Homework 3
CS 414/415, Spring 2002
Emin Gün Sirer

1. Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time	Priority
P1	3	2
P2	8	1
P3	4	3
P4	1	4
P5	4	1

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5 all at time 0.

a) Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a nonpreemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1 ms) scheduling.

b) What is the turnaround time of each process for each of the scheduling algorithms in part a?

c) What is the waiting time of each process for each of the scheduling algorithms in part a?

d) Which of the schedules in part a results in the minimal average waiting time (over all processes)?

2. Why does a multilevel feedback queue usually have different quanta for each level ? How do processes get promoted (and demoted) from one queue to another ?

3. Semaphore waiting lists are often implemented as FIFO queues. Could they be implemented as stacks? What problems might this cause?

4. Figure 7.22 (pg. 219 in the book) proposes a solution to the dining philosopher problem. What would happen if we comment out the line "state[i] = thinking" in the procedure putdown() and why ?

5. You are designing a data structure for efficient dictionary lookup in a multithreaded application. The design uses a hash table that consists of an array of pointers each corresponding to a hash bin. The array has 1001 elements, and a hash function takes an item to be searched and computes an entry between 0 and 1000. The pointer at the computed entry is either null, in which case the item is not found, or it points to a doubly linked list of items that you would search sequentially to see if any of them matches the item you are searching for. There are three functions defined on the hash table: Insertion (if an item is not there already), Lookup (to see if an item is there), and deletion (to remove an item from the table). Considering the need for synchronization, would you:

1. Use a mutex over the entire table?
2. Use a mutex over each hash bin?
3. Use a mutex over each hash bin and a mutex over each element in the doubly linked list?

Justify your answer.

6. Ping and pong are two separate threads executing their respective procedures. The code below is intended to cause them to forever take turns, alternately printing “ping” and “pong” to the screen. The `minithread_stop`() and `minithread_start` () routines operate as they do in your project implementations: `minithread_stop`() blocks the calling thread, and `minithread_start` () makes a specific thread runnable if that thread has previously been stopped, otherwise its behavior is unpredictable.

```
void ping()
{
     while(true) {
          minithread_stop();
          Printf(“ping is here\n”);
          minithread_start(pongthread);
     }
}
void pong()
{
     while(true) {
          Printf(“pong is here\n”);
          minithread_start(pingthread);
          minithread_stop();
     }
}
```

a. The code shown above exhibits a well-known synchronization flaw. Briefly outline a scenario in which this code would fail, and the outcome of that scenario.

b. Show how to fix the problem by replacing the `minithread_stop and start` calls with semaphore P and V operations.

c. Implement ping and pong correctly using a mutex and condition variables.

7. You have been hired to coordinate people trying to cross a river. There is only a single boat, capable of holding at most three people. The boat will sink if more than three people board it at a time. Each person is modeled as a separate thread, executing the function below:

```
Person(int location)
// location is either 0 or 1;
// 0 = left bank, 1 = right bank of the river
{
     ArriveAtBoat(location);
     BoardBoatAndCrossRiver(location);
     GetOffOfBoat(location);
}
```

Synchronization is to be done using monitors and condition variables in the two procedures `ArriveAtBoat` and `GetOffOfBoat`. Provide the code for `ArriveAtBoat` and `GetOffOfBoat`. The `BoardBoatAndCrossRiver()` procedure is not of interest in this problem since it has no role in synchronization. `ArriveAtBoat` must not return until it safe for the person to cross the river in the given direction (it must guarantee that the boat will not sink, and that no one will step off the pier into the river when the boat is on the opposite bank). `GetOffOfBoat` is called to indicate that the caller has finished crossing the river; it can take steps to let other people cross the river.

8. A car is manufactured at each stop on a conveyor belt in a car factory. A car is constructed from the following parts - chassis, tires, seats, engine (assume this includes everything under the hood and the steering wheel), the top cover, and painting. Thus there are 6 tasks in manufacturing a car. However, tires, seats or the engine cannot be added until the chassis is placed on the belt. The car top cannot be added until tires, seats and the engine are put in. Finally, the car cannot be painted until the top is put on.

A stop on the conveyor belt in your car company has four technicians assigned to it - Abe, Bob, Charlie, and Dave. Abe is skilled at adding tires and painting, Bob can only put the chassis on the belt, Charlie only knows how to attach the seats, and Dave knows how to add the engine as well as how to add the top.

Write pseudocode for Abe, Bob, Charlie and Dave to be able to work on the car, without violating the task order outlined above. You can use semaphores, mutexes or monitors in your solution.

9. Write a simulator for Dragon Day, where each architect and engineer is modeled as a separate thread executing their respective functions.

```
void initsimulator() {
      // TODO: initialize any synch variables
}
void architecture_student() {

      // TODO: Architect enters equad
      EnterEngineeringQuad();

      // TODO: an architect should do nothing until s/he
      // observes a snowball thrown by an engineer
      WaitUntilSnowballIsThrown()

      // run out of the Equad, but only after seeing a snowball
      // thrown by an engineer
      ExitEngineeringQuad();
}
void engineer(){
      // TODO: an engineer should wait until there
      // are at least two architects who are not
      // running around.
      WaitUntilThereAreTwoArchitects();

      // throw two snowballs
      ThrowSnowball(); // first one
      ThrowSnowball(); // second one
}
```

Your task is to write `EnterEngineeringQuad, WaitUntilSnowballIsThrown, WaitUntilThereAreTwoArchitects, and ThrowSnowball().`

10. You have been hired by Large-Concurrent-Systems-R-Us, Inc. to review their code. Below is their **atomic_swap** procedure. It is intended to work as follows:

Atomic_swap should take two queues as arguments, dequeue an item from each, and enqueue each item onto the opposite queue. If either queue is empty, the swap should fail and the queues should be left as they were before the swap was attempted. The swap must appear to occur atomically – an external thread should not be able to observe that an item has been removed from one queue but not pushed onto the other one. In addition, the implementation must be concurrent – it must allow multiple swaps between unrelated queues to happen in parallel. Finally, the system should never deadlock.

Note if the implementation below is correct. If not, explain why (there may be more than one reason) and rewrite the code so it works correctly. Assume that you have access to enqueue and dequeue operations on queues with the signatures given in the code. You may assume that q1 and q2 never refer to the same queue. You may add additional fields to stack if you document what they are.

```
extern Item *dequeue(Queue *);     // pops an item from a stack
extern void enqueue(Queue *, Item *); // pushes an item onto a stack

void atomic_swap(Queue *q1, Queue *q2) {
      Item *item1;
      Item *item2; // items being transferred

      P(q1->lock);
      item1 = pop(q1);
      if(item1 !=  NULL) {
           P(q2->lock);
           item2 = pop(q2);
           if(item2 != NULL) {
                push(q2, item1);
                push(q1, item2);
                V(q2->lock);
                V(q1->lock);
           }
      }
}
```