New York University
CSCI-UA.0202-001: Operating Systems (Undergrad): Fall 2025
Midterm Exam (Token: V0)

- **Write your name and NetId on this cover sheet (where indicated, at the bottom). Write your name, NetId, and token on the cover of your blue book.**
- **Put all of your answers in the blue book; we will grade only the blue book. At the end, turn in both the blue book and the exam print-out.** "Orphaned" blue books with no corresponding exam print-out will not be graded.
- This exam is **75 minutes**. Stop writing when "time" is called. Do not get up or pack up in the final five minutes. The instructor will leave the room 78 minutes after the exam begins and will not accept exams outside the room.
- There are **9** questions (in **7** sections) in this booklet. 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. It must be written in English, and may not have functions from the labs.
- If you find a question unclear or ambiguous, state 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.*

Do not write in the boxes below.

I (xx/15)	II (xx/10)	III (xx/14)	IV (xx/10)	V (xx/25)	VI (xx/14)	VII (xx/12)	Total (xx/100)

Name:

NetId:

I Short answer (15 points)

1. [5 points] What is the name of the operating system developed at Bell Labs starting in the late 1960s, which originally ran on the PDP-7 and PDP-11 computers?

2. [5 points] A given data type can take any value in hexadecimal from `0x00` to `0xff`, inclusive. How many possible values can this data type take? Give your answer in decimal (meaning ordinary base-10).

3. [5 points] Consider the x86-64 paging structures, in which the MMU uses a 36-bit virtual page number, 9 bits at a time, to walk four levels of page tables. Now imagine the same structure, but it is only three levels. Thus, in this alternative, the virtual page number is 27 bits, and the offset is now 12+9=21 bits, yielding a page size of 2MB rather than the usual 4KB.

One advantage of these larger pages is that the MMU has to walk only three (as opposed to four) levels of page tables on a TLB miss; another advantage is that there is less pressure on the TLB generally.

What is the *disadvantage* of having larger pages? Choose the BEST answer.

- **A** Greater internal fragmentation
- **B** Slower page table walks
- **C** Poorer cache locality
- **D** Weaker spatial reuse
- **E** Larger page tables
- **F** Larger number of bits to represent physical page number

II Process control (10 points)

4. [10 points] Consider the program below. Assume that `stdout` (file descriptor 1) begins by referring to the terminal (meaning the output that you will see interactively). Further assume that the program does not encounter error. Two of the system calls used by this program are:

- `dup2(int existingfd, int newfd)`: This resets `newfd` to describe the same file or device as `existingfd`.
- `wait(int* status)`: The caller waits for any forked child process to exit. The argument will be set to NULL and hence ignored in this problem.

```
int main()
{
    pid_t pid;
    int fd;

    if ((pid = fork()) == 0) {
        printf("WWW\n");

        if ( (fd = open("cs202", O_CREAT | O_TRUNC | O_WRONLY, 0666)) < 0) {
            fprintf(stderr, "Couldn't open file cs202\n");
            exit(-1);
        }

        dup2(fd, 1);
        printf("XXX\n");

    } else if (pid > 0) {
        wait(0);
        printf("YYY\n");
    } else {
        fprintf(stderr, "Error forking\n");
        exit(-1);
    }

    printf("ZZZ\n");

    exit(0);
}
```

What does this program print to the terminal?

III Lab 2 (14 points)

5. [14 points] Consider the simplified and non-standard version of `ls` below, which we will call `myls`. This program uses the `strcmp` standard library function, which returns 0 if its two string arguments are identical, and something non-zero otherwise. The syscalls `readdir()` and `opendir()` were used in lab 2. Assume that everything happening is in the class's Docker container.

```
int main(int argc, char** argv)
{
    DIR* d;
    struct dirent* dp;

    if (argc != 2) {
        fprintf(stderr, "%s requires two arguments\n", argv[0]);
        exit(-1);
    }

    if ((d = opendir(argv[1])) == NULL) {
        perror(argv[0]);
        exit(-1);
    }

    while ((dp = readdir(d)) != NULL) {
        if (strcmp(dp->d_name, "."))
            printf("%s\n", dp->d_name);
    }
    closedir(d);

    exit(0);
}
```

Now, assume the directory `testdir` contains exactly two files: `rho` and `sigma`.

What is the output of running `myls` as follows?

```
$ ./myls testdir
```

IV The concurrency commandments (10 points)

6. [10 points] Below is an interaction your instructor had with ChatGPT:

> Please tell me the four-commandment NYU CS202 crib sheet for concurrency commandments.

