Operating Systems I: An Operating System Simulation Developed as Homework

Dr. Ronald Marsh
Computer Science Department
University of North Dakota
Grand Forks, ND 58201
rmarsh@cs.und.edu
701-777-4013

Abstract

This paper describes the series of homework assignments made during the fall 2000 Operating Systems I course at the University of North Dakota as well as the authors experience in making the assignments. The purpose of the assignments was not only to provide the student with a good understanding of how an operating system is developed and operates, but also to reinforce some basic skills that all graduating computer science majors ought to possess.

Starting with a simple operating system simulation (one-queue round-robin CPU scheduler and one I/O queue) and progressing to a credit based priority CPU scheduler, multiple I/O queue, and page fault managing operating system simulation, the assignments mandated that the student develop a series of operating system simulations ranging in complexity.

Introduction

The primary intent of CSci-451 (Operating Systems I) at the University of North Dakota is to expose students to the fundamental theories behind operating systems. A secondary purpose is to further their development as systems analysts and improve their programming and debugging skills. CSci-451 is taught in the traditional manner - relying on a mixture of lectures, homework assignments, and tests.

In previous offerings of this course the homework assignments, while relating to operating systems concepts, were independent of each other. The author noticed two disturbing phenomena:

1. A number of students would simply ignore the more difficult assignments and count on the remaining assignments to maintain a passing grade in the course.

2. The student's systems analysis and design, programming, and debugging skills were not improving as much as expected (or desired).

A reoccurring theme in the columns in the various trade publications[1] as well as in the editorials and letters-to-the-editors in these same publications relates to the latter of the above. Namely, computer professionals are not receiving the training required to meet the demands of the current IT driven world. This is not a new revelation. Anyone who watches the evening news is well aware of the media attention that has been focused on the alleged failure of the American public school system and how American students have allegedly failed to master basic skills in reading, writing, arithmetic, and science. A more disturbing finding made in the early 90s was that few American students could use their knowledge effectively in thinking and reasoning[2]. It would appear that students simply memorize the material (the learning method they think they know best[3]) and fail to learn how to use complex reasoning skills to discover higher relationships. Most students master the material and do well in the course. However, it is also evident that a disconcerting number of students need to develop better design and debugging skills. The maxim "**What you learn not only changes what you think about, but how you think.**" could easily be applied to the IT profession. This author firmly believes in developing course materials that force the student to develop their design and debugging skills.

In an effort to reduce the occurrence of the phenomena listed above the decision was made to develop a series of assignments that would force to the student to build on previous assignments (admittedly, not a new idea). Since the class focuses on operating systems and since the best way to understand how something works is to build one, the assignments mandated that each student develop a series of operating system simulations ranging in complexity. The "trick" was to organize the development of the different operating system simulations such that poor programming practices or such that an initially poor design would haunt them (the student) for the rest of the semester and reinforce the precept of having a good design to start with.

Rules Of The “Game”

As any educator knows, different students have different strengths and weaknesses and preferential ways of completing a task. Therefore, only a few ground rules were enforced regarding the simulation development:

- The development language was limited to C or C++.
- The simulation was required to run on the department’s main UNIX cluster (Linux).
- Plagiarism was not allowed.
- Achieve the goals as set forth in each assignment.

Otherwise, the students were allowed to:

- Use any data structures they desired.
- Use an object oriented design or not.
- Make use of the available queue libraries.
- Implement a simple time-keeping mechanism.
- Discuss implementation issues amongst themselves.

The above “rules” allowed the students to develop the simulation using methods / tools they felt comfortable with. Yet, forced them to understand the problem and devise an implementation providing a solution.

Assignments

The course typically requires students to complete ten homework assignments. For the semester in question, all but three involved the development of the operating system simulation. The assignments are discussed individually in the following sections.

Assignment #1 (a non-simulation assignment)

Assignment 1 required the development of a simple UNIX shell to accept a DOS command and execute (via a “system” call) the corresponding UNIX command. The purpose of this assignment is to insure UNIX knowledge and to develop an understanding of how a UNIX shell operates.

