Linux Kernel Project
COSC 519-01

Rana Almakabi

Leena Naik

Shivani Malhotra

Sushma Shrestha

Douglas Keeley

Contents

Abstract

This report describes the steps taken to insert a user-created module into the kernel to extend the running kernel – or the so-called base kernel – of an operating system. It involves downloading complete linux kernel source code, setting the configurations, compiling the kernel source and building and linking the user module to the kernel . We compiled the kernel source into a new kernel version and had our module load during the boot time along with all preinstalled kernel modules.

Introduction

The project has several high-level objectives. Primarily we sought to execute a modifications to the kernel, and utilize all the necessary tools and techniques to do so. However, we must understand not just the use of these things, but what they themselves do and what tells us about the design of the system. Having succeeded these things, we reflect on the body of knowledge and experience we gained from this, and the value of the project

1. Linux Kernel Concepts

1.1. The Kernel

Kernel is the central component of most computer operating system and it provides basic services to different parts of the operating system. It acts as a bridge between application and the actual data processing which is done at the hardware level. The main responsibility of a kernel is to manage the computer resources and allow other programs to run and use these resources.

The Central processing unit of a computer is responsible for running or executing a program on it. The kernel responsibility is to decide out of many running programs which program should be allocated to the processor.

In order to execute a program, both program instructions and data should be store in the memory. Multiple programs want to access the memory at the same time and requesting for the more memory than physically available in the computer. The kernel on such situation is responsible for managing the memory for each process.

Various Input/Output devices are present in the computers such as keyboard, printer, disk drive etc. and the Kernel allocates the request form the application to perform the I/O operation to the appropriate drivers.

1.2. Kernel Modules

Modules are codes which can be loaded and unloaded into the kernel when required. Example of module is device driver which allows the kernel to access hardware connected to the computer.

Kernel Modules work a bit differently than a program. Mostly a program start with a main() function executes the instructions and terminates after the completion of the instructions. Whereas a module always begin with the init_module or the function mentioned within the module_init call. The entry functions of the modules tell the kernel what functionality the module has provided. It also guides the kernel to run the module's functions when they are needed. Once the entry function returns, the module does nothing until the kernel wants to execute the code that the modules provide.

Modules end by calling cleanup_module or the function which is specified with the module_exit call. The exit function for the modules unregisters the functionality that the entry functions registered.

If the concept of the module was not present then we would have to build huge kernel and add new functionality directly into the kernel image and rebuild and reboot the kernel every time we want new functionality.

1.3. Kernel Boot Process

When a kernel is loaded it is typically a compressed image file. At the head of the kernel image there is a routine functions which does some basic hardware setup and then decompressed the kernel contained within the kernel image and places it into the high memory. After that the control is give to the main kernel start process.

The startup functions for the kernel also known as swapper or process 0 established the memory management, detect the type of CPU, and then switched to the Linux Kernel functionality via a call to start _kernel (). The start_kernal functions execute a wide range of initialization functions. It configure memory, starts the Init process which is the first user-space process, set up interrupt handling, and then starts the idle task through cpu_idle ().

The kernel startup process also mounts the initial-RAM disk (initrd) which was loaded as the temporary root filing system during the boot phase and mounted. This initrd act as a temporary root file system in RAM and help the kernel to boot fully without having to mount any physical disks. Necessary modules required to interface with the peripherals can be part of the initrd. The kernel might be very small but still can support the possible hardware configuration. The initrd function allows creating a small kernel with drivers compiled as loadable modules. This loadable module allows the kernel to access disks and the file system on those disk and driver for the hardware also.

Once the kernel is booted and initialized it starts the first user-space application Init.

Init process primary role is to create processes from a script stored in the file /etc/inittab and it also control the autonomous processes required by any particular system. Init's established and operates the entire user space which includes checking and mounting the file system, starting up necessary user services, and finally switching to a user-environment when the system startup process is completed.

Like the Linux the Linux boot processes is also flexible and also support large number of processor and hardware platforms. The new generation of the boot loader such as GRUB allows Linux to boot from a wide range of file systems.

1.4 Linux Source Distribution

Being open-source software, Linux provides several convenient means for acquiring and handling the source code. Official Linux code is maintained in source control, which is generally freely accessible via various client applications such as for Git or Subversion. For each version release, all of the source code is added to an archive file. This file can be downloaded, unpacked, and compiled with the included scripts.

