An Analysis and Comparison of Processor Scheduling Techniques

Albert Haque
University of Texas at Austin
ahaque@cs.utexas.edu

December 7, 2012

Abstract

In this paper, we survey various methods of scheduling jobs on both uniprocessor and multiprocessor systems. We explore the different metrics associated with processor performance and provide real-world examples. We compare scheduling methods on a uniprocessor system by assigning both iterative and recursive Fibonacci computation tasks. Simulations are run with a variety of process arrival scenarios and scheduling methods. Our results show that depending on the set of tasks arriving, not one method is superior but a combination of scheduling policies is required to give the best performance.

1 Introduction

Even before the first commercial processor (Intel 4004 in 1971), researchers were already exploring ways to multi-task on hardware [4]. Today, processors are burdened with performing hundreds of task simultaneously. Some jobs require more processing power than others. What if we have a process who does one computation then reads from disk? What should the processor do while the hard drive spins and transfers the data? Should it wait? The processor is a scarce resource and we should maximize its utilization. This is done by processor scheduling.

A processor scheduler orders tasks so it can multi-task. We are able to share the CPU among multiple threads, processes, and programs. While one process is reading from disk, another process can use the CPU to do its computations. When the disk I/O is complete, we can then switch it back to the processor. The time a task uses the CPU between I/O waits is called a CPU burst [14]. We will use CPU bursts as one metric for our analysis.

In this paper, we will explore different scheduling methods on both a uniprocessor and multiprocessor system. This will be done using both a qualitative and quantitative approach. First we will review the concepts of each scheduling method and then we will devise an algorithm to act as the scheduler. We will then test the scheduling policy under a different set of job arrival conditions. Key performance metrics include CPU utilization, process throughput per time unit, and average waiting time. Throughout this paper, the words job, task, and process are used interchangeably.

2 Uniprocessor Systems

2.1 First Come First Serve

FCFS (also known as first in first out, FIFO) is a simple scheduling mechanism. We execute tasks in the order they arrive. This minimizes overhead since each job gets run on the processor only once. However, FCFS has a major weakness. Consider the case where a large matrix

multiplication job M arrives before a short job S. Although S is shorter, it must wait on the queue before M has completed. Thus our average wait time, also known as response time, has increased. Since processes run to completion, there is no context switching.

Despite this weakness, many major websites such as Facebook, Youtube, and Reddit use a FCFS queue to handle certain requests [13]. Take Facebook for example. All their requests are for small amounts of data (we can consider a web request equivalent to a processor job). Since all jobs are small, memcached (the Facebook storage system) responds to jobs in a FCFS order thus minimizing overhead, and in this case, maximizing throughput [3].

2.2 Shortest Job First

SJF places the shortest job at the front of the queue. This is good because we achieve faster response times. In fact, SJF gives the minimum average wait time. Also there is no context switching since jobs run to completion. One major disadvantage of this method: consider our matrix multiplication job M. The author of the job can divide their large job into many smaller, sub-jobs. The author has essentially cheated and M will run before legitimately short jobs.

Additionally, the length of the process is not always known. So how do we know what constitutes a short job? In these instances, it may be necessary to approximate the length of the job. This can be done with a fixed average $\tau_{n+1} = (t_n + t_{n-1} + \cdots + t_1)/n$ or an exponential average $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$ where τ_{n+1} = Predicted Job Length, t_n =Actual Length of the Previous n^{th} Job, and $0 \leq \alpha \leq 1$ [1].

2.3 Round Robin

We now introduce pre-emptive scheduling. Jobs can be interrupted against their will and moved back into the queue. This was not the case for FCFS and SJF where jobs would run to completion. Round robin scheduling is pre-emptive. The main idea is to use a timer. After a specific time quantum, you are placed at the end of a FIFO queue.

