Operating Systems – Project 1

- Simple Scheduling -

ChangYoon Lee, ChanHo Park

Dankook University, Korea Republic
32183641@gmail.com, chanho.park@dankook.ac.kr

Abstract. CPU scheduling is the art, theory, and practice of deciding which process needs to be executed in the next process. CPU scheduler selects a process from the ready queue and allocates the CPU resources to it. In the CPU scheduler, there are some concepts, system calls, and structures: CPU scheduler, inter-process communication (IPC), signal and handler, and scheduling policies with data structures. In this paper, we will discuss the concept of scheduling, the systems call used in the scheduler, and the scheduling algorithms. Also, we will implement a simple scheduling program that operates in a similar way to the actual scheduler in the operating system. At the end of this project, we will present the result of the execution of a simple scheduling program and evaluate it.

Keywords: CPU Scheduling, CPU Scheduler, Inter-Process Communication, IPC, Scheduling Algorithms, First-Come First-Served, FCFS, Round Robin, RR, Shortest Job First (SJF), Signal and Handler

1 Introduction

In a single-processor computer system, only one process can run at a single time. Other processes must wait until the CPU resources are free and can be rescheduled. A process is executed until it must wait, typically for the completion of the I/O request. However, in multiprogramming, some process always runs, to maximize CPU utilization. Multiprogramming tries to use the waiting time productively. Some process is loaded into the memory at one time, and when one process must wait, the operating system takes the CPU resources away from the process and gives the CPU resources to another process. The following progress continues, every time one process must wait, while another process takes over the use of the CPU resources. Scheduling the following progress is a fundamental operating system function. Almost all the computer resources are scheduled before use. Since the CPU is one of the primary computer resources, CPU scheduling is central to the operating system design.

In this paper, we will first explain the concepts of process, process scheduling, Inter-process communication (IPC), and CPU scheduling. By applying these concepts, we will explain how we implemented the simple scheduling program and its results for different algorithms, such as first-come-first-served (FCFS), shortest job first (SJF), round-robin (RR), and completely fair safe (CFS). Also, we will show the performance of each algorithm based on the features discussed in a later section. At the end of the paper, we will present the result of the execution of the different scheduling algorithms and compare them.

2 Requirements

Level	Process	Index	Requirement
Basic	Parent	1	Create 10 child processes.
		2	Schedule the child processes according to the Round Robin scheduling policy.
		3	Receive ALARM signal periodically by registering the timer event.
		4	Maintain run-queue and wait-queue.
		5	Accounts the remaining time quantum of all the child process and gives time slice to the child process by sending IPC message through the message queue using system calls.
	Child	1	Workload must consists of infinite loop of dynamic CPU burst and I/O curst.
		2	The values of CPU burst and I/O burst are generated randomly.
		3	When the process receives the time slice from the operating system, it makes the progress.
		4	While parent process sends the IPC message to the current child process, if the child process decreases the CPU burst value.
Optional	Parent	1	If the parent process gets the message from the child process, it checks whether the child process begins I/O burst or not.
		2	If the current process finished the CPU burst, the parent process puts it into the wait queue. If not, the current process is rescheduled again.
		3	For every time tick, parent process decreases the I/O value of the processes in the wait queue.
	Child	1	Child process makes I/O request after CPU burst. Therefore, child process must account the remaining CPU burst.
		2	If the CPU burst reaches to zero, the child sends the IPC message to the parent process with the next I/O burst.

Figure 1 - Requirement Specification

Figure 1 shows the requirements for a simple scheduling program. The implementations for these requirements will be described in detail afterwards.

3 Concepts

3.1 Process

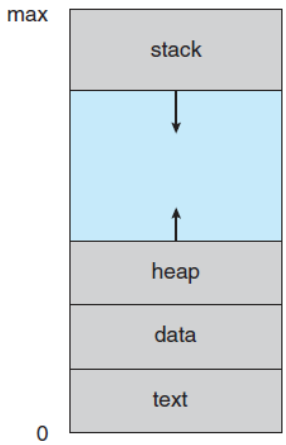

Figure 2 - Layout of the Process in the Memory (Silberschatz et al., 2014)

Modern computers allow multiple programs to be loaded into the memory and executed concurrently. For this operation, the numerous programs need to be more strictly under control and divided. Operating systems need the concept of a process, which is a program in execution, to satisfy this requirement. In a modern time-sharing system, a process represents one unit of work.

Processes are instances of programs that are running. This process is more than the program code, which is known as the text section. It is also consisting of a program counter (PC) and the contents of the registers that present the current state of the process. A process also includes the stack, which contains the temporary data, the data section, which contains global variables, and the heap, which is the memory that is dynamically allocated during the process runtime. The structure of the process is presented in Figure 2.

3.1.2 Process State

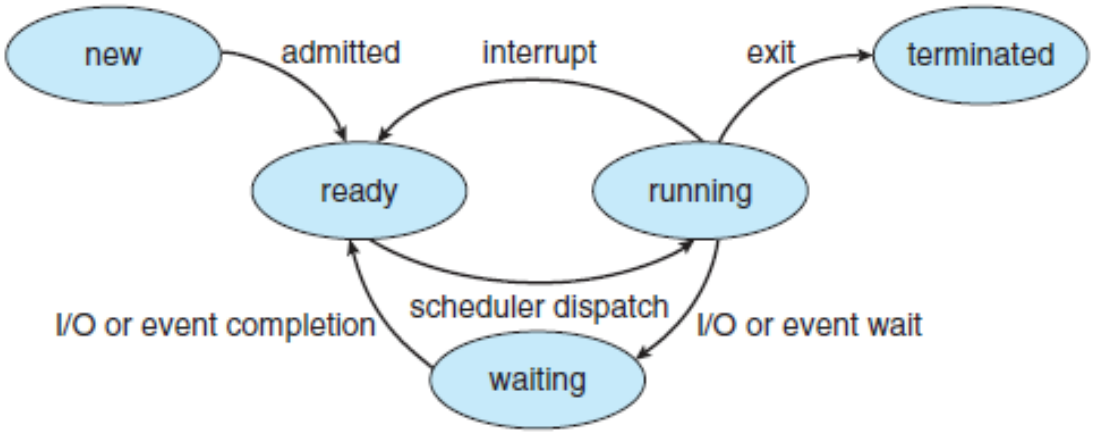

Figure 3 - Diagram of the Process State (Silberschatz et al., 2014)

As a process is in execution, it changes its state. The state of a process can be defined in the part by the current activity of the process. The process may be in one of the following states:

- **New**: The process is being created.
- **Running**: Instructions are executing.
- **Waiting**: The process waits for some event.
- **Ready**: The process waits to be assigned to a process.
- **Terminated**: The process has finished the execution.

It is substantial to realize that only one process can be run on any processor at any instant. However, many processes may be ready and waiting. The state diagram that presents these states is presented in Figure 3.

3.1.3 Process Control Block

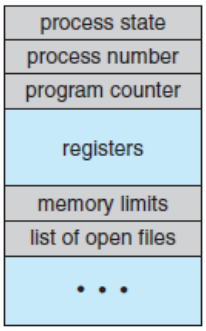

Figure 4 - Process Control Block (PCB) (Silberschatz et al., 2014)

Each process in the operating system is presented in a process control block (PCB), which is also called a task control block. The PCB is presented in Figure 4. It contains much information associated with a specific process, including the following information

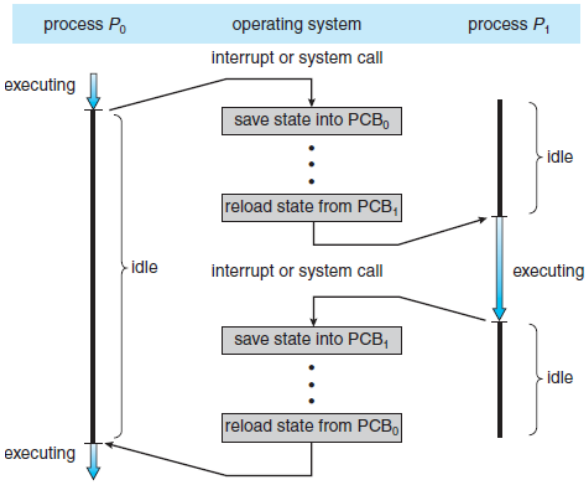

Figure 5 - Diagram of the Context Switching from Process to Processs (Silberschatz et al., 2014)

- **Process state**: The state may be new, ready, running, waiting, or terminated.
- **Program counter**: The program counter indicates the address of the next instruction to be executed for this process.
- **CPU registers**: Depending on the computer architecture, the registers vary in number and type. They include accumulators, index registers, stack pointers, and general-purpose registers. Along with the program counter, the following state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (*Silberschatz et al., 2014*). The following progress is presented in Figure 5.
- **CPU scheduling information**: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory management information**: This information may include such items as the value of the base and limit registers and the page table, or the segment tables, depending on the memory system used by the operating system (*Silberschatz et al., 2014*).

- **Accounting information**: This information includes the amount of CPU and real-time used, time limits, account numbers, and job or process numbers (*Silberschatz et al., 2014*).
- **I/O status information**: This information includes the list of I/O devices allocated to the process and a list of open files (*Silberschatz et al., 2014*).

In short, the PCB simply used as the repository for any information that may vary from process to process.

3.2 Process Scheduling

In multiprogramming, it has some process always running, to maximize CPU utilization. Also, time-sharing switches the CPU among processes so that the users can interact with each program frequently while it is running. To implement the following operations, the process scheduler selects an available process for program execution on the CPU. In the case of the system with single processor, there will not be more than one process in running state. If there are more processes, the remaining processes will have to wait until the CPU resources are free and then can be rescheduled.