Assignment #2

Assignment 2 required the development of a simple one ready queue and one I/O queue operating system simulation. The simulation required:

- A process control block (PCB) structure including a Process ID (PID), a CPU usage term (CUT), an I/O request term (IRT), and a waiting term (WT).

- Use of the computer's internal clock to get the time (seconds) values and the system's random number generator.

- The ready queue and I/O queues to be arrays of structures. The array size was left to the student to determine.

The program was to behave as follows:

1. Created a new process and assign it a PID (set the CUT, IRT, and WT to zero), and put it into the ready queue (since the ready queue is empty, this PCB will be at the front of the queue).

2. Move the PCB at the front of the ready queue into the CPU. Call the system clock to get the time the process started. To simulate the processing time, call the random number generator to select a value from 0 to 10,000, and executed the following loop:
   ```
   For (i = 0; i < value; i++) j=sin(i);
   ```

3. When the loop finishes call the system clock to get the time the process stopped (or was forced to stop). Using the start and stop time add the CPU usage time to the CUT value. Use those values to adjust the WT. Use the random number generator to select a value from 0 to 3 to decide what the process needs to do next. The options are:
   0 – process terminates and is removed from the system.
   1 – process returns to the ready queue to wait its turn.
   2 – process requires I/O and goes into the I/O queue.

4. If the process terminates – print out the PID, CUT, IRT, and WT values.

5. If the process returns to the ready queue – repeat from step 2 above.

6. If the process goes into an I/O queue call the random number generator to select a value from 0 to 100. Assume that the number returned is the number of seconds that the process had to wait for the I/O to complete. This value will affect the WT value. Then place the PCB back in the ready queue.

7. Occasionally, create a new process and let the simulation run until 25 processes have successfully terminated.

The operating system that was simulated is shown in Figure 1 below.

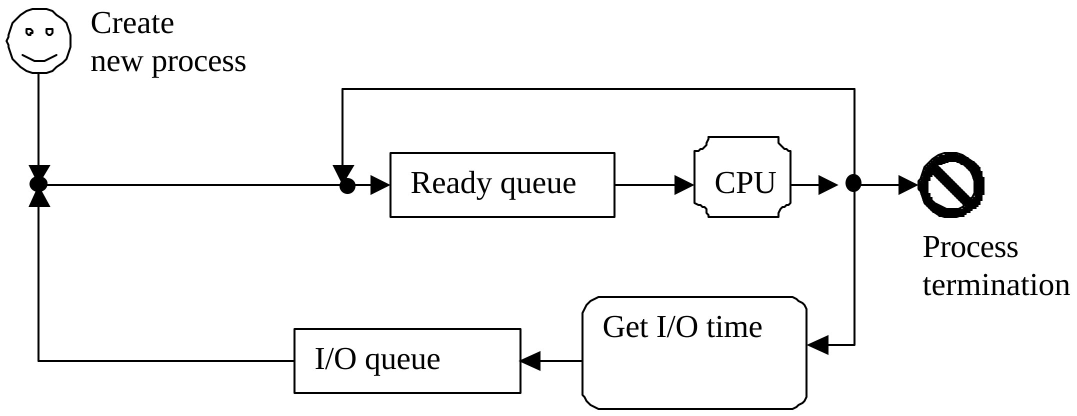

Figure 1: HW 2 - Operating system simulation.

As well as being the entry point to the simulation development, the assignment required the use of the system clock forcing the student to explore UNIX's "man page" system.

Assignment #3 (a non-simulation assignment)

Assignment 3 required the development of a pair of communicating processes. However, the assignment had no solution and students who failed to analyze the problem before commencing coding spent a lot of time struggling to arrive at a (non-existent) solution.

Assignment #4

Assignment 4 made several changes intended to result in a more realistic and consistent simulation. To those ends we:

- Augmented the system's random number generator with a random number generator based on the Weibull distribution (code provided) to calculate a more realistic value for the newly added CPU and I/O burst times.

- Replaced the system clock (too dependant on system load) with our own "clock". The students were allowed to construct a simple looping control mechanism for monitoring the queues and for moving PCB's between the queues and CPU. Each loop traversal was assumed to be 1 time unit. While this did result in a lot of wasted CPU cycles on Agassiz, it was easy to implement.

