Homework 3 solutions

1. **7.1. Busy Waiting**

   Busy waiting is continuously executing (using the CPU) while waiting for something to happen. This is typically implemented by spinning; e.g. testing a variable in a tight loop until it changes.

   The alternative to busy waiting is *blocking*, where the waiting process is suspended an other processes can execute while the process is waiting.

2. **7.2. Spinlocks**

   Spinlocks are not appropriate for uniprocessor systems because by definition only one process is executing at a time. As a result, no other process can ever change release the lock – the scheduler has to intervene and switch out the spinning process first. As a result, the spinning process might as well suspend itself instead of spinning. Or, locking should disable interrupts to prevent the scheduler from running, so that the process is guaranteed that it can execute the entire critical section.

   On a 2 CPU multiprocessor, for example, a process scheduled on processor 1 can release the lock while a process on processor 2 is spinning on it. No interrupt is needed. Thus, spinlocks are not necessarily wasteful on a multiprocessor.

Synchronization Problems

Notes:

- Please use real binary semaphores, counting semaphores, or monitors. If you invent your own synchronization method, there is a 99% chance you will get it wrong. Plus it makes it very hard to grade - we look very closely and are more likely to find problems. If you must do this, provide an implementation of your construct, because otherwise we don't know if:
  - It is possible to implement
  - How it works
- If you use semaphores, you must specify initial values for your semaphores (mandatory).
- If you have a shared state, you should say what mutex protects the shared state (not mandatory)
- With semaphores, if you wake up on one semaphore and then call signal() to wake up the next waiter you might wake yourself up - semaphores have history, so your next wait might be the one that awakes, because the other waiters might have been context switchted out before they call wait().
- With semaphores, you shouldn't wait while holding a mutex, unless the routine signaling you does not need the mutex to wake you up.
- With monitors, you don't need to grab a mutex (or, shudder, a condition variable) to access shared state. The monitor, by definition, ensures mutual exclusion.
- With monitors, wait() doesn't take any parameters - it just waits until somebody then calls signal().
- When writing synchronization code, you should try to minimize the number of context switches and the number of times a process is awoken when it doesn't need to be. Ideally, in a while() loop, a process should only be woken once.

3. **7.8. Sleeping-barber problem.**

   Barbershop Requirements:

   - **Barber sleeps if no customers waiting**
   - **Customers leave if no chairs for waiting**
   - **Waiting customers can't leave until haircut done.**

   Solution: we use semaphores

   - **Mutex = 1**
   - **counting: barber_sleeping, customer_queue, cut_done**

Barber Code

```
wait(mutex)
if (customers_waiting == 0) {
  signal(mutex);
  wait(barber_sleeping);
  wait(mutex);
}
customers_waiting--;
signal(mutex);
signal(customer_queue);
do_cut_hair();
signal(cut_done);
```

Customer Code

```
wait(mutex);
if (customers_waiting == n) {
   signal(mutex);
   return;
}
customers_waiting++;
if (customers_waiting == 1) {
  signal(barber_sleeping);
}
signal(mutex);
wait(customer_queue);
get_hair_cut();
wait(cut_done);
```

As a monitor

```
monitor barbershop {
  int num_waiting;
  condition get_cut;
  condition barber_asleep;
  condition in_chair;
  condition cut_done;

  Barber routine
  barber() {
    while (1);
      while (num_waiting == 0) {
        barber_asleep.wait();
      }
      customer_waiting.signal();
      in_chair.wait();
      give_hait_cut();
      cut_done.signal();
    }
  Customer routine
  customer () {
    if (num_waiting == n) {
      return;
    }
    if (num_waiting == 0) {
      barber_asleep.signal();
    }
    customer_waiting.wait();
    in_char.signal();
    get_hair_cut();
    cut_done.wait();
  }
```

7. **7.9. The Cigarette-Smokers Problem.**

   Semaphores are a convenient mechanism for implementing this, because they remember what elements are on the table.

```
Sempahore TobaccoAndPaper = 0;
Sempahore PaperAndMatches = 0;
Semaphore MatchesAndTobacco = 0;
Sempaphore DoneSmoking = 1;

void agent()
{
      wait(DoneSmoking);
      int r = rand() % 3;


      //
      // Signal which ever combination was
      // chosen.
      //



      switch( r ) {
            case 0: signal(TobaccoAndPaper);
            break;
            case 1: signal(PaperAndMatches);
            break;
            case 2: signal(MatchesAndTobacco);
            break;
      }
}

void Smoker_A()
{
      while(true) {

            //
            // Wait for our two ingredients
            //

            wait(TobaccoAndPaper);
            smoke();

            //
            // Signal that we're done smoking
            // so the next agent can put down
            // ingredients.
            //

            signal(DoneSmoking);
      }
}
```

   Smoker b and c are similar.

8. **7.15. File-synchronization**

   From the book you should know how to use a *conditional-wait* construct (priority-based signaling), so we'll use it here. We'll assume for simplicity that no process has an id greater than or equal to n (or we'd just print an error message).

```
 type file = monitor

   var space_available: binary condition

       total: integer


   procedure entry file_open(id)

   begin

     while (total + id >= n)

       space_available.wait(id)

     total = total + id;

     if (total < n - 1)

       space_available.signal();

   end


   procedure entry file_close(id)

   begin

     total = total - id;

     space_available.signal();

   end
```

   What happens here? As long as the incoming processes find adequate space available (i.e. sums less than n), they will claim that space and open the file. Note that since we're using a binary condition, we can signal() at will even if nothing is waiting. If a process finds inadequate space, it will block. When a process is done, it wakes up the process with the smallest id (the most likely to fit in the available space). This process, upon waking, then signals the next process if space is still available, but only after successfully claiming its own space. At some point (quite possibly with the first process), the space made available may not be enough for the waking process, and so a process will be woken up prematurely. This is what the loop is for; it will immediately block again.

6. **8.4 b**
   install traffic lights :)
7. **8.8**
   The system can always make progress in all possible allocation scenarios (use pigeonhole principle to show this).