CS3204 Operating Systems - Spring 2001
Instructor: Dr. Craig A. Struble
Tracking Process Creation in Linux

Assigned: Thursday, Feb. 8 **Due:** 11:59.59 p.m., Friday, Feb. 23

1 Introduction

In the first programming assignment, you learned about using `fork` and `exec` to create processes in Linux. In this assignment, you will implement your own system calls for tracking the number of processes created and destroyed during the system `uptime`, the amount of time the system has been up since the last reboot. You will implement system calls to allow user processes to access the information your modified kernel trackes.

2 Specification

In this assignment, you will implement a single system call, `procstat` that returns the number of processes created or destroyed during the system uptime. The system call will take one integer parameter, `info`, which is 0 signifying that `procstat` should return the number of processes created, or 1, signifying that `procstat` should return the number of processes destroyed. If any other value is passed, `procstat` should return -1, signifying an error occured.

Using your newly created system call, you will write a small user program named `procs` that prints out the number of processes created or destroyed during a specified time interval. The command line for `procs` is

```
procs samples rate -c | -d
```

where `samples` is the number of samples to take, `rate` is the rate in seconds to sample process statistics, `-c` specifies that we want to track the number of processes created and `-d` specifies that we want to track the number of processes destroyed. One and only one of `-c` or `-d` is required as an argument. A usage message should be printed if the program is not executed with proper arguments.

The output of `procs` should be repeated lines of process creation/destruction statistics. Suppose that we take 3 samples of process creation every 5 seconds. An example of expected output is

```
Time	Created	Rate
5	27	5.40
10	33	6.60
15	3	0.60
```

where the first column is the elapsed time in seconds since starting the program, the second column is the number of processes created (or destroyed), and the third column is the rate per second at which processes were created during the last monitoring time (to two digits of precision following the decimal point). Each column is tab delimited, so they won't line up in all cases. When processes are being destroyed, the "Created" header should read "Destroyed".

Consider using `gnuplot` to plot your process statistics graphically.

3 Implementing System Calls

Recall that system calls are used to transfer execution from user-space code into kernel-space code. The code for system calls is executed while the processor is in supervisor mode. To accomplish this, Linux on Intel platforms generates an interrupt **0x80** to occur, with a parameter set to the system call number to execute. This system call number is an offset into the `sys_call_table`, the table of all system call entries (stored in `/usr/src/linux/arch/i386/kernel/entry.S`). In RedHat 6.2, the table is defined as follows:

```
.data
ENTRY(sys_call_table)
    .long SYMBOL_NAME(sys_ni_syscall) /* 0  -  old "setup()" system call*/
    .long SYMBOL_NAME(sys_exit)
    .long SYMBOL_NAME(sys_fork)
    .long SYMBOL_NAME(sys_read)
    .long SYMBOL_NAME(sys_write)
    .long SYMBOL_NAME(sys_open)      /* 5 */
    ...
    .long SYMBOL_NAME(sys_sigaltstack)
    .long SYMBOL_NAME(sys_sendfile)
    .long SYMBOL_NAME(sys_ni_syscall)       /* streams1 */
    .long SYMBOL_NAME(sys_ni_syscall)       /* streams2 */
    .long SYMBOL_NAME(sys_vfork)            /* 190 */

    /*
     * NOTE!! This doesn't have to be exact - we just have
     * to make sure we have _enough_ of the "sys_ni_syscall"
     * entries. Don't panic if you notice that this hasn't
     * been shrunk every time we add a new system call.
     */
    .rept NR_syscalls-190
            .long SYMBOL_NAME(sys_ni_syscall)
    .endr
```

Entry 1 contains the address of the `exit()` system call, 2 is for `fork()`, and so on. Any entry labeled `sys_ni_syscall` is a system call that is not implemented.