- Limited the time a process could be in the CPU by adding a CPU time quantum.

- Added a CPU utilization calculation.

The most significant change was how the CPU was modeled and when processes leave the CPU. A process now leaves the CPU as two cases apply:

1. The time slice has expired, but the CPU burst time has not. At which point the process must return to the ready queue.

2. The CPU burst time has expired, at which point the process must either exit or go to the I/O queue. The system random number generator was used to decide which (we assumed that only 5% of the time a process will terminate).

The operating system that was simulated is shown in Figure 2 below.

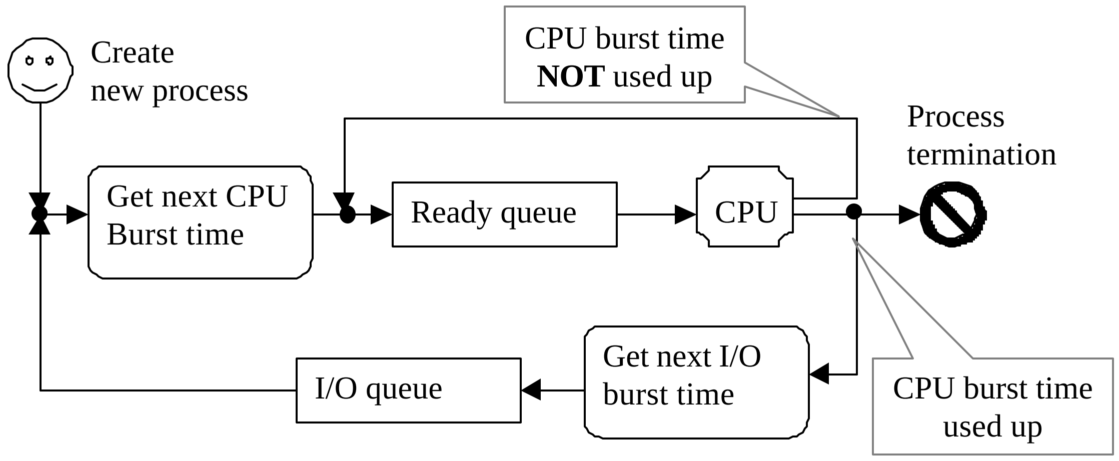

Figure 2: HW4 - Operating system simulation.

Assignment #5

Assignment 5 made additional changes intended to result in an even more realistic simulation and emphasize the importance of a good original design. To those ends we:

- Added a second ready queue and I/O queue to simulate a priority based operating system and to emphasized the importance of a good original design. Students that used objects to represent the queues had little difficulty with this task, while students that did not use objects did have difficulty.

- Added a context-switch time (CST) penalty factor to punish processes that were long-running and CPU bound (a common practice in operating systems).

- Adding an “aging” mechanism of each student’s choice to facilitate the (temporary) transfer of processes in the low priority queue to the high priority queue.

The operating system that was simulated is shown in Figure 3 below.

