New York University
CSCI-UA.0202: Operating Systems (Undergrad): Spring 2022
Midterm Exam

- This exam is **75 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 78 minutes after the exam begins and will not accept exams outside the room.
- There are **13** 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 with 10 point or larger Times New Roman font, 1 inch or larger margins, and a maximum of 55 lines per side.
- Do not waste time on arithmetic. Write answers in powers of 2 if necessary.
- 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 NetId on this cover sheet and on the bottom of every page of the exam.**

Do not write in the boxes below.

I (xx/14)	II (xx/8)	III (xx/22)	IV (xx/6)	V (xx/16)	VI (xx/20)	VII (xx/14)	Total (xx/100)

Name: Solutions

NetId:

I Getting started (14 points)

1. [6 points] We described three ways that the machine goes from "user mode" (running a process) to "kernel mode" (running the OS). What are two of them?

Exception, trap (or system call), interrupt.

2. [8 points] As a reminder, file descriptor (fd) 1 is the "standard output" (stdout) for a Unix process, and the prototypes for the `write()` and `open()` system calls are:

```
int write(int fd, const void* data_to_write, size_t number of bytes);
int open(const char* pathname, int flags, mode_t mode);
```

Now consider the program below. Be careful; think about every line.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main(int argc, char** argv) {
    int fd, rc;
    rc = fork();
    if (fork() == 0) {

        // The next line opens a file for reading and writing,
        // creating it if it didn't previously exist. After this
        // line, 'fd' refers to the opened file.
        fd = open("/tmp/hello.txt", O_CREAT | O_RDWR, 0666);
        if (fd < 0)
            perror("open failed");

        // After the next line, file descriptor 1 no longer refers to
        // what it was referring to, and instead refers to the same file
        // that fd was referring to.
        dup2(fd, 1);
    }

    write(1, "cs202\n", 6);

    return 0;
}
```

Assuming that there are no errors, what does the program above do? Write no more than two sentences.

It twice does the following: outputs `cs202` to the terminal (or whatever stdout was when this program is loaded) and also writes `cs202` to the beginning of the file `/tmp/hello.txt`. We said "twice" because

the program forks in the first line. That creates two versions of the program. And then each of those versions forks, but then does something different in the child versus the original process.

A common error was to assume that `dup2` adjusted not only the child's file descriptor but also the parent's (and thus there is different output depending on which process finishes first). But child processes can't affect parent processes; if they could, then the examples we saw in class about file descriptor redirection wouldn't make sense. Another error was to overlook the fact that both a child process and the parent process would execute the `write`.

II Pipelines (8 points)

3. [8 points] This question asks you to compose a shell pipeline (a command at the shell) to perform a particular task. The task is: *out of all filenames and directory names in the current directory tree and that contain* `cs202` *in the name, print out the alphabetically-first filename or directory. By "directory tree," we mean the current directory, the directory's sub-directories, their sub-directories, etc.).*

Some helpful Unix utilities are described below. You have seen most or all of these in labs and homeworks.

- `ls`. When invoked with `-1` (that's the numeral one), `ls` produces its output left-aligned, single-column.
- `head`. When invoked as `head -n [NUM]`, this utility reads from standard input and outputs the first `NUM` lines.
- `sort`. This utility reads from standard input, expecting one data item per line, and by default alphabetically sorts the input, writing the sorted output to standard output. `sort` has many arguments that control its behavior, but you don't need them for this problem.
- `grep`. The syntax is `grep pattern [filenames]`. If `filenames` are supplied, `grep` searches the files themselves for `pattern`, outputting matching lines. If no filenames are supplied, `grep` searches its standard input instead.

Write a suitable shell pipeline at the command prompt below.

