Northeastern University
CS5600: Computer Systems: Spring 2022

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/18)	II (xx/6)	III (xx/10)	IV (xx/12)	V (xx/21)	VI (xx/20)	VII (xx/13)	Total (xx/100)

Name: Solutions

NUID:

I Computer system basics (18 points)

1. [12 points] Answer the questions below.

a) One byte is _8__ bits.

b) For an x86-32 machine (32-bit virtual addresses), what is the maximum memory a process can have?

4GB (given 32bit VA, the maximum addressable memory is 2^{32} Bytes, namely 4GB.)

c) Assume a virtual memory system whose page size is 16KB. How many bits does the offset in virtual addresses need? (to access all bytes in a page)

14bits (because 16KB=2^{14}Bytes, you need 14bits to access all these bytes.)

d) If a thread acquires the same mutex twice, there is a deadlock.
Circle your answer: [True / False]

e) When OS context switches from one process to another, all TLBs have to be flushed.
Circle your answer: [True / False]

f) If a device uses DMA, it must also use interrupts to notify CPU when a job finished.
Circle your answer: [True / False]

[updated 05/05: This question is ambiguous because it wasn't specify if the thread releases the lock. Either True or False will get points]

~~True: the second time the thread acquiring the mutex, it will wait indefinitely because it waits itself to release the lock.~~

True: different processes have different VA-to-PA mapping (TLBs contain these mappings). OS needs to get rid of one process's mappings in TLBs when context switching. [note: if you assume PID tags on TLBs and said so, False is also a correct answer.]

False: No. Polling + DMA is also fine.

2. [6 points] There are three ways to trap to kernel (switching from user-level processes to kernel). We studied them in various topics in class.
What are they? and give an example of each category.

(1) system call: fork() [or any other system calls]

(2) interrupt: timer interrupt [or any other hardware interrupts]

(3) exception: page fault [or any other exceptions]

[updated 05/05: if you didn't give examples, you lose points.]

II Processes (6 points)

3. [6 points] Below is a file called "code.c". Read the code and then answer questions.

```
 1  #include "stdio.h"            // the first char of this file is '#'
 2  #include "unistd.h"
 3  #include "fcntl.h"
 4
 5  int main()
 6  {
 7      char c1, c2;
 8      int fd = open("code.c", O_RDONLY, 0);         // open this file
 9
10      read(fd, &c1, 1);                // read one byte from fd to c1
11      if (fork() > 0) {
12          read(fd, &c2, 1);            // read one byte from fd to c2
13          printf("c1 = %c, c2 = %c\n", c1, c2);
14      } else {
15          sleep(1);
16          read(fd, &c2, 1);            // read one byte from fd to c2
17          printf("c1 = %c, c2 = %c\n", c1, c2);
18      }
19      return 0;
20  }
```

Note: the file opened (`line 8`) is the source code file itself ("code.c").

Write down the output of the parent process.

```
c1 = #, c2 = i
```

Write down the output of the child process.

```
c1 = #, c2 = n
```

[After fork,

- the `c1` with '#' is copied (in fact, we all know this is COW) to the child process.
- fd (pointing to the file "code.c") is shared between processes. (note: this is how pipe works in Lab2)]

III Shell, File permission, and Softlink (10 points)

4. [10 points] We run the following shell commands on an *empty* Unix machine.

```
$ mkdir /tmp/a; mkdir /tmp/b
$ ln -s /tmp/a /tmp/b/a; ln -s /tmp/b /tmp/a/b
```

Note:

- "`ln -s target linkname`" creates a softlink named "`linkname`" pointing to a file or a directory "`target`".
- "`2>/dev/null`" redirects error outputs to `/dev/null`, so they will not appear on screen.
- "`2>&1`" redirects error outputs to standard outputs.

(2 points) Then, we run `$ ls /tmp/a/b/`. What's the output of this command?
Write down the output of `ls`.

