CSE 351 Spring 2025 Final Exam

Name: ______Sample Solution______________

UW NetID: ___________________________(@uw.edu)

Instructions:

- You have 110 minutes for this exam. Don't spend too much time on any one problem!
- The exam is CLOSED book and CLOSED notes (no summary sheets, no calculators, no mobile phones).
- The last page is a reference sheet. Feel free to detach it from the rest of the exam.
- When a box or line is provided, write your answers in the box or on the line provided.
- For answers that involve bubbling in a ◯ or □, make sure to fill in the shape completely.
- **Relax and take a few deep breaths. You've got this! :-).**

Total: 68 points

Q1: Caching and Code (6 pts)

You are using an x86-64 processor with 128 KiB of Physical address space. You have a direct mapped cache with a total size of 256 bytes and a cache block size of 16 bytes. The cache uses LRU replacement and write-allocate and write-back policies.

Assume that in main memory, array **A** starts at address 0x0 and array **B** starts immediately afterwards. Arrays **A** and **B** contain 1024 elements each. Assume that both **A** and **B** have been initialized to contain values. Assume that **i** is in a register and that the cache is initially empty at the start of the function.

```
#define STEP 2
#define SIZE 1024
int func(int A[], int B[]) {
   for (int i = 0; i < SIZE; i += STEP){
      A[i] = A[i] + i;
      B[i] = B[i] + i;
      A[i] = A[i] + i * i;
   }
}
```

a) (2 pts) Give the **miss rate** (as a simplified fraction or a %) for the code above:

5/12

The step is 2 * 4 bytes/int = 8 bytes. Therefore, 2 iterations will access the same blocks of A and B. The access pattern is:
A[0]: RM, WH
B[0]: RM, WH (evict A[0] block)
A[0]: RM, WH (evict B[0] block)
A[2]: RH, WH,
B[2]: RM, WH (evict A[2] block)
A[2]: RM, WH (evict B[2] block)
- end of block -
=> 5 misses out of 12 accesses

Q1 (continued)

b) (4 pts) For each of the changes proposed below, indicate how it would affect the **miss rate** of the code shown above *assuming that all other factors remained the same* as they were in the original problem. Select one of: “increase”, “no change”, or “decrease”.

i) Change Associativity to 2

- ○ **Increase**
- ○ **No Change**
- ● **Decrease**

Accesses to conflicting elements of A and B will no longer kick each other out

ii) Change STEP to 1

- ○ **Increase**
- ○ **No Change**
- ● **Decrease**

Reduces cold misses from moving on to a new block

iii) Change Cache size to 512 bytes

- ○ **Increase**
- ● **No Change**
- ○ **Decrease**

We did not have capacity problems, so this will have no impact

iv) Change Block size to 8 bytes

- ● **Increase**
- ○ **No Change**
- ○ **Decrease**

Increases cold misses, as every iteration will be a new block

Q2: Caching and Bits (8 pts)

You are given a cache with the following parameters:

Cache size: 512 bytes
Block size: 16 bytes
Associativity: Direct Mapped
Physical Address width: 15 bits
Cache Policies: write-allocate, write-back, LRU replacement

a) (2 pts) Give the **number of bits needed** for each of these:

Cache Block Offset: _____**4**_____ Cache Tag: _____**6**_____

b) (1 pt) How many **sets** does the cache have? _____**32**______

c) (1 pt) We define **tag overhead** as a comparison of the total combined tag and management **bits**, to the cache size in **bytes**:

$$tag\ overhead\ =\ \frac{total\ tag\ bits + total\ \ management\ bits}{cache\ size\ in\ bytes}$$

The cache described above uses 2 management bits (valid, dirty). Calculate the tag overhead of the cache, **in terms of bits per byte of cache**, leaving your answer as a **simplified fraction**:

Tag Overhead in bits per byte:

1/2

32 sets * (6 tag bits + 1 valid bit + 1 dirty bit) = 256 bits
=> 256 bits/512 bytes = 1/2

Q2 (continued)

d) (4 pts) For each of the changes proposed below, indicate how it would affect the **tag overhead** of this cache *assuming that all other factors remained the same* as they were in the original problem. Select one of: “increase”, “no change”, or “decrease”.