Here's the short, four-commandment crib sheet I teach/use for CS202:

1. **Bind invariants to locks**
   For every shared data invariant, choose **one** lock that protects it. Any code that might *break* or *assume* that invariant must hold that lock. (One invariant → one lock; hold the lock for the entire critical section.)
2. **Check under the lock; wait in a while**
   Evaluate the condition while holding the lock; if it's not true, wait with a **while-loop** (not if) and re-check after waking:

   c — Copy code

   ```c
   mutex_lock(m);
   while (!predicate())        // re-check after wake to handle spurious wakeups & races
       cond_wait(cv, m);
   /* proceed */
   mutex_unlock(m);
   ```

   Signal/broadcast when you make the predicate true, typically while holding the same lock.
3. **Impose a total lock order (and don't block while holding locks)**
   If you must hold multiple locks, acquire them in a **single, global order** to avoid deadlock. Keep critical sections short; avoid I/O, sleeps, or long work while holding a lock.
4. **No data races; publish safely**
   Every access to shared **mutable** state is synchronized (by the right lock or by atomics with acquire–release semantics). Do not rely on "single CPU" or "the compiler won't reorder." Establish a clear **happens-before** edge for any cross-thread communication.

That's it—four rules that cover almost every bug you'll meet in starter concurrency code.

Is ChatGPT's answer accurate? (Ignore the commentary before and after the numbered list, such as "That's it...".) Choose one of the options below and then explain in no more than one sentence.

- **A** Accurate as written
- **B** Accurate with minor edits
- **C** Inaccurate in a substantive way

Explanation:

V Stone hopping (25 points)

7. [25 points] There is a grid of `ROWS` rows and `COLS` columns, and each square on the grid can contain a stone or not. The grid has *horizontal crossers* (each of which is assigned to a row) and *vertical crossers* (each of which is assigned to a column). Each crosser is trying to traverse the grid in the respective orientation, and each is modeled as a thread. A third kind of entity, a stone putter (also modeled as a thread), drops stones on the grid randomly.

To traverse the grid, a horizontal crosser requires a stone in every column in the crosser's assigned row, and a vertical crosser requires a stone in every row in the crosser's assigned column. Two synchronization requirements are: *(1) A crosser should not traverse the grid unless its entire row or column is filled in with stones* and *(2) Only one crosser can be traversing the grid at once.* As a crosser moves, it removes the stones that it has traversed, which you can model as simply zeroing out all of the relevant state after crossing.

The shared memory is a two-dimensional array of Booleans, `array[ROWS][COLS]`. This array is encapsulated in a monitor, called `Grid`. Pseudocode is listed below for each crosser type and the stone putter.

```
Grid grid;

void horizontal_crosser()
{
    int i = get_random_integer(0, ROWS-1);

    grid.CrossHorizontal(i);

    // continue on with life on the other side
}

void vertical_crosser()
{
    int j = get_random_integer(0, COLS-1);

    grid.CrossVertical(j);

    // continue on with life on the other side
}

void stone_putter()
{
    while (1) {
        int i = get_random_integer(0, ROWS-1);
        int j = get_random_integer(0, COLS-1);

        grid.PutStone(i, j)
    }
}
```

Your job is to implement `Grid`. Things to keep in mind:

- You may want to draw a picture before beginning.
- Your solution should work for an arbitrary number (including an endlessly arriving set) of horizontal and vertical crossers.
- Follow the class's concurrency commandments and coding standards.
- Do not wake threads unnecessarily; in particular, having one condition variable for the entire problem will not get full credit.
- Similarly, do not block threads unnecessarily; even though only one crosser can traverse at once, you are not permitted to (for example) permanently prohibit traversal by particular kinds of crossers.
- You may need to define and use helper functions. We do not give prototypes for those.

Fill in variables and methods for the `Grid` object. There are five (5) places to fill in code. Pseudocode is acceptable.

```
class Grid {
    public:
        Grid();
        ~Grid(); // you do not have to implement this

        void CrossHorizontal(int row);
        void CrossVertical(int col);
        void PutStone(int row, int col);

    private:
        bool array[ROWS][COLS];

        // ADD MATERIAL HERE (1)

};

void
Grid::Grid()
{
    // set the entire array to false values;
    // the grid starts out with no stones
    memset(array, 0, sizeof(array));

    // FILL THIS IN (2)









}
```

```
// Can and should succeed (finish) if every column in row 'row' has a stone.
// If this condition is not met, this method should block until it is met.
// This method should, before exiting, remove all stones in row 'row'.
void
Grid::CrossHorizontal(int row)
{
    // FILL THIS IN (3)









}



// Can and should succeed (finish) if every row in column 'col' has a stone.
// If this condition is not met, this method should block until it is met.
// This method should, before exiting, remove all stones in column 'col'.
void
Grid::CrossVertical(int col)
{
    // FILL THIS IN (4)









}



// Place a single stone at grid position given by row 'row' and column
// 'col'. This method should set the "array" at the suitable place.
// What else should this method do?
void
Grid::PutStone(int row, int col)
{
    // FILL THIS IN (5)









}
```

