Objective: In this recitation, you will learn about software performance on multicore systems.

1. Introducing the ‘worker’ program

Download the worker.c program and compile it:

```
mkdir ece250-rec14
cd ece250-rec14
wget http://people.duke.edu/~tkb13/courses/ece250/recitations/resources/worker.c
g++ -g -o worker worker.c
```

This program uses some C features we haven’t seen yet: it uses the `alarm` system call to request the kernel to send a `signal` (a kind of message that acts like an interrupt for a user program) every one second. When not handling this signal, the program performs some fake “work”, counting how much it can do before the next signal. At each signal, it prints out how much work it was able to do over the last second. In other words, it performs constant computation and prints periodically how much it’s able to get done over time.

Take a look at the source code, then try running it.

The program will run forever, so use Ctrl+C to stop it. How much work per second does it get done on average.

2. Multicore performance

On an otherwise idle computer, the kernel will schedule processes on the CPU’s various cores so that each process can get as much done as possible. If there are more tasks wanting to run than there are cores, then the tasks have to share a core over time (e.g. one task running for 1 ms, then a different task for 1 ms, etc.).

Using this property, we can determine how many cores are on a system just by how it performs as we add additional processes.

We need to open lots of terminal windows. If you’re doing a remote-login over SSH, enable X-Windows forwarding (see recitation 1) and launch multiple xterm sessions in the background (by typing “xterm &” repeatedly). (NOTE: You cannot just SSH to login.oit.duke.edu multiple times, because each time you connect to that, you are directed to a different actual computer in order to balance the load.) If on a local system with g++ (e.g. Mac/Linux), you can just run lots of terminals.
In the first terminal, run worker. Note the performance. Now run another worker in another window, and another, etc. It should look something like the figure below.

Keep doing so until you see a major change in performance for one or more workers. Assuming the system is otherwise idle, the number of workers it could sustain without performance degradation is the number of cores on that system.
While this is running, in another terminal, run `top`, which tells you about running processes and resource usage on the system. You’ll see something like this.

Under the CPU breakdown at the top, “%us” means percentage of time spent running user programs. The other fields are %system (kernel), %idle (waiting for work), %wait (waiting for IO device), and %hi/%si/%st (interrupts and other stuff). In the screenshot above and in your system, you should see almost all CPU time going to run our user processes (%us should be about 100%).

You can toggle the appearance of each individual core in the summary by pressing ‘1’. Doing so, you can verify your guess as to the CPU core count. You can also confirm the CPU core count by looking at the number of entries in `/proc/cpuinfo`.

3. **Multicore doesn’t help non-parallel programs.**

Kill all the worker processes, then run just one worker. Note how it gets the same individual performance as before. Run top, and observe how just one CPU core is busy, and the other are idle. Herein lies the key truth: **multiple cores do nothing to accelerate most software!** A program must be written to run as multiple threads of execution working together in order to take advantage of multiple cores. If not, then the best a multicore system can do is run multiple separate programs in parallel.