Operating Systems Project 2

Logan Lopez Arnold Lestin

March 4, 2019

1 Introduction

As we try to keep an illusion that there is an unlimited amount of space, we have to make use of pages for keeping track the parts of memory we write to disk. There are various approaches to this magic trick, each with varying performance. Here we will analyze 3 different forms of page replacement, which will be Least Recently Used (LRU), First In – First Out (FIFO), and the VMS Second Chance Replacement Policy (VMS). We are given miniaturized SPEC benchmark files which is a very simple format. Each line represents a memory access, and the first column being the hexadecimal memory address and then the second column being an uppercase character either a R or W which represents read or write, respectively. We write a C program which makes use of some simple data types such as queues to then simulate the memory and be able to collect statistics such as disk reads and writes from the approaches. This is important as we which to minimize both of these.

The reason why we wish to minimize disk reads and writes is because it is much slower to access than the memory, being a memory hierarchy. While this may be untrue of an Solid State Drive (SSD) which is solid state memory (which is much faster), a write to the disk is still unwanted as it does cause unneeded wear due to the nature of writing to SSDs. In addition the time it takes to go across the bus lines is minuscule for 1 or 2 accesses, but going further it adds up quickly when dealing with memory paging. Paging is a hack we wish to avoid, and these algorithms take advantage of two special properties with memory access. These are spatial locality which means a program is more likely to access memory spaces directly next to the current access, such as a for loop iterating through an array, it is reasonable to assume the next part of the memory is a prime candidate for future access. There is also temporal locality, which means that something that was last access will most likely be accessed again. Least Recently Used is more geared towards temporal locality for this exact reason, that it will start replacing the entries which are not prime candidates to be written again, as they haven't been used in a while. FIFO doesn't take advantage of any type of locality, but is easy to implement because it can be essentially represented as a queue. VMS is a much more complicated approach, but one that does give more flexibility to programs that do not immediately access the same memory spaces, hence "second chance".

We implement these using the `sys/queue.h` library which is the only library for queues in C. The documentation for it isn't the best and leaves some of the nuances on how to implement the struct as an exercise to the reader. However we had tried in the beginning implementing our own double ended queue (deque), which cause some runtime errors. The

use of this library greatly reduced the complexity and increased the readability in our code. The main datatype we used from this was a TAILQ or a tail-ended queue which is doubly linked. There is also Linked Lists and singularly linked datatypes which take on different and interesting names such as SLIST. We use global variables to keep track of reads writes and other statistics, which makes it easier when passing data between the mandated functions fifo, lru, and vms and the helper functions we defined ourselves.

2 Methods

We take a three-pronged approach to our memory simulation program, accounting for three different scenarios. We consider frame sizes which are very small to create a lot of misses as the page is constantly having to replace entries to make room for others. We'll say this is a size of any power of 2 up to 32, which is explained next on how we arrived at this number. Then also a "middle ground" where the page frame size an average size for normal processes, which would be around page size of 32 which would give the process around 100 MB in memory. This is the average size of a native C process (such as the speedy yet versatile editor Vim).

Then we will test it with a much larger value, which we will categorized as above 64, which is around 256 MB in memory, which may not seem small but a normal 4 GB would only be able to handle 16 processes with this amount of memory (comfortably). Computers normally run hundreds of processes at once with various daemons or services running in the background for everything such as handling file-system reads and writes, to a process to serve out web pages to connecting clients. Even having an encrypted hard drive (in software) is handled by a process, which means that we do need to be wary of how much memory we are giving out at once. Otherwise we'll be writing to the disk all day, which will slow down the program drastically, potentially even fill up the hard drive (although this is more a problem of the past), and cause unnecessary wear.

3 Results

We are only as perfect as our timing, and the timing itself has overhead, but as seeing that it is generally accepted practice to simply surrounding the benchmarked function with the gettimeofday() function. Also another thing we have to account for is the correct datatypes, I chose a double which is the most accurate native datatype that we can use to calculate averages for the sample sizes. Another issue is that we're limiting the CPU to show that these processes are being actually switched out instead of running in parallel, but real computers almost will never do it this way. So the applicability it has to the real world is somewhat diminished but it is the best we have to work with to show the difference in timing between the two.

Table 1: System call time

Trial	Time (in microseconds)
1	1.165110
2	0.507310
3	0.476000
4	0.490310
5	0.497790
6	0.496190
7	0.522950
8	0.500160
9	0.483730
10	0.486460
Average	0.562601

Table 2: Context switch time

Trial	Time (in microseconds)
1	533.000000
2	406.000000
3	381.000000
4	463.000000
5	382.000000
6	365.000000
7	357.000000
8	372.000000
9	374.000000
10	365.000000
Average	399.800000

4 Conclusion

The statistics we gathered from our simulations mainly matched what was hypothesized, that paging does help up to a point, and then after that point the optimization sort of plateaus. However smaller page frame sizes definitely show a negative impact on the system across any type of approach. VMS performed the best as it is the one with the most complexity in terms of allowing processes to reclaim memory that was about to be written to disk, helping prevent a "miss" to the hard disk. LRU came in second which is to be expected as it takes advantage of the previously mentioned spatial locality. While FIFO came in last (but not by as much as we expected on the bigger sizes) because it is the most simplest "quick and dirty" (pun not intended) page replacement algorithm aside from randomly evicting page entries.