process);
  }  /* end of if statement */

        if (blockedQ.is_fifoQ_empty() == FAILURE)
  {
    queue_count = blockedQ.getnumcount();
    for(i=0; i < queue_count; i++)
    {
      blockedQ.remove(blocked_process);
//        cout << "Before concatW: " << blocked_process->process_cycle << endl;
      strcat(blocked_process->process_cycle, blocked);
//        cout << "After concatW: " << blocked_process->process_cycle << endl;
                    blocked_process->waiting_time++;
      blockedQ.insert(blocked_process);
    }  /* end of for loop */
  }  /* end of if statement */

        if (readyQ.is_fifoQ_empty() == FAILURE)
  {
    queue_count = readyQ.getnumcount();
    for(i=0; i < queue_count; i++)
    {
      readyQ.remove(ready_process);
//        cout << "Before concatRe: " << ready_process->process_cycle << endl;
              strcat(ready_process->process_cycle, ready);
//        cout << "After concatRe: " << ready_process->process_cycle << endl;
                    ready_process->ready_time++;
      readyQ.insert(ready_process);
    }  /* end of for loop */
  }  /* end of if statement */


}  /* End of update_graphs() */

void update_priorities()
{
        PCB *process = NULL;
  int i, j, queue_count, temp_time;
  int time_buf[MAXPROCESS];

  /* Initialize time buffer to all -1 */
  for(i=0; i < MAXPROCESS; i++)
        time_buf[i] = -1;

  if (poolQ.is_fifoQ_empty() == FAILURE)
  {
    queue_count = poolQ.getnumcount();
    for(i=0; i < queue_count; i++)
    {
      /* Fill up time buffer with actual timeleft values */
      /* of all processes in the system */
              poolQ.remove(process);
        time_buf[i] = process->timeleft;
          poolQ.insert(process);
    }  /* end of for loop */
  }  /* end of if statemnt */

  /* Sort time buffer according to increasing timeleft values */
  for(i=0; i < queue_count; i++)
  {
        for(j=i+1; j < queue_count; j++)
    {
```

```
            if (time_buf[i] > time_buf[j])
         {
            temp_time = time_buf[i];
           time_buf[i] = time_buf[j];
           time_buf[j] = temp_time;
         }  /* end of if statement (swapping condition) */
      }  /* end of j for loop */
   }  /* end of i for loop */

   /* Update the priority field of each process according to */
   /* the sorted time buffer */
   for(i=0; i < queue_count; i++)
   {
            poolQ.remove(process);
      for(j=0; j < queue_count; j++)
      {
                  if (process->timeleft == time_buf[j])
         {
//            cout << "For process " << process->process_id << endl;
//            cout << "Old priority is: " << process->priority << endl;
//            cout << "New priority is: " << j+1 << endl;
                                          process->priority = j+1;
         }  /* end of if statement */
      }  /* end of j for loop (goes through time buffer) */
      poolQ.insert(process);
   }  /* end of i for loop (goes through processes) */


}  /* End of update_priorities() */

void print_results()
{
            int num_of_processes, total_turn_time, total_time, starting_time, i;
   float avg_wait_time, avg_turn_time, throughput, cpu_util;
   PCB *removed_process;

   num_of_processes = poolQ.getnumcount();

   poolQ.remove(removed_process);
   starting_time = strlen(removed_process->process_cycle);
   poolQ.insert(removed_process);

   /* Find out longest process cycle graph */
   for(i=0; i < num_of_processes; i++)
   {
                        poolQ.remove(removed_process);
                        if (starting_time < strlen(removed_process->process_cycle) )
                                          starting_time = strlen(removed_process->process_cycle);
      poolQ.insert(removed_process);
   }

   cout << "Total cpu time is:\t\t" << total_cpu_time << endl;
   cout << "Total wait time is:\t\t" << total_waiting_time << endl;
   cout << "Total ready time is:\t\t" << total_ready_time << endl;
   cout << "Number of processes:\t\t" << num_of_processes << endl;
   cout << "Total time is:\t\t\t" << starting_time << endl;

