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

- **Write your name and NetId on this cover sheet (where indicated, at the bottom).**
- This exam is **75 minutes**. Stop writing when "time" is called. *You must turn in your exams; we will not collect them.* 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** problems 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.
- Do not waste time on arithmetic. Write answers in powers of 2 if necessary.
- 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/8)	II (xx/10)	III (xx/7)	IV (xx/17)	V (xx/24)	VI (xx/12)	VII (xx/8)	VIII (xx/14)	Total (xx/100)

Name: Solutions

NetId:

I Call-preserved registers (8 points)

1. [8 points] Consider the following assembly code for a function `f`. Registers `%r12` and `%r13` are call-preserved, otherwise known as callee-saved. This question asks how to restore these registers at the end of a function.

```
 1  f:
 2      # prolog
 3      pushq %rbp        # push frame pointer
 4      movq  %rsp, %rbp  # frame pointer <-- stack pointer
 5
 6      pushq %r12
 7      pushq %r13
 8
 9      subq $64, %rsp
10
11      ;; ... body of the function....
12
13      # epilog
14      ;; YOUR CODE HERE
15
16
17
18      movq %rbp, %rsp
19      popq %rbp
20      ret
```

Assume that line 11 represents many lines of code, in which registers `%r12` and `%r13` are used for computation. Thus, they no longer contain the values that they did on lines 6 and 7.

Furthermore, you cannot make any assumptions about what happens to `%rsp` (the stack pointer) in the body of the function. By the time execution reaches line 14, `%rsp` might be the same as at line 9, or it might be much lower.

The epilog of the function (starting in line 14) needs to restore the values of `%r12` and `%r13` to what they were at the start of the function. You will do that below.

Write assembly instructions, in syntactically valid x86-64, to restore the values of `%r12` and `%r13` to the values that they held just before line 6. Use no more than six instructions. (In fact, it can be done in two instructions.)

Preferred solution:

```
movq -8(%rbp), %r12
movq -16(%rbp), %r13
```

The following would also have worked, say if you didn't remember the syntax used above for base+offset addressing:

```
movq %rbp, %rsp
subq $16, %rsp
popq %r13
popq %r12
```

Many students wrote

```
popq %r13
popq %r12
```

But that is not correct, because it presumes that, at the start of the epilog, the stack pointer is in the same place as it was in line 9, which is not something that the function can assume (per the instructions).

The "fixed" location of `%r12` and `%r13` on the stack is the offset relative to the *frame pointer* (`%rbp`), not the stack pointer. That is in fact one of the purposes of the frame pointer: to "hold a spot" so that local variables (including call-preserved registers) have a fixed and well-known location so that they can be referenced.

II Buggy code (10 points)

2. [10 points] In lab 1, you worked with linked lists. Each node has the following structure:

```
struct list_node {
    int value;
    struct list_node* next;
};
```

Consider a function `list_insert`, which inserts a node (`new_node`) directly after an existing node (`prev`) in an existing list. The caller is responsible for making sure that `prev` is non-NULL.

```
void list_insert(struct list_node* prev, struct list_node new_node)
{
    assert(prev);
    new_node.next = prev->next;
    prev->next = &new_node;
}
```

The implementation above is buggy. State the bug and which line or lines must change to fix it. Use no more than three sentences.

(This was a lightly modified version of a homework problem. There is no syntactic or type issue.)

The problem is that (1) `new_node` is passed by value, so changes that it makes are lost when the function returns, and (2) `prev->next` is pointing to a local address within a stack frame, which will have undefined behavior as soon as the function returns.

To fix it, the second parameter should be `struct list_node*`, and the latter two lines should read: `new_node->next = prev->next` and `prev->next = new_node`.

III Using the Unix shell (7 points)

3. [7 points] Consider a server log file, `log_file`, in the following format:

```
2025-03-07 12:34:56 alice action:login status:success ip:192.168.1.101
2025-03-07 13:45:21 bob action:download status:failed ip:192.168.1.102
2025-03-07 14:15:33 charlie action:upload status:success ip:192.168.1.103
[thousands of lines in the same format as the above]
```

Your task is to construct a *single* line that, when typed at a Unix shell, extracts all successful actions from the log file (that is, lines with `status:success`), sorts those lines by username alphabetically (this is in the third field of the log file: `alice`, `bob`, `charlie`, and so on), takes only the first 50 entries, and saves those 50 entries to a file called `output.txt`.

