PRACTICE SHEET 1: CS 170

1) Consider the following C program snippet:

```
int main(int argc, char **argv)
{
    fork();
    fork();
    while (1) ;   /* wait forever */
}
```

Someone compiles and runs this program. Once all calls to fork() are done and the processes are waiting in the infinite loop, how many copies (processes) of this program are running?

4

2) Consider the following C program snippet:

```
01: int global = 0;
02:
03: int helper()
04: {
05:     int local_2 = 42;
06:     printf("helper called");
07: }
08:
09: int start()
10: {
11:     int local_1 = 17;
12:     printf("thread is here");
13:     thread_exit();
14: }
15:
16: int main(int argc, char **argv)
17: {
18:     int cnt;
19:     for (cnt = 0; cnt < 3; cnt++)
20:         thread_create(start);
21:     wait_for_all_threads_to_finish();
22: }
```

Assume that the program is compiled and run. Further assume that the main function has spawned all three threads, which are currently in their start function, just before the printf statement on Line 12.

How many copies of the following variables do exist in the address space of the running process?

a. cnt: ***1***

b. local_1: ***3***

c. local_2: ***0***

d. global: ***1***

3) Consider an operating system that uses a priority-based algorithm to schedule user processes. The first table below shows the points in time at which different processes become READY, their corresponding run (execution) times, and their priorities (higher numbers mean higher priorities). Your task is to determine the order in which these processes are scheduled by the operating system. For your answer, fill in the tables and give all points in times at which a new process becomes RUNNING (i.e., give the point in time and the name of the process that gets control of the CPU). Use preemptive scheduling!

Process	READY Time	Run Time	Priority
A	0	2	3
B	3	7	1
C	4	3	4
D	5	1	5

a. Preemptive schedule

Schedule												
Time	***0***	***3***	***4***	***5***	***6***	***8***						
Process	***A***	***B***	***C***	***D***	***C***	***B***						

4) You have just been hired by Mother Nature to help her out with the chemical reaction to form water, which she doesn't seem to be able to get right due to synchronization problems. The trick is to get two *H* atoms and one *O* atom react together at the same time. For our problem, each atom is represented by a thread. Each *H* atom thread invokes a procedure called *hReady()* when it is ready to react, and each *O* atom thread invokes a procedure called *oReady()* when it is ready.

For this problem, your task it to write the code for the *hReady()* and *oReady()* procedures. The procedures must delay (block) the calling threads until there are at least two *H* atoms and one *O* atom present. When this is the case, then one (and only one) of the three threads must call the procedure *makeWater()* (which just prints out a debug message that water was made). After the *makeWater()* call, two instances of *hReady*() and one instance of *oReady*() should return.

Your solution must avoid starvation and busy-waiting! You can use any synchronization routine that we discussed in class. When using semaphores, you may assume that the semaphore implementation enforces FIFO order for wakeups – that is, the thread waiting longest in P() is the one woken up after a V() operation by another process.

Initialization:

```
Semaphore mutex = 1;
Semaphore h_wait = 0;
Semaphore o_wait = 0;


hReady() {
   V(o_wait);    /* you can alternatively write o_wait.V(); */
   P(h_wait);    /* you can alternatively write h_wait.P(); */
}

oReady() {
   P(mutex);

   P(o_wait);
   P(o_wait);
   V(h_wait);
   V(h_wait);
   makeWater();

   V(mutex)
}
```

5) Consider the following snapshot of a system with five processes (P1, P2, P3, P4, P5) and four resources (R1, R2, R3, R4). There are no current outstanding queued unsatisfied requests.

Currently Available Resources

R1	R2	R3	R4
2	1	2	0

	Current Allocation				Max Need				Still Needs			
Process	**R1**	**R2**	**R3**	**R4**	**R1**	**R2**	**R3**	**R4**	**R1**	**R2**	**R3**	**R4**
P1	0	0	1	2	0	0	3	2	0	0	2	0
P2	2	0	0	0	2	7	5	0	0	7	5	0
P3	0	0	3	4	6	6	5	6	6	6	2	2
P4	2	3	5	4	4	3	5	6	2	0	0	2
P5	0	3	3	2	0	6	5	2	0	3	2	0

a. Is this system currently deadlocked, or can any process become deadlocked? Why or why not? If not deadlocked, give an execution order.

Using the Banker's algorithm, the system is not deadlocked and will not become deadlocked. The process finishing order is: P1, P4, P5, P2, P3.

b. If a request from a process P2 arrives for (0, 1, 2, 0), can the request be immediately granted? Why or why not? If yes, show an execution order.

No, if granted, the resulting Currently Available Resources would be (2, 0, 0, 0), and there is no sequence of process executions that would allow (guarantee) the completion of all processes. This is an UNSAFE state.