   /* Statistical calculations */
   total_time = starting_time;
   throughput = (float)num_of_processes / total_time;
   avg_wait_time = (float)total_waiting_time / num_of_processes;
   total_turn_time = total_waiting_time + total_cpu_time + total_ready_time;
   avg_turn_time = (float)total_turn_time / num_of_processes;
   cpu_util = (float)total_cpu_time / total_time;
   cpu_util *= 100;

   cout << "Throughput Is:\t\t\t" << throughput << endl;
   cout << "Average Waiting Time Is:\t" << avg_wait_time << endl;
   cout << "Average Turnaround Time Is:\t" << avg_turn_time << endl;
```

Rami Abielmona
#1029817

```
   cout << "CPU Utilization Is:\t\t" << cpu_util << " percent" << endl;

   ofstream fout("results.dat",ios::app);
   if (!fout)
           cout << "Cannot open results.dat for writing!" << endl;

   fout << endl;
   fout << "Total cpu time is:\t\t" << total_cpu_time << endl;
   fout << "Total wait time is:\t\t" << total_waiting_time << endl;
   fout << "Total ready time is:\t\t" << total_ready_time << endl;
   fout << "Number of processes:\t\t" << num_of_processes << endl;
   fout << "Total time is:\t\t\t" << total_time << endl;
   fout << "Throughput Is:\t\t\t" << throughput << endl;
   fout << "Average Waiting Time Is:\t" << avg_wait_time << endl;
   fout << "Average Turnaround Time Is:\t" << avg_turn_time << endl;
   fout << "CPU Utilization Is:\t\t" << cpu_util << " percent" << endl;

   fout.close();

   /* Reset global variables to initial values */
   total_waiting_time = total_cpu_time = total_ready_time = total_ticks = 0;
}  /* End of print_results() */

void fprint_cycle(PCB *process)
{
           /* Open results.dat for appending */
   ofstream fout("results.dat",ios::app);
   if (!fout)
           cout << "Cannot open results.dat for writing!" << endl;


           /* Print out to file process cycle graph */
   fout << "Process cycle graph for process " << process->process_id << endl;
   fout << process->process_cycle << endl;

   fout.close();
}  /* End of fprint_cycle(PCB*) */
```

Appendix B-
Task Set Results

Rami Abielmona
#1029817

cpumax.dat:

```
p1
a2
t20
b5
c8
b10
c3
b18
c6
e
p2
a1
t5
b2
c4
e
p3
a1
t5
e
p4
a3
t25
e
p5
a2
t20
b2
c4
b8
c2
b11
c4
b16
c5
e
```

resultsmax.dat:

```
Testing FCFS...

Process cycle graph for process 3
///////-----
Process cycle graph for process 4
////////////-------------------------
Process cycle graph for process 2
/////--||||////////////////////////////---
Process cycle graph for process 1
-----||||||||/////////////////////////////-----|||///--------||||||--
Process cycle graph for process 5
/////////////////////////////////////--||||////------||//////---||||/-----|||||----

Total cpu time is:              75
Total wait time is:             36
Total ready time is:            132
Number of processes:            5
Total time is:                          83
Throughput Is:                          0.060241
Average Waiting Time Is:        7.2
Average Turnaround Time Is:     48.6
CPU Utilization Is:             90.3614 percent

Testing RR(3)...

