

New York University
CSCI-UA.0202: Operating Systems (Undergrad): Spring 2023
Midterm Exam (Token: V0)

- Write your name, NetId, and token on this cover sheet and on the cover of your blue book.
- Put all of your answers in the blue book; we will grade only the blue book. Thus, if the blue book answer is blank or incorrect, you will not get credit, regardless of the exam print-out.
- 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 **11** 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.*

Do not write in the boxes below.

I (xx/14)	II (xx/15)	III (xx/20)	IV (xx/23)	V (xx/8)	VI (xx/10)	VII (xx/8)	Fb	Total (xx/100)

Name: Solutions

NetId:

I C and assembly (14 points)

1. [8 points] Below is a buggy excerpt from a C program.

```
800     int *ptr1, *ptr2; // declare pointers to integers
801
802     void f() {
803         int a = 0;
804         int* b = (int*) malloc(sizeof(int));
805         ptr1 = &a;
806         ptr2 = b;
807     }

950     int main() {
951         f();
952         *ptr1 = 1;
953         *ptr2 = 2;
954         free(ptr2);
955         // ...
956     }
```

Why is this program excerpt buggy? Explain in 1-2 sentences, pointing to a specific line or lines by number.

At line 805, the program stores a reference to memory that lives within a stack frame; by the time of line 952, the stack frame (of function `f`) has been destroyed. Therefore, the memory pointed to by pointer `ptr1` is invalid.

2. [6 points] Assume you are writing an x86-64 assembly program. You want (for whatever reason) to read the value of `%rip` (the instruction pointer, or program counter) into register `%rbx`. Conceptually you want the following:

```
movq %rip, %rbx # <-- NOT LEGAL in x86-64
```

But the above is not a legal instruction: in x86-64 assembly, `%rip` cannot be read directly. So instead we will build a work-around. You'll implement a non-traditional function called `worker` that consists of only two lines (no prologue or epilogue); the result of calling `worker` is to get the value of `%rip` into `%rbx`. The first line you will fill in. The second line is `ret`.

```
call worker
# when control reaches here, %rbx holds %rip

// .. rest of the program ...

# function definition
worker:
    # YOUR CODE BELOW (should be one line)

    ret
```

Fill in the line of code to complete `worker`.

```
movq (%rsp), %rbx
```

II Lab 2: ls (15 points)

3. [15 points] In this problem, you will write `print_owner_perms()`, which takes as input a file name and prints to `stdout` (standard out) the owner's permissions (not the group permissions or other permissions), or "error" in case of error. The prototype of `stat()` and other helpful definitions are below and on the next page. Note that there is (purposefully) much more information than you need below.

Write `print_owner_perms()` in syntactically valid C.

```
void print_owner_perms(const char* pathname)
{
    // YOUR CODE HERE

}

/* 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. 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 : "-");

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

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

/* 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 */
struct dirent *readdir(DIR *dirp);
```

```

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 */
}

/*
   mask values (excerpt from inode(7) man pages)

   S_ISUID    04000    set-user-ID bit (see execve(2))
   S_ISGID    02000    set-group-ID bit (see below)
   S_ISVTX    01000    sticky bit (see below)

   S_IRWXU    00700    owner has read, write, and execute permission
   S_IRUSR    00400    owner has read permission
   S_IWUSR    00200    owner has write permission
   S_IXUSR    00100    owner has execute permission

   S_IRWXG    00070    group has read, write, and execute permission
   S_IRGRP    00040    group has read permission
   S_IWGRP    00020    group has write permission
   S_IXGRP    00010    group has execute permission

   S_IRWXO    00007    others (not in group) have read, write, and
                       execute permission
   S_IROTH    00004    others have read permission
   S_IWOTH    00002    others have write permission
   S_IXOTH    00001    others have execute permission

*/

```

```
void print_owner_perms(const char* pathname)
{
    struct stat sb;
    if (stat(pathname, &sb) < 0) {
        printf("error\n");
        return;
    }

    PRINT_PERM_CHAR(sb.st_mode, S_IRUSR, "r");
    PRINT_PERM_CHAR(sb.st_mode, S_IWUSR, "w");
    PRINT_PERM_CHAR(sb.st_mode, S_IXUSR, "x");
}
```

III Lab 1 and Lab3 (20 points)

4. [20 points] In this problem you will implement a simplified version of the Lab 3 TaskQueue as a monitor, but instead of implementing this monitor as you did in Lab 3, you will use the linked list data structure from lab 1.

The relevant data structures and objects are given below. Notice that the TaskQueue contains a pointer to the head of a linked list. This linked list holds the queue of Tasks; the head is the next item to dequeue, and items are inserted at the tail.

Follow the concurrency commandments. Also, do not wake threads unnecessarily. When considering how the TaskQueue will be invoked, assume the setup of Lab 3.

You do not have to exactly follow C and C++ syntax when making synchronization calls, but try to stick to C and C++ syntax when manipulating the actual state (for example, the linked list pointed-to by head).