The source is also organized into software packages, so that it can be manipulated by the major software package managers. One is the Red Hat Package manager (RPM) used by the Red Hat family of distributions such as RHEL, Fedora, CentOS and others. Another is Apt, used by the Debian family, which includes Ubuntu which we used for this project. Both package managers can determine what packages are installed, what packages are available, updates that are available, what dependencies or prerequisites are required, and automatically download and install what is requested. Both package managers have shell interfaces as well as a choice of GUI front-ends.

The packages managers are conveniences, but as they introduce a level of insulation the approach used on this project was to deal manually with the source archive.

1.5 Source Structure

Linux source is normally maintained under the /usr/src directory. There may be several source directory trees within it, named appropriately (eg: "linux-2.6.38.2", and if needed, a generic symbolic link "linux" pointing to the one being worked on to simplify build scripts. It may be arranged as follows:

Directory	Description
/usr	User applications and their files

/usr/src	Linux source
/usr/src/linux	Symbolic link selecting a kernel source directories (optional)
/usr/src/linux-2.6.38.2	Linux kernel source directory (expanded)

A linux source directory can be expanded (under /usr/src) directly from the archive. The directories often will have subdirectories of components, or support for particular architectures. Each of these contain similar files – implying that the items in a directory (such as the architectures) are modular, and that they are generalized by Linux. For example, each file system directory under fs/ contains a file.c, which implements the standard file operations for that file system, and other filesystems have their own file.c file to support standard file operations.

This main directory also contains the master Makefile for compiling the kernel. This relies on scripts in the scripts/ directory, and specifies code subdirectories which each have their own Makefiles, and so on. The Makefiles rely on tools in the /scripts folder for compilation.

The standard directory structure is explained by the table below. Different kernel versions will vary. The "Examples" are provided to give a sense of the contents and how they are organized within that directory.

Directory	Description	Example
arch/	Code supporting specific architectures / processors. Each OS component (which may be centralized elsewhere) organizes its files in each platform subdirectory that is supported	**arch/x86/power/cpu.c** - implements power features for the intel platform (suspend, hibernate)
block/	Block device support	
crypto/	Cryptography libraries	**crypto/blowfish.c** – implements the popular blowfish cipher
documentation/	Documentation explaining the functionality and status of a variety of OS and OS-related topics. It is not code documentation.	**documentation/ext3.txt** – explains functionality of the ext3 file system
drivers/	Drivers and support code, grouped by category	**drivers/usb/storage/usb.c** – mass storage device driver

fs/	File system code	**fs/ext3/file.c** - implements kernel file operations for the ext3 file system
include/	Include directory for kernel source code (where #include<> points)	**include/ext3_fs.h** – defines the ext3 file system
init/	Kernel initialization code.	**init/main.c** – the kernel's entry point after it is loaded.
ipc/	Inter-process communication code	**ipc/shm.c** – implements shared memory functions (same as used on HW#4 / shmem.c: shmat(), shmget(), shmctl().
kernel/	General kernel code. Platform-specific kernel code is organized in **arch/**	**Kernel/signal.c** – implements signal handling for the OS
lib/	Kernel library code	
mm/	Memory management code	**mm/swap.c** – implements virtual memory logic
modules/	Kernel modules	(does not exist in 2.6.38.2)
net/	Networking (eg. tcp stack)	**net/ipv4/tcp.c** -- implements TCP/IP (IP4)
scripts/	Kernel configuration and build scripts. These are referred to when configuring or building the kernel	**scripts/findheaders.pl** – a tool for finding duplicate headers in source code

1.6 Necessary Tools and Commands

This section discusses the different tools that are used in the implementation, in concept and usage.

Sudo

This temporarily elevates the user to root privileges, either for an individual command or for a session terminated by the user or timeout. Sudo maintains a detailed log of all activities by users in this state, delineated by "tickets". Sudo requires that the user be allowed to run particular commands, and that the user has been authenticated (issued a ticket) in the last 5 minutes; it will prompt if necessary.

Many steps in this project required privileged access, often simply because the directories being worked on can only be changed by root. Some commands, however, will only function correctly when they have access to settings in root's environment.

Example of individual command:

```
sudo touch /usr/src/hellothere
```

Example session:

```
$ sudo -s

[sudo] password for user1:

# touch /usr/src/hellothere

# exit

exit

$
```

Wget

This simply "gets" files via HTTP to the local filesystem. We use this to retrieve the kernel source archive file from the web.

Tar

This is the unix "tape archive", which creates or extracts archives, including handling zip compression. We used this to expand the source archive.

