HW 4

Spring 2017

Please write answers in the space provided and export as pdf. Show work for partial credit.

Problem 1: Memory Access

A. For this question, we will be using an architecture with the following attributes:

- 18-bit virtual address
- 16-bit physical address
- 8-entry, fully-associative TLB

The TLB, a partial page table, a few virtual addresses are shown below. Use this information to answer questions i through iv.

TLB			
Valid	Dirty	VPN	PFN
1	0	00 0000 0000	0001 0000
0	0	00 0000 0101	0000 1110
0	1	00 0001 0110	0000 0000
0	1	00 0001 1011	0000 0000
1	0	11 1101 0100	1000 0100
1	0	11 0100 0101	0100 0101
1	1	10 0010 1010	1100 0110
1	0	10 0000 1111	1000 1001

	Page Table (partial)	
	PFN	Valid
0	0001 0000	1
1	0011 0011	0
2	0000 0000	0
3	0010 0100	0
4	0001 1000	1
5	0000 1110	0
6	1100 0011	1
7	0101 0110	1
. . . .	. . . .	. . . .
1022	1010 0101	0

1023	1111 1111	1

```
Virtual Addresses:

A | 00 0000 0001 1001 1001

B | 00 0000 0101 0110 0110

C | 11 1111 1111 1111 1111

D | 00 0000 0111 0000 1111

E | 00 0000 0110 1100 0101
```

i. Which virtual address(es) cause page fault(s)?

A and B

A = 1, B = 5, both invalid in page table while all others are valid

ii. How many memory accesses in total were required to obtain the contents of virtual address E?

2, because not in TLB so we access the page table once for the valid PFN corresponding to VPN 6 and then we have to access the contents of the memory at the PFN

iii. For virtual address D, what physical address is generated?

PFN + Offset = `0101 0110 +  0000 1111 = 01100101`

B. Consider the following page-reference string: 3, 1, 4, 5, 3, 4, 1, 5, 3, 2, 4, 5, 2, 4, 1, 3, 1, 5

How many page faults would occur for the following replacement algorithms, assuming 4 frames? Remember that all frames are initially empty, so your first unique pages will all cost one fault each. Only calculate page faults for the initial sequence shown. You should assume that this pattern repeats endlessly (Hint: May be important for the optimal replacement algorithm.)

i. LRU replacement
   4 (initial) + 1 (2) + 1 (4) + 1 (1) + 1 (3) + 1 (5) = 9

ii. FIFO replacement

   4 (initial) + 1 (2) + 1 (3) + 1 (1) = 7

iii. Optimal replacement (Belady's Min)

   4 (initial) + 1 (2) + 1 (3) = 6

Problem 2: Cache Simulation

Given a **direct mapped cache** with the following specifications:

- 16 bit physical addresses
- 16 byte blocks
- 4 KB cache size for data
- Byte addressable

Mark each memory access as a cache hit or miss in the table provided. If it is a miss, specify the type of miss. Assume that the cache is initially empty, and none of these locations have been in the cache before.

Offset = log 16 = 4
Index = log 250 = 8
Tag = 16 – (4+8) = 4

ADDRESS	Tag	Index	Offset
0x1420	0001	01000010	0000
0x1240	0001	00100100	0000
0x1428	0001	01000010	1000
0x566F	0101	01100110	1111
0x266F	0010	01100110	1111
0x5664	0101	01100110	0100
0x5662	0101	01100110	0010
0x1242	0001	00100100	0010
0x8242	1000	00100100	0010
0x1248	0001	00100100	1000

ADDRESS	HIT/MISS	MISS TYPE (IF ANY)
0x1420	Miss	Compulsory
0x1240	Miss	Compulsory
0x1428	Hit	
0x566F	Miss	Compulsory
0x266F	Miss	Conflict
0x5664	Miss	Conflict
0x5662	Hit	
0x1242	Hit	
0x8242	Miss	Conflict
0x1248	Miss	Conflict

Problem 3: Cache Organization

Given a 16-way set associative cache in a 32-bit byte-addressed architecture with the following specifications:

- 1 MB cache size
- 512 byte block size
- Write-back with dirty bit at block granularity
- FIFO replacement strategy

A. How many bits long is the tag?

1 MB / 512 bytes = 2048 blocks. 2048 / 16 = 128 sets. Log 128 = 7, so 7 bits for the index.

512 bytes for a block, log 512 = 9, so 9 bits for the offset.

32 – 7 – 9 = 16 bits for the tag.

B. How many meta-data bits are required in this cache?

Tag = 16 bits, Dirty = 1, Valid = 1,

So metadata per block = 16 + 1 + 1 + = 18

FIFO needs log 16 = 4 bits.

So metadata per set = 4 bits.

