CS 350: Solution Key to Assignment I

Problem 1: (5 minutes)

Give brief (one word, or sentence) answers to the following

- **Semaphores can be built using locks (Yes/No)**

  Yes, semaphores can be built (implemented) using locks. We went over the implementation in class.

- **Monitors can be built using condition locks and semaphores (Yes/No)?** Yes, monitors implementation wise are just a single lock to enforce mutual exclusion on the monitor, and then condition locks to allow waiting on conditions within the monitor. The outside lock can be trivially implemented using a semaphore initialized to 1.

- **Mode switch is faster than context switch (Yes/No)?**

  Yes, mode switch incurrs the overhead of switching into the OS code (procedure call + kernel stack initialization + memory mapping some kernel pages), while a context switch has all that + the overhead of saving the process state, and restoring the other process state.

- **Enforcing mutual exclusion on a shared variable among multiple readers and writers is (incorrect/correct)?**

  Correct here refers to no inconsistent results happening. So, yes this will be correct, but...

- **Continuing the previous question, this approach is** (liberal/conservative)?

  ...it is conservative, since we are enforcing mutual exclusion even among multiple readers. A better implementation would allow multiple readers to go on at the same time.

Problem 2: (16 minutes) **Explain all of the following *potentially wrong* statements.**

1. **Test and set or similar hardware instruction is needed to implement semaphores**

   While its possible to implement semaphores using test and set, it is also possible to implement them using software locks; all we need is a mutual exclusion mechanism. If you answered that they are needed because software locks often dont work due to compiler/architecture issues you should get full credit.

2. **A system call is needed to create a user level thread**

   False, a user level thread is implemented completely in user space, and the OS does not get involved in any of their operations.

3. **The medium term scheduler should swap out/in kernel level threads instead of processes**

   The key here is that the medium term scheduler's job is to control load/free up resources when the machine is overloaded. Since processes, not threads, are the unit of resource allocation, the medium term scheduler swaps out processes, not threads. Of course, all the threads belonging to a processes that is swapped out will also be suspended.

4. **A concurrent program where two threads access a shared variable will either always work correctly (correctly synchronized) or never work correctly (incorrectly synchronized)**

   False. Depending on the scheduler, the order of execution may lead to sometime correct and sometimes incorrect results.

Problem 3: (15 minutes) **Drawbridges are bridges that open up to let boats/ships pass. Multiple ships can pass concurrently when the bridge is open, and multiple cars can pass concurrently when it is closed.**

(a) How is this problem similar/different from readers/writers?

Its similar to readers writers in that it allows multiple threads of the same time to enter the mutual exclusion region (the bridge) together. It is different in that both multiple readers (cars) and multiple writers (ships) can go on at the same time.

(b) Show psuedocode implementation for this problem (through a simple modification of readers/writers). What problems, if any, does it suffer from?

You could really implement this as both sides very similar to reader side.

```
//Semaphore mutex_cars initialized to 1
//Semaphore mutex_ships initialized to 1
//Semaphore get_bridge initialized to 1
cars                    ships
...                      ...
wait(mutex_cars);        wait(mutex_ships);
cars++;                  ships++;
if(cars == 1)            if(ships==1)
   wait(get_bridge);        wait(get_bridge);
signal(mutex_cars);      signal(mutex_ships);

cross bridge;            cross bridge;

wait(mutex_cars);        wait(mutex_ships);
cars--;                  ships--;
if(cars == 0)            if(ships==0)
   signal(get_bridge);      signal(get_bridge);
signal(mutex_cars);      signal(mutex_ships);
```

Problem 4: (20 minutes) **At the coffee shop in the library, people queue up to buy coffee from the register. After placing their order, they go and wait for their coffee. There are three workers, in addition to the person on the cash register, and each works on one order at a time.**

(a) Write psuedocode to simulate the coffee shop operation

There are three types of actors here: coffee junkies, register, and workers. You would initialize any number of junky threads, one register thread, and three worker threads. I think you may be able to abstract away the register since its function is so simple, but for now lets keep it.

