Homework week 8

Refined simulator pitch:

After the team meeting, we made some modifications to the preliminary design. Currently, our VMM simulator focuses on multi-level TLB and TLB warming.

Configurable parameters:

- TLB related:
  - L1 TLB entry size & L2 TLB entry size
  - TLB replacement algorithm: FIFO, RANDOM, CLOCK, LRU, LFU
  - TLB warming turn on or off
- Workload:
  - Processes like: Games, and machine learning tasks would have different memory usage behavior.

My role in the team, coding the VMM simulator together with Hank and supporting coming up with different workloads for the simulator.

Chapter 20

Q1. With a linear page table, you need a single register to locate the page table, assuming that hardware does the lookup upon a TLB miss. How many registers do you need to locate a two-level page table? A three-level table?

One would be enough because the reference to the next level page table is stored in the page directory or previous level page table.

Q2. Use the simulator to perform translations given random seeds 0,

1, and 2, and check your answers using the -c flag. How many memory references are needed to perform each lookup.

Two memory references are needed. The first is using PDBR to locate PDE, then using PDE to locate PTE.

Q3. Given your understanding of how cache memory works, how do you think memory references to the page table will behave in the cache? Will they lead to lots of cache hits (and thus fast accesses?) Or lots of misses (and thus slow accesses)?

I think loading the page table into the cache would cause lots of cache misses.

1. The page table is space-consuming.
2. The page table is less frequently accessed compared with instructions and data.

Chapter 21

Q1. How do the CPU usage statistics change when running mem? Do the numbers in the user time column make sense? How does this change when running more than one instance of mem at once?

Several columns of data change, “us” change from 0 to 8, “id” change from 100 to 92.

It makes sense because mem.c is a user-created process, thus the time spent on this process is added to the user time.

When running two instances at once, the user time doubled to 16-17, and idle time changed to 84.

Q2. What do you notice about the value? In particular, how does the free column change when the program exits? Does the amount of free memory increase by the expected amount when mem exits?

The free column’s value increases as the program exits, and the amount of increment is almost the amount of the memory allocated, which is 1025 MB.

Q3. Let’s assume it’s something like 8 GB of memory; if so, start by running mem 4000 (about 4 GB) and watching the swap in/out columns. Do they ever give non-zero values? Then, try with 5000, 6000, etc. What happens to these values as the program enters the second loop (and beyond), as compared to the first loop? How much data (total) are swapped in and out during the second, third, and subsequent loops? (do the numbers make sense?)

From my machine, I found there is almost 14G memory available, so to make swapping happen, I ran the command mem 14000. The result is as below, there is memory swapped out. It took a long time to finish the first loop, and within the first loop, the free space is almost used up and there is memory swapped out. For the latter loop, the swapped space and free space remain as they finished the first loop. 262912 are swapped in and out during the later loops, it makes sense, as 262912 + 14374336 equals to 13.97 GB almost 14G.