3.2.1 Scheduling Queues

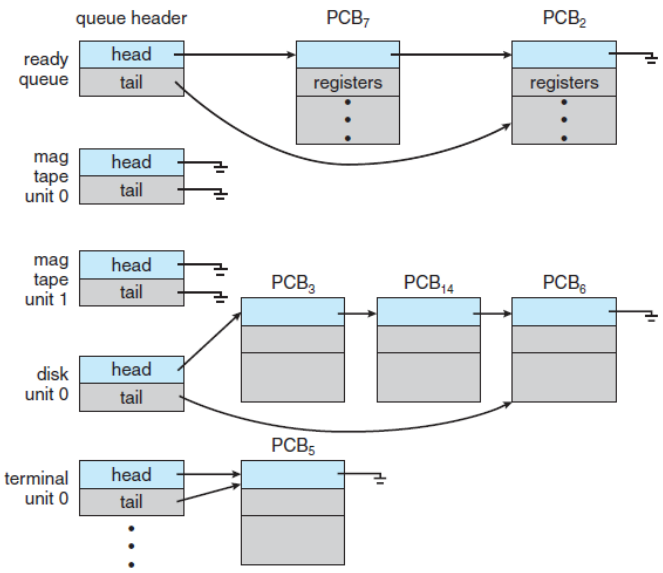

Figure 6 - Multiple Queues in the Operating System (Silberschatz et al., 2014)

As a process enter the system, it is put into the job queue, which consists of all processes in the system. The ready queue is a list that contains the processes that are stored in the main memory and are prepared and waiting to be executed.The queue is generally stored as a linked list (*Silberschatz et al., 2014*). A ready queue header contains the pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue (*Silberschatz et al., 2014*).

The system also includes the other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request (*Silberschatz et al., 2014*). Due to the system's large number of processes, another process's I/O request may be causing the disk to become busy. The process could therefore need to wait for the disk. A device queue is a list of processes that are awaiting a specific I/O device. In Figure 6, the following queue is shown.

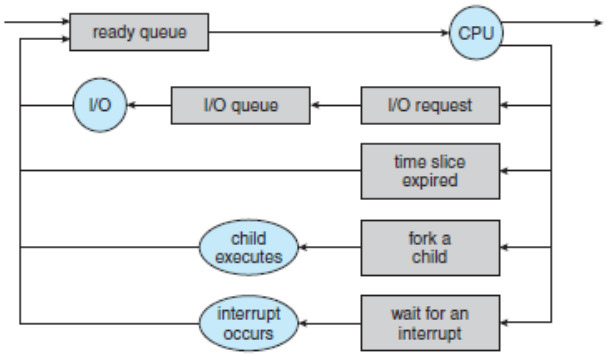

Figure 7 - Queueing-Diagram Representation of Process Scheduling (Silberschatz et al., 2014)

Figure 7 shows the representation of the queuing diagram. Each box in Figure 7 represents a queue. There are two types of queues, which are a ready queue and a set of device queues. The circles represented in Figure 7 are the resources that serve the queues, and the arrow indicates the flow of processes in the system.

A newly generated process is initially enqueued in the ready queue. It waits in the ready queue until it is selected for the next execution, or dispatched. Following the allocation of the CPU and while the process is running, the following things can happen:

- The process can send an I/O request and be added to an I/O queue.
- The process has the option to start a fresh child process and watch for its termination.
- An interrupt may force the process to be forcibly removed off the CPU and reinserted into the ready queue.

In the first two scenarios, the process is returned to the ready queue after transitioning from the waiting state to the ready state. A process continues this process until it terminates, at the time it is removed from all queues and has its PCB and resources deallocated.

3.2.2 Scheduler

Throughout its lifetime, a process migrates among the various scheduling queues. For scheduling purposes, the operating system must choose processes in some way from these queues. The selection of the process is carried out by the appropriate scheduler. There are two schedulers, which are long-term scheduler (job scheduler), and the short-term scheduler (CPU scheduler),

The long-term scheduler selects the processes to form the process pool and loads them into the memory for execution. The short-term scheduler selects the process that is ready to execute and allocates the CPU to one of them. The primary distinction between these two schedulers is in the frequency of the execution. The short-term scheduler must select a new process for the CPU frequently. However, the long-term scheduler executes with much less frequency, to create a new process. As a result, it manages the level of multiprogramming. The average rate of process creation must match the average rate of process departure for multiprogramming to be stable, and vice versa. The long-term scheduler might also only need to be used when a process exits the system. The long-term scheduler can afford to take more time deciding which process should be chosen for the execution because of the lengthy gap between executions.

In short, the long-term scheduler must make a careful selection. In most cases, most processes can be described as either I/O bound, or CPU bound. A process that is I/O-bound spends more time doing I/O operations than computing. A CPU-bound process, on the other hand, rarely generates I/O requests and spends more time performing calculations. The long-term scheduler must select a good process mix of I/O-bound and CPU-bound processes. If all the processes, are I/O bound, the ready queue will be always empty, and the short-term scheduler will have nothing to do. If all the processes are CPU bound, the I/O waiting queue will almost always be empty, hardware resources are not used, and again the system will be unbalanced. The system that performs the best will be comprised of both CPU-bound and I/O-bound processes.

3.3 Inter-Process Communication (IPC)

In the operating system, concurrently running processes can either be independent or collaborative. Other processes running in the system cannot affect or be affected by an independent process. A process is independent if it does not exchange data with any other processes. However, a process is cooperating if it can affect or be affected by the other processes executing in the system, which means that any process that shared data with other processes is a cooperating process.

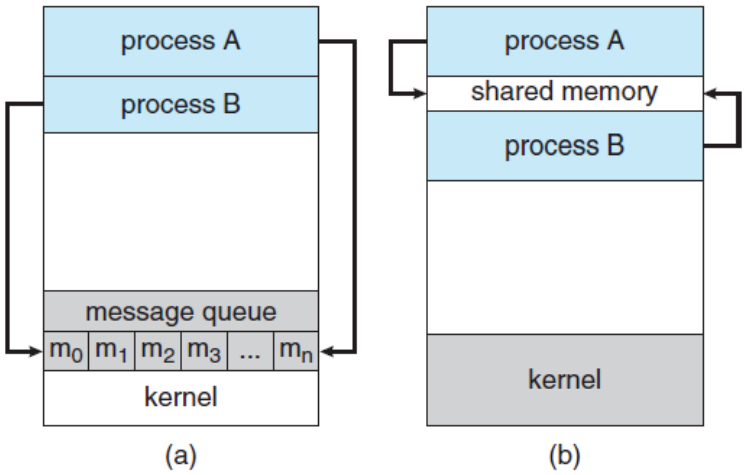

Figure 8 - Communications Models of Shared memory and Message Passing (Silberschatz et al., 2014)

Cooperating processes require an inter-process communication (IPC) operation that will allow them to exchange data and information. Shared memory and message passing are the two main types of inter-process communication. A memory area that is shared by cooperating processes is established in the shared memory model.The process can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place to be means of messages exchanged between the cooperating processes. The following models are presented in Figure 8.

Both models are commonly used in operating systems. Message passing is useful for exchanging smaller amounts of data because no conflicts need to be avoided. Since message-passing systems are frequently built using system calls and necessitate the more time-consuming job of kernel intervention, shared memory can be faster than message passing. In shared memory systems, system calls are required only to establish shared memory regions. Once the shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required (*Silberschatz et al., 2014*).

Latest research indicates that message passing provides better performance than the shared memory on such systems. Shared memory suffers from cache coherency issues, which arise because shared data migrate among several caches. As the number of processing cores on the systems increases, it is possible that message passing is the preferred method for the IPC.

3.3.1 Message Passing

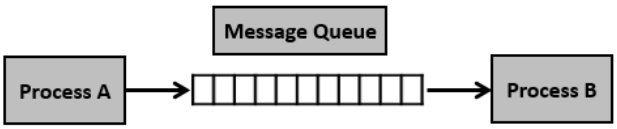

Figure 9 - Message Queue Structure (Message queues)

Message passing provides an operation that allows processes to communicate and synchronize their action without sharing the same address space. Processes A and B need to send and receive messages from one another to communicate. There must be a channel of communication between them. This link can be implemented in a variety of ways: direct or indirect communication, synchronous or asynchronous communication, and automatic or explicit buffering.

3.3.2 Message Queue Naming

Processes must be able to refer to one another to communicate. They can communicate directly or indirectly. Each process that wishes to communicate must formally identify either the sender or the receiver of the communication in direct communication. This means that both the sender process and receiver process must name the other to communicate.

A variant of this concept is indirect communication. In this case, only the sender names the receiver, while the receiver does not require the name of the sender. With indirect communication, the messages are sent to and received from the mailboxes, also known as ports. A mailbox can be conceptualized as an object that the processes can use to insert messages into and remove messages from. Each mailbox has a unique identification. A process can communicate with another process with several different mailboxes, but two processes can communicate only if they have a shared mailbox.

By default, the owner of a mailbox is the process that creates it. The owner is the only process that can initially access this mailbox to receive messages. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. The following provision can result in multiple receivers for each mailbox.

3.3.3 Message Queue Synchronization

Calls to send and receive primitives are the primary means of communication between processes. The implementation of these privities can be differed from blocking to nonblocking, which are also known as synchronous and asynchronous. Each primitive can be represented as below:

- **Blocking send**: Until the message is received by the receiving process or by the mailbox, the sending process is blocked.
- **Nonblocking send**: The sending process sends the message and resumes the operation.
- **Blocking receive**: The receiver blocks until the message is available.
- **Nonblocking receive**: The receiver retrieves either a valid message or not.

3.3.4 Message Queue Buffering

Generally, the queues can be implemented in three ways:

- **Zero capacity**: The link cannot have any messages waiting in it, and the queue can only have a maximum length of zero. The sender must block the queue in the situation below until the recipient receives the message.
- **Bounded capacity**: The queue has a finite length n of the queue, and n messages can reside in it. When a new message is sent, if the queue is not already full, the message is added to it and transmitted, allowing the sender to continue without stopping. However, the capacity of the link is finite. Therefore, if the link is full, the sender must block it until the space is available in the queue.
- **Unbounded capacity**: The length of the queue is infinite, and any number of messages can wait in the queue. The sender never blocks the queue.

