Question F5: Caching [10 pts]

We have 16 KiB of RAM and two options for our cache. Both are two-way set associative with 256 B blocks, LRU replacement, and write-back policies. **Cache A** is size 1 KiB and **Cache B** is size 2 KiB.

(A) Calculate the TIO address breakdown for **Cache B**: [1.5 pt]

Tag bits	Index bits	Offset bits

(B) The code snippet below accesses an integer array. Calculate the **Miss Rate** for **Cache A** if it starts *cold*. [3 pt]

```
#define LEAP 4
#define ARRAY_SIZE 512
int nums[ARRAY_SIZE];            // &nums = 0x0100 (physical addr)
for (i = 0; i < ARRAY_SIZE; i+=LEAP)
    nums[i] = i*i;
```

(C) For each of the proposed (independent) changes, write **MM** for "higher miss rate", **NC** for "no change", or **MH** for "higher hit rate" to indicate the effect on **Cache A** for the code above:[3.5 pt]

Direct-mapped ______ Increase block size ______

Double LEAP ______ Write-through policy ______

(D) Assume it takes 200 ns to get a block of data from main memory. Assume **Cache A** has a hit time of 4 ns and a miss rate of 4% while **Cache B**, being larger, has a hit time of 6 ns. What is the worst miss rate Cache B can have in order to perform as well as Cache A? [2 pt]

Name:____

4. *Processes* (**12** points) In this problem, assume Linux.

   (a) Can the same program be executing in more than one process simultaneously?

   (b) Can a single process change what program it is executing?

   (c) When the operating system performs a context switch, what information does *NOT* need to be saved/maintained in order to resume the process being stopped later (circle all that apply):

   - The page-table base register
   - The value of the stack pointer
   - The time of day (i.e., value of the clock)
   - The contents of the TLB
   - The process-id
   - The values of the process' global variables

   (d) Give an example of an exception (asynchronous control flow) in which it makes sense to later re-execute the instruction that caused the exception.

   (e) Give an example of an exception (asynchronous control flow) in which it makes sense to abort the process.

Question F7: Processes [9 pts]

(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) ____________________

(2) ____________________

(3) ____________________

(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	_______	Hardware failure	_______
Segmentation fault	_______	Mouse clicked	_______

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

Zombie process:
Reaping process:

(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 _______ PTBR _______ Stack _______ Code _______

Question M1: Number Representation [8 pts]

(A) Take the 32-bit numeral **`0xC0800000`**. Circle the number representation below that has the *most negative* value for this numeral. [2 pt]

Floating Point   Sign & Magnitude   Two's Complement   Unsigned

(B) Let `float f` hold the value $\mathbf{2^{20}}$. What is the *largest power of 2* that gets rounded off when added to `f`? Answer in exponential form, not just the exponent. [2 pt]

Traffic lights display three basic colors: red (R), yellow (Y), and green (G), so we can use them to encode base 3! We decide to use the encoding 0↔R, 1↔Y, 2↔G. For example, $5 = 1\times3^1+2\times3^0$ would be encoded as **YG**. Assume each traffic light can only display one color at a time.

(C) What is the *unsigned* decimal value of the traffic lights displaying **RGYY**? [2 pt]

(D) If we have **9 bits** of binary data that we want to store, how many *traffic lights* would it take to store that same data? [2 pt]

Question M2: Design Question [2 pts]

(A) The machine code for x86-64 instructions are variable length. Name one advantage and one disadvantage of this design decision. [2 pt]

Advantage:
Disadvantage:

3. Virtual Memory (9 points)

Assume we have a virtual memory detailed as follows:

- 256 MiB Physical Address Space
- 4 GiB Virtual Address Space
- 1 KiB page size
- A TLB with 4 sets that is 8-way associative with LRU replacement

For the following questions it is fine to leave your answers as powers of 2.

a) How many bits will be used for:

Page offset? ___________

Virtual Page Number (VPN)? _________ Physical Page Number (PPN)? _________

TLB index? __________________ TLB tag? __________________

b) How many entries in this page table?

c) We run the following code with an empty TLB. Calculate the TLB <u>miss</u> rate for data (ignore instruction fetches). Assume **`i`** and **`sum`** are stored in registers and **`cool`** is page-aligned.

```
#define LEAP 8
int cool[512];
... // Some code that assigns values into the array cool
... // Now flush the TLB. Start counting TLB miss rate from here.
int sum;
for (int i = 0; i < 512; i += LEAP) {
  sum += cool[i];
}
```