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 14374336 203060 1011400    0    0     0     0   15  109  0  0 100  0  0
 0  0      0 14374336 203060 1011400    0    0     0     0   35  161  0  0 100  0  0
 0  0      0 14374336 203060 1011400    0    0     0     0   15   96  0  0 100  0  0
 1  0      0 13904600 203060 1011400    0    0     0     0   42  153  2  1 97  0  0
 1  0      0 12722216 203060 1011400    0    0     0     0   25   91  4  5 92  0  0
 1  0      0 11622992 203060 1011400    0    0     0     0   32  127  4  4 92  0  0
 1  0      0 10486472 203060 1011400    0    0     0     0  152  453  4  5 92  0  0
 1  0      0 9403376 203060 1011400    0    0     0     0   32  126  3  5 92  0  0
 1  0      0 8311712 203060 1011400    0    0     0     0   24   92  3  5 92  0  0
 1  0      0 7225088 203060 1011400    0    0     0     0   30  102  4  4 92  0  0
 1  0      0 6139472 203060 1011400    0    0     0     0   28  108  4  5 92  0  0
 1  0      0 5029160 203060 1011400    0    0     0     0   38  136  4  4 92  0  0
 1  0      0 3946820 203060 1011400    0    0     0     0   22   92  5  3 92  0  0
 1  0      0 2929496 203060 1011400    0    0     0     0   35  125  4  5 92  0  0
 1  0      0 1910912 203060 1011400    0    0     0     0   41  120  4  5 92  0  0
 1  0      0 942224 203060 1011400    0    0     0     0   29  120  4  4 92  0  0
 1  0      0 160016 203060 962260    0    0     0     0   94  223  4  5 91  0  0
 1  0 262912 248952 203048 629316    0 262776     0 262776  197  464  5  5 90  0  0
 1  0 262912 248952 203048 629316    0    0     0     0   23   86  8  0 92  0  0
 1  0 262912 249160 203048 629328    0    0     0     0   46  127  8  0 92  0  0
 1  0 262912 249160 203048 629328    0    0     0     0   26   96  8  0 92  0  0
```

Q4. Do the same experiments as above, but now watch the other statis-
tics (such as CPU utilization, and block I/O statistics). How do they change when mem is running?

For block I/O, sometimes there are data transmissions. For CPU utilization, the cpu idle time drops as mem.c starts running.

Q5. Pick an input for mem that comfortably fits in memory (say 4000 if the amount of memory on the system is 8 GB). How long does loop 0 take (and subsequent loops 1, 2, etc.)? Now pick a size comfortably beyond the size of memory (say 12000 again assuming 8 GB of memory). How long do the loops take here? How do the bandwidth numbers compare? How different is performance when constantly swapping versus fitting everything comfortably in memory? Can you make a graph, with the size of memory used by mem on the x-axis, and the bandwidth of accessing said memory on the y-axis? Finally, how does the performance of the first loop compare to that of subsequent loops, for both the case where everything fits in memory and where it doesn't?

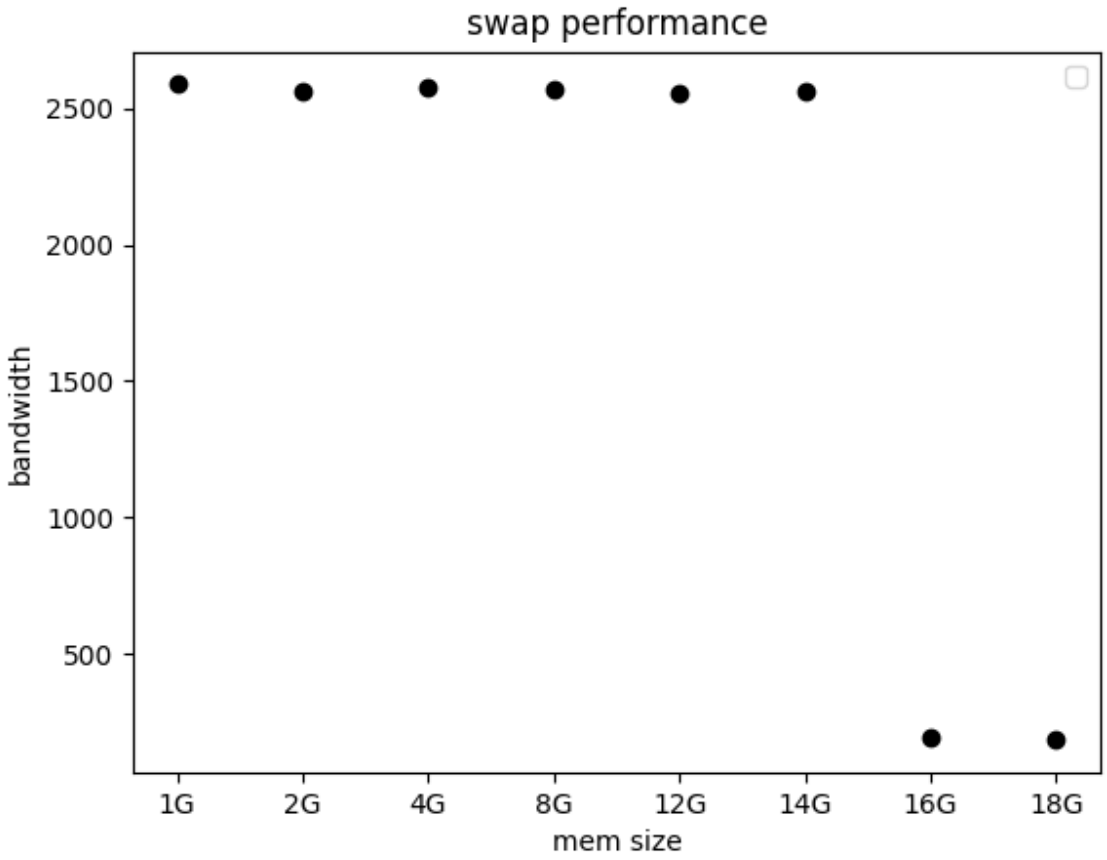
I took 1024 as the parameter, loop 0 took 863 ms, later loops only took around 400 ms,
loop 0 in 863.36 ms (bandwidth: 1186.06 MB/s)
loop 1 in 394.64 ms (bandwidth: 2594.80 MB/s)
loop 2 in 414.11 ms (bandwidth: 2472.77 MB/s)
loop 3 in 396.44 ms (bandwidth: 2582.97 MB/s)
loop 4 in 386.20 ms (bandwidth: 2651.46 MB/s)

Then I took 14000 as the parameter, loop 0 took 11600 ms, later loops took around 5500 ms

loop 0 in 11600.05 ms (bandwidth: 1206.89 MB/s)
loop 1 in 5686.39 ms (bandwidth: 2462.02 MB/s)
loop 2 in 5464.49 ms (bandwidth: 2562.00 MB/s)
loop 3 in 5453.45 ms (bandwidth: 2567.18 MB/s)
loop 4 in 5568.24 ms (bandwidth: 2514.26 MB/s)
loop 5 in 5429.68 ms (bandwidth: 2578.42 MB/s)

The bandwidth between the two conditions is almost the same.