i) Change Associativity to 2

● **Increase** ○ **No Change** ○ **Decrease**

Decreases index bits => increases tag bits

ii) Change Physical Address width to 12 bits

○ **Increase** ○ **No Change** ● **Decrease**

Decreases tag bits

iii) Change Write-hit policy to Write-through

○ **Increase** ○ **No Change** ● **Decrease**

Eliminates dirty bit

iv) Change Block size to 8 bytes

● **Increase** ○ **No Change** ○ **Decrease**

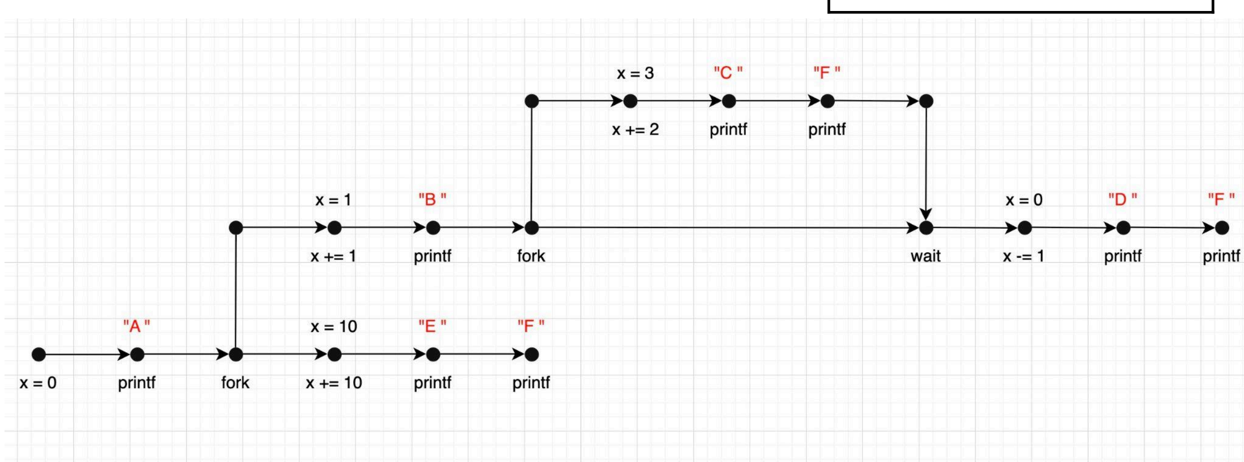
Decreases offset bits => increases index bits, tag stays the same, but total number of blocks goes up.

Q3: Processes (8 pts)

```c
void sunny() {
    int x = 0;
    printf("A ");

    if (fork() == 0) {
        x += 1;

        printf("B ");
        if (fork() == 0) {
            x += 2;
            printf("C ");
        } else {
            wait();
            x -= 1;
            printf("D ");
        }

    } else {

        x += 10;
        printf("E ");
    }
    printf("F ");

}
```

a) (2 pts) What is the total number of processes created by this function (include the original process that called **sunny**)?

3



Q3 (continued)

b) (2 pts) Which of the following outputs are *possible*. (Select ANY/ALL that are possible)

- [ ] A E F B D F C F (“D” cannot print before “C F” due to **`wait()`**)
- [x] **A E F B C F D F**
- [x] **A B E C F F D F**
- [ ] A B C F F D E F (Need a "C", "D", or "E" coming sometime before each "F")
- [x] **A B C F D F E F**

c) (2 pts) Is it possible to insert a single additional call to **`wait()`** in the function **`sunny`** to *guarantee* that “E F” is printed *last* in the output? If so, where? (Select ONE option)

- ○ Line 4
- ○ Line 7
- ○ Line 17
- ● **Line 19**
- ○ Line 24
- ○ Not possible

d) (2 pt) Select all possible values of **`x`** that could be printed out if we changed the print statement on line 23 to also print **`x`** (e.g **`printf("F%d ", x);`**). (Select ANY/ALL that are possible)

- [ ] 12
- [x] **0**
- [x] **10**
- [x] **3**
- [ ] 13

Q4: Virtual Memory (11 pts)

Assume we have a virtual memory system as follows:

