New York University

CSCI-UA.202-002: Operating Systems (Undergrad): Fall 2025

Midterm Exam

SOLUTIONS

- **Write your name and NetId 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 what is in 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 the exam print-out and blue books; we will not collect it.*
- There are **9** 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 cheat sheet.
- 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/10)	II (xx/20)	III (xx/20)	Total (xx/50)

I Getting Started

1. [2 points] What output is produced when the program below is run? (For this and later questions, assume that the appropriate files have been included to ensure that the program can be compiled.)

```
1  int main(int argc, char* argv[]) {
2    int x = 4;
3    if (fork() == 0) {
4      x *= 2;
5      printf("x = %d\n");
6    } else {
7      wait(NULL);
8      printf("x = %d\n", x);
9    }
10 }
```

```
x = 8
x = 4
```

2. [2 points] Alice executes the following command on the shell:

```
$ cat > f0 < f1
```

Within the `cat` process that is created as a result, which (if any) file descriptor points to the file `f0`? Which if any file descriptor points to the file `f1`?

`stdout (1)` points to `f0` (output is redirected to `f0`); `stdin (0)` points to `f1` (input is read from `f1`).

3. [2 points] In the code below function `f` calls function `g`, which in turn calls `h`. The compiler generates a prolog and epilog for each function, similar to what we saw in class.

Provide the code (this will likely be in assembly) for function `h` such that after you code is executed the register `%rax` contains the address of the base (i.e., the start) of function `f`'s stack frame.

```
1  void f() {
2      ...
3      g();
4      ...
5  }
6
7  void g() {
8      ...
9      h();
10     ...
11 }
12
13 void h() {
```

```
14      // TODO: YOUR CODE HERE.
15  }
```

```
1  movq (%rbp), %rax  // This gets g's base.
2  movq (%rax), %rax  // This gets f's base.
```

4. [2 points] Consider an operating system that uses SCTF scheduling, where three tasks t_0, t_1 and t_2, with completion times (in ms) shown below, are waiting to run:

t_0	2
t_1	4
t_2	3

Once the scheduler starts scheduling (and thus executing) the tasks, how long (in milliseconds) does it take for task t_2 to finish executing.

t_0 runs first and takes 2ms followed by t_2 which requires 3ms to finish. Thus overall t_2 finishes executing in 5ms.

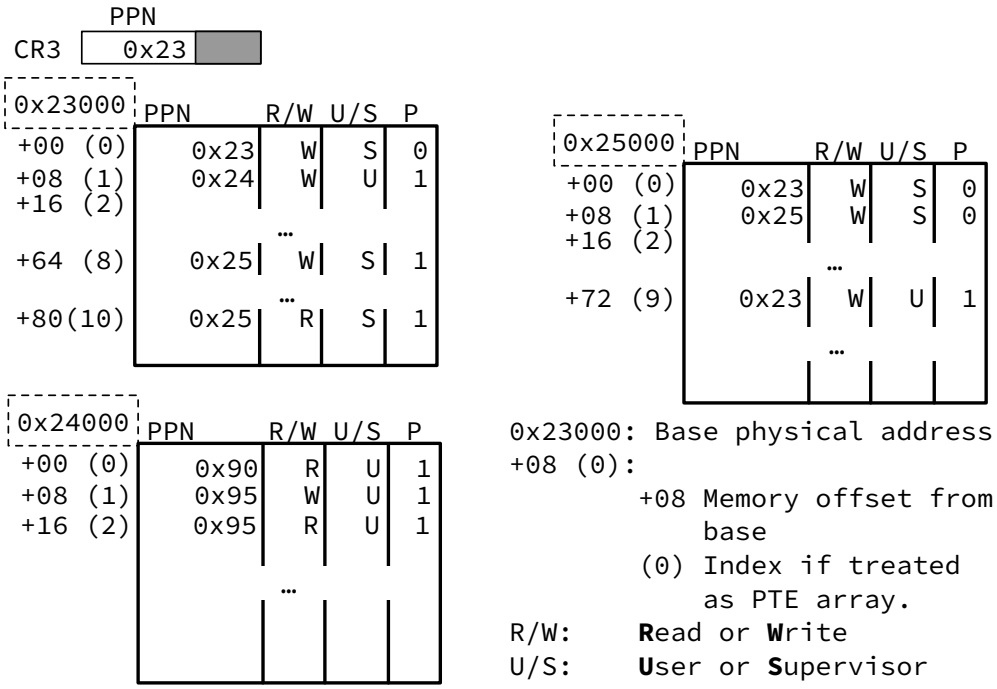
5. [2 points] Is `scond_signal` a blocking call, i.e., can a thread block because of a call to `scond_signal`.

