Operating System Exercises

Facultad de Informática, UCM

Unit 3.3: Communication and Synchronization

1.-Consider the following program, which aims to solve the critical section problem for two threads by employing a purely software scheme that does not rely on any OS-provided synchronization service:

```
void thread1()
{
  while(TRUE){
   //Busy wait
   while(turn!=0);
   <<Critical section T1 >>
   turn = 1;
   <<More code T1 >>
  }
}
```

```
void thread2()
{
  while(TRUE){
   //Busy wait
   while(turn!=1);
   <<Critical section T2 >>
   turn = 0;
   <<More code T2 >>
  }
}
```

Does the program solve the critical section problem correctly, thus ensuring mutual exlusion, progress and bounded waiting (no starvation)? Justify your answer.

2.-Write a concurrent program consisting of three threads that communicate with each other. Thread 1 will generate the first 1000 even numbers and thread 2 will take care of generating the first 1000 odd numbers. Thread 3, by contrast, will read (consume) these numbers and will print them on the screen. The concurrent program must ensure that numbers displayed on the screen are in order: 1,2,3,4,5... Impose the necessary synchronization by using:

a) Locks and condition variables

b) Semaphores

3.-Implement a general semaphore by using mutexes, condition variables and other shared variables (e.g., integers). In doing so, the new C data type `sem_t` must be defined with the necessary fields, by means of a structure definition and an associated typedef statement. The *wait* and *signal* operations on the semaphore must be implemented as two separate functions that accept a pointer to a `sem_t` as an argument.

4.-Provide a solution to the **readers-writers** problem by using mutexes, condition variables and other shared variables (e.g., integers or booleans).

a) Implement a solution that gives preference to readers: *no reader shall be kept waiting if other readers are inside the critical section.*

b) Implement a solution that gives preference to writers: *if a writer wants to access the critical section, it should not be kept waiting longer than absolutely necessary.* (This may require to explicitly deny access to the critical section to readers temporarily.)

5.-Provide a solution to the **dining philosophers** problem by using a monitor (a mutex, condition variables and other shared variables). The monitor must implement two procedures – `getChopsticks(i)` and `putChopsticks(i)` – which enable a philosopher (identified by the i parameter) to grab the two associated chopsticks and put them back on the table, respectively. The following code snippet represents the behavior of a philosopher (thread):

```
void philosopher(int i) {
  while(1){
    think();
    /* Ask the monitor authorization to grab the chopsticks */
    getChopsticks(i);
    eat();
    /* Tell the monitor to put the chopsticks back on the table */
    putChopsticks(i);
  }
}
```

6.-The cigarette smokers problem was originally presented by Suhas Patil. In this problem, four threads are involved: an agent and three smokers. The smokers loop forever, first waiting for ingredients, then making and smoking cigarettes. To make it possible for a smoker to make a cigarette and smoke it, each smoker needs three ingredients: tobacco, paper and matches. We assume that the agent has an infinite supply of all three ingredients, and each smoker has an infinite supply of one of the ingredients. As such, one smoker has matches, another has paper, and the third has tobacco.

The agent repeatedly chooses two different ingredients randomly and then makes them available to the smokers by putting them on a table. In doing so, and depending on which ingredients are chosen, the smoker with the complementary ingredient picks up both resources and proceeds to smoke.

Write a concurrent program that synchronizes the agent and the smokers by means of a monitor. Assume that the agent has absolutely no way to know the ingredients each smoker has.

7.-The Dining Savages Problem. A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary. When a savage wants to eat, he helps himself from the pot, unless it is empty. If the pot is empty, the savage wakes up the cook and then waits until the cook has refilled the pot.

A number of savage threads run the following code:

```
while (True){
  getServingFromPot()
  eat()
}
```

The cook thread behaves as follows:

```
while (True){
  putServingsInPot(M)
}
```

The synchronization constraints are as follows:

- Savages cannot invoke `getServingFromPot()` if the pot is empty.
- The cook can invoke `putServingsInPot()` only if the pot is empty.

Add the necessary code for the savages and the cook to satisfy the synchronization constraints by using the following resources:

a) Mutexes, condition variables and other shared variables, such as integers or booleans.

b) Semaphores and other shared variables, such as integers or booleans.

8.-Create a concurrent program that emulates the behavior of a gas station featuring two pumps enabling customers to pay with credit card. When a customer is using a pump, he helps himself from the pump by invoking a function –`void PumpFuel(int pump_id, int price)`. When done, the pump will be released and so another customer will be able use it from then on. In order for the gas station to be fair to customers, the following restrictions must be enforced:

- Pumps must be assigned to the customers in order according to each customer's arrival time.
- Two customers cannot be using the same pump simultaneously.
- No customers shall be kept waiting unnecesarily as long as there is a unused pump.

Each customer, modeled as a thread of the concurrent program, behaves as follows:

```
void customer(int price) {
      int pump;
      pump=getUnusedPump();
      /* Using pump */
      PumpFuel(pump,price);
      /* Done using pump */
      releasePump(pump);
}
```

By using mutexes, condition variables and other shared variables, implement the `getUnusedPump()` and `releasePump()` functions to impose the aforementioned synchronization restrictions. Note that the `getUnusedPump()` function shall block a customer until a free pump is assigned to the customer.