****

**School of Computing Science and Engineering**

**Operating System Project**

**Adding a system call to the Linux kernel**

**DIPESH JAIN 15BCE0856**

**ABHISHEK SURYAN 15BCE0377**

**SARTHAK VIJAY 15BCE0366**

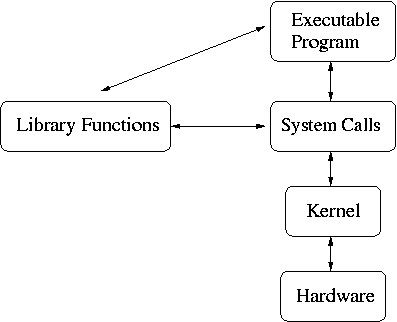
**Submitted To**

**Prof. USHUS ELIZEBETH ZACHARIAH**

**SCSE**

**ABSTRACT**

This project is based on adding a System Call to our operating system i.e UBUNTU. System calls are kernel functions that serve as an interface (for user mode application) to invoke kernel services like drivers, file systems, network stacks and others. System calls are also referred as system entry points since applications can enter kernel mode only through a valid system call interface. Applications can step into system calls using special processor specific soft interrupt instructions.



The aim of this project is to take a string as an argument and print the same when the cat command is executed. In our project, we will be working on Linux kernel **version 3.3.8**. We will create, compile and execute our system call on **UBUNTU**.

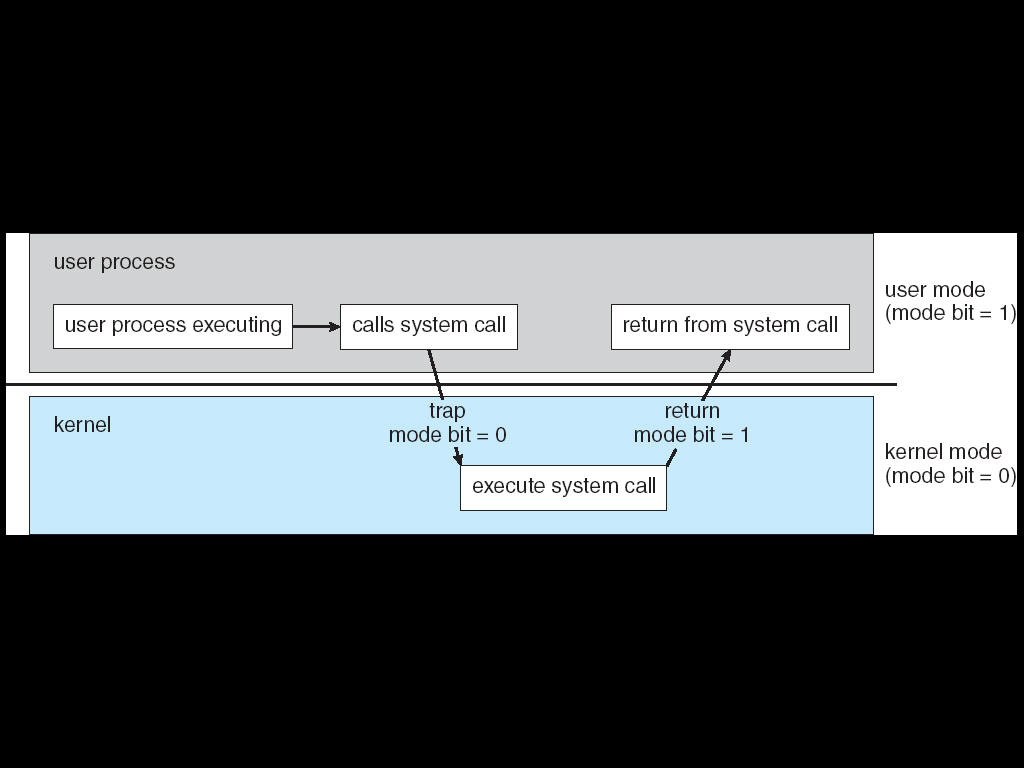
**INTRODUCTION**

This project is based on adding a System Call to our operating system i.e. UBUNTU. System calls are kernel functions that serve as an interface (for user mode application) to invoke kernel services like drivers, file systems, network stacks and others. System calls are also referred as system entry points since applications can enter kernel mode only through a valid system call interface. Applications can step into system calls using special processor specific soft interrupt instructions.

Background – User vs. Kernel mode

* Hardware provides two modes indicated by bit in PSW.
* Allows OS to protect itself and system components against faulty and malicious processes.
* Some instructions designated as privileged are only executable in kernel mode.
* System call, all traps and interrupts change mode from user to kernel and return from system call resets mode to user.

Transition from user to kernel mode



* Each different system call has its own number or other identity.
* Kernel trap handler uses system call number to index into the table of system call routines.

