SJSU CS 149 HW3 SPRING 2020

REMINDER: Each homework is individual. "Every single byte must come from you." Cut&paste from others is not allowed. Keep your answer and source code to yourself only - never post or share them to any site in any way.

[Type your answer. Hand-written answer is not acceptable.]
[Replace YourName and L3SID with your name and last three digit of your student ID, respectively.]

[This assignment does not have programming question.]
1. (30 pts) Consider the following algorithm that provides a solution to the 2-process critical section problem.

```
flag[0] = false; flag[1] = false; /* both variables are shared by P0 and P1*/
P0:                            P1:
0: while (true) {              0: while (true) {
1:   flag[0] = true;           1:   flag[1] = true;
2:   while (flag[1]) {         2:   while (flag[0]) {
3:     flag[0] = false;        3:     flag[1] = false;
4:     while (flag[1]) {       4:     while (flag[0]) {
5:       no-op;                5:       no-op;
6:     }                       6:     }
7:     flag[0] = true;         7:     flag[1] = true;
8:   }                         8:   }
9:   critical_section          9:   critical_section
10:  flag[0] = false;          10:  flag[1] = false;
11:  remainder_section         11:  remainder_section
12: }                          12: }
```

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, justify with a possible scenario by using two-column table (one column per process, interleaved execution with line numbers, see slide titled "Race Condition"). No credit if there is no justification, or justification without two-column table, or two-column without line numbers.

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

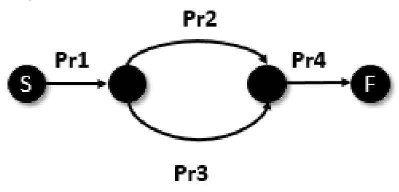
Note: for interleaved execution, a process can execute any number of statements before scheduler context switches to another process.
Hints: (a) mutual exclusion: while one process is already in critical section, can the other process get into critical section?
(b) Progress: No one is in critical section and when both are interested to get into critical section, can at least one eventually get in?
(c) Bounded waiting: process A is in critical section and process B stays outside. Is it possible that A gets out of critical section and then can re-enter critical section while B is still stuck outside of critical section?

2. (20 pts) Consider the following code example for allocating and releasing processes (i.e., tracking number of processes),

```
#define MAX_PROCS 1535
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()`. Annotate the above code with `acquire()` and `release()` to prevent the race condition(s). No credit if no annotated code.
c. (7 pts) Without adding any additional synchronization code (i.e., no mutex), could we replace the integer variable

```
        int number_of_processes = 0;
```
with the atomic integer
```
        atomic_t number_of_processes = 0;
```
(which implies we also replace `++number_of_processes` and `--number_of_processes` with `atomic_add()` and `atomic_sub()`, respectively) to prevent the race condition(s)? Why or why not?

3. (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.



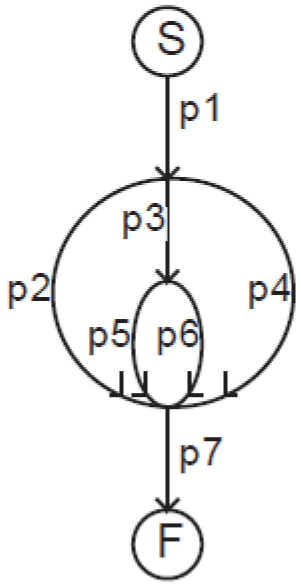
We can use semaphores in order to enforce the execution order. Semaphores have two operations as explained below.

- **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.

Let the semaphores **s1**, **s2**, and **s3** be 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.

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 P1, P2, …, P7, 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.

4. (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()`. Multiple producers and multiple consumers are running in parallel. After an item is produced, a producer invokes `produce()`. Before consuming an item, a consumer invokes `consume()`. The embedded buffer is currently full (since `numItems` has the value `MAX_ITEMS`). 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 constructs.

```
monitor bounded_buffer {
  int items[MAX_ITEMS]; /* MAX_ITEMS is a constant defined elsewhere; not a circular buffer */
  int numItems = MAX_ITEMS; /* # of items in the items array, 0 ≤ numItems ≤ MAX_ITEMS */
  condition full, empty;
  /* both produce() and consume() use numItems as index to access the array */
  void produce(int v); /* deposit the value v to the items array */
  int consume();       /* remove an item from the items array, and return the value */
}
```

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:
https://guides.instructure.com/m/4212/l/352349-how-do-i-view-annotation-feedback-comments-from-my-instructor-directly-in-my-assignment-submission