Below, fill in where indicated; there are four places to do so.

```

struct Task {
    handler_t handler;
    void* arg;
};

struct list_node {
    Task t;
    struct list_node *next;
};

// You can and should use these helper functions in your answer.
// However, you do not need to worry about their implementation; that
// is, assume they are defined elsewhere.
struct list_node *alloc_node(void);
void free_node(struct list_node *node);

class TaskQueue {
private:
    struct list_node* head;
    // (1) FILL IN: MORE REQUIRED HERE

public:
    TaskQueue();
    ~TaskQueue(); // implemented elsewhere; do not worry about this

    void enqueue(Task task);
    Task dequeue();
};

```

```
TaskQueue::TaskQueue()
{
    head = NULL;
    // (2) FILL THIS IN

}

// Insert the task at the back of the queue (the end of the linked
// list). This may require allocating memory for the linked list node.
void
TaskQueue::enqueue(Task t)
{
    // (3) FILL THIS IN

}

// Remove the Task at the front of the queue (the head of the
// linked list) and return it.
// If the queue is empty, block until a Task is inserted.
// Deallocate any memory you allocated when enqueueing.
Task
TaskQueue::dequeue()
{
    // (4) FILL THIS IN

}
```



```

class TaskQueue {
private:
    struct list_node* head;
    Mutex m;
    Cond c;

    ....

};

TaskQueue::TaskQueue()
{
    head = NULL;
    smutex_init(&m);
    scond_init(&c);
}

void
TaskQueue::enqueue(Task t)
{
    smutex_lock(&m);

    list_node* newp = alloc_node();
    newp->t = t;
    newp->next = NULL;

    if (head == NULL) {
        head = newp;
    } else {
        for (struct list_node* last = head; last->next; last = last->next) ;
        last->next = newp;
    }

    scond_signal(&c, &m);
    smutex_unlock(&m);
}

// Remove the Task at the front of the queue (the head of the
// linked list) and return it.
// If the queue is empty, block until a Task is inserted.
// Deallocate any memory you allocated when enqueueing.
Task
TaskQueue::dequeue()
{
    smutex_lock(&m);

    while (head == NULL)
        wait(&c, &m);

    list_node* retnode = head;

```

```
Task t = retnode->t;
head = head->next;

free_node(retnode);

smutex_unlock(&m);

return t;
}
```

IV Concurrency (23 points)

5. [10 points] Answer the next question with respect to the ChatGPT interaction below. Keep in mind that ChatGPT is very good at making nonsense sound highly plausible, including citing people, results, or phenomena that don't exist.

As context, `cli` and `sti` are instructions that disable (clear) and enable (set) interrupts on x86-64.

Prompt: How can an x86-64 bit operating system that runs multiple applications allow a multi-threaded application to synchronize by turning off interrupts?

ChatGPT: In an x86-64 bit operating system that runs multiple applications, a multi-threaded application can synchronize by turning off interrupts using the `cli` (clear interrupt) instruction, but care must be taken to ensure that this does not impact other applications or the overall system stability.

To safely turn off interrupts in a multi-application environment, the operating system must first ensure that interrupts are disabled only for the specific application that requires synchronization. This can be achieved by using protection mechanisms such as process isolation and privilege levels to prevent one application from accessing or modifying the resources of another application.

Once interrupts are safely disabled for the target application, the multi-threaded application can use synchronization primitives such as spinlocks or mutexes to protect critical sections of code from interruption by other threads or interrupt handlers.

After completing the critical section of code, the multi-threaded application can release the synchronization primitive and re-enable interrupts using the `sti` (set interrupt) instruction. This allows other applications and the operating system to resume normal operation.

It's important to note that turning off interrupts can have serious consequences, such as preventing the operating system from servicing important interrupts, leading to system instability or data corruption. Therefore, it's crucial to use this mechanism judiciously and only in situations where it's absolutely necessary.

Additionally, the operating system must ensure that interrupt disabling and re-enabling are coordinated between applications to avoid potential synchronization issues or data corruption. This may require the use of higher-level synchronization primitives such as semaphores or condition variables to enable more complex synchronization between applications.

Identify the problem in ChatpGPT's first paragraph, and then identify one other major problem.

The answer is essentially nonsense. Applications on such an OS do not have the ability to turn off interrupts. Specifically, a `cli` instruction by an application is an error, because that is a privileged operation. (As we discussed in class, if applications could disable interrupts, they could monopolize the processor.) Also, disabling interrupts for a specific application not only does not make sense (given the preceding point), but also has nothing to do with the typical notion of isolating applications' resources from each other. Finally, the concept of using synchronization primitives after interrupts are disabled doesn't make sense; synchronization primitives are needed in part because interrupts are enabled.

A correct answer to the question would be something like: "The premise is faulty: applications on such a system cannot disable interrupts. Second, applications wishing to synchronize using hardware primitives can either use those primitives directly or get support from the kernel, for example in

the form of mutexes. The implementation, within the kernel, of synchronization primitives such as spinlocks might turn off interrupts.”

