# CS-323: OS Mock Midterm (2020 Fall)

November 02, 2020

Last name, first name: SOLUTION

**GENERAL GUIDELINES AND INFORMATION**

1. This is a *closed* book exam. No extra material is allowed. If you have a question, raise your hand and wait for the proctor.
2. You have 60 minutes, and there are 60 points. Use the number of points as *guidance* on how much time to spend on each question.
3. Write your answers directly on the test. Use the space provided. If you need more space your answer is probably too long.
4. Be sure to provide (print) your name. **Do this first so you don't forget! Please write or print legibly.** State all assumptions that you make above those stated as part of a question.
5. Leave your CAMIPRO card on the table so it can be checked.
6. With your signature below you certify that you solved these problems on your own, that you turn in your solution, and that there were no environmental or other factors that disturbed you during this exam or that diminished your performance.

| Question | Points | Max |
|---|---|---|
| 1 | 14 | 14 |
| 2 | 6 | 6 |
| 3 | 15 | 15 |
| 4 | 25 | 25 |
| Total | 60 | 60 |

Name: SCIPER:

# 1 Scheduling (14p)

(a) What are the pros/cons of implementing a multithreading facility entirely in user space compared to with kernel support? (3p)

| Pros: | Cons: |
|---|---|
| • Lightweight<br>• Flexible policies | • No support for pre-emption<br>• Must re-implement all synchronization constructs<br>• Imprecise time measurements |

(b) Why is switching between threads not free? (3.5p)

Switching between threads still requires replacing the thread context, which is comprised of the CPU state (registers, PC, stack pointer, ...).

(c) Why is the idle process needed? (3.5p)

The scheduler runs this idle process whenever no other processes are ready to run. Without the idle process, the scheduler would have to continuously check the list of processes to determine if a process has become ready to run.

(d) What does the following code print if spawn i) creates a new process or ii) creates a new thread? (4p)

```
int a = 0;
__thread int b = 1;

void doit() {
  pid_t pid = spawn();
  if (pid != 0) {
    wait(pid);
    a = 2;
    b = 3;
  } else {
    a = 4;
    b = 5;
  }
  printf("a: %d, b: %d\n", a, b);
  exit();
}
```

```
i) spawning a process

a: ___4___ b: ___5___

a: ___2___ b: ___3___

ii) spawning a thread

a: ___4___ b: ___5___

a: ___2___ b: ___3___
```

Name: SCIPER:

## 2 Scheduling (6p)

Given the following set of processes identified by the tuple of (arrival time, length, task ID). For preemptive schedulers, the time slice is 1s. For round robin, new tasks are inserted at the end of the ready list. Complete the schedule for the different schedulers below.

Tasks: (0, 2, 1), (0, 4, 2), (1, 3, 3), (4, 1, 4), (4, 3, 5), (5, 1, 6), (6, 2, 7).

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FIFO | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 4 | 5 | 5 | 5 | 6 | 7 | 7 | | | | |
| SJF | 1 | 1 | 3 | 3 | 3 | 4 | 6 | 7 | 7 | 5 | 5 | 5 | 2 | 2 | 2 | 2 | | | | |
| RR | 1 | 2 | 1 | 3 | 2 | 3 | 4 | 5 | 2 | 6 | 3 | 7 | 5 | 2 | 7 | 5 | | | | |

Name: SCIPER:

# 3 Virtualizing memory (15p)

Assume your system has 64 KB of contiguous physical memory, a 64 KB virtual address space, 64 B page size. Page table entries are 2 bytes each.

(a) Explain internal and external fragmentation when considering fixed-size segmentation of base/bounds (2p).

External fragmentation is visible to the OS. As it allocates and frees segments for different processes, free space could become dispersed throughout the address space in small segments which might not be able to accommodate an object/process whose size equals the total free space in memory. Internal fragmentation occurs within the segment itself and is visible to the process. With fixed-size segments, allocated space cannot be reduced (if the process needs to release memory), thus resulting in unused free space within a segment.

