CSE 351 Section 8 – Solutions to Additional Problems from Autumn 2017 Final

1: Caching

We have 64 KiB of RAM and a 2-KiB L1 data cache that is 4-way set associative with 32-byte blocks and random replacement, write-back, and write allocate policies.

(A) Calculate the TIO address breakdown: [1.5 pt]

Tag bits	Index bits	Offset bits
7	4	5

16 address bits. $\log_2 32 = 5$ offset bits. 2-KiB cache = 64 blocks. 4 lines/set → 16 sets.

(B) How many management bits (bits *other* than the block data) are there in every line in the cache? [1 pt]

Tag bits + Valid bit + Dirty bit (write-back) **9** bits

(C) The code snippet below accesses an array of `doubles`. Assume `i` is stored in a register. Calculate the **Miss Rate** if the cache starts *cold*. [2.5 pt]

```
#define ARRAY_SIZE 256
double data[ARRAY_SIZE];        // &data = 0x1000 (physical addr)
for (i = 0; i < ARRAY_SIZE; i += 1)
    data[i] /= 100;
```

1/8 = 12.5%

Access pattern is read then write to `data[i]`. Stride = 1 double = 8 bytes. 32/8 = 4 strides per block. The offset of `&data` is 0b00000, so we start at the beginning of a cache block. First access (read) is a compulsory miss and the next 7 (over 4 different addresses) are hits. Since we never revisit indices, this pattern continues for all cache blocks.

(D) For each of the proposed (independent) changes, write **IN** for "increased", **NC** for "no change", or **DE** for "decreased" to indicate the effect on the **Miss Rate** for the code above: [4 pt]

Use `float` instead **DE**

Half the cache size **NC**

Split the loop body into: **DE**

```
data[i] /= 10;
data[i] /= 10;
```

No-write allocate **NC**

Using `floats` means more strides/block. We never revisit blocks, so cache size doesn't matter. Since the entire array fits in the cache, running it through a 2nd loop results in all hits. No-write allocate has no effect because all of our misses are on *reads*.

(E) Assume it takes 100 ns to get a block of data from main memory. If our L1 data cache has a hit time of 2 ns and a miss rate of 3%, what is the average memory access time (AMAT)? [1 pt]

AMAT = HT + MR x MP = 2 + 0.03 x 100 = 5 **5** ns

2: Processes

(A) The following function prints out four numbers. In the following blanks, list three possible outcomes: [3 pt]

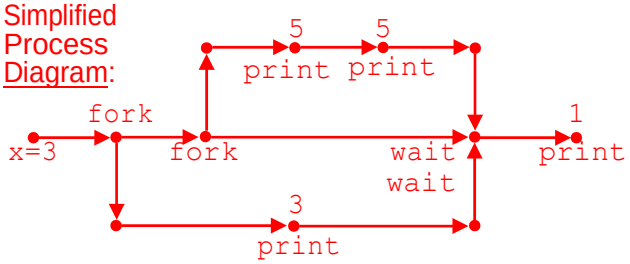
```
void concurrent(void) {
   int x = 3, status;
   if (fork()) {
      if (fork() == 0) {
         x += 2;
         printf("%d",x);
      } else {
         wait(&status);
         wait(&status);
         x -= 2;
      }
   }
   printf("%d",x);
   exit(0);
}
```

(1) _3, 5, 5, 1_____

(2) _5, 3, 5, 1_____

(3) _5, 5, 3, 1_____

Simplified Process Diagram:



(B) For the following examples of exception causes, write "**N**" for intentional or "**U**" for unintentional from the perspective of the user process. [2 pt]

System call __N__ Hardware failure __U__

Segmentation fault __U__ Mouse clicked __U__

Syscalls are part of code you are executing. The others are external to the process.

(C) Briefly define a **zombie** process. Name a process that can *reap* a zombie process. [2 pt]

Zombie process: A process that has ended/exited but is still consuming system resources.
Reaping process: The parent process or `init/systemd` (PID 1).

(D) In the following blanks, write "**Y**" for yes or "**N**" for no if the following need to be updated when **execv** is run on a process. [2 pt]

Page table __Y__ PTBR __N__ Stack __Y__ Code __Y__

The process already has its own page table, so while we will need to invalidate PTEs from the old process image, we don't need to create another page table, so the PTBR can remain the same. We replace/update the old process image's virtual address space, including Stack and Code.