

CS210 Computer Systems, Fall-2023

ps5 A

Instructions

You may work in teams of two for this assignment

For this problem set you will need to clone the assignment repository in the UNIX environment provided. The problems have you explore and evaluate the two provided binaries.

First Name: Jae Hong Last Name: Lee

BU ID: U27565203

Background

You are given two binaries: mm1 and mm2. These two binaries carry out the same calculation on a $n \times n$ matrix of values. The programs are identical except that the core routine that does the work, `mm`, is implemented differently between the two binaries. To run the programs, the binaries take a single command line parameter. This parameter is the path of a matrix file, where a matrix file is an $n \times n \times 8$ byte binary file of values. Each element of the matrix is an 8 byte signed integer.

Setup

Both programs do a series of setup steps to “memory map” the specified file and calculate what the dimension of the matrix is (what is the value of n for the file). Recalling from Lecture 15, “Processes and Virtual Memory”, operating systems can allow a process to create a virtual memory mapping that allows the data of a file to appear in the process’s address space.

<https://jappavoo.github.io/UndertheCovers/lecturenotes/assembly/L15.html#virtual-memory-trick-2-memory-mapping-files>

This is precisely what the setup does. It memory maps a file that contains the values of a matrix so that its contents appear at a location in the virtual address space. This allows the rest of the program to access the matrix values via this virtual memory address.

The `mm` routine

Once the setup is done the values of the specified matrix file will be accessible at a particular address, and the program will execute the `mm` routine. The address of the matrix data will be in `rdi` and the dimension of the matrix will be in `rsi`. The `mm` routine calculates a single 8 byte integer value that it returns in the `rax` register when it is done.

Part of your task will be figuring out what the `mm` routine is calculating and how the binaries differ with respect to their `mm` routine.

Result

Once the `mm` routine returns, the calling code writes the value returned to standard output. So to save the output you can use redirection. To interpret the binary output as a decimal number in ascii you can use `od -t d8`.

Matrix Data Files

We have created matrix files of various dimensions that you can use to explore the binaries. Each file has a url that you can use to get a copy. To get a copy into your UNIX environment you can use the `wget` command. Eg. for the `16x16.matrix` file you would use the command:

```
wget https://www.cs.bu.edu/courses/cs210/files/16x16.matrix
```

First Name: Joe Hong Last Name: lee BU ID: 027065203

16x16.matrix : <https://www.cs.bu.edu/courses/cs210/files/16x16.matrix> A 16 by 16 matrix of random 8 byte integer values

32x32.matrix :: <https://www.cs.bu.edu/courses/cs210/files/32x32.matrix> A 32 by 32 matrix of random 8 byte integer values

64x64.matrix :: <https://www.cs.bu.edu/courses/cs210/files/64x64.matrix> A 64 by 64 matrix of random 8 byte integer values

128x128.matrix : <https://www.cs.bu.edu/courses/cs210/files/128x128.matrix> A 128 by 128 matrix of random 8 byte integer values

256x256.matrix : <https://www.cs.bu.edu/courses/cs210/files/256x256.matrix> A 256 by 256 matrix of random 8 byte integer values

512x512.matrix : <https://www.cs.bu.edu/courses/cs210/files/512x512.matrix> A 512 by 512 matrix of random 8 byte integer values

1024x1024.matrix : <https://www.cs.bu.edu/courses/cs210/files/1024x1024.matrix> A 1024 by 1024 matrix of random 8 byte integer values

2048x2048.matrix : <https://www.cs.bu.edu/courses/cs210/files/2048x2048.matrix> A 2048 by 2048 matrix of random 8 byte integer values

4096x4096.matrix : <https://www.cs.bu.edu/courses/cs210/files/4096x4096.matrix> A 4096 by 4096 matrix of random 8 byte integer values

8192x8192.matrix : <https://www.cs.bu.edu/courses/cs210/files/8192x8192.matrix> A 8192 by 8192 matrix of random 8 byte integer values

Remember, the sizes of these files in bytes are 8 times the number of elements. This means the largest file is 512 Megabytes big, so it might take a few minutes to get a copy of the larger files.

Running the binaries

To run the binaries you will need to issue an appropriate command line, where the command is the path of the binary and the argument is the path of a matrix file. For example, let's assume you have gotten a copy the matrix file `16x16.matrix` in your current working directory, then you can run `mm1` with it as follows:

```
./mm1 16x16.matrix
```

Of course you will want to redirect the output to a file.

