SJSU CS/SE 149 HW3 Spring 2018

REMINDER: Each homework (including programming question) is individual. "Every single byte must come from you." Cut&paste from others is not allowed.

[This assignment does not have programming question.]

1. (20 pts) Consider the following code example for allocating and releasing processes (i.e., tracking number of processes),

```
#define MAX_PROCS 255
int number_of_processes = 0;

/* the implementation of fork() calls this function */
int allocate_process() {
      int new_pid;
      if (number_of_processes == MAX_PROCS)
            return -1;
      else {
            /* allocate process resources and assign the PID to new_pid */
            ++number_of_processes;
            return new_pid;
      }
}

/* the implementation of exit() calls this function */
void release_process() {
      /* release process resources */
      --number_of_processes;
}
```

a. (6 pts) Identify the race condition(s).
b. (7 pts) Assume you have a mutex lock named `mutex` with the operations `acquire()` and `release()`. Please <u>annotate</u> the above code with `acquire()` and `release()` to prevent the race condition(s).
c. (7 pts) Could we replace the integer variable

```
int number_of_processes = 0;
```

with the atomic integer

```
atomic_t number_of_processes = 0;
```

to prevent the race condition(s)? Why or why not? [note: this implies that one replaces any access to the atomic integer with the proper atomic operation atomic_operation, and you can assume all such atomic operations are available.]

2. (30 pts) Consider the following algorithm that provides a solution to the 2-process critical section problem.

```
flag[0] = false; flag[1] = false; /* both are shared variables */
```

P0:

```
while (true) {
  flag[0] = true;
  while (flag[1]) {
    flag[0] = false;
    while (flag[1]) {
      no-op;
    }
    flag[0] = true;
  }
  critical_section
  flag[0] = false;
  remainder_section
}
```

P1:

```
while (true) {
  flag[1] = true;
  while (flag[0]) {
    flag[1] = false;
    while (flag[0]) {
      no-op;
    }
    flag[1] = true;
  }
  critical_section
  flag[1] = false;
  remainder_section
}
```

Considering each statement to be atomic (i.e., no need to dig into low level assembly code). Specify which of the following requirements are satisfied or not by this algorithm. If it is satisfied, explain why (with line numbers). If it is not satisfied, come up with a possible scenario (two-column table, one column per process (i.e., the table format on the slide titled "Race Condition"), interleaved execution with line numbers). No credit if there is no justification, or justification without line numbers.

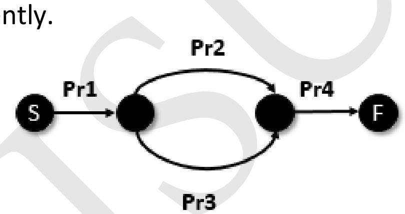
a. (10 pts) Mutual Exclusion
b. (10 pts) Progress
c. (10 pts) Bounded Waiting

3. (30 pts) The following partial code is a *bounded-buffer monitor* in which the buffers are embedded within the monitor (with two condition variables). Assume any condition variable `cond` has two methods: `cond.wait()` and `cond.signal()`. There are multiple producers that invoke `produce()` and there are multiple consumers that invoke `consume()`. Please implement the `produce()` and `consume()` methods in C (no need to have actual .c program). You *cannot* modify existing code and *cannot* have any additional synchronization mechanisms.

```
monitor bounded_buffer {
  int items[MAX_ITEMS]; /* MAX_ITEM is a constant defined elsewhere */
  int numItems = 0; /* # of items in the items array */
  condition full, empty;

  void produce(int v); /* deposit the value v to the items array */
  int consume(); /* remove the last item from items, and return the value */
}
```

4. (20 pts) In an operating system processes can run concurrently. Sometimes we need to impose a specific order in execution of a set of processes. We represent the execution order for a set of processes using a process execution diagram. Consider the following process execution diagram. The diagram indicates that **Pr1** must terminate before **Pr2**, **Pr3** and **Pr4** start execution. It also indicates that **Pr4** should start after **Pr2** and **Pr3** terminate and **Pr2** and **Pr3** can run concurrently.



We can use semaphores in order to enforce the execution order. Semaphores have two operations as explained below.

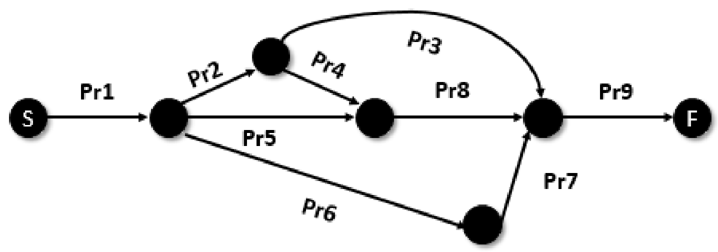
- **P** (or wait) is used to acquire a resource. It waits for semaphore to become positive, then decrements it by 1.
- **V** (or signal) is used to release a resource. It increments the semaphore by 1, waking up the blocked processes, if any.

Assume that the semaphores **s1**, **s2**, and **s3** are created with an initial value of **0** before processes **Pr1**, **Pr2**, **Pr3**, and **Pr4** execute. The following pseudo code uses semaphores to enforce the execution order:

s1=0; s2=0; s3=0;
Pr1: body; V(s1); V(s1);
Pr2: P(s1); body; V(s2);
Pr3: P(s1); body; V(s3);
Pr4: P(s2); P(s3); body;

It is obvious that a different process execution diagram may need different number of semaphores. Note we could consolidate s2 and s3 so that Pr3: …; V(s2) and Pr4: P(s2); P(s2)…. But we choose not to do so. That is, for each process that is followed by an immediate successor, we always create one new semaphore.

Please use pseudo code (which utilizes semaphores) to enforce execution order of the following process execution diagram.



Your pseudo code must specify semaphore initialization followed by the code for each process Pr1, Pr2, …, Pr9, similar to the example. For each process that is followed by an immediate successor, create one new semaphore, i.e., do NOT reuse nor consolidate any semaphore.

Submit the following file:

- CS149_HW3_YourName_L3SID (.pdf, .doc, or .docx), which includes answers to all questions.

The ISA and/or instructor leave feedback to your homework as comments and/or annotated comment. To access annotated comment, click "view feedback" button. For details, see the following URL:

http://guides.instructure.com/m/4212/l/106690-how-do-i-use-the-submission-details-page-for-an-assignment