`a`

[Notice that there is a cycle of softlinks: you could `ls /tmp/a/b/a/b/a/b....`]

(4 points) Next, we run the following commands in order:
(hint: a command "`$ chmod 0777 xyz`" changes the file `xyz`'s permission to "`rwxrwxrwx`".)

```
$ echo "CS5600" > /tmp/a/f             # create file /tmp/a/f
$ chmod 0555 /tmp/a/f
$ cat /tmp/a/f 2>/dev/null >/dev/null && echo "cat 1" || echo "cat 2"
$ echo "final" 2>/dev/null >/tmp/a/f  && echo "echo 1" || echo "echo 2"
```

Write down the outputs of `cat` and `echo`.

```
cat 1
echo 2
```

(4 points) Finally, we run commands:

```
$ chmod 0111 /tmp/a
$ cd /tmp/a/ >/dev/null 2>&1 && echo "cd 1" || echo "cd 2"
$ ls /tmp/a/ >/dev/null 2>&1 && echo "ls 1" || echo "ls 2"
```

Write down the outputs of `cd` and `ls`.

```
cd 1
ls 2
```

[Note: these are valid shell commands. You can try them in your computer.]

Name: Solutions **NUID:**

IV Concurrency (12 points)

5. [12 points] You're going to finish implementing a concurrent database called CS5600-DB. CS5600-DB has the following properties:

- CS5600-DB supports readers and writers.
- CS5600-DB allows concurrent readers: if there is no writer, readers should not be blocked.
- CS5600-DB has exclusive writer: when a writer is working on the database, all readers and other writers have to wait.
- CS5600-DB *prioritizes writers*: whenever there are waiting writers, the writers must go first, before all currently waiting readers (if any) and readers that come afterward (writers themselves do not have internal priorities).

Below are the code for CS5600-DB's reads and writes. You need to implement the function `beginRead()`, `endRead()`, `beginWrite()`, and `endWrite()` to satisfy the above properties. And, you must follow Mike Dahlin's six concurrency rules.

```
// the variables "db", "key", "val" are here for completeness,
// you will not use them.
char* dbRead(database *db, char *key) {
    beginRead();
    // read database
    ...
    endRead();
}

int dbWrite(database *db, char *key, char *val) {
    beginWrite();
    // write database
    ...
    endWrite();
}
```

Write down your code below.

```
// [updated 05/05: we didn't specify that ‘‘when readers are working on the DB,
// writers should wait’’, which I (Cheng) thought is a common sense.
// Due to this ambiguity, we give points to those who disobey this. But, you
// need other properties to get points. Violating 6 rules also lose many points.]

// There are many possible implementations.
// In fact, our handout Week6b panel 2 has one solution.
// (https://naizhengtan.github.io/22spring/notes/handout_w06b.pdf)
// Below is a simpler one.
```

Name: Solutions **NUID:**

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

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cv = PTHREAD_COND_INITIALIZER;
int waiting_writers = 0, running_writers = 0, running_readers = 0;



void beginRead() {
    /* TODO: your code here */

    lock(&m);
    while (waiting_writers > 0 || running_writers > 0) {
        cond_wait(&cv, &m);
    }
    running_readers++;
    unlock(&m)
}

void endRead() {
    /* TODO: your code here */

    lock(&m);
    running_readers--;
    broadcast(&cv);  // signal is incorrect here. why?
                     // what if signal wakes up a reader but there are waiting writers?
    unlock(&m);
}

void beginWrite() {
    /* TODO: your code here */

    lock(&m);
    while (running_writers > 0 || running_readers > 0) {
        waiting_writer++;
        cond_wait(&cv, &m);
        waiting_writer--;
    }
    running_writers = 1; // or ++
    unlock(&m)
}

