Page Fault Problem
COP 4634 Systems and Networks 1
Fall 2017

Jeffrey Murphy
Timothy L. J. Stewart
20 November 2017

Background:

For this problem, we considered a two-dimensional array and cthe column and row read operations exactly as specified. 20480 rows by 4096 columns were used to create the array. Since each page file cannot be stored in the heap or stack, they were stored globally. This presents a significant problem when considering the readColumn and readRow operations performed on this array. The total number of page faults for the two processes are computed. For these processes, the following was assumed:

1. Each process is given 10 frames of virtual memory by the system.
2. The two-dimensional array is global
3. 2 out of 10 frames are used for code and stack together
4. An LRU page replacement algorithm is applied to evict pages from memory

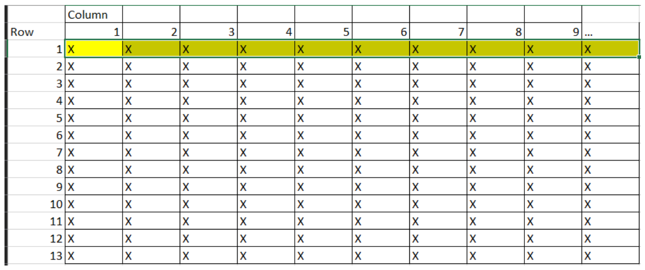
Discussion:

From these assumptions we made the following conclusions:

1. The two code pages are never replaced since they are re-used in every operation;
2. Since there are only eight free pages for each process having more than 8 rows will require every page to be loaded into memory generating 2 + 20480 page faults for the row operations;
3. For the column operations, 2 for the code plus 20480 rows multiplied by 4096 page faults;
4. 2 and 3 can be substantiated by the fact that the LRU (Least Recently Used) Algorithm was used.

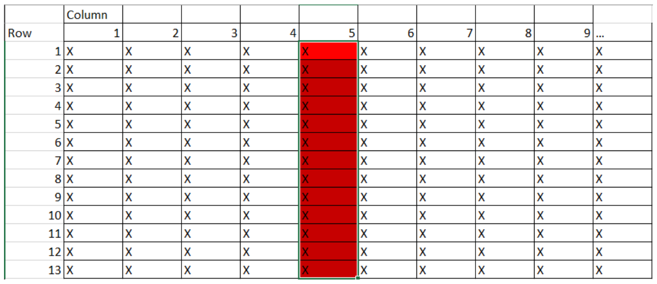
The total number of expected page faults for the read row operation totals 20482. Each byte in the row was read and a new page was not used until a new column was incremented. Figure 1. shows the how the rows are read. Figure 2. shows how the columns are read.

Essentially, readColumn has significantly more page faults than readRow.



	Column									
Row	1	2	3	4	5	6	7	8	9	...
1	X	X	X	X	X	X	X	X	X	X
2	X	X	X	X	X	X	X	X	X	X
3	X	X	X	X	X	X	X	X	X	X
4	X	X	X	X	X	X	X	X	X	X
5	X	X	X	X	X	X	X	X	X	X
6	X	X	X	X	X	X	X	X	X	X
7	X	X	X	X	X	X	X	X	X	X
8	X	X	X	X	X	X	X	X	X	X
9	X	X	X	X	X	X	X	X	X	X
10	X	X	X	X	X	X	X	X	X	X
11	X	X	X	X	X	X	X	X	X	X
12	X	X	X	X	X	X	X	X	X	X
13	X	X	X	X	X	X	X	X	X	X

