New York University
CSCI-UA.202: Operating Systems (Undergrad): Fall 2019

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/18)	II (xx/15)	III (xx/32)	IV (xx/21)	V (xx/14)	Total (xx/100)

Name: Solutions

NetId:

I Mechanics (18 points total)

1. [5 points] In the code below, `x` is an 8-bit, or single-byte, data type (an `unsigned char`, in C).

```
x = x + 1;
```

What does the above line of code do? Hint: remember the Therac-25.

The question is asking about overflow, and is intended as a reminder that even simple operations can have complexity, when working close to the machine's level of abstraction.

Solution: It adds one to x, unless x starts as $255 = 2^8 - 1$ in which case it sets x to 0. Or, call the value of x prior to this line x'. If $0 \leq x' < 255$, then x is set to $x' + 1$, whereas if $x' = 255$, then x is set to 0. More concisely, $x \leftarrow (x+1) \bmod 2^8$.

2. [5 points] Consider the following program; read it carefully:

```
#include <stdio.h>

void func(int* p) {

    int q = 2;

    p = &q;

    *p = 5;
}

int main() {

    int a = 9;

    func(&a);

    printf("%d\n", a);

    return 0;
}
```

What does this program print?

9. Reason: func resets its argument, a pointer, to be the address of a local variable (q). So in the line *p=5, the variable that is changed is int q. This leaves int a unaffected.

Name: Solutions **NYU NetId:**

3. [8 points] Consider the program below, which makes use of `fork()` and `exec()`. As a reminder, the Unix command `echo` outputs its arguments.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <errno.h>

int main()
{
    int rc = fork();

    if (rc < 0) {

      fprintf(stderr, "fork: %s\n", strerror(errno));
      exit(1);

    } else if (rc == 0) {

        char* argv[3];
        argv[0] = "echo";
        argv[1] = "xyz";
        argv[2] = NULL;  // tells execvp() that there are no more arguments

        // below, execvp() is a variant of exec()

        if (execvp(argv[0], argv) < 0)
            fprintf(stderr, "exec: %s\n", strerror(errno));

        printf("abc");

    } else {
        wait(NULL);
        printf("def");
    }

    return 0;
}
```

What is the output when this program is run?

xyz
def

II ls lab and debugging (15 points total)

4. [15 points] Your friend is taking a CS class. They are given an assignment but implement it buggily. We will state the assignment, then give you a sample run, then ask you to find and fix the bug.

Your friend's assignment is to write simplels, which is a version of ls. simplels takes one argument, a directory name. There are no command-line flags. So the form is:

```
$ ./simplels <dirname>
```

Your friend's assignment specified that simplels must operate as follows:

- First print out the directory name.
- Then, for each entry in that directory:
  - Skip entries that begin with . (the "dot" character);
  - Print the entry name;
  - Print a trailing / if the entry is a directory (as in lab2);
- In case of error, simplels's behavior is undefined, meaning students are free to handle errors (or not) however they want.

Below is a sample run of your friend's simplels implementation. The mkdir line creates a directory called foo, and the line after creates some files in that directory: abcd, mnopq, etc.

```
$ mkdir foo
$ touch foo/abcd foo/mnopq foo/rstuv foo/wxyz
$ ./simplels foo
foo:
$ echo $?
2
```

This run gives you some debugging hints. For example, notice that simplels runs and does so without dumping core. Thus, the bug is not a syntax or memory error. And notice that while simplels produces some output, it wrongly fails to print the files within foo.

Your friend's buggy implementation is on the next page, followed by some useful definitions.

Name: Solutions **NYU NetId:**

```
1   int main(int argc, char* argv[]) {
2       struct stat sb;
3       struct dirent* direntp;
4       DIR* d;
5
6       if (argc != 2) {
7           printf("error\n");
8           exit(2);
9       }
10
11      d = opendir(argv[1]);
12      if (d == NULL)
13          exit(2);
14
15      printf("%s:\n", argv[1]);
16
17      // assign result of readdir() to direntp, and check for NULL
18      while ( (direntp = readdir(d)) != NULL)  {
19
20          if (direntp->d_name[0] == '.')
21              continue;
22
23          if (stat(direntp->d_name, &sb) < 0)
24              exit(2);
25
26          printf("%s", direntp->d_name);
27
28          if (S_ISDIR(sb.st_mode)) // is directory?
29              printf("/");
30          printf("\n");
31      }
32      return 0;
33  }
```

```
// Useful definitions:
struct dirent {
   char   d_name[256]; /* Null-terminated filename */
   ...                 /* Other members; aren't needed here */
};

// opendir() opens a directory stream corresponding to the directory name,
// and returns a pointer to the directory stream.
DIR *opendir(const char *name);

// readdir() returns a pointer to a dirent structure representing the next
// directory entry in the directory stream pointed to by dirp. It returns NULL
// on reaching the end of the directory stream or if an error occurred.
struct dirent *readdir(DIR *dirp);