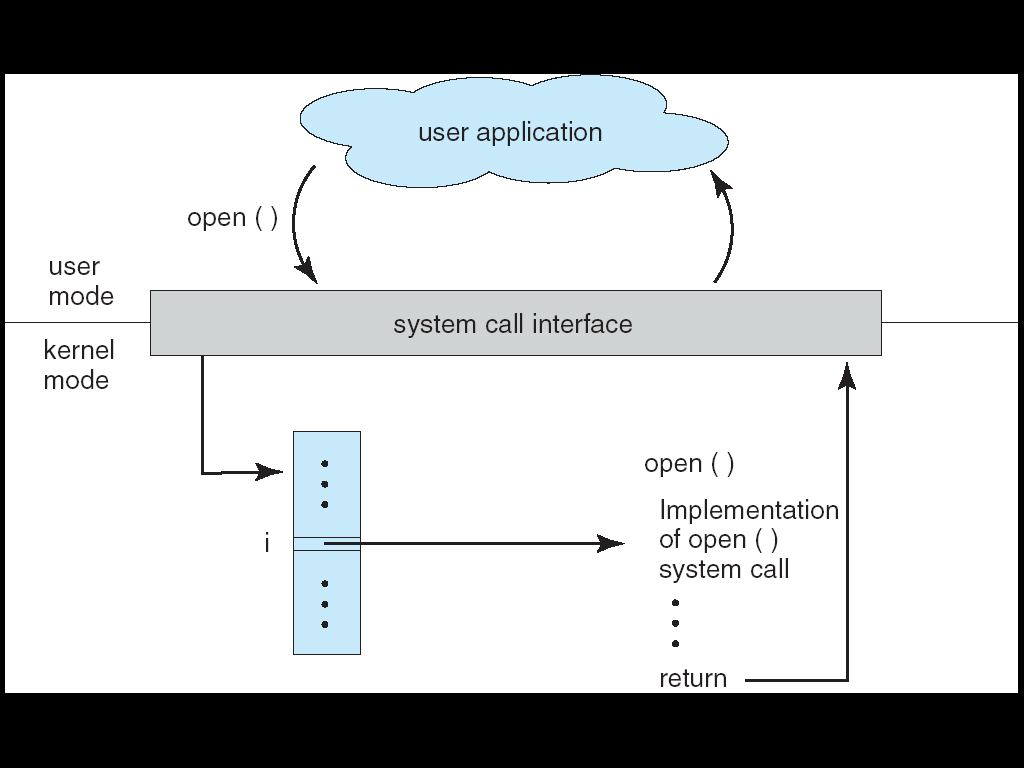
Inside kernel, the OS can

* Read and modify data structures not in user address space.
* Control devices and hardware settings forbidden to user processes.
* Invoke operating system functions not available to user processes.

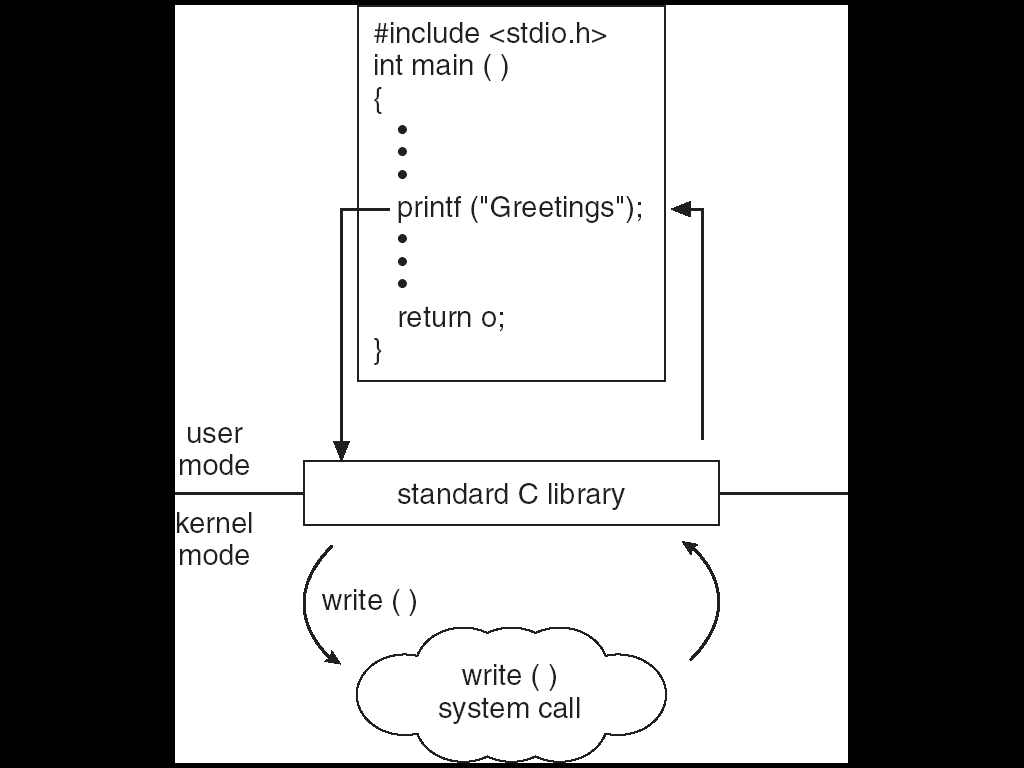
Accessing the kernel via system call

* Normally embedded within a library routine and the user API never makes system calls directly.
* System call mechanism is machine specific and different CPU architectures make system calls in different ways.
* System call numbers are different for various architectures even for same operating system and version.

Accessing kernel via library interface.