```
$
```

```
$ ls -1aR . | grep cs202 | sort | head -n 1
```

III Lab: `ls` (22 points)

The questions in this section assume that we are running on the cs202 devbox.

4. [7 points] The current directory holds the following files (note that we are using commas (`,`) to separate file names): `abc`, `def`, `.ghi`, `..jkl`. Assume that `ls` is the default utility on the system. A correct implementation of lab 2 would mimic the system `ls` utility, so you can also answer with respect to what your lab code is supposed to do. Consider the command below:

```
$ ls
```

What is the output from invoking `ls` at the command prompt as above?

`abc`
`def`, because files starting with `.` are not printed by default.

5. [7 points] In the current directory, you type `ls -l`. Two of the lines of output are as follows:

```
$ ls -l
d--------- 2 vagrant vagrant 4096 Mar  8 14:59 my_dir
-rw-r--r-- 1 vagrant vagrant   27 Mar  8 14:59 my_file
```

Notice that `my_dir` is a directory. Now, consider the following command, and again assume that `ls` is the default `ls` on the system:

```
$ ls -R my_dir
```

What happens when we issue the command above?

`ls` outputs an error message since there aren't permissions to descend into `my_dir`.

6. [8 points] Consider the option-handling code from lab 2, supplied below and streamlined slightly. Your task is to modify this code so that invoking `ls -Z` outputs `cs202 TAs are great`:

```
$ ls -Z
cs202 TAs are great
$
```

In the space below, fully describe the needed modifications in syntactically valid C. You can use line numbers for reference. Alternatively, you can modify the code directly with editing marks.

Solution omitted, as it's close to tasks in lab2.

```
1  int main(int argc, char* argv[]) {
2      int opt;
3      bool list_long = false, list_all = false;
4
5      // The ‘struct option‘ helps us parse arguments of the form ‘--FOO‘.
6      // Refer to ‘man 3 getopt_long‘ for more information.
7      struct option opts[] = {
8          {.name = "help", .has_arg = 0, .flag = NULL, .val = '\a'}};
9
10     // This loop is used for argument parsing. Refer to ‘man 3 getopt_long‘ to
11     // better understand what is going on here.
12     while ((opt = getopt_long(argc, argv, "1a", opts, NULL)) != -1) {
13         switch (opt) {
14             case '\a':
15                 // Handle the case that the user passed in ‘--help‘. (In the
16                 // long argument array above, we used '\a' to indicate this
17                 // case.)
18                 help();
19                 break;
20             case '1':
21                 // Safe to ignore since this is default behavior for our version
22                 // of ls.
23                 break;
24             case 'a':
25                 list_all = true;
26                 break;
27             default:
28                 printf("Unimplemented␣flag␣%d\n", opt);
29                 break;
30         }
31     }
```

IV Interleavings (6 points)

7. [6 points] Consider the code below. The two threads may run on different processors. Do not assume sequential consistency.

```
1     int a = 0, b = 0;
2
3     int main () {
4         tid id1 = thread_create (p1, NULL);
5         p2 (); // p1, p2 are running concurrently
6         thread_join(id1);
7         exit(0);
8     }
9
10    void p1 (void *ignored) {
11        a = 1;
12        b = 1;
13    }
14
15    void p2 (void *ignored) {
16      if (b == 1)
17        use (a);
18    }
```

Can `use()` be called with value 0? Why or why not? Write no more than two sentences.

Yes, we saw a similar example in class. Without sequential consistency (and in particular in memory models where the order of writes as seen by the issuing processor isn't the same as seen by other processors), it might be that `p2` sees the write from line 12 as happening first. This could result in an interleaving with the following lines: 12, 16, 17, 11.

V Mutexes and condition variables (16 points)

8. [8 points] Modify the code below to ensure that "I am foo!!!" prints before "I am boo!!!". Your solution should ensure that both phrases are actually printed. Use mutexes and condition variables.

```
int i = 0;
/* ADD SOME THINGS HERE */



/* ADD SOME CODE TO THIS FUNCTION */
void foo(void *ignored) {

    printf("I am foo!!!\n");







}

/* ADD SOME CODE TO THIS FUNCTION */
void boo(void *ignored) {







    printf("I am boo!!!\n");

}

int main(int argc, char** argv) {
    /* ADD SOME CODE HERE */



    tid tid_foo = thread_create(foo, NULL);
    tid tid_boo = thread_create(boo, NULL);

    // wait for threads to finish before exiting
    thread_join(tid_foo);
    thread_join(tid_boo);
    exit(0);
}
```

This was a homework problem.