Variants of round robin are commonly used in networking such as deficit round-robin scheduling, weighted round-robin scheduling, and weighted fair queuing [6]. However, if we were to impose one of these scheduling policies to a processor jobs, it would introduce a metric of importance, or priority, to a job and thus it would become a multi-level feedback queue.

2.4 Multi-Level Feedback

A multi-level feedback queue, or MFQ, contains multiple queues, each designated as a different priority. We would have a low, medium, and high priority queue. All jobs inside the high priority queue have the same priority level, $p = 3$ for example. All jobs inside the medium queue have $p = 2$ and the low queue has $p = 1$. Arpaci-Dusseau outline a set of rules which can be applied to a MFQ scheduler [2]:

Rule 1: If Priority(A) > Priority(B), A runs (B doesnt).
Rule 2: If Priority(A) = Priority(B), A and B run in round robin.
Rule 3: When a job enters the system, it is placed at the highest priority.
Rule 4a: If a job uses up an entire time quantum, its priority is reduced.
Rule 4b: If a job gives up the CPU before the time quantum is up (e.g. disk I/O), it stays at the same priority level.
Rule 5: After some time period S, move all the jobs in the system to the topmost queue.

It is important to prevent a process from gaming the system. A MFQ attempts to prevent

processes from taking advantage from the rules above. For example, under **Rule 4b**, if our matrix multiplication process requests a disk I/O every 19 ms when the round robin quantum is 20 ms, the process will remain at the highest priority level. This presents a problem. An alternative is to replace Rule 4a and 4b above with:

Rule 4: Demote a process's priority after a specified time quantum regardless of the number of I/O's or the number of times the process gives up the CPU.

Suppose we enforce this policy. Processes who try to game the system by requesting an I/O wait every 19 seconds will not gain additional time on its priority. It will be demoted after 100 ms regardless. Now the question arises: what about legitimate processes who request an I/O operation? Under this new policy, if a process truly requests I/O every 19 ms, it too will be moved down in priority after 100 ms of running time. However, it will not be stuck at the lowest priority level forever. The priority boost (**Rule 5**) will move it to the highest queue eventually [2].

2.5 Real-Time Systems

When deciding the scheduling policy, it is important to take into account the various scenarios that can occur. The Mars Explorer Rover in 1997 suffered a case of priority inversion where a low priority communications task pre-empted a high priority information bus thread [8]. This caused the rover to continuously reboot.

A multi-level feedback queue attempts to include various scenarios into its scheduling policies. MFQ dynamically alters the state of the queue as new processes arrive, finish, and request I/O. FCFS or SJF do not utilize the full suite of information available about the processes – such as a disk I/O or the fact a process is being starved. It is obvious that not one method is perfect for all situations. I recommend using a MFQ. This will prevent starvation and emulate fairness.

We can also implement additional policies to better fit our particular need. In the case of real-time systems, schedulers introduce another process metric – the process deadline. A process will request time on the CPU but will also provide a deadline. This deadline can be in CPU cycles or seconds. The scheduler then assigns higher priorities to those with the earliest deadline. Deadlines are typically decided pre-run-time [19]. This alleviates some of the overhead required with calculating a timeline of events. Before attempting to schedule the real-time processes, we must know if a schedule is even possible for a given set of tasks and deadlines. This is done with a schedulability test. For periodic tasks, such as a video process, we use the following test [9]:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

$$\textit{where } C_i = \text{worst-case computation time for Task } i$$
$$T_i = \text{time between arrivals for Task } i$$

