This lab concerns timing C and C++ programs and getting values for both user and system times. Note that user time is the same as virtual time and that profiling time is the sum of the user and system times. The real time is not very useful because it depends too much on machine load.

1. **First method:** Run your program using the “wrapper:” `/usr/bin/time -p`. This will run your program using the system calls fork() and execve() and then print out the real, user, and system times after the program ends. Note that the -p option is recommended so that you get the timing information in Posix format. If your program requires user input you can use a Unix pipe. For example, if you want to do a quicksort on the large dataset datafile6.txt you could type:

   sorting datafile6.txt

and manually enter the keystrokes “q” for quicksort and “x” to quit. But, you could time this as one atomic operation as follows:

   echo "q x" | /usr/bin/time -p sorting datafile6.txt

The format of the full command line in general will be:

   echo "{keystrokes}" | /usr/bin/time -p {command with any parameters}

**Note that:**

i. times reported by the operating system are usually only accurate to ±5–10%, depending on the system, so you may have to do several runs and take an average if you need better accuracy.

ii. if the program executes with a time under a hundredth of a second you will get 0.00 as an answer, so this method is not useful for quickly executing programs.

Time all of the different sorting options available in the sorting program on the large dataset datafile6.txt. Are the large variations in time you get surprising?

2. **Second method:** Set timers in your programs, using the timing system calls setitimer(), getitimer(), etc., or the newer POSIX calls timer_create(), timer_settime(), etc. To say that these calls are complicated is an understatement but I have written the C module timing.c with header file timing.h which you can use and which is much simpler. To see an example of this, get the sample program cache.c. Look at the source code, particularly the lines:

   #include "timing.h"

   ...
   init_timing(); /* initialize and start timers */
   ...
   read_timing(); /* get the results */
   pause_timing(); /* stop the timers */
   ...
   fprintf_timing(); /* write the results to stdout or file */
   ...

Compile the program (with the timing module) either via:

   gcc -g cache.c timing.c -o cache

or by using the makefile:

   make cache
make clean
This program tests the effective speed of the L2 cache by reading and writing a large amount of information to an array using a particular line_offset. It then uses the timing results to compute the net memory bandwidth in megabytes per second for that line_offset. Run it, trying some numbers from 127–4096 and see if you can discern a pattern.

Assignment Write a C or C++ program which will accept a positive integer n from 2–2000000 on the command line and which will display the first 20 primes greater than or equal to n. For an example executable, copy over cpbound from the class directory and run it, for example, with n=2000:
cpbound 2000
Time your program for values of n=2000, 20000, 200000, on up to 2000000000 using Method 1 (with /usr/bin/time -p) until you get non-zero time values. Of course, you can’t go much higher than 2000000000; why?
Email me your timings (user and system) for the values of n above and the path (and only the path) to your solution. For example, if your program is named lab2.c you might email me that
my solution is at: /usr/stu/myusername/cs312/lab2.c
Please:
  i. Keep line length under 80 characters per line.
  ii. Do not send any attachments.