Alberto Lucas

03/30/2021

CST334

Lab 4 Report

FILES TURNED IN

- fifo.c – The source code of the requested program
- makefile - The makefile for fifo.c
- test1head.jpg – A screenshot showing the program in action (only the first few lines)
- test1tail.jpg – A screenshot showing the program in action (only the last few lines)
- lab4report.docx – This file (lab report).

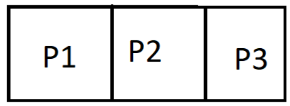
PROGRAM DESCRIPTION

Implementation:

This program uses a cache and a text file of page requests to simulate how an OS replaces pages in memory. To do this, the program contains an array called "cache" to store the most recently seen pages. An array data structure was used (over a linked list) because it performs better during searches (e.g., searching the cache) and because it is simpler to implement. However, this is at the cost of slower insertion/deletion times. Considering that this program does not insert or delete (instead, each page number is replaced), the main cost incurred by using an array is storage space. Each page is stored in a struct called ref_page, which contains all the attributes of the page (in this case, only the page number). Every time a page request is received (single number from accesses.txt), the program checks if the requested page is valid (greater than or equal to 0). If it is, then it is counted as a valid page request. If not, it is ignored. Next, the program scans the cache to see it the page exists there. If it does, nothing else happens (this is a cache hit). If it is not found in cache, then the page number is printed and the page number is inserted in the cache, replacing the one that was "first in". To keep track of which page was "first in" the cache, a counter called fifocounter is used. This counter simply cycles from 0 to C_SIZE, always keeping track of the next page number to replace. Once the pages are changed, the total faults counter is increased by one and the program repeats until no page requests are left (EOF). At this point, a statistics summary is displayed showing the total valid page requests and page faults.

Visual description:

A simple representation of how the program runs can be seen below. Imagine a cache size of 3 units with the following pages (assume the pages were put into memory in the following order -> p1, p2, p3):



C_SIZE = 3

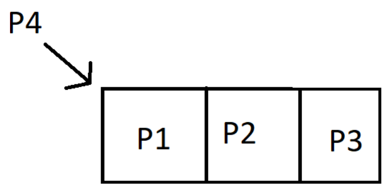
validPageRequests = 3

fifocounter = 0

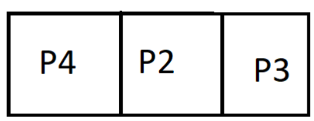
totalFaults = 3 (from filling up the cache)

If another page, P4, is requested, then P1 is replaced (it was the first one into the cache), the fifocounter is incremented, validpagerequests is incremented, and totalfaults is incremented. Then, the algorithm causes the cache and variables to change as shown below:

P4 is not found in the cache, so it will replace the page that was "first in."



After P4 is in the cache, the variables change as follows.



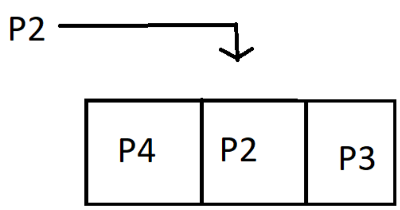
C_SIZE = 3

validPageRequests = 4

fifocounter = 1

totalFaults = 4 (from filling up the cache)

Note that fifocounter always cycles back to 0 when it reaches C_SIZE. Upon a cache hit, nothing happens, and the program continues onto the next page request as shown below:



C_SIZE = 3

validPageRequests = 5

fifocounter = 1

totalFaults = 4 (from filling up the cache)

Note that fifocounter does not change. This is because P2 was not replaced (it was simply found in memory). Therefore, from the perspective of the OS, P2 was still "first in" the cache. This is a limitation of the FIFO algorithm, as it does not consider the frequency of requests or the recency of requests. It simply checks which page was "first in" the array, and performs its logic based on that.

Test Runs at different cache sizes:

The following shows a test run of the program using cache sizes of 10, 300, 600, 900, 1200, and 1500 pages. The data from these runs is then plotted onto a graph to analyze any trends resulting from variations in cache size. The file accesses.txt was used as a stream of page requests to the program.

<u>Cache size of 10:</u>

```
******STATISTICS*******
10000 Valid Page requests
9916 Total Page Faults
```

Hit rate: ((10000-9916)/10000)*100 = **.84%**

<u>Cache size of 300:</u>

```
******STATISTICS*******
10000 Valid Page requests
7051 Total Page Faults
```

Hit rate: ((10000-7051)/10000)*100 = **29.49%**

<u>Cache size of 600:</u>

```
******STATISTICS*******
10000 Valid Page requests
4207 Total Page Faults
```

Hit rate: ((10000-4207)/10000)*100 = **57.93%**

<u>Cache size of 900:</u>

```
******STATISTICS*******
10000 Valid Page requests
1671 Total Page Faults
```

