15-410, Fall 2004, Homework Assignment 1.
Due Wednesday, October 21, 8pm

1 First-Come, First-Served? (10 pts.)

Thread 0	Thread 1
	choosing[1] = true;
	...max(...)... ⇒0...
choosing[0] = true;	
...max(...)... ⇒0...	
	number[1] = 1;
number[0] = 1;	
choosing[0] = false;	
	choosing[1] = false;
...choosing[0]...	...choosing[0]...
...number[0] != 0...	while (number[0] != 0 && ...) continue;
...(1,0) < (1,0)...	while (number[0] != 0 && ...) continue;
...choosing[1]...	while (number[0] != 0 && ...) continue;
...number[1] != 0...	while (number[0] != 0 && ...) continue;
...(1,1) < (1,0)...	while (number[0] != 0 && ...) continue;
critical section	while (number[0] != 0 && ...) continue;

Thread 1 certainly "came first" in the sense of beginning the critical-section protocol first, updating the global data structure first, and even in the sense of storing its ticket number to memory first, but Thread 0 will immediately drop through the two "proof spin loops" while Thread 1 must remain in the second one until Thread 0 exits the critical section. Furthermore, this unfairness is repeatable: the algorithm has a very predictable bias toward low-numbered threads.

2 What Could Possibly Go Wrong? (10 pts.)

There are many possible horrible things. For example:

- Two threads can be allocated the same memory if they overlap executions of `malloc()`,
- Memory can be lost (leaked) if two threads overlap executions of `free()`,
- The entire heap can be scrambled if two threads overlap execution of a coalesce operation,
- A thread can erroneously believe that the system is out of memory if it and another thread attempt to add pages to the heap at the same time.

3 Fly Away (15 pts.)

The trouble with this problem is that the natural thing to do is to create a condition variable for each of the supplies and have each person wait for the supplies that they need. The problem is that each person needs two supplies and would naturally want to wait for two different things simultaneously. Also, if the two supplies placed on the table are taken by different people then nobody will have enough supplies to build an airplane and no more supplies will be placed on the table. This will immediately result in deadlock.

The question then is how to solve this problem. One might try to wait for one resource, grab it, and then wait for the other and grab it as well. The trouble with this is that it can easily lead to different people splitting the two resources on the table. This can be corrected with a messy collection of mutexes where almost every time the table state changes, every thread wakes up, locks the table, checks for what they want, and goes back to sleep in disappointment.

There is a much better solution where the pairing of resources is explicitly represented in the synchronization data structures. Then each person can wait on the *pair* of resources necessary to build an airplane, and Alpha can signal the pair that he places on the table. An example of such a solution is below. By the way, this problem is (or, at least, we intend) isomorphic to the "Cigarette-Smokers Problem" (Patil, 1971).

```
cond_t plane_kit_and_propellor;
cont_t plane_kit_and_rubber_band;
cont_t propellor_and_rubber_band;

/* This mutex may seem superfluous given the invarients in the problem.
 * There is, however, a subtle problem than can arise if it is omitted.
 * We challenge you to figure out what this problem is.
 */
mutex_t table;

void Delta(void)
{
  while (1) {
    mutex_lock(&table);

    /* wait for the needed supplies */
    while (!table_has_propellor() ||
   !table_has_rubber_band())
      cond_wait(&propellor_and_rubber_band, &table);

    take_propellor();
    take_rubber_band();

    mutex_unlock(&table);

    build_and_fly_plane();
  }
}

void Echo(void)
{
  while (1) {
    mutex_lock(&table);

    /* wait for the needed supplies */
    while (!table_has_plane_kit() ||
   !table_has_rubber_band())
      cond_wait(&plane_kit_and_rubber_band, &table);

    take_plane_kit();
    take_rubber_band();

    mutex_unlock(&table);

    build_and_fly_plane();
  }
}

void Foxtrot(void)
{
  while (1) {
    mutex_lock(&table);

    /* wait for the needed supplies */
    while (!table_has_propellor() ||
   !table_has_plane_kit())
```

```
      cond_wait(&plane_kit_and_propellor, &table);

    take_propellor();
    take_plane_kit();

    mutex_unlock(&table);

    build_and_fly_plane();
  }
}

void Alpha(void)
{
  boolean supply_kit, supply_prop, supply_band;

  while(1) {
    /* get a message from the judges saying it's time to resupply the table */
    while (!received_text_message())
      await_text_message();

    mutex_lock(&table);

    /* pick some supplies */
    pick_ingredients(&supply_kit, &supply_prop, &supply_band);

    /* put supplies on table */
    if (supply_kit)
      put_plane_kit_on_table();
    if (supply_prop)
      put_propellor_on_table();
    if (supply_band)
      put_rubber_band_on_table();


    /* Signal the person who needs the supplied supplies */
    if (supply_kit && supply_prop)
      cond_signal(plane_kit_and_propellor);
    if (supply_kit && supply_band)
      cond_signal(plane_kit_and_rubber_band);
    if (supply_prop && supply_band)
      cond_signal(propellor_and_rubber_band);

    mutex_unlock(&table);
  }
}
```