Visualizing the CPU Scheduler and Page Replacement Algorithms

Sami Khuri
San José State University
Dept. of Math. and Computer Science
San José, CA 95192-0103, USA
khuri@cs.sjsu.edu

Hsiu-Chin Hsu
San José State University
Dept. of Math and Computer Science
San José, CA 95192-0103, USA
hsu0832@sundance.sjsu.edu

Abstract
In this paper, we present two packages that simulate the multilevel feedback queue scheduling algorithm for a single CPU, and five page replacement algorithms that are used in the context of memory management. The paper gives a brief description of the interactive, self-paced packages and explains how we use them in Operating System courses. We also highlight the merits of the packages and the benefits to the students derived from our Java written simulations.

1 Introduction

The advent of extensive access to the World Wide Web and Java applets have boosted the use of graphical visualization and animation to convey the dynamic behavior of computer algorithms [4]. The Web has numerous repositories of the animation of different traditional algorithms encountered in CS1, CS2, and data structures and algorithms courses, such as sorting, searches, traversals of trees and other graph algorithms. For many types of algorithms, animation enhances learning and understanding [2]. It is our belief that many algorithms encountered in operating systems fall under this category.

The main purpose of this work is to present two packages, MLFQ that simulates the multilevel feedback queue scheduling algorithm for a single CPU, and PAGE, a collection of five page replacement algorithms that can be used in Operating Systems courses. Most modern operating systems use intricate process scheduling algorithms that are not very easy to explain. Understanding each part of the scheduling separately is straightforward, but grasping the interaction between the different components is more challenging. We have designed, written and used in our classes a package, MLFQ that simulates the multilevel feedback queue scheduling algorithm for that purpose. We have also written PAGE, the simulation of algorithms for virtual memory, and have successfully used it in our courses. Both packages, written in Java, are highly interactive and user-friendly. By experimenting with MLFQ and PAGE, students get a full appreciation of all the intricate details that occur during process scheduling and page replacements.

In the next section, we describe MLFQ that simulates the multilevel feedback queue and show how we use it in our Operating Systems class. In Section 3, we give a brief description of PAGE, the package that simulates five page replacement algorithms. We conclude with some additional remarks and benefits of our packages.

2 The Multilevel Feedback Queue

Unix and Windows NT use a strict, multilevel feedback queue scheduling algorithm. Modern operating systems support up to 160 queues in which a process is placed, depending on its priority [6]. The scheduler can be priority-preemptive, where a running process can be preempted from the CPU by another process that has a higher priority, or non-priority preemptive where a process is allowed to complete its time-slice at the CPU.

A single process scheduler should satisfy the following scheduling objectives:

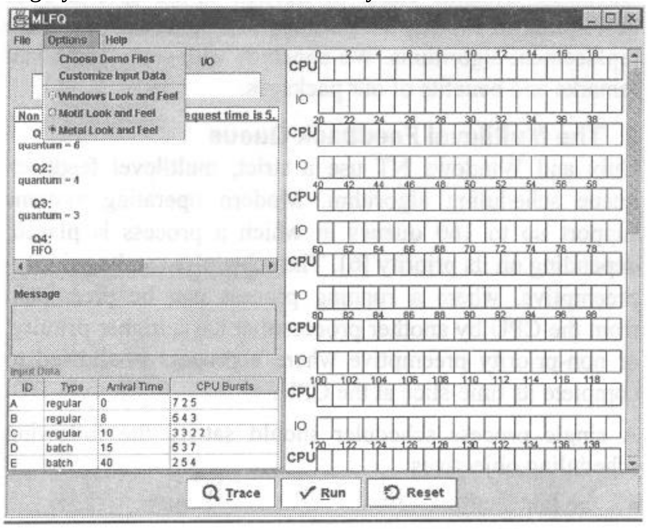
- be fair
- maximize throughput
- maximize the number of interactive users receiving acceptable response times
- be predictable
- minimize overhead
- balance resource utilization
- achieve a balance between response and utilization
- avoid indefinite postponement
- obey priorities
- give preference to processes that hold key resources
- give a lower grade of service to high-overhead processes
- degrade gracefully under heavy loads.