3.1 System Call Table

To implement your system call, you will need to modify several files. First, your system call must be added to the system call table just shown. If your system call is to be named `sys_my_call()`, then you change the table in `/usr/src/linux/arch/i386/kernel/entry.S` to reflect the new call:

```
ENTRY(sys_call_table)
    .long SYMBOL_NAME(sys_ni_syscall) /* 0  -  old "setup()" system call*/
    .long SYMBOL_NAME(sys_exit)
    ...
    .long SYMBOL_NAME(sys_vfork)      /* 190 */
    .long SYMBOL_NAME(sys_my_call)    /* 191 */

    /*
     * NOTE!! This doesn't have to be exact - we just have
     * to make sure we have _enough_ of the "sys_ni_syscall"
     * entries. Don't panic if you notice that this hasn't
     * been shrunk every time we add a new system call.
     */
    .rept NR_syscalls-190
            .long SYMBOL_NAME(sys_ni_syscall)
    .endr
```

This allows a trap (interrupt 0x80) with an argument of 191 to invoke `sys_my_call()`. **Before modifying `entry.S` be sure to make a backup of the original file! You may need it to recover from errors.**

3.2 System Call Stub

Even though you have added an entry to the system call table, you still need to generate a stub so that a C function call will invoke the new system call. The stub generates code initiating a trap with the proper argument. To generate the stub, you should first edit the `/usr/src/linux/include/asm/unistd.h` file to add constant definition for your new system call.

```
#define __NR_exit                 1
#define __NR_fork                 2
#define __NR_read                 3
#define __NR_write                4
#define __NR_open                 5
...
#define __NR_getpmsg            188  /* some people actually want streams */
#define __NR_putpmsg            189  /* some people actually want streams */
#define __NR_vfork              190
/* #define __NR_ugetrlimit      191     SuS compliant getrlimit */
#define __NR_mmap2              192
#define __NR_truncate64         193
#define __NR_ftruncate64        194
#define __NR_stat64             195
#define __NR_lstat64            196
#define __NR_fstat64            197
```

The system call number 191 is commented out, so you can replace it with the constant definition for your new call.

```
#define __NR_my_call            191
```

Again, be sure to make a backup of this file before you edit it! Macros are available for generating system calls with zero to five parameters. For example, to generate a stub for a system call with two parameters, the macro has the form

```
_syscall2(type, name, type1, arg1, type2, arg2);
```

In this macro, `type` is the return value type, `name` is the name of the stub, `type1` is the type of the first parameter `arg1` and `type2` is the type of the second parameter `arg2`.

To generate the stub for `my_call` in your user program, you make the following macro call.

```
#include <linux/unistd.h>
...
/* Generate system call stub for int my_call(int x, int y) */
/* This is placed above your main function, in the file scope */
    _syscall2(int, my_call, int, x, int, y);
...
```

The function `my_call` is the function call you would execute to call the system call `sys_my_call`.

3.3 Implementing Your System Call

To implement your system call, the easiest thing to do is modify one of the existing kernel source files to add a function implementing the system call. In this assignment, `/usr/src/linux/kernel/fork.c` or `/usr/src/linux/kernel/exit.c` will be a good file to use. Whichever file you choose to modify, **make a backup of the file first!**

The function implementing the system call uses an additional modifier to the return type, `asmlinkage` to denote that the function is interacting with assembler generated code. Suppose that `sys_my_call` sums its two arguments and returns the result as an integer value. The complete system call is

```
asmlinkage int sys_my_call(int x, int y) {
    return x + y;
}
```

You can browse the kernel source for functions named `sys_*` to find the implementations for several more system calls.

Finally, you may want to print out debugging information along the way. The `printf` function and several other C library functions are **NOT** available for use in kernel code. Thus another function, `printk` is used to print out information in kernel code. The `printk` function behaves in the same manner as `printf` (i.e., it has the same parameters and uses the same formatting codes). You can look at the manual page for `printf` for more information.