Some students said that turning off interrupts isn’t sufficient for synchronization on a multi-CPU system, because other CPUs could access shared memory. This answer did not get full credit because it does not correctly identify ChatGPT’s mistake. That is, even if ChatGPT had qualified its answer by restricting to single CPU systems, it would still be nonsense – again, because applications aren’t permitted to disable interrupts in the first place. Students who answered this way were likely remembering that we said in class and in the notes that disabling interrupts is sufficient for synchronization on a single CPU machine. But that is only for threads running inside the kernel.

6. [8 points] Carol and Dave each have an account in a bank. Dave wants to transfer money to Carol. Assume sequential consistency.

```

300 // assume all the variables are initialized correctly
301 double carol_balance, dave_balance;
302 smutex_t mtx;
303
304 bool
305 transferDave2Carol(double trans) {
306     if (dave_balance > trans) {
307         smutex_lock(&mtx);
308         dave_balance = dave_balance - trans;
309         carol_balance = carol_balance + trans;
310         smutex_unlock(&mtx);
311         return true;
312     }
313     return false;
314 }

```

The implementation of function `transferDave2Carol` is not correct.

(a) Give a problematic interleaving.

(b) State the fix in one sentence.

(a) The outer “if” is not protected by a mutex. So the interleaving can be:

T2: ...

T1: check Dave’s balance

T2: decrease Dave’s balance

T1: acquire mutex

T1: cause balance to be negative...

(b) Place the `lock()` around the if check. Depending on the implementation, there may need to be two instances of `unlock()` in the code.

7. [5 points] Below, assume that the compiler does not reorder the instructions of a program.

Write True for False for each item below:

a If a machine has a single CPU, then the memory model is sequential consistency.

True. Sequential consistency ensures that the executions that appear on a processor are equivalent to those that could have happened on a single CPU.

b The mutex implementation that we covered in class uses a semaphore.

False.

c The mutex implementation that we covered in class uses a spinlock.

True.

d The spinlock implementations that we covered in class use a mutex.

False.

e The spinlock implementations that we covered in class use atomic hardware instructions.

True.

V Scheduling (8 points)

8. [8 points] Bitcoin mining is completely *compute-bound*, meaning that it relies exclusively on the CPU, with no need for any I/O or other resources. Bitcoin mining requires repeatedly computing a mathematical function, specifically a (cryptographic) *hash*, on different *input values* until you find one input value whose corresponding output satisfies some mathematical condition.

Define *mining throughput* as hashes computed per second. Imagine that each process gets an *input set*: that is, a list of new input values to hash. While every individual hash computation requires the same computational time, processes get differently-sized input sets.

Suppose that in 2010 (before the advent of special-purpose Bitcoin mining hardware), you dedicated a general-purpose computer to Bitcoin mining. Assume mining is highly predictable, so that previous CPU bursts perfectly predict future ones. Further, assume that the cost of a context switch (and the computation of a scheduling decision) is negligible. Which of the following scheduling algorithms would lead to the highest mining throughput?

State the BEST choice below, and justify your answer briefly (1-2 sentences):

- A First-come first-served (FCFS / FIFO)
- B STCF (preemptive shortest job first)
- C Round-robin
- D Linux CFS
- E They would all be approximately the same

Justify your answer:

E. Since Bitcoin mining is 100% CPU-bound, any work conserving scheduler (meaning a scheduler that doesn't leave the CPU idle when there is work to do) should get approximately the same mining throughput. Scheduling only improves (the relevant notion of) throughput when there are resources whose utilization can be improved.

Some students answered in terms of *process* throughput, by saying that if we schedule the shorter jobs first, that would mean more jobs completing per unit time. But the question did not ask about process throughput, so credit was not given for that answer.

VI Therac-25 (10 points)

9. [10 points] Which of the following factors contributed to the Therac-25 disasters, as reported in the assigned paper?

Write down ALL that apply:

- A A software issue of deadlock.
- B The absence of hardware interlocks to guard against inconsistent state.
- C A buggy call to `fork()`, creating an extra process.
- D Incomprehensible error messages.
- E The company that made and sold the Therac-25 lied when asked if there had been problems.
- F A software issue of priority inversion.
- G A software issue of overflow/wrap-around.
- H A buggy call to `exec()`, replacing the memory image with a previous version of the software.
- I A beam flattener (the tungsten inverted ice cream cone) with a loose screw, causing radiation to get through.
- J The operator sometimes miscalculated the appropriate dosage.

B, D, E, G

VII Virtual memory (8 points)

10. [8 points] For each of the instructions below, prefixed by the location of the instruction, state how many virtual memory translations happen, and which address or addresses are translated. You should assume that these instructions are executed one after the other.

```
# (a)
0x900    movq $0x4, %rax    # move the number 4 into %rax

# (b)
0x908    addq %rax, 1, %rcx    # add 1 to %rax, put result in %rcx

# (c)
0x910    movq %rcx, 0x400000    # no comment

# (d)
0x918    pushq $0x5          # push the number 5 on the stack
```

(a) 1 translation: code address 0x900.

(b) 1 translation: code address 0x908.

(c) 2 translations: code address 0x910 and 0x400000.

(d) 2 translations: code address 0x918 and the address `%rsp - 8`.

VIII Feedback (2 points)

11. [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
Enjoy Spring Break!!