Homework #3 Solutions

1. Suppose that the following processes arrive for execution at the times indicated. Each process will run with a single burst of CPU activity (i.e., no I/O) which lasts for the listed amount of time.

process	arrival time	CPU burst time	priority
p1	0ms	18ms	2
p2	1ms	12ms	1
p3	20ms	16ms	3
p4	31ms	14ms	4

a. What is the job throughput, average waiting time and average turnaround time for these processes with non-preemptive, FCFS scheduling?

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
P1	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**												
P2		R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
P3																					R	R	R	R	R	R	R	R	R	R
P4																														

	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
P1																														
P2																														
P3	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**														
P4		R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**

Job throughput: 4 jobs in 60 time units, or **(4/60) jobs per time unit**

Avg waiting time: if wait time == time on ready queue (textbook definition), then this is (0 + 17 + 10 + 15)/4 = **42/4 time units**

if wait time == time on wait queue (my definition from slides), then this is 0 time units.

Avg turnaround time: turnaround time = time of completion of process – time of submission of process

avg turnaround = ((18-0) + (30-1) + (46-20) + (60-31))/4
= **102/4 time units**

b.With preemptive RR (quantum = 10ms) scheduling? (Different strategies might be used to add a newly submitted process to the ready queue. Explain what strategy you're using.)

Assume new processes go on the TAIL of the ready queue:

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
P1	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	R	R	R	R	R	R	R	R	R	R	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**		
P2		R	R	R	R	R	R	R	R	R	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	R	R	R	R	R	R	R	R	**X**	**X**
P3																					R	R	R	R	R	R	R	R	R	R
P4																														

	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
P1																														
P2																														
P3	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	R	R	R	R	R	R	R	R	R	R	**X**	**X**	**X**	**X**	**X**	**X**				
P4		R	R	R	R	R	R	R	R	R	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	R	R	R	R	R	R	**X**	**X**	**X**	**X**

Job throughput: 4 jobs in 60 time units, or **(4/60) jobs per time unit**

Avg waiting time: if wait time = time on ready queue (textbook definition), then this is (10 + 17 + 20 + 15)/4 = **62/4 time units**

if wait time == time on wait queue (my definition from slides), then this is 0 time units.

Avg turnaround time: turnaround time = time of completion of process – time of submission of process (page 128 of textbook)

avg turnaround = ((28-0) + (30-1) + (56-20) + (60-31))/4
= **122/4 time units**

Assume new processes go on the HEAD of the ready queue:

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
P1	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
P2		R	R	R	R	R	R	R	R	R	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	R	R	R	R	R	R	R	R	R	R
P3																					**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
P4																														

	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
P1	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**																						
P2	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	**X**	**X**										
P3	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	**X**	**X**	**X**	**X**	**X**	**X**				
P4		R	R	R	R	R	R	R	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	R	R	R	R	R	R	R	R	**X**	**X**	**X**	**X**

Job throughput: 4 jobs in 60 time units, or **(4/60) jobs per time unit**

Avg waiting time: if wait time = time on ready queue (textbook definition), then this is (20 + 37 + 18 + 15)/4 = **90/4 time units**

if wait time == time on wait queue (my definition), then this is 0 time units.

Avg turnaround time: turnaround time = time of completion of process – time of submission of process (page 128 of textbook)

avg turnaround = ((38-0) + (50-1) + (56-20) + (60-31))/4
= **152/4 time units**

c. With preemptive priority scheduling (given the above priorities)?

Assuming higher numbers mean higher priority:

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
P1	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**												
P2		R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	**X**	**X**	R	R	R	R	R	R	R	R	R	R
P3																					**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
P4																														

	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
P1																														
P2	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**
P3	**X**	R	R	R	R	R	R	R	R	R	R	R	R	R	R	**X**	**X**	**X**	**X**	**X**										
P4		**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**															

Job throughput: 4 jobs in 60 time units, or **(4/60) jobs per time unit**

Avg waiting time: if wait time = time on ready queue (textbook definition), then this is (0 + 47 + 14 + 0)/4 = **61/4 time units**

if wait time == time on wait queue (my definition), then this is 0 time units.

Avg turnaround time: turnaround time = time of completion of process – time of submission of process (page 128 of textbook)

avg turnaround = ((18-0) + (60-1) + (50-20) + (45-31))/4
= **121/4 time units**

2. Consider the Sleeping-Barber Problem (p233, question 6.11 in the textbook,) with the modification that there are k barbers and k barber chairs in the barber room, intead of just one. Write a program to coordinate the barbers and the customers using Java, C, or pseudo-code. You can use either semaphores or monitors.

