ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 3, March 2015

Simulation and Performance Evaluation of CPU Scheduling Algorithms

Sultan Almakdi[1], Mohammed Aleisa[2], Mohammed Alshehri[3]

Lecturer, College of Computer Science and Information System, Najran University, Najran, KSA [1,3]

Lecturer, College of Science and Humanities at Alghat, Almajmaah University, Alghat, KSA [2]

Abstract: An operating system provides numerous functions, such as I/O management, memory management, process management, and file management. Since the operating system is a set of the programs that interacts with computer hardware during executing time, process management is the most important function provided by an operating system. CPU scheduling is extremely necessary, as it makes a multi-tasking environment that keeps the CPU and I/O devices busy at all times which results in increased CPU utilization [1]. However, numerous scheduling algorithms have already been designed to regulate the access of threads and processes to the CPU, such as FCFS-SJF-SRT-RR. We simulated these scheduling algorithms and evaluated their performance (throughput, latency, utilization, turnaround time, and waiting time) in a multi-processor environment.

Keywords: OS, CPU scheduling, Multitasking, FCFS, RR, SRT, SJF.

I. INTRODUCTION

The major purpose of multiprogramming is to have more than one process running at all times. All computer resources need to be scheduled before using them; thus, scheduling is a main function provided by the operating system [1]. Since the CPU is one of a computer's main resources, its scheduling technique is fundamental to operation system design. When we have more than one process that can be run, CPU scheduling determines which process will be run.

As a result, resource utilization and overall system performance will be affected by CPU scheduling, which is very important [2]. There are four types of scheduling involved in a multitasking system, with each solving a scheduling problem for each area of operating system functionality:thelong-term scheduler, mid-term scheduler, and short-term scheduler.

A. Long-Term Scheduler

Thelong-term scheduler, or admission scheduler, is used to decide which process ought to be brought to the ready queue. When we have a process attempting to be executed, the long-term scheduler decides whether to admit or delay this process [1] [2].

B. Mid-Term Scheduler

The mid-term scheduler is used to temporarily remove processes from the main memory and put them onto secondary memory and vice versa. This can be generally referred to as "swapping processes out" or "swapping processes in." [1] [2].

C. Short-Term Scheduler

The short-term scheduler is used to decide which processes in the ready queue—in the memory—are to be executed (allocated to a CPU) next, following a clock interrupt, an input–output (I/O) interrupt, an OS call, or another form of signal. Making scheduling decisions is more frequent for the short-term scheduler than for the long-term and mid-term schedulers. The short-term scheduler can be either preemptive or non-preemptive. A preemptive scheduler forces any process to exit the CPU when it decides to allocate another process to the CPU. A non-preemptive scheduler cannot force any process to exit the CPU [1].

The design of a high-quality scheduling algorithm plays a role in the success of a CPU scheduler. In addition, high-quality CPU scheduling algorithms generally rely on criteria such as throughput, CPU utilization rate, response time, turnaround time, and waiting time. Therefore, the main impetus of this work is to design a generalized, optimum, high-quality scheduling algorithm suitable for all types of jobs [3].

The main goal is to simulate the different types of CPU scheduling algorithms. Processor scheduling is the foundation of operating systems, and through advanced study and innovation, we have seen major improvements in computing power. In this paper, we will explore how the algorithms compute resource allocations and processing time, as well as their applications in the latest computing innovations. Our key motivation was to use the computing systems to reduce the cost of both the hardware and software, while increasing efficiency at the same time.

Investments on software will increase significantly, both in the development and deployment of software systems, which will create the need to save on hardware investments. Effective use of resources and the elimination of upgrade costsare needed to save on hardware investments. There have been significant advances in the computing power of processors, which today are smaller, cheaper, and more effective than 10 years ago. This advancement can be attributed to a number of factors,

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 3, March 2015

including up-to-date materials, better production methods, and advancement in the understanding of computing resources. Unlike a decade ago, we do not have over-clocked processors or even cases where a processor is overpowered.

This has led to the testing of the available processor scheduling algorithms, in order to determine which is best suited to simulate and measure. Our team faced a number of challenges, in which we could not decide which characteristic would give us the most conclusive results. We wanted our results to be less complicated and easier to implement. We overcame our challenge by starting from the final result we wanted and working backwards, while characterizing and differentiating the simulated algorithms. Last but not least, we wanted to provide a comparative analysis of the different algorithms that we tested and measured.