The following is an example session of running both binaries on the `16x16.matrix` file.

```
$ pwd
/home/jovyan/ps5b
$ ls
16x16.matrix  mm1  mm2
```

First Name: Jae Hong Last Name: Lee BU ID: U27565203

```
$ ./mm1 16x16.matrix > out1
$ ./mm2 16x16.matrix > out2
$ ls
16x16.matrix  mm1  mm2  out1  out2
$ od -Ad -t d8 out1
00000000          115
00000008
$ od -Ad -t d8 out2
00000000          115
00000008
$
```

The Hunt

Given that we don't have source code, you will need to put your 210 puzzling skills to use. As always, gdb is your friend. Remember gdb can work with any executable binary. Even if you don't have the source code, you can still:

- set breakpoints
- single step instructions
- print values of registers
- examine memory
- disassemble opcodes in memory

In gdb, to start a process running with a command line argument you specify the argument to the 'run' command. Additionally to disassemble instructions you can use the examine command with the 'i' format.

Eg.

```
$ gdb mm1
...
Reading symbols from mm1...
(No debugging symbols found in mm1)
(gdb) set disassembly-flavor intel
(gdb) b _start
Breakpoint 1 at 0x401034
(gdb) run 16x16.matrix
Starting program: /home/jovyan/mm/mm1

Breakpoint 1, 0x0000000000401034 in _start ()
(gdb) x/20i _start
=> 0x401034 <_start>:  pop    rdi
```

```
0x401035 <_start+1>: cmp     rdi,0x2
0x401039 <_start+5>: jne     0x401151 <usageError>
0x40103f <_start+11>: pop     rdi
0x401040 <_start+12>: pop     rdi
0x401041 <_start+13>: call    0x4010b5 <mapFile>
0x401046 <_start+18>: mov     rsi,QWORD PTR ds:0x403068
0x40104e <_start+26>: shr     rsi,0x3
0x401052 <_start+30>: mov     QWORD PTR ds:0x403070,rsi
0x40105a <_start+38>: fild    QWORD PTR ds:0x403070
0x401061 <_start+45>: fsqrt
0x401063 <_start+47>: fistp   QWORD PTR ds:0x403070
0x40106a <_start+54>: mov     rsi,QWORD PTR ds:0x403070
0x401072 <_start+62>: mov     rdi,QWORD PTR ds:0x403060
0x40107a <_start+70>: call    0x401000 <mm>
0x40107f <_start+75>: mov     QWORD PTR ds:0x403078,rax
0x401087 <_start+83>: mov     rax,0x1
0x40108e <_start+90>: mov     rdi,0x1
0x401095 <_start+97>: mov     rsi,0x403078
0x40109c <_start+104>: mov     rdx,0x8
(gdb) x/10i mm
0x401000 <mm>:      lea     rcx,[rsi*8+0x0]
0x401008 <mm+8>:      xor     rdx,rdx
0x40100b <mm+11>:     xor     r8,r8
0x40100e <fori>:      cmp     rdx,rsi
0x401011 <fori+3>:     jge     0x401030 <done>
0x401013 <fori+5>:     xor     rax,rax
0x401016 <forj>:      cmp     QWORD PTR [rdi+rax*8],0x0
0x40101b <forj+5>:     js      0x401020 <neg>
0x40101d <forj+7>:     inc     r8
0x401020 <neg>:      inc     rax
(gdb)
```

Happy hunting!

The Questions

The Setup

1. (1 point) Which section of the program's memory layout does `_start` in the setup code find the string of the matrix file path ?

stack

2. (1 point) How many "function" calls are made prior to calling the `mm` routine? (Eg. the number of times the code executes the `call` instruction prior to calling the `mm` routine.)

1.

3. (2 points) How many system calls are invoked in total if no errors were raised? (Note: both programs are identical with respect to the setup code. Additionally, make sure to also consider the code after the `mm` routine. You might want to set multiple breakpoints).

4

4. (3 points) What are the unique Linux system calls used **prior** to calling `mm`? (Your answer should identify the system call numbers and the Linux english system call routine name).

2: open, 8: lseek, 9: mmap

Notes

Note1: You can find a table of the system calls here : <https://filippo.io/linux-syscall-table/> and here <https://hackeradam.com/x86-64-linux-syscalls/>.

Note2: You don't need to fully understand what each system call does. You simply need to figure out what system calls are being made by the setup code. This is to give you an idea for how a "real" program uses system calls to create a memory mapping of a file.