So total metadata = 2048 * 18 + 4 * 128 = 37376 bits = 4.672 KB.

C. What is the total size of the cache in bits (including metadata and data)?

Data = 1 MB, Metadata = 4.672 KB, so total size = 1.004672 MB.

Problem 4: Cache Timing

EMAT = Time for a hit + (Miss rate x Miss penalty)

A. Find the EMAT for a machine with a 1-ns clock, a miss penalty of 40 clock cycles, a miss rate of 0.05, and a cache access time (including hit detection) of 1 clock cycle. Assume that the read and write miss penalties are the same and ignore other write stalls.

1 + (40 * 0.05) = 3 ns

B. Suppose we can improve the miss rate to 0.03 by doubling the cache size. This causes the cache access time to increase to 2 clock cycles. Using the EMAT as a metric, determine if this is a good trade-off. Please show your work.

2 + (40 * 0.03) = 3.2 ns, so no it is not a good tradeoff.

C. Consider the following memory hierarchy:

- A TLB with an access time of 1 cycle. The hit rate for the TLB is 95%.
- An L1 cache with a 1 cycle access time, and 99% hit rate.
- An L2 cache with a 5 cycle access time, and a 90% hit rate.
- An L3 cache with a 20 cycle access time, and a 80% hit rate.
- A physical memory with a 100 cycle access time.
- Page faults occur .1% of the time. Servicing a page fault takes 2000 cycles

You may assume that all page table entry data can be found in one memory access. Also, note that the page entry itself can be in the cache.

i. What is the EMAT for accessing memory given a **physical** address?

EMAT = L1 = 1 + 0.01 * L2 = 1.09

L2 = 5 + 0.1 * L3 = 9

L3 = 20 + 0.2 * PM = 40

PM = 100

ii. What is the EMAT for accessing memory given a **virtual** address?

EMAT = 6.1 + 1.09 = 7.19 ns

TLB = 1 + 0.05 * PM = 6.1 ns
PM = 100 + 0.001 * PF = 102

PF = 2000

Problem 5: Producer-Consumer

This problem has you solve the classic "bounded buffer" problem with one producer and multiple consumer threads.

The program takes the number of consumers as an argument (defaulting to 1) and a sequence of numbers from stdin. We give you a couple of test sequences: shortlist and longlist. For more explanation of how this works, see the comment at the top of hw4.c

The producer thread reads the sequence of numbers and feeds that to the consumers. Consumers pick up a number, do some "work" with the number, then go back for another number.

The program as provided includes output from the producer and consumers. For reference, a working version of the code with a bounded buffer of size 10 running on shortlist with four consumers produces this output (the comments on the right are added): (NOTE: Your printed console output may not match what is shown identically due to the randomness of thread scheduling. However, your output should show all entries being produced in the correct order and consumed in the correct order).

```
turku% ./hw4 4 < shortlist
```

```
main: nconsumers = 4
  consumer 0: starting
  consumer 1: starting
  consumer 2: starting
  consumer 3: starting
  producer: starting
producer: 1
producer: 2
producer: 3
producer: 4
producer: 5
producer: 6
producer: 7
producer: 8
producer: 9
producer: 10
producer: 9
producer: 8
producer: 7
producer: 6
  consumer 0: 1
producer: 5
  consumer 1: 2
producer: 4
  consumer 2: 3
producer: 3
```

```
  consumer 3: 4

producer: 2

  consumer 0: 5

producer: 1

  consumer 1: 6

producer: read EOF, sending 4 '-1' numbers

  consumer 2: 7

  consumer 3: 8

  consumer 0: 9

  consumer 1: 10

producer: exiting

  consumer 2: 9

  consumer 3: 8

  consumer 0: 7

  consumer 1: 6

  consumer 2: 5

  consumer 3: 4

  consumer 3: exiting

  consumer 0: 3

  consumer 0: exiting

  consumer 2: 1

  consumer 2: exiting

  consumer 1: 2

  consumer 1: exiting
```

Finish the bounded-buffer code in hw4.c, adding synchronization so that the multiple threads can access the buffer simultaneously.

NOTE: If you are running your linux environment in a virtual machine, make sure to enable multiple cores. If you do not enable multiple cores, running your code on our machines may produce deadlocks that are not reproducable on your machine.

- There are really two problems here: managing the bounded buffer and synchronizing it. We suggest to write and test your bounded buffer implementation first before implementing synchronization.
- Your implementation must not spin-wait. There are several possible strategies. An excellent strategy with pthreads is to use a mutex and condition variables, i.e. using `pthread_cond_wait()` to wait when the buffer is empty or full.

Testing suggestions:

- You should be able to reproduce the output above.
- Try measuring the execution time with `/bin/time`. When running with longlist, doubling the number of consumers should roughly halve the execution time. What is the minimum possible execution time?