Package Management

An alternative to manually downloading, unpacking and installing software (and kernel source), the Advanced Packaging Tool package manager allows you to manipulate these more easily as discrete packages. Note that there are other package managers; the Debian family of linux (including Ubuntu) uses dpkg and apt.

Dpkg is a lower level program which installs, uninstalls and queries status of installed packages. Packages are contained in *.deb files. Apt wraps and extends the dpkg program, managing indexes of available software (and versions), indexes of installed software (and versions), and downloading updates. There are GUIs which then abstract apt, such as the Ubuntu Software Center.

Example apt command used to prepare for a kernel build:

```
$sudo apt-get install fakeroot kernel-wedge build-essential makedumpfile
      kernel-package libncurses5 libncurses5-dev
```

Apt-get is the main Apt tool. Here it is given the "install" command to install the list of named packages that follow. There are other commands, such as remove, purge, update, upgrade, check (and others). There are other apt tools, such as apt-cache, which acts on the local index of available packages, synchronized from the web repository (via "update").

Make

By convention, Linux uses the 'make' program to control large compilation jobs. Make processes a configuration script (makefile) which indicates how a project should be built, but in a structural rather than stepwise manner. The make 'language' is primarily rules of dependencies between files – Make processes these into a series of optimal ordered steps which take into account not just the dependencies, but avoids redundant compilation (eg two source files with same include) , unnecessary compilation (eg. a compiled object is up-to-date), and can parallelize steps for improved performance.

Make can use variables, lists, substitutions, wildcards, and conditionals. Most importantly, this means that the rules can be made implicit, or describe a class of file and how it should be processed, and Make will evaluate the implied dependencies. Each rule is defined as a target and list of requirements. For each rule, there can be a "recipe", or set of instructions on how to use the required files (eg, source code) to achieve the target. The target can be a compiled object file, or a defined objective like "**config**ure the build" or "**clean** out previously compiled code".

Below is a trivial example makefile:

```
helloworld.o : helloworld.c
        gcc –o helloworld.o helloworld.c
```

The target helloworld.o depends on source code helloworld.c. The 'recipe' is the command to compile with gcc. Make will not re-compile helloworld.c unless the file dates are changed.

Below is a more sophisticated makefile, and the one that builds our kernel module:

```
obj-m += hello.o

all:
        make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules

clean:
        make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
```

Here are two targets (as tasks or objectives): “all” which builds the code (default) and “clean” which clears out previous build files. No file dependencies are presented for either target. The recipes build a shell command which actually calls Make on the kernel’s own makefile (in the header source) to handle compilation. The top line is a variable assignment which adds the target (hello.o) to the list of modules to be compiled, and this list is processed by the kernel header makefile. Make finds the hello.c source in directory of the original makefile, and processes the build instructions for modules.

There are many ways to use Make’s capabilities beyond its intended purpose; it can process any task where dependencies must be evaluated to drive conditional scripting.

Module Management

There are several commands for managing modules at the shell. **Insmod** attaches a module to the running kernel (from a *.ko object). This can be verified by querying the running modules with **lsmod**. The module can be removed with **rmmod**. Finally, details about the module object (ie. *.ko) can be queried with **modinfo**.

Grub

Grub stands for “GRand-Unified Bootloader” and it is one of the standard boot-loaders for linux (also LILO). In essence, it allows the user to select an installed operating system at boot-time. Part of Grub replaces the master boot record (MBR) on the drive, which loads the main Grub program and configuration file(s). The configuration file specifies what operating systems can be booted and how to boot them- specifically, in which partition they reside, and the kernel image files that should be loaded from it. In the case of linux, it specifies the initrd/initramfs pre-boot image and the full, compressed vmlinux kernel image.

The script, **update-grub** may be run from the shell to rebuild grub.cfg, which defines the boot menu and how each OS selection must be booted. This tool can be used when replacing the kernel to insure Grub loads the correct kernel files.

Relevant files for GRUB 2:

File / Directory	Description
/boot/grub/	Grub initialization files (modules, configurations)
/boot/grub/grub.cfg	The selection list of OS’s and boot instructions
/etc/grub.d/	Ordered scripts used by **update-grub** to automatically rebuild Grub boot option list
/etc/default/grub	Grub program settings