```
int i = 0;
/* ADD SOME THINGS HERE */
scond_t cond;
smutex_t mutex;


/* ADD SOME CODE TO THIS FUNCTION */
void foo(void *ignored) {

  smutex_lock(&mutex);
  printf("I am foo!!!\n");
  i = 1;
  scond_signal(&cond, &mutex);
  smutex_unlock(&mutex);

}

/* ADD SOME CODE TO THIS FUNCTION */
void boo(void *ignored) {

  smutex_lock(&mutex);
  while (!i) {
    scond_wait(&cond, &mutex);
  }
  printf("I am boo!!!!\n");
  smutex_unlock(&mutex);

}

int main(int argc, char** argv) {
    /* ADD SOME CODE HERE */

    smutex_init(&mutex);

    scond_init(&cond);

    tid tid_foo = thread_create(foo, NULL);
    tid tid_boo = thread_create(boo, NULL);

    // wait for threads to finish before exiting
    thread_join(tid_foo);
    thread_join(tid_boo);

    smutex_destroy(&mutex);
    scond_destroy(&cond);

    exit(0);
}
```

9. [8 points] Let `cv` be a condition variable, and let `mutex` be a mutex. Assume two threads. The pattern below is incorrect: the `if` should be a `while`. This question is asking why—beyond the fact that the concurrency commandments assert that the pattern is incorrect.

```
acquire(&mutex);
if (not_ready_to_proceed()) {
    wait(&mutex, &cv);
}
release(&mutex);
```

Why, precisely, is this pattern incorrect? (Write no more than two sentences.)

After a thread signals to indicate "ready", circumstances could have changed, perhaps because of actions later taken by the signaling thread itself, to make it not safe for the waiting thread to proceed. Indeed, the implementation of `wait` in common threading packages (including the one that we used in lab 3) can wake the thread up at any time; there is simply no guarantee that the predicate that is being tested in `not_ready_to_proceed()` is actually true.

VI Programming with monitors (20 points)

10. [20 points] In this problem you will implement a mechanism for serving buyers according to the order in which they arrive at a bakery. You will use tickets for this: a buyer gets a ticket shortly after it arrives, and these tickets are assigned in ascending order. Buyers are serviced by workers in order of their numbers. The analogy is with the "take a number" system used in bakeries, delis, and groceries. We model the problem with each buyer and each worker as a thread, synchronized with a shared monitor called a `Bakery`.

```
Bakery bakery;   // global

void buyer() {

    int num;

    /* May have to wait in here */
    num = bakery.TakeANumber();

    order_goods(num);
}

void worker() {

    int num;

    while (1) {

        num = bakery.Service();

        find_buyer_and_help_them(num);

    }
}
```

Your job is to implement `Bakery`. A few notes and hints:

- There are one or more worker threads, which each call the `Bakery::Service()` method. This method simply causes the next-thread-to-be-serviced to stop waiting, and proceed.
- You don't have to worry about overflow; you can assume that the Bakery never processes more than `INT_MAX` orders over its lifetime.
- You don't have to worry about "joining" the currently-served buyer with the worker who is helping them; assume that work is done by `order_goods()` and `find_buyer_and_help_them()`.
- You must follow the class's concurrency commandments.

Where indicated, fill in the variables and methods for the `Bakery` object.

```
class Bakery {

    public:
        Bakery();
        ~Bakery();
        int TakeANumber();
        int Service();

    private:

        // FILL THIS IN

};



Bakery::Bakery()
{
    // FILL THIS IN
    mutex









}
```

```
int
Bakery:TakeANumber()
{
    // FILL THIS IN









}


int
Bakery:Service()
{

    // FILL THIS IN









}
```

```
class Bakery {

    public:
        Bakery();
        ~Bakery();
        int TakeANumber();
        int Service();

    private:

