Writing Simulation Programs as a Tool for Understanding Internal Computer Processes

Michael D. Filsinger
University of Cincinnati

Abstract

The usual approach to a hands-on understanding of process scheduling in a computer operating system is to either modify an existing operating system or use a pre-written simulation program. In an Engineering Technology program, the first approach can be too difficult for the level of programming expertise possessed by the students, while the second approach does not give enough of a hands-on feel. In my Operating Systems class, I have the students write their own simulations of process scheduling. This approach provides many of the same benefits of modifying an existing operating system, while keeping the complexity of the task relatively low. This approach has the side benefit of providing the students with valuable additional programming practice.

Introduction

In our Computer Engineering Technology program, we teach two courses – Computer Architecture and Operating Systems – with a high theory content which can be difficult for a technology student to grasp. The difficulty lies in the fact that the details are buried deep within the relatively inaccessible core of the computer. Traditional approaches to this problem have been to use pre-existing simulation programs or, in the case of Operating Systems, to modify the kernel of an existing operating system. I have found that, given the lack of a strong Computer Science background among our students, a better approach to this problem is to have the students write simple simulations themselves. This technique provides a more intimate understanding of the processes they model while simultaneously giving the students valuable programming practice.

In this paper, I will examine one such problem from my Operating Systems course – modeling process scheduling algorithms. Computers are time-sharing devices: only one process can actually be using the CPU at any given instant. The procedure for selecting which process to run and for how long can have a strong impact on the performance of the system. Unfortunately, the scheduling algorithm is often buried deep within the core of the operating system. Furthermore, due to the non-deterministic nature of process introduction and completion, the effects of changing the algorithm are extremely difficult to measure under actual operating conditions.

By writing their own simulations of this process, students can filter out all of the distracting details and focus on the effects of changing the algorithm. Instead of actually running processes

in the operating system itself, a small set of "processes" is trivially simulated within the program. All processes are assumed to be present at the start, and the effects of I/O are ignored. The "processes" themselves are nothing more than numbers representing the required runtimes for the simulated processes, with no reference to the "tasks" performed by those processes. The response time and turnaround time of each process under the chosen algorithm can be easily measured within this controlled environment.

In later sections, I outline the basic problem of process scheduling, the actual programming project assigned to the students, the benefits of this approach, and student feedback.

Problem Description

One of the primary roles of an operating system is to manage the limited resources of the computer. Perhaps the most important (and limited) resource is the CPU itself. A single CPU is only able to execute a single instruction at any given time. Technically, a pipelined CPU executes different parts of several instructions at once, but since only a single instruction may be started or completed at any given time, this distinction can be ignored for our purposes.

On the other hand, while the CPU can only execute a single instruction (and hence, a single program) at a time, modern operating systems perform many tasks (and hence, run many programs) "at the same time". Each program in execution is called a process, and each process may, in turn, be broken into smaller pieces called threads. However, since threads are effectively the same as processes for scheduling purposes, I will restrict the discussion to processes only.

These processes must share the limited CPU resources of the system. The primary way of sharing these resources is through a form of time-sharing. Essentially, the operating system gives control of the CPU to a single process for a time, and then gives control to a different process. The heart of the process scheduling problem lies in the answers to two questions. First, how long should a process be allowed to run before it is interrupted by another process? Second, when a process is suspended, which of the available processes should be chosen to replace it? The policies determined by these decisions and the software to implement those policies form the scheduling algorithm of the operating system.

The choice of scheduling algorithm can have a profound impact on system performance. Performance is measured in terms of three metrics: turnaround time - the amount of time required to complete a single process; response time - the amount of time required to begin execution of a single process; and throughput - the amount of "work" completed per unit of time. Often, improving one of these performance characteristics has a negative impact on the others, and the exact usage of the computer system determines the relative importance of these three metrics. In interactive systems (such as the PCs we use every day), response time and turnaround time are considered most important.

A batch scheduling algorithm, such as the first-come first-served (FCFS) algorithm, which allows each process to run to completion before starting the next process, provides excellent throughput, but potentially at the cost of the other performance metrics. Suppose two or three

very long processes are at the front of the queue. All of the other (presumably shorter) processes must wait for these long processes to complete. As a result, the response times and turnaround times of these processes may be unnecessarily extended. This dramatically increases the average response time and average turnaround time of the system.

A simple time-sharing scheme, such as the round-robin algorithm, which allows a process to run for a predetermined time (time quantum) and then suspends that process in favor of another, dramatically improves response time, but as each process switch has an associated time overhead, throughput can be negatively affected. A small time quantum strongly favors response time over throughput, while a large time quantum begins to resemble a FCFS implementation. The effect on turnaround time is less clear, though low time quanta tend to have a negative impact on turnaround time as well.