No.

II Virtual Memory

6. [20 points] In this question we are considering a simplified architecture: pages are 4KB in size (similar to the x86-64 architecture we have been using); the machines uses **2-level** page tables; and to simplify things each page when used as a page table contains 256 (2^8) entries and is thus indexed using 8-bits. Similar to x86-64, each page table entry is 8-bytes in size.

Below we show the value of `CR3` for one process, and a few pages that are being used as page tables.



For each page, the dashed box to its left shows the physical address at which it starts (e.g., `0x23000` for the page on the upper left), and a subset of entries in the page. For each entry, we provide both its offset from the beginning of the page (e.g., the second entry in the left most page is at offset `+08`) and its index if we treat the page as an array of PTEs (e.g., the same entry is at index 1). Further, each entry specifies the physical page number (PPN), whether it can be written to (W) or is read only (R), whether it can be accessed from userspace (U) or only from the kernel (S), and whether or not it is present.

Using this information, for each of the memory accesses below, write down *what physical address the MMU would access* (note, address not page number). If the access would cause an exception (that is a **page fault**) write that down instead. That is, each answer should either be a physical address or say that **page fault** would occur.

(i) Read from the userspace to `0x0808120`.
0x95120

(ii) Read from the kernel to `0x0808120`.
0x95120

(iii) Write from the userspace to `0x0016000`.
Page fault: 0x00 in the L1 page table is set so it can only be accessed from the kernel.

(iv) Write from the kernel to `0x0800008`.
0x23008

(v) Read from the userspace to `0x0a09010`.
Page fault: 0x0a in L1 page table only allows access from the kernel.

III Concurrency

7. [10 points] As a part of taking an operating systems class, Liz had to write a program that would output:

```
At zero
At one
At two
```

using three threads. In response, Liz wrote the program below. However, she found that while sometimes the program does produce the desired output, other times it gets stuck, e.g., it might print out `At zero` and then print nothing else and instead hang. Identify and fix the problem in Liz's code. Your answer should identify where changes are required by line number.

```
1  class VisitInOrder {
2  public:
3    VisitInOrder() {
4      where = 0;
5      scond_init(&c);
6      smutex_init(&m);
7    }
8
9    void zero() {
10     smutex_lock(&m);
11     printf("At zero\n");
12     where = 1;
13     scond_signal(&c, &m);
14     smutex_unlock(&m);
15   }
16
17   void one() {
18     smutex_lock(&m);
19     while(where != 1) {
20       scond_wait(&c, &m);
21     }
22     printf("At one\n");
23     where = 2;
24     scond_signal(&c, &m);
25     smutex_unlock(&m);
26   }
27
28   void two() {
29     smutex_lock(&m);
30     while(where != 2) {
31       scond_wait(&c, &m);
32     }
33     printf("At two\n");
34     where = 0;
35     smutex_unlock(&m);
36   }
37
```

```
38 private:
39   uint32_t where;
40            scond_t c;
41   smutex_t m;
42 }
43
44 void zero(VisitInOrder* v) {
45   v.zero();
46   sthread_exit();
47 }
48
49 void one(VisitInOrder* v) {
50   v.one();
51   sthread_exit();
52 }
53
54 void two(VisitInOrder* v) {
55   v.two();
56   sthread_exit();
57 }
58
59 int main(int argc, char* argv[]) {
60   VisitInOrder v;
61   sthread_t t[3];
62   sthread_create(&t[0], zero, &v);
63   sthread_create(&t[1], one, &v);
64   sthread_create(&t[2], two, &v);
65   for (int i = 0; i < 3; i++) {
66     sthread_join(t[i]);
67   }
68 }
```

The problem is with the use of `scond_signal` which only wakes up one waiting thread. But two threads are waiting, and waking up the wrong thread would block progress.