- 8-bit virtual addresses, 6-bit physical addresses
- Page size = 16 bytes
- TLB: 2-way set associative, 4 entries total

a) (3 pts) **How many bits** will be used for:

Virtual page number (VPN)? ___**4**___ Physical Page number (PPN) __**2**___

TLB Tag? _____**3**_____

b) (2 pt) How many **total entries** are in this page table? (It is fine to leave your answer in powers of 2)

_______**2^4 or 16**_______

The current contents of the TLB and Page Table (partial) are shown below:

TLB (2-way set associative)

Set	Tag	PPN	Valid	Tag	PPN	Valid
0	0x5	0x3	1	0x0	–	0
1	0x7	0x2	1	0x2	0x1	0

Page Table (partial)

VPN	PPN	Valid
0x0	0x3	1
0x1	0x0	1
0x2	0x0	0
0x3	0x2	1
0x4	–	0
0x5	0x1	1
0x6	–	0
0x7	–	0

Q4 (continued)

c) (6 pts) Fill in the following information for the two virtual addresses provided. If you cannot determine the answer for a particular item write "ND" for non-determinable). Be sure to give your answer using the correct number of bits.

Virtual Address	VPN (give bits)	TLB tag (give bits)	TLB index (give bits)	PPN (give bits)	Physical Address (give bits)	TLB Miss? (Y/N)	Page Fault? (Y/N)
0x13	**0001**	**000**	**1**	**00**	**00 0011**	**Y**	**N**
0x24	**0010**	**001**	**0**	**ND**	**ND**	**Y**	**Y**

Q5: Memory Puzzles (11 pts)

```
1    #include <stdlib.h>
2    int zero = 0;
3    int* party() {
4        int cake;  (local variable declared)
5        return &cake;  (address to local variable returned out of the function)
6    }
7    int main(int argc, char *argv[]) {
8        char *str = "cse351";
9        int *foo = malloc(8);
10       int bar = 16;
11       int* dessert = party();  (attempt to reference non-existent variable)
12       free(foo);
13       return 0;
14   }
```

a) (8 pts) Consider the C code shown above. Assume that the **malloc** call succeeds and that all variables are stored in memory (not registers). Fill in the following blanks with "<" or ">" or "UNKNOWN" to compare the *values* returned by the following expressions just before **return 0** on line 13 executes.

&party _____<_____ **foo** **(&party in Code section, foo in heap)**

foo _____<_____ **&foo** **(foo in heap, &foo on stack)**

str _____<_____ **&dessert** **(str in literals, &dessert on stack)**

&bar _____>_____ **&zero** **(&bar on the stack, &zero in Static Data)**

b) (3 pts) The code above has an error that can best be described as: (Select ONE option)

- ○ A. Dereferencing a non-pointer
- ○ B. Memory leak
- ○ C. Reading uninitialized memory
- ● D. Referencing a nonexistent variable (On line 5, we should not return the address of a variable allocated on the stack. The stack frame for **party()** is deallocated when **party()** returns.)
- ○ E. Type mismatch
- ○ F. Passing a bad pointer to **free()**

Please list the line number(s) that are relevant to this error: 5 (with 4 and/or 11 also accepted)

Q6: Memory Allocation (3 pts)

Consider the diagram of a heap implemented using an implicit free list, where **each square represents 8 bytes** of memory. Allocated squares are shaded and contain a letter, while free squares are unshaded.

Assume an allocation request is made that results in a heap block with a **total size of 24 bytes**. For each of the following allocation strategies, **<u>fill the appropriate squares with the letter "D" to indicate the ones that would be allocated to fulfill this request.</u>** If it is not possible to fulfill the request, you may note "not possible" beneath the corresponding diagram. You may assume:

- Each part of the question is independent. The heap returns to its original state before each new allocation strategy is applied.
- The heap block (2 squares) labeled **C** was the block most recently allocated prior to this request.

a) First Fit:

		A	A	A	D	D	D			B	B				C	C	

Start of Heap End

The first-fit algorithm scans the heap from the beginning and selects the first free block large enough to satisfy the allocation request. In this case, it would allocate the first 24 bytes of the free block immediately following allocation A.

b) Next Fit:

		A	A	A	D	D	D			B	B				C	C	

Start of Heap End

The next-fit algorithm starts scanning from where the last allocation ended, continuing to the end of the heap and wrapping around to the beginning if needed. Since there isn't a large enough free block after C through the end of the heap, the search wraps around to the start of the heap. It then finds the first free block large enough for the 24-byte request and allocates the first 24 bytes of that block.

c) Best Fit:

		A	A	A						B	B	D	D	D	C	C	

Start of Heap End

The best-fit algorithm scans the entire heap to find the smallest free block that fits the request, aiming to minimize fragmentation. In this case, the 24-byte free block between B and C is an exact fit, making it the optimal choice.