You may find it helpful to compose some of the following commands, though there may be more commands below than you need:

`head -n NUM` `NUM` is a placeholder standing for a number. Reads from standard input, and writes the first `NUM` lines to standard output.

`find <starting point> -print` Writes to standard output the name of all files starting at `<starting point>`.

`grep "status:success" <file>` Extracts lines containing `status:success` from file `<file>`, and writes those lines to standard output.

`sort -k NUM` `NUM` is a placeholder standing for a number. This reads from standard input, and sorts lines based on field `NUM` (the first field on the left is field 1), writing the results to standard output.

`uniq` Omits repeated, adjacent lines from standard input, and writes the results to standard output.

Write a single line at the shell to accomplish the task.

$

```
grep "status:success" log_file | sort -k 3 | head -n 50 > output.txt
```

or

```
cat log_file | grep "status:success" | sort -k 3 | head -n 50 > output.txt
```

Name: Solutions **NYU NetId:**

IV Lab 2: `ls` (17 points)

4. [17 points] In this problem, you will implement a simplified, *non-recursive* version of `ls -n`, which in our `ls` lab counts the number of files in a given directory. We will break this into two steps, outlined below and on the next page.

First step: Implement a function `count_files(DIR* dir)` that returns the number of files in an open directory (non-recursively), or -1 if it encounters error. You can assume that `dir` is already open, and you do not need to close it. You can include pseudofiles ("`.`", "`..`"). We include some possibly useful reference functions at the end of this question (two pages ahead).

Implement `count_files` below in syntactically valid C.

Solution omitted because close to a lab assignment.

```
int count_files(DIR* dir)
{
    // YOUR CODE HERE
























}
```

Second step: Process the user's input so that when the user types `ls -n`, the number of files in the current directory, or "error" (in case of error), is printed. We include below an excerpt from the source for lab 2. You should assume that you can call a function:

```
DIR* get_curr_dir();
```

This helper function returns a directory stream representing the current directory, or NULL on error. You do not have to implement it.

Please modify or add to the code below, with reference to line numbers, so that `ls -n` works as described. Your modifications should be in syntactically valid C.

```
 1  int main(int argc, char* argv[]) {
 2      int opt;
 3      err_code = 0;
 4      bool list_long = false, list_all = false;
 5      // We make use of getopt_long for argument parsing, and this
 6      // (single-element) array is used as input to that function. The `struct
 7      // option` helps us parse arguments of the form `--FOO`.
 8      struct option opts[] = {
 9          {.name = "help", .has_arg = 0, .flag = NULL, .val = '\a'}};
10
11      while ((opt = getopt_long(argc, argv, "1a", opts, NULL)) != -1) {
12          switch (opt) {
13              case '\a':
14                  // Handle the case that the user passed in `--help`. (In the
15                  // long argument array above, we used '\a' to indicate this case.)
16                  help();
17                  break;
18              case '1':
19                  // Safe to ignore since this is default behavior for our version
20                  // of ls.
21                  break;
22              case 'a':
23                  list_all = true;
24                  break;
25              default:
26                  printf("Unimplemented␣flag␣%d\n", opt);
27                  break;
28          }
29      }
30
31      exit(err_code);
32  }
```

```
/* Return information about the file given by pathname.
   On success, return 0. On error, return -1. */
int stat(const char* pathname, struct stat *statbuf);

/* Same as the macro that you were given in lab2. Take as
  input 'info' and prints the given 'ch' if the permission 'mask' exists,
  or "-" otherwise. */
#define PRINT_PERM_CHAR(info, mask, ch) printf("%s", (info & mask) ? ch : "-");

/* Tests whether the argument refers to a directory */
bool is_dir(char* pathandname);

/* Get username for uid. Return 1 on error, 0 otherwise. */
static int uname_for_uid(uid_t uid, char* buf, size_t buflen);

/* Get group name for gid. Return 1 on error, 0 otherwise. */
static int group_for_gid(gid_t gid, char* buf, size_t buflen);

/* Open a directory */
DIR *opendir(const char *name);

/* Closes the directory stream associated with dirp. Returns 0 on success.
   On error, -1 is returned, and errno is set to indicate the error. */
int closedir(DIR* dirp);

/* Read a directory entry, returning a pointer to a dirent. It returns NULL on reaching
   the end of the directory stream or if an error occurred. If an error
   occurred, errno is set to a non-zero value. */
struct dirent *readdir(DIR *dirp);

/* A function that you may have filled in when implementing lab 2 */
void list_file(char* pathandname, char* name, bool list_long);

/* A function that you may have filled in when implementing lab 2 */
void list_dir(char* dirname, bool list_long, bool list_all, bool recursive);

struct stat {
   dev_t     st_dev;
   ino_t     st_ino;
   mode_t    st_mode;
   nlink_t   st_nlink;
   uid_t     st_uid;
   gid_t     st_gid;
   dev_t     st_rdev;
   off_t     st_size;
   blksize_t st_blksize;
   blkcnt_t  st_blocks;
}
```