The zero-capacity case is referred to as a message system with no buffering. Other cases are referred to as systems with automatic buffering.

3.4 CPU Scheduling

Whenever the CPU becomes idle, the operating system must select the process in the ready queue to be executed. The selection of the process is carried out by the CPU scheduler. CPU scheduler selects a process from the process in the memory that is ready to execute and allocates the CPU to that process.

We denote that the ready queue is not necessarily a first-in, first-out (FIFO) queue. We can consider the various scheduling algorithms, and the ready queue can be implemented in not only the FIFO queue, but also the priority queue, tree, or simply an unordered linked list. All the processes, nevertheless, are lined up in the ready queue and are awaiting their turn to run on the CPU. The process control block (PCB) of the processes is often represented by the records in the queues.

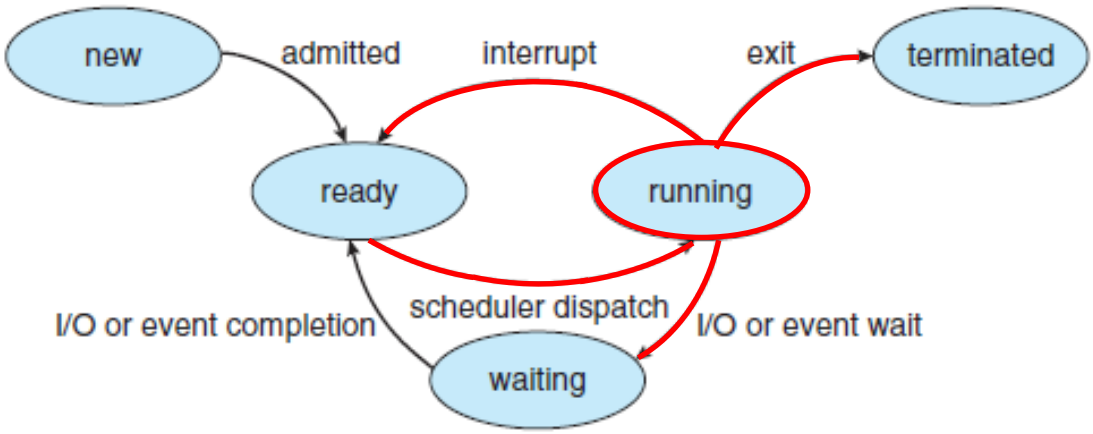

Figure 10 - Diagram of the Process State where Scheduler Affects (Silberschatz et al., 2014)

CPU scheduling decisions can take place under the following circumstances, which are presented in Figure 10:

- When a process switches from a running state to a waiting state.
- When a process switches from a running state to a ready state.
- When a process switches from the waiting state to the ready state.
- When the process terminates.

There is no other option in terms of scheduling for cases 1 and 4. It is necessary to choose a new process to run. However, there is a choice for situations 2 and 3. When scheduling takes place only under 1 and 4, the scheduling is non-preemptive or cooperative. In contrast, it is preemptive.

Once the CPU has been assigned to a process under non-preemptive scheduling, the process retains the CPU until it releases it either by terminating or by moving to the waiting state. However, preemptive scheduling can result in race conditions when the data are shared among several processes.

3.4.1 Scheduling Criteria

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

The following criteria are the characteristics of the scheduling algorithms that are used for the comparison to make a substantial difference in which algorithm is judged to be best:

- **CPU utilization**: What we want to do is to keep the CPU as busy as possible. The range that shows how busy the CPU is working is CPU utilization.

- **Throughput**: If the CPU is busy executing thee processes, then the work is being done. Throughput presents the number of completed processes per time unit.

- **Turnaround time**: The duration of the process from the perspective of that particular process is a crucial criterion. The interval from the time of the submission of a process to the time of the completion is the turnaround time. This time stands for the sum of periods spent waiting in the ready queue, executing on the CPU, and doing the I/O operations.

- **Waiting time**: The length of time required to complete an I/O operation or for a process to execute is unaffected by the CPU scheduling mechanism. It only has an impact on how long a procedure waits before being made ready. The total amount of time spent waiting is the length of time in the ready queue.

- **Response time**: In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output early and can continue computing the new results while the previous results are outputting. Also, another measure is the time from the submission of a request until the first response is produced. The following measurement is the response time, which is the time it takes to start responding.

It is desirable to maximize the CPU utilization and throughput and to minimize the turnaround time, waiting time, and response time. In general, we optimize the average measurement. In other cases, though, we prefer optimizing the minimum or maximum values over the average.

3.4.2 Scheduling Algorithms

The issue of choosing which task in the ready queue should receive a CPU core is dealt with by CPU scheduling. There are many different CPU scheduling algorithms. In the later sections, we will describe some of them with the implementation. Although most modern CPU architectures have multiple processing cores, we will describe the scheduling algorithms in the context of a single processing core, which means that the system is only capable of running one process at a time.

4 Simple Scheduling

4.1 Signal and Handler

A program, which is also known as a process, must deal with unexpected or unpredictable events, such as a floating-point error, an alarm clock rings, a termination request from a user, and so on. In the following situations, the signal is used. Signals are the software interrupts and provide a way to handle asynchronous events. As a signal is generated by the event, it is delivered to a process, which then takes the action in response to the signal. Upon delivery of a signal, a process can operate in the following ways:

- **Ignore it**: literally, ignore the signal.
- **Catch it**: A signal handler is a specialized piece of code that a programmer can use to process a specific signal.
- **Accept the default**: Have the kernel do whatever is defined as the default action for the signals, such as the signal is ignored, the process is terminated, the process is stopped, and so on.

SIGNAL	NBR	DESCRIPTION	DEFAULT ACTION
SIGABRT	6	Process abort (sent by assert)	Implementation dependent
SIGALRM	14	alarm clock	Abnormal termination
SIGCONT	25	Execution continued if stopped	Continue
SIGFPE	8	Arithmetic error (i.e., divide by zero)	Implementation dependent
SIGINT	2	Interactive attention signal (Ctrl-C)	Abnormal termination
SIGKILL	9	Termination (cannot be caught or ignored)	Abnormal termination
SIGQUIT	3	Interactive termination; core dump	Implementation dependent
SIGSEGV	11	Segmentation fault	Abnormal termination
SIGSTOP	23	Execution stop (cannot be caught or ignored)	Stop
SIGTERM	15	Termination	Abnormal termination
SIGUSR1	16	User-defined signal 1	Abnormal termination
SIGUSR2	17	User-defined signal 2	Abnormal termination

Figure 11 - POSIX Signals

In the signal header file, which is offered by the POSIX system, the signals presented in Figure 11 are defined as the integer value.

Signals can occur during the process execution, and we can also generate the signal explicitly to another process. Generating the signal explicitly is called signal dispatching.

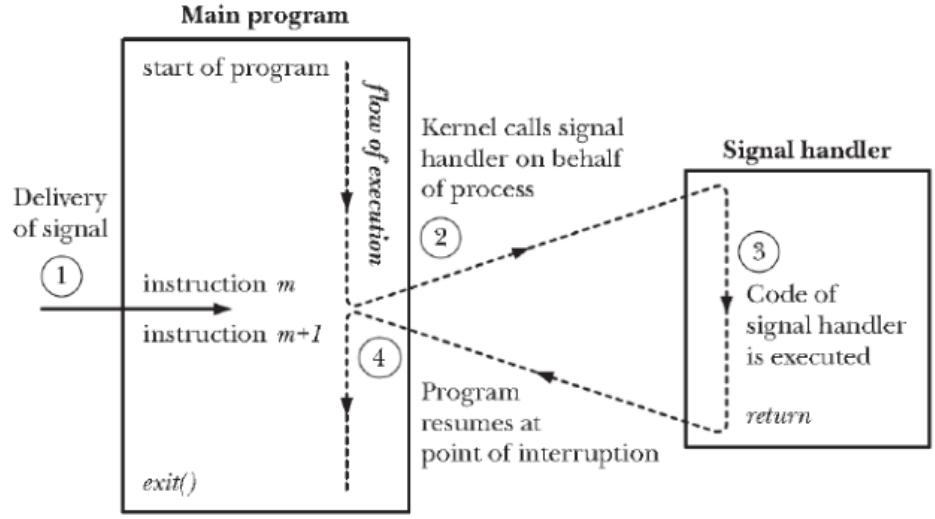

Figure 12 - Signal Delivery and Handler Execution (Can I Trap Sigsegv, 1966)

As the signals are generated asynchronously, we cannot predict the period of the signal generation, which means that the signal can be generated whenever the program is in execution. Figure 12 presents the operation of the signal delivery and the handler execution. According to Figure 12, first, if the signal occurs while the program is executing, it temporarily stops the execution. Next, the signal handler is executed to proceed with the signal. Then, right after the signal proceeds, the next instruction, right after the signal is executed.

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

Figure 13 – Structure of the SIGACTION System Call (SIGACTION(2) - linux manual page)

The POSIX system provides the sigaction system call to register the signal handler. The following system call can be used to modify the action that a process does in response to a particular signal. Figure 13 presents the structure of the sigaction system call.

In this project, we set three main signals, which are the I/O signal, CPU signal, and time clock (tick) signal. In the I/O signal, the program ferrets out each PCB in the waiting queue and decreases the I/O burst time. If the I/O burst time is zero, it pops the PCB from the waiting queue and appends it into the ready queue. In the CPU signal, it selects the next executing process from the ready queue and decreases the CPU burst time. In the time clock (tick) signal, it increases the time tick of the CPU, and prints out the result after executing the I/O and CPU signal.

4.2 Message Passing in System V Messages

System V messages allow the processes to exchange data in the form of messages. Messages can be assigned to a specific type. The server process can direct message traffic between clients to its queue by using the client process PID as the message type. For single message transactions, multiple server processes can work in parallel with transactions that are sent to a shared message queue.