TLB <u>Miss</u> Rate: (fine to leave you answer as a fraction) **______________**

2. Buffer Overflow (15 points)

The following code runs on a 64-bit x86 Linux machine. The figure below depicts the stack at point `A` before the function `gatekeeper()` returns. The stack grows downwards towards lower addresses.

```
void get_secret(char*);
void unlock(void);
void backdoor(void);

void gatekeeper() {
    char secret[8];
    char buf[8];

    fill_secret(secret);
    gets(buf);

    if (strcmp(buf, secret) == 0)
        unlock();
A:
    return 0;
}
```

0x7ffffffffffe0080	...
	Return Address
	`secret`
0x7ffffffffffe0068	`buf`

You are joining a legion of elite hackers, and your final test before induction into the group is gaining access to the CIA mainframe. `gatekeeper()` is a function on the mainframe that compares a password you provide with the system's password. If you try to brute force the password, you will be locked out, and your hacker reputation will be tarnished forever.

Assume that `fill_secret()` is a function that places the mainframe's password into the `secret` buffer so that it can be compared with the user-provided password stored in `buf`.

Recall that `gets()` is a `libc` function that reads characters from standard input until the newline ('\n') character is encountered. The resulting characters (not including the newline) are stored in the buffer that's given to `gets()` as a parameter. If any characters are read, `gets()` appends a null-terminating character ('\0') to the end of the string.

`strcmp()` is a function that returns 0 if two (null-terminated) strings are the same.

(a) Explain why the use of the `gets()` function introduces a security vulnerability in the program.

(b) You think it may be possible to unlock the mainframe, even without the correct password. Provide a hexadecimal representation of an attack string that causes the `strcmp()` call to return `0`, such that `unlock()` is then called. `gatekeeper()` should return normally, as to avoid raising any suspicion.

(c) The function `backdoor()` is located at address `0x0000000000000351`. Construct a string that can be given to this program that will cause the function `gatekeeper()` to unconditionally transfer control to the function `backdoor()`. Provide a hexadecimal representation of your attack string.

(d) How should the program be modified in order to eliminate the vulnerabilities the function `gets()` introduces?

(e) Describe two types of protection operating systems and compilers can provide against buffer overflow attacks. Briefly explain how each protection mechanism works.

4. Processes (10 points)

(a) After a context switch, the VPN to PPN mappings in the TLB from the previous running process no longer apply. A simple solution to this problem is to "shoot down" the TLB, by invalidating all the entries in the TLB, but this can often cause inefficiency if there is frequent context switching.

What additional information can be added to the TLB that can be utilized to reduce this inefficiency in the TLB on a context switch? *Hint: consider how processes can be uniquely identified by the MMU.*

(b) Suppose you are in control of the CPU and operating system, and you realize that you have a process A that requires a large uninterrupted chunk of CPU time to perform its important work. What would you adjust to ensure that this can happen?

(c) Consider an OS running a process A which incurs a timer interrupt at time t_1. The OS context switches to some other processes which do some work. Later, the OS context switches back to process A at time t_2. Note that process A was not run between t_1 and t_2.

Circle the items which are guaranteed to be the same at time t_1 and t_2.

Register `%rbx`	Process A's Page Table	Instruction Pointer
L1 Cache	Page fault handler code	Page Table Base Register

(d) Consider the following C program, running on an x86-64 Linux machine. The program starts running at function `main`. Assume that `printf` flushes immediately.

```
int main() {
    int* x = (int*) malloc(sizeof(int));
    *x = 1;
    if (fork() == 0) {
        spoon(x);
    } else {
        *x = 8 * *x;
        printf("%d\n", *x);
    }
}

void spoon(int* x) {
    printf("%d\n", *x);
    if (fork() == 0) {
        *x = 2 * *x;
    } else {
        *x = 4 * *x;
    }
    printf("%d\n", *x);
}
```

Provide **two** possible outputs of running this code.

Output 1: Output 2:

Question 4: Procedures & The Stack [24 pts.]