There are various scheduling algorithms, such as round robin (first-in first-out, where each process has a quantum), shortest job first, priority scheduling, and the multilevel feedback queue. Although various modern operating systems have adopted the multilevel feedback queue model for their short-term CPU scheduler, they do not necessarily have the same characteristics. Each scheduler needs to define:

- the number of queues
- the scheduling algorithm for each queue
- the conditions for increasing/decreasing a process' priority
- the rules for deciding which queue a process will first enter.

While these concepts are covered in class in traditional lecture form, the students can get a better understanding of the different stages of the scheduler and its intricacies through a simulation of the CPU scheduler.

The package we wrote in Java and use in our classes, MLFQ, simulates the CPU scheduler by using four queues and two different kinds of jobs: regular and batch. It is highly interactive and user-friendly.



Non ... equest time is 5.

Q1: quantum = 6

Q2: quantum = 4

Q3: quantum = 3

Q4: FIFO

Message

Input Data

ID	Type	Arrival Time	CPU Bursts
A	regular	0	7 2 5
B	regular	8	5 4 3
C	regular	10	3 3 2 2
D	batch	15	5 3 7
E	batch	40	2 5 4

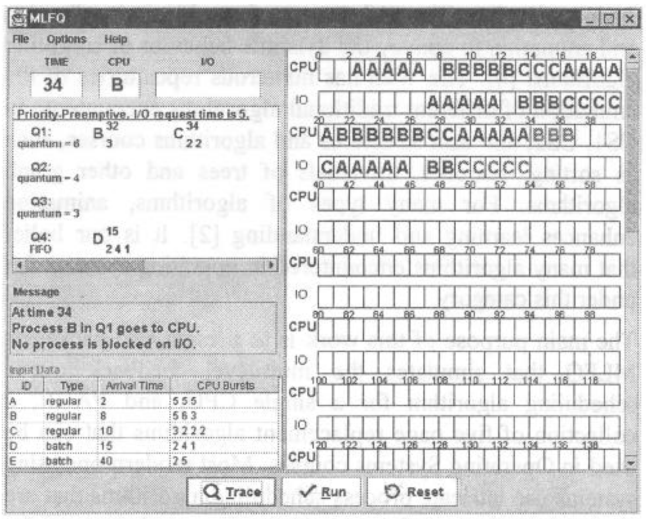
Figure 1. Snapshot of MLFQ with a Demo Input File.

As can be seen in Figure 1, under "Options" users can click on "Choose Demo Files" and choose among four available input data tiles, or enter their own input data. They can also customize their simulation by adjusting:

- the process selection type:
  1. priority preemptive: If a job becomes ready while another job with lower priority is running, the running job is stopped (preempted) and the new job is scheduled.
  2. non-priority preemptive: a process executes until either it uses its entire quantum or it blocks on I/O (finishes its CPU burst).
- the number of processes
- the type of each process (regular or batch), its arrival time, and its execution trace (CPU bursts)
- the I/O request time (the number of units a process is blocked on I/O)
- the time quantum of each round robin queue

The "Trace" and "Run" buttons control the execution of the program. The user can choose between a step-by-step approach in order to understand the details of the algorithm, by repeatedly clicking on "Trace". Alternatively, the user can choose "Run" and have the simulator go through the steps and display the CPU schedule for the input data.

In the simulation of Figure 2, we have chosen to run the first demo file which is summarized in the lower left hand comer under "Input Data". We note that the input tile here is different than the one found in Figure 1. Process A is a regular process. It arrives at time 2 and requests a burst of 5 units of CPU time, then blocks on I/O, requests a second burst of 5 units, blocks on I/O again, and then requests one last burst of 5 units. In other words, we assume that demands for two consecutive bursts always include blocking on I/O between the CPU requests. Process B is also regular, arrives at 8, and has three CPU bursts of 5, 6 and 3 units, and so on for the remaining three processes, C, D, and E.