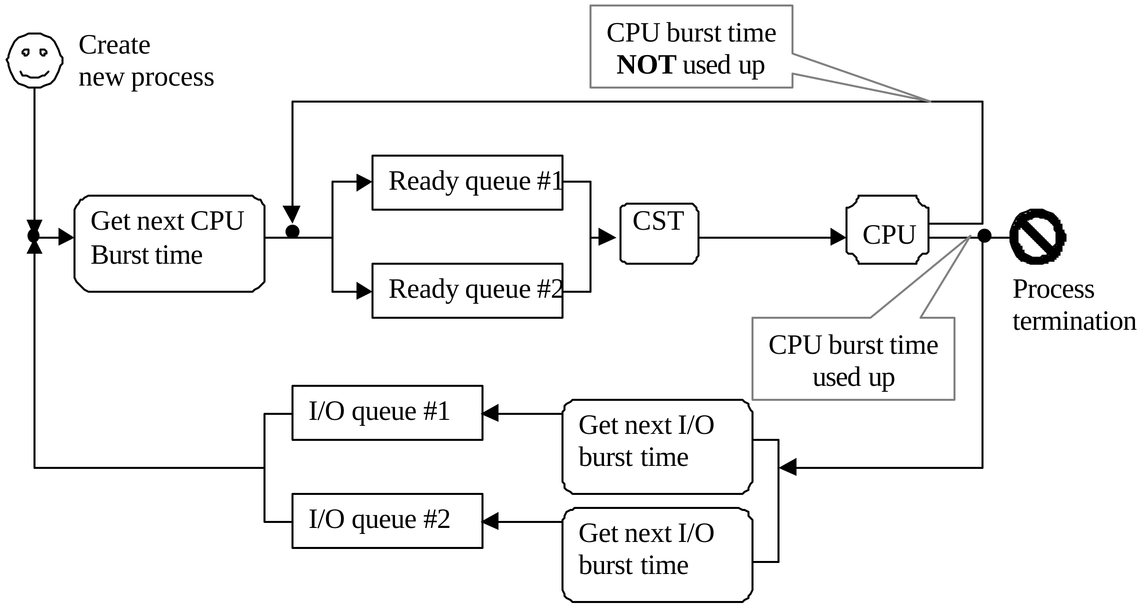

Figure 3: HW5 - Operating system simulation.

As stated above, part of the purpose of this assignment was to enforce the idea of starting with a good design. Several students complained about having to start over (because of a poor original design).

Assignment #6

Assignment 6 required the student to clearly comment their code, to refine some of the Weibull distribution's parameters, and to apply a battery of test cases. The purpose for this assignment was to verify that the current simulations were working correctly and to give students a “breather” before the next major change.

Assignment #7

Assignment 7 required the addition of a page replacement algorithm to the simulation. Since we were developing a simulation, we allowed the use of programming constructs to simulate the disk. Namely:

- A (global) data structure to represent the virtual memory space (the pages) was permissible. A simple linear integer array of length 500 initially populated with zeros was allowed. The array values were the number of times each respective page had been removed from memory by a page fault (ie: a page fault counter).

- A (global) data structure to represent the physical memory space (the frame pages) was also permissible. A simple 2-column (column 1 for a free/VM-page-number and column 2 for a usage counter) linear integer array of length 50 initially populated with zeros was allowed.

- A table in the PCB to record which pages (of the 500 possible) were used by the program was also allowed.

The page replacement algorithm was to behave as follows:

1. When a new process is created the random number generator is used to determine how many pages the process will require (students could assume that at most 1 to 10 pages per process would be required). Students could also assume that this number would not change during the life of the program.

2. Assign values (1 through 500) to each of the applicable PCB table entries using the random number generator. Multiple processes are allowed to use any given page (ie: shared memory). Students could also assume that these pages would not change during the life of the program (we assumed that a process would know at startup what pages it requires).

3. For every process sent to the CPU (context switch) check to insure that the pages required by that process are in physical memory. If any pages are missing from memory a "page fault" needs to be generated (for each missing page) to retrieve the missing page. Therefore, a new process generated several page faults, since none of its pages were be in memory (unless shared pages were involved). Each page replacement cost 10 cycles.

4. Apply a page replacement algorithm (of their choice - any from the book) for every page fault generated to find a location to insert the required page. Every time a page is replaced the applicable page fault counter is incremented.

5. Maintenance of a "usage" value for each process's page (while in physical memory) is allowed if required.

6. When a process terminates its pages are to be removed from physical memory.

The operating system that was simulated is shown in Figure 4 below.

