Northeastern University
CS5600: Computer Systems: Fall 2021

Final Exam

- This exam is **120 minutes**.
- Stop writing when "time" is called. *You must turn in your exam; we will not collect it.* Do not get up or pack up in the final ten minutes. The instructor will leave the room 123 minutes after the exam begins and will not accept exams outside the room.
- There are **11** problems in this booklet. Many can be answered quickly. Some may be harder than others, and some earn more points than others. You may want to skim all questions before starting.
- **This exam is closed book and notes. You may not use electronics: phones, tablets, calculators, laptops, etc.** You may refer to ONE two-sided 8.5x11" sheet.
- If you find a question unclear or ambiguous, be sure to write any assumptions you make.
- Follow the instructions: if they ask you to justify something, explain your reasoning and any important assumptions. **Write brief, precise answers. Rambling brain dumps will not work and will waste time.** Think before you start writing so that you can answer crisply. Be neat. If we can't understand your answer, we can't give you credit!
- If the questions impose a sentence limit, we will not read past that limit. In addition, *a response that includes the correct answer, along with irrelevant or incorrect content, will lose points.*
- Don't linger. If you know the answer, give it, and move on.
- **Write your name and NUID on the document in which you are working the exam.**

Do not write in the boxes below.

I (xx/10)	II (xx/10)	III (xx/10)	IV (xx/10)	V (xx/20)	VI (xx/20)	VII (xx/20)	Total (xx/100)

I Basics (10 points)

1. [3 points] What's the functionality of the register cr3 in x86 CPUs?
Explain in 1 sentence.

cr3 points to the root of virtual memory page tables

2. [3 points] Below is a buggy C program:

```
1   #include "stdio.h"
2   #include "stdlib.h"
3
4   int *ptr1, *ptr2;
5
6   void f() {
7       int a = 0;
8       int* b = (int*) malloc(sizeof(int));
9       ptr1 = &a;
10      ptr2 = b;
11  }
12
13  int main() {
14      f();
15      ... // run some code
16      *ptr1 = 1;
17      *ptr2 = 2;
18      ... // run some other code
19      free(ptr2);
20  }
```

Why is this program buggy?
Explain why in 1-2 sentences.

At line 16, the program writes to a piece of invalid memory. The memory is pointed by pointer "ptr1" and is part of a stack frame (of function "f") that has been destroyed.

3. [4 points] Read the following shell commands and then answer questions.

```
$ echo "abc" > /tmp/file1
$ ln /tmp/file1 /tmp/hardlink
$ ln -s /tmp/file1 /tmp/softlink
$ rm /tmp/file1
```

Circle your answers below.

After running the above commands, "`cat /tmp/hardlink`" will print:

A. nothing (empty string)

B. "abc"

C. an error

After running the above commands, "`cat /tmp/softlink`" will print:

A. nothing (empty string)

B. "abc"

C. an error

B, C

II Authentication and Access control (10 points)

4. [4 points] In class, we have learned about three types of authentications: (1) authentication based on what you know, (2) authentication based on what you have, and (3) authentication based on what you are. Describe an authentication process that uses all three types of authentications (either an existing one or a new one that you design).

Write down your authentication process in 2-3 sentences and pinpoint which step uses what authentication type.

One possible example:

An example is NEU login with two-factor authentication with iPhone. NEU authentication needs my password (what you know) and my cellphone confirmation (what you have); in addition, iPhone login requires face id (what you are).

5. [6 points] In a Unix system,

- there are three users: `user1`, `user2`, and `user3`.
- `user1` and `user2` are in the same group `cs5600`; `user3` is not.
- file `/file1` and directory `/dir1` both are created by `user1` with group `cs5600`.

Here is the access control list of `file1` and `dir1`: (the access control list format follows "`ls -l`")

```
/file1:  rw-r-----
/dir1:   r-xr-xr-x
```

Read the following commands and choose if Unix will pass or reject these accesses.
(Note: the shell prompt shows the user.)

Circle your answers.