TIME 34 CPU B

Priority-Preemptive, I/O request time is 5.

Q1: quantum = 6 B^{32}_{3} $C^{34}_{2\,2}$

Q2: quantum = 4

Q3: quantum = 3

Q4: FIFO $D^{15}_{2\,4\,1}$

Message

At time 34
Process B in Q1 goes to CPU.
No process is blocked on I/O.

Input Data

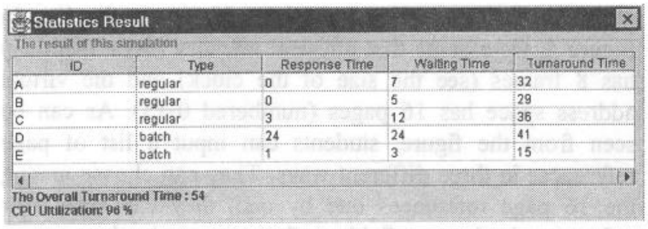
ID	Type	Arrival Time	CPU Bursts
A	regular	2	5 5 5
B	regular	8	5 6 3
C	regular	10	3 2 2 2
D	batch	15	2 4 1
E	batch	40	2 5

Figure 2. Snapshot of MLFQ at Time 34.

Figure 2 also shows the quanta settings that were chosen for queues Ql, Q2, and Q3 as well as the process selection type (priority preemptive or non-priority preemptive) and the length of time a process is blocked on I/O. As can be seen in the upper left comer of Figure 2, the simulation we run is priority-preemptive, a process blocks on I/O for 5 units of time, and the quanta of Ql, Q2, and Q3 are 6, 4, and 3, respectively.

Figure 2 gives a snapshot of the scenario at time 34. In the window showing the queues, we have B^{32}_{3} and $C^{34}_{2\,2}$ in Q1. The superscript represents the time at which the corresponding process was placed in the queue. The subscript represents the remaining CPU burst requests [3]. At time 34, process B has just been scheduled and no process is blocked on I/O. This information can be seen in the three boxes in the top left comer of Figure 2, and in the "Message" box. Process B waited in Ql for two units of time (34 - 32) before being scheduled. As can be seen in the tracing window in the right half of Figure 2, process B has already completed its first CPU burst of 5 units between 8 and 13, and the second CPU burst of 6 units between 21 and 27. Note that, at time 34, process C has just come back from I/O and is now in Ql. Process D has been waiting for its turn in Q4 since its arrival at time 15. Process A has just finished executing its third CPU burst and is done, while process E is neither in the queue window nor in the tracing window simply because it will arrive 6 units (40 - 34) later.

We note that the message window and I/O box will always record all processes that are blocked on I/O. The tracing window has room for only one process: the most recent one.



Statistics Result

The result of this simulation

ID	Type	Response Time	Waiting Time	Turnaround Time
A	regular	0	7	32
B	regular	0	5	29
C	regular	3	12	36
D	batch	24	24	41
E	batch	1	3	15

The Overall Turnaround Time : 54
CPU Ultilization: 96 %

Figure 3. The Results of the Simulation.

At the end of the simulation (see Figure 3), a "Statistics Result" window pops up and gives the response time, waiting time, process turnaround times and overall turnaround times and the CPU utilization for this simulation.

We note that this is a simple example where no process is preempted by another of higher priority.

As mentioned above, we use MLFQ for in-class demonstrations. The "Trace" option allows to pause the simulation, explain the details and answer students' questions. The simulation has also proven to be a valuable tool for open laboratory exercises. Students can experiment with MLFQ at their own pace and compare the performance of the scheduler for various values of the quantum. They can see for example, that if the quantum is too high, the performance of the round robin queue degrades into that of a first-in first-out queue.

