New York University
CSCI-UA.0202-001: Operating Systems (Undergrad): Spring 2026
Midterm Exam (Token: V0)

- **Write your name, NetId, and assigned seat on this cover sheet (where indicated, toward the bottom).**
- 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 **11** questions (in **8** 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/9)	II (xx/10)	III (xx/10)	IV (xx/8)	V (xx/8)	VI (xx/15)	VII (xx/20)	VIII (xx/20)
						Total	

Name: Solutions

NetId:

Assigned Seat:

I Short answer (9 points)

1. [3 points] What specific term do we use to describe a segment of code that accesses shared resources and must not be concurrently executed by more than one thread?

Provide your answer in one phrase.

Critical section.

2. [3 points] Two CPUs that share memory are each trying to acquire a spinlock on an x86-64 system.

What specific mechanism ensures that the lock is only ever successfully acquired by one CPU at a time?

The atomic exchange operation, `xchg`

3. [3 points] Consider this system call:

```
int open(const char* path, int flags);
```

The returned value is either -1, indicating an error, or else a non-negative integer.

What is that integer called?

File descriptor

II Assembly (10 points)

4. [10 points] Consider the following x86-64 assembly. Line numbers are shown for reference.

```
90
91  caller: # this is a label
92  movq  $1, %rdi
93  addq  $2, %rdi
94
95  call  f
96
97  subq  $3, %rax

100 f:
101 pushq %rbp
102 movq  %rsp, %rbp
103 subq  $32, %rsp
104
105 # code that implements f
106
107 movq  %rbp, %rsp
108 popq  %rbp
109 ret
```

Starting with the instruction at line 92, in what order are the numbered lines executed?

Write the line numbers in the order they execute. List only the numbered lines that correspond to executed instructions.

92, 93, 95, 101, 102, 103, 105, 107, 108, 109, 97

Right before executing line 95, `%rsp` equals X. What is the value of `%rsp` immediately after executing line 109?

A X

B X - 8

C X + 8

D Cannot be determined

A

III Process control and shell (10 points)

5. [10 points] Consider the code below. This code uses the following system calls:

- `dup2(int existingfd, int newfd)`: This resets `newfd` to describe the same file or device as `existingfd`.
- `pipe(int fds[2])`: This allocates two file descriptors in the `fds` array. Data written to `fds[1]` appears on (can be read from) `fds[0]`.
- `execl(const char* path, const char* arg0, ...)`: One of the variants of `exec()`. The full pathname to the program appears in `path`. The following arguments (starting with `arg0`, which contains the program name itself) are passed as the command-line. These arguments are terminated with a `NULL`.

Assume that the code below executes without error. Recall that standard input is represented by file descriptor 0, and standard output is represented by file descriptor 1.

```
int main()
{
    int rc;
    int fdarray[2];
    pipe(fdarray);

    if ( (rc = fork()) == 0) {

        dup2(fdarray[0], 0);

        execl("/usr/bin/grep", "grep", "foo", NULL);

        printf("hello\n");

    } else if (rc > 0) {

        dup2(fdarray[1], 1);

        execl("/bin/ls", "ls", NULL);

        printf("greetings\n");

    }

}
```

The code above is equivalent to what command that one could type at a shell?

`ls | grep foo`

IV Memory consistency (8 points)

6. [8 points] In the code below, assume that the compiler does not reorder instructions. Assume that `write_first` and `write_second` are variables in memory that are initialized to 6 and shared between two threads. One thread runs `f()`, and the other thread concurrently runs `g()`.

```
int write_first = 6;
int write_second = 6;

int main(int argc, char** argv) {
    sthread_t tid_f, tid_g;

    create_thread(&tid_f, f, NULL);
    create_thread(&tid_g, g, NULL);

    sthread_join(tid_f);
    sthread_join(tid_g);
}

void f() {
    write_first = 7;
    write_second = 7;
}

void g() {
    int r1, r2; // local variables
    r2 = write_second;
    r1 = write_first;
    printf("r1: %d\n", r1);
    printf("r2: %d\n", r2);
}
```

Consider the following outputs, for the question on the next page:

A

```
r1: 7
r2: 7
```

B

```
r1: 6
r2: 6
```

C

```
r1: 7
r2: 6
```

D

```
r1: 6
r2: 7
```

E

```
<no output; deadlock>
```

Assume hardware that is not sequentially consistent. Because of buffering, memory operations issued by one CPU may be observed by other CPUs in a different order.

Which of the output choices on the prior page are possible? Choose ALL that apply.