// stat() retrieves information about the file pointed to by
// name, placing the information in the stat structure.
int stat(const char *name, struct stat *statbuf);
```

Name: Solutions **NYU NetId:**

What is the bug in the code? State the line number(s) and the problem. You don't need more than one sentence.

Line 23 (inferrable from the sample run). `stat()` is being passed only the filename, but `stat()` requires the full pathname. Most of you would have run into this issue in lab 2; in fact, the lab 2 template code is structured a particular way, in part to help you get around this issue.

State the fix below, with reference to line number(s). Use syntactically correct C. There is much more space below than you need.

III Concurrency (32 points total)

5. [6 points] Let `cv` be a condition variable, and let `mutex` be a mutex. Assume that there are only two threads, and a single CPU. Consider this pattern:

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

Under the above assumptions, when is this pattern correct? Follow the concurrency commandments. Your answer should not be longer than one sentence.

Never.

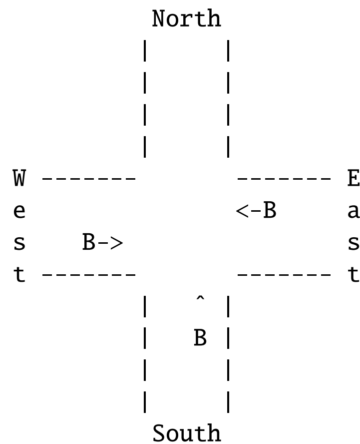
6. [6 points] Consider the implementation of the spinlock that we saw in class. This implementation relies on which of the following mechanisms?

Circle ALL that apply:

- **A** A mutex
- **B** An atomic processor instruction
- **C** A queue of quiescent waiters
- **D** An implicit memory barrier
- **E** Deadlock detection
- **F** A ticketing algorithm
- **G** Busy-waiting
- **H** A monitor

B,D,G

7. [20 points] In this problem, you will synchronize access to a *cross-roads* traveled by bicycles. Bicycles travel in straight lines. Each bicycle is headed north, south, east, or west. (If a bicycle is headed east, for example, then it approaches the cross-roads from the west.)



The cross-roads can be in *east-west mode*: this means that east and west bicycles can go, and the north and south bicycles must wait. Or it can be in *north-south mode*, which is the other way around. The mode changes if and only if the cross-roads is empty *and* an orthogonal (to the current mode) bicycle wishes to enter. The cross-roads can hold 5 bicycles in each of the two current directions. For example, if the cross-roads is in east-west mode, it can accommodate 5 bicycles going east and 5 going west.

The cross-roads is a monitor; you will complete the implementation of this monitor. Each bicycle is a thread that calls into the monitor before entering the cross-roads; this call may cause the bicycle to wait. A bicycle also invokes the monitor after exiting the cross-roads. Pseudocode for a bicycle is on the next page. Some notes about your task:

- Your solution must allow multiple bicycles to be using the intersection at once, and should not make any assumptions about the number of bicycles.
- You must follow the concurrency commandments. You must use only one mutex. You must not have busy waiting or spin loops.
- We have given you some helper functions that may be useful.

Below, write down the conditions under which a bicycle can and cannot enter the cross-roads. Informal text is fine. This exercise will help you with the rest of the problem.

Name: Solutions **NYU NetId:**

In the remainder of the problem, complete the implementation of the Xroads monitor.

```
typedef enum {NORTH=0, SOUTH=1, EAST=2, WEST=3} dir_t;
typedef enum {NORTHSOUTH=0, EASTWEST=1} mode_t;

void bicycle(thread_id tid, Xroads* xr, dir_t direction)
{
    /* you should not modify this function */
    Ride_up_to_crossroads();

    xr->Enter(direction);
    Ride();
    xr->Exit(direction);

    Ride_after_crossroads();

}

class Xroads {

    public:
        Xroads(); // You will complete this
        ~Xroads() { }
        void Enter(dir_t d); // You will implement this
        void Exit(dir_t d);  // You will implement this

    private:
        bool IsEmpty();
        mode_t Dir2Mode(dir_t d);
        mode_t mode;
        uint32_t num[4];

        // ADD MATERIAL BELOW THIS LINE




};

// HELPER FUNCTIONS
bool Xroads::IsEmpty()
{
    return num[0] == 0 && num[1] == 0 && num[2] == 0 && num[3] == 0;
}

mode_t Xroads::Dir2Mode(dir_t d)
{
    if (d == NORTH || d == SOUTH) return NORTHSOUTH;
    else                          return EASTWEST;
}
```

```
// Below, complete the implementation of
//    Xroads::Xroads()
// and give the implementations of
//    void Xroads::Enter(dir_t d)
//    void Xroads::Exit(dir_t d)
// Reminder: you need to add to the definition of Xroads on the previous page
Xroads::Xroads()
{
    memset(num, 0, sizeof(num));
    mode = NORTHSOUTH;

    // ADD SOME STUFF HERE


}
```

Additional data members in `Xroads`:

```
class Xroads {

....
   private:
        ...
        Mutex m;
        Cond cv;

};
```

Methods:

```
Xroads::Xroads()
{
    ....
    m.init();
    cv.init();
}

void
Xroads::Enter(dir_t d)
{
    m.acquire();
```

```
    while (!IsEmpty() && (dir2mode(d) != mode || num[d] >= 5))
        cv.wait(&m);

    if (IsEmpty())
        mode = dir2mode(d);