Visualization and simulation tools have been proven very valuable but students learn even better when they write their own implementations of the algorithms studied in class. When we teach our undergraduate Operating Systems course at San Jose State University, we reinforce the concepts learnt in the classroom by giving a programming term project. Simulating the CPU scheduler is an example of a good topic for term projects. In the next section, we introduce the project we gave in the Fall'97 semester, where the students were given the choice between C and C++ for the implementation.

2.1 Project Specification

In this project the students are asked to implement the strict, priority preemptive multilevel feedback queue scheduling algorithm with four queues (Ql, Q2, Q3, Q4) for a single-CPU system. The program should output the response time, the turnaround time, the waiting time for each process and the overall turnaround time (makespan) for the given input data.

The following assumptions apply:

1. Arriving jobs can be either batch or regular.
1. Regular processes enter in Q1, and batch processes enter in Q4.
2. Q1 has the highest priority, followed by Q2, then Q3, while Q4 has the lowest priority.
3. The scheduling algorithm for each of the four queues is:

   Q1: round robin with quantum 3

   Q2: round robin with quantum 6

   Q3: round robin with quantum 8

   Q4: first come first served (FIFO)
4. I/O requests always take five units of time.
5. Assume no context switching overhead.
6. Process selection is strict. In other words, higher priority queues are emptied first.
7. Process selection is priority preemptive.
8. Whenever a regular process is preempted by a higher priority process, and is then scheduled again, it should get a whole new quantum, not the remainder of its last quantum. A preempted process is put at the end of its queue.
9. If a process in Q1, Q2 or Q3 uses its entire quantum without finishing its CPU burst, it is put at the end of the next lower queue. If the CPU burst is equal to the quantum, it does not move down, it is put at the end of its queue.
10. After returning from I/O, regular processes move up one level and are placed at the end of the queue. Processes from Q1 return to Q1, and batch processes return to Q4.
11. If a process blocks on I/O before its quantum has completely expired, the system immediately schedules another process.
12. The input data (e.g., files used to test the program) has the following format:

```
1 r 2 5 5 5
2 r 8 5 6 3
.....
5 b 40 2 5
```

where each line represents a process (number), type (r or b: for regular or batch), arrival time (integer value), and CPU bursts (integer values separated by spaces).

For extra credit, students are asked to study the effect of several parameters used with the multilevel feedback queue scheduler. The program should allow the user to choose between priority preemptive and non-priority preemptive. They are also asked to write a discrete simulation to study the performance of the scheduler under different quanta and different context switching times of 0.0005, 0.005, 0.01, and for time quanta of a) 3, 4, 6; b) 4, 8, 10 and c) 5, 8, 12. Needless to mention that MLFQ helped our students in the design, implementation and especially testing of their projects.

In the next section, we introduce PAGE, a package that contains more operating systems algorithms used for virtual memory management.

3 Page Replacement Algorithms

Most virtual memory systems use paging, where the virtual address space is divided into pages and the corresponding units in physical memory are called frames. The algorithms we implemented in PAGE are well explained in Tanenbaum et al. [5], which is the book we use.

When a page fault occurs and physical memory is full, the operating system removes a page from memory and moves in a new one. But which page should be chosen for removal? The optimal replacement algorithm would want us to remove the page that will not be used for the longest period of time. This algorithm guarantees the lowest possible page-fault rate for a fixed number of frames. But this algorithm is unrealizable since it would require from the operating system to have a crystal ball to be able to see the future and to determine which page will remain unreferenced for the longest period of time. Most of the existing page replacement algorithms are approximations of the optimal algorithm.

PAGE simulates the five page replacement algorithms depicted in Figure 4, which we now briefly describe.