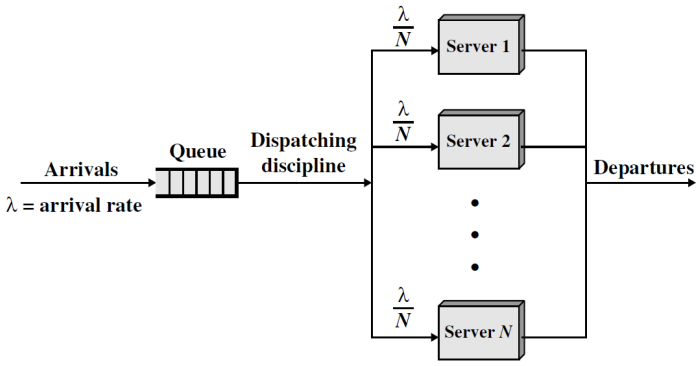
Earliest deadline first scheduling allows for additional information to be communicated between the process and the scheduler but still allows processes to game the system. Our matrix multiplication process can simply break its job into smaller ones and set early deadlines. It is difficult to differentiate between gaming processes and legitimate processes solely on the deadlines they provide. Thus it is recommended to use an earliest deadline first scheduling policy when all processes can be trusted and not in a public computation system.

3 Multiprocessor Systems

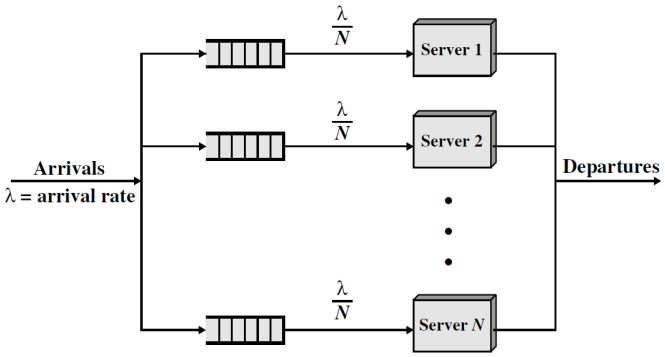
Uniprocessor scheduling techniques can be easily adapted to work on multiprocessors. In this section, we discuss the number of queues feeding the processors, balancing, and communication among the processors. Throughout the discussion, we will assume we have 4 processors.

3.1 Multiple Single-Server Queue

A fairly obvious design is to have a separate queue for each of our processes. This is common in real-life shopping checkouts where a person has the ability to select a queue. An incoming job gets placed into the queue with the least number of jobs. We can improve our definition of "free-est" queue by approximating the job length as outlined in the SJF section. We now have a system which places jobs into a queue where it will wait the least. But how do we distinguish the important jobs from the non-important?



(a) Multiserver queue



(b) Multiple Single-server queues

Figure 1: Multi-Server Queue Versus Multiple Single-Server Queues [16]

Another design is to implement a MFQ for each processor [7]. In **Figure 1b**, assume each queue contains three queues. Take an incoming task of priority 3 for example. The scheduler will look at the four level 3 queues and based on the policies of the MFQ, it will determine the shortest queue and place the task appropriately.

What are the drawbacks of this approach? It is quite possible the scheduler incorrectly predicts the duration of a process. If a process was predicted to run for 50 ms but actually

requires 200 ms, a task waiting behind this process on the queue will be forced to wait. This is where the dynamic nature of a MFQ comes into play. If we do have an extremely long process, which the scheduler thought was small, then MFQ **Rule 4** will allow other processes to make progress. The matrix multiplication task will run in between our smaller tasks. It could have been the case that had the scheduler correctly estimated the length of the matrix task, it may have been run to completion on a different CPU. However, not all systems are perfect and the MFQ attempts to correct mistakes.

Another important concept is load-balancing [18]. If the scheduler notices that CPU 2 is experiencing a large load or long queue for any reason, it can move some tasks from CPU 2's queue to CPU 1. By equally distributing the incoming jobs among the various processors, we improve the utilization of the system.

3.2 Single Multi-Server Queue

One problem with the multiple, single-server queue model is that occasionally, a processor will encounter a very large task. Regardless of the scheduling policy, the average wait time of jobs on that queue will increase. This is equivalent of a person going to the grocery store, picks a line for checkout, and the person currently with the cashier requires additional customer service. The queue will not make progress while this person is being served. This is the same for processor jobs. A matrix multiplication job will hog the processor. How can we minimize scenarios like this?