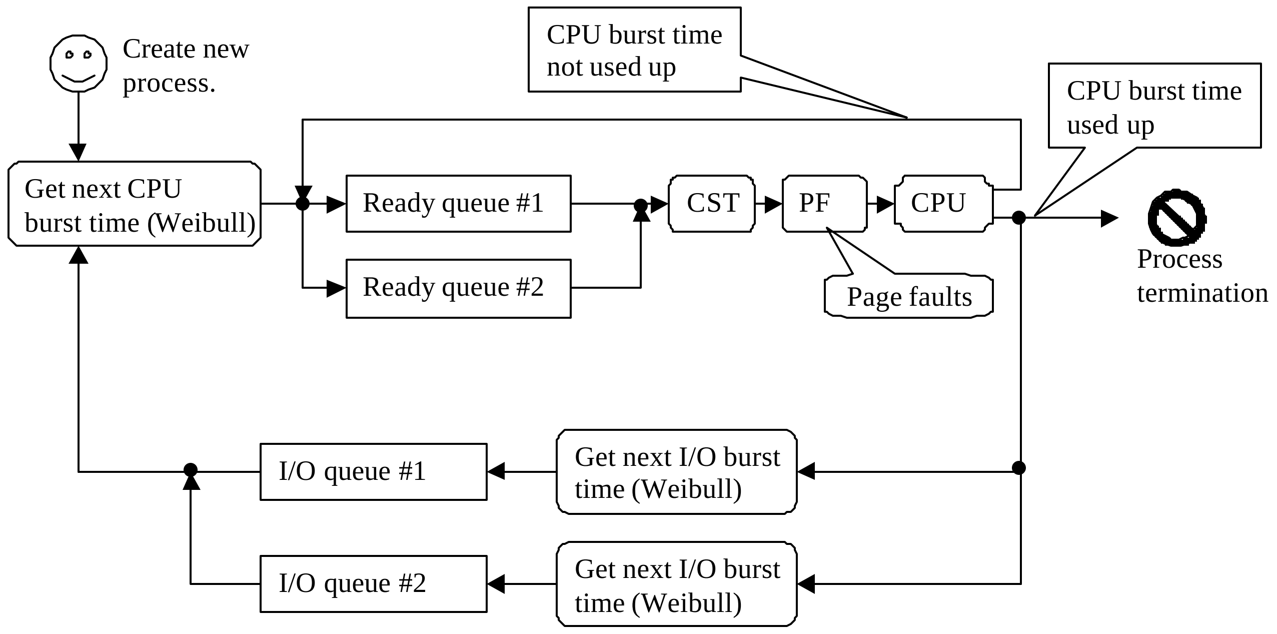

Figure 4: Page replacement simulation.

Assignment #8

Assignment 8 required the student to refine the page replacement implementation methodology (to more accurately model the cost of page faults), to further modularize their code, and to apply a battery of test cases. A problem occurs with this assignment, in that a real operating system employs a complex "sleep", "awake", queuing mechanism to efficiently manage processes that generate page faults as they enter the CPU. While the concept of page faults and page replacement is an important operating system concept, one can argue that the implementation details are of a lesser concern. Hence, a simple 1 process look-ahead scheme was devised as is described below:

- We argue that we have a system that does a 1 process look-ahead into the ready-queue and prefetches any pages required. So while process N is running, the system is loading process N+1's pages. Pages still require time to load, and if the number of pages to load is below some value (dependant on the remaining time that process N holds the CPU) there will be no page fault cost. Otherwise, there will be (since we can't load a process into the CPU until it has all of its pages in memory, there may be times when the CPU has to sit and wait idle for the process).

The operating system that was simulated is shown in Figure 5 below. Note the repositioning of the page fault (PF) mechanism to manage our 1 process look-ahead scheme.