(b) How many bits are needed to specify (1.5p):

| | |
|---|---|
| Page offset | 6 bits |
| Physical page number | 10 bits |
| Virtual page number | 10 bits |

(c) Split an address into components to issue a two-level lookup: use X for first level page table index, Y for second level page table index, and Z for page offset (1.5p)

```
 0 | Z Z Z Z Z Z Y Y Y Y Y X X X X X | 16 (bit)
```

(d) Assume that virtual addresses (not page numbers) 0x0000 – 0x02FF and 0x1FC0 – 0x2FFF are allocated. How many pages of memory are required for a single, flat page table? How many are needed for a two-level page table? How many pages are required to hold the data? (10p)

| | |
|---|---|
| Flat page table | 32 |
| Two-level page table | 5 |
| Data pages | 77 |

# 4 Concurrent programming (25p)

(a) The following code snippet, from the chapter 31 (Semaphores) of OSTEP, has a bug: find it and explain the problem that may occur - consider the case where we have two consumers and one producer. Two consumers run first and then the producer runs. (15)

```
int buffer;
int count = 0;      // initially, empty
int loops;          // must initialize somewhere...
cond_t  cond;
mutex_t mutex;

void put(int value) {
  assert(count == 0);
  count = 1;
  buffer = value;
}

int get() {
  assert(count == 1);
  count = 0;
  return buffer;
}

void* producer(void* arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);           // p1
    while (count == 1)                    // p2
      pthread_cond_wait(&cond, &mutex);   // p3
    put(i);                               // p4
    pthread_cond_signal(&cond);           // p5
    pthread_mutex_unlock(&mutex);         // p6
  }
}

void* consumer(void* arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);           // c1
    while (count == 0)                    // c2
      pthread_cond_wait(&cond, &mutex);   // c3
    int tmp = get();                      // c4
    pthread_cond_signal(&cond);           // c5
    pthread_mutex_unlock(&mutex);         // c6
    printf("%d\n", tmp);
  }
}
```

Name: SCIPER:

We have two consumers and one producer. Consider the following sequence of steps:

1. Consumer Tc1 runs first, finds count as 0 (c2), then goes to sleep (c3).
2. Consumer Tc2 goes through the same steps: finds count as 0 (c2), then goes to sleep (c3).
3. Then the producer runs, puts a value in the buffer, and wakes one of the consumers (say Tc1).
4. The producer then loops back (releasing and reacquiring the lock along the way). Because the buffer is full, the producer waits on the condition (thus sleeping). Now, one consumer is ready to run (Tc1), and two threads are sleeping on a condition (Tc2 and Tp).
5. The consumer Tc1 then wakes by returning from wait() (c3), re-checks the condition (c2), and finding the buffer full, consumes the value (c4).
6. This consumer then, critically, signals on the condition (c5), waking only one thread that is sleeping (Tc2). Although it should wake the producer, by signaling the condition variable it wakes the consumer Tc2.
7. The consumer Tc2 will wake up and find the buffer empty (c2), and go back to sleep (c3). The producer Tp, which has a value to put into the buffer, is left sleeping. The other consumer thread Tc1 also goes back to sleep.

All three threads are left sleeping, which is a clear bug. Essentially, CVs are unable to differentiate between producers and consumers. Under producer-consumer semantics, a consumer signalling a CV should only wake up a producer thread, not another consumer (and vice versa). Simply using a CV is insufficient to express the correct semantics with more than one consumer (or producer).

(b) Let us continue with the code from the example above. Now, assume that we have two consumer threads (Tc1 and Tc2) and one producer thread (Tp). The figure below shows the actions each thread takes, as well as its scheduler state (Ready, Running, or Sleeping). Please fill in the count value in the table bellow based on the provided thread trace. (10p)

| Tc1 | State | Tc2 | State | Tp | State | Count |
|---|---|---|---|---|---|---|
| c1 | Run | | Ready | | Ready | 0 |
| c2 | Run | | Ready | | Ready | 0 |
| c3 | Sleep | | Ready | | Ready | 0 |
| | Sleep | c1 | Run | | Ready | 0 |
| | Sleep | c2 | Run | | Ready | 0 |
| | Sleep | c3 | Sleep | | Ready | 0 |
| | Sleep | | Sleep | p1 | Run | 0 |
| | Sleep | | Sleep | p2 | Run | 0 |
| | Sleep | | Sleep | p4 | Run | 1 |
| | Ready | | Sleep | p5 | Run | 1 |
| | Ready | | Sleep | p6 | Run | 1 |
| | Ready | | Sleep | p1 | Run | 1 |
| | Ready | | Sleep | p2 | Run | 1 |
| | Ready | | Sleep | p3 | Sleep | 1 |
| c2 | Run | | Sleep | | Sleep | 1 |
| c4 | Run | | Sleep | | Sleep | 0 |
| c5 | Run | | Ready | | Sleep | 0 |
| c6 | Run | | Ready | | Sleep | 0 |
| c1 | Run | | Ready | | Sleep | 0 |
| c2 | Run | | Ready | | Sleep | 0 |
| c3 | Sleep | | Ready | | Sleep | 0 |
| | Sleep | c2 | Run | | Sleep | 0 |
| | Sleep | c3 | Sleep | | Sleep | 0 |