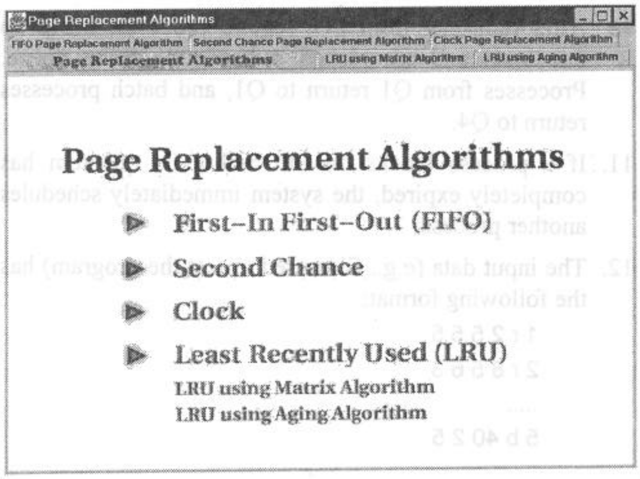
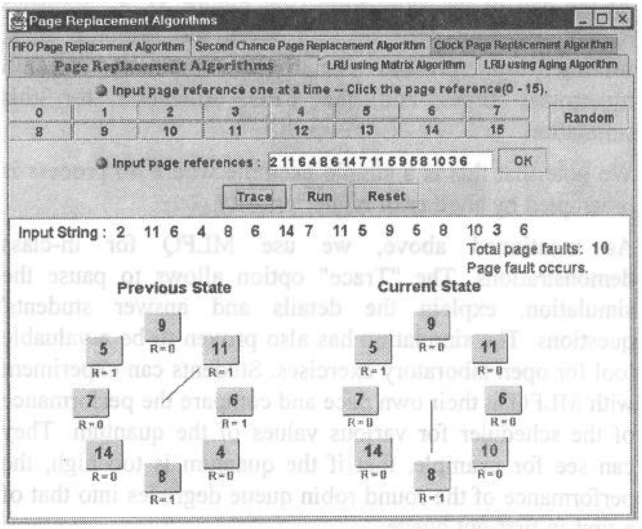

Page Replacement Algorithms
FIFO Page Replacement Algorithm
Second Chance Page Replacement Algorithm
Clock Page Replacement Algorithm
Page Replacement Algorithms
LRU using Matrix Algorithm
LRU using Aging Algorithm
Page Replacement Algorithms
First-In First-Out (FIFO)
Second Chance
Clock
Least Recently Used (LRU)
LRU using Matrix Algorithm
LRU using Aging Algorithm

Figure 4. The Five Algorithms of PAGE

In order to replace the page that has been in memory the longest, we associate with each arrival the time when that page was brought into memory. When we have a page fault and a page must be replaced, we choose the oldest one. With First-In First-Out (FIFO), the operating system maintains a queue, with the oldest page at the head of the queue and places the most recent arrival at the tail of the queue. On a page fault, the page at the head of the queue is removed and the new page is added at the tail of the queue.

The Second Chance algorithm is a modification of FIFO and checks the referenced bit of the page before swapping it out of memory. If the referenced bit is set (=l), then the bit is cleared (=O), the page is put at the end of the FIFO queue, and its load time is updated as though it had just arrived in memory. In other words, the page is given a second chance.

To avoid moving too many pages, the Second Chance algorithm can be implemented by using a circular list. In Figure 5, we assume that physical memory (main memory) has 8 frames (see the size of the clock) and the virtual address space has 16 pages (numbered O-15). As can be seen from the figure, students can input a list of page references in three different ways. They can choose among the 16 page references one by one, they can type their references in the text field, or they can randomly generate an input list by repeatedly clicking on “Random”.


