OPERATING SYSTEMS PROJECT

“Implementing a Kernel level system call to monitor all the processes in the Linux system”

SUBMITTED BY :

GSV KRISHNA SAI	B140356CS
MS BHANU PRAKASH	B140445CS
V MADHU SAI	B140250CS

<u>ABSTRACT :</u>

Learning how to write a system call and adding into the set of already existing system calls in system_64.tbl, defining prototype of new system call in syscalls.h and compiling the kernel to get the new bzimage.

<u>Introduction :</u>

The system call linkage within any operating system is an important piece of functionality for the computer on which it is running. It provides an interface through which software (in user space) can request an action by the operating system itself. On the following pages, the process for successfully adding a new function and system call to the Linux kernel is examined and explained in detail.

<u>Description of Trap table and User Space Interaction</u>

Kernel functions appearing in the system call interface cannot be called directly from user space, but must be called through the trap table, also known as the system call table. This is accomplished by creating a new entry in the kernel trap table, which references the newly created function. Once created, the entry in the trap table can be called through a routine sys_call(). This function includes a trap instruction that changes the CPU from user to supervisor (administrative) mode so it can run the kernel function. In Linux, the trap instruction included here causes an interrupt (0x80), and uses the arguments passed through the sys_call() to access the system call table at the correct location.

Once identified, the system call function (syscall()) checks to ensure the requested function (passed as an argument from user space) is acceptable and then calls the kernel function, which executes according to instructions in its body. This will only occur, however, if the compiler agrees that the arguments passed from the function in user space are correct in number and type with regard to the kernel function.

Creating a New Kernel Function

Creating the new kernel function, named sys_process(), was a required task in this programming exercise. The prototype of the function needed to reside in the **/usr/src/linux-4.8.6/include/linux/syscalls.h** file within the kernel of the operating system. A new folder is made where our system call will be added and its corresponding Makefile. In the system call code, we traverse through all process using the for_each_process macro and check the state of the process and print the pid,state of the process. We print the following to kernel log using Printk. Printk() is almost identical to the printf function in user space, but it just adds the ability to print to stdout from kernel space. “dmesg” command is used to view the kernel log and “dmesg -c” is used to remove the existing kernel log.

Revision of System Call Table

After having completed the new kernel function, it was necessary to revise the sys_call_table so the operating system would recognize calls to this function.This was accomplished by editing the table in the **/usr/src/linux-4.8.6/arch/x86/entry/syscalls/syscall_64.tbl** file.There were already 328 entries in the system call table, so the new function would be identified as number 329 . It was also necessary to update the total number of system calls recognized by the system to 329, and this was done in the same file.With these new additions, whenever a trap/interrupt 0x80 occurs with an argument of 329, the new function will be invoked (assuming the kernel has been rebuilt with the new code included).

Rebuilding the Kernel

The last stage of the adding the function to the kernel project was to rebuild the kernel itself, and this proved to be one of the more challenging aspects of the entire assignment. The process can be time-consuming and very frustrating if your code does not compile (because you are not notified until step 4 of 5 in the process). All of the instructions for rebuilding the kernel in its various phases are found within the Makefile for each directory inside the kernel. That is, each directory has its own specific Makefile that hooks into the bigger Makefile in the top-level kernel directory. This ensures all pieces of the kernel are affected in the proper way when each step of the rebuild process is executed.

The first phase of rebuilding the kernel is to issue the “make clean” statement from the command line prompt. This causes the operating system to remove old object files so they may be replaced during the rebuild process. After this statement executes, the “make <config_opt>” command is issued from the command line. The <config_opt> is replaced by “oldconfig” and creates the kernel configuration based upon the old configuration that was present before the make clean statement was issued. After execution of this step, the command “make depend” is given. This command ensures that certain files within the kernel are scheduled for compilation before others, making certain all dependencies between various files are maintained. The fourth step in the process is issuing the “make” statement. This compiles all of the kernel source code, and results in the executable file version of the kernel.

It is in this step where the new code created in the kernel source files is compiled, and where the programmer is notified of successful or unsuccessful compilation of that code. If the code does not compile here, he/she will have to return and make adjustments based upon the errors encountered during compilation. This can prove to be a daunting task, because once the changes are made, all of the previous steps in rebuilding the kernel must be run to ensure integrity throughout the rebuild process. The last step of rebuilding the kernel includes issuing the “make <boot_opt>” command. The <boot_opt> is replaced, in this exercise, by bzImage. This creates a bootable kernel image and installs it within the kernel boot directory. If all five steps complete successfully, the new kernel function has been implemented correctly as far as compilation. The true test will be calling the function from syscall() in user space.

Creation of User Space Program

Finally, after the function was added correctly to the kernel, the user-space program was created.This was an easy task, as compilation and running of the program is all done within user space.The created function is invoked(called) in the user program to display the processes and their states.

Conclusion

The successful completion of a change to the system call interface coupled with creating a user space function is a time-consuming process. The end product, however, is more than just a change to the operation of an operating system. It includes attaining hands-on experience, and knowledge of a computer at a level not attainable unless each step of the process is worked through and completed.