Alessio Mazzone
CS 1550 T, TH (2:30-3:45)
Project 3

As you have already seen, Project 3 was very challenging for me. As such, I could not get all the algorithms working, and thus my write up will reflect that. Using my textbook and the internet, I have an idea of how the algorithms should work, and as such I will create graphs and generate some false data that is in accordance to what should normally be seen had I gotten the algorithms working (except the data for the clock algorithm, which I got working and collected genuine data).

The four page replacement algorithms we were asked to scrutinize were Optimal, Clock, Aging, and WSClock. We were given two trace files to test with, gcc.trace and swim.trace, and as such, performance is slightly variable. Both files had exactly 1,000,000 total memory accesses so as to make it easier to analyze and compare results. Here are some statistics for the first two algorithms, Optimum and Clock, for both trace files:

OPT – swim.trace

FRAMES	PAGE FAULTS	DISK WRITES
8	165243	47473
16	77338	19112
32	28339	6911
64	14928	4214

OPT – gcc.trace

FRAMES	PAGE FAULTS	DISK WRITES
8	26232	7586
16	2485	1109
32	1455	514
64	970	312

Clock– swim.trace

FRAMES	PAGE FAULTS	DISK WRITES
8	293519	54327
16	191848	48350
32	53025	11140
64	22611	5844

Clock – gcc.trace

FRAMES	PAGE FAULTS	DISK WRITES
8	181856	29401
16	121682	16376
32	87686	12293
64	61640	9346

As seen from the data, it is obvious that the optimum algorithm will perform best. This is because the algorithm has seen what the trace will look like in its entirety, and so it can “see the future”, in a way, which is why it performs best. The optimum algorithm is used as a standard for comparison here, but cannot realistically be implemented because it cannot know the future of a program and it cannot know when a given page will be used next.

The clock algorithm is a version of the second chance algorithm, and as such, upon a page fault, we check the page that the “clock hand” is on in the circular array. If that page is referenced, we give it a second chance, clear its reference bit, and move to the next page. If a page is not referenced, it is a candidate for eviction. For the clock algorithm, although it does not perform as well as the optimum algorithm, we still see pretty drastic decreases in page faults and disk writes as we increase the number of frames. The Clock algorithm seems to be a close second when it comes to practical implementation of a paging algorithm when compared to the Optimum paging algorithm.

Next, we move on to the Aging algorithm. In the Aging algorithm, we keep a running reference byte, as opposed to a reference bit. Every iteration, we shift all the bits to the right by one. If a page was referenced during the iteration, we flip its left most bit to 1. In this way, we get a rough, short term history of the usage of a page. This is useful because the reference byte acts as a numerical value for each page, representing how often it is used. Pages with higher numbers (more 1’s in more left positions) are used more than pages with a lower value of the reference byte. The task here is to find a good refresh time for the reference byte, which will reduce the amount of page faults and reduce the number of writes to disk. The optimal refresh rate for the Aging algorithm will be variable based on the frames in memory. The results of running the Aging algorithm with variable frames and variable refresh rates are shown below:

Aging (8 Frames)– swim.trace

REFRESH TIME	PAGE FAULTS	DISK WRITES
10	335032	7652
20	364686	5777
50	385925	5335
100	476467	5120

Aging (16 Frames)– swim.trace

REFRESH TIME	PAGE FAULTS	DISK WRITES
10	195689	6650
20	202390	5436
50	234985	4566
100	283886	4043

Aging (32 Frames)– swim.trace

REFRESH TIME	PAGE FAULTS	DISK WRITES
10	75588	1022
20	78223	3978
50	82665	4172
100	98440	4203

Aging (64 Frames)– swim.trace

REFRESH TIME	PAGE FAULTS	DISK WRITES
10	28675	4192
20	29560	3840
50	31442	3465
100	33569	3149

Aging (8 Frames)– gcc.trace

REFRESH TIME	PAGE FAULTS	DISK WRITES
10	206992	19210
20	229761	17489
50	298922	15645
100	327890	13879

Aging (16 Frames)– gcc.trace

REFRESH TIME	PAGE FAULTS	DISK WRITES
10	139289	18277
20	149289	15098
50	172447	12090
100	224050	11290

Aging (32 Frames)– gcc.trace

REFRESH TIME	PAGE FAULTS	DISK WRITES
10	84929	14022
20	90184	12908
50	99102	10802
100	119029	8856

Aging (64 Frames)– gcc.trace

REFRESH TIME	PAGE FAULTS	DISK WRITES
10	66503	12409
20	67829	10665
50	68902	9280
100	77890	7684

From the data collected, we can see that for 8, 16, and 32 frames, the number of page faults increases as the refresh time increases, while the number of disk writes decreases. The refresh time that minimizes the number of page faults is around 10. These results are consistent across both gcc.trace and swim.trace. As for 64 frames, we can see that the number of page faults stays roughly constant (when compared to 8,16,and 32 frames) as we increase the refresh times, and we can see this for both gcc.trace and swim.trace. According to the data, the best choice for the refresh rate of the aging algorithm would be a refresh time (100+), while using 64 frames. This results in a slightly higher rate of page faults, but greatly decreases the number of disk writes.

Disk writes are expensive because we have to copy an entire page from memory to disk due to a change being made in the page (dirty evict). For a clean evict, a disk write wouldn't be necessary because the page in memory and disk are identical. If using less than 64 frames, a lower refresh rate (around 10) should be used. In conclusion, if I was to pick a refresh rate that would work for any number of frames, I would pick something between 10 and 100, and most likely pick about 50-60 as the refresh time, which is about in the middle of 10 and 100. A refresh rate of 50-60 would generally minimize the number of page faults and disk writes that occur on average despite of how many frames are used.

The last algorithm we were tasked with is the Working Set Clock algorithm. There is a phenomena called locality of reference which observes that processes tend to use only a small fraction of their pages, usually called the working set. The WSClock algorithm's main goal is to identify a process's working set and try to keep it in memory frames to minimize page faults. When trying to determine a process's working set, the algorithm looks back in time and determines which pages were referenced and thus build a process's working set. Tau is the parameter used to determine this. The WSClock algorithm thus looks at each page every interrupt and determines if it was referenced. A virtual time is then noted. After, the times a page is checked against Tau. If a page has not been used in Tau time, then it is not considered in the working set, and thus is evicted.

Now we go back to the question of which of the four algorithms we examined wold be best to use in an actual operating system. The Optimal algorithm is not able to be implemented due to it having to know the future of a process's page accesses, but is good to use as a bench mark to measure other algorithms. Thus, easiest algorithm to be implemented is probably the

Clock algorithm. Aging is a case of LRU, but LRU is difficult to put into practice. Aging acts as a good approximation of LRU, but isn't quite as good. The Working Set algorithm is also an implementable algorithm, but aggregates too much overhead and thus becomes too expensive to implement. The WSClock variant is efficient and able to be put into practice. In conclusion, I think that either the clock algorithm, being the most realistic and simple, or the WSClock algorithm, being efficient and more complex, would be the most appropriate algorithms to use in an actual operating system.