Accessing kernel via library interface



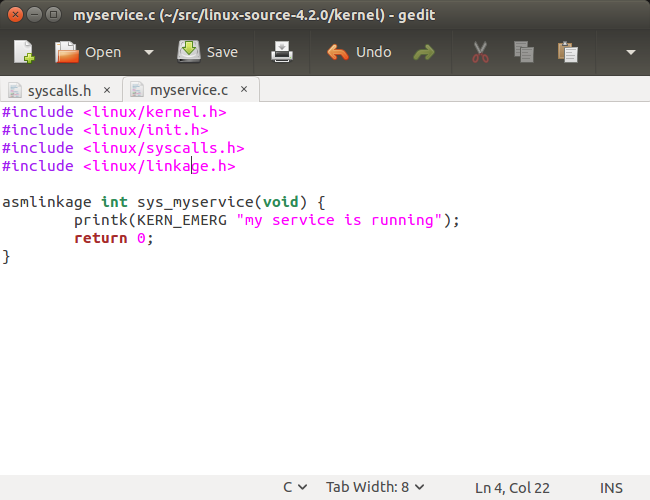
Requirements

* To carry out this project we need Ubuntu on our computers or we can carry this project out by installing VirtualBox on Windows Operating Systems to work on a virtual Linux Terminal.
* We will be using the Terminal interface since on the terminal we can directly install the software.
* We give a terminal command to get started:

sudo apt\_get install packagename

**PROPOSED MODEL**

**Design**



**Implementation**

1. **Getting Started**

A user-mode procedure call is performed by passing arguments to the called procedure either on the stack or through registers, saving the current state and the value of the program counter, and jumping to the beginning of the code corresponding to the called procedure. The process continues to have the same privileges as before. System calls appear as procedure calls to user programs but result i11 a change in execution context and privileges. In Linux on the Intel386 architecture, a system call is accomplished by storing the system-call number into the EAX register, storing arguments to the system call in other hardware registers, and executing a trap instruction (which is the INT Ox80 assembly instruction). After the trap is executed, the system- call number is used to index into a table of code pointers to obtain the starting address for the handler code implementing the system call. The process then jumps to this address, and the privileges of the process are switched from user to kernel mode. With the expanded privileges, the process can now execute kernel code, which may include privileged instructions that cannot be executed in user mode. The kernel code can then carry out the requested services, such as interacting with I/O devices, and can perform process management and other activities that cannot be performed in user mode. The system call numbers for recent versions of the Linux ker- nel are listed in lusr I src/linux-2. xl include/ asm-i386/unistd. h. (For instance, \_\_ NR\_close corresponds to the system call close 0, which is invoked for closin.g a file descriptor, and is defined as value 6.) The list of pointers to system-call handlers is typically stored in the file lusrlsrcllinux-2.x/arch/i386/kernel/entry.S under the head- ing ENTRY (sys\_calLtable). Notice that sys\_close is stored at entry number 6 in the table to be consistent with the system-call number defined in the unistd. h file. (The keyword .long denotes that the entry will occupy the same number of bytes as a data value of type long.)

**2) Building a New Kernel**

Before adding a system call to the kernel, you must familiarize yourself with the task of building the binary for a kernel from its source code and booting the machine with the newly built kernel. This activity comprises the following tasks, some of which depend on the particular installation of the Linux operating system in use.

Obtain the kernel source code for the Linux distribution. If the source code package has already been installed on your machine, the corresponding files might be available under lusr I srcllinux or /usr I src/linux-2. x (where the suffix corresponds to the kernel version number). If the package has not yet been installed, it can be downloaded from the provider of your Linux distribution or from http:l/www.kernel.org.

Learn how to configure, compile, and install the kernel binary. This will vary among the different kernel distributions, but some typical commands for building the kernel (after entering the directory where the kernel source code is stored) include:

o make xconfig

o make dep

o make bzimage

Add a new entry to the set of boatable kernels supported by the system. The Linux operating system typically uses utilities such as lilo and grub to maintain a list ofbootable kernels from which the user can choose during machine boot-up. If your system supports lilo, add an entry to lilo. conf, such as:

image=/boot/bzimage.mykernel label=mykernel root=/dev/hda5 read-only

where lbootlbzimage. my kernel is the kernel image and my kernel is the label associated with the new kernel. This step will allow you to choose the new kernel during the boot-up process. You will then have the option of either booting the new kernel or booting the unmodified kernel if the newly built kernel does not ftmction properly.

**3) Extending the Kernel Source**

You can now experiment with adding a new file to the set of source files used for compiling the kernel. Typically, the source code is stored in the lusr I srcllinux-2. xlkernel directory, although that location may differ in your Linux distribution. There are two options for adding the system call. The first is to add the system call to an existing source file in this directory. The second is to create a new file in the source directory and modify lusr I srcllinux-2. xlkerneliMakefile to include the newly created file in the compilation process. The advantage of the first approach is that when you modify an existing file that is already part of the compilation process, the Makefile need not be modified.

**4) Adding a System Call to the Kernel**

Now that you are familiar with the various background tasks corresponding to building and booting Linux kernels, you can begin the process of adding a new system call to the Linux kernel. In this project, the system call will have limited functionality; it will simply transition from user mode to kernel mode, print a message that is logged with the kernel messages, and transition back to user mode. We will call this the myservice system call. While it has only limited functionality, it illustrates the system-call mechanism and sheds light on the interaction between user programs and the kernel.

1. Create a new file called helloworld. c to define your system call. Include the header files linuxllinkage.h and linuxlkernel. h. Add the following code to this file:

#include <linuxllinkage.h>

#include <linuxlkernel.h>

asmlinkage int sysJ

myservice()

{ printk(KERN\_EMERG "my service is running");

return 1;

}

This creates a system call with the name sys\_myservice (). If you choose to add this system call to an existing file in the source directory, all that is necessary is to add the sys\_myservice () function to the file you choose. In the code, asmlinkage is a rellli<ant from the days when Linux used both C++ and C code and is used to indicate that the code is written in C. The printk () function is used to print messages to a kernel log file and therefore may be called only from the kernel. The kernel mes- sages specified in the parameter to printk () are logged in the file /var/log/kernel/warnings. The function prototype for the printk () call is defined in /usr /include/linux/kernel. h.

