Solutions
15-410, Spring 2007 Homework Assignment 1.

1 Stack Madness (10 pts.)

This setup is halfway between a real stack and genuine heap-allocated frames. In some sense it suffers from an ISA with too few registers, since it would be nice to have one register pointing to parameters in the calling frame and another pointing to locals in the called frame. But having a "more real" setup would have made the code more complicated.

1.1 Entry protocol (7pts)

```
.globl frametops
.globl framenum

    ###########################################################################
    # "function entry protocol" (use only caller-save registers at first)
    # N.B. "There is more than one way to do it"
    ###########################################################################

    movl framenum, %eax             # value of C's framenum
    movl frametops(,%eax,4), %eax   # value of C's frametops[framenum]
    movl %esp, (%eax)               # save caller's %esp
    movl %eax, %esp                 # assume new %esp, and...
    pushl %ebp                      # ...save caller's %ebp
    movl %esp, %ebp                 # %ebp ready to address our locals
    subl $1, framenum               # your soln might have this earlier
                                    # depending on your interpretation

    ###########################################################################
    # compiler or human inserts function body here
    # note clever omission of references to params
    ###########################################################################
```

1.2 Exit protocol (2pts)

```
    # "function exit protocol" (don't whack %eax)
    addl $1, framenum
    movl %ebp, %esp                 # point to caller's %ebp
    popl %ebp                       # restore caller's %ebp
    popl %edx                       # obtain caller's %esp...
    movl %edx, %esp                 # ...and assume it...
                                    # (I wonder what "popl %esp" does?)
    ret                             # CALL saved ret addr on caller's stack
```

1.3 Restrictions (1pt)

No function can have more than 4088 bytes of local variables and scratch space (4084 if the function calls another).

2 Get to work! (10 pts.)

2.1 How to add preforking to thrgrps (9pts)

The basic idea is for `thrgrp_prepare()` to pre-`thr_create()` a bunch of threads which enqueue themselves inside the threadgroup so that `thrgrp_create()` can give the one at the head of the queue its `func` and `arg`

and start it running.

Probably the "approach of least typing" involves:

- To the `thrgrp_group_t` struct add a "prefork queue" analogous to the existing "zombie queue." The prefork queue will be protected by the same mutex currently covering the rest of the struct. Elements of the queue will be the `thrgrp_tmp_data_t` arm of the union, with the interesting fields (`func` and `arg`) not yet filled in.
- Also add to the `thrgrp_group_t` struct a semaphore, initialized to zero, which tracks the number of entries at the head of prefork queue which contain valid `func` and `arg` fields.
- The number of *blank* elements in the queue will be tracked by an int, `pf_nblank`.

The system operates as follows:

- The `thrgrp_prepare()` function locks the thrgrp, adds `n` "blank" prefork blobs to the tail of the prefork queue, adds `n` to `pf_nblank`, and unlocks the thrgrp. It then uses `thr_create()` to launch `n` threads running `thrgrp_prefork_bottom(&eg)` (that is, the threads receive a pointer to the threadgroup they have been pre-forked into).
- The `thrgrp_prefork_bottom()` function `wait()`s on the semaphore, locks the thrgrp, removes the prefork element at the head of the queue, and unlocks the thrgrp. Then it can invoke the existing `thrgrp_bottom()`.
- Now `thrgrp_create()` can operate as follows. First, lock the thrgrp mutex. If `pf_nblank` is positive, the prefork queue contains at least one not-filled-in element (and a corresponding thread either has already slept on the semaphore or will "soon"). Fill in the first blank queue element, decrement `pf_nblank`, drop the mutex, and signal the semaphore. If `pf_nblank` is less than one, there are no blank prefork elements on the queue, so drop the thrgrp mutex and just spawn a thread the old way.

Note that this solution is less than optimal in the sense that the semaphore provides thread queueing but does not in parallel queue the data the threads need to operate, so two queues exist when one could suffice. Semaphores are like that.

2.2 A further extension? (1pt)

It is possible for `thrgrp_bottom()` and `thrgrp_join()` to cooperate so that logically-exiting worker threads don't actually exit but instead return themselves to the prefork queue. This would add noticeable complexity, since one might wish to cap the size of the prefork queue and would regardless need to handle the case of destroying a thread group which still contained invisible threads (so far we have assumed that a call to `thrgrp_prepare()` will be followed by an appropriate number of `thrgrp_create()` calls, so that `thrgrp_destroy_group()` could include a check that the prefork queue is empty instead of needing to deal with cleaning up preforked threads).

3 Plumbers (4 pts.)

3.1 First state (2 pts)

This state is not safe.

Given the resources currently held by each plumber and the resources currently unallocated, *no* plumber can assuredly run to completion. In particular, no hammers are available, which means that any plumber requesting a hammer will sleep. But since each plumber's pre-declared maximum usage includes one more hammer than he or she currently holds, every plumber is allowed to make a sleep-inducing hammer request. If they all do, a deadlock will result. Since there is no "progressor," there can be no safe sequence, so the state is unsafe.

Observe that the system is *not* deadlocked and would not necessarily enter a deadlock even if two of the plumbers went to sleep. It is perfectly possible for two plumbers to request hammers and go to sleep but for the third plumber (which would need to be Bob or Cameron) to finish the current job and relinquish all tools.

3.2 Second state (2 pts)

This state is safe (even though no resources are currently free!) because Bob has all of his resources and can complete his current job without sleeping. Bob's current holdings are enough for Cameron to acquire a full set of tools, complete a job, and release tools. Alice's situation is interesting—on the one hand, she already has a full set of tools and can complete her job without sleeping. On the other hand, her set of tools isn't enough to satisfy Cameron, and doesn't help Bob, so her completion doesn't help us find a safe sequence.

One safe sequence is "Bob, then Cameron, then Alice," so the system is in a safe state (there are other safe sequences, but we need only one to constructively prove safety).