University of Toronto
CSC150H: Introduction to Computer Science
Fall 2002

Assignment 2: Page Replacement Algorithms

Introduction

How much do you know about the performance of your computer? You probably know what kind of processor you have (e.g., Pentium) and how fast it is (e.g., 850 MHz). You probably also know how fast your modem transfers data (e.g., 56 Kbps). If you have looked into other hardware specifications, you may also know the peak transfer rate of your hard drive, floppy drive, or CD-ROM. But, the performance of an application depends on how well the operating system can deliver the capabilities of the hardware to the application, not just on the raw hardware speed.

The operating system is the software interface that controls access to all of the hardware devices: floppy drive, CD-ROM, hard drive, modem, etc. Some of the responsibilities of the operating system include

- determining which application should be running,
- intercepting key presses and mouse movements and passing those events to the application,
- deciding how to divide up the memory among the currently running applications.

The operating system plays a major role in making us feel like we are getting good (or bad) performance from our computer.

This assignment introduces you to some algorithms that the operating system uses to manage the computer's memory. More specifically, you will investigate two algorithms that are used by operating systems to control which parts of the currently running programs stay in memory.

Background

The memory requirements of a program can be divided into three parts: the code, the data (both statically and dynamically allocated data), and information about the program's current state (runtime stack). As a program executes, it may refer to any part of this memory. Figure 1(a) shows the memory requirements of a program broken into these three categories. However, not all of this needs to be resident in physical memory (the computer's memory) the whole time a program is running. In fact, for many programs, only a small part of the program needs to be in memory at each point in time while the program runs. The rest of the program is left on the disk until it needs to be brought into physical memory.

To make it possible to keep smaller pieces of a program in memory, we need to divide up the memory required by a program in some way. The method that modern operating systems use is to divide memory into fixed-size chunks called pages (usually between 4 and 8Kbytes each). Then the system can decide for each page of a program whether to bring it into memory or not. In Figure 1(b), the program requires 15 memory pages, but only pages 2, 5, 6, 7, 11, and 12 are in the computer's memory. If the operating system is clever about which pages it keeps in memory, it can quickly switch between programs and run programs larger than will fit in main memory without the user noticing.

Now the operating system has two problems: it must decide which pages to bring into memory, and when the memory fills up, it must decide which pages to eject so that it can bring in new ones. It takes time to bring a page into memory, so we want to minimize the number of times we bring each page into memory. (If a page gets ejected we might have to bring it into memory again later.) The first problem is relatively easy to solve: every time a program references a page, bring that page into memory (called demand paging)[1]. This solution is the main one used by modern operating systems.

[1] The mechanisms used to accomplish demand paging are beyond the scope of this assignment.

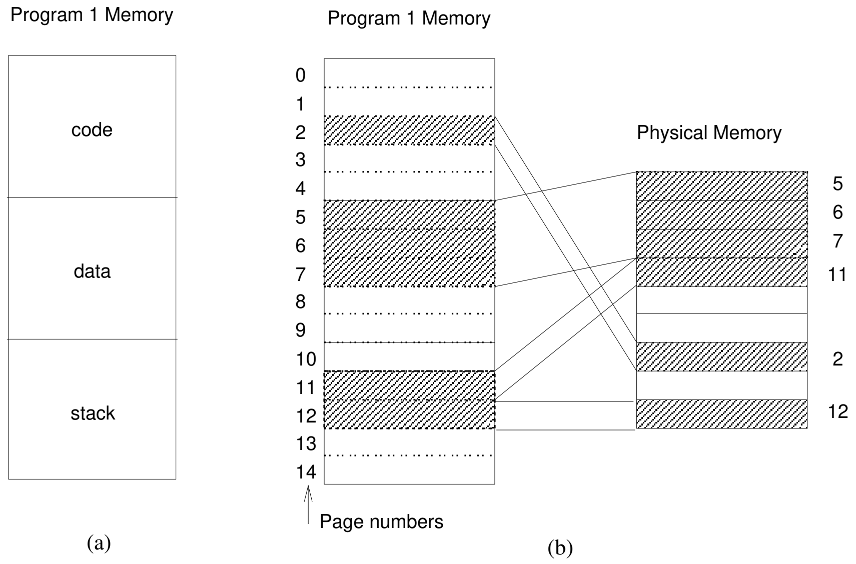

Figure 1: Program Memory Requirements

The second problem, that of deciding which page to replace when a new one is needed, is more complicated. We would like to replace a page that will not be needed again. To do that though, we would need to be able to see into the future. Since we cannot do that, we must guess which page might not be used again, or at least might not be used for a long time. In this assignment, you will implement and compare two algorithms for replacing pages.

