This lab shows how to use the signal SIGALRM (with `alarm()`) together with `setjmp()` and `longjmp()` to set a timer to break out of a code routine (which may involve an indeterminate number of procedure calls) which is either taking too much time or blocking in I/O indefinitely. Signals are very much like the software version of an interrupt and are they delivered to a process by the operating system. `setjmp()` and `longjmp()` are C macros. They are not system calls since they allow one to restore a stack execution context which was previously saved and no intervention of the operating system is required to do this. The `typedef jmp_buf` in which the information (program counter, processor status flags, special and general purpose registers and floating point registers, and possibly signal mask information) is saved is usually an array of long words (you can check by looking at the header file `setjmp.h`).

1. Get copies of the sample program `volatile.c`. This program shows how `setjmp()` and `longjmp()` work as well as testing side effects on “register” and “volatile” variables. First compile the program via
   ```bash
gcc -g volatile.c -o volatile
```
and run it. Then compile the program with optimization
   ```bash
gcc -O2 -g volatile.c -o volatile
```
and run it. Why are the results different?

2. Get copies of the sample program `alarm.c` and the current version of the `Makefile` (or edit your old version for the new program `alarm.c`). Note that `alarm.c` doesn’t use our timing routines since it sets its own SIGALRM handler. Compile and link the program by typing:
   ```bash
make alarm
```

3. Run the program asking for different values of starting number and number of desired primes. You can run this faster using a pipe and try more values (e.g.):
   ```bash
   echo "'500000 36'" | ./alarm
   echo "'500000 360'" | ./alarm
   echo "'50000000 3600'" | ./alarm
   echo "'500000000 36000'" | ./alarm
   ```

4. Refer to a printout of the source code and read the introductory remarks. Note again that we run this program with reduced priority using `nice()`. Trace the code to see precisely how `setjmp()` and `longjmp()` are used, and how the signal handler `sigalrm_handler()` works.

Questions Answer the following questions:

1. If the function `find_next_primes()` had called another function `A` and if the function `A` had called another function `B` when the timer suddenly expired would
setjmp() and longjmp() still have worked? Why or why not?

2. Given what has to be saved, do you think the actual information put in jmp_buf by setjmp() is platform (i.e. processor and operating system) dependent? Why or why not?

3. How could the framework used by this program alarm.c be useful if you wanted to write a network request to another machine but you didn’t want to wait forever to see if there was going to be a reply to read?

Email me your explanation in plain ascii text observing the following:

i. keep line length under 80 characters per line and don’t send quoted-printable text (which is painful to read since each line ends in an ‘=’ sign and it is full of stupid =0A’s and =20’s).

ii. send the text in-line and not as an attachment (e.g. in pine you can use CTRL-R to read text into the email body).

iii. if you need to refer to files that you have in your account, just give me the path to the file and the filename. Don’t use spaces in filenames; you can use the underscore character to improve readability.