void endWrite() {
    /* TODO: your code here */

    lock(&m);
    running_writers = 0; // or --
    broadcast(&cv);  // signal is incorrect here. why?
                     // what if signal wakes up a reader but there are waiting writers?
    unlock(&m);
}
```

V Virtual memory (21 points)

6. [4 points] Suppose we have a 32-bit virtual memory machine, called `x97-32`.
Are the following statements True or False? **Circle your answers.**

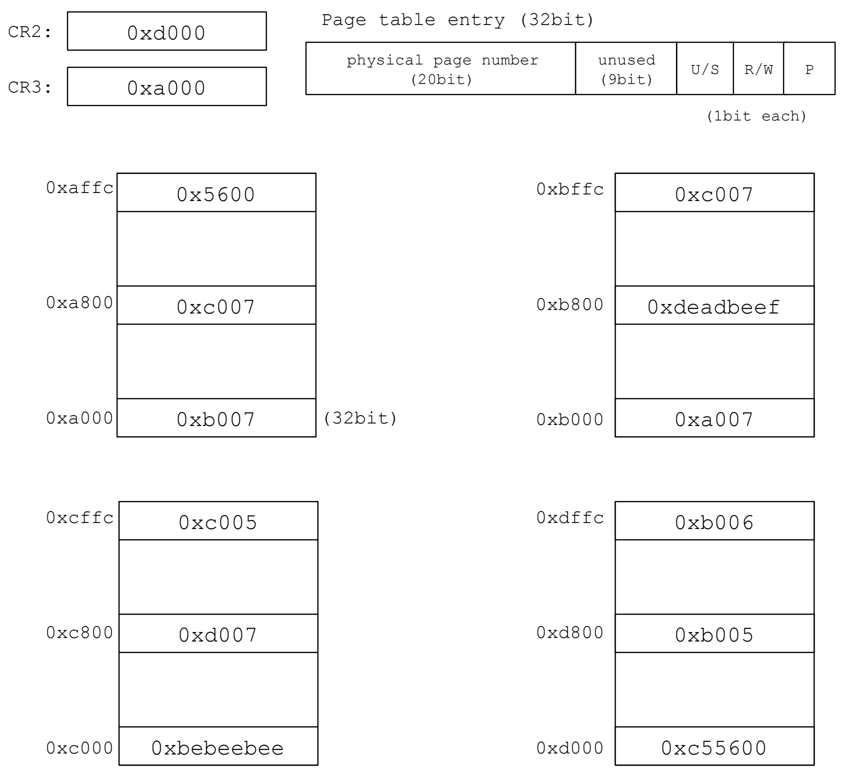
If `x97-32` uses 4KB pages, virtual address `0x1234500` cannot map to physical address `0x4567800`.
[True / False]

If `x97-32` uses 1KB pages, virtual address `0x1234500` can map to physical address `0x4567800`.
[True / False]

True: 4KB implies that the offset is 12bits. The last 12bits of `0x1234500` (`0x500`) is different from `0x800` in `0x4567800`. So, this VA cannot map to the PA.

False: 1KB implies that the offset is 10bits. The last 10bits of `0x1234500` is `0x100`, where as the last 10bits of `4567800` is `0x000`. Still, the VA cannot map to the PA.

7. [17 points] Below is a memory snapshot of a x86-32 machine.



- The snapshot shows four physical pages. Physical addresses on the left of the page.
- Each page is 4KB. Memory is drawn in 32-bit units.
- Values of two registers, `CR2` and `CR3`, can be found on the top-left.
- The definition of "page table entry" (PTE) can be found on the top-right.
- x86-32 uses two-level page tables.
- (hint: "VA" should be "broken down" to "10:10:12"; as a hint, we intentionally use jargon.)

(3 points) What is the maximum physical memory that a x86-32 machine can use?
Write down your answer and explain why in 1 sentence.

4GB

2^{20} * 4KB = 4GB. [Note: physical page number is 20bits in PTE (see figure).]

(2 points) What are the meaning of "`R/W`" bit and "`P`" bit in `page table entry`?
Explain in no more than 2 sentences for each bit.

`R/W` bit holds read/write permissions. If `R/W=0`, the corresponding page(s) is read-only; otherwise, pages are read-write.

`P` bit is the present bit that indicates whether a page table entry can be used in address translation. `P=1` indicates that the entry can be used.

Name: Solutions **NUID:**

(3 points) Given the above memory snapshot, what's the content of virtual address `0xffc`?
Write down the content and the L1/L2 page table pages while walking the page table.

`0xffc`'s content: `0x5600`

L1 page physical address: `0xa000`

L2 page physical address: `0xb000`

[Here is the page walk:

- `0xffc` can be split into `0x0` (L1 index), `0x0` (L2 index), and `0xffc` (offset).
- `CR3` points to L1 page table page `0xa000`
- based on the L1 index `0x0`, we get L2 page table page `0xb000`
- based on the L2 index `0x0`, we get data page `0xa000` (yes, this is the same page of L1 page table page, which is valid.)
- the offset within the page is `0xffc`, meaning PA is `0xaffc`.
- the content of `0xaffc` is `0x5600`]

(3 points) What's the content of virtual address `0x803ff000`?
Write down the content and the L1/L2 page table pages while walking the page table.

`0x803ff000`'s content: `0xbebeebee`

L1 page physical address: `0xa000`

L2 page physical address: `0xc000`

[page walk:

- `0x803ff000` can be split into `0x200` (L1 index), `0x3ff` (L2 index), and `0x000` (offset)
- `CR3` points to L1 page table page: `0xa000`
- by L1 index `0x200`, we got L2 page table page: `0xc000`
- by L2 index `0x3ff`, we got data page: `0xc000`
- the offset within the page is `0x000`, meaning PA `0xc000`.
- Content is `0xbebeebee`.]

(3 points) Here is a piece of C code:

