Lab 08 - Memtop

Out: November 19-20, 2012

1 Introduction

At some point in your Computer Science career, you may have run the shell command `top`, which displays a list of processes currently consuming the most system resources (by default it ranks them by CPU usage, but can be configured to rank them otherwise). In this lab you will be writing a critical component of something similar: a basic process monitor called `memtop`. Such a program can be quite useful when, for example, your computer is slowing down and you want to identify and kill the process that is hogging your system's memory or CPU time. The `top` program installed on the CS department machines has a lot of different features; this program implements a particular subset of those features.

2 Assignment

In this lab, you'll be improving `memtop`, a simple system monitor. It will monitor primarily memory usage rather than CPU usage.

2.1 Memtop Design

This system monitor tracks the following memory usage and process data of each process:

- `PID`: the integer identification number of the process.
- `USER`: the username of the process owner.
- `GROUP`: the group identification of the process.
- `VIRT`: the virtual memory usage of the process.
- `S`: the state of the process (running, sleeping, etc.)
- `RUNTIME`: the CPU time, in seconds, that the process has used.
- `COMMAND`: the name of the process.

As it stands, memtop cannot accept user input - it begins, sorts the processes by `PID`, and then loops forever. You will add the ability to get input and process input in an event loop (using the `select` system call) to continually check for user input, which will come in the form of single-character commands to the program. You will then pass these characters on to the `process_input()` support function, which executes the command corresponding to the character. These user commands will instruct the program to either quit (`q` and `Q`), or change the order in which processes are displayed in the terminal (by picking the field by which they are ranked). A table of such commands is given here:

`c`	Alphabetize by `COMMAND`, descending	`C`	Alphabetize by `COMMAND`, ascending
`g`	Sort by `GROUP`, descending	`G`	Sort by `GROUP`, ascending
`p`	Sort by `PID`, descending	`P`	Sort by `PID`, ascending
`q`	Quit	`Q`	Quit
`r`	Sort by `RUNTIME`, descending	`R`	Sort by `RUNTIME`, ascending
`s`	Alphabetize by `S`, descending	`S`	Alphabetize by `S`, ascending
`u`	Alphabetize by `USER`, descending	`U`	Alphabetize by `USER`, ascending
`v`	Sort by `VIRT`, descending	`V`	Sort by `VIRT`, ascending

In addition, the program will scroll down through the process list when its input is '`.`', scroll up through the process list when its input is '`,`', and scroll to the top of the process list whenever either the sort order is changed, or any other character is input.

Note that memtop displays its output using the Curses library, which is appropriate for a program of this nature.

3 Solving the Concurrency Problem

The memory monitor must continuously query the system for changes and update its display accordingly (this functionality is already present). However, it also needs to constantly be ready to accept user input, which could come at any time, which presents a problem: normally, a call to `read()` or `fread()` blocks until it reads new data, at which point it returns and the program can continue. If `memtop` is blocking while waiting for user input, it cannot simultaneously scan for updated process information[1]. You will be adding the code to get user input without blocking the program from updating its data.

3.1 Event-Driven Programming

Fortunately, there is a way around this problem: event-driven programming. Instead of simply reading from standard input and waiting for data to appear, we check whether there is data available at the top of each loop (referred to as an "event loop" here), and then read data if and only if there is data available to read. This allows the program to perform other tasks while waiting for input, interrupting those tasks to read input only when input arrives.

Many event-driven programs can be modeled by the following abstract pseudocode:

```
while true
        do check for user input
        if input has arrived then
                do <process input>
        end if
        <program body>
end while
```

[1]It is possible to do this, but not in a single-threaded program. Multi-threaded programs can use multiple threads to block on input from multiple sources, but doing so is unlikely to be the best design choice.

3.2 Using `select()`

The function you'll be using to check for user input at the top of your event loop is called `select`. This system call, its associated structs, and basic usage is fairly thoroughly described in the man pages of the departmental Linux machines if you want more information than is provided in this document.