Note3: FYI: In the setup code you will find the following sequence of instructions:

1	shr	rsi, 0x3	- shift
2	mov	QWORD PTR ds:0x403070, rsi	
3	fild	QWORD PTR ds:0x403070	
4	fsqrt		
5	fistp	QWORD PTR ds:0x403070	

This code calculates the dimension of the matrix in elements. It starts with the size of the file in bytes in `rsi`. Using a shift right instruction, `shr`, it shifts the size in bytes by three bits to the right. This is the same as dividing by eight. At this point we have the size of the matrix in 8-byte elements in `rsi`. The remaining instructions calculate the square root of this value to determine the dimension of the matrix. Specifically, they use support that the INTEL processors have for doing more complex math using what are called floating point instructions and registers.

The Calculation

The standard way for representing a two dimensional array/matrix in memory is called row major ordering. In this ordering we place the elements of the first row of the array in memory first. We then place the second row of elements and so on. The following diagram illustrates row-major ordering.

Our standard notation for 2D matrices from math is $M[r][c]$. Eg $M[0][0]$ means the element at row 0 column 0 and $M[4][3]$ means the element at row 4 column 3.

Notice the values are placed in memory according to their row.

$M[0][0]$	$M[0][1]$	$M[0][2]$	$M[0][n-1]$
$M[1][0]$	$M[1][1]$	$M[1][2]$	$M[1][n-1]$
$M[2][0]$	$M[2][1]$	$M[2][2]$	$M[2][n-1]$
.
.
$M[n-1][0]$	$M[n-1][1]$	$M[n-1][2]$	$M[n-1][n-1]$

Memory	
0x400080	$M[0][0]$
0x400088	$M[0][1]$
0x400090	$M[0][2]$
0x400098	...
0x4000a0	...
0x4000a8	$M[0][n-1]$
0x4000b0	$M[1][0]$
0x4000b8	$M[1][1]$
0x4000c0	$M[1][2]$
0x4000c8	...
0x4000d0	...
0x4000d8	$M[1][n-1]$
0x4000e0	$M[2][0]$
0x4000e8	...
0x4000f0	...

Our programs and data files assume row major ordering of the matrixes.

LEA

The code for the `mm` routine uses the INTEL `lea` instruction. LEA stands for Load Effective Address. This instruction can be used to do simple address calculations and load a register with the resulting address.

Here is the use of LEA that you will find in the `mm1` version of the `mm` routine

```
lea    rcx, [rsi*8+0x0]
```

The effect of this instruction is to load `rcx` with the value of `rsi * 8`. Eg. if `rsi` has the value 2 then this instruction will set `rcx` with the value of `rsi * 2 = 2 * 8 = 16`.

For the `mm2` version of `mm` has an additional use of `lea`

First Name: Jae Hong Last Name: lee BU ID: 029565223

```
lea    r9,[rdi+rax*8]
```

In this case `r9` will be loaded with the result of the following calculation: `rdi + (rax * 8)` where we would substitute the current values in the `rdi` and `rax` registers. Eg. if `rdi = 12` and `rax = 3` the `r9` would be set to the result of `rdi + (rax * 8) = 12 + (3 * 8) = 36`.

First Name: Joe Hong Last Name: Lee BU ID: U20565203

5. (10 points) Using the mm1 binary, figure out what the 'mm' routine is doing. This version is composed of 17 instructions (Eg. x/17i mm will disassemble them in gdb, don't forget to set `disassembly-flavor intel`). Do not simply state the instructions that are being executed. Rather, provide pseudocode (<https://en.wikipedia.org/wiki/Pseudocode/>) that describes what the mm routine is doing. Then provide a short english summary of what the result represents. Do NOT include registers or assembly instructions in your pseudocode. Eg. The mm routine calculates the sum of all the elements of the matrix. Here is an example of pseudocode:

```
n : number of students
scores : is an n element array. Each element is the score for the ith student.

passes : count of the number of students that passed
failures: count of the number of students that failed

initialize passes to zero
initialize failures to zero

for (i=0; student < n; i++):
    if scores[i] a passing grade
        passes++
    else:
        failures++
```

Your answer should be provided in the space below.

```
m: input matrix
numRow: number of Rows in input matrix
numCol: number of columns in input matrix
num_pos: number of positive integers in matrix

initialize: 0 to current Row
initialize : 0 to num pos
```

```
for current Row < than numRow:
    current col = 0;
    for current column < num col:
        current element = m[current Row][current column]
        if current element > 0:
            continue
        else
            increase num-pos
        current col++
    current Row++
```

Return num pos

This code loops through the input array and count the positive integers in matrix and return the number

First Name: Jue Hong Last Name: lee BU ID: 027565203

6. (10 points) Using the mm2 binary figure out how its version of the mm routine works. This version is composed of 18 instructions. Again, provide pseudocode for the routine's operation and state how it differs from the mm1 version.

