Solutions
15-410, Spring 2005, Homework Assignment 1.

1 CLOCKNAFORK (10 pts.)

The problem with `clocknaforth()` is a concurrency problem which occurs even if there is only one program thread! The fact that we are reading a 32-bit quantity through two 16-bit windows makes it difficult to obtain a consistent snapshot of the entire 32-bit quantity. The effect is that whatever program is running on the CPU constitutes one thread, and the clock constitutes another.

Consider the following sequence of events:

Execution Trace

time	clocknafork()
0001fc17	lsw = 0xFC17;
0001fc17	msw = 0x0001;
0001fc17	return(0x0001FC17);
...	...
0001ffff	lsw = 0xFFFF;
00020000	msw = 0x0002;
00020000	return(0x0002FFFF);
...	...
000203e8	lsw = 0x03E8;
000203e8	msw = 0x0002;
000203e8	return(0x000203E8);

Observe that the second value returned is exactly 65536 larger than it should be because the top half of one clock value and the bottom half of another were incorrectly combined. Note that this problem occurs even if interrupts are disabled (or not present in the environment).

To phrase the situation somewhat abstractly, the problem we face is that the low-order half we fetched might "match" an "old" high-order half or a "new" high-order half. Once we conceptualize it this way, the solution becomes clear: fetch the high-order half *twice*, once before and once after the low-order half. If the two samplings of the most-significant half return the same value, the least-significant half must match both, unless your execution was suspended for 2^{32} milliseconds, in which case you have other problems. If the two most-significant halves *don't* match, something bad happened (a low-order overflow or perhaps an interrupt), and the three-step sampling is attempted again.

2 Mutex Mutation (10 pts.)

Hopefully thinking through a deadlock-detection design leaves you with a more-solid appreciation of the dimensions of the problem.

One big issue is that `mutexmutex` serializes all mutex operations behind one global mutex. In other words, two threads running on different processors attempting to lock different mutexes, who deserve to run entirely in parallel, must wait for each other. If a many threads are running on many processors, and are frequently locking and unlocking uncontested mutexes in order to protect short instruction sequences (again, this is what should be happening), performance will approximate that of a single processor. Your manager will not approve.

Another issue is that any spurious firing of the detection scanner will be very expensive. A mutex you are trying to lock contains the thread id of the locker, but each step after that through a potential deadlock cycle requires mapping from thread id to TCB (this may be fast or slow) and from a mutex id in that TCB to a mutex pointer (this requires linear search). If many threads in the system are in a wait chain which is not a cycle, a potential-deadlock scan will take time proportional to the number of threads times the number of mutexes if the tid-to-TCB mapping is fast, or the number of threads squared times the number of mutexes if the mapping is slow. During that time, of course, no thread can lock or unlock any mutex.

3 Burning Question (10 pts.)

The system (four identical resources, three processes which each use 0, 1, or 2 resources) cannot deadlock. Clearly two processes cannot deadlock by themselves, since the maximal requirements of both can be satisfied by the four resources. In order for there to be a deadlock, then, all three processes would need to be in a hold-and-wait cycle.

At one extreme, each process in the wait cycle might have zero resources. This would mean that four are free and two processes could immediately receive full allocations, i.e., they are not in fact waiting for resources held by other processes in the cycle.

Likewise, if the free list contains three or two resources, one process can be satisfied and there is not a wait cycle.

If the free list contains one resource, then three resources are already distributed among three processes; either one "waiting" process already has two resources, in which case it isn't waiting, or all three "waiting" processes each have one resource and need one. Assign the free resource to one of them and there is no wait cycle.

In the final case the free list contains zero resources, meaning that four resources are distributed among three "waiting" processes. Clearly at least one process already has two resources and isn't waiting, so once again there is no wait cycle.

By the way, this is a re-statement of a problem from the text, which was first posed in 1971.

4 Racy Bakers (10 pts.)

Here is one plausible execution trace. Many variations are possible.

Execution Trace

time	Thread 0	Thread 1	Thread 2
0	`choosing[0]=1`		
1	`number[0]=1`		
2	`choosing[0]=0`		
3	`while(choosing[0]);`		
4	`while(...[0]...);`		
5	`while(choosing[1]);`		
6	`while(...[1]...);`		
7	`while(choosing[2]);`		
8	`while(...[2]...);`		
9	`...enter...`	`choosing[1]=1`	
10			`choosing[2]=1`
11		`number[1]=2`	`number[2]=2`
12		`choosing[1]=0`	`choosing[2]=0`
13		`while(choosing[0]);`	`while(choosing[0]);`
14		`while(...[0]...);`	`while(...[0]...);`
15		`while(...[0]...);`	`while(...[0]...);`
16	`...exit...`	`while(...[0]...);`	`while(...[0]...);`
17	`number[0]=0`	`while(...[0]...);`	`while(...[0]...);`
19		`while(...[0]...);`	`while(...[0]...);`
20		`while(choosing[1]);`	`while(choosing[1]);`
21		`while(...[1]...);`	`while(...[1]...);`
22		`while(...[2]...);`	`while(...[1]...);`
23		`...enter...`	`while(...[1]...);`