Solutions
15-410, Fall 2013, Homework Assignment 1.

1 Chefs (4 pts.)

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

Chef	Knives	Bowls	Woks
Robin	7	1	1
Sam	5	2	2
Tracy	4	6	2

1.1 2 pts

Is the following state safe? Why or why not?

Chef	Knives	Bowls	Woks
Robin	4	0	0
Sam	0	2	2
Tracy	3	6	0
Available	4	1	0

Robin can ask for more woks (1) than there are available (0), so Robin can't be the first process in a safe sequence.

Sam can ask for more knives (5) than there are available (4), so Sam can't be the first process in a safe sequence.

Tracy can ask for more woks (2) than there are available (0), so Tracy can't be the first process in a safe sequence.

Since no process can safely complete first, this state is not safe.

1.2 2 pts

Is the following state safe? Why or why not?

Chef	Knives	Bowls	Woks
Robin	6	0	0
Sam	0	2	2
Tracy	3	3	0
Available	2	3	1

Robin's resources (6/7, 0/1, 0/1) plus the free resources (2, 3, 1) yields more than his or her maximal resource usage (7/7, 1/1, 1/1). This allows Robin to run to completion, yielding (8, 3, 1).

Sam's resources (0/5, 2/2, 2/2) plus the free resources (8, 3, 1) yields more than his or her maximal resource usage (5, 2, 2), allowing Sam to run to completion. This yields the free resources (8, 5, 3).

Tracy's resources (3/4, 3/6, 0/2) plus the free resources (8, 5, 3) yields more than his or her maximal resource usage (4, 6, 2). This allows Tracy to run to completion, resulting in an end free resource count of (11, 8, 3).

This state is safe because there exists a safe sequence (Robin, Sam, then Tracy) which the chefs can follow to allow all of them to complete.

(Continued on next page)

2 Alerting threads (6 pts.)

Consider the "Sim Ciy disaster daemon" as depicted in code found on slides 17–19 of the "Synchronization #3" lecture.

What if somebody accidentally replaced one (or more) of the `cond_broadcast()` *calls with* `cond_signal()`*? Obviously "some thread could be awakened late." But the situation is more troublesome than that! It is possible that some thread might essentially never complete its "wait for triumph" loop (as coded on slide 17).*

2.1 5 pts

Use the tabular trace format found in the lecture slides to show what could go wrong. If you can, change just one of the `cond_broadcast()` *calls to* `cond_signal()` *(make sure it is clear which one you are breaking) and show the "essentially never" case rather than the "late" case.*

In this solution, `cond_broadcast(&new_year)` has been replaced with `cond_signal(&new_year)`. In addition, for formatting purposes `scenario_lk` has been renamed `s_lk`.

It can be observed that `cond_signal(&new_year)` will only be called 100 times. Consequently, if more than 100 threads attempt to wait on `&new_year` there will be threads which will never be woken up. Note that if some disaster daemon starts up after the progression of some time, some threads may get stuck even if 100 daemon threads are not used.

Execution Trace

time	Thr 0	Thr 1	...	Thr 100	Time Thread
0	`mutex_lock(&s_lk)`				
1	`wait_on()`				
2	`return &new_year`				
3	`cvarp = &new_year`				
4	`cond_wait(cvarp, &s_lk)`				
5		`mutex_lock(&s_lk)`			
6		`wait_on()`			
7		`return &new_year`			
8		`cvarp = &new_year`			
9		`cond_wait(cvarp, &s_lk)`			
...			Same		
a+0				`mutex_lock(&s_lk)`	
a+1				`wait_on()`	
a+2				`return &new_year`	
a+3				`cvarp = &new_year`	
a+4				`cond_wait(cvarp, &s_lk)`	
a+5					`cond_signal(&new_year)`
a+6	`wait_on()`				
...					A year passes
a+b+0					`cond_signal(&new_year)`
a+b+1		`wait_on()`			
...					At year 1999
a+b+c+0					`cond_signal(&new_year)`
a+b+c+1					`cond_signal(&new_month)`
a+b+c+2					`cond_signal(&new_day)`
a+b+c+3					`cond_signal(&new_hour)`
a+b+c+d					... All loops are done

Another solution can occur when `cond_broadcast(&new_month)` has been replaced with `cond_signal(&new_month)`. Imagine that there are always 12 daemons waiting on `new_month` (we will explain how this can occur later). It is possible for one of these daemons to always be woken up by a `cond_signal(&new_month)` before April. For concreteness, let us say that the current month is January. When this daemon calls `wait_on()`, it will return `&new_month` since `m < 4`, so the daemon will once again wait on `&new_month`, and again be woken up in January. This can occur indefinitely, causing this daemon to be woken up on January 1904, January 1905, January 1906, and so on. `cond_signal(&new_month)` enables this behavior because it can only wake up one daemon per month by taking it out of the queue, and if there are 12 daemons, our unlucky daemon can end up in a periodic cycle like this one. Ways for there to be always 12 daemons include:

- When a daemon is woken up and sees the correct month, a daemon forks.
- There could be more arbitrary users of the `&new_month` condition variable, such as a thread that tries to display the current month in a game UI; such a thread would naturally loop around waiting on `&new_month`.

The restriction on always having 12 daemons is not exact - there is some wiggle room.

Finally, note that there is a serious structural race condition in this code which is unrelated to `cond_broadcast()` versus `cond_signal()`: there is no way for a `cond_broadcast()` (or `cond_signal()`) to be delayed until all the threads from previous invocations have been woken up and had an opportunity to run. That is, the "clock thread" may broadcast one event and then move on to broadcasting another event before all threads have even become aware of the first event. A more-careful simulation would need some way for the "clock thread" to know when it is safe to move on to the next time step.

2.2 1 pt

The bug described above suggests that perhaps programmers should always use `cond_broadcast()` instead of `cond_signal()`. Is this true? Explain.

Programmers should **not** always use `cond_broadcast()` instead of `cond_signal()`! Generally speaking, as long as threads blocking on a condition variable are aware of "Paradise Lost" and "`if` versus `while`," it is "narrowly correct" to replace `cond_signal()` with `cond_broadcast()`. However, waking up extra threads is expensive (often system calls will be needed to wake up the thread and to block it again, plus there will be inter-processor cache-line costs). Also, repeatedly randomizing the order of multiple threads waiting for the same thing could lead to starvation.

Fundamentally, `cond_broadcast()` should be used when the code legitimately *wants* to wake up multiple threads. Sometimes, if threads of different "roles" are waiting on a single condition variable, it may be necessary to wake everybody up to ensure that at least one thread of the right "role" is reached. However, this often indicates that the code's use of condition variables should be re-designed.

Note

Some questions on this homework assignment were probably "too easy" compared to most exam questions. But hopefully the assignment served its purpose, which was to focus you on working through some examples, as a warm-up or inducement to look at old exams.

Also, hopefully it provides some guidance as to what we consider to be a clear execution trace. *Clarity—providing easy-to-evaluate evidence of your claim—is very important.* Please look over your traces and verify that they are at least as easy to read as ours. Note that in most cases it is necessary to show more lines of code (i.e., it is often not possible to collapse entire loops into a single line). But sometimes it is... abbreviation and other notational devices are fine as long as it is clear to the reader that you understand the execution flow. *On an exam, if you are arguing that a trace shows that an execution pattern can repeat, be sure to show exact repetition!*