Before a process can send or receive a message, it must initialize the queue through the msgget system call. The owner or creator of the queue can change their ownership or permissions using the msgctl system call. Any process with permission can use the msgctl system call for control operation. Operations to send and receive messages are performed by msgsnd and msgrcv system calls. A message's text is copied to the message queue when it is sent. The system calls msgsnd and msgrcv can be executed as blocking or non-blocking activities. Until one of the following three events takes place, the call succeeds, the process receives a signal, or the queue is cleared, the blocked message action is paused.

```
#include <sys/ipc.h>
 #include <sys/msg.h>

 ...
        key_t   key;            /* key to be passed to msgget() */
        int     msgflg, /* msgflg to be passed to msgget() */
                        msqid;  /* return value from msgget() */
        ...
        key = ...
        msgflg = ...
        if ((msqid = msgget(key, msgflg)) == -1)
        {
                perror("msgget: msgget failed");
                exit(1);
        } else
                (void) fprintf(stderr, "msgget succeeded");
        ...
```

Figure 14 - msgget System Call Code (Msgget(2) - linux manual page)

msgget system call initializes a new message queue. It can also return the message queue ID to the queue corresponding to the key argument. The values passed as the message flag argument must be an octal integer with a setting for the queue's permissions and control flags. The following system call is presented in Figure 14.

```
#include                        <sys/types.h>
 #include                       <sys/ipc.h>
 #include                       <sys/msg.h>

        ...
        if (msgctl(msqid, IPC_STAT, &buf) == -1)  {
                perror("msgctl: msgctl failed");
                exit(1);
        }
        ...
        if (msgctl(msqid, IPC_SET, &buf) == -1) {
                perror("msgctl: msgctl failed");
                exit(1);
        }
        ...
```

Figure 15 - msgctl System Call Code (MSGCTL(2) - linux manual page)

msgctl system call alters the permissions and other characteristics of the message queue. The message queue ID argument must be the ID of an existing message queue. The commands of the msgctl system call are the IPC_STAT (place information about the status of the queue in the data structure pointed to buffer), the IPC_SET (Set the owner's user and group ID, the permissions, and the size of the message queue), and the IPC_RMID (remove the message queue specified by the message queue ID argument. The following system call is presented in Figure 15.

```
#include <sys/types.h>
 #include <sys/ipc.h>
 #include <sys/msg.h>

 ...
        int                             msgflg; /* message flags for the operation */
        struct msgbuf   *msgp;  /* pointer to the message buffer */
        size_t                  msgsz;  /* message size */
                size_t                  maxmsgsize;
        long                            msgtyp; /* desired message type */
        int                             msqid           /* message queue ID to be used */
        ...
        msgp = malloc(sizeof(struct msgbuf) - sizeof (msgp->mtext)
                                                        + maxmsgsz);
        if (msgp == NULL) {
                (void) fprintf(stderr, "msgop: %s %ld byte messages.#n",
                                "could not allocate message buffer for", maxmsgsz);
                exit(1);
                ...
                msgsz = ...
                msgflg = ...
                if (msgsnd(msqid, msgp, msgsz, msgflg) == -1)
                        perror("msgop: msgsnd failed");
                ...
                msgsz = ...
                msgtyp = first_on_queue;
                msgflg = ...
                if (rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg) == -1)
                        perror("msgop: msgrcv failed");
                ...
```

Figure 16 - msgsnd and msgrcv System Calls Code (MSGSND(3P) - linux manual page)

msgsnd and msgrcv system calls send and receive messages. The message queue ID argument must be the ID of the existing message queue, The message pointer argument is a pointer to a structure that contains the type of the message and its text. The message size argument specifies the length of the message in bytes. The message flag argument passes the various control flags. The following system call is presented in Figure 16.

4.3 First-Come First-Served Algorithm (FCFS)

The simplest CPU scheduling is the first-come, first-serve (FCFS) scheduling algorithm. In this algorithm, the process that requests the CPU resources first is allocated to the CPU first. The implementation of the FCFS policy is simply managed by a FIFO queue. The process PCB is connected to the tail of the queue when it enters the ready queue. aThe running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.

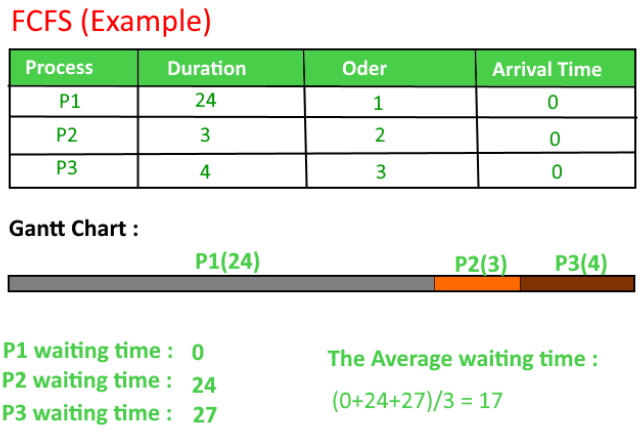

Figure 17 - Example of FCFS Scheduling (Program for FCFS CPU scheduling: Set 1 2022)

Figure 17 shows an example of the FCFS scheduling case. As we can see from Figure 17, even though the FCFS is easy to implement, it has the disadvantage that its average waiting time is often quite long. Also, consider the performance of FCFS scheduling in a dynamic situation. If we have one CPU-bound process and multiple I/O-bound processes. The CPU-bound process will acquire and hold the CPU as it moves through the system. All other programs will complete their I/O during this period and go into the ready queue to wait for the CPU. The I/O devices are not in use while the processes are in the ready queue. All I/O-bound processes, which have short CPU bursts, execute fast and move back to the I/O queues after the CPU-bound process has finished its CPU bursts and moved to an I/O device. At this point, the CPU sits idle. The following situation is called the convoy effect, which presents the case of all the other processes waiting for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first (*Silberschatz et al., 2014*).

4.4 Shortest Job First Algorithm (SJF)

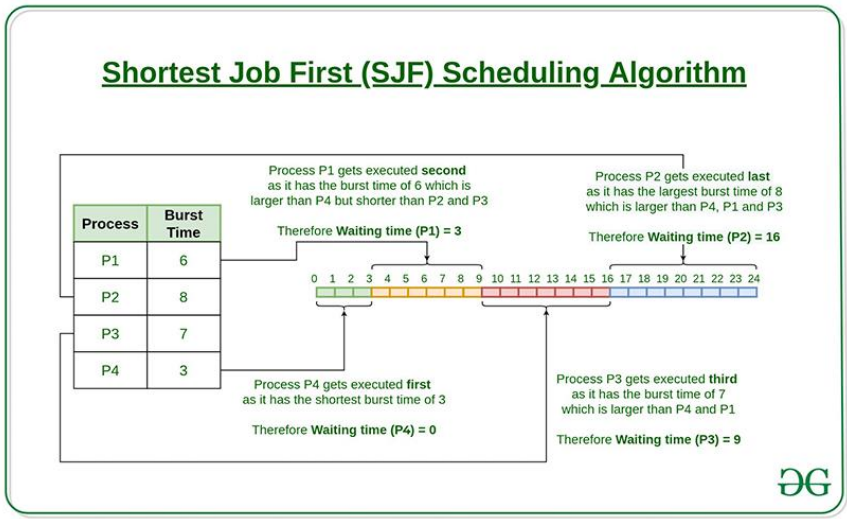

Figure 18 – Example of SJF Scheduling (Program for shortest job first (or SJF) CPU scheduling: Set 1, 2022)

A different approach to CPU scheduling is the shortest job first scheduling algorithm. This algorithm is associated with each process, the length of the process's next CPU burst time. When CPU resources are available, it is assigned to the process that has the smallest next CPU burst time. If the next CPU bursts of the two processes are the same, FCFS scheduling is used. Figure 18 presents the execution of the SJF scheduling.

The SJF scheduling algorithm is probably optimal, due to that gives the minimum average waiting time for a given set of processes. Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process (*Silberschatz et al., 2014*). As a result, the average waiting time decreases.

However, the SJF scheduling algorithm cannot be implemented at the level of CPU scheduling, as there is no way to know the length of the next CPU burst. One approach to this problem is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value (*Silberschatz et al., 2014*). By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst (*Silberschatz et al., 2014*).

4.5 Round Robin Algorithm (RR)

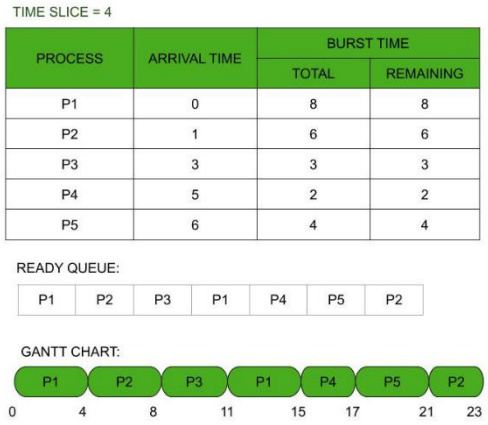

Figure 19 - Example of RR Scheduling (Round robin scheduling algorithm, 2020)

The round-robin scheduling algorithm is like FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, which is time quantum or time slice, is defined. Generally, time quantum has a range of 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. As CPU is allocated to each process in the ready queue, the CPU scheduler loops around the queue once every time quantum. To implement RR scheduling, we need to treat the ready queue as a FIFO queue of processes. New processes have been added to the tail of the ready queue. Processes are dispatched by the CPU scheduler after they are selected from the ready queue, set a timer for one quantum interval, and are selected from the queue. Figure 19 shows the execution of the RR scheduling algorithm.

One of two things will happen when we apply the RR scheduling algorithm. In some cases, CPU bursts are less than 1 time quantum. The CPU will be released voluntarily by the process itself in this case. However, if the CPU burst of the currently running process is longer than the 1-time quantum, the timer will go off and will cause an interruption to the operating system. The process will then be moved to the end of the ready queue when a context switch is carried out.