II. SCHEDULING CRITERIA

Since each CPU scheduling algorithm has its own properties, choosing a particular algorithm might favor one class of processes over another. To choose which algorithm to use in a particular situation, the properties of the various algorithms must be considered. We suggest several criteria to compare the CPU scheduling algorithms [1]:

A. Utilization/Efficiency
The CPU must be kept busy with useful work 100% of the time.

B. Throughput
The number of processes that completed per time unit.

C. Turnaround Time
The timefrom submitting a process to the time of completion.

D. Waiting Time
The sum of the time that processes spend in the ready queue.

E. Response Time
The time from process submission until producing the first response.

F. Fairness:
Whether the processes share the CPU fairly.

An operating system must choose a process from the ready queue to execute whenever the CPU becomes idle. The short-term scheduler (or CPU scheduler) is the scheduler who is responsible to carry out a process to the CPU. The scheduler selects a process among those that reside in the memory, are ready to be executed, and are allocated to the CPU [5].

The ready queue contains the processes that are ready to be executed, but is not necessarily a FIFO (first-in, first-out) queue. It might be designed as priority queue, a FIFO queue, simply an unordered linked list, or a tree. All the processes in the ready queue are lined up and wait to be allocated by the CPU. When multiple processeshave competing requirements, the operating system must allocate computer resources among them. The scheduleris the component of the operating system responsible for granting the CPU access to a list of several processes that are ready to execute.

In the cases of the new and exit states, there is no choice, in terms of scheduling. However, there is a choice in the ready and running cases. Under non-preemptive scheduling, when a process has been allocated to the CPU, the process will not release the CPU until it terminates or switches to the waiting state, such as I/O functions [1] [2].

III. CPU SCHEDULING ALGORITHMS

CPU scheduling is the process of determining which process in the queue to allocate first to the CPU. There are two types of scheduling algorithms: pre-emptive scheduling and non–pre-emptive scheduling, which are divided by how they handle clock interrupts [1] [2].

A. Pre-emptive Scheduling
This type of scheduling is runnable. Once the process has been initiated by the CPU, it can be temporarily suspended for a given period of time. The act of temporary suspending a task or taking it away is what gives it the name pre-emptive scheduling [1].

B. Non–Pre-emptive Scheduling
This process is a clear contrast to pre-emptive scheduling because, once a process has been initiated in the CPU, it cannot be suspended or taken away. This type of scheduling differs from pre-emptive scheduling in a number of ways since the scheduler executes the job when the process terminates or when the process switches from the running to the waiting state. Another difference is that, unlike pre-emptive scheduling, the response times are predictable. The overall treatment of all processes is fair; even a high-priority job cannot displace the waiting job [1].

C. Priority Allocation System
There was a need to divide processes into high-priority and low-priority before introducing them into the scheduler. A priority allocation system allows the scheduler to select a process with high priority, rather than a low-priority one. The process of priority allocation starting with high-priority processes presented a challenge, in which the low-priority processes were starved and lacked time in the processor. A solution for this was to change the priority allocating system, so that priority could change not only according to whether a process was high or low priority, but according to execution history and age. Other ways that are used to assign priorities to processes include [3] [5]:

a. *Internal or Dynamic:* The priorities are assigned according to specific algorithms.
b. *External or Statically:* Priorities are assigned by an external system manager before being scheduled to the processor.
c. *Hybrid:* Priority is assigned by both internal and external schemes.

D. Timer Interruption
Timer interruption is an important process that protects the processes from getting stuck in an infinite loop, which will lead to the system hanging. Timer interruption is both system- and process-dependent, and is seen as a real-time

DOI 10.17148/IJARCCE.2015.4301

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 3, March 2015

system. When a process is initiated in the CPU, the timer begins and counts in an interval. The time interval can expire; when it does, the process completes in the CPU. A new process can be installed in the processor with the scheduling decision.

A content switch is an installation process that involves switching the processor's context. The CPU carries a number of tasks, which include registering all saved and loaded processes, preparing the processes, and running the processes, once they are ready. The registered files are stored in the Process Control Block (PCB). The PCB has all of the information about the processes, and each process has a PCB associated with it. The PCB is vital, since, other than being an important data structure, it contains process stacks, process control information, process register values, and process priority and process identification information. The system experiences a change in processes from one to another to prevent system overload and poor performance, while maintaining the processes at a minimum [1] [5].