2 Implementation Tasks

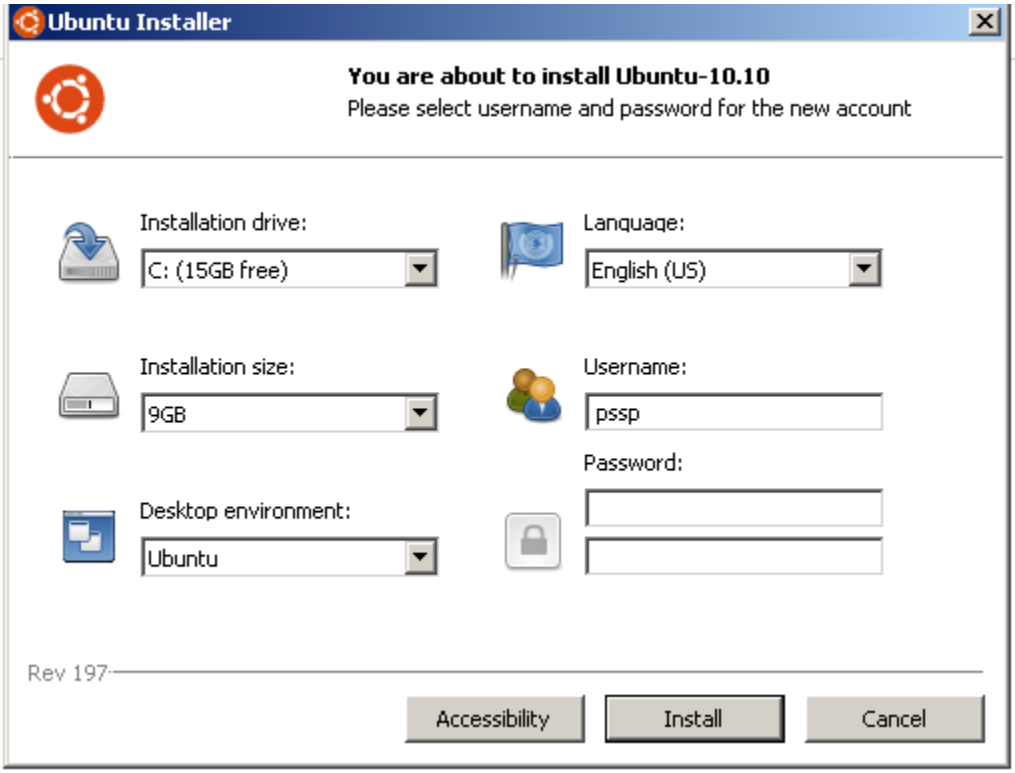
2.4 Installing Linux Ubuntu 10.10

2.4.1 Download the Ubuntu installer for windows which could run alongside with windows operating system from http://www.ubuntu.com/desktop/get-ubuntu/windows-installer

2.4.2 The download size of the Ubuntu installer for windows is approximately 1.5 MB

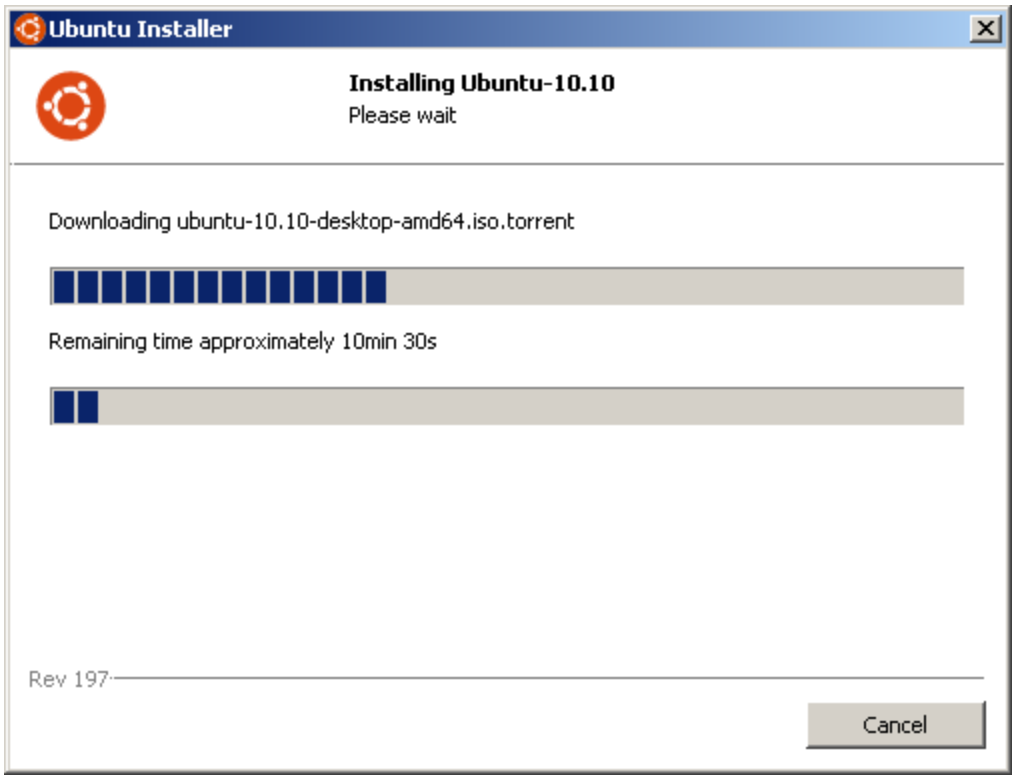
2.4.3 Once you run the Ubuntu installer, a screen will pop up and ask you for the drive where you want to install Ubuntu 10.10.