[Pass / Reject] `user1$ cat /file1`

[Pass / Reject] `user2$ cat /file1`

[Pass / Reject] `user3$ cat /file1`

[Pass / Reject] `user1$ cd /dir1`

[Pass / Reject] `user1$ ls /dir1`

[Pass / Reject] `user1$ touch /dir1/file2`

answer: Pass, Pass, Reject, Pass, Pass, Reject

III Process and Buffer overflow (10 points)

6. [10 points] Read the following code and answer questions. Note that `vulnerable_block_read` is the malicious version of Lab4's `block_read` which reads data supplied by a remote attacker.

```
1     #include "stdio.h"
2
3     int f() {
4         char buf[4096];
5         int ret = vulnerable_block_read(&buf, 10, 1);
8         printf("first byte of block#10: %c\n", buf[0]);
9         return ret;
10    }
11
12    int main() {
13        f();
14   }
```

Here is the program's stack after executing line 4.

```
high address |  ...    | (a)
             +---------+
             |  %rip   | (b)
             +---------+
             |  %rbp   | (c)
             +---------+
             |         |
             |   buf   | (d)
             | [4096]  |
             |         |
             +---------+ <-- %rsp
low address  |  ...    | (e)
```

(4 points) The `vulnerable_block_read` conducts a buffer overflow attack by sending a payload larger than 4096 bytes. When does this attack take effect?
Write down the line number after which the attacker's code is executing. Explain why it's the line number you have chosen, using no more than two sentences.

line 9

When "f()" returns, the CPU pops the %rip on stack which now points to the attacker's code. Then the CPU will start running the attacker's code.

(3 points) Stack canaries are one way to detect stack smashing attacks. The compiler inserts "canaries" (random values) on the stack to check if stack data were overwritten by malicious payloads. Where should the canaries be placed on the stack?
Circle your answer below.
(Note: (a)–(e) are labels on the stack figure above.)

A. between (a) and (b)

B. between (b) and (c)

C. between (c) and (d)

D. between (d) and (e)

B or C

(3 points) Are the following statements True or False?
Circle your answers.

[True / False] Doubling `buf` size to be 8192 bytes could secure the program because the payload is not large enough to overflow the stack then.

[True / False] It's safe for compilers to fix canary values.

[True / False] The canary-check happens when the function `f` returns.

False, False, True

IV Scheduling and Concurrency (10 points)

7. [10 points] Consider the setup below, which follows the scheduling game from class. Time is divided into epochs. Jobs arrive at the **very beginning** of the epoch listed under "arrival"; after they have been given the number of CPU epochs listed under "running", they leave. Assume processes do no I/O and no synchronization.

```
       arrival    running
P1     0          5
P2     1          2
P3     2          3
```

(3 points) The system runs the Shortest Time-to-Completion First (STCF), a classic preemptive scheduling algorithm.

Write down the process scheduled for each epoch.

```
0    1    2    3    4    5    6    7    8    9
```

(3 points) Later, the system administrator assigns priorities to processes as follows:

```
P1: low
P2: medium
P3: high
```

Then, the system runs the preemptive priority scheduling algorithm.

Write down the process scheduled for each epoch.

```
0    1    2    3    4    5    6    7    8    9
```

(4 points) Later, the programmers add synchronization to the processes as follows. Note that P1 and P3 share the same `lock`.

```
P1:
  acquire(&lock);
  run1();  // takes 5 epochs
  release(&lock);


P2:
  run2();  // takes 2 epochs


P3:
  run3a();  // takes 2 epochs
  acquire(&lock);
  run3b();  // takes 1 epochs
  release(&lock);
```

The system still uses the preemptive priority scheduling algorithm.

Write down the process scheduled for each epoch now.

```
0    1    2    3    4    5    6    7    8    9
```

Solution:

STCF:

```
0   1   2   3   4   5   6   7   8   9
P1  P2  P2  P3  P3  P3  P1  P1  P1  P1
```

Priority scheduling:

```
0  1  2  3  4  5  6  7  8  9
P1 P2 P3 P3 P3 P2 P1 P1 P1 P1
```

Priority + synchronization (priority inversion):

```
0  1  2  3  4  5  6  7  8  9
P1 P2 P3 P3 P2 P1 P1 P1 P1 P3
```

V Virtual Memory (20 points)