We can therefore conclude that the size of the time quantum has a significant impact on the RR scheduling algorithm's performance. The RR policy is identical to the FCFS policy in one extreme scenario, where the time quantum is extraordinarily big. In contrast, the RR approach might lead to numerous context switches if the time quantum is very small.

Additionally, the size of the time quantum affects turnaround time. As the time quantum rises, the average turnaround time of a group of processes does not always get better. Generally, if most processes complete their subsequent CPU burst in a single time quantum, the average turnaround time can be reduced. The average turnaround time grows much more for a smaller time quantum when context switch time is taken in because more context switches are needed.

4.6 Program Definition

Before implementing the multi-threaded word count program, we will state the additional program definition that will be used in the real implementation.

- Global Variables

Variables	Data Type	Definition
RUN_TIME	10000	Limitation of the program run time.
MAX_PROCESS	10	Number of the child processes.
TIME_QUANTUM	5	Time quantum for the round robin (RR) algorithm.
TIME_TICK	1000	SIGALRM interval time, which is 0.001 second (10 millisecond).
Counter	Int	Counts the time tick of CPU.
Run_time	Int	Substitution of the RUN_TIME and the Counter.
Readyq	Void*	Pointer that points at the ready queue.
Waitq	Queue*	Pointer that points at the waiting queue.
Max_limit	Int	Limitation of the maximum CPU and I/O burst time.
Proc_count	Int	Counter that counts the completed process.
Cur_wait	Node*	Pointer that points at the current node in the waiting queue.
Cur_ready	Node*	Pointer that points at the current node in the ready queue.
Wait_time	Double	Variable used to calculate the total waiting time among the processes.
Set_scheduler	Int	Index that is used to choose the scheduling algorithm.
Turnaround_time	Double	Variable used to calculate the total turnaround time among the processes.
Key	Int[]	Keys that are used to access the message queue between processes.
Cpid	Pid_t[]	Array for storing process ID of the child processes.
Io_burst	Int[]	Array for storing I/O burst time of the child processes.
Cpu_burst	Int[]	Array for storing CPU burst time of the child processes.
End_time	Int[]	Array for storing the end time of the child processes.
Service_time	Int[]	Array for storing the service time of the child processes.
Arrival_time	Int[]	Array for storing the arrival time of the child processes.

- Modules and Functions

Modules	Functions	Definition
Queue	CreateNode	Creates the new node and returns it.
	IsEmpty	Returns whether the following queue is empty or not.
	CreateQueue	Creates the new queue and returns it.
	Dequeue	Dequeues the node from the front of the following queue.
	RemoveQueue	Removes all the nodes in the queue. After removing them, it frees the queue.
	PrintQueue	Prints all the data for each node in the following queue.
	FprintQueue	Prints all the data for each node in the following queue to the text file.
	Enqueue	Enqueues the new node into the rear of the following queue.
Heap	CreateHeap	Creates the new heap and returns it.
	RemoveHeap	Removes all the nodes in the heap. After removing them, it frees the heap
	DeleteHeap	Deletes the node from the front of the following heap.
	PrintHeap	Prints all the data for each node in the following heap.
	Heapify	Readjust the following array into the heap.
	FprintHeap	Prints all the data for each node in the following heap to the text file.
	InsertHeap	Insert the new node into the following heap.
Msg	Cmsgsnd	Message send function of the child process. Child process sends the message structure to the parent process through the IPC communication.
	Pmsgrcv	Message receive function of the parent process. Parent process receives the message structure from the child process through the IPC communication.
Main	Dump_data	Writes the data of ready queue dump and wait queue dump on the text file.
	Signal_io	Signal handler that checks whether the child process has the IO burst or not.
	Signal_rr	Signal handler that enqueues the current CPU process into the end of the ready queue and dequeues the next process from the ready queue which is to be executed. The applied scheduling algorithm is round robin policy.
	Signal_sjf	Signal handler that enqueues the current CPU process into the end of the ready queue and dequeues the next process from the ready queue which is to be executed. The applied scheduling algorithm is shortest hob fist policy.
	Signal_fcfs	Signal handler that enqueues the current CPU process into the end of the ready queue and dequeues the next process from the ready queue which is to be executed. The applied scheduling algorithm is first-come, first-served policy.
	Signal_count	Signal handler of SIGALRM which is called for every count.

5 Implementation

5.1 Queue Header

```
typedef struct PCB {
        int idx;
        pid_t pid;
        int io_burst;
        int cpu_burst;
} PCB;
```

```
typedef struct Node {
        PCB pcb;
        struct Node* next;
} Node;
```

```
typedef struct Queue {
        int count;
        Node* tail;
        Node* head;
} Queue;
```

Figure 20 - Structures used in the Queue

Figure 20 shows the process control block (PCB), node, and queue structures that are used in the implementation of the queue. As the PCB stores the information of the process, we set the PCB to contain the index, process ID (PID), I/O burst time, and CPU burst time. Then, in the node, which is used in the queue, we set the node to contain the PCB with the node pointer that points to the next node in the queue. Finally, in the queue structure, we set it to contain the count, which represents the number of the node in the queue, tail, and head, which are pointing to the last node and the first node of the queue.

```
Node* createNode();
int isEmpty(Queue* q);
Queue* createQueue();
Node* dequeue(Queue* q);
void removeQueue(Queue* q);
void printQueue(Queue* q, char c);
void fprintQueue(Queue* q, char c, FILE* fp);
void enqueue(Queue* q, int idx, int cpu_burst, int io_burst);
```

Figure 21 - Functions used in the Queue

Figure 21 shows the functions of the queue operations. We can create the node and the queue by using the createNode and createQueue functions. By using dequeue and enqueue functions, we can pop out the first node in the queue and insert the node into the end of the queue structure. Each printQueue and fprintQueue function are used to output the progress and the results of the program execution on the screen and the dump file. After all the operations, we can remove the queue structure and the remaining nodes by using the removeQueue function.

The following queue header is used in the implementations of first-come, first-served (FCFS) and round-robin (RR) scheduling algorithms.

5.2 Heap Header

```
typedef struct Heap {
        int count;
        Node** heap;
} Heap;
```

Figure 22 - Structure used in the Heap

Figure 22 shows the heap structures that are used in the implementation of the heap, which is also called a priority queue. In the case of the process control block (PCB) and node structure, we used the same ones that we implemented in the queue header. In the heap structure, we set it to contain the count, which represents the number of nodes in the heap, and the heap pointer, which is pointing the root node of the heap.

```
Heap* createHeap();
void removeHeap(Heap* h);
Node* deleteHeap(Heap* h);
void printHeap(Heap* h, char c);
void heapify(Heap* h, int index);
void fprintHeap(Heap* h, char c, FILE* fp);
void insertHeap(Heap* h, int idx, int cpu_burst, int io_burst);
```

Figure 23 - Functions used in the Heap

Figure 23 shows the functions of the heap operations. We can create the heap by using the createHeap function. By using deleteHeap and insertHeap functions, we can delete and get the root node in the heap and insert the node into the end of the heap structure and perform the heapify operation. Each printHeap and fprintHeap function are used to output the progress and the results of the program execution on the screen and the dump file. After all the operations, we can remove the heap structure and the remaining nodes by using the removeHeap function.

The following queue header is used in the implementation of shortest job first (SJF) scheduling algorithms.

5.3 Inter-Process Control (IPC) Message Passing Header

```
typedef struct msgbuf {
        long  mtype;

        // pid will sleep for io_time
        int pid;
        int io_time;
        int cpu_time;
} msgbuf;
```

Figure 24 - Structure of the Message Buffer

According to the previous section, we can see that inter-process control (IPC) message passing is one of the operations that can control from one process to another process by sending and receiving the messages. To implement the following operations, we implemented the structure of the message that is used in the message-passing communication between the parent process and child processes. First, we defined the type of the message, to specify the role of the message. Then, we set the message structure to contain the information of process id (PID), I/O burst time, and CPU burst time, to notice that the child process has processed how much time tick for its CPU or I/O burst operation. After getting this message from the child processes, the parent process can identify what has happened to the child processes during the time tick. The implemented code of the message structure is presented in Figure 24.

```
void cmsgsnd(int k, int c, int io) {
        int qid = msgget(k, IPC_CREAT | 0666);

        msgbuf msg;
        memset(&msg, 0, sizeof(msgbuf));

        msg.mtype = 1;
        msg.io_time = io;
        msg.cpu_time = c;
        msg.pid = getpid();

        if (msgsnd(qid, &msg, sizeof(msgbuf) - sizeof(long), 0) == -1) {
                perror("msgsnd");
                exit(EXIT_FAILURE);
        }
}
```

Figure 25 - Message Send Function of the Child Process

Figure 25 shows the message send function that is used in the child process. First, the child process gets the message queue ID (QID) from the message queue. Then by setting the parameters in the message structure, it sends the message to the parent process through the message queue that has the ID of QID. If the message is sent successfully, it terminates the operation. If not, it will print out the error statement.

```
void pmsgrcv(int idx, Node* node) {
        int k = 0x12345 * (idx + 1);
        int qid = msgget(k, IPC_CREAT | 0666);

        msgbuf msg;
        memset(&msg, 0, sizeof(msgbuf));

        if (msgrcv(qid, &msg, sizeof(msgbuf) - sizeof(long), 0, 0) == -1) {
                perror("msgrcv");
                exit(EXIT_FAILURE);
        }

        node->pcb.pid = msg.pid;
        node->pcb.io_burst = msg.io_time;
        node->pcb.cpu_burst = msg.cpu_time;
}
```

Figure 26 - Message Receive Function of the Parent Process

Figure 26 shows the message-receiving function that is used in the parent process. First, the parent process generates the key that is used to create the message queue and gets the message queue ID (QID) by using the following key. After that, the parent process checks whether the message has been received in the message queue. If there is a received message in the queue, it reads it. If not, wait for the message to be filled out, and if the error occurs, it will print out the error statement.