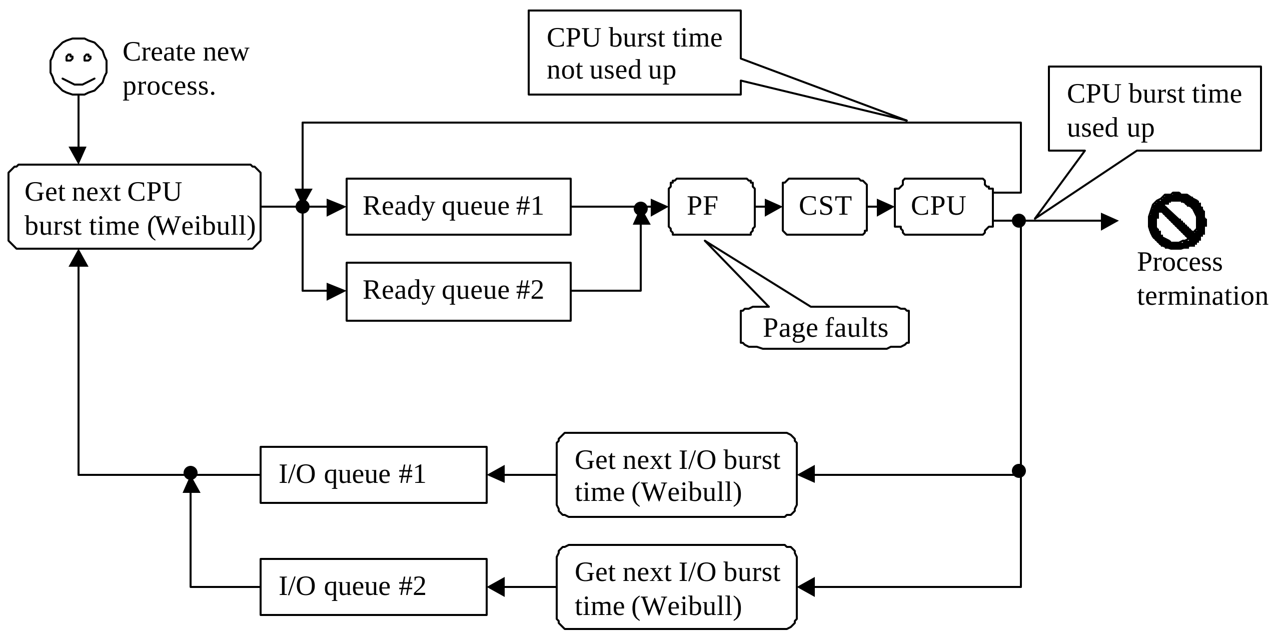

Figure 5: Revised page replacement simulation.

Assignment #9

Assignment 9 required the student to replace their existing aging/priority method with the credit-based aging/priority method that the Linux operating system uses. This requirement actuality simplified the code, since the credit based method achieves a priority-like behavior using one ready queue. Students who did not have a good design complained about "throwing away" code that they had worked hard to develop. Students who did develop good designs commented about how easy (compared to some of the other assignments) this one was.

Assignment #10 (a non-simulation assignment)

Assignment 10 required the development a suite of communicating processes using UNIX's inter process communication (IPC) facility. The assignment required the development of a parent process that would spawn (fork) three child processes. The child processes were required to communicate (pass data) via three UNIX pipes and one shared memory location. A pair of semaphores controlled shared memory access. Few students ever get this assignment to fully work and those that do are usually very satisfied with themselves.

Conclusion

This paper describes the series of homework assignments made during the fall 2000 Operating Systems I course at the University of North Dakota. The purpose of the assignments was not only to provide the student with a good understanding of how an operating system is developed and operates, but also to reinforce some basic skills that all graduating computer science majors ought to possess.

Due to the conflicting requirements, all students were forced to revise their work (to some extent) as the semester progressed. Some students encountered many problems and were forced to "start over." Others managed to adapt their existing code, but not without great difficulty. As expected there were complaints from some students who felt that the assignments were unfair and that I did not think the assignment out well enough to avoid the conflicting requirements. However, many students mastered the material and some even questioned why we (the class) didn't pursue a more accurate / complex simulation. Students asked why their simulations behaved as they did or why we used certain values for the different costs (they felt the numbers used were unrealistic – and in some cases they were, but it made the simulation easier to verify). I frequently heard students discussing the merits of using objects for some components, but not other components.

After the semester ended, I asked several students what they thought of the assignments. Most said that they were a lot of work, but they learned a lot from doing them. One student commented on how she finally felt like she was ready to graduate – having accomplished all of the assignments successfully. The general consensus was to keep the simulation development but to simply (eliminate the conflicts in) the requirements.

From student comments it is my opinion that the combination of a large long-term development project assigned in a piecemeal manner such that the subsections include partially conflicting requirements (with previously assigned subsections) reinforced the basic skills that we expect such students to possess.

References

1. Holmes, N. (2000). *Why Johnny Can't Program*. (IEEE) Computer, Vol. 33, No. 12, 158-160.
2. Neubert, G. A and Binko, J. B. (1992). *Inductive Reasoning in the Secondary Classroom*. National Education Association of the United States, Washington, DC.
3. Clarke, J. H., Raths, J., and Gilbert, G. L. (1989). *Induction Towers: Letting Students See How They Think*. Journal of Reading, Vol. 33, No. 2, 86-95.