4 Installing the Kernel Sources

If you do not have the sources in `/usr/src/linux` then you need to install them. You should obtain the following RPMs for RedHat 6.2 from a RedHat mirror:

- `kernel-headers-2.2.14-5.0.i386.rpm`
- `kernel-source-2.2.14-5.0.i386.rpm`

- `kernel-doc-2.2.14-5.0.i386.rpm`

If you are using a laptop, you will also want:

- `kernel-pcmcia-cs-2.2.14-5.0.i386.rpm`

You might be interested in debugging kernel dump files as well (should you cause a kernel panic). If you want this capability, you will also want:

- `kernel-utils-2.2.14-5.0.i386.rpm`

Each of these files can be installed on your system by using the command

```
rpm -ivh filename
```

as the `root` user.

5 Building and Running Your Kernel

The final step is to build and run your new kernel. The process I outline here assumes that you have Linux installed in its own partition and that you are using LILO (the Linux Loader) to boot your system. For those running partitionless systems or boot disks, I would like volunteers to help test building and installing instructions appropriate for those platforms.

Compilation of the kernel should be the same as below, but installing kernel and running LILO is not appropriate for boot disk or partitionless systems. Be careful!

I used this process myself on a laptop, and it generally worked, with the exception of one kernel module, `emu10k1.o`. This is a soundcard module and is not necessary. However, if anyone wants to help me identify the problem, please contact me.

With the following instructions, I have RedHat 6.2 set up to boot either the original kernel installed or the modified one I built. I encourage you to use this kind of setup so that you can recover in the case your kernel has problems. Some more information can be obtained in the `README` file in `/usr/src/linux`. I will only focus on what I did. If you're a Linux guru, I'd appreciate any corrections to the process below, but you can do your own thing when building your kernel. All commands are assumed run in `/usr/src/linux` as the `root` user unless otherwise specified.

5.1 Starting with a Clean Environment

It's a good idea to clean out any old object files from the kernel source directory by using the command

```
make clean
```

After cleaning the environment, you should modify the `Makefile` and edit the `EXTRAVERSION` variable so that you do not overwrite the original RedHat Kernel. I changed `EXTRAVERSION` to be

```
EXTRAVERSION = -os
```

to denote that this is my OS class kernel.

5.2 Configuring Your Kernel

The first step is to configure your kernel. There are several options for configuration, which will allow you to build a lean kernel: `make config`, `make menuconfig`, and `make xconfig`. I went the easy route and copied the kernel configuration used by RedHat, which is stored in `/usr/src/linux/configs/kernel-2.2.14-i386.config`. To use this, copy the file to `/usr/src/linux/.config` and then configure the kernel with

```
make oldconfig
```

This command uses the configuration file in `/usr/src/linux/.config` to configure the kernel. I also tried to use `/usr/src/linux/configs/kernel-2.2.14-i686.config` but this caused several module errors. Again, if someone can help me figure this out, I would appreciate it.

5.3 Making Source Dependencies

After configuring the kernel, you should make the source dependencies with either

```
make depend
```

or

```
make dep
```

Both commands do the same thing.

5.4 Compile the Kernel

The next step is to compile the kernel. This is accomplished with the command

```
make
```

5.5 Building Kernel Modules

The default RedHat kernel configuration configures several drivers as kernel modules: shared object files loadable by the kernel as needed. To build all of these modules, execute the command

```
make modules
```

You only need to do this the first time you build and install your kernel.

5.6 Installing the New Kernel

Once the kernel and kernel modules are built, you should install the kernel and modules in such a way that doesn't overwrite the original Linux kernel. Fortunately, RedHat 6.2 provides nicely set up Makefiles for installation. If you followed the directions above, the commands

```
make install
make modules_install
```

will install a new kernel and modules without overwriting the old ones.

5.7 LILO

Before booting your new system, you need to edit your LILO configuration file, which is stored in `/etc/lilo.conf`. You need to add an entry for your newly created kernel. I assume that you have followed my directions above, in which case the kernel you should boot for the OS project is `/boot/vmlinuz-2.2.14-os`.