Your answer should be provided in the space below.

m: input matrix
numRow: number of Rows in input matrix
numCol: number of columns in input matrix
num_pos: number of positive integer in matrix
initialize: 0 to current Row
initialize: 0 to num pos

```
for current column < numCol:
    current Row = 0
    for current Row < numRow:
        current element = m[current Row][current column]
        if current element < 0:
            continue
        else
            increase num_pos
    current column ++
    current Row ++
Return num_pos
```

mm1 routine iterate the loop each row, each column However
mm2 It iterate through by setting the current column and
for each column, goes through each row in the column.

MM1 is row then column loop mm2 is column and Row loop

Measure the performance

Bash provides a builtin command called `time`. While it is not always the most accurate way of measuring the time it takes for a program to execute, it will be good enough for our purposes. If you are interested in the details see `man time`.

If you place the `time` command at the start of a command line, bash will give you a rough measure of how long your command line takes to execute. Eg.

```
$ ls
16x16.matrix  mm1  mm2  ps5a.pdf  README.md
$ time ./mm1 16x16.matrix > out1

real 0m0.001s
user 0m0.001s
sys 0m0.000s
$ time ./mm2 16x16.matrix > out2

real 0m0.001s
user 0m0.000s
sys 0m0.001s
$
```

Time reports three values. In our case the only one we will care about is the value called 'user'. This value attempts to measure how much time was spent executing our binary.

7. (10 points) Run each binary 10 times for each matrix file and create a line plot of the results, where the plot has one line for each program's results. The x-axis should be the matrix size of the matrix in bytes and the y-axis the user time in seconds. Remember the size of a matrix in bytes is the dimension squared times 8. This size should match the file size for the matrix. Eg. The size in bytes of the `16x16.matrix` is $16 \times 16 \times 8 = 2048$ bytes.

For each program you will have 10 time measurements for each matrix size. On a single graph you should plot two lines, one for each program, where each line is composed of the results for the particular program. The points on the line should be the mean value of the 10 runs for a particular matrix size. You should also check these runs to understand how much the min and max times of the 10 runs vary.

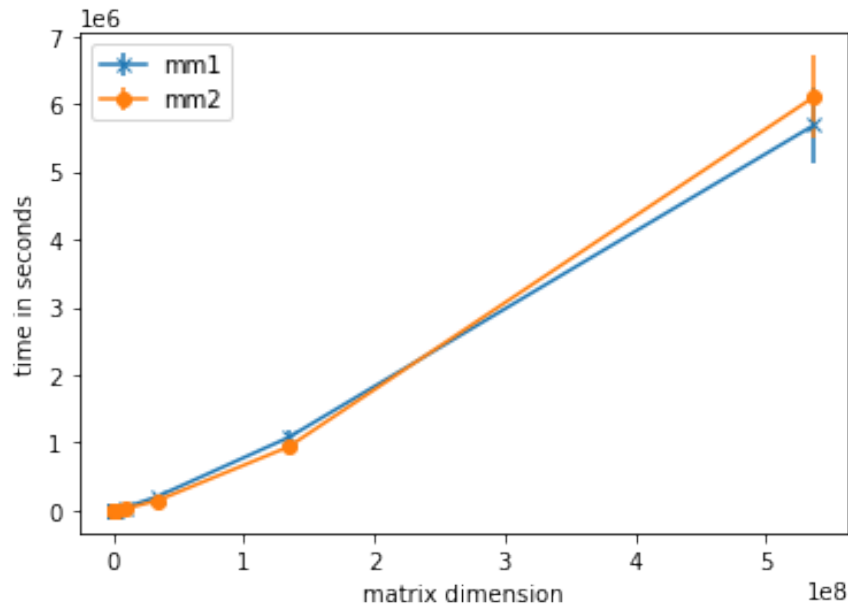
Notes

Note1: We recommend writing a script to automate gathering your data to save you time. We looped over each program (`mm1` and `mm2`) as `p` and over each matrix size as `s`, then ran this loop to generate 10 times for every combination:

```
for ((i=0; i<10; i++)); do
    time ./$p ${s}x${s}.matrix > $p.$s.out
done 2>&1 | grep user | cut -c 8-11 > $p.$s.times
```

