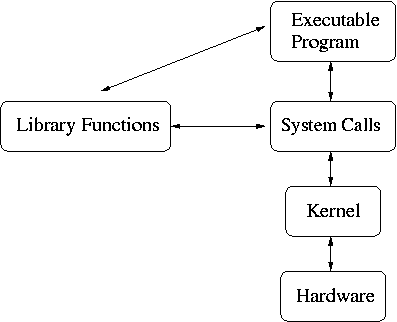
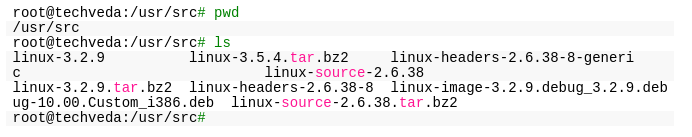
****Introduction****

System-calls are kernel functions that  serve as an interface (for user mode applications ) to invoke kernel services like drivers, file-systems, Network stacks  and others. system calls are also referred as “kernel 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. Linux kernel is widely being deployed and used in virtually every computing platform from desktops, servers, enterprise platforms to Mobile, Deep embedded, Robotics, consumer electronics, medical device platforms  and the list can go on and on.

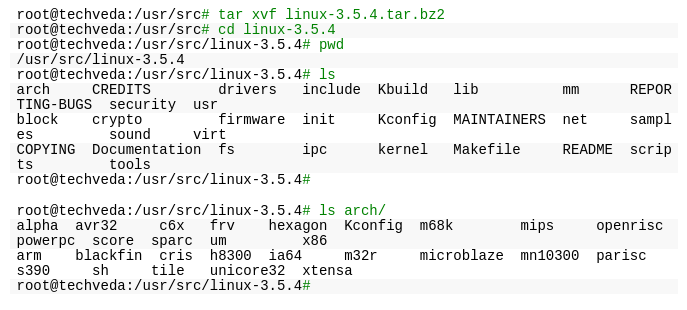
while customizing and deploying Linux on to various platforms  there may be need for adding new services and system calls to kernel sources.  Fresh programmers to Kernel programming concepts  may find it more interesting to explore system call , API concepts practically by adding new system call, and implementing applications to invoke those calls . [(watch linux system calls video tutorial here.)](http://blog.techveda.org/index.php/linux-system-calls-video-tutorial/" \o "Linux system calls: video Tutorial) This document provides details of how to add  new system calls to Linux kernel Sources (3.3 version on-words) for x86 32 & 64 bit arch.

****Getting kernel sources****

we will start by downloading latest stable kernel from kernel community web-site www.kernel.org.  it is a common standard to host source packages in /usr/src branch of rootfs (but you may copy compressed source tar file into any folder).  This document assumes /usr/src/ to be the location of source tar file .

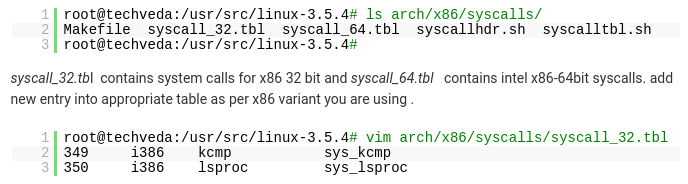


let’s Extract source from compressed tar file and change our working directory into kernel source tree.



Above dump shows the contents of *arch* branch of kernel source.  *arch* contains Linux hardware abstraction layer code for various architectures that Linux kernel  already includes  support for, kernel maintains an architecture specific system call table  which holds system call addresses.  To add an new system call we need to append a new entry into system call table and store address of syscall function.

syscall table for x86 architecture can be  found in  *arch/x86/syscalls* branch of kernel source.



New system call is implemented to show list of processes currently load and is given a name*lsproc.*  we have added a new entry into syscall table*,*entry  begins with offset no of the table , we added a new offset 350(already table had 249 offsets) , next we mention ABI tag as i386 for 32 bit x86,  next we mentioned name of new system call (lsproc) and at last address of the system call function *sys\_lsproc (fucntion name resolves to its address).*system call functions in Linux kernel  are assingned names as per  *sys\_syscallname*naming convention, this makes it easy to identify a function as system call  while browsing kernel sources.

User Mode apps will use the offset no of the system call table while invoking system call, in our case the offset application will have to use  350 for invoking *lsproc* syscall.