A solution is to have a single queue, shared by all the processors. By having all processes wait in a single queue, a process is not committed to a processor until we know it will be able to execute immediately. This is not the case for multiple, single-server queues. A distinct advantage of a single queue is that we are reducing the average wait time. Compared to the multiple, single server queues, a single queue is superior [17]. This is assuming that processes are placed into the multiple, single server queues at random.

We can also add MFQ-like policies to our single queue to increase fairness. However, as we enforce more policies, more computations are required and thus performance decreases. We must take care when implementing a scheduler.

4 Methodology

4.1 Test Cases

We create two jobs to run on each scheduling mechanism. This allows us to compare scheduling algorithms under various job ordering and size conditions. We have a Job class which consists of a combination of CPU/wait and start/end times as well as fields to store the job duration, result, and data structures necessary to save the execution state of the job in the event it is pre-empted.

The jobs to execute will be predetermined and given to all the schedulers. The order of the initial job list will represent different arrival times. The moment the scheduler receives the initial job list, the timer begins and the scheduler orders the jobs such that the order best fits the schedule's policies. After this, a final job list will have been created. This final job list is then sent to the processor to begin execution. Once a job comes off the queue, the wait time is calculated and we start the timer for CPU running time. When a result has been computed, we calculate the time spent on the CPU and continue reading from the queue.

4.1.1 Recursive Fibonacci

The task is to compute Fibonacci numbers recursively. A job's duration is determined by the n^{th} Fibonacci number we wish to calculate. For example, calculating the 10^{th} Fibonacci number has a job duration of 10. We give each scheduler 40 Fibonacci numbers to calculate. The initial list of 40 numbers will represent the arrival order of the jobs. Various arrival orders will be tested:
Shortest Jobs Arrive First. {0,1,2,...,39}
Longest Jobs Arrive First. {39,38,37,...,0}
Uniform Distribution With Duplicates. {39,4,25,4,39...}
Uniform Distribution Without Duplicates {9,18,30,23...}
Normal Distribution With Duplicates {18,22,19,17,8,20,38...} $\mu = 20, \sigma = 3$

4.1.2 Iterative Fibonacci

Similar to the recursive Fibonacci test, the duration is the number we wish to calculate. We use the same arrival orders as above.

4.1.3 Code Snippet

The following is code for the iterative Fibonacci test. When a process gets pre-empted, it's current sum is saved and the job is moved back into the queue.

```
1  private long fib(long n) {
2     long start = System.nanoTime();
3     long savePrev1;
4     for (int i = current_i; i < n; i++) {
5        current_i = i;
6        long quanta_diff = System.nanoTime() - start - QUANTA;
7        if(quanta_diff > 0){
8           isPaused = true;
9           return 0;
10       } else {
11       savePrev1 = prev1;
12       prev1 = prev2;
13       prev2 = savePrev1 + prev2;
14       }
15    }
16    isPaused = false;
17    isDone = true;
18    return prev1;
19 }
```

5 Results

Performance metrics are calculated using the following: total wait time w, total CPU time c, total running time t, and number of jobs completed n. Average Wait Time $= w/n$. Throughput $= n/t$. CPU Utilization $= c/t$.

For recursive Fibonacci, only first come first serve (FCFS) and shortest job first (SJF) were tested. This was due to the difficulty of preempting a recursive process. From the data above, it is clear that SJF decreased the average wait time regardless of job arrival order. However, SJF caused CPU utilization to change by +0.44%, -0.28%, -0.22%, -0.54%, and -10.54%. This is due to the added overhead of ordering the jobs such that the shortest jobs are processed first.