        // FILL THIS IN

        // some students tried to solve this with a single counter,
        // but that won't work because we separately need to allow
        // the customer to take a new number while tracking what
        // is the next number to be served.
        int customer_num;
        int service_num;
        mutex m;
        cond number_increased;
        cond work_to_do;

};



Bakery::Bakery()
{
    // FILL THIS IN
    mutex_init(&m);
    cond_init(&number_increased);
    cond_init(&work_to_do);
    customer_num = 0;
    service_num = 0;
}

int
Bakery:TakeANumber()
{
    // FILL THIS IN
    int mynum;

    mutex_acquire(&m);

    // Note that we need a local variable here (mynum) because
    // multiple customers can arrive, and each needs to track its
    // own number.
    mynum = customer_num;
    customer_num++;
    cond_signal(&m, &work_to_do);
```

```
    // below, some students had:
    //   while (mynum != servicing_num)
    // but this is not correct, because the present thread
    // might get "unlucky." As an example, imagine the first thread
    // through. It gets mynum=0. Now imagine that 10 customers come
    // in, then 10 workers, but the present thread happens not to be
    // scheduled yet, even though it's been made runnable 10 times by
    // the 10 broadcasts. When this thread eventually runs, we have
    // mynum=0 and servicing_num=10, so this thread again keeps
    // waiting (because 0 != 10), when the correct behavior would be to continue
    // past while and get service from the worker who was looking for
    // customer 0.
    while (mynum > service_num) {
        cond_wait(&m, &number_increased);
    }

    mutex_release(&m);
    // some students returned customer_num or customer_num + 1.
    // that would mean reading shared memory outside the mutex.
    return mynum;
}



int
Bakery:Service()
{
    // FILL THIS IN
    int mynum;

    mutex_acquire(&m);

    // some students had the condition as service_num >= customer_num.
    // that isn't wrong, but it's not possible for service_num to be
    // greater than customer_num.
    while (service_num == customer_num)
        cond_wait(&m, &work_to_do);

    mynum = service_num;
    service_num++;

    // has to be broadcast to ensure that the customer waiting on
    // our 'mynum' gets a wakeup.
    // some students didn't have any kind of notification to waiting
    // customers; that was contrary to the problem description, which
    // specified that a worker "causes the next-thread-to-be-serviced to stop
    // waiting, and proceed."
    cond_broadcast(&m, &number_increased);
    mutex_release(&m);
```

```
    // some students returned service_num or service_num + 1.
    // that would mean reading shared memory outside the mutex.
    return mynum;
}
```

VII Scheduling and feedback (14 points)

11. [6 points] What is one disadvantage of Shortest Time to Completion First (STCF)? Only several words are needed, and do not write more than one sentence.

A. It requires predicting the future.

B. Potential starvation

C. It does not optimize response time

12. [6 points] This question is about the Linux Completely Fair Scheduler (CFS), as described in the OSTEP text. Which of the following statements are true?

Circle ALL that apply:

A CFS is concerned with the proportional assignment of CPU cycles to processes.

B CFS assigns tickets and selects a winning ticket in each quantum.

C With CFS, all processes run for equal lengths of time.

D I/O-bound processes (meaning those that spend most of their time waiting for I/O to complete) do not get a fair share of the CPU.

E CFS avoids starvation.

A, D, and E.

B is about lottery scheduling.

Many students selected C. But C is not correct because in CFS there is a concept of weights that can allow one process to run for (say) twice as many CPU cycles as another.

Many students missed D. D is correct because the technique for handling I/O-bound processes is to "catch them up", but that means they don't get a proportional share of the CPU.

This question was graded as follows:

if the student circled A,D,E: +6
if the student circled A,E and not D: +4
if the student circled any other combination: going item-by-item, a correct answer (circling if it should be circled, not circling if it shouldn't be) is +1, an incorrect answer is -1, and the floor is 0.

13. [2 points] This is to gather feedback. Any answer, except a blank one, will get full credit.

Please state the topic or topics in this class that have been least clear to you.

Please state the topic or topics in this class that have been most clear to you.

Scratch space if needed

Scratch space if needed

Scratch space if needed

End of Midterm
Enjoy Spring Break!!