****X86 64bit****

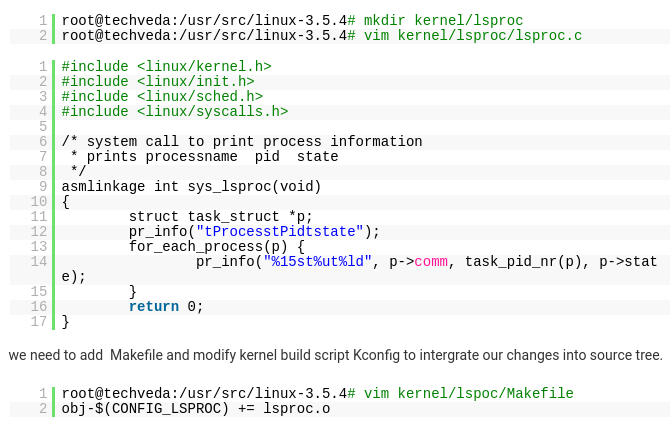
****4****

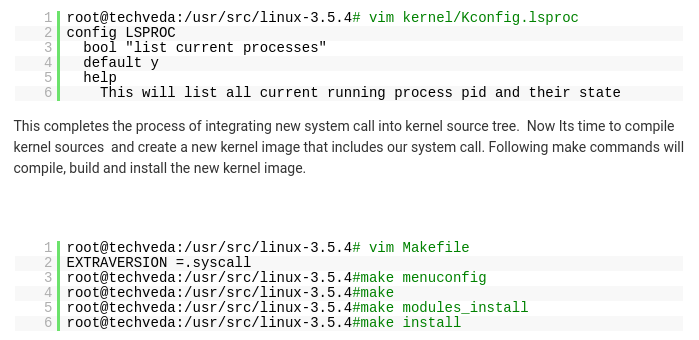
313 is the offset, 64  indicates  x86 64bit  ABI and lsproc is the name of the system call and sys\_lsproc is the address of function

Let us now implement our system call function with the name sys\_lsproc. we will first declare fucntion prototype in the*linux/syscalls.h* kernel header

5

asmlinkage  is kernel tag  #defined with a magic  no to instruct compiler   that all arguments to the function must be accessed from kernel stack.  let us implement function code in a new source file*lsproc.c* . we will host our source file *lsproc.c* in*lsproc* directory under *kernel* branch of source tree.

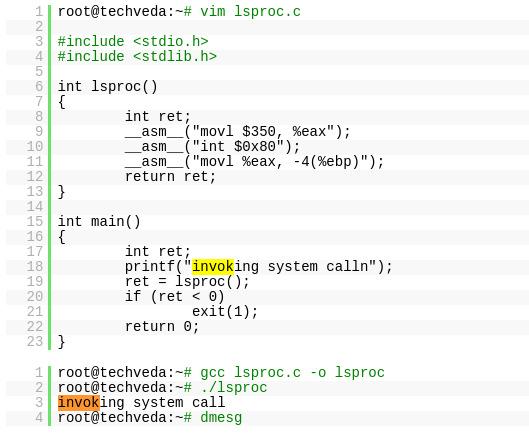




****Test Application****

Application can invoke the system call using x86-32bit  trap exception  with system call offset as a parameter passed into *eax* accumulator register.

Dmesg would list process list extracted and printed by our system call on console.



# Adding a Hello World System Call to Linux kernel 3.16.0

Hey folks!! I am going to use this article to explain how to add a “hello world system call” to your kernel as I recently accomplished this task. My OS is Linux Ubuntu 14.04 and the kernel version I have implemented this in  is, 3.16 and I have used  a 32 bit system. The following steps will be more or less the same with little changes here and there for other versions of kernel.

The steps to be followed to add a hello world system call in your kernel are :-

You can gain root access by using  *“sudo -s*” command and typing in the password when prompted;  to avoid typing “sudo” separately for all commands.

* ****Download the kernel source****

Open the terminal and use the following command to download the kernel course file.

*wget <https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.16.tar.xz>*

wget command :- GNU Wget is a free utility for non-interactive download of files from the Web.

* ****Extract the kernel source code****

Extract the kernel source code from the linux-3.16.tar.xz file in /usr/src/ directory using the following command. Since the downloaded tar file will be in Downloads folder, use cd to change into Downloads folder before executing the below command.

*sudo tar -xvf linux-3.16.tar.xz -C/usr/src/*

sudo – to gain root access.

tar –  Tar stores and extracts files from a tape or disk archive.