IV. SIMULATED CPU SCHEDULING ALGORITHMS

There are different types of CPU scheduling algorithms, including: first-come, first-served, priority-based scheduling, round robin, and shortest job first. Below is a detailed look into each type of scheduling algorithm.

A. First-Come, First-Served (FCFS)

Of all the scheduling algorithms, this is the simplest and most effective policy. The new process usually enters the end of the schedule and goes on until it is complete, before moving to the next queue. The FCFS goes by many names, which are worth noting to avoid confusion: cycling scheduling, first in, first out (FIFO), non-priority non-preemptive FCFS, priority preemptive FCFS (PP-FCFS, and priority non-preemptive FCFS (PNP-FCFS). These priorities are mostly used in timeshared systems. Once the process starts in the CPU, it holds until completion or when the CPU is yielded. Like any other types of scheduling, the FCFS has its own set of challenges, as the system can get stuck when there is a heavy workload that monopolizes the CPU. Due to this, there can be a lot of wasted time, since the process is made to wait at the end of the queue and might not get time in the CPU. This challenge has been introduced addressed by introducing a timer interruption, which limits the time of a given process [3] [5].

B. Round Robin (RR)

This is an improved version of the FCFS where arriving processes are queued as circular and placed at the end of the queue. The scheduler then selects the jobs that are first and runs them in the CPU until they are completed. During the process, if the time interval expires, that specific process is placed at the end of the queue. The pros of using the round-robin method is that it is simple; the cons are that a lot of time is wasted if the job is too large and that the quantum too small [4].

C. Shortest Job First: (SJF)

As the name suggests, this method of scheduling chooses its priority based on which process has the smallest CPU time requirement. The dispatcher selects the shortest jobs in the queue and runs them to completion. One of its advantages is that it has a quick turnaround time; on the other hand starvation can occur, plus it cannot be implemented [1].

D. Shortest Remaining Time (SRT)

This method of scheduling is similar to that of SJF, in the sense that the scheduler will pick the processes that have the shortest remaining time and move them in front of the queue. When the CPU is running a process and another process that is even shorter arrives, a preemption occurs, which is an interruption. This leads to the division of the process into two parts, thus creating additional context switching. The additional overhead created leads to an increase in both the waiting and response times. Longer processes in the queue are affected significantly by the process, which makes it hard to maintain a deadline. SRT experiences starvation when the CPU is running multiple small processes. Due to the different challenges involved in this policy, it is not widely used; however, those who still want to use it will need to use two different priorities [3] [5].

V. SYSTEM DESIGN

We used C++ for our simulations because it allows us to standardize the interface, use a common base, and utilize our algorithm in a polymorphic way. We did not pay much attention to the environment in which we ran the simulation, since it was not affected by the operating system or our platform of choice. We agreed to use a personal laptop, which offered flexibility and made it easier to gather results from each team member.

In the simulations, we accounted for variables like core active time, number of cores, process waiting time, core active time, and simulation duration. We simulated the cores as a vector of process objects; we also inserted a specialized idle process together with the vector, which we simulated as an idle core. The simulation was for scheduling algorithms, which are referred to as schedulers in the source code. We wanted to determine which process we could execute next and what the best scheduling algorithm was. The algorithms we selected from the common base class included: shortest-job-first (SJF), shortest remaining time (SRT), first-come, first-served (FCFS), and round robin (RR).

The basic schedulable unit for our design was that of a process determined by its own lightweight class. We excluded the priority-oriented scheduling algorithm, since it only contributed performance drawbacks. We chose the double-ended queue collection for the waiting and completed queues in our design. We wanted to allow the sorting of the different queues, in order to make our presentation easier and also to optimize the scheduler implementations. In this simulator, tasks or jobs can be assigned to the simulator by generating an arrival time and burst time for each task randomly; optionally, the user can also assign tasks. The number of cores and number of tasks are identified by the user.

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 3, March 2015

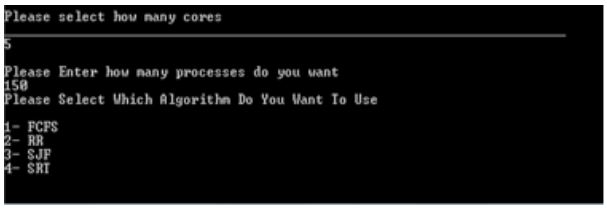

Fig.1. Simulator Main Interface

The above figure shows the main interface for our simulator. A user will be asked to enter the number of cores to use in the simulation; then, the user will be able to enter any number of tasks or processes. Afterwards, the user can select an algorithm by entering 1 for FCFS, 2 for RR, 3 for SJF, or 4 for SRT. The user then has the ability to enter the burst time and arrival time for each process or let the simulator generate them. Then, the simulation process will start by pressing the ENTER key on the keyboard. The simulator can track the time until the simulation is completed. Finally, the results will be shown (see Figure 2).