1. Define a new system call number for \_\_ NR\_myservice in /usr/src/linux-2.x/include/asm-i386/unistd.h. A user program can use this number to identify the newly added system call. Also be sure to increment the value for \_\_ NR\_syscalls, which is stored in the same file. This constant tracks the number of system calls currently defuced in the kernel.
2. Add an entry .long sys\_myservice to the sys\_calLtable definedinthe/usr/src/linux-2.x/arch/i386/kernel/entry.S file. As discussed earlier, the system-call number is used to index into this table to find the position of the handler code for the invoked system call.
3. Add your file myservice.c to the Makefile (if you created a new file for your system call.) Save a copy of your old kernel binary image (in case there are problems with your newly created kernel). You can now build the new kernet rename it to distinguish it from the unmodified kernet and add an entry to the loader configuration files (such as lilo. conf). After completing these steps, you can boot either the old kernel or the new kernel that contains your system call.

**5) Using the System Call from a User Program**

When you boot with the new kernet it will support the newly defined system call; you now simply need to invoke this system call from a user program. Ordinarily, the standard C library supports an interface for system calls defined for the Linux operating system. As your new system call is not linked into the standard C library, however, invoking your system call will require manual intervention. As noted earlie1~ a system call is invoked by storing the appropriate value in a hardware register and performing a trap instruction. Unfortunately, these low-level operations cannot be performed using C language statements and instead require assembly instructions. Fortunately, Linux provides macros for instantiating wrapper functions that contain the appropriate assembly instructions. For instance, the following C program uses the \_syscallO () macro to invoke the newly defined system call:

#include <linux/errno.h>

#include <sys/syscall.h>

#include <linux/unistd.h>

\_syscallO(int, myservice();

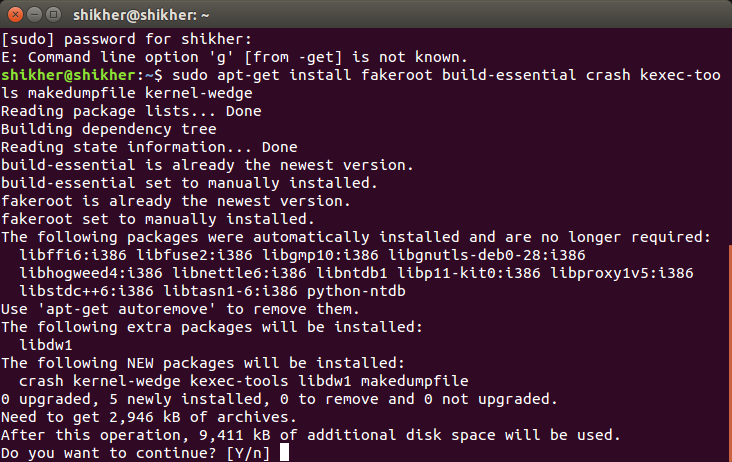
main()

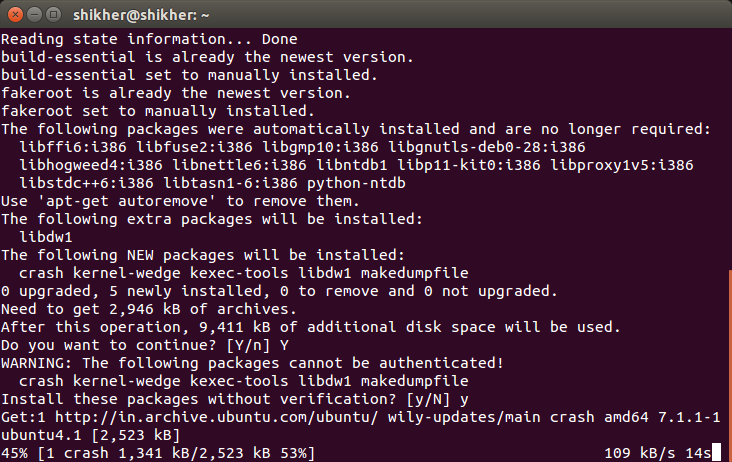
{

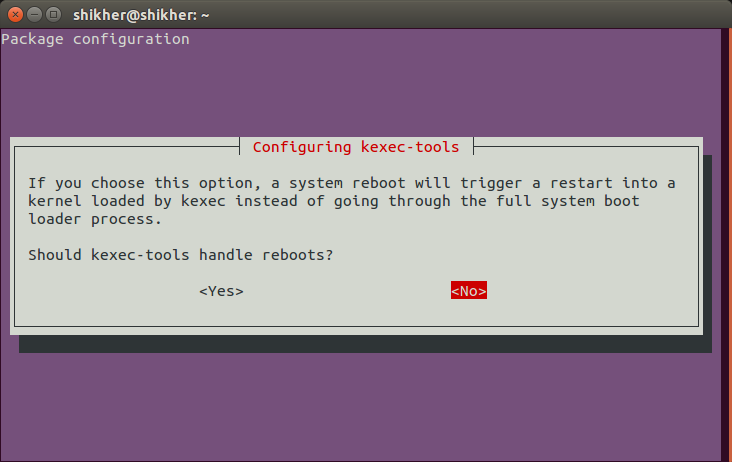
myservice ();

