New York University
CSCI-UA.202: Operating Systems (Undergrad): Spring 2021
Midterm Exam

- This exam is **75 minutes**.
- The answer sheet is available here: **Answer sheet**. The answer sheet has the hand-in instructions and the selfie upload instructions.
- There are **12** 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.
- If you have questions about the exam please go to https://campuswire.com/p/XXXXXXX (if you need access, use code YYYY).
- **Write your name and NetId on the document in which you are working the exam.**

Do not write in the boxes below.

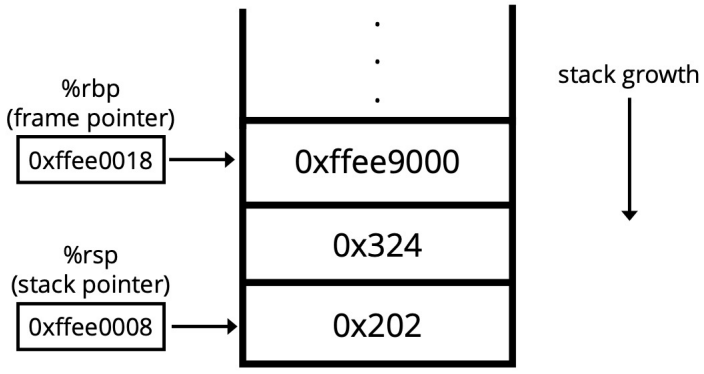
I (xx/8)	II (xx/25)	III (xx/12)	IV (xx/24)	V (xx/18)	VI (xx/13)	Total (xx/100)

I Stack frames (8 points)

1. [8 points] Below is an excerpt, in x86-64 assembly, from a function `f`, as well as the full listing of a function `g`. Assume each stack slot contains 8 bytes.

```
1       excerpt from f:
2
3           movq $0x324, %rbx ;; %rbx <-- 0x324
4           pushq %rbx        ;; put the value in %rbx on the stack
5           movq $0x202, %rax ;; %rax <-- 0x202
6           pushq %rax        ;; put the value in %rax on the stack
7           call g            ;; invoke g
8
9           ;; <rest of function omitted>
10
11      g:
12          pushq %rbp        ;; push frame pointer
13          movq  %rsp, %rbp  ;; frame pointer <-- stack pointer
14
15          ;; swap the contents of memory addresses 0x20000 and 0x30000
16          movq 0x20000, %rax  ;; load from 0x20000 to %rax
17          movq 0x30000, %rcx  ;; load from 0x30000 to %rcx
18          movq %rcx, 0x20000  ;; store from %rcx to 0x20000
19          movq %rax, 0x30000  ;; store from %rax to 0x30000
20
21          movq %rbp, %rsp    ;; stack pointer <-- frame pointer
22          popq %rbp          ;; pop frame pointer
23
24          retq
```

Assume that right before line 7 executes, the frame pointer, stack pointer, and top of the stack (meaning the growing end) look like this:



What are the stack contents when the flow of execution reaches line 23? Also state the values of the stack pointer and the frame pointer at that point.

Name: Solutions **NYU NetId:**

stack:
0xffee9000
0x324
0x202
8 ;; return address from g

The stack pointer points to the 8, which is memory address 0xffee0000.

The frame pointer holds 0xffee0018.

II Labs (25 points)

2. [10 points] In Lab 2, you likely invoked `stat()` or `lstat()`, to request information about a file, based on name. Using the `fstat()` system call, you can also ask for information about already-open files, using the file descriptor as a reference to the file. In this problem, you will use `fstat()` to write `file_size()`, which takes as input a file descriptor and returns the file size in bytes, or -1 in case of error. The prototype of `fstat()` and other helpful definitions are below. Note that there is (purposefully) much more information than you need below.

Write `file_size()` in syntactically valid C, using `fstat()`.

```
/* fd represents a file. Return file size in bytes, or -1 on error.*/
off_t file_size(int fd)
{
    // YOUR CODE HERE









}
```

```
/* Return information about the file specified by file descriptor fd.
   Places the returned information in the buffer pointed to by statbuf.
   On success, return 0. On error, return -1. */
int fstat(int fd, struct stat *statbuf);

struct stat {
   dev_t     st_dev;         /* ID of device containing file */
   ino_t     st_ino;         /* Inode number */
   mode_t    st_mode;        /* File type and mode */
   nlink_t   st_nlink;       /* Number of hard links */
   uid_t     st_uid;         /* User ID of owner */
   gid_t     st_gid;         /* Group ID of owner */
   dev_t     st_rdev;        /* Device ID (if special file) */
   off_t     st_size;        /* Total size, in bytes */
   blksize_t st_blksize;     /* Block size for filesystem I/O */
   blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

   struct timespec st_atim;  /* Time of last access */
   struct timespec st_mtim;  /* Time of last modification */
   struct timespec st_ctim;  /* Time of last status change */

};
```

```
/* fd is supposed to represent an open file */
off_t file_size(int fd)
{
    // YOUR CODE HERE
    struct stat sb;
    if (fstat(fd, &sb) < 0)
        return -1;

    return sb.st_size;
}
```