-x – extract files from an archive

-v – requested using the –verbose  option, when extracting archives

-f –  –file archive;  use archive file or device archive

-C, –directory DIR,change to directory DIR( here to change to /usr/src/)

Now after extraction change to the kernel source directory using,

*cd /usr/src/linux-3.16/*

* ****Define a new system call sys\_hello()****

Create a directory hello in the kernel source directory:-

*mkdir hello*

Change into this directory

*cd hello*

Create a “hello.c” file in this folder and add the definition of the system call to it as given below (you can use any text editor ).

*gedit hello.c*

add the following code:-

*#include <linux/kernel.h>*

*asmlinkage long sys\_hello(void)*  
*{*  
*printk(“Hello world\n”);*  
*return 0;*  
*}*

Note that printk prints to the kernel’s log file.

2.  Create a “Makefile” in the hello folder and add the given line  to it.

*gedit Makefile*

add the following line to it:-

*obj-y := hello.o*

This is to ensure that the hello.c file is compiled and included in the kernel source code.

* ****Add the hello directory to the kernel’s Makefile****

change back into the linux-3.16 folder and open Makefile

*gedit Makefile*

goto line number 842 which says :-   “*core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/* ”

change this to   “*core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ hello/*”

This is to tell the compiler that the source files of our new system call (sys\_hello()) are in present in the hello directory.

* ****Add the new system call (sys\_hello() ) into the system call table (syscall\_32.tbl file)****

If your system is a 64 bit system you will need to alter the syscall\_64.tbl file.

*cd arch/x86/syscalls*

*gedit syscall\_32.tbl*

add the following line in the end of the file :-

*354    i386    hello    sys\_hello*

354 – It is the number of the system call . It should be one plus the number of the last system call. (it was 354 in my system). This has to be noted down to make the system call in the userspace program.

* ****Add the new system call(sys\_hello() )  in the system call header file.****

*cd  include/linux/*

*gedit syscalls.h*

add the following line to the end of the file just before the #endif  statement at the very bottom.

*asmlinkage long sys\_hello(void);*

This defines the prototype of the function of our system call.”asmlinkage” is a key word used to indicate that all parameters of the  
function would be available on the stack.

* ****Compile this kernel on your system.****

To compile Linux Kernel the following are required to be installed.

1. gcc latest version,
2. ncurses development package
3. system packages should be up-to date

Hence do the following in the terminal.

1. *sudo apt-get install gcc*
2. *sudo apt-get install libncurses5-dev*
3. *sudo apt-get update*
4. *sudo apt-get upgrad*e

=> To configure your kernel use the following command:-

*sudo make menuconfig*

Once the above command is used to configure the Linux kernel,  you will get a pop up window with the list of menus and you can select the  
items for the new configuration. If your unfamiliar with the configuration just check for the file systems menu and check whether “*ext4*” is chosen or not, if not select it and save the configuration.  
If you like to have your existing configuration then run the below command.

*sudo make oldconfig*

Now to compile the kernel ; do make .

*cd /usr/src/linux-3.16/*

*make*

This might take several hours depending on your system (using hypervisors can take a longer time ). It took me 2-3 hours to get this compiled.

* ****To install /update the kernel.****

To install this edited kernel run the following command:-

*sudo make modules\_install install*

The above command will install the Linux Kernel 3.16 into your system.It will create some files under /boot/ directory and it will  automatically make a entry in your grub.cfg.  To check whether it made correct entry ; check the files under  /boot/ directory . If you have followed the steps without any error you will find the following files in it in addition to others.

1. System.map-3.16.0
2. vmlinuz-3.16.0
3. initrd.img-3.16.0
4. config-3.16.0

Now to update the kernel in your system reboot the system . You can use the following command.

*shutdown -r now*

After rebooting you can verify the kernel version using the following command;

*uname -r*

* ****To test the system call.****

Create a “userspace.c” program in your home folder and type in the following code :-

*#include <stdio.h>*  
*#include <linux/kernel.h>*  
*#include <sys/syscall.h>*  
*#include <unistd.h>*  
*int main()*  
*{*  
*long int amma = syscall(354);*  
*printf(“System call sys\_hello returned %ld\n”, amma);*  
*return* *0;*  
*}*

Now compile this program using the following command.

*gcc userspace.c*

If all goes well you will not have any errors else, rectify the errors.

Now run the program using the following command.

*./a.out*

You will see the following line getting printed in the terminal if all the steps were followed correctly.

“*System call sys\_hello returned 0*“.

Now to check the message of the kernel you can run the following command.

*dmesg*

This will display “*Hello world*” at the end of the kernel’s message and that indeed did make me happy. We being so used to the user space environment, where we can just type a simple hello world program, this makes us look at hello world program from another angle!!