Figure 1. readRow Operations Performed

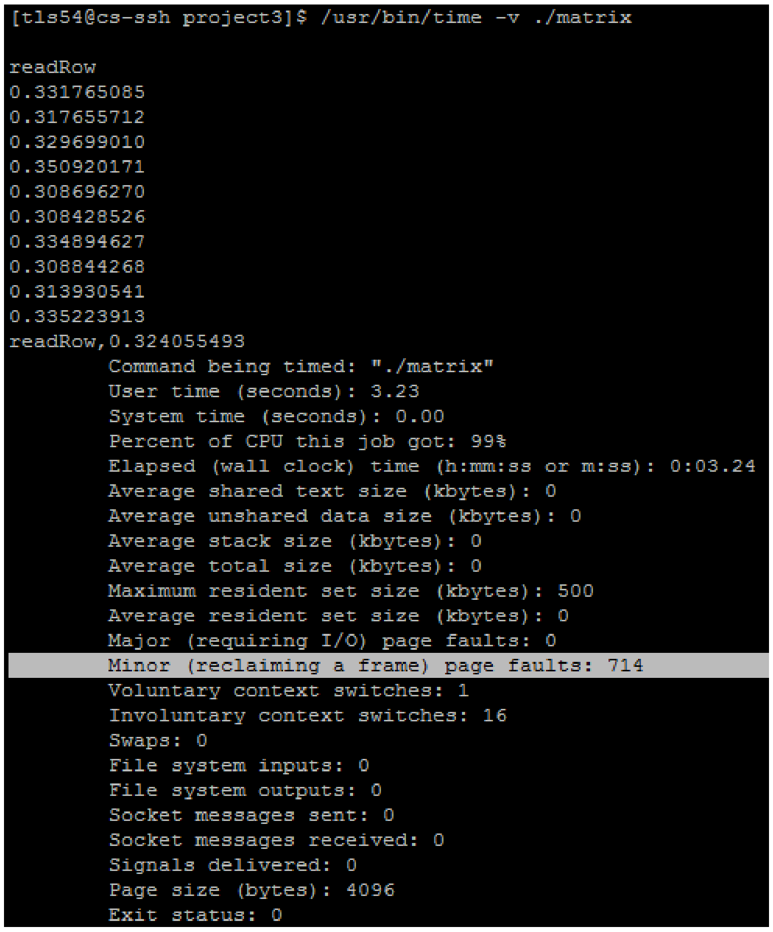


	Column									
Row	1	2	3	4	5	6	7	8	9	...
1	X	X	X	X	X	X	X	X	X	X
2	X	X	X	X	X	X	X	X	X	X
3	X	X	X	X	X	X	X	X	X	X
4	X	X	X	X	X	X	X	X	X	X
5	X	X	X	X	X	X	X	X	X	X
6	X	X	X	X	X	X	X	X	X	X
7	X	X	X	X	X	X	X	X	X	X
8	X	X	X	X	X	X	X	X	X	X
9	X	X	X	X	X	X	X	X	X	X
10	X	X	X	X	X	X	X	X	X	X
11	X	X	X	X	X	X	X	X	X	X
12	X	X	X	X	X	X	X	X	X	X
13	X	X	X	X	X	X	X	X	X	X

Figure 2. readColumn Operations Performed

The total number of expected page faults for the readColumn is 83,886082. Since there are no reused pages, one should expect this many page faults to occur. Each time a page fault occurs the process only reads one of the bytes from the row, then a new page is loaded to read the next row. If there were less than 8 columns and 8 rows, then there wouldn’t be any page faults.

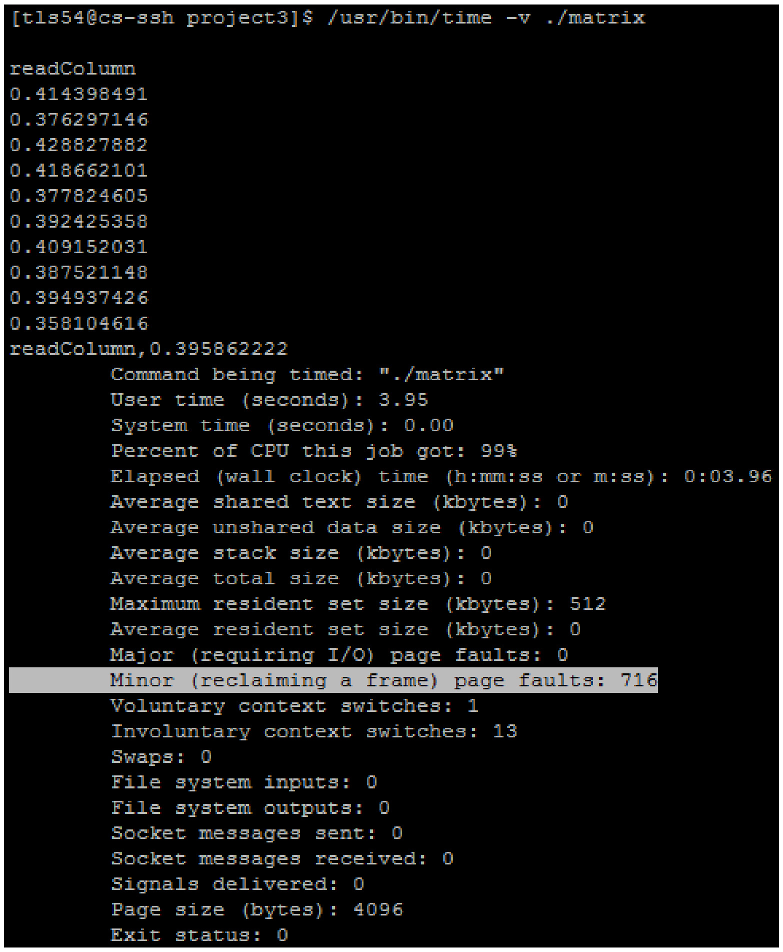
An interesting result from running the programs on the network show significantly different results from what was expected. Even though optimization was turned off using -O0 the number of actual page faults did not vary or even come close to the 20480 and 20480*4094 page faults expected in the thought experiment above. Figures 3 and 4 show the page faults by row and column, respectively. This may be due to the server forcing optimization? The size of the memory may also have something to do with it.



```
[tls54@cs-ssh project3]$ /usr/bin/time -v ./matrix

readRow
0.331765085
0.317655712
0.329699010
0.350920171
0.308696270
0.308428526
0.334894627
0.308844268
0.313930541
0.335223913
readRow,0.324055493
        Command being timed: "./matrix"
        User time (seconds): 3.23
        System time (seconds): 0.00
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:03.24
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 500
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 714
        Voluntary context switches: 1
        Involuntary context switches: 16
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

Figure 3. Ready by Row Page Faults



```
[tls54@cs-ssh project3]$ /usr/bin/time -v ./matrix

readColumn
0.414398491
0.376297146
0.428827882
0.418662101
0.377824605
0.392425358
0.409152031
0.387521148
0.394937426
0.358104616
readColumn,0.395862222
        Command being timed: "./matrix"
        User time (seconds): 3.95
        System time (seconds): 0.00
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:03.96
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 512
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 716
        Voluntary context switches: 1
        Involuntary context switches: 13
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

Figure 4. Read by Column Page Faults

Conclusion:

In conclusion, the thought experiment seems justifiable in the context of what we are studying. However, when checking the actual system page faults, the system experienced drastically fewer page faults, at least two orders of magnitude less than expected. This leaves one wanting for answers. A couple possibilities come to mind, is that gcc is actually doing some optimization that is unwanted for testing. Perhaps the paging system is doing some optimization with an algorithm that does predictive analysis of the matrix program and isn't evicting the pages. **I eagerly would like to know the cause of this discrepancy in page faults and the lack of positive correlation between runtime and page faults.**