3. [15 points] Consider a simplified variant of the Lab 3 setup, with these components:

- A *BoolQueue*. This is like the Lab 3 `TaskQueue`, but it holds `bool`s rather than `Task`s. The declaration of this object is below; its interface is all that matters for this problem. The `BoolQueue` is a monitor that is implemented correctly and internally synchronized.
- One or more *consumer* threads that loop, removing `bool`s from the queue, and acting based on the value of the dequeued `bool`:
  - If the value is `true`, the consumer prints "hello" on its own line.
  - If the value is `false`, the consumer thread exits.
- A *producer* thread that places `bool`s in the queue, in such a way that the consumer threads together print "hello" 22 times, and all consumer threads eventually exit. Your code should make no assumptions about which consumer threads do which printing.
- A `start()` function that creates six concurrent threads: one thread that executes `produce()` and five threads that execute `consume()`. `start()` should return only after all of the created threads have exited. The sthread interface will be handy; it is included below.

On the next page, implement `start()`, `produce()`, and `consume()` in syntactically valid C++.

```
class BoolQueue {
    // This is a monitor that is implemented correctly and internally
    // synchronized. Callers simply invoke enqueue() and dequeue().

    public:
      BoolQueue();                  // you can ignore for this problem
      ~BoolQueue();                 // you can ignore for this problem

      // the main interface.
      void enqueue(bool directive); // enqueues a Boolean (rather than a Task)
      bool dequeue();               // dequeues a Boolean (rather than a Task)

    private: // Omitted; the implementation doesn't matter in this problem
};


// Create a thread, returning its id in the address "thrd".
// The created thread will start by executing start_func(argToStart).
void sthread_create(sthread_t *thrd, void (*start_func)(void*), void *argToStart);
void sthread_exit(void);

// Block until the specified thread exits. If the thread has
// already exited, this function returns immediately.
void sthread_join(sthread_t thrd);
```

```
void start() {

    BoolQueue bq;
    sthread_t producer_tid;
    sthread_t consumer_tid[5];

    // YOUR CODE HERE:
    // Start 6 concurrent threads:
    //  - 5 threads running consume()
    //  - 1 thread running produce()
    // Return after all 6 threads finish









}

void produce(void* arg)
{
    BoolQueue* bq = (BoolQueue*)arg;

    // YOUR CODE HERE:

    // Place values in the queue such that: the consumers will
    // collectively print "hello" 22 times, and all 5 consumers exit.









}

void consume(void* arg)
{
    BoolQueue* bq = (BoolQueue*)arg;

    // YOUR CODE HERE:









}
```

```
void start() {
    BoolQueue bq;
    sthread_t producer_tid;
    sthread_t consumer_tid[5];
    int i;

    // YOUR CODE HERE:
    // Start 6 concurrent threads:
    //  - 5 threads running consume()
    //  - 1 thread running produce()
    // Return after all 6 threads finish

    sthread_create(&producer_tid, produce, &bq);

    for (i = 0; i < 5; i++)
        sthread_create(&consumer_tid[i], consume, &bq);

    sthread_join(producer_tid);

    for (i = 0; i < 5; i++)
        sthread_join(consumer_tid[i]);
}

void produce(void* arg)
{
    BoolQueue* bq = (BoolQueue*)arg;

    // YOUR CODE HERE:
    // Place values in the queue such that: the consumers will
    // collectively print "hello" 22 times, and all 5 consumers exit.
    int i;

    for (i = 0; i < 22; i++)
        bq->enqueue(true);

    for (i = 0; i < 5; i++)
        bq->enqueue(false);
}

void consume(void* arg)
{
    BoolQueue* bq = (BoolQueue*)arg;

    // YOUR CODE HERE:

    while (bq->dequeue())
        printf("hello\n");

    sthread_exit();
}
```

III System calls (12 points)

4. [4 points] Which system call that we have studied creates processes?

State the system call.

`fork()`.