Additional space if needed

V Producer-Consumer patterns (24 points)

5. [24 points] In this problem, you will implement a variant of the producer-consumer pattern that we studied in class. Producer threads add items to one of two buffers: a *red* buffer or a *blue* buffer. Consumer threads remove items in pairs: one item from the red buffer and one from the blue buffer, at the same time. Pseudocode for the threads is below:

```
BufferWrapper bw;

// Data structure representing two items
struct ItemPair {
    Item item1;
    Item item2;
    ItemPair(Item, Item);
};

void producer()
{
    while (1) {

        // next line produces an item and puts it in nextProduced
        Item nextProduced = means_of_production();

        if (get_random_bit() == 0)
            bw.AddToRed(nextProduced);
        else
            bw.AddToBlue(nextProduced);
    }
}

void consumer()
{
    while (1)
        ItemPair item_pair = bw.ConsumeTwo();

        // next line abstractly consumes both items
        consume_item(item_pair);
    }
}
```

Your job is to implement `BufferWrapper`. A few notes:

- You must implement the buffers as last-in, first-out (LIFO), which means that the *most recently* added item to a buffer should also be the item to be removed next from that buffer. This is sometimes known as a *stack* (in comparison to a queue), but if that confuses you, you can ignore this point about vocabulary.
- The buffers are implemented as arrays, and have a maximum size.
- Each array requires an accompanying count, and the count also helps determine which item should be consumed next.
- You must follow the class's concurrency commandments.
- Do not wake threads unnecessarily.

