Matrix Analysis
Page Fault Analysis
Project 4
COP 4634 Systems & Networks 1
Fall 2016

Ashley Miller/Torrey Bettis
3 November 2016

This analysis will be of a large two dimensional array, and what happens in memory when it iterates through the major aspect. It will also look at precautions that can be taken to minimize any negative impacts to processing time and memory manipulation. When accessing a page that is not currently in memory, a page fault is generated. There are 15 frames when starting this experiment. Two frames are taken up by the code and the stack, leaving 13 frames free for page storage. The size of the row is one page size which is designated as 4096 or 2^{12} bytes, and it will be filled with 1 byte randomly selected characters other than zero.

When looking at navigating the matrix the expectations would be a page fault per page size, or total number if index/page size. In the matrix studied this would be (20480*4096)/4096 = 20480. The number of page faults will only be for row major operations. Column major operation should have more many more page faults as is evidenced by the runtime analysis. How many more page faults and why they are generated will be discussed next.

Column major operations take much longer than row major operations. The predictions are correct for row major operations, but something that is not obvious at first glance happens when column major operations are invoked. The column calls a whole page to read one bit. This can be seen in the tables below.

Table 1 Row Major Operations

0,0	0,1	0,2	0,3	0,4	0,5	0,6
1,0	1,1	1,2	1,3	1,4	1,5	1,6
2,0	2,1	2,2	2,3	2,4	2,5	2,6
3,0	3,1	3,2	3,3	3,4	3,5	3,6
4,0	4,1	4,2	4,3	4,4	4,5	4,6
5,0	5,1	5,2	5,3	5,4	5,5	5,6
6,0	6,1	6,2	6,3	6,4	6,5	6,6
7,0	7,1	7,2	7,3	7,4	7,5	7,6
8,0	8,1	8,2	8,3	8,4	8,5	8,6
9,0	9,1	9,2	9,3	9,4	9,5	9,6

Table 2 Column Major Operations

0,0	0,1	0,2	0,3	0,4	0,5	0,6
1,0	1,1	1,2	1,3	1,4	1,5	1,6
2,0	2,1	2,2	2,3	2,4	2,5	2,6
3,0	3,1	3,2	3,3	3,4	3,5	3,6
4,0	4,1	4,2	4,3	4,4	4,5	4,6
5,0	5,1	5,2	5,3	5,4	5,5	5,6
6,0	6,1	6,2	6,3	6,4	6,5	6,6
7,0	7,1	7,2	7,3	7,4	7,5	7,6
8,0	8,1	8,2	8,3	8,4	8,5	8,6
9,0	9,1	9,2	9,3	9,4	9,5	9,6

Due to this in-efficiency, there are actually many more page faults than row major operations. As can be seen since column major operation really accesses memory for each index in a matrix, so the number of page fault when traversing a matrix this was will be ***m*** * ***n*** page faults when ***m*** is the number of row of the matrix and ***n*** is the number of columns. In the case of the test matric the number of page faults is 20480 * 4096 or 83886080 page faults. The only time the mathematical prediction that was made before would be true for both methods is when the page size is the size of a singular index.

What method chosen to navigate through a matrix weighs heavily on how quickly the matrix can be traversed. It was discovered that our mathematical hypothesis was only true for row major operations. Navigating the matrix using column major operations presents more problems. When using this method, you really get charged multiple times for accessing the same page. To eliminate some of the page faults in this experiment we would need to run read operations immediately after write operation, but this would only be beneficial in the row major operations. Since the page is already in the memory from the write operations, we would be able to read it for “free”. The term free in this case means no page fault would be generated, since the page fault we be only generated on the write actions.