5.4 Main

```
void dump_data(FILE* fp) {
        switch (set_scheduler) {
        case 1: fp = fopen("fcfs_dump.txt", "a+"); break;
        case 2: fp = fopen("sjf_dump.txt", "a+"); break;
        case 3: fp = fopen("rr_dump.txt", "a+"); break;
        default:
                perror("scheduler");
                exit(EXIT_FAILURE);
        }

        fprintf(fp, "----------------------------------------\n");
        fprintf(fp, "Time: %d\n", RUN_TIME - run_time);
        fprintf(fp, "CPU Process: %d -> %d\n", cur_ready->pcb.idx, cur_ready->pcb.cpu_burst);

        for (int i = 0; i < waitq->count; i++) {
                cur_wait = dequeue(waitq);
                fprintf(fp, "I/O Process: %d -> %d\n", cur_wait->pcb.idx, cur_wait->pcb.io_burst);
                enqueue(waitq, cur_wait->pcb.idx, cur_wait->pcb.cpu_burst, cur_wait->pcb.io_burst);

        }

        if (set_scheduler == 2) fprintHeap(readyq, 'r', fp);
        else fprintQueue(readyq, 'r', fp);
        fprintQueue(waitq, 'w', fp);
        fprintf(fp, "----------------------------------------\n");
        fclose(fp);
}
```

Figure 27 - Dump Data Function

Figure 27 shows the code of the dump data function that dumps the progress of the program execution and the result into the dump file. In this function, we output the status of the ready and waiting queue to the screen and the dump file.

```
/*
 * void signal_io(int signo)
 *
 * Signal handler that checks whether the child process has the IO burst or not.
 * If it has the remained IO burst, it enqueues the process into the wait queue.
 * If not, it enqueues the process into the ready queue.
 */
void signal_io(int signo) {
        pmsgrcv(cur_ready->pcb.idx, cur_ready);
        if (cur_ready->pcb.io_burst == 0) {
                if (set_scheduler == 2) insertHeap(readyq, cur_ready->pcb.idx, cur_ready->pcb.cpu_burst, cur_ready->pcb.io_burst);
                else enqueue(readyq, cur_ready->pcb.idx, cur_ready->pcb.cpu_burst, cur_ready->pcb.io_burst);
        }

        else enqueue(waitq, cur_ready->pcb.idx, cur_ready->pcb.cpu_burst, cur_ready->pcb.io_burst);

        if (set_scheduler == 2) cur_ready = deleteHeap(readyq);
        else cur_ready = dequeue(readyq);
        counter = 0;
}
```

Figure 28 - I/O Operation Signal

Figure 28 presents the implemented code of the I/O operation signal. Ever since the I/O operation signal is generated, it checks the message queue whether there is a message or not. If the received message contains the information of I/O burst time with zero, then it enqueues or inserts it into the ready queue. If not, it enqueues the node into the waiting queue again. After the following progress, it pops out the node in front of the ready queue.

```
/*
 * void signal_fcfs(int signo)
 *
 * Signal handler that enqueues the current cpu process into the end of the ready queue
 * and dequeues the next process from the ready queue which is to be executed.
 */
void signal_fcfs(int signo) {
        cur_ready->pcb.cpu_burst--;
        if (cur_ready->pcb.cpu_burst == 0) {
                enqueue(readyq, cur_ready->pcb.idx, cur_ready->pcb.cpu_burst, cur_ready->pcb.io_burst);
                cur_ready = dequeue(readyq);
        }
}


/*
 * void signal_sjf(int signo)
 *
 * Signal handler that insert the current cpu process into the end of the ready queue
 * and deletes the next process from the ready queue which is to be executed.
 */
void signal_sjf(int signo) {
        counter++;
        cur_ready->pcb.cpu_burst--;
        if (cur_ready->pcb.cpu_burst == 0) {
                insertHeap(readyq, cur_ready->pcb.idx, cur_ready->pcb.cpu_burst, cur_ready->pcb.io_burst);
                cur_ready = deleteHeap(readyq);
        }
}


/*
 * void signal_rr(int signo)
 *
 * Signal handler that enqueues the current cpu process into the end of the ready queue
 * and dequeues the next process from the ready queue which is to be executed.
 */
void signal_rr(int signo) {
        counter++;
        cur_ready->pcb.cpu_burst--;
        if (counter >= TIME_QUANTUM) {
                enqueue(readyq, cur_ready->pcb.idx, cur_ready->pcb.cpu_burst, cur_ready->pcb.io_burst);
                cur_ready = dequeue(readyq);
                counter = 0;
        }
}
```

Figure 29 - Signals for CPU Scheduling Algorithms

Figure 29 shows the implemented code of the signals that are used to schedule the CPU with different scheduling policies. First, in a signal of FCFS policy decrease the CPU burst time of the current node, which is the front node of the ready queue, and if the CPU burst time converges to zero, it enqueues the node into the waiting queue and pops out the node from the ready queue. Next, in a signal of SJF policy decreases the CPU burst time of the current node, which is the root node of the ready heap, and if the CPU burst time converges to zero, it enqueues the node into the waiting queue and deletes the node from the ready heap. Third, a signal of RR policy decreases the CPU burst time of the current node, which is the front node of the ready queue, and if the CPU burst time converges to zero, it enqueues the node into the waiting queue. If the CPU burst time does not converge to zero and it spent all its allocated time quantum, the node is enqueued in the ready queue.

```
if (cur_ready->pcb.cpu_burst == 1) {
        proc_count++;
        end_time[cur_ready->pcb.idx] = RUN_TIME - run_time;
        wait_time += end_time[cur_ready->pcb.idx] - arrival_time[cur_ready->pcb.idx];
        turnaround_time += end_time[cur_ready->pcb.idx] - service_time[cur_ready->pcb.idx] - arrival_time[cur_ready->pcb.idx];
}

int length = waitq->count;
for (int i = 0; i < length; i++) {
        cur_wait = dequeue(waitq);
        cur_wait->pcb.io_burst--;
        printf("I/O Process: %d -> %d\n", cur_wait->pcb.idx, cur_wait->pcb.io_burst);
        if (cur_wait->pcb.io_burst == 0) {
                if (set_scheduler == 2) insertHeap(readyq, cur_wait->pcb.idx, cur_wait->pcb.cpu_burst, cur_wait->pcb.io_burst);
                else enqueue(readyq, cur_wait->pcb.idx, cur_wait->pcb.cpu_burst, cur_wait->pcb.io_burst);
                arrival_time[cur_wait->pcb.idx] = RUN_TIME - run_time;
                service_time[cur_wait->pcb.idx] = 0;
                end_time[cur_wait->pcb.idx] = 0;
        }
        else enqueue(waitq, cur_wait->pcb.idx, cur_wait->pcb.cpu_burst, cur_wait->pcb.io_burst);
}
if (set_scheduler == 2) printHeap(readyq, 'r');
else printQueue(readyq, 'r');
printQueue(waitq, 'w');
printf("----------------------------------------\n");

dump_data(fp);

if (cur_ready->pcb.idx != -1) kill(cpid[cur_ready->pcb.idx], SIGCONT);

if (run_time != 0) {
        run_time--;
}
```

Figure 30 - Signal for Count which is Executed for Every Time Tick

Figure 30 presents the implemented code for the count signal, which is called at every time tick. For every call of the count signal, it performs the following sequence of operations:

1. Check if the current node, which is the front node of the ready queue, is done processing its CPU burst time. If so, the function will record the scheduling criteria, which are the number of completed processes, end time, wait time, and turnaround time.

2. Then, the function will decrease the I/O burst time of every node in the waiting queue. If there is a node with an I/O burst time of zero, the following node is enqueued in the ready queue.

3. After the following progress, the function will dump all the logs and the results on the screen and in the dump file.

By processing the following sequence of operations by calling the count signal, we can simulate the CPU scheduling with the proper scheduling policies.

```
// set the timer for SIGALRM
struct itimerval new_itimer;
struct itimerval old_itimer;

new_itimer.it_interval.tv_sec = 0;
new_itimer.it_interval.tv_usec = TIME_TICK;
new_itimer.it_value.tv_sec = 1;
new_itimer.it_value.tv_usec = 0;

struct sigaction io;
struct sigaction count;
struct sigaction scheduler;

memset(&io, 0, sizeof(io));
memset(&count, 0, sizeof(count));
memset(&scheduler, 0, sizeof(scheduler));

io.sa_handler = &signal_io;
count.sa_handler = &signal_count;

switch (set_scheduler) {
case 1: scheduler.sa_handler = &signal_fcfs; break;
case 2: scheduler.sa_handler = &signal_sjf; break;
case 3: scheduler.sa_handler = &signal_rr; break;
default:
        perror("scheduler");
        exit(EXIT_FAILURE);
}

sigaction(SIGUSR2, &io, NULL);
sigaction(SIGALRM, &count, NULL);
sigaction(SIGUSR1, &scheduler, NULL);
```

Figure 31 - Main Function Part 1

Figure 31 shows the first substantial implemented code of the main function. It presents the initialization of the signals and the timer set of the SIGALRM signal. First, we set the timer for the SIGALRM to 0.001 seconds (10 milliseconds) of timer interval and set the timer to start after the 1 second of the program execution. Next, we initialized the signals for I/O, count, and CPU scheduler, which are presented previously. Then, by using the sigaction function, we generated all three signals, which are called for every timer interval.

```
// create the nodes and the queues for the scheduling operation
waitq = createQueue();
cur_wait = createNode();
cur_ready = createNode();
if (set_scheduler == 2) readyq = createHeap(MAX_PROCESS); else readyq = createQueue();
if (waitq == NULL || readyq == NULL || cur_wait == NULL || cur_ready == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
}

// create the message queues for each child process.
for (int i = 0; i < MAX_PROCESS; i++) {
        key[i] = 0x12345 * (i + 1);
        msgctl(msgget(key[i], IPC_CREAT | 0666), IPC_RMID, NULL);
}
```