Hit rate: ((10000-1671)/10000)*100 = **83.29%**

<u>Cache size of 1200:</u>

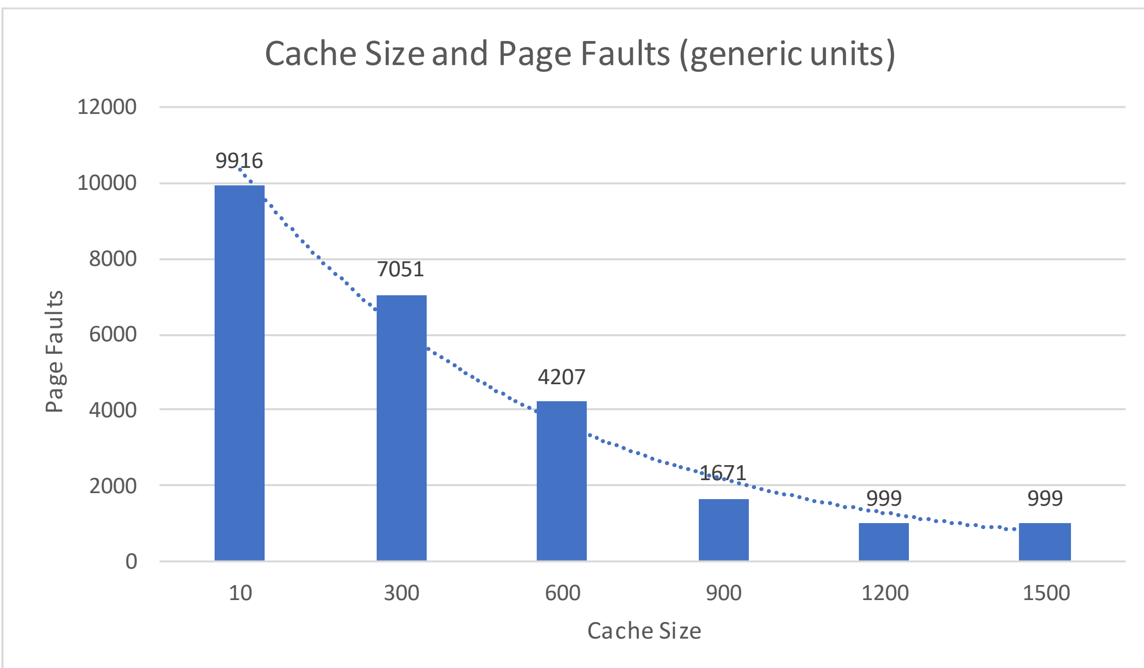
```
******STATISTICS*******
10000 Valid Page requests
999 Total Page Faults
```

Hit rate: ((10000-999)/10000)*100 = **90.01%**

<u>Cache size of 1500:</u>

```
******STATISTICS*******
10000 Valid Page requests
999 Total Page Faults
```

Hit rate: ((10000-999)/10000)*100 = **90.01%**



The data above shows that as the cache size grows, the number of page faults decreases down to the limit of 999. As a result, the hit rate also increases as cache size increases. From my analysis of the accesses.txt file, the number 999 is the limit because the file contains a total of 999 unique numbers. When the cache stores all 999 unique numbers, then it will no longer generate page faults. This graph, however, is not a good indication of performance. In this program, an array was used as the data structure for the cache. By increasing the size of the cache, we are also increasing the size of the array. Ultimately, the cost of the linear search of the cache will cause the performance of the program to decrease. I hypothesize that as cache size grows, then the performance of the FIFO algorithm will decrease due to the increased cache search time. Thus, for the FIFO algorithm, cache size and hit rate seem to have a direct relationship, whereas cache size and performance seem to have an inverse relationship.

Test run showing sample output:

```
[alberto@localhost]~/Workspace% make
gcc -D DEBUG -o fifo fifo.c
[alberto@localhost]~/Workspace% cat accesses.txt | ./fifo 1500
********* PAGES NOT IN CACHE **********
725     840     279     231     250
469     839     426     376     255
776     921     542     432     879
101     558     84      449     605
444     216     419     8       112
821     275     505     846     151
687     430     462     937     618
21      994     277     579     916
538     13      515     915     832
74      718     282     398     163
498     817     889     329     639
166     834     487     317     66
893     747     248     85      269
635     81      265     215     716
804     946     951     720     26
```

…output omitted.

```
189     106     833     578     360
340     799     254     384     239
652     2       389     781     10
88      229     670     583     577
187     409     710     564     378
649     482     361     485     672
613     241     740     314     532
157     205     595     310     217
351     242     671     315     945
987     917     689     596     665
641     7       511     306     823
102     782     838     745     925
198     36      447     245     950
953     611     318     92      73
420     676     612     656     923
381     38      323     814     517
204     733     100     51
******STATISTICS*******
10000 Valid Page requests
999 Total Page Faults
[alberto@localhost]~/Workspace%
```