Seth Cattanach
CSE-34341 Operating Systems
Assignment 4: Paging Algorithms
Due: 11 April 2019

I. Overview

This project implements and demonstrates six different algorithms used for page replacement and compares their performance by simulating page references and measuring the number of page faults. Five of the algorithms implemented were discussed in class:

(1) Optimal, (2) First-come first-served, (3) Second Chance, (4) Not Recently Used, (5) Aging

I designed the sixth algorithm, called “wait and confirm.”

The following files should be included in the **.zip** package:

- **paging.py** Source code for the assignment
- **test-references.txt** 50-line sample of page references; not randomly generated
- **references1000.txt** 1000-line sample of page references; randomly generated
- **report.pdf** This report

II. Running the Program

This program was written with Python 3. The user must set necessary parameters (lines 320-324):

- **generate_random_file = [True/False]**
  - Use **True** to generate a random test file (can specify number of references, max page number, etc. below). Use **False** to use the file specified by **default_file_name**
- **default_file_name = "String"**
  - The test file containing page references (tab-separated format). The default file "test-references.txt" is included in the project files
- **page_table_size = int**
  - Specify the number of page frames in the page table. In general, a smaller page table size will lead to more page faults, and vice versa
- **num_references = int**
  - Specify the number of page references to randomly generate. Only needed if generate_random_file is set to **True**
- **max_page_number = int**
  - Specify the range of page numbers the randomly-generated file will use (that is, all randomly-generated referenced will references pages from one to this value). Only needed if generate_random_file is set to **True**

When the program is run, the algorithms will execute in the following order and print out the number of page hits, page misses (aka page faults), and the page fault percentage.

III. Implementation Details

Optimal: When choosing a page to evict, this algorithm performs a linear search of all future page references and evicts the page that will be referenced the latest of all the pages currently in the page table. If there are multiple pages that will never be referenced again, the algorithm selects the first such page to evict (using an OrderedDict container to simulate the page table). This algorithm is obviously unrealistic in real life, but it serves as a benchmark to measure the performance of other algorithms.

First-come first-served (FIFO): When choosing a page to evict, the FIFO algorithm always selects the oldest page in the page table. An OrderedDict container was used to implement the page table.

Second Chance: This algorithm is a modification of the FIFO algorithm and gives every page two chances before being evicted. When choosing a page to evict, this algorithm looks at the oldest page in the page table. If its "second chance" bit is set to 0 (default), the algorithm sets this bit to 1 and moves the page to the end of the page table so it won't immediately be scheduled for eviction again. An OrderedDict container was once again used to implement the page table.

Not Recently Used: This algorithm is sets "referenced" and "modified" bits to determine which page should be evicted upon a page fault. Whenever a page is referenced (a page reference hit) from the page table, the "referenced" bit for that page is set to 1, and if the reference type was a "write" and not a "read," the "modified" bit is also set to 1. When choosing a page to evict, this algorithm selects a page from the *lowest page class* available: pages that have not been referenced or modified (00) are in class 0, pages that have not been referenced but have been modified (01) are in class 1, pages that have been referenced but not modified (10) are in class 2, and pages that have been referenced and modified (11) are in class 3. To implement this algorithm, I appended two bits to each page table reference (initialized to 0) and updated them accordingly on each page reference. Since the page table was implemented with an OrderedDict container, the algorithm will select the first (oldest) page to evict if multiple pages exist within the lowest page class.

Aging: The aging algorithm is a software simulation and modification of the "least recently used" algorithm. It ensures pages that were referenced often and recently are less likely to be evicted, while pages that are not often referenced or have not been referenced for a long time are more likely to be

evicted. With hardware, this would be implemented by shifting a string of bits. However, to simulate this process in a simple manner, I used a counter: each time a page is referenced in the page table, the algorithm adds an arbitrary value (50) to its "aging" value, and the algorithm subtracts 1 from the aging value of all other page table entries. This ensures a page that has not been referenced in the last 50 page references is likely to be evicted and preserves pages that have been recently referenced. This algorithm also uses an OrderedDict container, which means the oldest page will be evicted when two pages have an identical aging value.