```
uint32_t *ptr = 0x803ff000;
*ptr = *ptr + 1;
```

When this code runs with the memory snapshot, will it successfully finish?
If yes, write down the content of `*ptr` in hex number. If no, explain why in 1 sentence.

No, there will be a page fault because PTE `0xcffc` says read-only pages.

Name: **Solutions** **NUID:**

(3 points) According to the memory snapshot, write down a virtual address whose content is "`0xc55600`". If there is none, write "none".

`0x80200000`

[Note you need to "walk backward" from the content `0xc55600`.

- There is only one PA `0xd000` having the content, meaning `0xd000` is the data page. Also, offset is `0x000`.

- Walking backward for L2 page table page, there is only one PTE (PA `0xc800`) starting with `0xd000`, then L2 is the page `0xc000`. Also, L2 index is `0x200` (why? because PTE is 4B, and the PTE's offset is `0x800` in the L2 page, `0x200=0x800/4`.)

- we know that `CR3` points to `0xa000`, hence we needs to pick the PTE with PA `0xa800` that points to the L2 page `0xc000`. Now, we know the L1 index is `0x200`.

- finally, concatenate L1 index `0x200` (10bit), L2 index `0x200` (10bit), and offset `0x000` (12bit), resulting in `0x80200000`.]

VI I/O device and Crash recovery (20 points)

8. [8 points] In class, we studied the internals of disks (e.g., head, platter, track, sector) and SSDs (e.g., bank, block, page, erase, program, wear-out). Explain some of their performance characteristics below from their organization level.

(2 points) For disks, sequential reads are much faster than random reads. Why?
Explain in no more than 2 sentences.

Sequential reads only need the head to move once whereas random reads need to move head back and forth between tracks.

(3 points) For disks, random reads are faster than random writes. Why?
Explain in no more than 2 sentences.

The head settle time for reads is shorter than writes because if read strays, the error will be caught, and the disk can retry. However, if the write strays, data will get clobbered.

(3 points) Consider two 1TB SSDs of the same model. One is half full, with 500GB available; the other is 95% full, with 50GB available. It turns out that the writing speed to the half-full SSD is much higher than to the 95%-full SSD. Why?
Explain in no more than 3 sentences.

To avoid wear-out, SSDs need to spread out writes to available blocks. When most blocks are used, SSDs need to garbage collect the stale/invalid data. If an SSD is almost full, there will be frequent garbage collections, hence influencing the write performance.

9. [12 points] We design a variant of `fs5600` (Lab4) to be crash consistent, called `fs5600-cc`. `fs5600-cc` has the same design and data structures as `fs5600`, with an extra feature: crash recovery.

Assume `/file1` is an empty file with no data block. When writing to `/file1`, at least three disk blocks need to be updated. `fs5600-cc` updates them in the following order:

A. `block#21`: the first data block of `/file1` (updating file contents)

B. `block#10`: `/file1`'s inode (updating metadata)

C. `block#1`: block bitmap (allocating `block#21` and set its bit to be used)

(3 points) If there is no crash recovery, when crash happens in-between step **B** and **C** (**B** finished; **C** didn't), what can go wrong?
Write down one possible consequence in no more than 2 sentences.

After reboot, bitmap will think `block#21` is free and may allocate it to other files—a conflict with `/file1`.

Name: Solutions **NUID:**

One way to implement crash recovery is `fsck` (the ad-hoc crash recovery we studied in class).

(3 points) If you're asked to implement `fsck` for `fs5600-cc`, how you can fix the inconsistency caused by crashing in-between **B** and **C**?
(Note that `fsck` will scan the entire disk after reboot.)
Write down how your `fsck` will fix the inconsistency in no more than 2 sentences.

Another way to implement crash recovery is using redo logging (journaling). There are five steps to finish a file system operation as a transaction using redo logging. What are the step 2, 3, and 4 of redo logging?

(3 points) **Write down step 2-4 briefly in 1 sentence each.**

1. planning
2. write begin transaction to journal (or anything equivalent)
3. write redo records to journal (or anything equivalent)
4. write commit transaction to journal (or anything equivalent)
5. checkpointing

(3 points) `fs5600-cc` is configured to be "metadata consistent": only metadata is guaranteed to be consistent after crash. Assume that the write operation to `/file1` finished successfully, with a transaction id `5600`. What does the journal look like at the time when finishing the write?
Fill in the journal slots below.