Page Replacement Algorithms
FIFO Page Replacement Algorithm
Second Chance Page Replacement Algorithm
Clock Page Replacement Algorithm
Page Replacement Algorithms
LRU using Matrix Algorithm
LRU using Aging Algorithm
Input page reference one at a time -- Click the page reference(0 - 15).
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Random
OK
Trace
Run
Reset
Previous State
Current State
9
5
11
7
6
14
4
8
R=0
R=1
R=1
R=0
R=1
R=0
R=0
R=1
9
5
11
7
6
14
10
8
R=0
R=1
R=0
R=0
R=0
R=0
R=0
R=1

Input page references : 2 11 6 4 8 6 14 7 11 5 9 5 8 10 3 6

Input String : 2 11 6 4 8 6 14 7 11 5 9 5 8 10 3 6

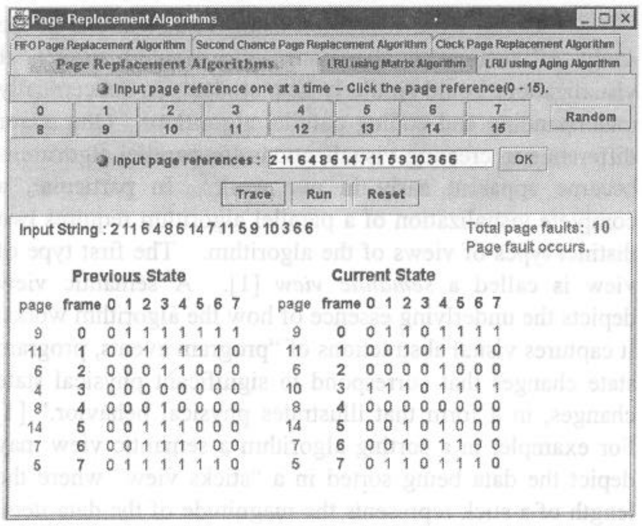
Total page faults: 10

Page fault occurs.

Figure 5. The Clock Page Replacement Algorithm

The scenario in Figure 5 illustrates how the Clock page replacement algorithm handles a page replacement. The algorithm has already taken care of page references 2, 11, 6, 4, 8, 6, 14, 7, 11, 5, 9, 5, 8 from the input page references. These requests give rise to 9 page faults and result in the clock depicted under “Previous State” in Figure 5, where the clock arm points to the frame containing page 11. The next request of page 10 is not in physical memory. A message “Page fault occurs” is displayed, as can be read in Figure 5, and a page has to be replaced. The algorithm inspects the frame containing page 11, clears its R bit (sets the referenced bit's value to 0) and advances to the frame containing page 6. Likewise, here too, the R bit is cleared and the clock arm advances to the frame containing page 4. Since the referenced bit of the frame containing page 4 is zero, that page is evicted and is replaced by page 10. We now have the clock depicted under "Current State" in Figure 5 and the page fault count is now 10.

When a page fault occurs, the Least Recently Used page replacement algorithm (LRU) evicts the page that has been inactive (unreferenced) for the longest time. LRU can be implemented with hardware. The Matrix Algorithm of Figure 6 maintains an n **x** n bit matrix, where n is the number of frames. Thus, if we consider the settings we have with the Clock algorithm, we have an 8 **x** 8 bit matrix that is initialized to zero entries. When a page in frame k is referenced, the hardware first sets all bits of row k to one, then sets all bits of column k to zero. At any given instant, the row whose binary value is lowest contains the page that is the least recently used and is the one to be evicted when a page fault occurs. If we consider the same scenario as the one we studied with the Clock algorithm, then frame 3 has the lowest binary value (zero) and therefore its content, page 4, is evicted and replaced by page 10, as can be seen in Figure 6.