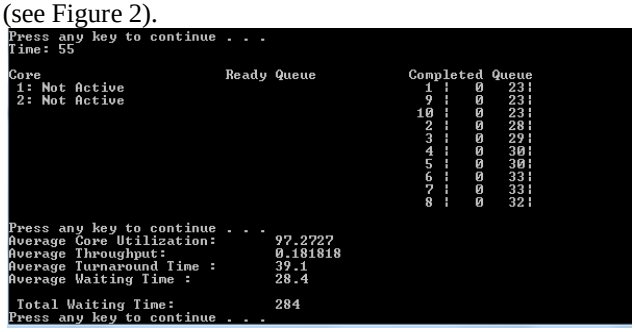

Fig. 2. Results

VI. ANALYSIS, EVALUATION, AND FINAL RESULTS

All of the measurements we collected from the various simulations we ran were in the form of a set of variables, core active time, process waiting time, process length, simulation duration, and number of cores. The collected variables were used to evaluate the scheduling algorithms. We derived the statistical outcomes from the data we had gathered from the variables. The scheduling criteria we used included:

A. Average Waiting Time

The time the process took while waiting to execute.

B. Average Throughput

The number of processes completed successfully per time unit.

C. Average Core Utilization

The percentage of time the core remained active per total simulation time.

D. Average Turnaround Time

It is a taken time to complete a process.

In the simulation, we left out the I/O bursting and its intended effects, since we wanted to focus on the aspects that affected CPU scheduling. We could also have used average response time for measurement, but we disregarded this approach because it is more specific to individual process instructions and thus could not fulfill out intended goal. All we wanted to do is distinguish the processes by their arrival time and length.

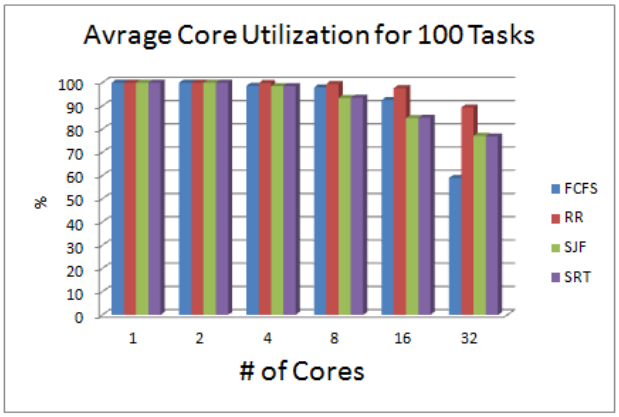

Fig. 3. Avg core utilization for 100 tasks

As shown in the figure above, 100 tasks were tested with different numbers of cores (i.e., 1, 2, 4, 8, 16, and 32 cores) using simulated scheduling algorithms (FCFS, RR, SJF, and SRT). When the number of cores was increased, the average core utilization decreased. However, the RR algorithm had the highest average core utilization for all numbers of cores. Moreover, as shown on the figure, SJF and SRT are close to each other.

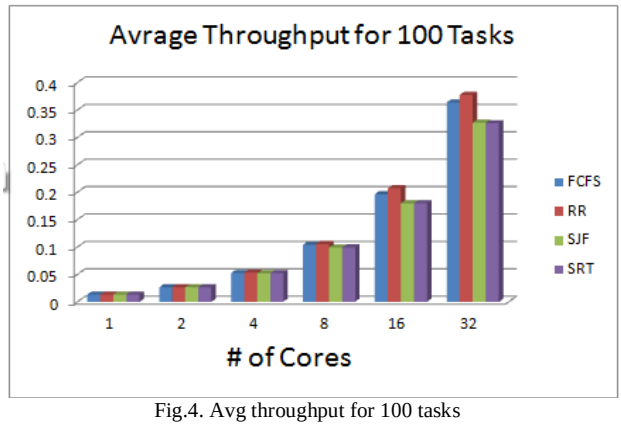

Fig.4. Avg throughput for 100 tasks

This figure illustrates that, as the number of cores increased, the average throughput (for 100 tasks) also increased. Here, the average throughput was almost same when using 1, 2, 4, or 8 cores, but there was a change in the average throughput percentage with 16 and 32 cores. However, SJF and SRT are still close to each other.