Junky basic algorithm: wait for cash register, place order, wait for order, get coffee register: wait for customer, get order, pass order to worker, repeat

worker: wait for order from register, process order, pass it to customer, repeat

Register to worker synchronization is basically producer consumer. Its ok to assume infinite buffer (that is the register will just process orders indefinitely, regardless of the worker progress. Junky queues up for coffee (waits on cash register sempahore initialized to 1), then waits for the order semaphore signalled by the workers (here we are not making sure that they get their order, just that they will get an order; we'll leave that to the second part).

```
Junky                Register           Workers
-----                --------           -------
Semaphore register initialized to 1;
Semaphore order_ready initialized to 0;
Sempahore customer initialized to 0;
Semaphore orders initialized to 0;

wait(register);      while(1){          while(1){
signal(customer);    wait(customer);    wait(orders);
place order/pay      get_order;         process_order;
signal(register)     signal(orders);    signal(order_ready);
wait(order_ready);   }                  }
get coffee
```

(b) Since there are many types and flavors of coffee, its important that everyone picks up their correct order. Assume now that coffees may not be ready in the order that they were placed – for example, regular coffee is ready much faster than a double shot latte. Explain (in words, but be specific) how the implementation could be done now.

Essentially, you want to associate orders with customers. Since the customers have mutual exclusion at the register via the register semaphore, we can in there associate a ticket number with them that gets increased with every order. Similarly, the workers can increment the order number (need to add a lock on that since currently the workers dont have a mutual exclusion portion of their code). The ready orders can be tagged into a linked list (with mutual exclusion). The waiting customers could wait on a condition lock with condition their order being in the linked list. Every time an order is done, we broadcast to the waiting customers to check if their order is ready. All accesses to the linked list of orders have to have mutual exclusion.

Problem 5: (8 minutes) **For each of the following alternatives, discuss one advantage and one disadvantage.**

(a) pipes (or message passing Inter Process Communication) vs. Shared memory IPC

Pipes simplify programming because thread interaction is now explicit and there is no shared memory to worry about coordination. However, they are more expensive to use than shared memory because they require a system call and data copies.

(b) Threads vs. Processes

Both can give us concurrency. Processes get resource separation from other processes unless they choose to communicate via IPC–this is great if your processes are not tightly integrated (for example, doing two parts of the same application and requiring frequent communication). Threads are much more efficient to create and schedule, and can communicate more easily because they share the same resource space (they reside in the same process).

(c) Kernel Level threads vs. User level threads

Kernel level threads provide true concurrency; that is, if one thread blocks, the others continue running. User level threads reside completely in user space (the OS is not needed to create or schedule them) and are therefore much more efficient to create and schedule.

Problem 6 (problem 6.11 from book): (20 minutes) **The sleeping barber problem. A barbershop consists of a waiting room with n chairs and a barber room with one barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write psuedo-code to coordinate the barber (who is a thread) and the customers (who are each a thread) using semaphores or condition locks.**

The barber just waits on customers, then goes to sleep.

Customers arrive: if barber is asleep/no other customers, wake them. If they are not asleep and there is a seat, wait. Otherwise leave.

Since the customers have different behavior depending on how many other customers are there (e.g., leave rather than just wait), we really have to maintain the customer count.

```
//Lock mutex
//Semaphore sleeping initialized to 0
//Semaphore haircut_done initialized to 0
//Condition chairs

barber                       clients
while(1) {                   mutex.Lock();
wait(sleeping);              if (client_count == CHAIRS) {
cut hair                         mutex.Unlock();
signal(haircut_done);            return; //leave
}                                }
                             client_count++;
                             if(client_count > 1)
                                 chairs.wait(mutex);
                             signal(sleeping);
                             mutex.Unlock();
                             getting_haircut;
                                wait(haircut_done);
                             mutex.Lock();
                                client_count--;
                             chairs.signal(mutex);
                             mutex.Unlock();
```

Problem 7 (problem 6.19 and 6.20 from book): (15 minutes) **A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: the sum of all unique numbers associated with all the processes currently accessing the file must be less than n.**

(a) Write a monitor to coordinate access to the file.