Table 1: Recursive Fibonacci

	Wait Time (μs)		Throughput (jobs/sec)		CPU Utilization	
Arrival Type	**FCFS**	**SJF**	**FCFS**	**SJF**	**FCFS**	**SJF**
Shortest Jobs First	2432	671	1764	2501	98.94%	99.38%
Longest Jobs First	11249	633	2431	2583	99.60%	99.32%
Uniform No Duplicates	4829	633	2546	2562	99.64%	99.42%
Uniform With Duplicates	3136	445	5135	5003	99.30%	98.77%
Normal Distribution	87	57	141405	127522	83.27%	72.73%

An iterative Fibonacci calculation method was tested next. The time quantum used for preemption is 10 μs. After a job has been on the processor for 10 μs, it's current progress is saved and it is moved to the end of the queue. The next time the job is allowed on the processor, it will resume from its saved state.

For the multi-level feedback queue, we have three queues for low, normal, and high priority jobs. If a job uses the processor for a full 10 μs, its priority is reduced and it is placed next lower queue. We process jobs from the highest priority queue first. For every 3 jobs that are processed, we process one job from the normal queue. For every 3 jobs that are processed from the normal queue, we process on job from the low level queue. Every 1 millisecond (1000 μs), all jobs have their priority reset to P_{MAX}. This will allow all processes to make progress, even if they are at the lowest priority. This also acts as a second chance for processes who are close to completion, to finish.

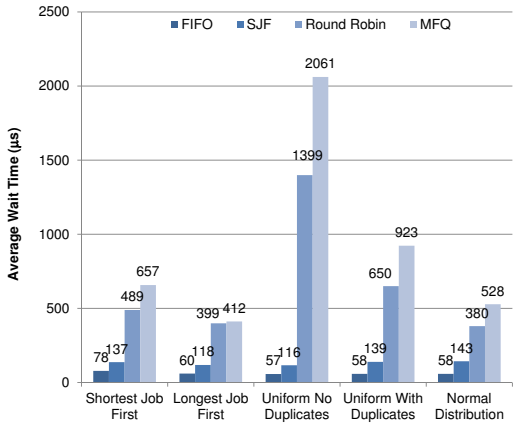


Figure 2: Average Wait Time (μs)

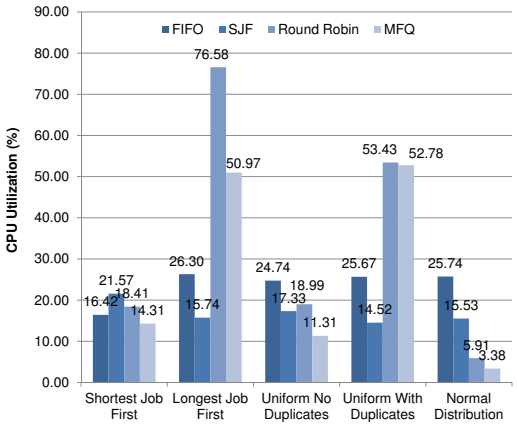


Figure 3: CPU Utilization

The above results indicate that the simplicity of the FCFS and SJF algorithms cause lower average waiting times than round robin and MFQ. It is important to note that a scheduling method may be optimal for one set of tasks but not suitable for another set. In fact, since iterative Fibonacci jobs are so short, the added overhead due to round robin and MFQ do not justify the added overhead. The CPU utilization of MFQ and round robin for a normal distribution of job arrival lengths highlights how FCFS and SJF are superior in this scenario. Because the jobs are so short, a smaller percentage of time is being spent executing the jobs at hand and a larger percentage is spent on housekeeping such as retrieving data from memory (and context switching on a processor).

One mathematical-based comparison using Markov chains indicates that round robin processor scheduling is superior to FCFS [12]. Again, this is dependent on the set of jobs. Microsoft

even says that the most common bottleneck on multiprocessor systems are when multiple processors request the same operating system or hardware resource [15].