Page Replacement Algorithms

The goal of a page replacement algorithm is to choose pages to replace such that the number of times each page must be brought into memory is minimized. We say that a memory access to a page that is not in memory is a *page miss*. A *page hit* occurs when a memory access is to a page that is in memory.

By keeping track of the number of page hits and misses, we can compare our two algorithms. The one that has the fewest page misses, is a better algorithm because the operating system spends less time transferring pages from disk into memory. We usually use the *miss ratio*, the number of misses divided by the total number of references, as a measure of a page replacement algorithm's performance.

Comparing algorithms

How can we tell which algorithm is better in practice? One way to do it is to implement both algorithms and put them into the operating system of two different computers and ask the users what they think. This isn't very scientific or practical, so we would like to test them outside of a real operating system.

We will use a trace of the memory accesses of a program, i.e., a list of the pages that the program accesses while it is running. For example, if a program first accesses page number 3, then accesses page 8, then page 10, then page 3, and then page 20, our memory reference trace would simply be the list of pages: 3, 8, 10, 3, 20. Using the memory reference trace we can simulate the transfer of pages to and from memory. This memory reference trace is used as an example in Figure 2.

Both the size of available memory and the page replacement algorithm will affect the miss ratio.

We will use an extended queue ADT to simulate the storing of pages in memory. If a page number is in the queue, then that page is in memory.

Memory reference trace	Queue contents when FIFO is used	Queue Contents when LRU is used
3	3	3
4	3 4	3 4
10	3 4 10	3 4 10
8	3 4 10 8	3 4 10 8
3	3 4 10 8	4 10 8 3
20	4 10 8 20	10 8 3 20

Figure 2: An example demonstrating the FIFO and LRU page replacement algorithms

The FIFO algorithm

The page that has been in memory the longest might be a good candidate to replace. We can keep track of pages in memory by using a *first-in-first-out* (FIFO) queue. When a page is brought into memory it is placed at the end of the queue and the page that is replaced is taken from the front of the queue. Figure 2 shows what the contents of the queue are when the example memory reference trace is used as input. Note that when page 3 is accessed for the second time, the contents of the queue do not change, because page 3 is in memory. When page 20 is accessed, the queue is full, so the page at the front of the queue (page 3) is removed, and page 20 is added to the end of the queue. The size of the queue is the number of pages that will fit in memory. In this example, only 4 pages can be in memory at the same time.

The LRU algorithm

Under some circumstances that are often seen in practice, the FIFO algorithm does not seem to choose good pages to replace. Another way to guess which page might not be used for a long time is to choose that page that was last accessed the farthest back in the past. We call this the *least recently used* (LRU) algorithm. The rationale for choosing the least recently used page is, if it hasn't been accessed in a while then it may not be accessed again. We use a modified queue ADT to keep track of which page is least recently used. Each time a page is accessed we check if it is in the queue and remove it and place it at the back of the queue. Therefore, the the least recently used page is always at the front of the queue. Do you see why? Figure 2 demonstrates the LRU algorithm using our example memory reference trace. This time, when page 3 is accessed again, it is removed from the queue and reinserted at the back of the queue.

Implementation

Both the FIFO algorithm and the LRU algorithm will use a modified queue to keep track of which pages are in memory. We need a modified queue because we also need to check to see if a page is in the queue, and we may need to remove a page from the middle of the queue. Your `LinkedQueue` class will implement the `ExtendedQueue` interface which itself extends the `Queue` interface.

Your program will have three classes in addition to the main driver program which will be called `Experiment`: a `LinkedQueue` class, a `FIFOPager` class, and an `LRUPager` class. Both of the pager classes will use the `ExtendedQueue` ADT. The two Pager classes will implement the `Pager` interface. See the comments in `Pager.java` for details on each of the methods.

