CISC 3595 Operating Systems

Fall 2022

Assignment #4

Out 11/04; back 11/18

Q1. In the reading, it is mentioned that disabling interrupts frequently can affect the system's clock. Explain why this can occur? (5pts)

The system clock is most often implemented as a programmable interval timer that periodically interrupts the CPU. In response, the timer interrupt service routine increments the clock on tick before doing some housekeeping. Disabling interrupts would to often prevent the timer interrupt from happening and would cause the clock to miss updates.

How can such effects be minimized? (5 pts)

Disabling interrupts less frequently, and for shorter durations can avoid these consequences.

Under which conditions should interrupts be ignored? (5 pts)

Interrupts should be ignored when servicing another interrupt or when interrupts are disabled, or during atomic operations. In some cases, lock arbitration is uninterruptible. ***(if two out of 3, take -1)***

Q2. What is the meaning of the term *busy wait*? (5 pts) What other kinds of waiting is there in an operating system? (5 pts)

Waiting for a condition in a while loop, consuming CPU cycles while waiting for the condition to be satisfied.

A blocking call that context switches the process out onto a waiting state queue and places the process in the ready queue once the wait condition is satisfied.

Explain how busy waiting can be avoided altogether? (5 pts) Under what conditions is it OK to use a busy wait? (5 pts)

Busy wait can be avoided using a system call that blocks the thread until a condition is met such as a mutex or semaphore. In a single processor system, it possible to protect
critical regions by simply turning off (disabling) interrupts, and this is more efficient than spinlocks, cutting down on context switching. (5 pts)

In a multiprocessor system, it's difficult to coordinate all the processors turning off interrupts at the same time, so spinlocks are a reasonable alternative for small critical sections (5 pts)

Q3. Show that, if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated. (5 pts)

wait() is defined as:

```
wait(S) {
  while (S <= 0) // if 0 first waiter, <0 means queue
      ; // busy wait
  S--;
}
```

If wait() is not atomic, then more than 1 thread/process can see S becoming greater than 0, none will have to wait in the loop, all will get access to the critical section which defies mutual exclusion. Even a race condition in S-- can cause S to not be decremented properly for every access, leaving S one more than it should be. (-1 for missing the second part)

```
signal() is defined as:
signal(S) {
   S++;
}
```

The same kind of race condition on S++ can happen where more than 1 thread/process can increment S at the same time resulting in S being less than it should be, This could causing starvation of subsequent threads/processes.

Illustrate how a binary semaphore can be used to implement mutual exclusion among n threads. Assume that the threads are running the same code with the same critical section. (5 pts)

```
do {
      wait(mutex);
          /* critical section */
      signal(mutex);
         /* remainder section */
} while (true);
This would ensure mutual exclusion
```

No, deadlock could not happen because there is only one resource under contention, and it is accessed by one thread at a time. (take -2)

Yes, starvation can happen because a thread may never be removed from the semaphore queue in which it is suspended, usually because of the scheduler. A round-robin schedular would not result in starvation. (take -2 or 3 depending on explanation)

Yes, priority inversion can occur because a lower-priority thread can hold a lock needed by a higher-priority thread, but the low priority thread would never run. (take -2 or 3)

Q4. Explain how the synchronization works using semaphores in the Readers-Writers Problem 1 where Readers are prioritized (15 pts) **(take -4 to -8 unless not answered)**

In Readers-Writers Problem 1, Readers are prioritized. A mutex protects entry into the critical section. Only one writer can be in the critical section at a time, but multiple readers are allowed.

When a reader enters the critical section, the reader-writer Semaphore is locked. A counter keeps track of the number of subsequent readers in the critical section. Writers are blocked, possibly indefinitely (starvation), while readers are in the critical section. A blocked writer will continue to be blocked until there are no readers in the critical section. As readers leave the critical section, the counter is decremented until it reaches 0, at that point the reader-writer semaphore is released and the writer can proceed. The writer then locks the reader-writer semaphore given the critical section over exclusively to the writer. If readers come, then subsequent writers will be locked out and if a writer comes, before the next reader, the writer will get the lock. Readers will always get the next lock before a writer.

Explain how the synchronization changes in the Readers-Writers Problem 2 where Writers are prioritized. (15 pts)) **(take -4 to -8 unless not answered)**

In Readers-Writers Problem 2, Writers are prioritized. A mutex protects entry into the critical section. Only one writer can be in the critical section at a time, but multiple readers are allowed.

A read semaphore controls whether readers can enter the critical section. When a reader wants to enter the critical section, if there are no writers waiting, the reader can enter the critical section.