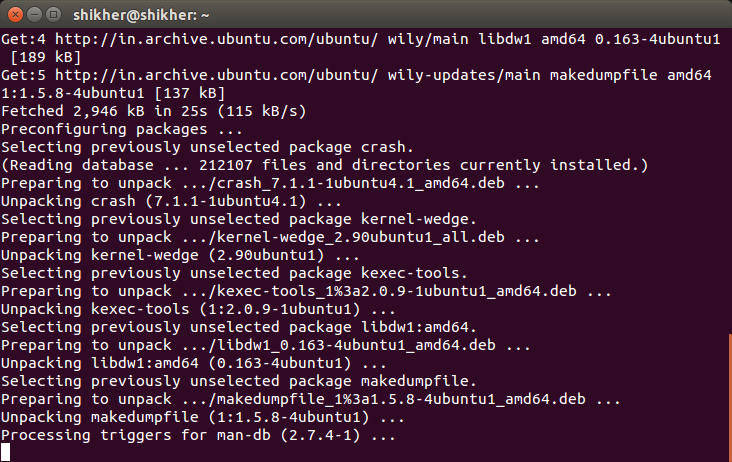
}

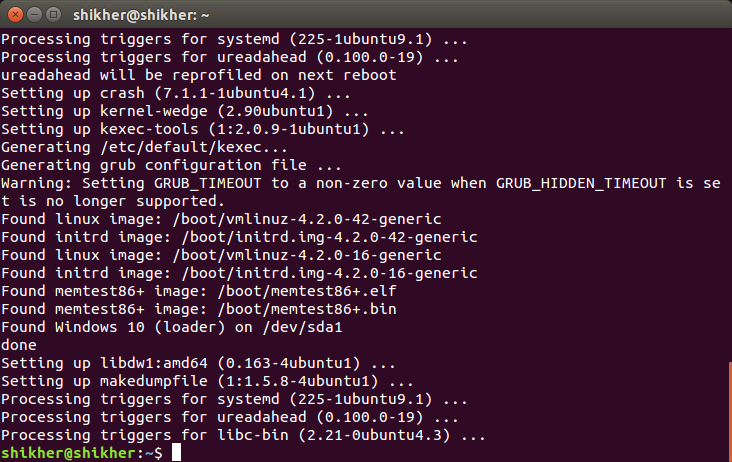
1. The \_syscallO macro takes two arguments. The first specifies the type of the value returned by the system call; the second is the name of the system call. The name is used to identify the system- call number that is stored in the hardware register before the trap instruction is executed. If your system call requires arguments, then a different macro (such as \_syscallO, where the suffix indicates the number of arguments) could be used to instantiate the assembly code required for performing the system call.
2. Compile and execute the program with the newly built kernel. There should be a message “my service is running" in the kernel log file /var/log/kernel/warnings to indicate that the system call has executed.