Consider the following x86-64 assembly and C code for the recursive function `rfun`.

```
// Recursive function rfun
long rfun(char *s) {
  if (*s) {
    long temp = (long)*s;
    s++;
    return temp + rfun(s);
  }
  return 0;
}

// Main Function - program entry
int main(int argc, char **argv) {
  char *s = "CSE351";
  long r = rfun(s);
  printf("r: %ld\n", r);
}
```

```
00000000004005e6 <rfun>:
  4005e6: 0f b6 07                movzbl (%rdi),%eax
  4005e9: 84 c0                   test   %al,%al
  4005eb: 74 13                   je     400600 <rfun+0x1a>
  4005ed: 53                      push   %rbx
  4005ee: 48 0f be d8             movsbq %al,%rbx
  4005f2: 48 83 c7 01             add    $0x1,%rdi
  4005f6: e8 eb ff ff ff          callq  4005e6 <rfun>
  4005fb: 48 01 d8                add    %rbx,%rax
  4005fe: eb 06                   jmp    400606 <rfun+0x20>
  400600: b8 00 00 00 00          mov    $0x0,%eax
  400605: c3                      retq
  400606: 5b                      pop    %rbx
  400607: c3                      retq
```

(A) How much space (in bytes) does this function take up in our final executable? [2 pts.]

(B) The compiler automatically creates labels it needs in assembly code. How many labels are used in `rfun` (including the procedure itself)? [2 pts.]

(C) In terms of the C function, what value is being saved on the stack? [2 pts.]

(D) What is the return address to `rfun` that gets stored on the stack during the recursive calls (in hex)? [2 pts.]

(E) Assume main calls `rfun` with `char *s = "CSE351"` and then prints the result using the `printf` function, as shown in the C code above. Assume `printf` does not call any other procedure. Starting with (and including) `main`, how many total stack frames are created, and what is the maximum depth of the stack? [2 pts.]

Total Frames:	Max Depth:

(F) Assume main calls `rfun` with `char *s = "CSE351"`, as shown in the C code. After `main` calls `rfun`, we find that the return address to `main` is stored on the stack at address `0x7fffffffdb38`. On the first call to `rfun`, the register `%rdi` holds the address `0x4006d0`, which is the address of the input string "CSE351" (i.e. `char *s == 0x4006d0`). Assume we stop execution prior to executing the `movsbq` instruction (address `0x4005ee`) during the **fourth** call to `rfun`. [14 pts.]

*For each address in the stack diagram below, fill in both the **value** and a **description** of the entry.*

*The **value** field should be a hex value, an expression involving the C code listed above (e.g., a variable name such as `s` or `r`, or an expression involving one of these), a literal value (integer constant, a string, a character, etc.), "unknown" if the value cannot be determined, or "unused" if the location is unused.*

*The **description** field should be one of the following: "Return address", "Saved %reg" (where reg is the name of a register), a short and descriptive comment, "unused" if the location is unused, or "unknown" if the value is unknown.*

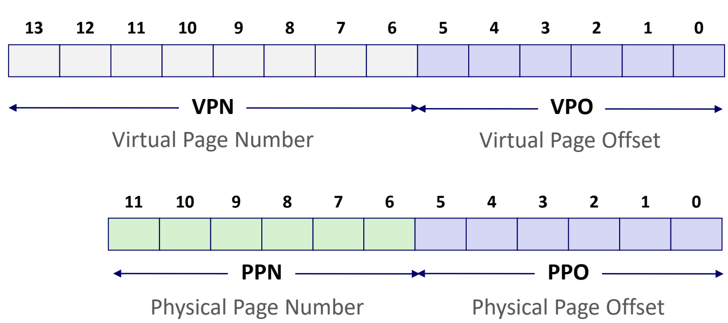
Memory Address	Value	Description
0x7fffffffdb48	unknown	%rsp when main is entered
0x7fffffffdb38	0x400616	Return address to main
0x7fffffffdb30	unknown	original %rbx
0x7fffffffdb28		
0x7fffffffdb20		
0x7fffffffdb18		
0x7fffffffdb10		
0x7fffffffdb08		
0x7fffffffdb00		

