# CS414 SP 2007 Assignment 2

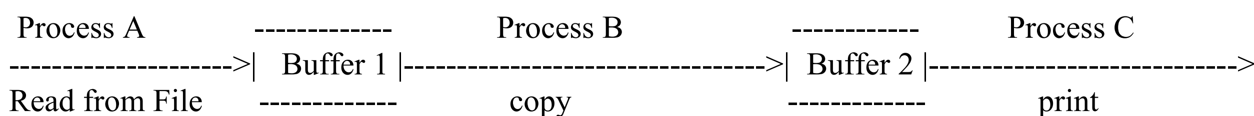
**Due Feb. 19 at 11:59pm Submit your assignment using CMS**

1. Explain what goes wrong in the following variation of Peterson's algorithm:

```
Process Pi:
          do {
                flag[i] = TRUE;
                turn = i;
                while (flag[j] && turn == j);

                <critical section>

                flag[i] = FALSE;

                <remainder section>

          } while (TRUE);
```

Answer:
Process j could be in its critical section (with flag[j] == true && turn == j), and since process i sets turn to i before checking the value, process i will then pass the critical section guard and begin executing in the critical section as well, thus mutual exclusion is not preserved.

2. Three processes are involved in printing a file (pictured below). Process A reads the file data from the disk to Buffer 1, Process B copies the data from Buffer 1 to Buffer 2, finally Process C takes the data from Buffer 2 and print it.




Assume all three processes operate on one (file) record at a time, both buffers' capacity are one record. Write a program to coordinate the three processes using semaphores.

Answer:

```
semaphore empty1 = 1;
semaphore empty2 = 1;
semaphore full11 = 0;
semaphore full12 = 0;

Process_A () {
    while(1) {
        wait(empty1);
        read(next_file(), Buffer_1);
        signal(full1);
```

```
    }
}

Process_B () {
    while(1) {
        wait(full1);
        wait(empty2);
        copy(Buffer_2, Buffer_1);
        signal(empty1);
        signal(full2);
    }
}

Process_C () {
    while(1) {
        wait(full2);
        print(Buffer_2);
        signal(empty2);
    }
}
```

3. Some monkeys are trying to cross a ravine. A single rope traverses the ravine, and monkeys can cross hand-over-hand. Up to five monkeys can be hanging on the rope at any one time. if there are more than five, then the rope will break and they will all die. Also, if eastward-moving monkeys encounter westward moving monkeys, all will fall off and die.

Each monkey operates in a separate thread, which executes the code below:

```
typedef enum {EAST, WEST} Destination ;

void monkey(int id, Destination dest) {
        WaitUntilSafeToCross(dest);
        CrossRavine(id, dest);
        DoneWithCrossing(dest);
}
```

Variable id holds a unique number identifying that monkey. CrossRavine(int monkeyid, Destination d) is a synchronous call provided to you, and it returns when the calling monkey has safely reached its destination. **Use semaphores to implement WaitUntilSafeToCross(Destination d) and DoneWithCrossing(Destination d).**

(a) For Part (a) You implementation should ensure that :

i) At most 5 monkeys simultaneously execute CrossRavine().

ii) All monkeys executing in CrossRavine() are heading in the same direction.

iii) No monkey should wait unnecessarily. (This way, you can maximize the throughput)

Answer:

```
//Shared data
int monkey_count[2] = {0,0};        /* monkey counter for each direction */
```

```
semaphore mutex[2] = {1,1};         /* mutual exclusion for each direction*/
semaphore max_on_rope = 5;          /* ensure maximum 5 monkey on the rope*/
semaphore rope = 1;                 /* ensure monkey on the rope is heading
                                       the same direction */

void WaitUntilSafeToCross(Destination dest)
{
   wait(mutex[dest]);
   monkey_count[dest]++;
   if (monkey_count[dest] == 1)
       //is the first monkey in line, waiting to acquire the rope
       wait(rope);
   signal(mutex[dest]);
   wait(max_on_rope);
}

void DoneWithCrossing(Destination dest)
{
   wait(mutex[dest]);
   signal(max_on_rope);
   monkey_count[dest]--;
   if (monkey_count[dest] == 0)
       //is the last monkey, release the rope
       signal(rope);
   signal(mutex[dest]);
}
```

(b) Instead of maximizing the throughput, your solution for Part (b) should ensure there is no starvation (the first two conditions in part (a) still apply). That is, no monkey should wait at the ravine forever. You can assume that semaphore's wait queue is FIFO queue.

```
//Shared data
int monkey_count[2] = {0,0};          /*monkey counter for each direction */
semaphore mutex[2] = {1,1};           /*mutual exclusion for each direction*/
semaphore max_on_rope = 5;            /*ensure maximum 5 monkey on the rope*/
semaphore rope = 1;                   /*ensure monkey on the rope is heading
                                        the same direction */
semaphore order = 1;                  /*Use to prevent starvation */

void WaitUntilSafeToCross(Destination dest)
{
   wait(order);
   wait(mutex[dest]);
   monkey_count[dest]++;
   if (monkey_count[dest] == 1)
       //is the first monkey in line, waiting to acquire the rope
       wait(rope);
   signal(mutex[dest]);
   signal(order);
   wait(max_on_rope);
}

void DoneWithCrossing(Destination dest)
{
   wait(mutex[dest]);
   signal(max_on_rope);
   monkey_count[dest]--;
```

```
    if (monkey_count[dest] == 0)
        //is the last monkey, release the rope
        signal(rope);
    signal(mutex[dest]);
}
```