Process cycle graph for process 3
/////---///////--
Process cycle graph for process 2
///--||||////////---
```

Process cycle graph for process 4
////////---/////////---///---//////---/////---///---//////---////----
Process cycle graph for process 1
---//////////--|||||||//////---//////--|||///---//////---/////--|||||/--
Process cycle graph for process 5
///////////--||||//////---//////---||///---||||/////---//////--|||||/---//-

Total cpu time is: 75
Total wait time is: 36
Total ready time is: 144
Number of processes: 5
Total time is: 75
Throughput Is: 0.0666667
Average Waiting Time Is: 7.2
Average Turnaround Time Is: 51
CPU Utilization Is: 100 percent

Testing RR(6)...

Process cycle graph for process 3
///////-----
Process cycle graph for process 2
/////--||||/////////---
Process cycle graph for process 4
////////////------//////////------////////////------/////------/////-
Process cycle graph for process 1
-----||||||||//////////-----|||/////////------/////////--|||||//////--
Process cycle graph for process 5
/////////////////--||||//////////------||//////////---||||////-----|||||----

Total cpu time is: 75
Total wait time is: 36
Total ready time is: 141
Number of processes: 5
Total time is: 77
Throughput Is: 0.0649351
Average Waiting Time Is: 7.2
Average Turnaround Time Is: 50.4
CPU Utilization Is: 97.4026 percent

Testing STCF() without preemption...

Process cycle graph for process 3
//-----
Process cycle graph for process 2
--||||/---
Process cycle graph for process 4
//////////////////////-------------------------
Process cycle graph for process 5
//////////--||||/------||/////////////////////---||||/-----|||||///----
Process cycle graph for process 1
////////////-----||||||||/////////////////////////-----|||//--------|||||--

Total cpu time is: 75
Total wait time is: 36
Total ready time is: 104
Number of processes: 5
Total time is: 77
Throughput Is: 0.0649351
Average Waiting Time Is: 7.2
Average Turnaround Time Is: 43
CPU Utilization Is: 97.4026 percent

Testing STCF() with preemption...

Process cycle graph for process 3
//-----
Process cycle graph for process 2
--||||/---
Process cycle graph for process 5

//////////--||||/------||/---|||/-----||||///----
Process cycle graph for process 1
////////////-----|||||||////-----|||//--------||||||/--
Process cycle graph for process 4
//////////////////////---/////////////////////////---//-------------------

Total cpu time is:	75
Total wait time is:	36
Total ready time is:	88
Number of processes:	5
Total time is:	75
Throughput Is:	0.0666667
Average Waiting Time Is:	7.2
Average Turnaround Time Is:	39.8
CPU Utilization Is:	100 percent

DONE ALL ALGORITHMS!

cpumaxed.dat:

p1
a2
t20
b5
c8
b18
c6
e
p2
a1
t5
b2
c4
e
p3
a1
t5
e
p4
a3
t25
e
p5
a2
t20
b2
c4
b8
c2
b11
c4
b16
c5
e
p6
a4
t28
b5
c3
b10
c3
b15
c3
b20
c3
b25
c3
e
p7
a5

Rami Abielmona
#1029817

t32
b2
c5
b27
c4
e

resultsmaxed.dat:

Testing FCFS...

Process cycle graph for process 3
///////-----
Process cycle graph for process 4
///////////------------------------
Process cycle graph for process 2
/////--||||///////////////////////////////////---
Process cycle graph for process 1
-----|||||||||//////////////////////////////////////-------------||||||/////////////////////////////////--
Process cycle graph for process 7
//--|||||///////////////////////-------------------------||||//////-----
Process cycle graph for process 5
///////////////////////////////////////--||||///////////////////------||////////////////////////////////---||||//////-----|||||----
Process cycle graph for process 6
///////////////////////////////////////-----|||//////////////////////-----|||///////////////////////////////-----|||///////-----|||/-----|||---

Total cpu time is: 135
Total wait time is: 57
Total ready time is: 384
Number of processes: 7
Total time is: 138
Throughput Is: 0.0507246
Average Waiting Time Is: 8.14286
Average Turnaround Time Is: 82.2857
CPU Utilization Is: 97.8261 percent

Testing RR(3)...

Process cycle graph for process 3
/////---///////////--
Process cycle graph for process 2
///--||||/////////////---
Process cycle graph for process 5
//////////--||||/////////////---/////////---||//////////---||||///////---////////////--|||||///////---/////////-
Process cycle graph for process 1
---///////////////--|||||||||///////////---////////////---//////////---////////////---//////////-||||||//////////--
Process cycle graph for process 4
////////---//////////////---////////---////////////---//////////---////////////---//////////---///////---/////////-
Process cycle graph for process 7
////////////////--|||||//////////---////////////---//////////---/////////---//////////---//////---/////////---//////----||||/-----
Process cycle graph for process 6
////////////---///////////--|||////////////---///////////--|||////////////---//////////--|||//////////---///////--|||/-----|||//---

Total cpu time is: 135
Total wait time is: 57
Total ready time is: 466
Number of processes: 7
Total time is: 135
Throughput Is: 0.0518519
Average Waiting Time Is: 8.14286
Average Turnaround Time Is: 94
CPU Utilization Is: 100 percent

Testing RR(6)...

Process cycle graph for process 3
///////-----
Process cycle graph for process 2
/////--||||/////////////////---
Process cycle graph for process 4

Rami Abielmona
#1029817

////////////------////////////////////------//////////////////////////------/////////////////------//////////////////-

Process cycle graph for process 1

-----|||||||///////////////////------//////////////////////////------//////////////////////-||||||///////////////////--