The main method in `Experiment.java` will take a file name as a command line argument. The file will contain a list of hexadecimal page numbers that represent the memory reference stream. There is one page number per line of the file. The main method will read in the memory reference stream and register

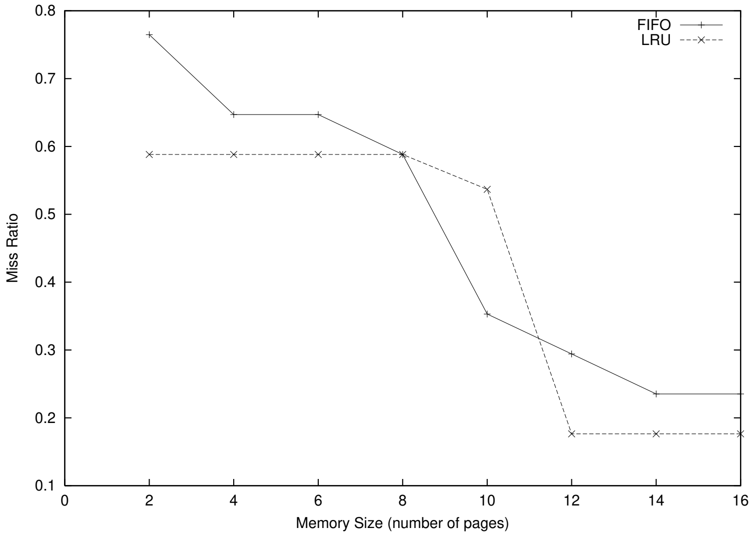

Figure 3: A plot of the miss ratios for the colmat trace.

each reference with the Pager classes. You should set up your main method so that it runs both the FIFO and LRU pagers on the memory reference stream using a set of memory references. How you organize the program to do this is up to you.

The output of your program will be results in 3 columns: memory size, FIFO miss ratio, LRU miss ratio. You will need to experiment to find appropriate values for the memory sizes. If the miss ratio is always 1.0, then you need to try larger memory sizes. If the miss ratios are all the same, then you need to try a wider span of memory sizes. The number of unique pages is the upper bound on the memory size. Set up your experiments so that there your final results contain miss ratios for no more than 8 memory sizes.

Sample output might look like the following. The first column is the physical memory size, the second column is the FIFO miss ratio, and the third column is the LRU miss ratio

```
2 0.7647058823529411 0.5882352941176471
4 0.6470588235294118 0.5882352941176471
6 0.6470588235294118 0.5882352941176471
8 0.5882352941176471 0.5882352941176471
10 0.35294117647058826 0.5367647058823529
12 0.29411764705882354 0.17647058823529413
14 0.23529411764705882 0.17647058823529413
16 0.23529411764705882 0.17647058823529413
```

Your task

1. Implement the `LinkedQueue` class. You must use a linked list to implement this class. Note that much of the code for this class has been developed on the course slides. You are welcome to use and modify that code. Remember that if you use code from a text book, you should say where you got it from.

2. Implement the two paging algorithms, each as their own class: `FIFOPager` and `LRUPager`. Also, implement the main program in the class `Experiment` that reads in an input file of memory references and passes them to the pager class.

3. Produce results for mm16, sort1, and sawtooth. Hand in the output for these traces. For these traces, create graphs similar to Figure 3. If you don't have access to a graphing program, you can draw the graphs on graph paper.

4. Answer the following questions (a short paragraph for each is sufficient):
   (a) Will the miss ratio ever become zero? Why or why not?
   (b) What characteristics of a memory reference trace would cause the FIFO algorithm to perform poorly (i.e., have a high page miss count)?
   (c) What characteristics of a memory reference trace would cause the LRU algorithm to perform poorly (i.e., have a high page miss count)?
   (d) If you had to choose either the FIFO or the LRU page replacement algorithm to put in your operating system, which one would you choose? Why?

Testing

The testing component of this assignment has two parts: testing the queue class, and testing the pager classes.

Rather than handing in a test class, you will write a short report describing a test strategy for the LinkedQueue class and for the pager classes. The test strategy is a list of all of the test cases, and a explanation of what each test case is intended to demonstrate. Your test cases should convince the reader that your program will work in all cases. The explanations should be brief but precise. Your report should be typed, and must be written using proper English.

We have provided a few real memory reference traces that you may use to try out your algorithms on. You can download these traces from the course web page. This set of traces is not meant to be a complete test suite.

What to hand in

Hand in on paper (location TBA)

1. Answers to the questions in part 4.
2. A printout of the code for parts 1, and 2. (Do not hand in the code that was provided to you: Pager.java, Queue.java, ExtendedQueue.java.)
3. Your test strategy.
4. The results and graphs for the specified 3 traces.

Also, submit your code electronically including Experiment.java, FIFOPager.java, LRUPager.java, LinkedQueue.java and any other classes you want to write. Do not need to submit Pager.java, Queue.java, or ExtendedQueue.java.

Tentative Marking Scheme

Testing	10
Correctness	30
Programming Style	5
Comments	5
Questions	8
Output and Graphs	2
Total	60