This lab concerns the run-time interface and how a program dialogues with the operating system.

1. Get copies of the test program `vmcputest.c` and the makefile in one of your subdirectories. First compile and link the program *with the standard C Run Time Interface libraries* by typing (note the ‘C’):

   ```
   make vmcputestC < cr >
   ```

   or compiling and linking directly via:

   ```
   gcc -g vmcputest.c -o vmcputest < cr >
   ```

2. Run the program in native mode first to see what it does. It is a typical CPU-bound program which will find the first 20 primes starting after the number you put on the command line (if you don’t give a number it uses `start_primes=75000`). Note that all the file and console I/O is done through seven system calls: `open()`, `read()`, `write()`, `close()`, `unlink()`, `lseek()`, and `stat()`. The include of `stdio.h` is only for the utilities `perror()` and `sprintf()`. We need `sprintf()` because `write()` does *no* formatting at all; it just outputs bytes.

3. It is best to do the next test *cooperatively* when helios is lightly loaded and it is best to have only one person running `top`. This program shows current system load. You stop it by typing ‘q’. What is important besides the list of the top CPU users is the row labelled “CPU states:”. In that row you want to pay attention to “% user” and “% kernel” values. CPU-bound programs should primarily elevate the user time; I/O-bound programs should primarily elevate the kernel time. Why? Run `vmcputest 150000` while watching the system with `top`. What happens?

4. Make the virtual library `libvm.a` and the virtual batch operating system `vm_batch`. You will need the source files `libvm.c`, `libvm.h`, and `vm_ops.c` in your current working directory.

   ```
   make libvm.a vm_batch < cr >
   ```

   Compile and link the test program with the virtual libraries by typing:

   ```
   make vmcputest < cr >
   ```

   or compiling and linking directly via:

   ```
   gcc -g vmcputest.c -o vmcputest -L. -lvm < cr >
   ```

   If you try to run this from the command line you will get an error message saying that `msgget()` failed since `init_libvm()` could not open a connection to a message queue created by the virtual batch operating system `vm_batch`. You must execute it *through `vm_batch*:

   ```
   vm_batch vmcputest 10000 < cr >
   ```

   You can, of course, vary the options on the command line of `vmcputest`. `vm_batch` makes some comments and dumps the GDT for the execution of the child process after it runs. `vm_batch` also allows command line options. Open another `xterm` and give the ‘tty’ command in it (suppose it returned `/dev/ttyp3`). Tell `vm_batch` to redirect its child’s I/O to the `/dev/ttyp3 xterm` by typing:

   ```
   vm_batch -tty3 vmcputest 10000 < cr >
   ```

   On a Solaris machine ‘tty’ will return something like `/dev/pts/10` so in this case type

   ```
   vm_batch -tty10 vmcputest 10000 < cr >
   ```

5. This week’s assignment is to familiarize yourself with the structure of `vm_batch.c` and `libvm.c`. Next week’s lab will have an assignment to turn in.