(c) Now instead of monkeys, we have students crossing the ravine. Because students are much smarter than monkeys, assume that students heading different directions **CAN** be on the rope at same time. Write a semaphore implementation that ensures the following requirements.

i) At most M eastward-moving students simultaneously execute CrossRavine()
ii) At most N westward-moving students simultaneously execute CrossRavine()
iii) At most K students simultaneously execute CrossRavine()
iv) you solution should maximize the throughput under above conditions.

Answer:

```
//Shared Data
semaphore max_oneway[2] = {M, N};
semaphore max_total = K;

void WaitUntilSafeToCross(Destination dest)
{
  p(max_oneway[dest]);
  P(max_total);
}

void DoneWithCrossing(Destination dest)
{
  V(max_total);
  V(max_oneway[dest]);
}
```

.4. A commonly used group synchronization mechanism is called **barrier**. Some computations are divided into phases and have the rule that no thread may proceed into the next phase until all threads are ready to proceed to the next phase. The behavior may be achieved by letting each thread execute **barrier.done** at the end of each phase. Suppose there are N threads in the computation, a call to barrier.done blocks until all of the N threads have called barrier.done. Then all thread proceed. There could be multiple phases requiring synchronization Here is implementation of barrier.done.

```
Semaphore mutex = 1;
semaphore barrier = 0;
int count = 0;

void barrier.done() {
      wait(mutex);
      count++;
      if (count < N ) {
            signal(mutex);
            wait(barrier);
      }
```

```
        else {
              signal(mutex);
              count = 0;
              for (int i = 1; i < N; i++)
                    signal(barrier);
        }
  }
```

Explain why above solution might not work. (Hint: your solution should not depend on the implementation of the semaphore's wait queue)

Answer:
The main problem is that the state of “count” and the state of “barrier” might be inconsistent. That is, if correct, the value of “count” should represent the number of processes ALREADY waiting on the semaphore “barrier”. However it's possible for some thread(s) to be context-switched out before it calls wait(barrier) but after it has increased the counter. Consequently, the Nth thread will signal more times than the number of waiting threads, causing the semaphore value of barrier to be positive, then the threads that get waked up first might run through the second barrier before others go through the first one. Note, this problem exists no matter what queuing strategy the semaphore wait list is.

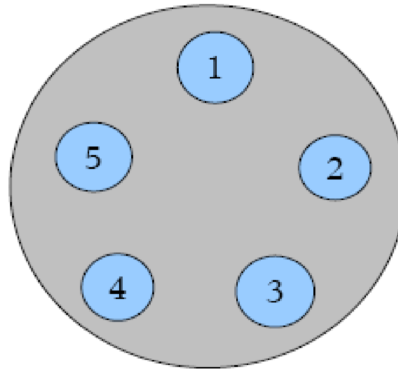
Since the hint implies (wrongly) that the problem is related to the queuing strategy. The following answer also gets full credit. This implementation might fail when there are more than one phases. The correctness should not depend on the queuing strategy of the semaphore wait list. Say if priority queue is used, then the thread with the high priority might go through 2nd barrier before others go through the first one.

5.
(a) Suppose we replace the wait() and signal() operations of monitors with a single construct await(B), where B is a general Boolean expression that causes the process executing it to wait until B become true. Explain why, in general, this construct cannot be implemented efficiently.

Answer: The system would have to check which one of the waiting threads have to be awakened by checking which one of their waiting conditions are satisfied. This requires considerable complexity as well as might require some interaction with the compiler to evaluate the conditions at different points in time.

(b) The textbook gives a monitor solution to the dining-philosopher problem (on page 214, Figure 6.19). One problem with this solution is that it is possible for a philosopher to starve to death. Illustrate why?

Answer:




To simplify the analysis, let's assume that all the philosopher does very minimal thinking. That is, they eat for a while, give up the chopstick, then immediately come back to wait for eating again. Suppose, all philosophers indicated their wish to eat around the same time. Consider the following sequence:

- 1 and 3 start eating, 2, 4 and 5 are blocking
- 3 stops eating. 2 and 5 remain blocked since 1 is eating. So 4 gets the chopstick and start eating.
- 4 stops eating. 2 and 5 remain blocked since 1 is eating. So 3 gets the chopstick and start eating.
- 1 stops eating. 2 and 4 remain blocked since 3 is eating. So 5 gets the chopstick and start eating.
- 5 stops eating, 2 and 4 remain blocked since 3 is eating. So 1 gets the chopstick and start eating.
- repeat from the second step.

In above sequence, 2 will never get the chance to eat.