Fill in variables and methods for the `BufferWrapper` object. There are six (6) places to fill in code. Pseudocode is acceptable.

```
class BufferWrapper {

  public:
    BufferWrapper();
    ~BufferWrapper();

    void AddToRed(Item);
    void AddToBlue(Item);
    ItemPair ConsumeTwo();

  private:
     Item red_buffer[LIFO_SIZE];     // array of Items
     Item blue_buffer[LIFO_SIZE];    // array of Items

     // ADD MATERIAL HERE (1)
     mutex M;
     cond bothnonempty;
     cond redspace;
     cond bluespace;
     int red_size;
     int blue_size;
};

void
BufferWrapper::BufferWrapper()
{
    // FILL THIS IN (2)

    red_size = blue_size = 0;
    mutex_init(&M);
    cond_init(&bothnonempty);
    cond_init(&redspace);
    cond_init(&bluespace);
}

void
BufferWrapper::~BufferWrapper()
{
    // FILL THIS IN (3)
    mutex_destroy(&M)
    cond_destroy(&bothnonempty);
    cond_destroy(&redspace);
    cond_destroy(&bluespace);
}

// Add the Item to the next free slot in the red buffer. If
// this buffer is full (there are already LIFO_SIZE elements),
// then block.
void
BufferWrapper::AddToRed(Item item)
{
    // FILL THIS IN (4)
    M.acquire();

    while (red_size == LIFO_SIZE)
        cond_wait(&redspace, &M);

    red_buffer[red_size] = item;
    ++red_size;
    if (blue_size > 0)
        cond_signal(&bothnonempty, &M);
    M.release();
}

// Add the Item to the next free slot in the blue buffer. If
// this buffer is full (there are already LIFO_SIZE elements),
// then block.
void
BufferWrapper::AddToBlue(Item item)
{
    // FILL THIS IN (5)
    M.acquire();

    while (blue_size == LIFO_SIZE)
        cond_wait(&bluespace, &M);

    blue_buffer[blue_size] = item;
    ++blue_size;
    if (red_size > 0)
        cond_signal(&bothnonempty, &M);
    M.release();
}
```

```
// Remove, and return, two Items, one each from the red
// buffer and the blue buffer. If the consumer cannot get one
// item from each of the buffers, it should block until there
// is at least one item in each buffer. Recall that you
// are removing in LIFO order.
ItemPair
BufferWrapper::ConsumeTwo()
{
    // FILL THIS IN (6)
    //
    //  Hint: you can make an ItemPair from two
    //  items i1 and i2 by writing "ItemPair(i1, i2);"

    M.acquire();
    while (red_size == 0 || blue_size == 0)
         cond_wait(&bothnonempty, &M);

    --red_size;
    --blue_size;
    Item ret1 = red_buffer[red_size];
    Item ret2 = blue_buffer[blue_size];

    cond_signal(&redspace, &M);
    cond_signal(&bluespace, &M);
    M.release();
    return ItemPair(ret1, ret2);
}
```

VI Mutex implementation (12 points)

6. [12 points] We went over a mutex implementation in class. It is on page 16 for reference. As a hint, you do not need to understand the details of the "else" branch to do this question. Assume a multi-CPU system, and assume that the hardware provides sequential consistency. Assume that multiple threads use this mutex.

A developer makes a *buggy* modification, removing a call to `release(&m->splock)`. This line is prefaced with `-`, in the diff below:

```
--- fair-mutex.c 2025-02-12 09:50:10
+++ fair-mutex-mod1.c 2025-03-11 23:36:29
@@ -21,11 +21,10 @@
     acquire(&m->splock);

     // Check if the mutex is held; if not, current thread gets mutex and returns
     if (m->owner == 0) {
         m->owner = id_of_this_thread;
-        release(&m->splock);
     } else {
         // Add thread to waiters.
         STAILQ_INSERT_TAIL(&m->waiters, id_of_this_thread, qlink);

         // Tell the scheduler to add current thread to the list
```

What kind of problem does the bug create? Circle the BEST answer below:

A Race condition

B Progress problem

B. Progress problem.

Now justify your answer above: (1) give a problematic execution or interleaving with reference to specific lines of code, and (2) say why the execution or interleaving leads to a race condition or progress problem. Use no more than three sentences.

After a thread T1 acquires an unheld mutex, it will spin forever in `mutex_release`, when it tries to acquire the spinlock. Similarly, after thread T1 acquires an unheld mutex, every other thread getting to `mutex_acquire` will spin endlessly (instead of entering the wait state).

The developer tries something else: they apply the modification below. The modifications are shown relative to the original on the next page (page 16), where lines prefaced with - are removed, and lines prefaced with + are added. Unfortunately, the code is still buggy.

```
--- fair-mutex.c 2025-02-12 09:50:10
+++ fair-mutex-mod2.c 2025-03-11 23:38:21
@@ -18,13 +18,13 @@

 void mutex_acquire(struct Mutex *m) {

-    acquire(&m->splock);
-
     // Check if the mutex is held; if not, current thread gets mutex and returns
     if (m->owner == 0) {
         m->owner = id_of_this_thread;
-        release(&m->splock);
     } else {
+
+        acquire(&m->splock);
+
         // Add thread to waiters.
         STAILQ_INSERT_TAIL(&m->waiters, id_of_this_thread, qlink);
```

What kind of problem does the new bug create? Circle the BEST answer below:

A Race condition

B Progress problem

A. Race condition. No issue with progress.

Now justify your answer above: (1) give a problematic execution or interleaving with reference to specific lines of code, and (2) say why the execution or interleaving leads to a race condition or progress problem. Use no more than three sentences.