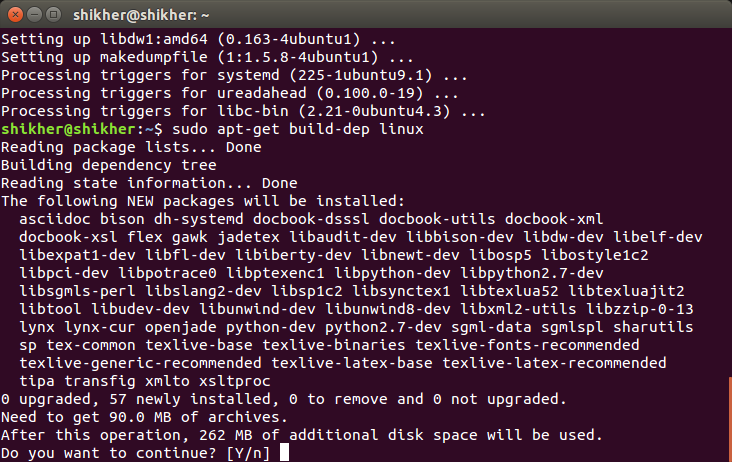


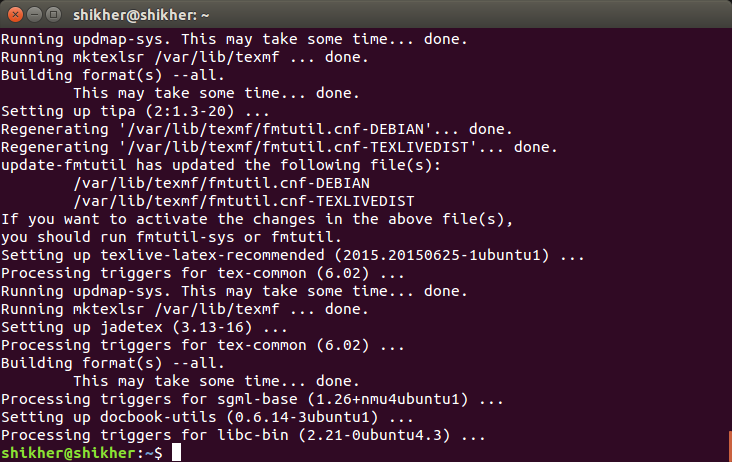


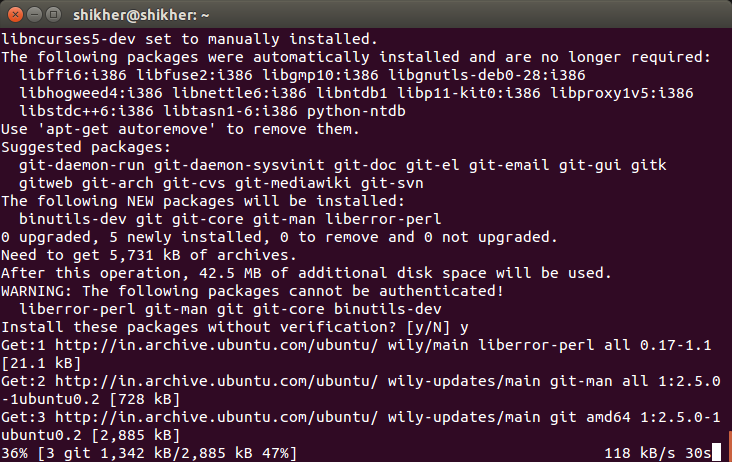


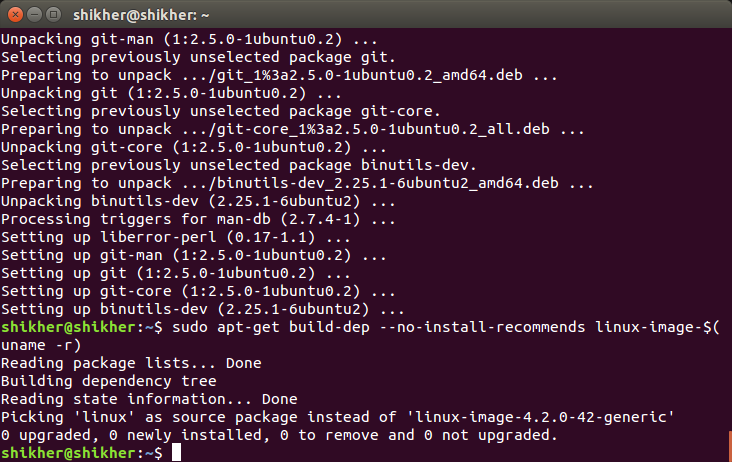


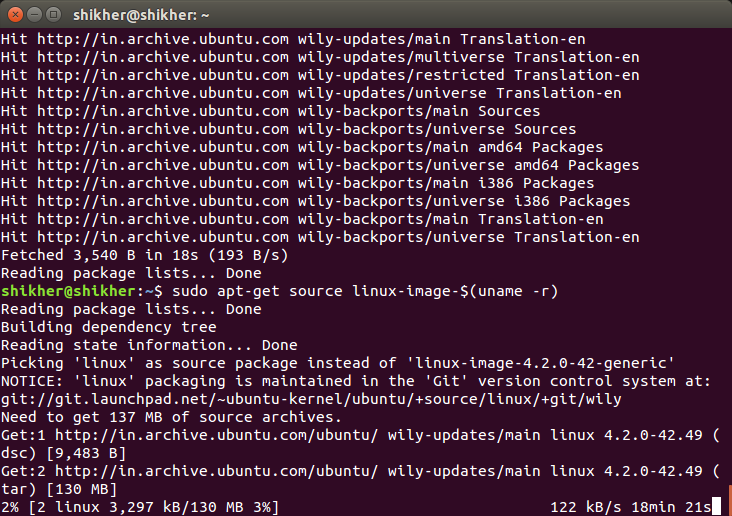


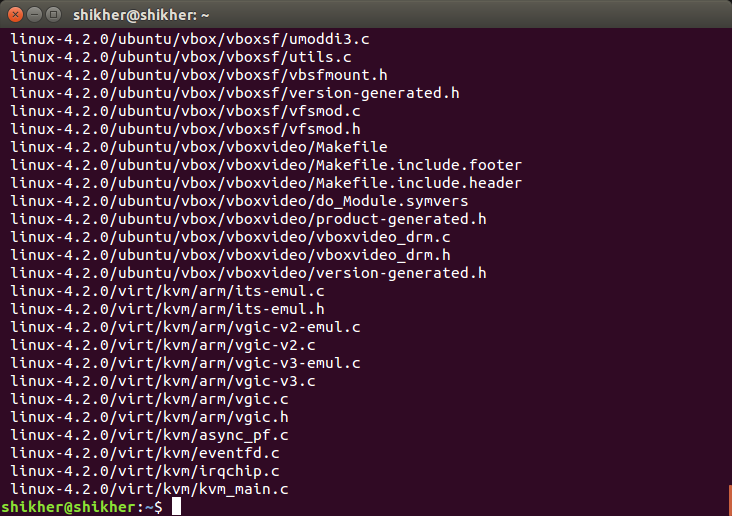


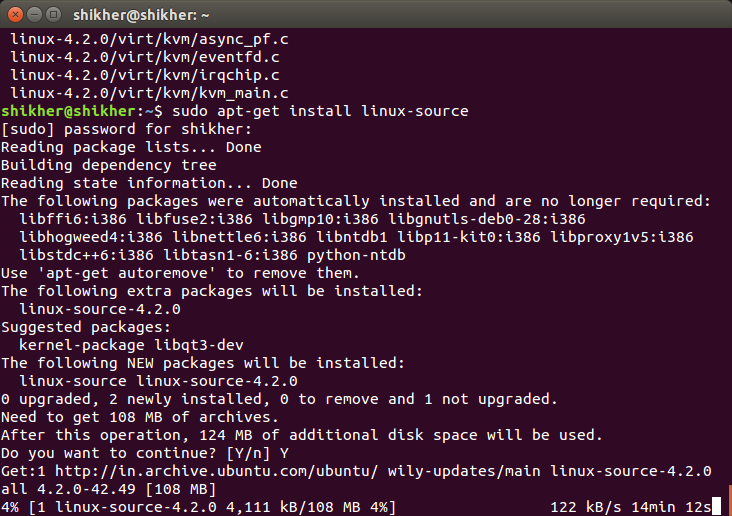


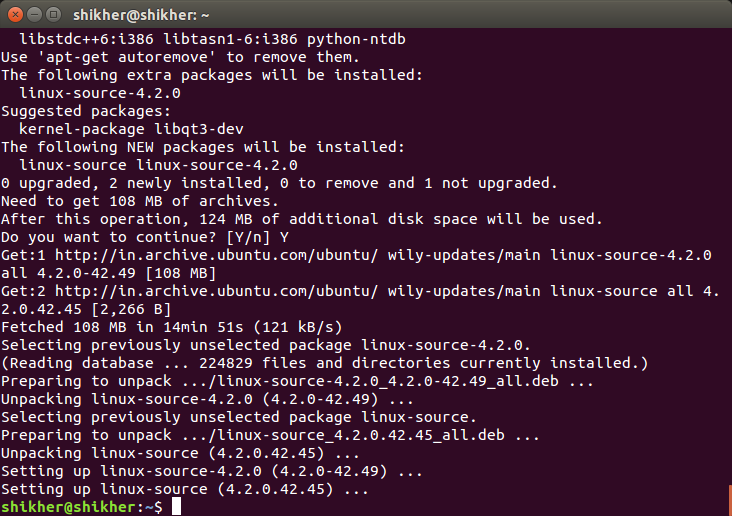


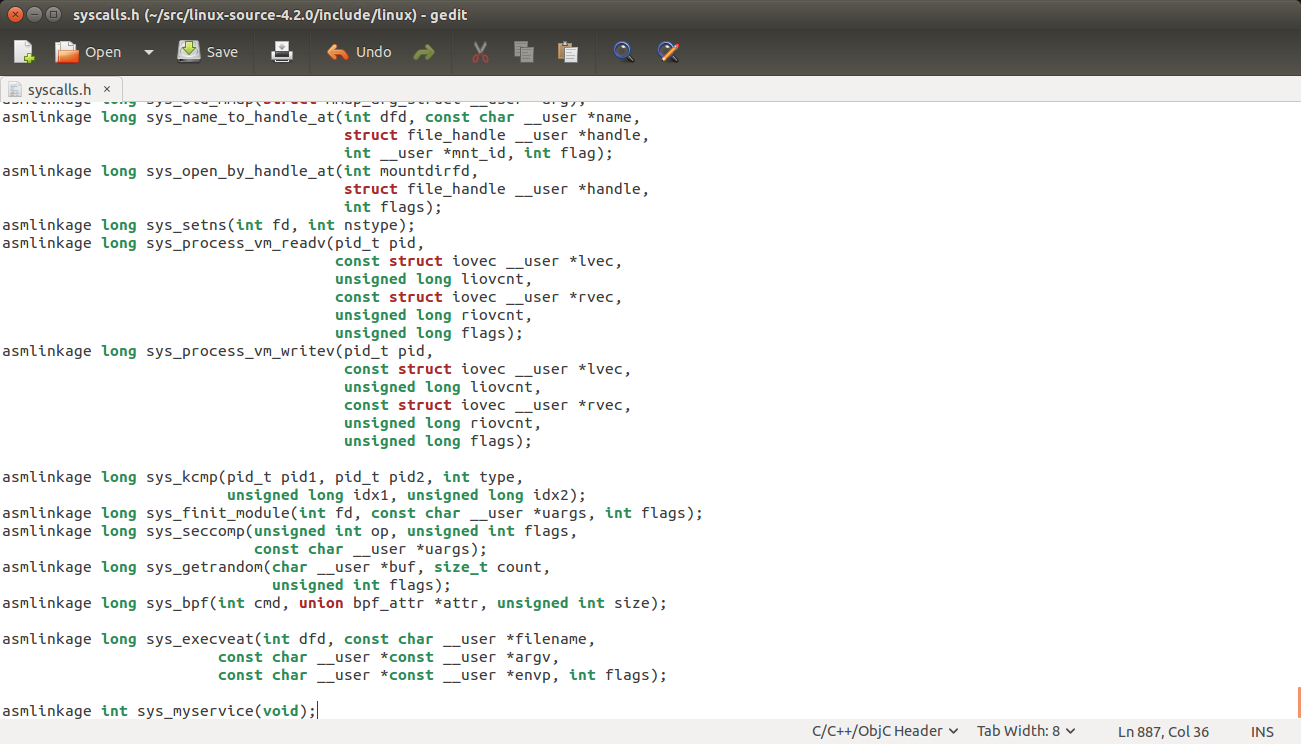


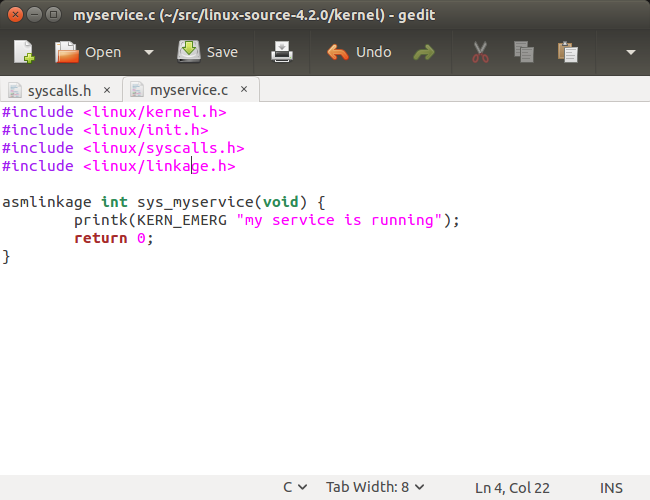
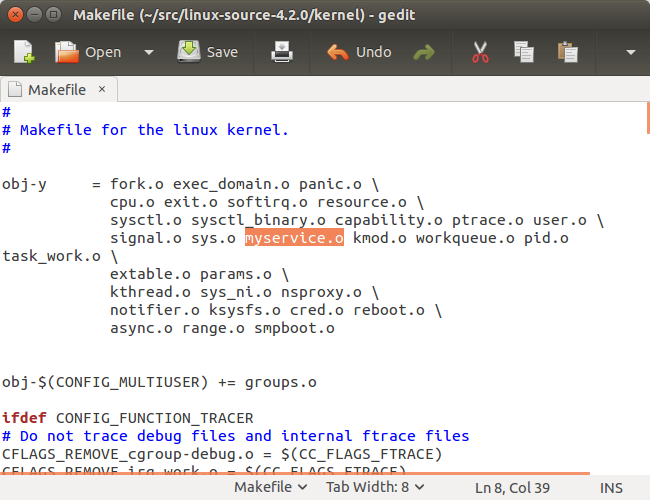


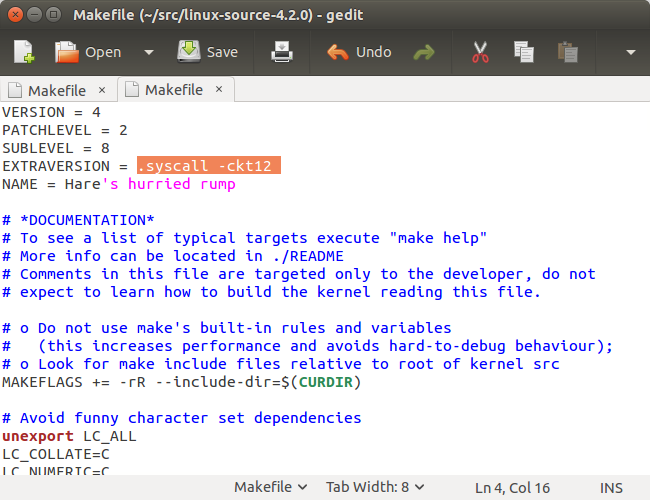


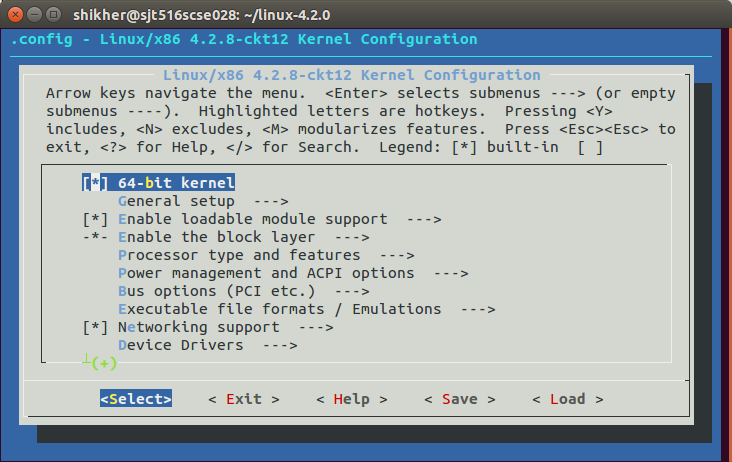


