15-410, Spring 2006, Homework Assignment 1.

Due Wednesday, March 1, 6:59:59 p.m.

Please observe the non-standard submission time... As we intend to make solutions available on the web site immediately thereafter for exam-study purposes, please turn your solutions in on time.

Homework must be submitted in either PostScript or PDF format (not: Microsoft Word, Word Perfect, Apple Works, LaTeX, XyWriter, WordStar, etc.). Submit your answers by placing them in the appropriate hand-in directory, e.g., `/afs/cs.cmu.edu/academic/class/15410-f05-users/$USER/hw1/$USER.ps`.

1 Academic Fare (35 pts.)

Consider the following variant of the dinining philosophers:

Four philosophers are seated at a table. Each philosopher requires two forks to eat, and all of the philosophers want to eat. Due to budget cuts at their university, there are only four forks. The university has decided to manage this problem by hiring a dean to delegate the forks.

To eat, a philosopher asks the dean for the first fork and waits to receive it. The philosopher then asks the dean for a second fork and waits to receive it. The philosopher then uses the two forks to eat. The philosopher then returns the second fork, and finally returns the first fork. After this the philosopher begins the cycle again. Philosophers will always advance to the next step in this pattern in a finite amount of time (no philosopher will eat forever, and no philosopher will retire while holding a fork).

The dean wants to avoid deadlock (a case when no philosopher can eat because all of the philosophers are waiting for forks). To expedite the fork allocation the dean keeps only the following records:

1. The number of forks held by the dean
2. The number of forks held by each philosopher

The dean runs in two modes: abundant forks and constrained forks. Forks are abundant if the dean holds enough forks that he can assign a fork to any philosopher without creating a deadlock. In this mode the dean will assign a fork to any philosopher who requests it. In all other cases forks are constrained, and the dean will assign forks only in ways guaranteed not to create deadlocks.

1.1 5pts

Which of the three deadlock coping strategies is the dean using?

1.2 5pts

How many forks must the dean hold when a fork request arrives in order for the request to be handled in abundant-fork mode?

1.3 5pts

If forks are not abundant, what additional check must the dean perform before giving a fork to a philosopher?

1.4 10pts

The code below implements the system described above, with the added restriction that philosophers request specific forks. While answering this question you may assume that the requests for specific forks do not change the validity of the allocation policy outlined above. What code would you need in `allocate_ok()` to implement the policy?

1.5 10pts

Assume a working implementation of `allocate_ok()`. Is the call to `mutex_avail()` needed? If so, show how removing it would lead to a deadlock. If not, make an argument that removing `mutex_avail()` creates a deadlock-free system. Be certain to phrase your answer in terms of the four ingredients needed for deadlock.

The code follows:

```
/*
* Where:
*
*   random behaves like the C library function
*
*   mutex_lock and mutex_unlock follow the specification from project 1.
*
*   mutex_avail returns a non-zero value if and only if the mutex is not
*   currently locked.
*
*/

#include "mutex.h"
#include "thread.h"

#define NUM_PHIL 4

typedef struct {
  mutex_t *left;
  mutex_t *right;
} forkpair_t;

void fork_lock(mutex_t *m, int *forks_held);
void fork_unlock(mutex_t *m, int *held);

void * philosopher(void *f_v)
{
  forkpair_t *f = (forkpair_t *)f_v;
  int forks_held_by_me = 0;

  while(1) {
    if (random() & 1) {
      fork_lock(f->left, &forks_held_by_me);
      fork_lock(f->right, &forks_held_by_me);
    } else {
      fork_lock(f->right, &forks_held_by_me);
      fork_lock(f->left, &forks_held_by_me);
    }
    dine();
    fork_unlock(f->left, &forks_held_by_me);
    fork_unlock(f->right, &forks_held_by_me);
  }
  return 0;
}

/* The dean is represented by the code below: allocate_ok, fork_lock,
  * and fork_unlock
  *
  * There is no dean thread. The philosopher that holds the dean_mutex
  * is temporarily the dean.
  */
```

```
int allocate_ok(int forks_held_by_this_phil, int forks_held_by_dean)
{
   /* Need to return 1 if the fork can be allocated without creating a
    * deadlock, 0 if it would create a deadlock */
   /* Note that we do know that the fork in question is unlocked */
}

static mutex_t dean_mutex;
int forks_allocated;

void fork_lock(mutex_t *m, int *forks_held_by_phil)
{
  int waiting = 1;

  while (waiting) {
    mutex_lock(&dean_mutex);
    if (mutex_avail(m) && allocate_ok(*forks_held_by_phil, NUM_PHIL -
forks_allocated)) {
      mutex_lock(m);
      *forks_held_by_phil++;
      forks_allocated++;
      waiting = 0;
    }
    mutex_unlock(&dean_mutex);
  }
}

void fork_unlock(mutex_t *m, int *forks_held_by_phil)
{
  mutex_lock(&dean_mutex);
  *forks_held_by_phil--;
  forks_allocated--;
  mutex_unlock(m);
  mutex_unlock(&dean_mutex);
}

main()
{
  mutex_t forks[NUM_PHIL];
  forkpair_t pairs[NUM_PHIL];
  int i;

  mutex_init(&dean_mutex);
  for (i = 0; i < NUM_PHIL; i++) {
    mutex_init(forks + i);
    pairs[i].left = forks + i;
    pairs[i].right = forks + ((i + 1) % NUM_PHIL);
  }
  for (i = 0; i < NUM_PHIL; i++) {
    thr_create(philosopher, pairs + i);
  }
}
```

2 "Six of One..."? (10 pts.)

According to the stack discipline we are following in this class, C function parameters are pushed onto the stack last-first: if we are calling `foo(a, b, c);` the value of `c` is pushed onto the stack, then the value of `b`, and so on.

While the C language specification allows parameters to be pushed in either order, the order we are using turns out to be very convenient for particular functions, including one you used in Project 0. Identify the function and explain how it benefits.

3 e-Commerce (10 pts.)

Consider the example posed on slide 14, "Commerce," of the first synchronization lecture, 8b. How many final values of `store->cash` can you come up with via various interleavings of Customer 0 and Customer 1? Show the "most interesting" two cases in tabular form (as exemplified by slide 26, "Mutual Exclusion," of that lecture), and briefly summarize the other cases.