5. [4 points] Which system call that we have studied replaces a process's memory space (its text, data, and so on) with that of a newly loaded program?

State the system call.

`exec()`.

6. [4 points] You want to learn about a system call, `mmap()`. What command do you issue at the shell prompt on our devbox (the Linux virtual machine) to read the system manual pages for `mmap()`?

State the command.

`man mmap`

IV Concurrent programming (24 points)

7. [24 points] You are admitting groups of tourists to a site. A group must have exactly 10 people (not more, not fewer). After a group is admitted, a new group of 10 can be admitted to the site only after everyone from the prior group has left. Everyone enters the site and exits through the same door. We model the problem with each tourist as a thread, in the loop below, synchronized with a shared monitor called a `Grouper`.

```
Grouper grouper;   // global

void tourist() {

    /* Tourist approaches the door, may have to wait inside Enter() */
    grouper.Enter();

    /* When we get here, this tourist and 9 others from the same group
       are visiting the site */

    visit_site();

    /* Tourist is done visiting, now walks out the door */
    grouper.Exit();
}
```

Your job is to implement `Grouper`. As usual, you must follow the class's concurrency commandments. A few notes and hints:

- Of course tourists may leave (via `Exit()`); however, consistent with the problem setup, a new group cannot form until everyone from the current group has left.
- You need not let tourists out of `Enter()` in the order in which they arrived.
- You cannot assume that a tourist will ever progress out of `visit_site()`.
- Don't wake threads unnecessarily.
- The `Grouper` can be in one of two states:
  - "There is room". In this state, tourists can add themselves to a forming group. When the group size is 10 people, the state transitions to "there is no room".
  - "There is no room". In this state, the group members from the formed group (see above) can and should visit the site (meaning, progress out of `Enter()`). But no new group can form. The state transitions to "there is room" only when all 10 group members have left the site, via `Exit()`.
- **Hints:** There are multiple points in `Enter()` where a thread may have to wait. Also, you may need to `broadcast()` in `Enter()`.

Where indicated, fill in the variables and methods for the `Grouper` object. Remember to follow the concurrency commandments.

```
class Grouper {

    public:
        Grouper();   // initializes state and synchronization variables
        ~Grouper();

        void Enter();
        void Exit();

    private:
        bool is_room; // tracks the state. true means "there is room".
        // ADD MORE HERE

};

Grouper::Grouper()
{
    is_room = true; // Monitor begins in the state "there is room"

    // FILL THIS IN




}

void
Grouper::Enter()
{
    // FILL THIS IN









}

void
Grouper::Exit()
{
    // FILL THIS IN




}
```

```
class Grouper {

    public:
        Grouper();   // initializes state and synchronization variables
        ~Grouper();

        void Enter();
        void Exit();

    private:
        bool is_room; // tracks the state. true is "there is room".
        int  num_in_group;
        mutex m;
        cond cv_theresRoom;
        cond cv_groupFormed;
};


Grouper::Grouper()
{
    is_room = true; // Monitor begins in the state "there is room"

    // FILL THIS IN
    num_in_group = 0;
    mutex_init(&m);
    cond_init(&cv_theresRoom);
    cond_init(&cv_groupFormed);
}

void
Grouper::Enter()
{
    // FILL THIS IN
    m.acquire();

    while (!is_room)
        cond_wait(&m, &cv_theresRoom);

    ++num_in_group;

    if (num_in_group == 10) {
        is_room = false;
        cond_broadcast(&m, &cv_groupFormed);
    }

    while (is_room)  // alt: "while (num_in_group < 10)"
        cond_wait(&m, &cv_groupFormed);

    m.release();
}
```

```
void
Grouper::Exit()
{
    // FILL THIS IN
    m.acquire();

    --num_in_group;

    if (num_in_group == 0) {
        is_room = true;
        cond_broadcast(&m, &cv_theresRoom);
    }

    m.release();
}
```

V Virtual memory (18 points)

8. [10 points] Consider a byte-addressed processor architecture with 30-bit virtual addresses. In this architecture, the memory management unit (MMU) expects a three-level page table structure: the upper 7 bits of an address determine the index into the first-level page table, the next 7 bits determine the index in the second-level page table, the next 7 bits determine the index in the third-level page table, and the bottom 9 bits determine the offset.

How many entries are in a first-level page table? Justify, in a single sentence.

The index is 7 bits, which implies $2^7 = 128$ possibilities.

How many entries are in a third-level page table? Justify, in a single sentence.

The index is 7 bits, which implies $2^7 = 128$ possibilities.