6 Conclusion

We have discussed various scheduling techniques for both uniprocessor and multiprocessor systems. The scheduler has the option to allow processes to run to completion or to preempt them. First come first serve and shortest job first allow jobs to run to completion while round robin gives all processes an equal time quantum on the CPU. A multi-level feedback queue assigns priorities to processes, executes in a round robin manner, and can optionally enforce additional scheduling policies. Multiprocessor systems can implement a single, multi-server queue which feeds all processors or multiple, single-server queues.

The optimal scheduling policy largely depends on the tasks being processed. In public computation systems, it is possible for someone to game the system by finding loopholes in the scheduling policy. Thus it is important to use a MFQ to ensure fairness. On the other hand, in real-time systems, we must ensure high priority processes are executed and deadlines are met. With short jobs who require a few micro or even milliseconds, adding a scheduling policy may not improve system performance. If a combination of long and short jobs arrive, then a MFQ or round robin policy may be best. The experiment performed in this paper introduced immense overhead for implementing preemption and managing process priorities. In addition, the program was written in Java.

As processors approach their maximum, feasible clock speed, it is important to maximize CPU utilization in both uniprocessor and multiprocessor systems. As we add more cores, we will be able to execute larger jobs. It is important that the scheduling policy allows all processes to make progress and eventually get time on the CPU. We must design schedulers that adapt to the set of jobs in the queue. Perhaps, 5-10 years from now, we can implement more complex, artificially intelligent schedulers who take advantage of every scheduling mentioned in this paper.

References

[1] Abdel-Wahaba, H., *Scheduling*, 2012. http://www.cs.odu.edu/cs471w/spring12/lectures/Scheduling.htm

[2] Arpaci-Dusseau, A. and Arpaci-Dusseau, R., *Operating Systems: Three Easy Pieces*, 2012.

[3] Anderson, T. and Dahlin, M., *Operating Systems: Principles and Practice*, 2012.

[4] Corbato, F., Merwin-Daggett, M., and Daley, R., *An Experimental Time-Sharing System*, 1962.

[5] Halfin, S., *The Shortest Queue Problem*. Journal of Applied Probability. 1985.

[6] Hahne, E., *Round-Robin Scheduling for Max-Min Fairness in Data Networks*, 1991.

[7] Harchol-Balter, M., Osogami, T., Scheller-Wolf, A., and Wierman, A., *Multi-Server Queueing Systems with Multiple Priority Classes*, 2005.

[8] Reeves, G., *What Really Happened on Mars?*, NASA JPL, 1997.

[9] Liu, C., and Layland, J., *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the Association for Computing Machinery, 1973.

[10] Muller, G., *Scheduling Techniques and Analysis*, Buskerud University College, 2012.

[11] Mohammadi, A. and Akl, S., *Scheduling Algorithms for Real-Time Systems*, School of Computing, Queens University, 2005.

[12] Shukla, D., Jain, S., and Singhai, R., and Ojha, S., *Markov Chain Model for the Analysis of Round-Robin Scheduling Scheme*, 2009.

[13] Saab, P., *Scaling Memcached at Facebook*, 2008. http://www.facebook.com/note.php?note_id=39391378919

[14] Silberschatz, A., *Operating System Concepts*, 2005.

[15] Microsoft, *Monitoring Multiple Processor Computers*. http://technet.microsoft.com/en-us/library/cc722478.aspx

[16] Stallings, W., *Queuing Analysis*, 2000. http://ftp.csci.csusb.edu/ykarant/courses/w2008/csci531/QueuingAnalysis.pdf

[17] Raz, D., Avi-Itzhak, B., and Levy, H., *Fairness Considerations in Multi-Server and Multi-Queue Systems* 2005.

[18] Valerio, P., *Load Balancing for Disaster Recovery*, 2011. http://content.dell.com/us/en/enterprise/d/large-business/load-balancing-disaster.aspx

[19] Xu, J., and Parnas, D., *Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations*, IEEE Transactions on Software Engineering, 1990.