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 relinquishing the processor, and block on a condition and wait to be awakened at some appropriate time in the future.

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, 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. To prove that the correctness of the algorithm, we need to show that all of the following 3 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.

To prove (1), observe that for all i, P_i enters its critical section only if flag[j] != in_cs for all j != i. Since only P_i can set flag[i] = in_cs, and since P_i inspects flag[j] only while flag[i] =

in_cs, we can see that mutual exclusion is preserved.

To prove (2), observe that the value of turn can be modified only when a process enters its critical section, and when it leaves its critical section. Therefore, if no process executes or leaves its critical section, 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 critical section. Therefore, the progress requirement is satisfied.

To prove (3), observe that when a process leaves the critical section, 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 critical section will do so within n-1 turns. Therefore, the bounded-waiting requirement is met.

Since all three conditions are met, the algorithm correctly solves the critical-section problem.

Question 5.11

Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems. Interrupts are not sufficient in multiprocessor systems since 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, and 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 = 0;
        while (waiting[i] == 1 && key == 1)
                key = compare\_and\_swap(&lock, &key, 0);
        waiting[i] = 0;
        /* critical section */
        j = (i+1) % n;
        while ((j != i) && !waiting[j])
                j = (j+1) % n;
        if (j == i)
                lock = 0;
        else
                waiting[j] = 0;
        /* remainder section */
} while (true);
```

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. On the other hand, 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 critical section only if there are no waiting writers. These restrictions would guarantee fairness.