Here's a solution that uses semaphores:

```
// shared data
Semaphore waiting_room_mutex = 1;
Semaphore barber_room_mutex = 1;

Semaphore barber_chair_free = k;
Semaphore sleepy_barbers = 0;

Semaphore barber_chairs[k] = {0, 0, 0, …};
int       barber_chair_states[k] = {0, 0, 0, …};

int       num_waiting_chairs_free = N;


boolean customer_entry( ) {

   // try to make it into waiting room
   wait(waiting_room_mutex);
   if (num_waiting_chairs_free == 0) {
      signal(waiting_room_mutex);
      return false;
   }
   num_waiting_chairs_free--;   // grabbed a chair
   signal(waiting_room_mutex);

   // now, wait until there is a barber chair free
   wait(barber_chair_free);

   // a barber chair is free, so release waiting room chair
   wait(waiting_room_mutex);
   wait(barber_room_mutex);
   num_waiting_chairs_free++;
   signal(waiting_room_mutex);

   // now grab a barber chair
   int mychair;
   for (int I=0; I<k; I++) {
     if (barber_chair_states[I] == 0) {  // 0 = empty chair
        mychair = I;
        break;
        }
      }
      barber_chair_states[mychair] = 1;  // 1 = haircut needed
      signal(barber_room_mutex);

      // now wake up barber, and sleep until haircut done
      signal(sleepy_barbers);
      wait(barber_chairs[mychair]);

      // great! haircut is done, let’s leave.  barber
      // has taken care of the barber_chair_states array.
      signal(barber_chair_free);
      return true;
   }

   void barber_enters() {
     while(1) {
       // wait for a customer
       wait(sleepy_barbers);

       // find the customer
       wait(barber_room_mutex);
       int mychair;
       for (int I=0; I<k; I++) {
         if (barber_chair_states[I] == 1) {
           mychair = I;
           break;
         }
       }
       barber_chair_states[mychair] = 2;  // 2 = cutting hair
       signal(barber_room_mutex);

       // CUT HAIR HERE
       cut_hair(mychair);

       // now wake up customer
       wait(barber_room_mutex);
       barber_chair_states[mychair] = 0;  // 0 = empty chair
       signal(barber_chair[mychair]);
       signal(barber_room_mutex);

       // all done, we’ll loop and sleep again
     }
   }
```

3. “Spot the bugs”

a & b: Here are all the bugs I deliberately planted…

i. The return values of pthread_* functions are not checked. These functions can return error codes; since they aren’t checked, all sorts of bad things could go wrong. For example, if the pthread_mutex_init() functions fail, the mutex may not function correctly. The fix is simply to check the return values, and exit the program if the return values indicate an error. The bug is potentially non-deterministic, especially if the mutexes aren’t working correctly.

ii. Malloc return value not checked. This means the malloc might fail, and depending on what the compiler sets an uninitialized pointer to point at, pretty much anything could happen. The fix is to check the return value of malloc, and exit if malloc fails

iii. The 4th arguments to pthread_create point to memory that is dynamically allocated by malloc but free()’d a short time later. Depending on whether the created threads run first, they may attempt to reference memory that has been freed, or worse, allocated and handed to somebody else! The fix is to not free the memory until after the pthread_join().

iv. The producer and consumer threads both busy wait, but this assumes that the threads are preemptively scheduled (as opposed to cooperatively context switched). The fix is to check the man pages for the thread package and operating system carefully. The bug would manifest itself as an infinite loop (deterministically) if the thread package is not preemptive.

v. (not really a bug, but a performance flaw) Busy waiting is wasteful – you ought to use a condition variable, or use sched_yield() if the OS or thread package supports that system call.

vi. The line:

```
if (production_done) return NULL;”
```

might return before consuming everything that is produced. Replace with:

```
if (production_done && (items_in_buffer == 0))
```

This bug is timing dependent.

vii. There is the potential for a deadlock, as the locks are grabbed in different orders by the different threads. Whether or not a deadlock happens is timing dependent. The fix is for both threads to grab the locks in the same order.

viii. The consume rthread releases the locks before it should -- if a context switch happens between the time the consumer releases the locks and the time the consumer does its printf, the producer could overwrite the item being printed by the printf.

c. There's no reason to have two locks – you could replace the two locks with a single lock, as long as you made sure the single lock was held when reading or writing from any of the shared variables.

d. (hard) It is possible to build a producer-consumer solution that uses no locks, but only if there is a single producer and a single consumer. The trick is to define the buffer as a circular array, with the producer having the right to modify a pointer into the head of the array, and the consumer having the right to modify a pointer to the tail. When the producer wants to add, it makes sure there is empty space by comparing the tail to the head. When the consumer wants to consume, it does the same thing. However, if there are more than one producer or consumer threads, then you will need some kind of synchronization.

4. Use Java, C, or pseudo-code to implement:

   - monitors using semaphores
   - semaphores using monitors

   Your solution may *not* busy-wait.

```
monitors using semaphores:

   - the answer, for the most part, is in section 6.7 of the text.
     here's some brief pseudocode to fill in the blanks.

     Semaphore mutex = 1, next = 0;
     int next_count = 0;

     For each external procedure F:

     wait(mutex);
      ...
     body of F;
     ...
     if (next_count > 0)
         signal(next);
     else
         signal(mutex);
```

```
For each condition x

int x_count = 0;
semaphore x_sem = 0;

///////////////////////////
// x.wait
x_count = x_count + 1;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count = x_count - 1;

///////////////////////////
// x.signal
if (x_count > 0) {
    next_count = next_count + 1;
    signal(x_sem);
    wait(next);
    next_count = next_count - 1;
}
```

- semaphores using monitors:

```
class Semaphore : public Monitor
{
protected:
    int count;
    condition cond;

public:
    Semaphore(int initial) {
        count = initial;
    }

    void wait() {
        count = count - 1;
        while (count < 0) {
            cond.wait();
        }
    }

    void signal() {
        count = count + 1;
        cond.signal();
    }
};
```