Summary of Address Translation Symbols

- Basic Parameters
  - $N = 2^n$ Number of addresses in virtual address space
  - $M = 2^m$ Number of addresses in physical address space
  - $P = 2^p$ Page size (bytes)
- Components of the virtual address (VA)
  - **VPO** Virtual page offset
  - **VPN** Virtual page number
  - **TLBI** TLB index
  - **TLBT** TLB tag
- Components of the physical address (PA)
  - **PPO** Physical page offset (same as VPO)
  - **PPN** Physical page number

Simple Memory System Example (small)

- Addressing
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes



Simple Memory System: Page Table

- Only showing first 16 entries (out of _____)
  - **Note:** showing 2 hex digits for PPN even though only 6 bits
  - **Note:** other management bits not shown, but part of PTE

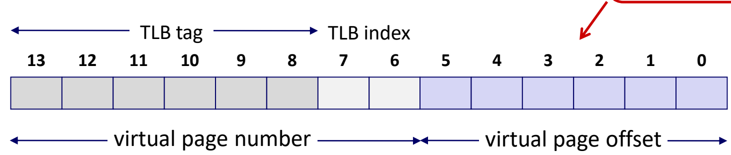
VPN	PPN	Valid
0	28	1
1	–	0
2	33	1
3	02	1
4	–	0
5	16	1
6	–	0
7	–	0

VPN	PPN	Valid
8	13	1
9	17	1
A	09	1
B	–	0
C	–	0
D	2D	1
E	–	0
F	0D	1

Simple Memory System: TLB

- 16 entries total
- 4-way set associative

Why does the TLB ignore the page offset?



Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Simple Memory System: Cache

Note: It is just coincidence that the PPN is the same width as the cache Tag

- Direct-mapped with K = 4 B, C/K = 16
- Physically addressed

cache tag						cache index				cache offset	
11	10	9	8	7	6	5	4	3	2	1	0
physical page number						physical page offset					

Index	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Index	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Current State of Memory System

TLB:

Set	Tag	PPN	V	Tag	PPN	V	Tag	PPN	V	Tag	PPN	V
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Page table (partial):

VPN	PPN	V	VPN	PPN	V
0	28	1	8	13	1
1	–	0	9	17	1
2	33	1	A	09	1
3	02	1	B	–	0
4	–	0	C	–	0
5	16	1	D	2D	1
6	–	0	E	–	0
7	–	0	F	0D	1

Cache:

Index	Tag	V	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Index	Tag	V	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Memory Request Example #1

Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: `0x03D4`

TLBT (bits 13–8) | TLBI (bits 7–6)

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

VPN (bits 13–6) | VPO (bits 5–0)

VPN ______ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

- Physical Address:

CT (bits 11–6) | CI (bits 5–2) | CO (bits 1–0)

11	10	9	8	7	6	5	4	3	2	1	0

PPN (bits 11–6) | PPO (bits 5–0)

CT ______ CI _____ CO _____ Cache Hit? ___ Data (byte) _______

Memory Request Example #2

Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: `0x038F`

TLBT (bits 13–8) | TLBI (bits 7–6)

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	0	1	1	1	1

VPN (bits 13–6) | VPO (bits 5–0)

VPN ______ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

- Physical Address:

CT (bits 11–6) | CI (bits 5–2) | CO (bits 1–0)

11	10	9	8	7	6	5	4	3	2	1	0

PPN (bits 11–6) | PPO (bits 5–0)

CT ______ CI _____ CO _____ Cache Hit? ___ Data (byte) _______

Memory Request Example #3

Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: `0x0020`

TLBT (bits 13–8) | TLBI (bits 7–6)

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0

VPN (bits 13–6) | VPO (bits 5–0)

VPN ______ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

- Physical Address:

CT (bits 11–6) | CI (bits 5–2) | CO (bits 1–0)

11	10	9	8	7	6	5	4	3	2	1	0

PPN (bits 11–6) | PPO (bits 5–0)

CT ______ CI _____ CO _____ Cache Hit? ___ Data (byte) _______

Memory Request Example #4

Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: `0x036B`

TLBT (bits 13–8) | TLBI (bits 7–6)

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	1	0	1	0	1	1

VPN (bits 13–6) | VPO (bits 5–0)

VPN ______ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

- Physical Address:

CT (bits 11–6) | CI (bits 5–2) | CO (bits 1–0)

11	10	9	8	7	6	5	4	3	2	1	0

PPN (bits 11–6) | PPO (bits 5–0)

CT ______ CI _____ CO _____ Cache Hit? ___ Data (byte) _______