Wait and Confirm: This algorithm creates a small, temporary list to hold page references before putting them into the full page table. If a page is referenced from the list, it is said to be "confirmed" and a special bit is set to 1; this page is then added to the page table and removed from the temporary list. If a page is added to the temporary list but not referenced in time, it will still be added to the full page table, but its "confirmed" bit will still be set to 0. The algorithm then chooses the oldest "non-confirmed" bit ("confirmed" bit value = 0) in the full page table when it needs to evict a page. The rational behind this algorithm: a second page reference to the same page within a reasonably close number of references likely indicates the page is useful and worth preserving. If a page is only referenced once, it will still be added to the temporary table and eventually the full page table, but it will have higher eviction priority compared to pages that were "confirmed" while staged in the temporary list. An OrderedDict container was again used to implement the page table, and a standard Python list was used to implement the temporary page table.

Wait and confirm has a major advantage over other page algorithms whenever high-frequency pages are referenced close together; in these cases, the confirmation bit ensures important pages are less likely to be evicted. Its major disadvantage comes when high-frequency pages are all referenced at evenly-spaced intervals; in these cases, the confirmation bit will likely not be set, and this algorithm will perform much like the first-come, first-served algorithm.

IV. Benchmarking and analysis

I benchmarked these six algorithms according to one metric – number of page faults – using both randomly-generated and human-generated test files. For the randomly-generated page referenced, five trials were performed (with different page references each time) and the results were averaged. I used the following parameters for these random tests:

page size = 100, number of references = 1000, max page number = 150

The performance for the random benchmarks is **reported as the percentage of page faults** (smaller percentage = better performance). The results are shown below in *Figure 1*:

Optimal	FIFO	Second Chance	NRU	Aging	Wait & Confirm*
29.7%	37.9%	36.8%	37.8%	37.1%	37.0%
31.4%	39.2%	38.7%	37.0%	37.5%	39.1%
30.2%	36.6%	37.0%	37.6%	35.4%	35.9%
29.4%	39.6%	38.5%	38.8%	39.1%	38.9%
31.0%	38.9%	39.0%	38.7%	38.1%	37.4%
30.3%	*38.4%*	*38.0%*	*38.0%*	*37.4%*	*37.7%*

Figure 1: benchmark results (page fault percentages) for the six algorithms using randomly-generated page references. Five trials were conducted, each with a different set of page references, and the averages are reported in the final row.

* I designed the “wait and confirm” algorithm; see “Implementation Details” (above) for more info

Conclusions drawn from these results: the optimal algorithm was, as expected, easily the best, consistently causing the fewest number of page faults. The other algorithms all yielded similar results; this seems to be due to the fact I used *random* page references. Each algorithm is designed to take advantage of page table locality in some way, and implicit in that assumption is that *pages that were referenced heavily in the past will continue to be referenced more heavily in the future*. However, this is not true for random page references – each page is equally likely to be referenced at any given time

(although the *random* algorithm is pseudo-random, which may be why there are still minor variations in performance across the last five algorithms).

With this in mind, it's clear that random page references alone won't provide a lot of information about the performance of each algorithm, so I created my own test file to simulate a more realistic set of page references (where some pages are referenced many times in a row, for instance). I used the following parameters for this experiment:

page size = 5, number of references = 50, max page number = 10

The performance for this human-generated set of page references is again reported **in terms of page fault percentage**, and the results are shown below in *Figure 2***:**

Optimal	FIFO	Second Chance	NRU	Aging	Wait & Confirm*
44%	52%	48%	52%	44%	48%

Figure 2: benchmark results for the specified test file (**test-reference.txt**) as a percentage of page faults.

* Again, the "wait & confirm" algorithm is my own design – see "Implementation Details" for more info

Figure 2 suggests a little more differentiation between algorithmic performance exists than in the randomly-generated benchmarks. **Aging** was the only algorithm that matched **optimal** in terms of minimizing page faults on this small test set. **Wait and confirm** performed better than both **NRU** and **FIFO** – also expected, since FIFO doesn't account for reference locality at all, and because **NRU** is hindered since the "read/write" reference bits were still set semi-randomly (i.e. not intentionally). **Wait and confirm** also performed equally well as **second chance**, both of which yielded slight improvements over the **FIFO** algorithm. It was also expected that **second chance** would have equal or better performance than **FIFO**, since it allows heavily-referenced pages to have a second chance before getting evicted from the page table.

Obviously, these results depend heavily on the test file being used. Using randomly-generated files allowed me to test the algorithms quickly and with many different parameters, but it had one major drawback – it was difficult to distinguish algorithm performance, because these algorithms all (to some degree) make predictions about which pages are likely to be referenced based on prior reference history, and no such assumptions can be made about random references. The human-generated file provided a clear look at performance, but the sample size was limited – it took significant time to create a realistic set of references even for a small test file (50 references).