If the mutex is currently unheld, and two threads on two processors try to acquire it at the same time, they could both read `m->owner` as 0, and both execute the "if" block, and then both exit `mutex_acquire`, thereby defeating the purpose: this is most definitely not mutual exclusion. The now-removed spinlock acquisition would have prevented that.

```
void mutex_acquire(struct Mutex *m) {

    acquire(&m->splock);

    // Check if the mutex is held; if not, current thread gets mutex and returns
    if (m->owner == 0) {
        m->owner = id_of_this_thread;
        release(&m->splock);
    } else {
        // Add thread to waiters.
        STAILQ_INSERT_TAIL(&m->waiters, id_of_this_thread, qlink);

        // Tell the scheduler to add current thread to the list
        // of blocked threads. The scheduler needs to be careful
        // when a corresponding sched_wakeup call is executed to
        // make sure that it treats running threads correctly.
        sched_mark_blocked(&id_of_this_thread);

        // Unlock spinlock.
        release(&m->splock);

        // Stop executing until woken.
        sched_swtch();

        // When we get to this line, we are guaranteed to hold the mutex. This
        // is because we can get here only if context-switched-TO, which itself
        // can happen only if this thread is removed from the waiting queue,
        // marked "unblocked", and set to be the owner (in mutex_release()
        // below).
    }
}

void mutex_release(struct Mutex *m) {
    // Acquire the spinlock in order to make changes.
    acquire(&m->splock);

    // Assert that the current thread actually owns the mutex
    assert(m->owner == id_of_this_thread);

    // Check if anyone is waiting.
    m->owner = STAILQ_GET_HEAD(&m->waiters);

    // If so, wake them up.
    if (m->owner) {
        sched_wakeone(&m->owner);
        STAILQ_REMOVE_HEAD(&m->waiters, qlink);
    }

    // Release the internal spinlock
    release(&m->splock);
}
```

VII Scheduling (8 points)

7. [8 points] Which of these scheduling disciplines are guaranteed not to starve processes, regardless of the workload? Recall that a discipline is susceptible to starvation if there exists a workload that could forever prevent at least one job from ever getting processor time. Assume as a technical point that the system has infinite capacity to hold waiting jobs (if this assumption confuses you, you can ignore it).

Choose ALL that apply:

A FCFS/FIFO

B Round-robin

C Priority

D SJF

E STCF

F Stride

G Lottery

H CFS

I None of the above (meaning that for each scheduling discipline there are workloads that result in starvation)

J All of the above, if jobs are I/O bound

K All of the above, if jobs are CPU bound

A, B, F, G, H

VIII Virtual memory (14 points)

8. [7 points] Determine the number of page table entries (PTEs) that are needed for a machine architecture where virtual addresses are 20 bits and the page size is 2^{11} bytes (=2KB). (This question refers to *last-level* PTEs, so the number of levels of mapping is irrelevant. If this parenthetical confuses you, you can ignore it.)

How many PTEs are needed to map the entire address space? Show your work.

There are 9 $(= 20 - 11)$ bits to reference the VPN, which means there are 2^9 possible VPNs, which means $2^9 = 512$ entries are needed.

9. [7 points] This question assumes the x86-64 paging architecture that we covered in class: page size of $2^{12} = 4096$ bytes, four levels of page tables, virtual addresses are 64 bits but the upper 16 bits are not used.

Consider a virtual address whose bottom 48 bits are as follows:

```
0xffff ffff f000
```

To map the virtual address above to a physical address, which entry (by 0-based index) in the given process's L1 (top-level) page table would be used? "0-based" means that the entries are numbered starting from 0 (the first entry is index 0, the second entry is index 1, and so on).

Choose the BEST answer:

A 0
B 1
C 2
D 202
E 511
F 512
G 1023
H 1024
G With only the bottom 48 bits, we don't have enough information to answer.

E. The index into the L1 page table is determined by the upper 9 bits, which are all 1s. That indexes into the last entry in the L1 page table, which has index 511.

Additional space if needed

End of Midterm