CSC 4103: Operating Systems
Prof. Aisha Ali-Gombe
Writing Assignment // 1: Process Synchronization
Assign: September 28th, 2023, Due: October 10th, 2023 @ 10:30am Instruction
Lillian Moreau

(1) Prove that Peterson's algorithm (discussed in class) correctly satisfies the two properties of critical-section problem - mutual exclusion and progress.

Peterson's algorithm is a solution to the critical section problem for two processes that share a signal use resource.

1. **Mutual exclusion** is ensured by using flag and turn variables. The flag array is used to indicate if a process wants to enter the critical section. The turn indicates whose turn it is to enter the critical section. A process can then only enter the critical section only when it is its turn, and the flag is set to true. This ensures that only one process can enter the critical section at a time.
2. **Progress** is ensured by giving priority to the other processes if a process has finished executing its critical section. If multiple processes want to enter their critical section, then the turn variable decides who goes next. Also, this ensures that a process is not waiting indefinitely.

(2) Prove that the Bakery algorithm (discussed in class) correctly ensures n-process mutual exclusion, progress and bounded wait

The Bakery algorithm provides a solution for the mutual exclusion problem when having multiple processes by using a numbering system. When a process wants to enter its critical section, it "pulls a ticket" and waits in line.

1. **Mutual Exclusion** is guaranteed by giving each process that wants to enter a critical section a unique number. A process can only enter its critical section if it is the smallest number ensuring that only one process is running. The algorithm prevents processes from selecting the same number by using a choosing array. When a process wants to enter its critical selection and thus selecting a number, it first sets its corresponding element in the choosing array to true. This indicated that the process is currently choosing a number. After selecting a number one greater than the maximum value, it sets its choosing value back to false. This will allow for the next process to select a number. All of this is to prevent race conditions.
2. **Progress** is ensured by allowing the process with the smallest number to enter its critical section. Once a process is done with its critical section, it resets its number to 0. This allows other processes with non-zero numbers to get a chance to enter their critical sections.
3. **Bounded Waiting** is ensured by assigning increasing numbers to processes over time. A process that has been waiting for a long time will eventually have the smallest number and get to enter its critical section.

(3) Write (in pseudocode) a **STRONG WRITERS** solution to the readers-writers problem using semaphores. You must indicate if waiting readers must wait until ALL waiting writers have proceeded (STRONG STRONG writers) or not (just STRONG writers).

In the pseudocode below, if a writer is waiting to write, new readers will have to wait, but current readers are allowed to finish reading.

```
//initialize semaphores
semaphore mutex = 1
semaphore write = 1

//initialize reader count
int readers = 0

//reader process
def reader():
     while true:
          wait(mutex) //request access to critical sections
          readers += 1//increment number of readers
          if readers == 1: //if first reader, block writes
               wait(write)
          signal(mutex) //release mutex lock
          << critical section: read data >>
          wait(mutex) //request access to critical section
          readers -= 1 //decrement number of readers
          if readers == 0: //if no more readers, unblock write
               signal(write)
          signal(mutex) //release mutex lock

//writer process
def writer():
     while true:
          wait(write) //request access to critical section
          << critical section: write data >>
          signal(write) //release write lock
```

(4) The "solution" to the two-process mutual exclusion problem below was actually published in the January 1966 issue of *Communications of the ACM*. It doesn't work! Your task: Provide a detailed counter example that illustrates the problem. As in lecture, assume that simple assignment statements (involving no computation) like `turn = 0` are atomic.

```
Shared Data:
     blocked: array[0..1] of Boolean;   turn:
0..1;
          blocked[0] = blocked[1] = false;
          turn = 0;
Local Data:
          ID: 0..1;   /* (identifies the process;
                          set to 0 for one process,
                          1 for the other) */
Code for each of the two processes:
          while (1) {
               blocked[ID] = true;
               while (turn <> ID) {
                     while (blocked[1 - ID]);
                     turn = ID;
               }

               << critical section >>

               blocked[ID] = false;

               << normal work >>
          }
```

The "solution" above does not work because it leads to deadlock. Here is an example of why it fails: There are two processes P0 and P1.

1. P0 enters the while loop and sets `blocked[0] = true`
2. It becomes interrupted by P1 before it checks `turn <> ID`
3. P1 enters the while loop, sets `blocked[1] = true`, and since `turn = 0`, it enters the inner while loop `blocked[1 - ID]`
4. Since `blocked[1 - ID]` evaluates to `blocked[0] = true`, it is stuck in an infinite loop
5. At this point, both processes are stuck in their respective while loops.

P0 is waiting for its turn to set `turn = ID`, and P1 is waiting for `blocked[0] = false`. But, neither of these conditions can be met because both processes are waiting for each other to do something. This is a deadlock situation because two or more processes are each waiting for the other to release a resource.