Q7: C and Java (6 pts)

a) (6 pts) Use the following terms to fill in each empty cell in the table below with the most similar concept.

Terms (you cannot use a term more than once):

vtable	pointer	interpreter	garbage collection
calling convention	virtual machine	null terminator	struct
ArrayList resizing	object file	malloc	buffer overflow
executable	operand stack	java bytecode	explicit free list

C Concept	Java Concept
malloc	Object creation via `new`
x86 assembly instructions	**Java bytecode**
pointer	reference
`free`	**garbage collection**
null terminator	string length stored in header
`realloc`	**ArrayList resizing**

Q8: Assembly Fun (6 pts)

Fill in the remainder of the C code that corresponds to the x86-64 assembly code given below:

```
mystery:
        movl    (%rsi), %eax
        cmpl    %eax, (%rdi)
        jle     .L2
        movl    %edx, (%rdi)
        ret
.L2:
        movl    %edx, (%rsi)
        ret
```

```
void mystery(int * a, int * b, int c){

    if (*a > *b){
        *a = c;
    } else {
        *b = c;
    }
```

Q9: Pointers & Memory (9 pts)

We are using a 64-bit x86-64 machine (**little endian**). Refer to the disassembly below showing where the function **fireworks** is in memory. **Read the questions before reading the assembly!**

```
0000000000401106 <fireworks>:
  401106:       48 83 ec 18             sub     $0x18,%rsp
  40110a:       89 7c 24 0c             mov     %edi,0xc(%rsp)
  40110e:       83 7c 24 0c 00          cmpl    $0x0,0xc(%rsp)
  401113:       7e 14                   jle     401129 <fireworks+0x23>
  401115:       8b 44 24 0c             mov     0xc(%rsp),%eax
  401119:       83 e8 01                sub     $0x1,%eax
  40111c:       89 c7                   mov     %eax,%edi
  40111e:       e8 e3 ff ff ff          callq   401106 <fireworks>
  401123:       48 c1 e0 02             shl     $0x2,%rax
  401127:       eb 05                   jmp     40112e <fireworks+0x28>
  401129:       b8 11 00 00 00          mov     $0x11,%eax
  40112e:       48 83 c4 18             add     $0x18,%rsp
  401132:       c3                      ret
```

a) (4 pts) What are the values (in hex) stored in each register shown after the following x86 instructions are executed? **Use the appropriate bit widths**. If a register's value cannot be determined, write N/A. Assume registers are initialized as shown in the table below.

```
movslq 0x3(%rax), %rcx

leaw   0x1(%rsi,%rsi,2), %di
```

Register	*Value (in hex):*
%rax	0x0000 0000 0040 111e
%rsi	0x0000 0000 0000 000A
%rcx	0xffff ffff c148 ffff
%di	0x001f

Q9 (continued)

b) (4 pts) Complete the C code below to fulfill the behaviors described in the inline comments using pointer arithmetic. Let **`int* intP = 0x401120`**.

```
short* v1 = (short*)((long* or double*)intP + 2) // set v1 = 0x401130

// Assuming that the statement above succeeded:

((___char*___)v1)[___-6___] = 5;// set the byte at 0x40112a to 0x05
```

c) (1 pt) What would happen if you ran the **C code from part b)**? (Assume that the **`fireworks`** function is in memory as shown in the disassembly above, and that the code in b) has been filled in with correct values.) Give your answer in terms of how the C code would affect the execution of a program that later calls the fireworks function, and explain your answer in 2-3 sentences.

Running the C code from part b) would cause a segfault. The second line tries to modify an address in the instructions segment of memory, which is non-writable due to permissions, and so will cause the program to crash.

This is a Blank Page - enjoy!!!