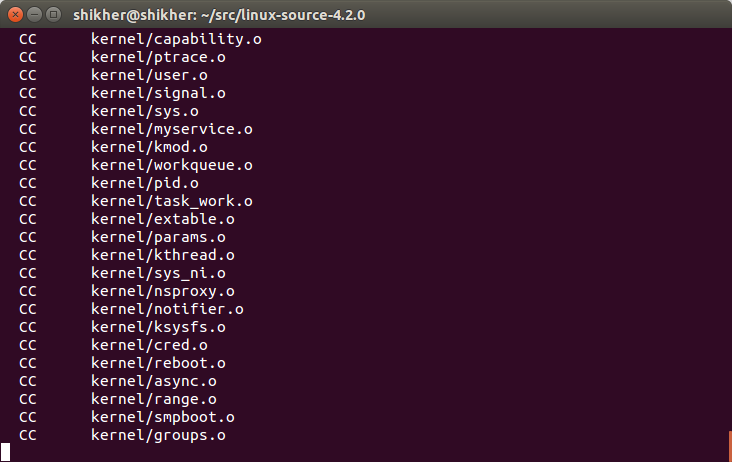












**Conclusion**

Our project on Operating Systems was to add a System call to the Linux operating system. In our system call, we take a string as an argument and print it on the execution of cat command.

We were successfully able to execute our system call and this project led to a deeper understanding about the linux operating system. We got to know about how a new system call is associated to the original source code and we also learned about all the changes that are needed to add a system call.

**References**

Internet-

* <https://www.youtube.com/watch?v=5rr_VoQCOgE&feature=youtu.be>
* <https://www.youtube.com/watch?v=vZaA2mRT5eg&feature=youtu.be>
* <https://www.youtube.com/watch?v=1Rido46nao8>
* <http://www.csee.umbc.edu/courses/undergraduate/CMSC421/fall02/burt/projects/howto_build_kernel.html>
* <http://www.csee.umbc.edu/courses/undergraduate/CMSC421/fall02/burt/projects/howto_add_systemcall.html>
* <http://franksthinktank.com/howto/addsyscall/>

Book-

* Operating System Concepts, Silberschatz, Galvin, Gagne