For example, given ten processes with the runtimes specified as below (in arbitrary time units), and assuming an overhead of 50 time units for each process switch, we will examine the effect of the following algorithms on response time and turnaround time.

- Batch Algorithms
  1. First Come First Served
  2. Shortest Process First (run to completion)
  3. Longest Process First (run to completion)

- Interactive Algorithms
  1. Round Robin, Q=1000
  2. Round Robin, Q=10000
  3. Round Robin, Q=10

Table 1 – Processes and Required Runtimes

Process Number	1	2	3	4	5	6	7	8	9	10
Run Time	1000	1300	600	800	1500	500	5000	3000	1000	500

Note that, for the sake of simplicity, I make two unrealistic assumptions:

1. Each process is assumed to have been initiated (in numeric order) at time t=0.

2. The effects of process I/O are ignored.

Table 2 – Response Times for Given Algorithms

	FCFS	SPF	LPF	RR Q=1000	RR Q=10000	RR Q=10
1	50	2650	11050	50	50	50
2	1100	4750	9700	1100	1100	110
3	2450	1150	14000	2150	2450	170
4	3100	1800	13150	2800	3100	230
5	3950	6100	8150	3650	3950	290
6	5500	50	14650	4700	5500	350
7	6050	10700	50	5250	6050	410
8	11100	7650	5100	6300	11100	470
9	14150	3700	12100	7350	14150	530
10	15200	600	15200	8400	15200	590
Avg.	*6265*	*3915*	*10315*	*4175*	*6265*	*320*

Table 3 – Turnaround Times for Given Algorithms

	FCFS	SPF	LPF	RR Q=1000	RR Q=10000	RR Q=10
1	1050	3650	12050	1050	1050	50100
2	2400	6050	11000	9250	2400	57420
3	3050	1750	14600	2750	3050	34500
4	3900	2600	13950	3600	3900	42960
5	5450	7600	9650	9800	5450	61080
6	6000	550	15150	5200	6000	29760
7	11050	15700	5050	16100	11050	91200
8	14100	10650	8100	14000	14100	79200
9	15150	4700	13100	8350	15150	50400
10	15700	1100	15700	8900	15700	30000
Avg.	*7785*	*5435*	*11835*	*7900*	*7785*	*52662*

As can be seen, among the batch algorithms, shortest process first is best for both metrics, while longest process first (an impractical algorithm provided for comparison only) is the worst for both metrics. The round robin algorithm is more frequently used on interactive systems. The extremely low value of Q produces outstanding response time, but at the cost of an extraordinarily bad turnaround time. A value between 100 and 1000 provides a more typical behavior.

The difficulty with studying the effects of the choice of scheduling algorithm lies in the fact that the scheduling algorithm is generally buried deep within the core of the operating system. It is one of the lowest-level functions performed by the system. Furthermore, due to the non-

deterministic nature of process introduction and completion, the effects of changing the algorithm are extremely difficult to measure under actual operating conditions.

Project Description

The usual way of dealing with the difficulties of measuring the effects of different choices of scheduling algorithms is to run simulations of process execution. Instead of running a sequence of actual processes on a system and measuring the desired times, a sequence of processes is instead modeled in a tightly controlled simulation environment. A number of existing simulators exist, but I find that many students still fail to grasp the concepts when using such pre-built software. They lack the hands-on feel of building the scheduling routines for themselves.

My solution is to find a middle ground. Modifying the scheduling parameters of an actual operating system is too difficult, and the effects are too difficult to measure anyway, while using an existing simulator package does not provide the nuts and bolts understanding of the algorithm itself. Therefore, I have students construct a simple simulator environment of their own. In this way, the distracting details can be filtered out, while still involving the students intimately with the algorithms they are measuring.

In the environment constructed by the students, a small set of "processes" is trivially simulated. All processes are assumed to be present at the start, and the effects of I/O are ignored. The "processes" themselves are nothing more than numbers representing the required runtimes for the simulated processes, with no reference to the "tasks" performed by those processes.

Pseudocode of Simulation Program

Data:

```
startTime[]        // Keeps track of start time for each process
endTime[]          // Keeps track of completion time for each process
remainingTime[]    // Keeps track of time remaining for each process
                   // Initialized to process required runtime
systemClock        // Keeps track of total "elapsed time"
```

Procedures:

```
main()
{
    while (at least one unfinished process exists)
    {
        nextProcess = schedule()
        run(nextProcess)
    }
```

```
        output results
}

int schedule()
{
        update systemClock to reflect "overhead" in the scheduling algorithm

        choose process according to the rules associated with the chosen scheduling
        algorithm

        return process
}

void run(process)
{
        if (startTime[process] == 0)
                update startTime[process] to record process start time

        update remainingTime[process] according to the rules associated with the chosen
        scheduling algorithm

        update systemClock to reflect "elapsed time" from the previous step

        if (remainingTime[process] == 0)
                update endTime[process] to record process completion time
}
```

I usually have students model three different batch algorithms (first-come first-served [FCFS], shortest process first [SPF], and longest process first [LPF]) as well as the round robin algorithm with various time quanta. The core operation of these programs is the same, with only the bolded parts of the schedule() and run() procedures requiring change.

