CSE 430 Homework 3

Ryan Dougherty

Question 5.3

What is the meaning of the term **busy waiting**? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer. When a process waits for a condition to be satised in a loop without giving up the processor, that is called busy waiting.

Another kind of waiting in an OS is: a process could wait by giving up the processor, and block on a condition and wait to be awakened at some future time.

Busy waiting can be avoided altogether, but incurs the overhead associated with putting a process to sleep and having to wake it up when the appropriate program state is reached.

Question 5.5

Show that, if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated. Assume wait() and signal() are not executed atomically. The function of wait() is to decrement a semaphore's value atomically (in theory). Consider the case when 2 wait operations execute on the same semaphore with a positive value. Since we assumed wait() is not executed atomically, the 2 operations may decrement the same value before an update of the value, which violates mutual exclusion. The answer for signal() has a similar construction.

Question 5.9

The first known correct software solution to the critical-section problem for n processes with a lower bound on waiting n-1 turns was presented by Eisenberg and McGuire...Prove that the algorithm satisfies all three requirements for the critical-section problem. We need to show that the following are true: (1) mutual exclusion is preserved, (2) the progress requirement is satisfied, and (3) the bounded-waiting requirement is met. Assume that P_i means processor i. Let CS mean "critical section".

To prove (1), see that P_i enters its CS iff flag[j] is not equal to in_cs for all j different than i. Since only P_i can set flag[i] to be equal to in_cs, and since P_i inspects flag[j] only

while flag[i] is equal to in_cs, we can see that mutual exclusion is preserved.

To prove (2), see that only when a process enters its CS and when it leaves its CS does the value of turn can be modified. Therefore, if no process executes or leaves its CS, the value of turn remains unchanged. The first contending process in the cyclic ordering (turn, turn+1, ..., n-1, 0, ..., turn-1) will enter the CS. Therefore, the progress requirement is satisfied.

To prove (3), see that when a process leaves the CS, it must designate as its unique successor the first contending process in the cyclic ordering (turn+1, ..., n-1, 0, ..., turn-1, turn), ensuring that any process wanting to enter its CS will do so within n-1 turns. Therefore, the bounded-waiting requirement is met.

All three conditions are met, so our proof is done.

Question 5.11

Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems. Interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems because disabling interrupts only prevents other processes from executing on the processor in which interrupts were disabled. There are no limitations on what processes could be executing on other processors. Therefore, the process that disables interrupts cannot guarantee mutually exclusive access to program state.

Question 5.14

Describe how the compare_and_swap() instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.

```
int lock; // initially set to 0
int waiting[n]; // all initially set to 0
// ...
do {
        waiting[i] = 1;
        int key = 1;
        while (waiting[i] == 1 && key == 1)
                key = compare\_and\_swap(&lock, 0, 1);
        waiting[i] = 0;
        /* critical section */
        j = (i+1) % n;
        while ((j != i) && waiting[j] == 0)
                j = (j+1) % n;
        if (j == i)
```

```
                    lock = 0;
          else
                    waiting[j] = 0;
          /* remainder section */
} while (true);
```

Note: We can only enter the CS if waiting[i] or key is == to 0. The value of key can only be this if compare_and_swap() is executed.

Let P_i be the first process to execute compare_and_swap(). It will give a result of key == 0 after the method call finishes, and therefore exits the while loop. All other processes must wait. The value of waiting[i] can only be 0 if another process leaves its CS. Since only waiting[i] is 0 for one i, the mutual-exclusion requirement is met.

For progress, we have the same argument as for mutual exclusion. A process exiting the CS either sets lock to 0 or waiting[j] to 0. Therefore, the progress requirement is met.

For bounded-waiting, when a process leaves its CS, the waiting array does a cyclic ordering from i+1, i+2, ..., n-1, 0, ..., i-1. It says that the first process in the ordering that is in the entry section (waiting[j] == 1) as the next to enter the CS. Therefore, any processing wanting to enter the CS will have to wait n-1 turns, and the bounded-waiting requirement is met.

Question 5.28

Discuss the tradeoff between fairness and throughput of operations in the readers-writers problem. Propose a method for solving the readers-writers problem without causing starvation. Throughput in the readers-writers problem is increased by favoring multiple readers as opposed to allowing a single writer to exclusively access the shared values. However, favoring readers could result in starvation for writers.

The starvation in the readers-writers problem could be avoided by keeping timestamps associated with waiting processes. When a writer is finished with its task, it would wake up the process that has been waiting for the longest duration. When a reader arrives and notices that another reader is accessing the database, then it would enter the CS only if there are no waiting writers. These restrictions would guarantee fairness.