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
     fd_set *exceptfds, struct timeval *timeout)
```

(Be sure to include `<sys/time.h>`, `<sys/types.h>`, and `<unistd.h>` to have access to the structs and functions necessary for this lab.)

The `select()` function checks on sets of file descriptors to see if they are ready for reading or writing. For `memtop`, we just want to check if `STDIN_FILENO` has data available to read (which means that `read()` will not block (most likely—see section 3.3)). This means that we need to set up `readfds` to be a `fd_set` containing the `STDIN_FILENO` descriptor, and that `writefds` and `exceptfds` can both be `NULL`.

To set up `readfds`, declare a `fd_set`, clear it using the macro `FD_ZERO(fd_set *set)`, and fill it using the macro `FD_SET(int fd, fd_set *set)` (note that multiple file descriptors could be added to the set using `FD_SET()`, but `memtop` is only interested in `STDIN_FILENO`). Be aware that calls to `select()` modify the contents of your `fd_sets`, so to produce correct behavior you'll have to reset your `fd_set` in each iteration of the loop before the call to `select()`.

Since we know what file descriptor we're interested in, we can easily determine what the first parameter, `nfds`. This parameter's value must be one more than the highest file descriptor in any of the sets `readfds`, `writefds`, `exceptfds`. Since we're using only one set and the numeric value of `STDIN_FILENO` is guaranteed to be 0, we can simply pass 1.

The final parameter to `select`, `timeout`, specifies how long `select()` should wait if no descriptors are ready when it is called. Since the point of using `select()` in our case is to avoid blocking, we can set both fields of a `struct timeval`, `tv_sec` and `tv_usec`, to 0. Note that on Linux this struct is also modified by a call to `select()`, so you'll have to re-fill it before every call to `select()` too.

Now you're finally ready to call `select`! Be sure to check if the return value is -1 to see if there was an error. If `select` completed successfully, it will return the number of ready file descriptors. When `select` indicates that `STDIN_FILENO` is ready, go ahead and call read - check for this using the macro `FD_ISSET(int fd, fd_set *set)` on `readfds` after `select()` has returned.

3.3 `read()`

Use `read()` to actually read the available input. You can read the data into `buf`, whose length is `BUFFER_SIZE`, both of which are already defined in the support code. When you successfully read data, call the support function `process_input()` with arguments `buf` and the number of bytes read. The function `process_input()` returns 0 unless it gets a character indicating that memtop should quit. Thus, if you get a non-zero value from `process_input()` you should break out of the while loop. Once you've done this, there is just one more error condition to handle.

Using an event loop prevents blocking in most situations. However, it is still possible for a `read()` to block in certain situations. Therefore, it is advisable to set the file descriptor from which you will read (in this case, `STDIN_FILENO`) to be *non-blocking*, such that it will not block even if there is no data available. The `fcntl()` function, declared in the header <`fcntl.h`>, allows you to do this. In general, this function performs a control operation on the file descriptor indicated by `fd`. You only need to call this function once for each control operation for each file descriptor. Then calls to `read()` under conditions where it would normally block will instead return `-1` immediately, setting `errno` to `EAGAIN` or `EWOULDBLOCK`.

`int fcntl(int fd, int cmd, int flags)`[2]

To set a file descriptor to non-blocking using `fcntl`, use `F_SETFL` as the `cmd` argument, and `O_NONBLOCK` as the `flags` argument. Given these arguments, this function returns 0 on success and -1 on error.

4 Getting Checked Off

Once your `memtop` program properly responds to user input and you've completed the appropriate error checking, you've finished the lab. Call a TA over to have them check your work. If you do not complete the lab during your lab session, you can get credit for it by completing it on your own time and bringing it to TA hours before the next week's lab. Remember to read the course missive for information about course requirements and policies regarding labs and assignments.

[2] This signature aptly describes how you will use this function. However, not all commands indicated by the `cmd` argument require `flags`, so that third argument is optional.