If a writer is waiting, all subsequent readers block on the read semaphore. As readers finish, the count is decremented. When the count reaches 0, the read-write semaphore is released allowing the writer to enter the critical region. When the writer finishes, if there are writers and readers waiting, the writer will be granter the read-write semaphore and the readers will continue to be blocked, possibly indefinitely (starvation). There are two controlling semaphores: reader-writer and read as well as a mutex protecting the critical regions.

Q5. Dining Philosophers problem in class uses a monitor. Implement the same solution using semaphores (20 pts) *Hint: Use a class*

```
class DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self[5]; // struct condition { semaphore x_sem; int
x_count; } or two parallel arrays, take –1
   semaphore mutex;
```

```
   semaphore next;
   int next_count = 0;

   public void pickup (int i) {
      wait(mutex);
      state[i] = HUNGRY;
      test(i); // Tests if chopsticks are available
      // If philosopher not eating, wait for chopsticks.
      if (state[i] != EATING) wait(self[i]); // leave it if
self[i].wait()
      if (next_count > 0)
         signal(next);   // Release a thread waiting in monitor
      else
         signal(mutex);  // Unlock monitor since no one waiting.
   }

   public void putdown (int i) {
       wait(mutex);
      state[i] = THINKING;
       // test left and right neighbors want to eat.
      test((i + 4) % 5);
      test((i + 1) % 5);
      if (next_count > 0)
         signal(next);   // Release a thread waiting in monitor.
      else
         signal(mutex);  // Unlock monitor since no one waiting
   }

   // Important that this method can only be called from
pickup/putdown.
   private void test (int i) { // if not private, take -1
      // both chopsticks must be available
      if ((state[(i + 4) % 5] != EATING) &&
          (state[i] == HUNGRY) &&
          (state[(i + 1) % 5] != EATING) ) {
           state[i] = EATING ; // Gets chopsticks
           signal(self[i]); // leave it if self[i].signal()
       }
   }

   private signal(condition need) // take -2 if not implemented
   {
      if (need.x_count > 0) { // if any condition waiting
          next_count++;       // keep track of next queue size
          signal(need.x_sem); // release condition thread to go
          wait(next);         // stop this thread, wait next
          next_count--;       // released from next queue, 1 less
      }
   }

   private wait(condition need) // take -2 if not implemented
   {
```

```
      need.x_count++;      // May end up waiting
      if (next_count > 0)  // Something waiting?
         signal(next);     // signal next P to go
      else                 // nothing waiting.
         signal(mutex);    // open monitor
      wait(need.x_sem);         // wait for x_sem
      need.x_count--;
   }

   DiniingPhilosophers() {
       for (int i = 0; i < 5; i++) {
          state[i] = THINKING;
          init(self[i].x_sem);   // take -1 if left out
          self[i].x_count = 0;
       }
       init(mutex, 0, 1);
       init(next, 0, 0);  // Counting semaphore

   }

   class condition {
      int x_count = 0;
      semaphore x_sem;

      WAIT(){
         x_count++;          // waiting on condition

         if (next_count > 0) // if any waiting inside the monitor
            signal(next);    // let the waiter use the monitor
         else                // none waiting in monitor
            signal(mutex);   // open it up

         wait(x_sem);        // Now wait on the x.condition…
         x_count--;          // condition happened.
      }

      SIGNAL() {
         if (x_count > 0) { // if any waiting on x.condition
           next_count++;    // increment the count waiting on the
monitor
           signal(x_sem);   // signal the condition, which gives away
monitor
           wait(next);      // wait inside the monitor
           next_count--;    // no longer waiting.
         }
      }

      condition() }
         sem_init(x_sem, 0, 1);
      }
   }
}
```

Accepted the following:

```
monitor DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   semaphore self [5];
   semaphore mutex;

   void pickup (int i) {
             wait(mutex);
              state[i] = HUNGRY;
              test(i); // Tests if chopsticks are available
              if (state[i] != EATING)
                  {signal(mutex); wait(self[i]);}
             else signal(mutex);
   }

   void putdown (int i) {
              wait(mutex);
              state[i] = THINKING;
                     // test left and right neighbors
               test((i + 4) % 5);
               test((i + 1) % 5);
               signal(mutex);
   }

   private void test (int i) {
               // both chopsticks must be available
               if ((state[(i + 4) % 5] != EATING) &&
               (state[i] == HUNGRY) &&
               (state[(i + 1) % 5] != EATING) ) {
                    state[i] = EATING ; // Gets chopsticks
                   signal(self[i]);
               }
   }

   initialization_code() {
        for (int i = 0; i < 5; i++) {
          state[i] = THINKING;
         sem_init(self[i], 0, 0);
        }
        sem_init(mutex, 0, 1);
   }
}
```