Replacing `scond_signal` on lines 14 and 25 with `scond_broadcast` should fix the problem. (Observe that replacing only the `scond_signal` on line 14 also suffices in this case.)

8. [10 points] The code below is a partially implemented reader-writer lock. In this lock, readers acquire the lock by calling `lock_read` while writers do so by calling `lock_write`. Both release by calling `release`. A correct implementation would meet the following requirements:

(i) **Mutual exclusion.** Only one thread should hold the lock at a time. Other threads trying to acquire the lock should block and wait.

(ii) **Reader priority.** Readers have priority: a writer can only acquire the lock when no readers are waiting for it.

(iii) **Progress.** if no one holds the lock and one or more threads are waiting (i.e., blocked) to acquire it, then one of them must acquire the lock. Similarly, if no thread holds the lock and no thread is waiting for it, then any thread that calls one of the acquire function must acquire the lock.

Fill in the missing `lock_write` function to ensure that the three requirements are met. You should pay particular attention to ensuring that **reader priority**is met. (If you think changes are required to other parts of the code, you can suggest them.)

```
1  class RWPrio {
2    public:
3      RWPrio() {
4        r_waiting = 0;
5        owner = NONE;
6        smutex_init(&m);
7        scond_init(&c);
8      }
9
10     void lock_read() {
11       smutex_lock(&m);
12       r_waiting++;
13       while (owner != NONE) {
14         scond_wait(&c, &m);
15       }
16       owner = READER;
17       r_waiting--;
18       smutex_unlock(&m);
19     }
20
21     void lock_write() {
22                 // TODO: YOUR CODE HERE.
23     }
24
25     void release() {
26       smutex_lock(&m);
27       owner = NONE;
28       scond_broadcast(&c, &m);
29       smutex_unlock(&m);
30     }
31
32   private:
33     smutex_t m;
34     scond_t c;
35     uint32_t r_waiting;
36     const uint32_t NONE = 0;
37     const uint32_t READER = 1;
38     const uint32_t WRITER = 2;
39     uint32_t owner;
40 }
```

```
1  class RWPrio {
2    public:
3      RWPrio() {
4        r_waiting = 0;
5        owner = NONE;
6        smutex_init(&m);
7        scond_init(&c);
```

```
 8      }
 9
10      void lock_read() {
11        smutex_lock(&m);
12        r_waiting++;
13        while (owner != NONE) {
14          scond_wait(&c, &m);
15        }
16        owner = READER;
17        r_waiting--;
18        smutex_unlock(&m);
19      }
20
21      void lock_write() {
22        smutex_lock(&m);
23        while (owner != NONE && r_waiting > 0) {
24          scond_wait(&c, &m);
25        }
26        owner = WRITER;
27        smutex_unlock(&m);
28      }
29
30      void release() {
31        smutex_lock(&m);
32        owner = NONE;
33        scond_broadcast(&c, &m);
34        smutex_unlock(&m);
35      }
36
37    private:
38      smutex_t m;
39      scond_t c;
40      uint32_t r_waiting;
41      const uint32_t NONE = 0;
42      const uint32_t READER = 1;
43      const uint32_t WRITER = 2;
44      uint32_t owner;
45  }
```

9. [0 points] This is to gather feedback.

What topic or topics that we have covered thus far have been the least clear for you?

What topic or topics that we have covered thus far have been the most clear for you?

End of Midterm