Place in the monitor the current count and the file descriptor but not the file since the requirement is that the file can be accessed simultaneously (placing it in the monitor would ensure mutual exclusion). One guard method is needed to get the file descriptor, lets call it getDescriptor. This method will check if your id + the current count ¡= n. while the number is not less than n, the threads should wait on a condition lock (we use while instead of if so that the thread checks again when it is woken up). If it is less than current count, it will increment the current count and return the descriptor. The other guard method called, say, releaseDescriptor() will decrement the current count by the current thread number, and then broadcast the other threads to wake up and check (if you signal, you might wake up a thread with a large number that would not be able to go in, while another waiting after it, could have gone in).

(b) When a signal is performed on a condition inside a monitor, the signaling process can either continue its execution (Mesa style monitors) or transfer control to the process that is signaled (Hoare style monitors). How would the solution to the problem change with the two different ways in which signaling can be performed?

In Hoare style, when you broadcast the threads, you have to make sure that you decremented the number first.

<u>*Problem 8:*</u> (bonus credit; 25 minutes) Recall Peterson's algorithm for two processes as shown below. Can you extend it to work for 3 processes?

```
bool flag[2];
int turn = 0;

Process 0                  Process 1
.                          .
.                          .
flag[0] = 1;               flag[1] = 1;
turn = 1;                  turn = 0;
while (flag[1] == 1        while (flag[0] == 1
       && turn == 1);             && turn == 0);
[Critical Section]         [Critical Section]
flag[0] = 0;               flag[1] = 0;
```

A starting point to attacking this problem is to remember that for mutual exclusion, a thread has to compete with and win over all the other threads (for example, see the bakery algorithm code, how we check all the tickets after getting ours). So, process 0 can first compete with process 1, then process 2. Process 1, can compete with process 0, and then 2. Process 2 can compete with 0 then 1.

```
bool flag[3];
int turn01 = 0;  //to simplify understanding, we'll make turn take the
int turn02 = 0;  //process number
int turn12 = 1;

Process 0                  Process 1                      Process 2
.                          .
.                          .
flag[0] = 1;               flag[1] = 1;                   flag[2]=1;
turn01 = 1;                turn01 = 0;                    turn02=0;
while (flag[1] == 1        while (flag[0] == 1            while(flag[0]==1
       && turn01 == 1);           && turn01 == 0);             && turn02==0);
turn02 = 2;                turn12=2;                      turn12=1;
while (flag[2] == 1        while (flag[2] == 1            while(flag[1]==1
      && turn02 == 2);           && turn12 == 2);          && turn12==1);
[Critical Section]       [Critical Section]          [Critical Section]
flag[0] = 0;             flag[1] = 0;                flag[2]=0;
```

Essentially we are getting two of the three available locks to go in (the two that involve us competing with the other two threads). In such a case, its important to be careful not to get into deadlock. If we switch the order of the competition at P1, so that it competes with P2 first then P0, the following deadlock situation may happen. P0 wins 01, but loses 02; P1 wins 12, but loses 01; and P2 wins 02 and loses 12. All three are blocked.

9(a) Bonus Problem (5%): Explain how the multiple readers multiple writers problem implementation below works

```
//All semaphores initialized to 1
reader                    writer
...                       ...
wait(one_reader);
wait(read);               wait(mutex2);
wait(mutex1);             writers++;
readers++;                if(writers == 1)
if(readers == 1)            wait(read);
  wait(write);            signal(mutex2);
signal(mutex1);           wait(write);
signal(read);             WRITE;
signal(one_reader);
READ;                     signal(write);
wait(mutex1);             wait(mutex2);
readers--;                writers--;
if(readers==0)            if(writers == 0)
  signal(write);            signal(read);
signal(mutex1);           signal(mutex2);
```

The idea of behind this implementation is to give writers priority. In the standard implementation, writers have to wait for all the readers to leave before they can go in. If readers continue to arrive, the writers never get a chance to make updates. The key to this implementation is to allow writers to go in after the current readers leave (but without allowing new readers to go in). The `one_reader` semaphore makes sure the readrs queue on semaphore `read` one at a time. Once a writer goes in, it will wait on read and disable subsequent readers from going in (one of them will be waiting at read, and the others at `one_reader`). It will then go and wait on write. Thus, the readers already in READ will all finish, and the last one will signal write allowing the writer to go in next. Note that subsequent writers will queue on write, taking priority over readers.