VI Lab 3 (14 points)

8. [14 points] Consider a variant of the Lab 3 architecture, in which the `TaskQueue` is replaced with a `TaskStack`. The difference is that the first item placed in the stack is the last item removed. (To accommodate this, the code that would ordinarily call `RequestGenerator::enqueueTasks` followed by `RequestGenerator::enqueueStops` is switched to first place the stop requests in the stack followed by the tasks themselves, to ensure that the stops are removed at the end of the simulation. If this parenthetical confuses you, you can ignore it.)

On the next page is code for the `TaskStack`. When the code for the entire EStore (including this `TaskStack`) runs, the program hangs. Why? (Note that the initialization and cleanup code, otherwise known as the constructor and destructor, is omitted, but you should assume these are not buggy.)

State the problematic line or lines of code, and state the fix.

```
1  const int STACK_MAX = 1000;
2
3  class TaskStack {
4      private:
5         bool empty();
6
7         smutex_t mutex;
8         scond_t  not_empty;
9         Task stack[STACK_MAX];
10        int next_index = 0;   // always equal to the *next* index to add to
11
12     public:
13        TaskStack();  // implementation omitted, but not buggy
14        ~TaskStack(); // implementation omitted, but not buggy
15
16        void Add(Task task);
17        Task Remove();
18 };
19
20 bool TaskStack::
21 empty()
22 {
23     smutex_lock(&mutex);
24     bool is_empty = (next_index == 0);
25     smutex_unlock(&mutex);
26     return is_empty;
27 }
28
29 void TaskStack::
30 Add(Task task)
31 {
32     smutex_lock(&mutex);
33     stack[next_index++] = task;  // insert and then increment next_index
34     scond_signal(&not_empty, &mutex);
35     smutex_unlock(&mutex);
36 }
37
38 Task TaskStack::
39 Remove()
40 {
41     Task task;
42
43     smutex_lock(&mutex);
44     while (empty())
45         scond_wait(&not_empty, &mutex);
46     task = stack[--next_index]; // decrement next_index and return task at next_index
47     smutex_unlock(&mutex);
48
49     return task;
50 }
```

VII Stack frames and virtual memory (12 points)

9. [12 points] Consider the following machine:

- It uses an instruction set and register names like the x86-64. It also has a special `OUT` instruction that delivers its argument to the output terminal. For example, `OUT 'Y'` prints `Y` to the screen.
- It's byte-addressed (like the x86-64). This machine's memory addresses are 16 bits, and there is virtual address translation via paging. 12 bits are used for the offset, so the page size is 4 KB (4096 bytes), as in x86-64. That leaves four bits of the address for the virtual page number, which is used as an index directly into a single *linear* page table; there is no page table walking.
- A page table entry indicates that a virtual memory page is valid if the bottom bit of the entry is set (there is no R/W or U/S memory protection).
- There is no paging to the disk: if a process references a virtual page whose corresponding page table entry is invalid, the OS handles the corresponding page fault by ending the process.
- Every stack slot is 8 bytes.

The assembly program below (on the left) executes, during which the machine uses the page table below, on the right (arranged from highest index to lowest).

Notice that this assembly program is recursive. Assume that the program begins with the instruction pointer, `%rip`, set to the address of `f` while the stack pointer, `%rsp`, starts out equal to `0xfff8` (this is the address of the last 8-byte quantity in the process's virtual memory space). Further assume that the code lives in the virtual address space between addresses `0x1000` and `0x1fff`.

```
f:
    pushq %rbp
    movq %rsp, %rbp

    # subtract 4080 (in decimal) from
    # the stack pointer
    subq $4080, %rsp

    OUT 'X'

    call f

    movq %rbp, %rsp
    popq %rbp
    ret
```

PPN	valid	index
0x30	1	15
0x7C	1	
0x12	1	
0x9F	1	
0x45	0	
0xB2	0	
0x02	1	
0xF8	0	
0x5A	1	
0xAF	1	
0x0C	0	
0x8E	0	
0x6B	1	
0xE0	0	
0x3C	1	
0xDA	0	0

What does this program output?

A Nothing; it page faults before producing output
B Nothing; it hangs without page faulting
C Infinite sequence of `X`
D One `X`
E Two `X`
F Three `X`
G Four `X`
H Five `X`
I Nine `X`
J Sixteen `X`

End of Midterm