# Laboratory Exercise 6

cpe 357 Fall 2010

Due by (or before) 11:59pm, Friday, November 12th.
The program is to be done individually.

## Laboratory Exercises

This is a pared-down version of what would have been Asgn5 had the quarter not been abbreviated by furloughs. I've published the full version as "Asgn X" if you'd like to do that instead. What I've removed is all the terminal handling, because we probably won't get to cover that in class.

1. There's one task for this week: Write a program, `timeit`, that acts as a simple count-down timer using the interval timer to keep track of the passage of time.

   The timer takes a single command-line argument indicating the number of seconds to count down and counts down to zero in half-second increments printing "`Tick...`" on odd half-seconds and "`Tock\n`" on even ones. When time's up it prints "`Time's up!\n`" and ends. Since an example is worth a whole mess of words, consider the following example:

   ```
   % ./timeit 2
   Tick...Tock
   Tick...Tock
   Time's up!
   %
   ```

   Note that the last "`Tock`" will come immediately before the "`Time's up!`"

   **Details**

   - If the argument is missing, not a number, or less than zero, `timeit` should print a usage message and exit with nonzero status.
   - `timeit` must use `setitimer(2)` to generate the `SIGALRM`s for each clock tick. Because the display requires half-second accuracy, the granularity of the timer must be finer than one tick per second. There's really no reason to tick faster than that, though.
   - Be a good computing citizen: Don't just spin while waiting for signals. Use `pause(2)` or `sigsuspend(2)`.

   **Tricks and Tools**

   You should probably look into the functions and system calls in Figure 1.

   Debugging a program with asynchronous events can be very tricky. Remember also that while debugging, you can also generate SIGALRMs via `kill(1)` rather than from the timer.

   **Caution:** If you are working remotely you may not see the behavior you expect because remote operation adds another layer of potential buffering. This is particularly true when using ssh, because ssh uses block-ciphers for its secure link. Plain old-fashioned net-lag can

| | |
|---|---|
| `getitimer(2)`<br>`setitimer(2)` | for getting and setting the system interval timer |
| `sigaction(2)` | the POSIX reliable interface for signal handling |
| `sigprocmask(2)` | for manipulating the blocking or unblocking of particular signals |
| `sigemptyset(3)`<br>`sigfillset(3)`<br>`sigaddset(3)`<br>`sigdelset(3)`<br>`sigismember(3)` | for manipulating signal sets used by `sigaction(2)` and `sigprocmask(2)` |
| `atoi(3)`<br>`strtod(3)`<br>`sscanf(3)` | for checking on the command-line arguments |
| `fflush(3)` | to flush a stdio stream |
| `pause(2)` | Suspend execution until a signal is delivered |
| `sigsuspend(2)` | Temporarioy replace the procmask and suspend execution until a signal is delivered |

Figure 1: Useful system calls and library functions

have the same effect, though. If you experience bursty behavior, you may want to go to the lab in person and work at the console to be sure you are seeing what you think you are seeing.

Also, **remember that stdio is buffered.** If you write something to the console with `putchar()` or `printf()`, it may not appear immediately. You should either use unbuffered writes, or flush the output stream after writing.

## What to turn in

**For the Laboratory Exercises:** Submit the program described above to the `lab06` directory of `pn-cs357` via handin.

### Sample Runs

Since the execution of the program evolves over time it's difficult to give the full effect. I've included some examples below, but I will also place an executable version on the CSL in `~pn-cs357/demos` so you can run it yourself.

```
% timeit 23skedoo
23skedoo: malformed time.
usage: timeit <seconds>
% timeit one hour
usage: timeit <seconds>
% timeit 3
Tick...Tock
Tick...Tock
Tick...Tock
Time's up!
%
```