My `/etc/lilo.conf` is

```
boot=/dev/hda3
map=/boot/map
install=/boot/boot.b
prompt
linear
default=linux
message=/boot/message

image=/boot/vmlinuz-2.2.14-5.0
        label=linux
        read-only
        root=/dev/hda3

image=/boot/vmlinuz-2.2.14-os
        label=os
        read-only
        root=/dev/hda3

other=/dev/hda1
        label=windows
```

which is set up to boot the original RedHat kernel, the kernel I modified (the second `image` entry), and Windows 98. This also is set up to display a message contained in the file `/boot/message` which is

```
Command         Operating System
-------------------------------
linux           RedHat Linux 6.2
windows         Windows 98
os              OS kernel

Type one of the commands above to boot the specified operating system.
```

Your own LILO configuration file may look very different. My suggestion is to copy the entry for the original Linux kernel and change it to load your modified kernel (don't forget to change the label!). You should only need to edit your file once, unless you make a mistake.

Each time you compile and install a modified kernel, you must run LILO to set up booting:

```
/sbin/lilo
```

5.8 Running the Modified Kernel

To run your modified kernel, simply type in the label you gave it (`os` in my example) at the LILO `boot:` prompt. To run the original kernel, simply type the label you gave the original kernel (`linux` in my example). By setting things up this way, you can recover by switching to the original kernel.

5.9 Boot Disks

If you make and install the kernel and modules as described above, the following command can be used to make a bootdisk for partitionless or other Linux setups that require a bootdisk:

```
/sbin/mkbootdisk 2.2.14-os
```

This will create a bootdisk that will boot the kernel `/boot/vmlinuz-2.2.14-os`, which should exist if you followed the instructions in the specification.

6 Hints

- The variable `nr_tasks` tracks the number of tasks existing in the system. You might want to see where this variable is updated.
- In `/usr/src/linux`, try executing the command `grep -r string .` to search for `string` in EVERY file in `/usr/src/linux`. You can find the previously mentioned variable by using this command.
- You will need to declare your own variable for tracking the processes created and destroyed. You may need an `extern` declaration to access your variable in several files.
- SCSI users may need to compile in SCSI support directly into their kernel instead of using Linux's module support.

7 Submission

We will use the Curator, `http://ei.cs.vt.edu/~eags/Curator.html` to collect program submissions. The URL for submission is `http://spasm.cs.vt.edu:8080/curator/`. Only the servlet interface to the Curator is supported. No grading will be done by the Curator.

You are to submit a single tarred (`man tar`) and gzipped (`man gzip`) archive containing

- A text file named `README` describing the changes you made to the kernel, how your user program handles sampling process statistics in the specified increments, and a description of each file included in the archive.
- The modified `entry.S`, `linux/unistd.h`, and `fork.c` or `exit.c` kernel sources
- The source file(s) for `procs`.
- Sample output from running `procs` at least 4 different times, two each for process creation and destruction. You should use different sample rates and lengths of time for each test run.

Your files must be contained in a directory named `project2` of the archive. Be sure to include only the files listed above. Do not include extra files from an integrated development environment such as `configure` scripts, automake related files, etc. This is primarily an issue if you are using KDevelop.

Be sure to include your name in all files submitted. **DO NOT** include executables or object files of any type in the archive. **Submissions that do not gunzip and/or untar will not be graded. Be careful to FTP in binary mode if you are transferring your file to a Windows machine before submitting to the Curator. Invalid submission format will receive a grade of zero (0)!**

8 Programming Environment

As stated in the syllabus, you must use Linux and `gcc/g++` to implement this project. You must use C to implement your system call, but C++ is acceptable for implementing `procs`. **All data structures used in your program must be student implemented. Using the standard template library (STL) or other third party libraries for data structure implementations is strictly prohibited.** Using C++ input and output streams and C++ strings is OK.

9 Acknowledgements

Portions of this exercise are synthesized from *Kernel Projects For Linux* by Gary Nutt.