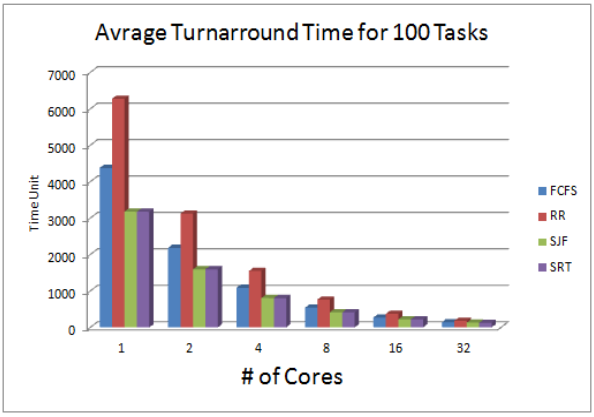

Fig.5. Avg turnaround time for 100 tasks

DOI 10.17148/IJARCCE.2015.4301

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 3, March 2015

In this figure, RR holds the highest average turnaround time, compared to the other algorithms, at different numbers of cores. The average turnaround time went down as the number of cores increased see figure [5].

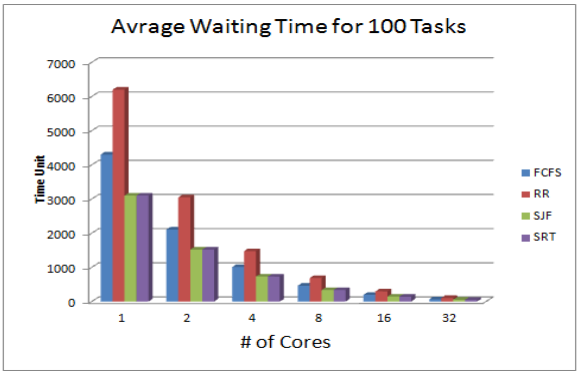

Fig.6. Avg waiting time for 100 tasks

Figure 6 demonstrates that the average waiting time dropped as the number of cores went up. Furthermore, RR had the maximum average waiting time with all cores, while FCFS had the second highest average waiting time. The other algorithms (SJF, SRT) had almost the same performance.

In the following figures, we evaluate our algorithms with different numbers of tasks using 8 cores:

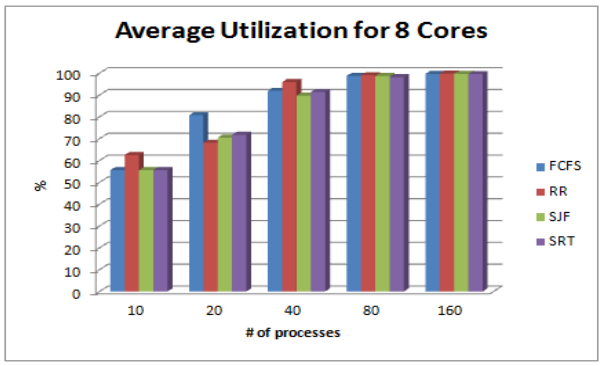

Fig.7. Avg utilization for 8 Cores

The average utilization was less than 62% in the case of only 10 tasks. However, when increasing the number of tasks, the curve of average utilization also increased for all algorithms.

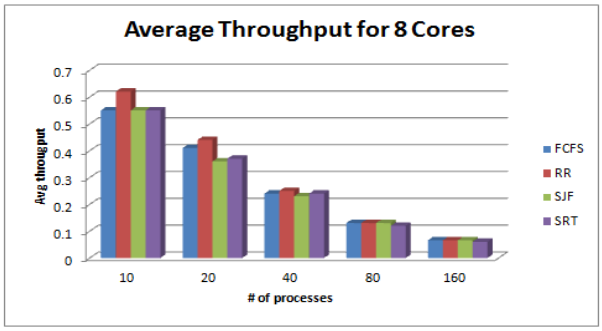

Fig.8. Avg throughput for 8 Cores

The average throughput decreased when we had more tasks to execute. When executing 80 tasks or more, the average throughput for different scheduling algorithms converged, and their percentage was minimized.

