New York University
CSCI-UA.0202-002: Operating Systems (Undergrad): Fall 2024
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. *You must turn in your print-out and blue books; we will not collect them.* 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 **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/7)	III (xx/26)	IV (xx/7)	V (xx/4)	VI (xx/6)	VII (xx/12)	Total (xx/70)

Name: Solutions

NetId:

,

I Buggy compiler (8 points)

1. [8 points] Consider this C code: `f()` is supposed to return `2*x + (x+5)`, for `x` set to 17. (We do not show the C code for the functions that `f()` calls.) The C code is not buggy:

```
// computes and returns 2*x + (x + 5), for x = 17
int f() {

    int x, y, z;

    x = 17;

    y = double_x(x);
    z = x_plus_5(x);

    z += y;  // equivalent to: z = z + y

    return z;
}
```

But `f()` returns the wrong answer because the compiler has a bug! The compiler produces the buggy x86-64 code given on the next page.

Identify specifically what the code does wrong. If you cannot pinpoint it, then for partial credit state both (1) what a correct `f()` would return and (2) what the compiled `f()` does return.

Hints:

- The arguments to `movq` are in the order `movq SOURCE, DESTINATION`.
- When a function is called, `%rdi` holds the first argument to it. When a function returns, it places the return value in `%rax`.
- Because we know that `x` is 17, the bug here is not any kind of overflow.
- If a line is commented, then the bug is guaranteed not to be in that line.

```
1  f:                   # f has no prolog, but that is fine (not a bug)
2      movq $17, %rdi
3      call double_x
4      movq %rax, %rbx
5
6      movq $17, %rdi
7      call x_plus_5
8
9      addq %rbx, %rax  # this means "%rax <-- %rax + %rbx" (adds %rbx to %rax)
10
11     ret
12
13 double_x:
14     pushq %rbp       # Standard beginning of prolog
15     movq %rsp, %rbp  # Standard second line of prolog
16
17     imul $2, %rdi    # Multiplies %rdi by 2, puts result in %rdi
18     movq %rdi, %rax  # The return value of double_x will be the contents of %rdi
19
20     movq %rbp, %rsp
21     popq %rbp
22     ret
23
24 x_plus_5:
25     pushq %rbp       # Standard beginning of prolog
26     movq %rsp, %rbp  # Standard second line of prolog
27
28     movq %rdi, %rbx
29     addq $5, %rbx    # Adds 5 to %rbx, puts result in %rbx
30     movq %rbx, %rax  # The return value of x_plus_5 will be the contents of %rbx
31
32     movq %rbp, %rsp
33     popq %rbp
34     ret
```

The return value of `double_x()` is put in `%rbx` in line 4, and used in line 9. In between, the value is overwritten in line 28. To make the code correct, either `x_plus_5()` should have saved/restored `%rbx` in the prolog/epilog, or (if it were a call-clobbered register) `f()` should have saved `%rbx` (by pushing it on the stack) before calling `x_plus_5()` and restored `%rbx` (by popping the stack) after.

`f()` should return $34+22=56$, but instead returns $22+22=44$.

Name: Solutions **NYU NetId:**

II Memory consistency (7 points)

2. [7 points] In the code below, assume that that the compiler does not reorder instructions. Assume that `x` and `y` are variables in memory that are initialized to 0 and shared between two threads. One thread runs `f()`, and the other thread concurrently runs `g()`.

```
int x = 0;
int y = 0;

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

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

    sthread_join(tid_f);
    sthread_join(tid_g);

    return 0;
}

void* f(void*) {
    int r1; // local variable
    x = 1;
    r1 = y;
    printf("r1: %d\n", r1);

    sthread_exit(); return NULL;
}

void* g(void*) {
    int r2; // local variable
    y = 1;
    r2 = x;
    printf("r2: %d\n", r2);

    sthread_exit(); return NULL;
}
```

Consider the following outputs, which we have labeled with boldface **A** through **D**.

A
```
r1: 0
r2: 0
```

B
```
r1: 1
r2: 0
```

C
```
r1: 0
r2: 1
```

D
```
r1: 1
r2: 1
```

Assume sequentially consistent hardware (equivalent to executing on a single processor). Which of those options can be printed? Choose ALL that apply.

B, C, D. We did a version of this example in class. If it's on sequentially consistent hardware, there are interleavings that will produce B, C, D. For example, option D is produced by running `x=1` and `y=1` before either of the memory reads. Option A is not possible because if (for example) r1 read y as 0, then it means that r1=y happened before all of g, which means r2 will read x as 1. So both reads cannot return 0.

Now assume that the hardware does not provide sequential consistency. For example, assume that there are multiple CPUs, providing Total Store Order. Which of the output options can be printed? Choose ALL that apply.

A, B, C, D

If the hardware is not sequentially consistent, it certainly allows anything that is allowed in sequential consistency, so options B, C, D are possible. Option A is possible because with total store order, it might be that both memory writes are delayed in getting to shared memory, while the memory reads consume the initial values of x,y.