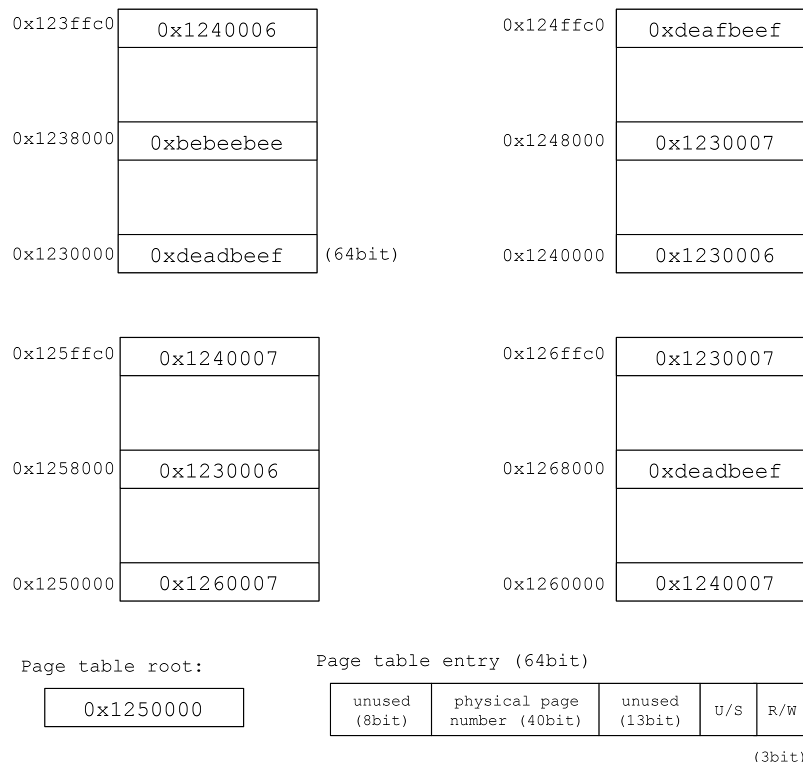
8. [20 points] Assume a system `vm5600` that uses **bit-addressable** memory; that is, two consecutive memory addresses are referring to individual bits (instead of bytes in x86 CPUs). Like x86, `vm5600` uses paging for virtual memory; but unlike x86, the page size is **8KB**. `vm5600` uses a 2-level page table and a virtual address that has **36 bits** for addressing, broken down as follows:

```
[ 10 bits | 10 bits | 16 bits ]
```

The top 10 bits index into the L1 page table; the next 10 bits index into a L2 page table; the bottom 16 bits determine the **bit** offset into the physical page.

The figure below is a snapshot of `vm5600` memory that shows four physical pages.
In the figure,

- memory is drawn in 64-bit units.
- the address on the left of a unit is the starting physical address of these 64 bits.
- the page table root and page table entry (64bit) are described at the bottom.
- `vm5600` uses big endian. (If you don't understand, you can safely ignore this note.)



(4 points) Are the following statements True or False?
Circle your answers.

[True / False] Virtual addresses `0x1230000` and `0x1230001` must be located on the same physical page.

[True / False] Virtual addresses `0x123ffff` and `0x1240000` must be located on the same physical page.

[True / False] Virtual addresses `0x1235000` and `0x1240000` must be located on the same physical page.

[True / False] Virtual addresses `0xdeadbeef` and `0xbeebee` must be located on the same physical page.

True, False, False, False

Virtual memory with paging cannot change offsets within a page, and the `vm5600`'s offset is the last 16bit (that's 4 digits in hex). So, by ignoring the last four digits in hex, all choices have the same virtual page numbers must be located on the same physical page.

(2 points) `vm5600`'s physical page numbers (PPN) use 40-bit addresses, how many bits does a physical address have?
Write down the number of bits and explain why in 1 sentence.

40 + 16 = 56.

The physical address have two parts, PPN (40 bits) and offset (16 bits).

(3 points) In `vm5600`, what's the maximum memory size of a process? And what's the maximum memory size of a machine? (Note: `vm5600` is bit-addressable and the page size is 8KB. 2^{10}B is 1KB; 2^{20}B is 1MB; 2^{30}B is 1GB; 2^{40}B is 1TB; 2^{50}B is 1PB.)

Circle your answer.

A. 2 MB and 4 GB

B. 2 GB and 4 TB

C. 2 GB and 8 TB

D. 8 GB and 8 PB

D

process: VA is 36-bit, bit-addressable; that is (2^{36} bits) / (8 bits/Byte) = 2^{33} Bytes = 2^{3} GB.

machine: PA has 40 bits for PPN; each page is 8KB; then 2^{40} * 8 KB = 8 PB.

(3 points) Given a virtual address `0xffe008000`, please split the address into L1 index (10bit), L2 index (10bit), and page offset (16bit).