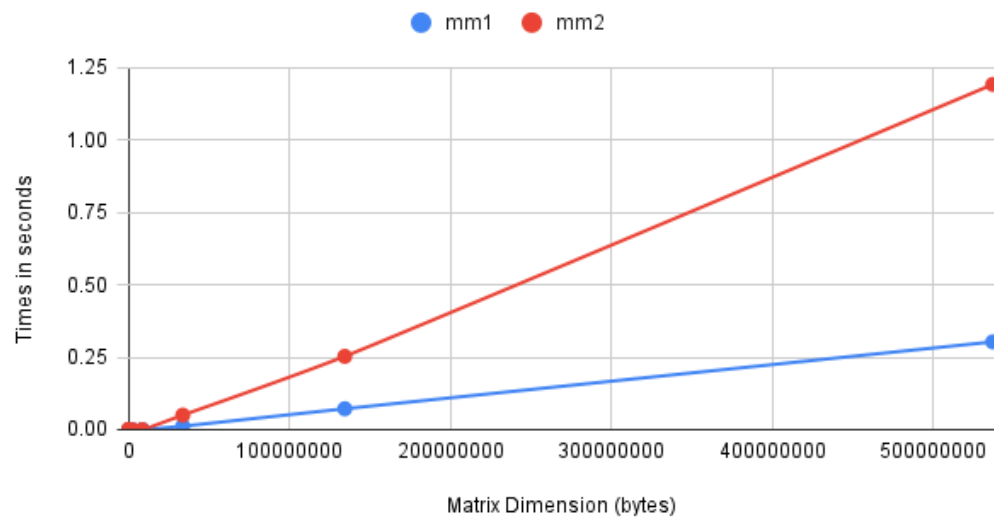
Note2: You can use any software you like to create the graph from the data you gather. Hand-drawn graphs will not be accepted. We included error bars in our graphs but these are optional.

The following is an example of how the graph should look with respect to formatting. **The data on the following graph is dummy data - do not expect your plots to look the same**



Please place a copy of your graph here.

mm1 and mm2



First Name: Joe Hong Last Name: Lee BU ID: 027565247

8. (5 points) Provide a brief paragraph, below, that summarizes what the plot is telling you. Also, in words, how much do the max and min vary between runs. This should be no more than three to four sentences in length.

I learned that mm1 has a much better time complexity than mm2, especially as the matrix dimension grows. The two mm functions are executed similarly at first, but the time for mm2 grows much more rapidly. In my opinion, this may be the case since data is organized in memory in a row-first way, so iterating through it in a row-first way is significantly more streamlined since there are fewer jumps and the cache can be used better. There is a small difference between max and min, which is significant because it means efficiencies aren't random.

What's going on?

Use the following commands to gather a report on how the programs behave with respect to their memory accesses and the caches.

```
$ valgrind --tool=cachegrind ./mm1 8192x8192.matrix > mm1.8192.out
```

```
$ valgrind --tool=cachegrind ./mm2 8192x8192.matrix > mm2.8192.out
```

These commands will produce quite a bit of output. This tool runs the given program and analyzes its memory access behaviour. Do a little research on the terms you see in the report. Identify what seems to be different between the two programs.

9. (5 points) Write one short paragraph, in the space provided below, describing what you learned and why there might be a difference in performance. Please follow the notes on how to find the reason they are different.

We learned in lecture during the "role-play" section that the memory bus is told to fetch a value in the fetch, decode, execute loop. When we didn't use a cache, it would run back and forth, frequently accessing data adjacent to memory. It seemed like a waste that he could only grab one byte at a time. By making use of a cache, the memory bus would grab a bunch of adjacent chunks of memory and store them. In that case, the bus would run back out if the CPU requested data at a particular address that the cache did not have. From valgrind, we can see that mm1 has a D1 miss rate of 12.5%, which means one in eight calls to the cache were missed. Therefore, 1/8 calls require the memory bus to run out. The D1 miss rate for mm2 is 100%, which means the cache is never hit, and the bus is constantly going back and forth. Data is stored in memory by iterating through the matrix in row major in mm1. In contrast, in mm2, the matrix is traversed in column major order, so changing the index requires a much larger memory jump each time.

Notes

Note1: “Valgrind” is a powerful tool that allows us to learn about how a program accesses memory. In particular we are using a sub-tool called “cachegrind” (<https://valgrind.org/docs/manual/cg-manual.html>). Cachegrind uses data it gathers about the memory accesses (what addresses and number of bytes the program loads and stores to) along with knowledge it has about the system’s caches to provide us with insight on how a program behaves with respect to caching.

Note2: Remember what we observed in the in-class “role” play we did in Lecture 16 (<https://jappavoo.github.io/UndertheCovers/lecturenotes/assembly/L16.html>). Remember what we said regarding “cache hits” and “cache misses”.

Note3: You may find the following useful (please note these are not academic articles and may have flaws):

- <https://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips>
- <https://www.makeuseof.com/tag/what-is-cpu-cache/>
- https://en.wikipedia.org/wiki/CPU_cache