Page Replacement Algorithms
FIFO Page Replacement Algorithm
Second Chance Page Replacement Algorithm
Clock Page Replacement Algorithm
Page Replacement Algorithms
LRU using Matrix Algorithm
LRU using Aging Algorithm
Input page reference one at a time -- Click the page reference(0 - 15).
0 1 2 3 4 5 6 7
8 9 10 11 12 13 14 15
Random
Input page references :
OK
Trace
Run
Reset

```
Input page references : 2 11 6 4 8 6 14 7 11 5 9 10 3 6 6

Input String : 2 11 6 4 8 6 14 7 11 5 9 10 3 6 6          Total page faults:  10
                                                          Page fault occurs.

      Previous State                    Current State
page  frame 0 1 2 3 4 5 6 7       page  frame 0 1 2 3 4 5 6 7

 9      0   0 1 1 1 1 1 1 1        9      0   0 1 1 0 1 1 1 1
11      1   0 0 1 1 1 1 1 0       11      1   0 0 1 0 1 1 1 0
 6      2   0 0 0 1 1 0 0 0        6      2   0 0 0 0 1 0 0 0
 4      3   0 0 0 0 0 0 0 0       10      3   1 1 1 0 1 1 1 1
 8      4   0 0 0 1 0 0 0 0        8      4   0 0 0 0 0 0 0 0
14      5   0 0 1 1 1 0 0 0       14      5   0 0 1 0 1 0 0 0
 7      6   0 0 1 1 1 1 0 0        7      6   0 0 1 0 1 1 0 0
 5      7   0 1 1 1 1 1 1 0        5      7   0 1 1 0 1 1 1 0
```

Figure 6. The LRU Matrix Page Replacement Algorithm

LRU using Aging Algorithm mentioned in Figure 4 is a software implementation of the LRU.

As is the case with MLFQ, we use PAGE during class time to demonstrate the workings of the various algorithms. The students also experiment with it in an open laboratory environment.

4 Conclusion

In this work, we present two packages MLFQ and PAGE that we use in our Operating Systems classes to explain the CPU scheduler and page replacement algorithms. The packages are valuable visualization tools for classroom lectures. Instead of tracing the algorithms by hand, or by overlaying transparencies, one can step through the programs, pause, consult "Previous State" snapshots, and answer questions. We also use the packages in an open lab environment, where students gain a better understanding of the workings of the algorithms by practicing at their own pace. They can experiment and input their own data, set their own parameters and compare the results. MLFQ is far from matching the large number of queues found in real operating systems, but nevertheless, it is closer to reality and yet not too large for tracing through the process with pencil and paper. We are aware that a four-queue model with two kinds of jobs is a far cry from the 160 priority levels (60 levels for real time, 40 for kernel and 60 for time shared processes) in UNIX SVR4 [6], or the 16 real-time priority classes and 16 variable priority classes that comprise the 32 multilevel feedback queues of Windows NT [1]. Nevertheless, we believe that MLFQ and PAGE do assist students in understanding the workings of CPU schedulers and page replacement algorithms implemented by modem operating systems, such as UNIX SVR4's two-handed clock page replacement algorithm. Moreover, our packages are easy to use and provide immediate feedback. The graphical interface is very efficient and enables the users to visualize every step of the algorithm. Help files are incorporated in the packages and contain information about algorithms and their implementation.

The two packages described in this work, MLFQ and PAGE, are available through a web site developed by the first author. The web site is located at:

http://www.mathcs.sjsu.edu/faculty/khuri/publications.html.

5 References

1. Custer, H., Inside Windows NT. *Microsoft Press*, 1993.
2. Lawrence, A., Badre, A., and Stasko, J. Empirically Evaluating the Use of Animations to Teach Algorithms. *Technical Report GIT-GVU-94-07* (Georgia Institute of Technology, Atlanta, GA, 1994).
3. McGrory and Migliore, CS240A: Operating Systems, Class Notes, Stanford University, Winter 1991.
4. Naps, T., Algorithm visualization served off the World Wide Web: why and how. *Proceedings of SIGCSE/SIGCUE'96,* 1996, pp. 66-71.
5. Tanenbaum, S. and Woodhull, A. Operating Systems: Design and Implementation. *Prentice Hall*, Second Edition, 1997.
6. Vahalia, U., UNIX Internals: The New Frontiers. *Prentice Hall*, 1996.