What is the page size on this machine? Justify, in a single sentence.

The offset is 9 bits, and each address addresses a byte, so a page is $2^9 = 512$ bytes.

What is the maximum number of virtual pages per process? Justify, in a single sentence.

The VPN is 21 bits, which implies 2^{21} virtual pages per process.

9. [8 points] In this problem, you will describe how the implementation of `malloc()` can exploit paging so that the system (as a whole) can detect certain kinds of *out of bound accesses*; an out of bound access is when a process references memory that is outside an allocated range. In this problem we focus on *overruns*. Consider this code:

```
int *a = malloc(sizeof(int) * 100); /* allocates space for 100 ints */
a[0] = 5;    /* This is a legal memory reference */
a[99] = 5;   /* This is also a legal memory reference */
a[100] = 6;  /* This is an overrun, and is an illegal memory reference. */
```

When the above executes, the process would ideally page fault as a result of an illegal memory reference, at which point the kernel would end the process.

Assume that `malloc()` is a system call, so its implementation is inside the operating system, and thus can manipulate the virtual address space of the process.

Describe how the implementation of `malloc()` can arrange for page faults when there are overruns like the one above. Do not write more than three sentences.

Lay out the allocated memory so that the last legitimately allocated byte is on the last byte of the allocated page (this "wastes" the first part of a page). Mark the next virtual page (past the array) as "not in use" (this does not cost physical memory). At that point, memory references past the end of the array will generate page faults.

VI Scheduling and feedback (13 points)

10. [5 points] In class, we covered various CPU scheduling disciplines (FIFO, SJF, etc.). But we also said that CPU scheduling algorithms alone do not give a full (or even mostly complete) picture of how the resources of a computer are scheduled. One reason is that there are implicit schedulers of the CPU besides the operating system's scheduling algorithm.

Below, give an example of an entity in the system that implicitly schedules the CPU. Justify, in one sentence.

Interrupts or mutexes. Interrupts because they force the interrupt handler to run, which is not something the scheduler selected. Mutexes because by blocking threads, they are implicitly allocating the CPU. Other answers would work too.

11. [7 points] Consider a machine with a single processor (meaning one CPU core), 32 GB (gigabytes) of physical RAM, and a 2 TB (terabyte) hard disk drive. The operating system uses paged virtual memory with a local replacement policy ("local" here means that when the OS needs to bring in a page from disk, the OS considers for eviction only other physical pages used by the faulting process). The operating system schedules the CPU with a multi-level feedback queue (MLFQ); for reference, a summary of MLFQ from the textbook is below. There are two computation-intensive jobs running: Job A and Job B. ("Computation-intensive" means that the program code doesn't make I/O requests.) Job A has a working set size of 100 GB. Job B has a working set size of 50 MB (megabytes).

Over time, what do you expect to be the relative priority of the two jobs? (Job A will tend to be higher? Job B higher? They will tend to be equal?) Justify in no more than three sentences.

Job A will tend to have higher priority. Job B's working set fits in physical memory, so it consistently uses its entire quantum, thereby causing it to descend in priority (Rule 4). Job A, by contrast, will spend most of its time page faulting; this is because, although Job A was coded to be computation-intensive, its working set doesn't fit in physical memory. As a consequence, Job A will very often not consume its quantum, and thus continue to be in a high priority queue.

Some students wrote that because B would complete faster, its priority would be increased. That's how SJF works (assuming it could be implemented), but that isn't how MLFQ schedules. See the text for more on what MLFQ is optimizing.

8.6 MLFQ: Summary

We have described a scheduling approach known as the Multi-Level Feedback Queue (MLFQ). Hopefully you can now see why it is called that: it has *multiple levels* of queues, and uses *feedback* to determine the priority of a given job. History is its guide: pay attention to how jobs behave over time and treat them accordingly.

The refined set of MLFQ rules, spread throughout the chapter, are reproduced here for your viewing pleasure:

- **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue.
- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

MLFQ is interesting for the following reason: instead of demanding *a priori* knowledge of the nature of a job, it observes the execution of a job and prioritizes it accordingly. In this way, it manages to achieve the best of both worlds: it can deliver excellent overall performance (similar to SJF/STCF) for short-running interactive jobs, and is fair and makes progress for long-running CPU-intensive workloads. For this reason, many systems, including BSD UNIX derivatives [LM+89, B86], Solaris [M06], and Windows NT and subsequent Windows operating systems [CS97] use a form of MLFQ as their base scheduler.

12. [1 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