F2) V(I/O)rtual Potpourri (23 pts)

a) Page table must contain an entry for every virtual page. Since we have 4 GiB of VM and 1 MiB pages, we have 4 Ki-entries = 2^{12} entries. (2 pts)

Common mistake: 512 (# of physical pages)

b) The page table base register holds the address of a page table, which we know sits in physical memory, so it must be at least as wide as a physical address. Since we have 512 MiB of physical memory, a physical address is $\log_2$(512 MiB) = 29 bits. (2 pts)

c) In worst case, the TLB is cold except for code page (we specified that ALL of code fit in one page, so the code page would be in physical memory because of the function call to `update_hist()`). Accessed data in `scores` fits in single page because those 40B of continuous memory start at a page boundary, so 1 page fault for `scores[0]`. Then values in `scores` point to data in `histogram` that live in 10 separate pages (yes, this implies `MAX_SCORE` is a ridiculously large #), leading to 11 page faults total. Because code statement was slightly ambiguous (whether we meant all of the program's code vs. all of the function's code), we accepted both 11 or 12 page faults. (6 pts)

Partial credit: +5 pts for 10 page faults, +3 pts for 1, 2, 3 page faults, +1 pt for >12 page faults

d) Best case means the TLB is already filled with the mappings for the pages holding the two arrays. Because the elements of `histogram` we access are determined solely by the entries of `scores`, the best case is we only access `histogram` elements in the same page (e.g. all entries in `scores` are the same). Because the index into `scores` always increments, that sets our maximum loop iterations. With 1 page for code and 1 page for `histogram`, we assume the remaining 30 TLB entries contain the `scores` data. 30 pages hold 30*(1MiB/4B) = $30*2^{18}$ ints. (7 pts)

Partial credit: 2^{20} (2 pts), 2^{18} (3 pts), 2^x where x > 0 (1pt)
32 (2 pts), 31 (3 pts), 30 (4 pts)

e) We want to improve the temporal locality of the data. Since our data set is mostly repeats of a small number of scores, we can generate more hits by sorting the scores. (6 pts)

Partial credit: warm up TLB (2 pts)
redefine histogram so that scores in clusters map to nearby indices (4 pts)

<u>Question 6</u>: *It's Virtual Insanity!* (13 points, 26 minutes)

Our 32-bit uniprocessor machine has 1 GiB of RAM with 1 KiB pages, a fully-associative TLB that holds 8 entries and uses LRU, and a direct-mapped, write-back *data* cache with 32 B blocks and 32 slots. The *instruction* cache is 256 B and fully-associative with 32 B blocks.

a) (1 point) What is the maximum number of valid entries in the page table for a single process? Answer in IEC.

1 Mi-entries

Page table valid entries set by size of PM. 1 GiB / 1 KiB = 1 Mi-entries.

+0.5 points for 1MiB, +0.5 points for 2^{20}

b) (1 point) What is the TLB Reach of our system? ___8 KiB_____

8 TLB entries that refer to a 1 KiB page each. TLB Reach = 8*1KiB.

No partial credit – full credit given for 2^{13} B.

Examine the following function. Assume the entire program's code takes the entirety of one page and `sizeof(int)=sizeof(int *)=4`.

```
void addConst(int *ptr, char c) {
    for(int i = 0; 1; i+=4)
        ptr[i] += c;
}
```

***If all of (c,d,f) were answered as if for i++, -2 points total after the scores for those are added (min 0)**

c) (2 points) If `ptr[]` lives in disk and `ptr[0]` is page-aligned, what is the TLB hit rate for data accesses only?

___127/128___

Lives in disk means TLB miss on `ptr[0]`. Loop jumps 4 ints = 16 B per iteration. 1 KiB/16 B = 64 array indices accessed per page. Since += is a read and a write, 1 TLB miss per 128 memory accesses in a page.

+ 1 pt total for answering 63/64 based on missing that += does read and write
+ 2 pt total for answering 511/512 if answering based on i++ interpretation*

d) (2 points) If `ptr[]` lives in disk and `ptr[0]` is page-aligned, what fraction of D$ misses are also TLB misses?

____1/32_____

From part (c), 1 TLB miss every 128 memory accesses. Lives in disk means not in cache. 32 B/16 B = 2 array indices accessed per cache block. Since += is a read and a write, you have 1 D$ miss per 4 memory accesses in a cache block. The fraction is then 4/128 = 1/32.

Partial credit only for slight arithmetic errors with work shown, which were evaluated individually.

e) (1 point) If `ptr[0]` is in physical memory, what is the *minimum* value of `i` that could cause a **page fault**?

_____4_______

`ptr[0]` is in valid entry in page table, but no mention of where in page. If we assume it is one of the last 3 integer spaces in the page, then the next loop iteration (`ptr[4]`) can cause a page fault if that page is invalid.