As mentioned in the previous section, I make two unrealistic assumptions for the sake of simplicity with this assignment. First, I assume that all processes are introduced at time t=0. This is not an unreasonable assumption for batch systems, but it is not at all the norm for interactive systems. However, for the sake of understanding the effects of different scheduling algorithms, allowing processes to be introduced at arbitrary times is not really necessary. On the other hand, removing this assumption would not add too much complication to the resulting programs, so this assumption can be easily removed if desired.

Second, I ignore the effects of I/O within the processes. Given that the need for processes to wait for I/O operations is one of the primary motivations for multiprogramming, this is a serious

limitation. However, modeling process I/O would add an extraordinary amount of complexity to the programs, so I sacrifice this aspect for the sake of ease of programming.

Benefits

This approach to studying process scheduling has two major benefits over more traditional approaches. First, it allows the student to focus on the algorithm itself and its effects, without dealing with the distracting details of the actual implementation. Furthermore, many of the activities in the operating system are (at least, to outside observation) nondeterministic in nature. These activities can mask the effects of changing the scheduling algorithm. Simulator environments remove these distractions and make the concepts much more accessible.

The second benefit is that this approach provides students with valuable programming experience, while not presenting them with challenges beyond their current level of ability. One of the major problems I encounter as a professor of Computer Engineering Technology is that many of the students have only had one course in C programming, and these students generally do not understand even the most basic programming concepts yet. This single, quarter-long course struggles to cover all of the basic ideas of programming, and students have no time to absorb and practice what they have learned. This assignment gives these students valuable experience, programming with functions, loops, and arrays - the most fundamental of programming skills.

Student Feedback

Only anecdotal feedback exists, in the form of informal personal comments and comments made in lab reports and teaching evaluations. In general, student comments fall into two categories. First (and by far more vocal), are the students who complain that "This is not a programming class, so why am I required to write programs?" To some extent, I see this attitude as validation of my belief that the students need more programming practice, not less. In fact, many of these students later come back to me and tell me how much more confidence they have concerning programming after the assignment. Much of their initial resistance stems from their lack of confidence and practice. As they discover that the problem is relatively simple to solve, that resistance fades.

Second, and more importantly, students in general seem to feel that they have a better understanding of the underlying scheduling concept. I have used the pre-written MOSS[1] simulator in previous OS classes, but never in the same class as student-written simulations, so I have no way of directly comparing the level of understanding. My overall impression as a teacher is that the students generally did not gain as much understanding from the pre-written simulator as from writing a simulation from scratch. The process of actually writing the algorithms forces the students to study the inner workings of the algorithm in much greater detail.

Conclusion

Many further applications of student-written simulation programs exist. In my Operating Systems class, I use this technique for demonstrating page replacement (memory allocation) algorithms. I also use this method in Computer Architecture for demonstrating the benefits of pipelined CPU operation. In both of these cases, the benefits have been as pronounced as for the process scheduling application.

Some other applications include the following. In Operating Systems, a simulation program could demonstrate the concepts of file allocation and disk fragmentation. In Computer Networks, such a project could be used to demonstrate such concepts as network routing computations (such as Dijkstra's Algorithm), network address computations, timesharing simulations, etc. The possibilities are virtually endless.

When studying Computer Science or Computer Engineering, many concepts exist which are very simple conceptually, but difficult to demonstrate practically. The writing of simulation programs to demonstrate these concepts is the best way I have yet found for making these ideas accessible to the students.

Bibliography

1. Ontko, R., Reeder, A., Tanenbaum, A. (2001). *Modern Operating Systems Simulators.* http://www.ontko.com/moss/

2. Tanenbaum, A. (2001). *Modern Operating Systems, 2nd ed.* Prentice Hall, Upper Saddle River, NJ, Chapter 2

3. Stallings, W. (2001). *Operating Systems: Internals and Design Principles, 4th ed.* Prentice Hall, Upper Saddle River, NJ, Chapter 9

Biography

MICHAEL D. FILSINGER is an Assistant Professor of Electrical and Computer Engineering Technology at the University of Cincinnati. He received a BA in Mathematics and MS degrees in Mathematics and Computer Science from the University of Cincinnati in 1990, 1992, and 1994, respectively. In addition to teaching, he has served as a computer system administrator. He is a member of IEEE, ASEE, and the Phi Beta Kappa honor society.