I have tested when allocating memory to 2G, 4G, 8G, 10G, 14G (all of these ar ebelow the free space), the bandwidth does not have much difference, for the first loop the bandwidth is around 1200 MB/s, while for the following loops, the bandwidth is around 2500. However, things change when it comes to 16G and 18G, there is an apparent drop in the bandwidth, only around 200 MB/s, as shown in the below graph.



Q6. What happens if you try to run mem with increasingly large values, beyond what seems to be available in swap? At what point does the memory allocation fail?

From swapon -s, I got to know that I only have 4G swap space, which is also proved by that when I increase the mem allocation size to 18.5 G, the allocation fails.

Chapter 22

Q1. Generate random addresses with the following arguments: -s 0 -n 10, -s 1 -n 10, and -s 2 -n 10. Change the policy from FIFO, to LRU, to OPT. Compute whether each access in said address traces are hits or misses

Here I just list a few example runs, because the calculation is done in my mind, and I think I am correct for most of the cases.

FIFO:

-s 0 -n 10

Access: 8 MISS FirstIn -> [8] <- Lastin Replaced:- [Hits:0 Misses:1]
Access: 7 MISS FirstIn -> [8, 7] <- Lastin Replaced:- [Hits:0 Misses:2]
Access: 4 MISS FirstIn -> [8, 7, 4] <- Lastin Replaced:- [Hits:0 Misses:3]
Access: 2 MISS FirstIn -> [7, 4, 2] <- Lastin Replaced:8 [Hits:0 Misses:4]
Access: 5 MISS FirstIn -> [4, 2, 5] <- Lastin Replaced:7 [Hits:0 Misses:5]
Access: 4 HIT FirstIn -> [4, 2, 5] <- Lastin Replaced:- [Hits:1 Misses:5]
Access: 7 MISS FirstIn -> [2, 5, 7] <- Lastin Replaced:4 [Hits:1 Misses:6]
Access: 3 MISS FirstIn -> [5, 7, 3] <- Lastin Replaced:2 [Hits:1 Misses:7]
Access: 4 MISS FirstIn -> [7, 3, 4] <- Lastin Replaced:5 [Hits:1 Misses:8]
Access: 5 MISS FirstIn -> [3, 4, 5] <- Lastin Replaced:7 [Hits:1 Misses:9]

FINALSTATS hits 1 misses 9 hitrate 10.00

-s 1 -n 10

Access: 1 MISS FirstIn -> [1] <- Lastin Replaced:- [Hits:0 Misses:1]
Access: 8 MISS FirstIn -> [1, 8] <- Lastin Replaced:- [Hits:0 Misses:2]
Access: 7 MISS FirstIn -> [1, 8, 7] <- Lastin Replaced:- [Hits:0 Misses:3]
Access: 2 MISS FirstIn -> [8, 7, 2] <- Lastin Replaced:1 [Hits:0 Misses:4]
Access: 4 MISS FirstIn -> [7, 2, 4] <- Lastin Replaced:8 [Hits:0 Misses:5]
Access: 4 HIT FirstIn -> [7, 2, 4] <- Lastin Replaced:- [Hits:1 Misses:5]
Access: 6 MISS FirstIn -> [2, 4, 6] <- Lastin Replaced:7 [Hits:1 Misses:6]

Access: 7 MISS FirstIn -> [4, 6, 7] <- Lastin Replaced:2 [Hits:1 Misses:7]
Access: 0 MISS FirstIn -> [6, 7, 0] <- Lastin Replaced:4 [Hits:1 Misses:8]
Access: 0 HIT FirstIn -> [6, 7, 0] <- Lastin Replaced:- [Hits:2 Misses:8]

FINALSTATS hits 2 misses 8 hitrate 20.00

Q2. For a cache of size 5, generate worst-case address reference streams for each of the following policies: FIFO, LRU, and MRU (worst-case reference streams cause the most misses possible. For the worst case reference streams, how much bigger of a cache is needed to improve performance dramatically and approach OPT?

FIFO: 1,2,3,4,5,6,7,8,9,10,1,2,3,4,5

LRU: 1,2,3,4,5,6,7,8,9,10,1,2,3,4,5

MRU: 1,2,3,4,5,6,5,6,5,6,5,6

For my provided examples, for FIFO and LRU, double the size of the cache would help.

For MRU, a cache of size 6 would help for the provided example.

However, there always exists a worst-case stream that no matter how big the cache is can make a mess.

Q3. Generate a random trace (use python or perl). How would you expect the different policies to perform on such a trace?

This is the random generated trace: [7, 1, 2, 6, 4, 9, 4, 1, 5, 7]