Figure 32 - Main Function Part 2

Figure 32 shows the second substantial implemented code of the main function. It presents the initialization of the ready queue, waiting for the queue, current wait node, current ready node, and the messages queues that are used between the parent process and child processes. First, by using the functions that we implemented previously, we created a ready queue, waiting queue, current wait node, and current ready node. Then we check if there is an error while generating the following structures. If so, the program prints out the error statement and terminates. After the following progress, the program generates the key, that is used to generate the message queue, for each child process. By using the generated keys, the program set the following keys as the identifiers for the message queues.

```
// creates the child process and operates the processes.
for (int i = 0; i < MAX_PROCESS; i++) {
        pid_t pid;

        // error on the fork operation.
        if ((pid = fork()) < 0) {
                perror("fork");
                exit(EXIT_FAILURE);
        }
```

```
        // child process
        else if (pid == 0) {
                int idx = i;
                int c_io = io_burst[i];
                int c_cpu = cpu_burst[i];

                // child process waits until the interrupt occurs.
                kill(getpid(), SIGSTOP);

                // child operates the cpu burst.
                while (1) {
                        c_cpu--; // decreases the cpu burst by 1.i
                        if (c_cpu == 0) {
                                c_cpu = cpu_burst[i];

                                // send the PCB data of the child process to the parent process.
                                cmsgsnd(key[idx], c_cpu, c_io);
                                c_io = io_burst[i];
                                kill(ppid, SIGUSR2);
                        }
                        else { // cpu burst is over.
                                kill(ppid, SIGUSR1);
                        }

                        // child proces waits for the next interrupt.
                        kill(getpid(), SIGSTOP);
                }
        }

        else {
                cpid[i] = pid;
                arrival_time[i] = RUN_TIME - run_time;
                if (set_scheduler == 2) insertHeap(readyq, i, cpu_burst[i], io_burst[i]);
                else enqueue(readyq, i, cpu_burst[i], io_burst[i]);
        }
}
```

Figure 33 - Main Function Part 3

Figure 33 presents the third substantial implemented code of the main function. In this part, it controls the operations of the processes due to whether the process is a parent or child process. If the following process is a child process, it waits until the interruption occurs. If an interruption has occurred, the child's process starts to consume the CPU burst time. While consuming the CPU burst time, as it consumed all of it, it sends the message to the message queue that the process has consumed all its allocated CPU burst time. If the following process is the parent process, it initially generates ten process control blocks (PCB) to store the information of the child processes and enqueues all of them into the queue or inserts all of them into the heap. If the following process is not either the child process or the parent process, it prints out the error statement on the screen and terminates.

6 Build Environment

- Build Environment:
  1. Linux Environment -> Vi editor, GCC Complier
  2. Program is built by using the Makefile.

- Build Command:
  1. $make main -> build the execution program for simple scheduler.
  2. $make clean -> clean all the object files that consists of the simple scheduler programs.

- Execution Command:
  1. ./main 1 {$max_limit}
     -> Execute the simple scheduler program with the scheduler of FCFS policy with the maximum CPU and I/O burst time of {$max_limit} of the child processes.
  2. ./main 2 {$max_limit}
     -> Execute the simple scheduler program with the scheduler of SJF policy with the maximum CPU and I/O burst time of {$max_limit} of the child processes.
  3. ./main 3 {$max_limit}
     -> Execute the simple scheduler program with the scheduler of RR policy with the maximum CPU and I/O burst time of {$max_limit} of the child processes.

7 Results

- Result of the simple scheduler with FCFS policy and 10 maximum burst time:

```
----------------------------------------
Process 0       -> CPU: 3,      IO:     6
Process 1       -> CPU: 3,      IO:     1
Process 2       -> CPU: 10,     IO:     3
Process 3       -> CPU: 4,      IO:     7
Process 4       -> CPU: 10,     IO:     9
Process 5       -> CPU: 7,      IO:     3
Process 6       -> CPU: 3,      IO:     3
Process 7       -> CPU: 2,      IO:     5
Process 8       -> CPU: 10,     IO:     2
Process 9       -> CPU: 4,      IO:     5
----------------------------------------

----------------------------------------
Time: 0
CPU Process: 0 -> 3
Ready: 1 2 3 4 5 6 7 8 9
Wait:
----------------------------------------
----------------------------------------
Time: 1
CPU Process: 0 -> 2
Ready: 1 2 3 4 5 6 7 8 9
Wait:
----------------------------------------
----------------------------------------
Time: 2
CPU Process: 0 -> 1
Ready: 1 2 3 4 5 6 7 8 9
Wait:
----------------------------------------
----------------------------------------
Time: 3
CPU Process: 1 -> 3
I/O Process: 0 -> 5
Ready: 2 3 4 5 6 7 8 9
Wait: 0
----------------------------------------
----------------------------------------
Time: 9996
CPU Process: 4 -> 2
Ready: 5 6 7 8 1 9 0 2 3
Wait:
----------------------------------------
----------------------------------------
Time: 9997
CPU Process: 4 -> 1
Ready: 5 6 7 8 1 9 0 2 3
Wait:
----------------------------------------
----------------------------------------
Time: 9998
CPU Process: 5 -> 7
I/O Process: 4 -> 8
Ready: 6 7 8 1 9 0 2 3
Wait: 4
----------------------------------------
----------------------------------------
Time: 9999
CPU Process: 5 -> 6
I/O Process: 4 -> 7
Ready: 6 7 8 1 9 0 2 3
Wait: 4
----------------------------------------
----------------------------------------
Time: 10000
CPU Process: 5 -> 5
I/O Process: 4 -> 6
Ready: 6 7 8 1 9 0 2 3
Wait: 4
----------------------------------------
----------------------------------------
Result
Troughput: 0.178
Average Wait Time: 51.472
Average Turnaround Time: 45.871
----------------------------------------
```

- Result of the simple scheduler with SJF policy and 10 maximum burst time:

```
----------------------------------------
Process 0       -> CPU: 1,      IO:     9
Process 1       -> CPU: 5,      IO:     7
Process 2       -> CPU: 3,      IO:     3
Process 3       -> CPU: 6,      IO:     2
Process 4       -> CPU: 2,      IO:     10
Process 5       -> CPU: 2,      IO:     4
Process 6       -> CPU: 4,      IO:     2
Process 7       -> CPU: 3,      IO:     5
Process 8       -> CPU: 5,      IO:     4
Process 9       -> CPU: 9,      IO:     5
----------------------------------------

----------------------------------------
Time: 0
CPU Process: 0 -> 1
Ready: 4 7 5 8 1 2 6 3 9
Wait:
----------------------------------------
----------------------------------------
Time: 1
CPU Process: 4 -> 2
I/O Process: 0 -> 8
Ready: 5 7 2 8 1 9 6 3
Wait: 0
----------------------------------------
----------------------------------------
Time: 2
CPU Process: 4 -> 1
I/O Process: 0 -> 7
Ready: 5 7 2 8 1 9 6 3
Wait: 0
----------------------------------------
----------------------------------------
Time: 3
CPU Process: 5 -> 2
I/O Process: 0 -> 6
I/O Process: 4 -> 9
Ready: 7 8 2 3 1 9 6
Wait: 0 4
----------------------------------------
----------------------------------------
Time: 9997
CPU Process: 5 -> 1
I/O Process: 0 -> 2
I/O Process: 4 -> 5
I/O Process: 7 -> 3
Ready: 2 8 6 3 1 9
Wait: 0 4 7
----------------------------------------
----------------------------------------
Time: 9998
CPU Process: 2 -> 3
I/O Process: 0 -> 1
I/O Process: 4 -> 4
I/O Process: 7 -> 2
I/O Process: 5 -> 3
Ready: 6 8 9 3 1
Wait: 0 4 7 5
----------------------------------------
----------------------------------------
Time: 9999
CPU Process: 2 -> 2
I/O Process: 4 -> 3
I/O Process: 7 -> 1
I/O Process: 5 -> 2
Ready: 0 8 6 3 1 9
Wait: 4 7 5
----------------------------------------
----------------------------------------
Time: 10000
CPU Process: 2 -> 1
I/O Process: 4 -> 2
I/O Process: 5 -> 1
Ready: 0 8 7 3 1 9 6
Wait: 4 5
----------------------------------------
----------------------------------------
Result
Troughput: 0.463
Average Wait Time: 4.880
Average Turnaround Time: 2.720
----------------------------------------
```

- Result of the simple scheduler with RR policy and 10 maximum burst time:

```
----------------------------------------
Process 0       -> CPU: 8,      IO:     8
Process 1       -> CPU: 5,      IO:     4
Process 2       -> CPU: 6,      IO:     6
Process 3       -> CPU: 1,      IO:     1
Process 4       -> CPU: 10,     IO:     9
Process 5       -> CPU: 5,      IO:     7
Process 6       -> CPU: 7,      IO:     3
Process 7       -> CPU: 9,      IO:     9
Process 8       -> CPU: 1,      IO:     4
Process 9       -> CPU: 3,      IO:     6
----------------------------------------

----------------------------------------
Time: 0
CPU Process: 0 -> 8
Ready: 1 2 3 4 5 6 7 8 9
Wait:
----------------------------------------
----------------------------------------
Time: 1
CPU Process: 0 -> 7
Ready: 1 2 3 4 5 6 7 8 9
Wait:
----------------------------------------
----------------------------------------
Time: 2
CPU Process: 0 -> 6
Ready: 1 2 3 4 5 6 7 8 9
Wait:
----------------------------------------
----------------------------------------
Time: 3
CPU Process: 0 -> 5
Ready: 1 2 3 4 5 6 7 8 9
Wait:
----------------------------------------
----------------------------------------
Time: 9996
CPU Process: 2 -> 3
Ready: 4 0 3 7 5 9 1 6 8
Wait:
----------------------------------------
----------------------------------------
Time: 9997
CPU Process: 2 -> 2
Ready: 4 0 3 7 5 9 1 6 8
Wait:
----------------------------------------
----------------------------------------
Time: 9998
CPU Process: 4 -> 10
Ready: 0 3 7 5 9 1 6 8 2
Wait:
----------------------------------------
----------------------------------------
Time: 9999
CPU Process: 4 -> 9
Ready: 0 3 7 5 9 1 6 8 2
Wait:
----------------------------------------
----------------------------------------
Time: 10000
CPU Process: 4 -> 8
Ready: 0 3 7 5 9 1 6 8 2
Wait:
----------------------------------------
----------------------------------------
Result
Troughput: 0.217
Average Wait Time: 40.811
Average Turnaround Time: 36.206
----------------------------------------
```