+ 0.5 pt for 256 based on assuming page alignment
+ 1 pt for answering 1 based on `i++` interpretation*

f) (1 point) If `ptr[0]` is in physical memory, what is the *minimum* value of `i` that could cause a **protection fault**?

_____0_______

Even with `ptr[0]` in valid entry in page table, no mention of access rights. The loop both reads and writes, so will cause a protection fault if the process is missing one of those access rights for that page.

+ 0.5 pt for 4
+ 0.5 pt for answering 1 based on i++ interpretation*

g) (2 points) If `ptr[0]` is in physical memory, what is the *maximum* value of `i` that causes the first **cache miss** in the loop?

____256______

For maximum, assume I$ already holds all instructions of the loop and that the D$ is filled with the first entries of `ptr[]`. The D$ holds 32 * 32B = 1 KiB of data = 256 ints.

+ 0.5 pt for answering 32 based on a proposed scenario of one block in the cache
+ 0.5 pt for answering 64, the max number of iterations

h) (3 points) If `ptr[0]` is in physical memory, what is the *maximum* value of `i` that causes the first **TLB miss** in the loop? You may leave your answer as a product.

___7*2^8_____

For maximum, assume TLB is full of page entries needed for our function. Each page holds 1 KiB = 2^8 ints. Need one page for code/instructions, so 7 remain for `ptr[]` entries. So first TLB miss (and replacement) will occur when `i` = $7*2^8$.

+ 2 pt for answering 2048 based on ignoring the code page
+ 1 pt for answering 256 based on assuming only the page holding `ptr[0]` was in physical memory
+ 1.5 pt for answering 512 based on counting the number of iterations

SID: ________________

Question 9: Virtual Memory (8 pts)

This election season, the US will computerize the voting system. There were approximately 2^{27} voters in 2012. There are four candidates in the running and so each voter will submit letter A, B, C, or D. The votes are stored in the `char votes[]` array.

The following loop will count the votes to determine the winner. We are given a 1 MiB byte-addressed machine with 4 MiB of VM and 128 KiB pages. Assume that `votes[]` and `candidates[]` are page-aligned and `i` is stored in a register.

```
#define NUM_VOTERS 134217728                    // 2^27
int candidates[] = {0,0,0,0};                   // initialize to 0s
for (int i = 0; i < NUM_VOTERS; i++) {          // Loop 1
      if (votes[i] == 'A')  candidates[0]++;
      if (votes[i] == 'B')  candidates[1]++;
      if (votes[i] == 'C')  candidates[2]++;
      if (votes[i] == 'D')  candidates[3]++;
}
```

a) How many bits wide are the following? [2 pt]

VPN __5__ Page Offset __17__

PPN __3__ Page Table Base Register __20__

b) We are given a fully-associative TLB with 4 entries and LRU replacement policy. One entry is reserved for the Code. In the *best case scenario*, how many votes will be counted before a TLB miss occurs? [2 pt]

2^{18}

Best case: TLB already has code page, candidate page, and 2 votes pages loaded. One page is 2^{17}B. votes is a character array so each page holds 2^{17} votes. $2 * 2^{17} = 2^{18}$ votes

---

We want to improve our machine by expanding the TLB to hold 8 entries instead of 4. We also revised our `for` loop, which replaces `Loop 1`. Assume `i` and `vote` are stored in registers.

```
for (int i = 0; i < NUM_VOTERS; i++) {      // Loop 2
      char vote = votes[i];
      if (vote == 'A')      candidates[0]++;
      if (vote == 'B')      candidates[1]++;
      if (vote == 'C')      candidates[2]++;
      if (vote == 'D')      candidates[3]++;
}
```

c) Now how many votes can be counted before a TLB miss in the *best case scenario*? [2 pt]

$6 * 2^{17}$

Even though we have 1 access per vote instead of 4 with the new for loop, this does not change the fact that in the best case, we will only miss in the TLB if we go through all our pre-loaded pages. Since our TLB can now hold 6 pages for the votes pages, we can get $6 * 2^{17}$ votes before a miss.

SID: ________________

d) In the *worst case scenario*, how many TLB misses would occur if this improved loop ran to completion? In other words, what is the highest number of TLB misses possible when running this loop? [2 pt]

$\boxed{2^{10} + 2}$

In the worst case scenario, we start out with a cold TLB with nothing that is usable/valid when the loop begins. We will miss to fetch the code page and then miss to load in the candidate page. Those are 2 misses, and those pages will remain in the TLB throughout the execution of the loop since this TLB is LRU and the code and candidate page will constantly be accessed. This leaves us with misses for each page we need to fetch for the votes array. Our votes array is 2^{27} bytes and our page size is 2^{17} bytes. $2^{27}/2^{17} = 2^{10}$ pages, which we all cause a TLB when they are first fetched into the TLB. Thus, we have $2^{10} + 2$ TLB misses.