In that case, f would see:
x = 1;
r1 ← [starting value of y]

and g would see:
y = 1;
r2 ← [starting value of x]

and thus r1,r2 would both be equal to 0

That could produce option A.

III Multithreaded `ls` (26 points)

3. [26 points] This problem considers a simplified, but multithreaded, version of `ls`, called `mls`, which prints all files in a single directory (as when `ls` is in `-a` mode), non-recursively, prefixed by their sizes. Here is sample output for a directory `~/cs202-labs/ls-multi/foo`:

```
cs202-user@108fdce95b64:~/cs202-labs/ls-multi$ ./mls foo
6454 EStore.cpp
5457 RequestGenerator.cpp
3602 RequestHandlers.cpp
1714 TaskQueue.cpp
3801 sthread.cpp
352 .
576 ..
```

`mls` has multiple threads, each working on one file at a time within the given directory. (The motivation is cases where reading a file's information has high latency, and I/O throughput benefits from parallelism. If this parenthetical confuses you, you can ignore it.) `mls` instantiates this idea with a monitor, called `Dispatcher`, that hands out work to each caller. `Dispatcher` is constructed with a pointer to the given directory, and has the following methods, which you will implement:

- `start_item()`: Returns a new directory entry (representing a file) for the calling thread to list, or `NULL` if there are no more entries in the directory. If there are already `WINDOW` concurrent workers, then wait; otherwise, allow concurrency.
- `finish_item()`: Informs the monitor that the thread is done processing the given work item.

You will also implement a function that uses one or more system calls to actually produce the output:

- `file_printer()`: For each call, print the size and name for a *single* file.

Here are further specifications:

- Your implementation of `Dispatcher` must ensure that each file within the given directory is processed exactly once.
- Your code should not make any assumptions about the relationship between `WINDOW` and `NUM_THREADS`. Both of them are constants that the code references.
- Follow the concurrency commandments.
- Write in syntactically valid C or C++, and use the `sthread` library.
- Don't wake threads unnecessarily.
- You can assume that each individual call to `printf()` is atomic, and does not need to be protected from concurrent access.
- Possibly helpful definitions, including the `sthread` library, are on page 10. Note that there is more information than you need.

Code for the overall setup is on the next page, and the code that you will fill in is on the pages after that.

```
#define NUM_THREADS 10
char* given_directory = NULL;

void* worker(void* arg);

// execution starts here, as a result of a user invoking "./mls <dir>"
int main(int argc, char **argv) {
    sthread_t threads[NUM_THREADS];
    DIR* dirp;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <dir>\n", argv[0]); exit(-1);
    }

    given_directory = argv[1];
    if ((dirp = opendir(given_directory)) == NULL) {
        perror("opendir"); exit(-1);
    }

    Dispatcher dispatcher(dirp);

    for (int i = 0; i < NUM_THREADS; i++)
        sthread_create(&threads[i], &worker, &dispatcher);

    for (int i = 0; i < NUM_THREADS; i++)
        sthread_join(threads[i]);

    closedir(dirp);
    return 0;
}

void* worker(void* arg) {
    Dispatcher* dispatcher = (Dispatcher*)arg;
    struct dirent* direntp;

    do {
        direntp = dispatcher->start_item();

        if (direntp != NULL)
            // you will implement this function on the next page
            file_printer(given_directory, direntp->d_name);

        dispatcher->finish_item();

    } while (direntp);

    sthread_exit();
    return NULL;
}
```

Fill in the five places marked FILL IN on the next pages.

Name: Solutions **NYU NetId:**

```
// Print the file's size and its name. On error, simply return.
// This function should do no synchronization; if dispatcher
// is doing its job, then no other thread is currently working on
// the file "name".
void file_printer(const char* given_directory, const char* name)
{
    char buf[256];
    struct stat sb;
    snprintf(buf, sizeof(buf), "%s/%s", given_directory, name);

    // (1) FILL IN, BASED ON THE SPECIFICATION AND THE COMMENT ABOVE

    if (stat(buf, &sb) < 0)
        return;

    printf("%ju %s\n", sb.st_size, name);
}


class Dispatcher {

    public:
        Dispatcher(DIR* dirp);
        struct dirent* start_item();
        void finish_item();

    private:
        const int WINDOW = 5;
        DIR* m_dirp;

        // (2) FILL IN: MORE REQUIRED HERE
        int      m_count;
        smutex_t m_mutex;
        scond_t  m_cv;

};



Dispatcher::Dispatcher(DIR* dirp)
{
    m_dirp = dirp;

    // (3) FILL IN

    m_count = 0;
    smutex_init(&m_mutex);
    scond_init(&m_cv);
}
```

```
struct dirent*
Dispatcher::start_item()
{
    struct dirent* e = NULL;

    // (4) FILL IN, BASED ON THE SPECIFICATION
    smutex_lock(&m_mutex);

    while (m_count >= WINDOW) {
        scond_wait(&m_cv, &m_mutex);
    }
    e = readdir(m_dirp);
    ++m_count;

    smutex_unlock(&m_mutex);

    return e;
}


void
Dispatcher::finish_item()
{
    // (5) FILL IN, BASED ON THE SPECIFICATION
    smutex_lock(&m_mutex);
    --m_count;
    scond_signal(&m_cv, &m_mutex);

    smutex_unlock(&m_mutex);
}
```