8 Evaluation

```
----------------------------------------
Process 0       -> CPU: 7,      IO:     4
Process 1       -> CPU: 6,      IO:     8
Process 2       -> CPU: 6,      IO:     4
Process 3       -> CPU: 3,      IO:     7
Process 4       -> CPU: 2,      IO:     10
Process 5       -> CPU: 8,      IO:     3
Process 6       -> CPU: 10,     IO:     1
Process 7       -> CPU: 7,      IO:     4
Process 8       -> CPU: 7,      IO:     1
Process 9       -> CPU: 7,      IO:     3
----------------------------------------
```

Figure 34 - Generated Job Processes List

```
----------------------------------------
Result
Throughput: 0.159
Average Wait Time: 58.331
Average Turnaround Time: 52.034
----------------------------------------
```

Figure 35 - Result of the Scheduling Done by FCFS Policy

```
----------------------------------------
Result
Throughput: 0.172
Average Wait Time: 52.879
Average Turnaround Time: 47.086
----------------------------------------
```

Figure 36 - Result of the Scheduling Done by RR Policy

```
----------------------------------------
Result
Throughput: 0.235
Average Wait Time: 9.750
Average Turnaround Time: 5.500
----------------------------------------
```

Figure 37 - Result of the Scheduling Done by SJF Policy

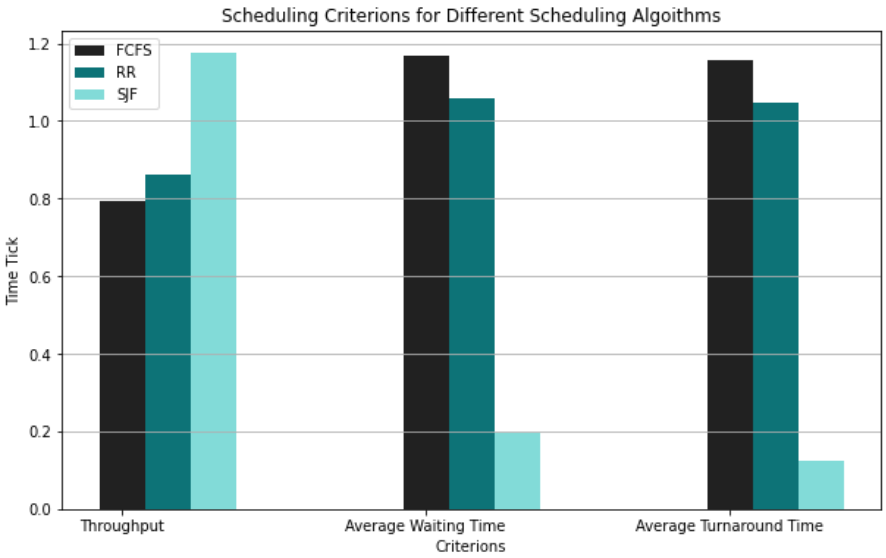

Figure 38 - Representation of the Normalized Scheduling Criterions for Different Scheduling Algorithms

Figures 35, 36, and 37 show the results of the scheduling through the different scheduling policies, such as first-come, first-served (FCFS), round robin (RR), and shortest job first (SJF) policies, that scheduled the process set that is presented in Figure 34.

The three main scheduling criteria that we are going to compare are throughput, average waiting time, and average turnaround time. The first is throughput, which stands for the number of completed processes per time tick. From the Figures, we can see that the SJF shows the best throughput, while the next is RR, and the last is FCFS. Next is average waiting time, which stands for the time that the process waited in the ready queue. From the Figures, we can see that the average waiting time is much lower in the SJF policy compared to the RR and FCFS policies. Between RR and FCFS, the RR policy shows a slightly lower average waiting time than the FCFS policy. The last is the average turnaround time. Still, the SJF shows a significant difference in the average turnaround time compared to the RR and SJF policies. Between RR and SJF policies, RR shows a lower average turnaround time than the SJF policy. From the following results, we can see that the performance of the scheduling algorithm has the order of SJF policy, RR policy, and FCFS policy.

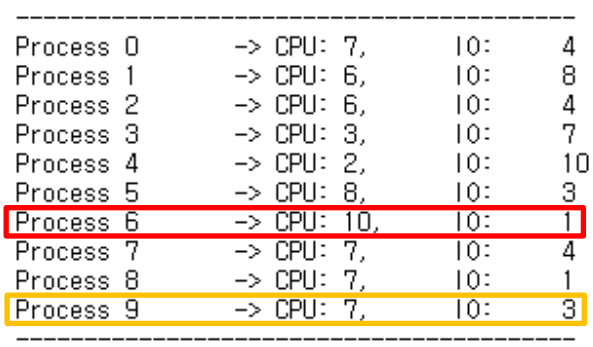

Figure 39 - Generated Job Process List with the Jobs that have Longest CPU Burst Time

However, there is a logical fallacy to the scheduler that uses the SJF policy. From Figure 39, we can see the process with the red rectangular that has the longest CPU burst time, and the process with the orange rectangular that has the second longest CPU burst time with the highest process index. Since the SJF policy allocates the CPU resources to the process that has the shortest CPU burst time, the following processes with the rectangular may not have the chance to be executed. Also, the dump file of the SJF policy scheduler shows the following situation: As a result, the SJF policy is not the ideal scheduling policy due to the starvation of the processes that have long CPU burst times.

9 Conclusion

The basis of multi-programmed operating systems is CPU scheduling. The operating system increases computer productivity by distributing the CPU between the many processes. This means that the goal of multi-programming is to have some processes always executing, to maximize CPU utilization. While some processes are kept in the memory at one time, when a process must wait, the operating system takes the CPU resources away from that process and gives the CPU resources to another process. The CPU scheduler is the function that processes the following progress continuously.

To implement the simple scheduler program that performs the following operation, we first explained the concepts that are used in it. We explained what a process is, the state of the process, process control block (PCB), process scheduling, scheduling queues, CPU scheduler, inter-process communication (IPC), message passing IPC, CPU scheduling criteria, and scheduling algorithms. Next, we introduced the system calls that are used in the simple scheduler program, such as signal and handler, and System V message. Also, we introduced the scheduling algorithms that are used in the simple scheduler, such as first-come, first-served (FCFS), round-robin (RR), and shortest-job first (SJF) policies. Then, we discussed our implementation code for a simple scheduler, which contains the code of the queue and heap header, the IPC message passing header, and the main function. At the end of the report, we presented results for the execution of simple scheduler programs with different scheduling policies. We presented scheduling criteria for throughput, average wait times, and average turnaround times. According to these criteria, we found out that the simple scheduler with SJF policy shows the best performance, while the RR policy showed better performance than the FCFS policy. However, due to starvation in the SJF policy, we concluded that it is not the best scheduling algorithm.

By understanding this paper, we can understand the basic concepts of CPU scheduling and the scheduling algorithms used in it. Also, we can understand the system calls for process communication and the signals that are used to control the process. In short, we get the knowledge of concepts and functions that are needed to implement the CPU scheduler.

Citations

[1] (1966, September 1). Can I Trap Sigsegv (on a linux) and what are are the conditions to make it works? (for a crackme). Reverse Engineering Stack Exchange. Retrieved November 17, 2022, from https://reverseengineering.stackexchange.com/questions/21597/can-i-trap-sigsegv-on-a-linux-and-what-are-are-the-conditions-to-make-it-works

[2] Message queues. Tutorials Point. (n.d.). Retrieved November 17, 2022, from https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_message_queues.htm

[3] MSGCTL(2) - linux manual page. (n.d.). Retrieved November 17, 2022, from https://man7.org/linux/man-pages/man2/msgctl.2.html

[4] Msgget(2) - linux manual page. (n.d.). Retrieved November 17, 2022, from https://man7.org/linux/man-pages/man2/msgget.2.html

[5] MSGSND(3P) - linux manual page. (n.d.). Retrieved November 17, 2022, from https://man7.org/linux/man-pages/man3/msgsnd.3p.html

[6] Program for FCFS CPU scheduling: Set 1. GeeksforGeeks. (2022, September 19). Retrieved November 17, 2022, from https://www.geeksforgeeks.org/program-for-fcfs-cpu-scheduling-set-1/

[7] Relation between preemptive priority and round robin scheduling algorithm. GeeksforGeeks. (2020, October 1). Retrieved November 17, 2022, from https://www.geeksforgeeks.org/relation-between-preemptive-priority-and-round-robin-scheduling-algorithm/

[8] Silberschatz, A., Galvin, P. B., & Gagne, G. (2014). 2.2.1 Command Interpreters. In Operating Systems Concepts. essay, Wiley.

[9] SIGACTION(2) - linux manual page. (n.d.). Retrieved November 17, 2022, from https://man7.org/linux/man-pages/man2/sigaction.2.html

[10] Program for shortest job first (or SJF) CPU scheduling: Set 1 (non- preemptive). GeeksforGeeks. (2022, November 9). Retrieved November 17, 2022, from https://www.geeksforgeeks.org/program-for-shortest-job-first-or-sjf-cpu-scheduling-set-1-non-preemptive/

[11] Relation between preemptive priority and round robin scheduling algorithm. GeeksforGeeks. (2020, October 1). Retrieved November 17, 2022, from https://www.geeksforgeeks.org/relation-between-preemptive-priority-and-round-robin-scheduling-algorithm/