

# CSIS0230A Principles of Operating Systems (Class A)

## Tutorial 4

### The making of a Linux system call

We will modify the kernel to make our own system call. Our focus is on the process needed to create the system call, so we make our call extremely simple to write:

**Specification.** Add an initially 0 counter (i.e., a global variable) to the kernel, which can hold a non-negative 31-bit number (i.e., with type `int`). There should be three system calls to manipulate it: `inccount` to increment the count, `deccount` to decrement it, and `getcount` to get the current value. Successful changes to the counter should also return the new value of the counter. If an attempt to modify the counter results in overflow (i.e., the resulting value would be negative), the system call should **not** modify the counter, but instead generating an error `EINVAL` (Invalid argument).

It is usually a good idea to start by compiling an unmodified kernel source, as that makes sure that if something went wrong after we modify it, the problem comes from our modifications. But the process (see the end of the tutorial worksheet) is quite time-consuming, and cannot fit into our limited tutorial time. So instead, the kernel is pre-compiled, and all files generated, including the intermediate files, are stored within the file `linux-2.4.20-8.tar.gz`. As we will see, this save us a lot of time because compiled files need not be compiled again. Follow the steps below to proceed.

1. Use `tar xzvf linux-2.4.20-8.tar.gz` to extract the files in the archive. You will find a new directory `linux-2.4.20-8` that contains the top level kernel source tree, compiled in advance. We will denote this directory as `<kroot>` from now on.
2. Create a new file `<kroot>/kernel/counter.c` in the kernel directory. Edit the file to implement the system calls as defined above, in three functions `sys_inccount`, `sys_deccount` and `sys_getcount`. As a convention, system calls with name `XXX` is implemented in a function named `sys_XXX`. Don't forget to include the `<linux/kernel.h>` and `<linux/errno.h>` headers, as the `asm_linkage` macro and the `EINVAL` constant are defined there.
3. Modify `<kroot>/kernel/Makefile` to add `counter.o` as an object file to build (in the `obj-y` list).
4. Modify `<kroot>/arch/i386/kernel/entry.S` to add three new entries at the **end** of the system call table. Remember the system call number of each system call you added.
5. Use `cd` to change to `<kroot>`, and type `make bzImage` to recompile the kernel. This step is relatively quick, because files that are not modified are not compiled again (thank to the `Makefile` mechanism). A warning will be emitted telling you that the kernel is too large to fit in the floppy, which we will ignore (as we aren't going to do so). If you find any other error, fix the problem and repeat.
6. A new kernel can be found in `<kroot>/arch/i386/boot/bzImage`. To make it a bit easier to boot the kernel, **copy it to the home directory** (i.e., `/home/os`). We will **not** install that kernel as a default kernel to boot, since the kernel is only for experiment.
7. Instead we will boot the kernel directly. Switch to the text console using Control-Alt-F1, and reboot using Control-Alt-Del. When the GRUB screen appears, quickly type `e` to edit boot entry (you only have 10 seconds to do that).

8. The second line specifies what kernel to boot. Use up and down arrows to move to that line, modify it by typing `e`, move to the place of the original kernel file (which begins with `/boot`), and replace it with `/home/os/bzImage`. Type `Enter`, and then `b` to boot your kernel.
9. After logging in again, **write a user program that calls your new system calls**, in the sequence `inccount`, `getcount`, `deccount`, `deccount` again, and finally `getcount` again. Remember to print out the return value, the `errno` variable and the corresponding error messages (use `strerror(errno)` to find it).
10. You will find that only the `inccount()` call can be used. It is because the kernel is hard-coded to only have 260 system calls, in the `NR_syscalls` constant of `<kroot>/include/linux/sys.h`. Modify it to 270, and repeat step 5 to 9.
11. Try the program again. If the results are not what you expect, debug it, and repeat the relevant parts of the above steps.

**Hint:** You might find it beneficial to use `printk` to print things for debugging. However, the output will only appear in the console, so if you need to do this, try to write your program in text-mode.

Happy kernel hacking!

## Appendix: Compiling the kernel the first time

In our tutorial we compile the most of the kernel for you. Later in the course you will have to do it to your own Linux. We use the following steps to compile it:

Step	Reason
<code>cd ~</code>	Copy the Linux source code to the home directory.
<code>cp -r /usr/src/linux-2.4.20-8/ .</code>	
<code>cd linux-2.4.20-8</code>	Change to the kernel root directory.
<code>make mrproper</code>	Clean up all old compiled files.
<code>cp configs/kernel-2.4.20-i686.config\ .config</code>	Reuse the default Redhat configuration.
Edit Makefile, remove <code>custom</code> in <code>EXTRAVERSION</code> .	So that all the old kernel modules can be used. (Otherwise we have to install our own using <code>make modules</code> , and install them using <code>make modules_install</code> .)
<code>make oldconfig</code>	Generate the kernel configuration files from <code>.config</code> .
<code>make dep</code>	Generate dependency files (which determines which files need to be recompiled when each file is modified).
<code>make bzImage</code>	Actually compile the kernel.

The whole process takes around 15 minutes in our computer if we don't recompile the modules, and around 1 hour if we do so.