```
Journal head                                tail
    ----+---------+-------+-------+---------+------
   prev | begin   | block | block | commit  |
   txn  | txn     |  #10  |  #1   | txn     |
   ...  | id=5600 |       |       | id=5600 |
    ----+---------+-------+-------+---------+------
```

[Note:

- the first slot must be a begin txn event and must contain the id 5600.

[updated 05/05: note that "begin txn" may not finish before other events in the same transaction, but it must be placed before others in journal.]

- the last slot must be a commit txn event and must contain the id 5600.

- in-between begin txn and commit txn, you must indicate that the journal contains `/file1`'s inode and bitmap. (you don't have to write the block ids.)

]

VII Labs (13 points)

10. [3 points] Testing concurrent programs is notoriously hard. How did you design your test cases in your Lab3 (this is Exercise 5)? If you haven't, design one now.
Write down one of your concurrency test cases in no more than 3 sentences.

Anything reasonable works.

11. [10 points] Below are the inode and directory entry definitions from `fs5600.h` in Lab4.

```
struct fs_inode {
    uint16_t uid;
    uint16_t gid;
    uint32_t mode;
    uint32_t ctime;     // time of last status change; see more in "man 2 stat"
    uint32_t mtime;     // last modification time
    int32_t  size;
    uint32_t ptrs[NUM_PTRS_INODE];
};

/* Entry in a directory */
struct fs_dirent {
    uint32_t valid : 1;
    uint32_t inode : 31;
    char name[28];              /* with trailing NULL */
};
```

Note: questions below are not self-contained; you will need some knowledge of Lab4 (but not much).

(5 points) How many data block pointers does the `fs_inode` contain (`NUM_PTRS_INODE` in `fs_inode`)?
Write down your answer and explain your calculation in 1 sentence.

1019 or $2^{10} - 5$ or any equivalent numbers

1019 = (4KB - 20B) / 4B

(you need to know the size of a fs5600's inode is a data block, 4KB.)

(5 points) In `fs5600`'s design, what is the maximum number of files in a directory?
Write down your answer and explain your calculation in 1 sentence.

128

#files/dirs = a data block / entry size = 4KB/32B = 128
(you need to have the knowledge that (i) the block size is 4KB and (ii) fs5600's directory uses only one data block.)

Name: Solutions **NUID:**

End of Final