If you do not know how to do this problem, you can write "Pass" for 1 point and move on.

A, B, C, D. Note that if the memory model is sequential consistency, then fewer interleavings are allowed, so anything that can happen in sequential consistency can also happen when hardware is not sequentially consistent. Under sequential consistency, there are interleavings that produce A, B, D. But with a weak memory model, option C is also possible, if the two memory writes done by `f()` arrive at shared memory in the opposite order to program order.

This is a variant of a problem that we worked in class.

V Scheduling (8 points)

7. [8 points] Consider a preemptively scheduled system where scheduling decisions happen every 1 time unit (you can think of a time unit as 100ms, but the unit does not matter). Starting in time unit 0, one job arrives every 2 time units, deterministically; every job needs 5 time units to complete. For each scheduling decision, the scheduler chooses to run the *most recently arrived job*. Assume that the system has capacity to hold all waiting and ready jobs.

What is the throughput of the system (jobs completed per unit time)? Explain, using no more than one sentence.

The throughput is 0. At each decision point, the scheduler chooses the most recently arrived job, and runs it for two time units, never to run that job again; but each job needs five time units to complete.

What's described here is a simplified version of a phenomenon known as livelock, where a system dedicates resources to recent arrivals and consequently never finishes processing jobs it had already dedicated resources to.

VI Virtual memory (15 points)

8. [5 points] A system uses 4KB pages. How many virtual pages are there if the virtual address size is 20 bits?

256. 4KB pages means 12 bits of offset. If the virtual address size is 20 bits, then there 8 bits for VPN, which means $2^8 = 256$ virtual pages.

9. [10 points] Consider two processes P1 and P2 running on the same x86-64 machine. Each process independently runs the program foo. The processes are isolated from one another. Here is an excerpt from `foo`:

```
char *p = 0x500000;
*p = 'a';
```

In both processes, this code executes successfully.

Assume that the L1 page table for P1 resides at physical memory address X, and that the L4 page table entry used when translating virtual address 0x500000 in P1 resides at physical memory address Y.

Circle True or False for each statement below.

True / False The L1 page table for P2 resides at physical memory address X.

False

True / False The L4 page table entry used when translating virtual address 0x500000 in P2 resides at physical memory address Y.

False

True / False The contents of the two L4 page table entries are the same.

False

True / False For virtual address 0x500000, the index into the L1 page table is the same in P1 and P2.

True

True / False For virtual address 0x500000, the index into the L4 page table is the same in P1 and P2.

True

VII Lab2: `ls` (20 points)

10. [20 points] You will write a C function in the context of lab2 that counts the total number of directories in a given path and all its subdirectories. Your function should:

- Count all directories in the given path, including the path itself
- Recursively count directories in all subdirectories
- Skip the pseudo-directories `.` and `..`

You will do this by filling in the TODOs on the next page. All of the system calls that you need, and useful helper functions, are on the two pages after the next one

Write your code in syntactically valid C. You do not have to worry about error handling.

```
int count_directories(const char *path) {
    DIR *dir;
    struct dirent *entry;
    struct stat stat_buf;
    char full_path[1024];
    // TODO: Your code here









}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <directory>\n", argv[0]);
        return 1;
    }
    int count = count_directories(argv[1]);
    printf("Total directories: %d\n", count);
    return 0;
}
```

Important Datatypes

```
struct dirent {
    uint32_t  d_ino;  /* Inode number */
    char   d_name[];  /* Filename string of entry (null-terminated) */
};

struct stat {
    mode_t st_mode; /* File type and mode */
    ino_t  st_ino; /* Inode number */
    dev_t  st_dev; /* Device ID */
    uid_t  st_uid; /* User ID of owner */
    gid_t  st_gid; /* Group ID of owner */
    off_t  st_size; /* Size in bytes */
    time_t st_atime; /* Time of last access */
    time_t st_mtime; /* Time of last modification */
    time_t st_ctime; /* Time of last status change */
    /* ... */
};
```

Utility functions

```
/**
 * Writes formatted output to a string, ensuring it's not larger than size.
 *
 * @param str Buffer to write to
 * @param size Maximum number of bytes to write
 * @param format Format string
 * @return Returns the number of characters
 *         that would have been written if size had been sufficiently large
 */
int snprintf(char *str, size_t size, const char *format, ...);
```