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

/* This is the same as the macro that you were given in lab2. */
#define PRINT_PERM_CHAR(info, mask, ch) printf("%s", (info & mask) ? ch : "-");

/* convert the mode field in a struct stat to a file type, for -l printing */
const char* ftype_to_str(mode_t mode);

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

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

/* Read a directory entry, representing a file within the directory */
struct dirent *readdir(DIR *dirp);

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;
}

void smutex_init(smutex_t *mutex);
void smutex_destroy(smutex_t *mutex);
void smutex_lock(smutex_t *mutex);
void smutex_unlock(smutex_t *mutex);

void scond_init(scond_t *cond);
void scond_destroy(scond_t *cond);

void scond_signal(scond_t *cond, smutex_t *mutex);
void scond_broadcast(scond_t *cond, smutex_t *mutex);
void scond_wait(scond_t *cond, smutex_t *mutex);

void sthread_create(sthread_t *thrd,
                    void* (*start_routine)(void*),
                    void *argToStartRoutine);
void sthread_exit(void);
void sthread_join(sthread_t thrd);
```

IV Preemptive scheduling (7 points)

4. [7 points] Consider the setup below, which follows a format from class. Time is divided into epochs. Jobs arrive at the *very beginning* of the epoch listed under "arrival epoch"; after they have been given the CPU for the number of epochs listed under "length", they leave.

process	arrival epoch	length
P1	0	6
P2	1	1
P3	2	3

The system runs *preemptive priority scheduling*: it schedules the highest-priority *runnable* (also known as "ready") process, preempting another running process if necessary.

The system administrator assigns priorities to processes as follows:

P1 low
P2 medium
P3 high

The processes have the following structure. Assume that releasing a lock is instantaneous, as is acquiring the lock if it is available. Note that P1 and P3 share the same lock:

```
P1:
  acquire(&lock);
  run1();  // takes 6 epochs
  release(&lock);

P2:
  run2();  // takes 1 epoch

P3:
  acquire(&lock);
  run3();  // takes 3 epochs
  release(&lock);
```

Write down the process scheduled for each epoch:

0	1	2	3	4	5	6	7	8	9
P1	P2	P1	P1	P1	P1	P1	P3	P3	P3

V Lottery scheduling (4 points)

5. [4 points] Consider the statements below about lottery scheduling. Which of them is true?

Write down the letters of ALL that apply:

A Lottery scheduling is susceptible to starvation.

B Lottery scheduling, despite the name, is deterministic rather than random.

C With lottery scheduling, if there are two compute-bound processes, the administrator of the system can arrange for one of those processes to get twice as much processor time, over the long-term, as the other.

D Lottery scheduling attempts to allocate the processor proportionally.

E Lottery scheduling requires keeping scheduler state about past scheduling decisions.

C, D. See the textbook reading and class notes on lottery scheduling.

VI Therac-25 (6 points)

6. [3 points] How many turntable positions does the Therac-25 have?

Three

7. [3 points] What are three *non-software* problems that led to the Therac-25 disasters?

See paper, class notes, and class discussion.

VII Virtual memory (12 points)

8. [6 points] Assume a 52-bit virtual address space, 42-bit physical address space, and page size of 2KB. Determine the number of bits in the VPN (virtual page number), PPN (physical page number), and offset. Show your work, possibly by drawing pictures.

# of VPN bits **# of PPN bits** **# of offset bits**

2KB is 2^{11} bytes. Assuming each address references a byte, there are 11 offset bits. That leaves $52-11=41$ bits for the VPN and $42-11=31$ bits for the PPN.

9. [6 points] This question is about the x86-64 page table structures.

What is the *maximum* number of physical pages that could be consumed by a process that allocates a single virtual page of size 4KB? Show your work.

5. There is 1 each of L1, L2, L3, and L4 page tables (each of which itself occupies one page), and the page itself. In this case, the minimum (discussed in lecture) and the maximum are the same.

What is the *maximum* number of physical pages that could be consumed by a process that allocates 2^9 virtual pages of size 4KB each? Show your work.

The maximal way to do this is to activate every entry in the L1 page table (there are 512 such entries). If that is done, then (temporarily ignoring the L1 page table), each virtual page induces the consumption of a separate L2 page table, L3 page table, and L4 page table, plus the physical page: 4 pages per virtual page. Bringing back the sole L1 page table, we have: $2^9 \cdot 2^2 + 1 = 2049$.

This corresponds to allocating one page within each successive chunk of 2^{27} pages in the virtual address space. Imagine for example that the upper 9 bits (of the 36-bit VPN) take all 512 values, while the lower 27 bits of the VPN are 0. That will result in the aforementioned page table structure.

End of Midterm