Process cycle graph for process 5

////////////////////--||||////////////////////------||////////////////////////---||||////////////////-----|||||//////////----

Process cycle graph for process 6

//////////////////////-----|||//////////////////////-----|||////////////////////-----|||/////////////////-----|||///////////-----|||///---

Process cycle graph for process 7

////////////////////////////--|||||////////////////////////------//////////////////////------///////////////////------/////////////------///-||||-----

Total cpu time is:	135
Total wait time is:	57
Total ready time is:	453
Number of processes:	7
Total time is:	139
Throughput Is:	0.0503597
Average Waiting Time Is:	8.14286
Average Turnaround Time Is:	92.1429
CPU Utilization Is:	97.1223 percent

Testing STCF() without preemption...

Process cycle graph for process 3

//-----

Process cycle graph for process 2

--||||/---

Process cycle graph for process 4

/////////////////////////-------------------------

Process cycle graph for process 1

////////////-----|||||||/////////////////////////////-------------||||||////--

Process cycle graph for process 5

//////////--||||/------||/////////////////////////---||||/////////-----|||||//----

Process cycle graph for process 7

//--|||||/--------------------------||||/-----

Process cycle graph for process 6

//-----|||///-----|||-----|||////////////////////////-----|||//-----|||---

Total cpu time is:	135
Total wait time is:	57
Total ready time is:	296
Number of processes:	7
Total time is:	139
Throughput Is:	0.0503597
Average Waiting Time Is:	8.14286
Average Turnaround Time Is:	69.7143
CPU Utilization Is:	97.1223 percent

Testing STCF() with preemption...

Process cycle graph for process 3

//-----

Process cycle graph for process 2

--||||/---

Process cycle graph for process 5

//////////--||||/------||/---||||/-----|||||///----

Process cycle graph for process 1

////////////-----|||||||////-----/////--------||||||/--

Process cycle graph for process 4

/////////////////////////---///////////////////////////---//-------------------

Process cycle graph for process 6

//-----|||-----|||/-----|||/-----|||/-----|||/---

Process cycle graph for process 7

//--|||||/----/////----/////----/////----///---------||||-----

Total cpu time is:	135
Total wait time is:	57
Total ready time is:	269
Number of processes:	7
Total time is:	140
Throughput Is:	0.05

Rami Abielmona
#1029817

Average Waiting Time Is: 8.14286
Average Turnaround Time Is: 65.8571
CPU Utilization Is: 96.4286 percent

DONE ALL ALGORITHMS!

cpumed.dat:

p1
a3
t20
b5
c8
b18
c6
e
p2
a1
t5
b2
c4
e
p3
a2
t10
e
p4
a4
t25
b7
c7
b14
c7
b21
c7
e

resultsmed.dat:

Testing FCFS...

Process cycle graph for process 3
///////----------
Process cycle graph for process 2
/////--||||/////////////---
Process cycle graph for process 1
-----|||||||//////////////-------------||||||/--
Process cycle graph for process 4
/////////////////-------|||||||/////////-------|||||||-------|||||||----

Total cpu time is: 60
Total wait time is: 39
Total ready time is: 66
Number of processes: 4
Total time is: 72
Throughput Is: 0.0555556
Average Waiting Time Is: 9.75
Average Turnaround Time Is: 41.25
CPU Utilization Is: 83.3333 percent

Testing RR(3)...

Process cycle graph for process 2
///--||||///////---
Process cycle graph for process 3
/////---/////---//////---////-
Process cycle graph for process 1
---////////--|||||||////---//---------///-||||||--
Process cycle graph for process 4
////////---////////---//////-|||||||///---/----||||||-------||||||----

Rami Abielmona
#1029817

Total cpu time is: 60
Total wait time is: 39
Total ready time is: 73
Number of processes: 4
Total time is: 72
Throughput Is: 0.0555556
Average Waiting Time Is: 9.75
Average Turnaround Time Is: 43
CPU Utilization Is: 83.3333 percent

Testing RR(6)...

Process cycle graph for process 2
/////--||||////////---
Process cycle graph for process 3
///////------/////////----
Process cycle graph for process 1
-----|||||||/////////////------/-------||||||/--
Process cycle graph for process 4
/////////////------/////////////-||||||-------||||||-------||||||----

Total cpu time is: 60
Total wait time is: 39
Total ready time is: 70
Number of processes: 4
Total time is: 72
Throughput Is: 0.0555556
Average Waiting Time Is: 9.75
Average Turnaround Time Is: 42.25
CPU Utilization Is: 83.3333 percent

Testing STCF() without preemption...

Process cycle graph for process 3
//----------
Process cycle graph for process 2
--||||//////---
Process cycle graph for process 1
///////////////-----|||||||-------------||||||/--
Process cycle graph for process 4
////////////////////-------||||||///////-------||||||-------||||||----

Total cpu time is: 60
Total wait time is: 39
Total ready time is: 51
Number of processes: 4
Total time is: 73
Throughput Is: 0.0547945
Average Waiting Time Is: 9.75
Average Turnaround Time Is: 37.5
CPU Utilization Is: 82.1918 percent

Testing STCF() with preemption...

Process cycle graph for process 2
--||||/---
Process cycle graph for process 3
//-----///-----
Process cycle graph for process 1
///////////////-----|||||||-------------||||||/--
Process cycle graph for process 4
////////////////////-------||||||///////-------||||||-------||||||----

Total cpu time is: 60
Total wait time is: 39
Total ready time is: 49
Number of processes: 4
Total time is: 73
Throughput Is: 0.0547945

Average Waiting Time Is: 9.75
Average Turnaround Time Is: 37
CPU Utilization Is: 82.1918 percent

DONE ALL ALGORITHMS!

cpumin.dat:

```
p1
a2
t4
e
p2
a1
t6
e
```

resultsmin.dat:

```
Testing FCFS...

Process cycle graph for process 1
----
Process cycle graph for process 2
////------

Total cpu time is:            10
Total wait time is:           0
Total ready time is:          4
Number of processes:          2
Total time is:                        10
Throughput Is:                        0.2
Average Waiting Time Is:      0
Average Turnaround Time Is:   7
CPU Utilization Is:           100 percent

Testing RR(3)...

Process cycle graph for process 1
---///-
Process cycle graph for process 2
///---/---

Total cpu time is:            10
Total wait time is:           0
Total ready time is:          7
Number of processes:          2
Total time is:                        10
Throughput Is:                        0.2
Average Waiting Time Is:      0
Average Turnaround Time Is:   8.5
CPU Utilization Is:           100 percent

Testing RR(6)...

Process cycle graph for process 1
----
Process cycle graph for process 2
////------

Total cpu time is:            10
Total wait time is:           0
Total ready time is:          4
Number of processes:          2
Total time is:                        10
Throughput Is:                        0.2
Average Waiting Time Is:      0
Average Turnaround Time Is:   7
CPU Utilization Is:           100 percent
```

Testing STCF() without preemption...

Process cycle graph for process 2

Process cycle graph for process 1
//////----

Total cpu time is: 10
Total wait time is: 0
Total ready time is: 6
Number of processes: 2
Total time is: 10
Throughput Is: 0.2
Average Waiting Time Is: 0
Average Turnaround Time Is: 8
CPU Utilization Is: 100 percent

Testing STCF() with preemption...

Process cycle graph for process 1
/----
Process cycle graph for process 2
-////-----

Total cpu time is: 10
Total wait time is: 0
Total ready time is: 5
Number of processes: 2
Total time is: 10
Throughput Is: 0.2
Average Waiting Time Is: 0
Average Turnaround Time Is: 7.5
CPU Utilization Is: 100 percent

DONE ALL ALGORITHMS!