    ++num[d];

    m.release();

}

void
Xroads::Exit(dir_t d)
{
    m.acquire();

    --num[d];

    cv.broadcast(&m);

    m.release();
}
```

IV Virtual memory (21 points total)

8. [5 points] Consider a (small) machine with a 11-bit virtual addresses, in which 8 bits are used for the VPN. (Assume that the machine is byte-addressable, as with all of the examples we have seen.)

What is the size of the virtual address space, in bytes?

2^{11}, or 2048, bytes.

9. [7 points] Consider a (huge) machine that has 60-bit virtual addresses, a page size of 1 terabyte (1 TB, or 2^{40} bytes), and 52-bit physical addresses.

How many bits is the VPN (virtual page number)?

$60-40=20$.

How many bits is the PPN (physical page number)?

$52-40=12$.

How many bits is the offset?

40.

10. [9 points] Now consider the x86-64 architecture. Below we are asking about the *physical pages consumed by a process, including the page tables themselves*. As you answer the question, assume that any allocated memory consumes physical pages in RAM; that is, there is no swapping or demand paging. Note that it may be helpful for you to draw pictures (but you don't have to).

As a reminder, the x86-64 imposes a multi-level page table structure: pages are 4KB, each page table entry is 8 bytes, and each individual page table (a node in the "tree") occupies one page. Thus, each page table holds $\frac{4\text{KB}}{8\text{B}} = 512 = 2^9$ entries. Recall that the structure is four levels; each level is indexed by 9 bits of the virtual address.

What is the minimum number of physical pages consumed by a process that allocates 12KB (for example, 1 page each for code, stack, and data)?

7 pages: 3 for the memory it's allocated, and 4 to implement the paging structure.

What is the minimum number of physical pages consumed by a process that makes $2^9 + 1$ allocations of size 4KB each? You can leave your answer in terms of powers of 2, and sums thereof.

$2^9 + 6$, or 518 pages. Each L4 page table holds 2^9 mappings. So we need two L4 page tables. The total is $1+1+1+2$ for the page structures plus the $2^9 + 1$ pages themselves.

What is the minimum number of physical pages consumed by a process that makes $2^{18} + 1$ allocations of size 4KB each? You can leave your answer in terms of powers of 2, and sums thereof.

$2^{18} + 2^9 + 6$. Each L3 page table points indirectly to 2^{18} last-level page entries (each L3 page table has 2^9 entries, each of which points to an L4 page table with 2^9 entries). Thus, the question requires two L3 page tables. The first L3 page table points to 2^9 L4 page tables; the second points to one L4 page table, for a total of $2^9 + 1$ L4 page tables. Thus, the total is: $1 + 1 + 2 + 2^9 + 1$ for the page structures plus $2^{18} + 1$ for the pages themselves.

V Scheduling and event-driven programming (14 points total)

11. [8 points] This question is about scheduling disciplines. MLFQ refers to the Multi-Level Feedback Queue presented in the textbook (OSTEP, Chapter 8), rather than in class. We grade True/False questions with positive points for correct items, 0 points for blank items, and negative points for incorrect items. The minimum score on this question is 0 points. To earn exactly 1 point on this question, cross out the question and write SKIP.

Circle True or False for each item below:

True / False MLFQ requires tracking how much CPU each process has used. True.

True / False Stride scheduling requires tracking how much CPU each process has used. True.

True / False With MLFQ scheduling, starvation is possible. False. Periodically, all processes are bumped to the highest priority level, and then run round-robin.

True / False With priority scheduling, starvation is possible. True.

True / False In MLFQ, a job's priority is enhanced when tickets are donated to it. False. MLFQ has no notion of tickets.

True / False Round robin (RR) optimizes CPU throughput. False. RR necessarily has context switches, which are overhead, and detract from throughput.

True / False Round robin (RR) optimizes average turn-around time. False. We saw counter-examples in class.

True / False First-come, first-served (FIFO) is an unfair scheduling policy.

Full credit for True, False, or blank. Intended answer: True, because while the name sounds fair, it isn't: an earlier long job prevents later jobs from getting processor time at all, until the long job completes. However, this answer is inconsistent with the course notes. According to the course notes, the answer should be False because one definition of fairness is freedom from starvation, and FIFO meets that definition since each job eventually runs. Given that we messed up the question, all answers received credit.

12. [4 points] The following questions are about event-driven programming. Assume that the machine has a single CPU and that the system supports asynchronous disk I/O. See above for our instructions on True/False questions. Again, to earn 1 point on this question, cross out the question and write SKIP.

Circle True or False for each item below:

True / False In event-driven programming, the functions dispatched by the event loop must be careful to always block. False. They must never block.

True / False In an event-driven program, the implementation of the event loop does not require locks. True. That is part of the point of the event-driven style.

True / False In an event-driven program, event handlers may need to acquire locks. False. Nothing about the event-driven paradigm requires locks at application-level (threading isn't required under the assumptions).

True / False Event-driven programming is compatible with asynchronous network I/O. True.

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.

End of Midterm