COLBY COLLEGE
Computer Science Department

**CS357** **Allen B. Downey**
**Spring 1998**

**Assignment 4: Multi-threaded Programs**

**Due:** Tuesday 14 March

The purpose of this assignment is to practice writing multi-threaded applications, and to check out the behavior of unsynchronized access to a shared variable.

### Run the program

1. Pick up the pthread program from

   `http://rocky.colby.edu/cs357/code/hw4/pthread.c`

   Study it carefully and print the man pages for the new system calls.

2. Write a Makefile for the program. In order to use the `pthread` library, you have to tell the compiler to link to is, using the `-l` option:

   ```
   gcc -o pthread pthread.c -lpthread
   ```

   Compile and run the program.

### Find out where the segments are

1. You can find out where the various segments of memory reside by creating variables in each segment and printing their addresses. To get the address of a variable, use the `&` operator. For example,

   ```
   int i;
   printf ("0x%x\n", &i);
   ```

   creates a local variable named `i` and prints its address in hexadecimal (the 0x prefix is the standard way to indicate that a number is hexadecimal).

2. Find the location of each of the following: the program text, the static data segment, the heap, the parent's stack and the child thread's stack. Draw a diagram of the address space. How big is the address space?

3. Do the stacks grow up or down? Does the heap grow up or down? Can you tell how much space the system left for the child thread's stack to grow?

4. Modify the program so it creates an additional child that also uses the procedure `entry` as its point of entry. Where in the address space is the second child's stack? See how many children you can create before you run out of space (or maybe run out of something else first. If `pthread_create` fails, use the return value to figure out what went wrong).

**Tug of war!**

In class, we will be discussing a race in which one thread tries to increment a variable while the other tries to decrease it. Let's say we start with the variable at $n$. We declare victory if the variable ever gets to 0 or $2n$.

1. Modify your program so that it conducts this race. Of course, if $n$ is small, you would expect whichever process runs first to win. But if it takes more than one quantum to iterate $n$ times, then there might actually be some back and forth. In fact, if the CPU scheduler strictly alternates, the program may never terminate.
2. Try out some different values of $n$ and see how long the program takes. I am not sure whether this will work or not, but it is possible that we will be able to estimate the quantum this way (or at least guess what the order of magnitude is).

**Crash the stacks**

1. Put one of your threads into an infinite loop like this:

```
int i = 712;
while (i == 712) {
    /* do nothing */
}
printf ("Mo-o-m!  One of my siblings just crashed my stack!!!");
```

2. Write a recursive procedure, like factorial, that contains a really big array as a local variable. That way, each activation record is really big, and you can control the depth of the stack by passing different parameters to the procedure.
3. Using your map of the address space and the size of the local array, figure out how many activation records it will take to crash into the first thread's stack. Run the program and see if you got it right.

**What to turn in**

Just like last time, I would like a single report from each group, including an explanation of each experiment you ran, with exerpts from your code included in the text of your report wherever you think they facilitate the explanation. In other words, don't just staple a bunch of code to the end of your report—use it to explain what you did. Also, feel free to use pseudocode.

As usual: Label your axes. Explain every number. Use more words than numbers. Use more pages for words than for graphs or code.

Have fun! Learn something!