Directory Operations

```
/**
* Opens a directory stream corresponding to the directory named by name.
* @param name The path of the directory to open
* @return On success, returns a pointer to the directory stream.
*         On error, returns NULL and sets errno appropriately.
*/
DIR *opendir(const char *name);

/**
* Get the next directory entry in the directory stream
* @param dirp Pointer to a directory stream obtained from opendir()
* @return On success, returns a pointer to a dirent structure.
*         On error or end of directory, returns NULL.
*/
struct dirent *readdir(DIR *dirp);

/**
* Closes the directory stream associated with dirp.
* @param dirp Pointer to a directory stream obtained from opendir()
* @return Returns 0 on success. On error, returns -1 and sets errno appropriately.
*/
int closedir(DIR *dirp);
```

File Status Operations

```
/**
* Gets information about the file pointed to by pathname
* and fills in the stat structure.
* @param pathname Path to the file
* @param statbuf Pointer to a stat structure where the information is stored
* @return Returns 0 on success. On error, returns -1 and sets errno appropriately.
*/
int stat(const char *pathname, struct stat *statbuf);

/**
* Macro that tests if a file is a directory.
* @param mode The st_mode field from a stat structure
* @return Returns non-zero if the file is a directory, zero otherwise.
*/
int S_ISDIR(mode_t mode);
```

VIII Concurrent programming (20 points)

11. [20 points] In this problem you will synchronize access to a shared array of integers. Producers insert a positive integer; the integer goes into any unused slot. If there are no unused slots, the producer waits.

A consumer removes and returns the largest stored integer. However, a consumer may proceed with such removal only if either (a) the sum of the stored integers exceeds 100 or (b) all slots are occupied. Otherwise, the consumer waits.

```
SharedArray mon;

void producer()
{
    for (;;) {
        // produce an integer to store
        int val = get_random_integer(1, 1000);

        mon.Put(val);
    }
}

void consumer()
{
    for (;;) {
        int val = mon.Take();

        // do something with the returned integer
        consume_value(val);
    }
}
```

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

- The `SharedArray` stores data in an array called `slots`. An entry in `slots` is 0 if the entry is unused.
- Your solution should work for an arbitrary number of producers and consumers.
- Follow the class's concurrency commandments and coding standards.
- Do not wake threads unnecessarily. Similarly, do not block threads unnecessarily.
- You may call the helper function `get_max_index()`, but you do not need to implement it. The definition is on the next page. You may need to define and use additional helper functions.

Fill in the missing variables and methods for `SharedArray` on the next two pages.

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

        void Put(int val);
        int Take();

    private:
        // the actual storage
        int slots[NUM_SLOTS];
        int get_max_index();
        int get_free_slot();
        int sum_all();

        mutex_t m;
        cond_t  can_put;
        cond_t  can_take;
};

SharedArray::SharedArray()
{
    // Initialize 'slots' to 0
    memset(slots, 0, sizeof(slots));

    // FILL THIS IN (2)
    smutex_init(&m);
    scond_init(&can_put);
    scond_init(&can_take);
}

// Add 'val' to an unused slot if one exists. Otherwise, wait until a slot
// is available.
void
SharedArray::Put(int val)
{
    // FILL THIS IN (3)

    smutex_lock(&m);
    int slot;
    while ((slot = get_free_slot()) == -1)
        cond_wait(&can_put, &m);
    slots[slot] = val;
    scond_signal(&can_take, &m);

    smutex_unlock(&m);
}

// If the sum of the stored integers is > 100 or the slots are full,
// then remove and return the maximum stored integer. Otherwise, wait.
int
```

```
SharedArray::Take()
{
    int i;
    int ret;
    smutex_lock(&m);

    while (!(sum_all() > 100 || get_free_slot() == -1))
        cond_wait(&can_take, &m);

    i = get_max_index();
    ret = slots[i];
    slots[i] = 0;

    scond_signal(&can_put, &m);
    smutex_unlock(&m);
    return ret;
}

int
SharedArray::get_free_slot()
{
    for (int i = 0; i < NUM_SLOTS; i++)
        if (slots[i] == 0)
            return i;

    return -1;
}


int
SharedArray::sum_all()
{
    int running_sum = 0;
    for (int i = 0; i < NUM_SLOTS; i++)
        running_sum += slots[i];

    return running_sum;
}

// not needed for solution
int
SharedArray::get_max_index()
{
    int max_index = 0;
    int max = 0;
    for (int i = 0; i < NUM_SLOTS; i++)
        if (slots[i] > max) {
            max_index = i;
            max = slots[i];
        }
```

```
    return max_index;
}
```

Scratch space if needed

Scratch space if needed

Scratch space if needed

End of Midterm