Write down the values in hexadecimal numbers (for example, `0x12345`).

L1 index:

L2 index:

offset:

L1 index: 0x3ff

L2 index: 0x200

offset: 0x8000

Note: an index is an ordinal number that indicates a PTE in the page table. For example, L1 index with 0x3ff means the 1023th (i.e. 0x3ff) PTE in the L1 page table. And, in `vm5600`, `sizeof(PTE)` is 8 Bytes.

(4 points) Given the `vm5600` memory snapshot above, what is the physical address of the virtual address `0xffe008000`?

Write down the physical address in hexadecimal numbers (for example, `0x12345`), and explain the page walk in 2-3 sentences.

PA: 0x1238000

page walk:

- root: page 0x1250000
- then, the L1 index (0x3ff) points to the last entry (1023th), which leads to page 0x1240000;
- then, the L2 index (0x200) points to the 512th entry, which leads to page 0x1230000;
- finally, concatenating the offset (0x8000) results in "0x1238000".

(4 points) Here is a C program that runs on `vm5600`.

```
int main() {
   int64_t *ptr1 = (int64_t *) 0xffe008000;
   char *ptr2 = (char *) 0x3ff8000;
   printf("int64 ptr1: 0x%x, char ptr2: 0x%x\n", *ptr1, *ptr2);
}
```

Note: `%x` prints the number in hexadecimal. `sizeof(int64_t)` is 8 (bytes), and `sizeof(char)` is 1 (byte).

Given the `vm5600` memory snapshot above, what will this piece of code print?
Write down the output of this code snippet.

You will get points if you answer:

"int ptr1: 0xbebeebee, char ptr2: 0xbebeebee"

OR

"int ptr1: 0x0, char ptr2: 0x0" (the real correct answer)

OR

"int ptr1: 0xbebeebee, char ptr2: 0xffffffee"

[note: this question is too tricky, and we are giving points to those who clearly show they understand walking the page tables.]

Both ptr1 and ptr2 point to the physical address 0x1238000. The 64-bit unit of 0x1238000 contains: "0x0000 0000 bebe ebee" (with leading zeros).

What printed on screen depends on how `printf` interprets the pointers. In our case, both ptr1 and ptr2 would be treated as `%x`, which is an "Unsigned hexadecimal integer" (see: `http://www.cplusplus.com/reference/cstdio/printf/`). So, the expected outcome should be the first 8Bytes of data pointed by physical address 0x1238000, namely "0x00000000".

VI I/O & File system & Crash recovery (20 points)

9. [20 points] `disk5600` is a new disk that exposes a sequence of blocks. Each block is **16KB**.

(3 points) Are the following statements True or False?
Circle your answers.

[True / False] Programmed I/O can be used to transfer both commands and data.

[True / False] DMA is good at transferring large chunks of data to and from devices.

[True / False] DMA always works with the interrupt mechanism.

True, True, False

(3 points) A file system `fs-disk5600` uses 8 blocks as bitmap to track free/used blocks.
What's the maximum size of the `fs-disk5600`?

16GB

Because each *bit* represents a block:

#blocks = 8 * #bits_in_a_block = 8 * (16 * 1024 * 8) = 2^{20}

`fs-disk5600` size: 16GB = 2^{20} * 16KB

(4 points) In `fs-disk5600`, a file's inode has 10 direct blocks and 10 indirect blocks. (hint: we learned about "indirect blocks" when introducing Unix inodes.)

Choose the size of a block pointer that *you think* makes sense. (Note: a block pointer should be able to address all blocks in the `fs-disk5600`.)
Circle your answer.

A. 16bit

B. 32bit

C. 64bit

D. 128bit

What is the maximum size of a file, based on your choice above?
Write down both the file size and your calculation.
(Note: it is okay to write the size in a product form, for example, "$123 \times 456 \times 789$TB".)

Choice A doesn't make sense because it cannot address a `fs-disk5600`, given there are at most 2^{20} blocks. B, C, and D are all okay.

(if B) #blocks = 10 + (16KB / 32bit) * 10 = 40970

file_size = 40970 * 16KB

(if C) #blocks = 10 + (16KB / 64bit) * 10 = 20490

file_size = 20490 * 16KB

(if D) #blocks = 10 + (16KB / 128bit) * 10 = 10250

file_size = 10250 * 16KB

(4 points) Assume one inode is always stored within one block (inode size $<$16KB), and buffer-cache is empty (meaning the process will read all data from `disk5600`).
If a process needs to read 161KB data from the beginning of a file (namely, file offset is 0), how many blocks the process will read from `disk5600`?
Write down the number of blocks and explain what these blocks are.

