Solutions
15-410, Fall 2017, Homework Assignment 1.

1 Chefs (4 pts.)

...The maximum numbers they need are shown in the following table.

Chef	Knives	Bowls	Woks
Jamie	1	2	3
Kelly	3	2	1
Morgan	2	2	2

Imagine that the kitchen contains 4 knives, 4 bowls, and 4 woks.
Imagine the system is in the state depicted below.

Chef	Knives	Bowls	Woks
Jamie	1	2	1
Kelly	1	0	1
Morgan	1	2	1
Available	1	0	1

List one request which the system should grant right away, and one request which the system should react to by blocking the process making the request. Briefly justify each of your answers. If you can, arrange for both of your answers to involve the same resource request, but made by two different processes.

In order for a deadlock-avoidance resource allocator to grant a request, granting the requested resources must leave the system in a safe state. For a grantable request to exist, the system's current state must be safe, so let's check.

This system is in a safe state because:

1. Morgan can be satisfied if given one knife and one wok, and the system has one free knife and one free wok. When Morgan runs to completion, the system will have two free knives, two free bowls, and two free woks.

2. If the system has available (2, 2, 2) then it is easy to satisfy Kelly, who already has (1, 0, 1) and thus needs only (2, 2, 0) to run to completion, leaving free resources of (3, 2, 3).

3. If the system has available (3, 2, 3) it is easy to satisfy Jamie, who already has (1, 2, 1) and thus needs only (0, 0, 2) to run to completion.

In other words, the system is in a safe state because the sequence "Morgan; Kelly; Jamie" is safe.

While there may exist many resource requests that the system could safely grant, we just showed that Morgan could be granted all un-requested resources (1, 0, 1) and the system would be in a safe state. Thus, any smaller request would also be safe, e.g., (1, 0, 0), which is "one knife." So if Morgan requests one knife we should grant that request immediately.

Now we would like to show that there is some other chef who is entitled to ask for a knife but who should *not* be granted one. Jamie isn't entitled to a knife, so we hope Kelly is. Indeed, Kelly is entitled to request two knives. Because there is only one knife on hand, it is not possible to grant a request from Kelly for two knives, so a two-knife request would block Kelly, and leave the system in a safe state. So that's not what we are looking for.

If Kelly requests one knife (the same request that we showed is safe for Morgan), the system could potentially grant that request (one knife is available). But then we would be in trouble: Morgan can't necessarily run to completion without a knife, Kelly can't necessarily run to completion without a knife, and Jamie can't necessarily run to completion without two woks (the system has only one wok on hand). So no chef can be the first process in a safe sequence, so no safe sequence can exist, so the state in which we issue Kelly a knife would not be safe, so we should not grant Kelly a knife.

The key thing to observe here is that we have one knife on hand, but it is safe to grant it to one chef but not another. That has nothing to do with the knife, but everything to do with the potential future knife needs of the various chefs, plus the interlocking potential needs for other resources of the various chefs.

2 "Nemo's Algorithm" (6 pts.)

There is a problem with this critical-section protocol. Identify a required property which this protocol does not have and then present a trace which supports your claim. You may use more or fewer columns or lines in your trace.

Two things before we get to the trace.

1. As you may know, "Nemo" is a Latin word meaning "nobody," so if you see somebody named "Nemo" often it is a sign that something fishy is going on. The code we asked you to evaluate in Homework 1 was a variant of Dijkstra's (1965) n-process generalization of Dekker's (1965) two-process solution, roughly presented in a 1986 scholarly book by Michel Raynal, *Algorithms for Mutual Exclusion*. This code doesn't quite have a name, because it's my re-formulation of a re-formulation by uncited authors of an algorithm by Dijkstra.

2. When writing a trace, claims that some sequence repeats must be both precise and correct—writing "now this repeats" at the bottom of a trace is usually not precise and is often incorrect, and thus often results in significant point deductions. In particular, any such claim must make it clear exactly *which part* of the trace repeats (the start and the end of the allegedly repeating part). In addition, *the claim must be true*, which it generally is not if variables have different values. That is, if an array starts off containing all zeroes, and you show a trace that sets the array to all ones, and then you claim "this repeats," your grader will be *very* skeptical. Said differently, for the state of the system to repeat it must be *all* of the relevant state, not just the values of the program counters!

With all that said, the code in question guarantees mutual exclusion and progress but not bounded waiting. Here is a trace.

time	Thread 0	Thread 1
0	e[0]=0	
1		e[1]=0
2	turn==-1	
3		turn==-1
4	turn=0	
5		turn=1
6	turn!=0	
7		turn==1
8		e[1]=1
9		n_e()==1 // w00t!
10	e[0]=0	
11	turn!=-1	
12	turn!=0	
13	e[0]=0	
14	turn!=-1	
15	turn!=0	
16		turn=-1
17		e[1]=0

For step 18, Thread 0 and Thread 1 can both execute code line 12. Also at step 18, e[ ] contains all zeroes and turn contains -1. Thus the steps from 0 to 17 could repeat an unbounded number of times, always giving the lock to Thread 1 and never to Thread 0. This is a bounded-waiting failure. □

In terms of difficulty, this homework is probably easier than typical exam question on similar topics. However, this homework represents some of the reasoning we expect you to carry out in order to detect and explain race conditions (and/or deadlocks).