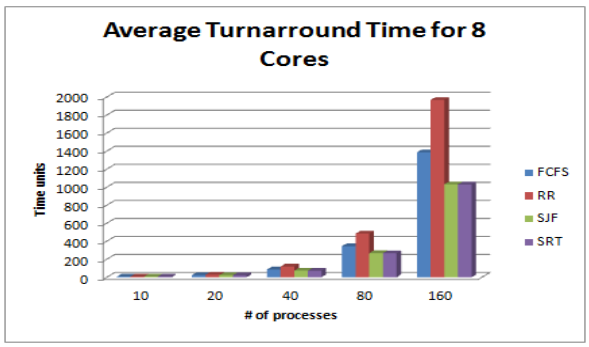

Fig.9. Avg turnaround time for 8 Cores

With 10, 20, and 40 tasks, the average turnaround time was below 150 time units, but RR reached about 420 time units with 80 tasks, while the others were below 350 time units. There was a large jump when running 160 tasks; RR took more than 1,900 time units, while SJF and SRT had the lowest average waiting times in this case.

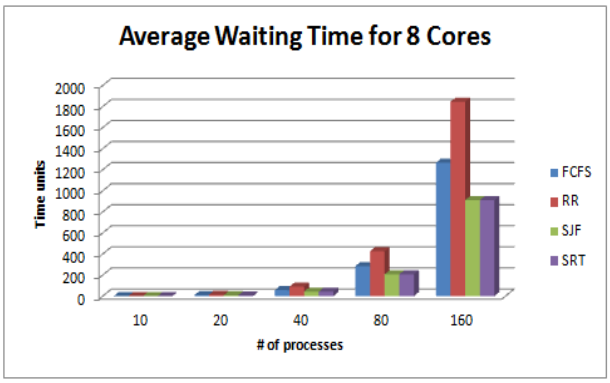

Fig.10. Avg waiting time for 8 Cores

With fewer than 20 tasks, the average waiting time was less than 15 time units. However, this average increased with the number of processes or tasks.

After carrying out the number of simulation, we were able to draw the following conclusions:

-The shortest remaining time algorithm demonstrated the lowest waiting and turnaround times, compared to the other algorithms used.

-The average waiting time and average turnaround time converged as the number of cores increased; the average core utilization and average throughput also converged.

-The round-robin algorithm showed a higher rate of core utilization and throughput than any other algorithm we ran.

-Both the round robin and the first-come, first-served algorithms could perform as a pair, as could the shortest-job-first and the shortest remaining time algorithms could also do the same.

VII. CONCLUSION

In this paper, we have discussed the importance of multiprogramming, and how processes are scheduled using different CPU-Scheduling algorithms. We have also simulated, and evaluated those algorithms in case of multi-core environment. In this paper, we visualized, and compared the performance of CPU scheduling algorithms with different parameters such as number of cores, and total number for processes.

DOI 10.17148/IJARCCE.2015.4301

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 3, March 2015

VIII. FUTURE WORK

There is a need for further study in simulation accounting sleeping processes, I/O bursting, irregular process execution, and other complexities. Moreover, an improved algorithm should be designed to provide more performance efficiency than other algorithms. More variables and criteria also should be used to provide a broader range for comparison. Finally, there is a need for an in-depth study of variation in process generation, which can be implemented to give more conclusive results when scheduling algorithms.

REFERENCES

[1] SILBERSCHATZ, ABRAHAM, PETER GALVIN, and GREG GAGNE. *Operating System Concepts*. 9th. New Jersey: John Wiley and Sons, INC, 2012. Print

[2] Sabrian, F., C.D. Nguyen, S. Jha, D. Platt and F. Safaei, (2005). Processing Resource Scheduling in Programmable Networks. Computer communication, 28:676-687.

[3] U, Saleem, and Javed M. Y. *Simulation of CPU Scheduling Algorithms*. Working paper no. 6893967. Vol. 2. Kuala Lumpur: Dept. of Comput. Eng., Nat.Univ. of Scis. & Technol., Rawalpindi, Pakistan, 2000.

[4] Sun Huajin', Gao Deyuan, Zhang Shengbing, Wang Danghui; " Design fast Round Robin Scheduler In FPGA",0-7803-7547-5/021@2002 IEEE

[5] Sukanya Suranauwarat, "A CPU Scheduling Algorithm Simulator", October 10-13, 2007, Milwaukee, WI 37th ASEE/IEEE Frontiers in Education Conference.