13

1 for file's inode

10 for direct data blocks (160KB data)

1 for the indirect block

1 for data block pointed by the indirect block (the final 1KB data)

(Note: if you assume starting from root directory and adding those blocks, that's fine.)

(6 points) `fs-disk5600` uses redo logging for crash recovery, and it is configured to log everything including user data. Now, a process wants to update the first 1KB of a file. What pieces of information should `fs-disk5600` write to the redo log (the journal)? And what dependencies do these pieces have? In particular, which piece(s) must be completed before which piece(s) start logging?
Write down the information `fs-disk5600` needs to log and describe their dependencies.

`fs-disk5600` writes to redo log (journal) with:

(1) a begin transaction message with txn id

(2) file inode block (with updated size, time, and other metadata changes)

(3) the first data block (with 1KB update)

(4) a commit transaction message with txn id

The dependency is: (4) must start after all (1), (2), and (3) finish.

Note:

- if you assume starting from root directory and adding those, that's fine.
- copy-pasting redo logging steps from your cheat sheet doesn't count.
- rambling irrelevant or incorrect content will lose points. (see instructions on the front page.)

VII Lab3 and Lab4 (20 points)

10. [10 points] In this problem, you will re-implement Lab3's concurrent queue. The queue should allow multiple threads to `enqueue` and `dequeue` concurrently. Again, you must follow the two requirements of concurrent programming below. (copied from Lab3 and simplified)

Requirement I: concurrency commandments (namely, six rules). You are required to follow these standards.

Requirement II: monitor. You must implement the concurrent queue as a monitor. Note that C programming language doesn't have native monitor supports or object oriented programming. Hence, you are going to implement "manually constructed" monitors in C style, wherein: (1) all function calls are protected by a mutex (that is, the programmer inserts those `acquire()`/`release()` on entry and exit from every procedure that works with the shared state). (2) synchronization happens with condition variables whose associated mutex is the mutex that protects the functions.

As a reminder, here are the relevant data structures in `queue.h`.

```
typedef struct task task_t;

struct task {
    task_t *next;
    int fd;
```

```
};

typedef struct queue {
    task_t *head;    /* remove from head */
    task_t *tail;    /* append to tail */
    int count;
} queue_t;
```

Finish the code below.

Note: there are three TODO pieces you need to fill in.

```
/* TODO: define your synchronization variables here */




// Append a task to the end of the queue.
void enqueue(queue_t *q, task_t *t) {
    /* TODO: your code here */

}


// Return the task at the head of the queue.
// If the queue is empty, the thread should wait.
task_t* dequeue(queue_t *q) {
    /* TODO: your code here */
```

```
}
```

One possible implementation:

```
pthread_mutex_t qm = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t qcv = PTHREAD_COND_INITIALIZER;


void enqueue(queue_t *q, task_t *t) {
    pthread_mutex_lock(&qm);
    t->next = NULL;
    if (q->head == NULL)
        q->head = q->tail = t;
    else {
        q->tail->next = t;
        q->tail = t;
    }
    q->count++;
    pthread_cond_signal(&qcv);
    pthread_mutex_unlock(&qm);
}

task_t* dequeue(queue_t *q) {
    pthread_mutex_lock(&qm);
    while (q->head == NULL)
        pthread_cond_wait(&qcv, &qm);
    task_t *t = q->head;
    q->head = t->next;
    q->count--;
    pthread_mutex_unlock(&qm);

    return t;
}
```

11. [10 points] Below are some questions about the design of the CS5600 File System in Lab4. These questions are intended to be straightforward, if you have basic familiarity with the lab. You can answer them even if you did not successfully submit the lab

(2 points) What is the block size in Lab4?
Write down your answer in KB.

(2 points) **How many blocks does `fs5600` use for the bitmap?**

(2 points) Which block does `fs5600` use as the root directory (namely, the "/")?
Write down its block id.

(2 points) What is the size of an inode in `fs5600`?
Write down your answer in KB.

Name: Solutions **NUID:**

(2 points) **How many data blocks does a `fs5600` directory use?**

4KB,

1,

2,

4KB,

1

End of Final