```
paging-policy.py -f 1.txt -p FIFO -c
result: FINALSTATS hits 1   misses 9   hitrate 10.00

paging-policy.py -f random.txt -p LRU -c
result: FINALSTATS hits 1   misses 9   hitrate 10.00
```

```
paging-policy.py -f random.txt -p OPT -c
result: FINALSTATS hits 2   misses 8   hitrate 20.00
```

Q4. Now generate a trace with some locality. How can you generate such a trace? How does LRU perform on it? How much better than RAND is LRU? How does CLOCK do? How about CLOCK with different numbers of clock bits?

I defined two types of page, one is frequently used called hot set, the other one is less frequently used called cold set. Hot set have a higher probability to be used again later.

This is the generated random trace: [3, 0, 0, 3, 5, 0, 3, 8, 5, 8, 5, 5, 6, 2, 3, 7, 5, 0, 3, 0]

```
python paging-policy.py -f random.txt -p LRU -c
result: FINALSTATS hits 8   misses 12   hitrate 40.00

python paging-policy.py -f random.txt -p RAND -c
result: FINALSTATS hits 8   misses 12   hitrate 40.00

python paging-policy.py -f random.txt -p CLOCK -c -b 1
result: FINALSTATS hits 9   misses 11   hitrate 45.00

python paging-policy.py -f random.txt -p CLOCK -c b 2
result: FINALSTATS hits 7   misses 13   hitrate 35.00

python paging-policy.py -f random.txt -p CLOCK -c -b 4
result: FINALSTATS hits 9   misses 11   hitrate 45.00

python paging-policy.py -f random.txt -p CLOCK -c -b 6
result: FINALSTATS hits 9   misses 11   hitrate 45.00
```