Homework 3: Multi-threaded Programs

cs341
Spring 2002

Allen B. Downey
Computer Science Department

Due: Monday 25 February

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. Read the handout from *Programming with POSIX Threads*.
2. Pick up the pthread program from

   `http://rocky.wellesley.edu/cs341/code/hw03/pthread.c`

   Study it carefully and read the man pages for the new system calls.
3. Compile the program with the following command:

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

   In order to use the `pthread` library, you have to tell the compiler to link it, using the `-l` option.
4. Play around with the `sleep` and `sched_yield` functions and figure out ways to control the order in which threads execute.

Experiment 1

As the program is written, it seems to show that if the parent makes a change in the heap, the child sees the change. What happens if the child runs first? What happens if both the parent and the child make a change? What if the parent and the child both allocate space in the heap? Can we conclude that the same heap is shared by parent and child?

Perform similar experiments to determine whether the text, static and stack segments are shared. Assuming that the stacks are not shared, which is what we expect, can you figure out where in virtual memory the child stack is, relative to the parent stack?

Experiment 2

Create more than one child thread and figure out where in virtual memory their stacks are. We would like to know whether the stacks of the parent and the children are in the same address space. If there are two children whose stacks are near each other, we might be able to crash their stacks.

1. Put one of your threads (the one with the lower stack address) 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!!!");
   ```

   This program will loop forever unless another thread somehow changes the value of `i`.

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.

Tug of war!

1. Create a variable named `n` that is shared between two threads and that is initialized to 100. Write a loop in one thread that increments `n` and that exits if `n` gets to 200. Write a loop in the other thread that decrements `n` and exits if `n` gets to 0.

   If the initial value of `n` is small, you would expect whichever process runs first to win. But if it takes more than one quantum to win, then there might actually be some back and forth. In fact, if the CPU scheduler strictly alternates, the program may never terminate.

2. Run the program several times and see who wins.

Experiment 3

Measure how long it takes for one of the threads to win the tug of war. Can you tell if there was any back and forth, or did one of the threads run first and win before the other got to play? You might be tempted to add print statements to the code, but system calls like that are going to mess up the timing.

What happens if you increase the initial value of `n`? Does the run time increase, suggesting that there is more back and forth?

Try adding some busy work to one or both of the threads so that each iteration of the loop takes longer. Can you get the program to run forever? What if you use `sched_yield` to force the threads to alternate?

Have fun!