2.4.4 Make sure you've at least 15 GB free, though the installation will only take 9.5 GB



2.4.5 Click on Install button.

2.4.6 The Ubuntu installer will start the process of downloading Ubuntu operating system package and installing in the drive mentioned.



2.4.7 The Ubuntu installer would prompt you to re-boot the machine in order to completely install Ubuntu.

2.4.8 Once the machine re-starts, the machine will show two boot orders (operating systems)

- Windows (XP, 7 or whatever version you have)
- **Ubuntu Linux 2.6 35-11 generic**

2.4.9 Select Ubuntu Linux, so that installer can complete its install. It will download and install the packages on the machine.

2.5 Installing and Compiling the Kernel source code

2.5.1 Open the terminal and type the apt get command which is used to install the package. Fake root is used to do the file manipulation by faking the root privileges as if the user would have root. Kernel-wedge is used to split the udeb packages to create list of packages i.e. build-essential.

sudo apt-get install fakeroot kernel-wedge build-essential makedumpfile kernel-package libncurses5 libncurses5-dev

2.5.2 Then run the following command

sudo apt-get build-dep --no-install-recommends linux-image-$(uname -r)

This command builds the package that we want to build.

2.5.3 Create a src directory, incase ,it does exist

mkdir ~/src (in case src directory does not exist)

2.5.4 Getting into src directory

cd ~/src

2.5.5 Downloading the source package. Uname –r gives us the kernel version

apt-get source linux-image-$(uname -r)

cd linux-2.6.35

2.5.6 Copying the same .config file as root is running and using it

cp -vi /boot/config-`uname -r` .config

2.5.7 Configure the kernel source.

make menuconfig

2.5.8 In case, Ubuntu is installed on a dual core processor the speed of the compilation can be increased by setting the concurrency level to a small integer value. Thumb rule is number of cores + 1

export CONCURRENCY_LEVEL=3

2.5.9 Preparation step before compilation.

make-kpkg clean

2.5.10 Compiles the kernel with new version (custombuild) by faking the root privileges and create linux image and header files at usr/src directory.

fakeroot make-kpkg --initrd --append-to-version=custombuild kernel-image kernel-headers

2.5.11 Set the new custombuild kernel as the new kernel. Use the packet manager to install the image and header debian packages.

sudo dpkg -i linux-image-2.6.35.11-custombuild_2.6.35.11-custombuild-10.00.Custom_amd64.deb

sudo dpkg -i linux-headers-2.6.35.11-custombuild_2.6.35.11-custombuild-10.00.Custom_amd64.deb

2.5.12 Manually create the custombuild initramfs

sudo update-initramfs -c -k 2.6.35-11-generic-custombuild

2.5.13 Add the initramfs to grub.cfg (to load the custombuild image and show up at the boot sequence)

sudo update-grub

2.6 Creating and building a Custom Kernel Module Into the Kernel

2.6.1 Creating a custom module hello.c

vi hello.c

Add the below source code to it:

```
#include <linux/module.h>

#include <linux/kernel.h>

int init_module(void)

{

   printk(KERN_INFO "hello.init_module()\n");

   return 0;

}

void cleanup_module(void)

{

   printk(KERN_INFO "hello.cleanup_module()\n");

}
```

2.6.2 Create new Makefile as follows:

vi Makefile

Hello.o is added to the list of modules that kernel would build (obj-m)

```
obj-m += hello.o

all:

        make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules

clean:

        make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
```

Save and close the file.

2.6.3 Compile hello.c module using make

make

This will create the hello.ko module which has additional information where the module is kept. See lines prefixed with CC[M] and LD[M]

```
work@ubuntu:~$ make
make -C /lib/modules/2.6.35-22-generic/build M=/home/work modules
make[1]: Entering directory `/usr/src/linux-headers-2.6.35-22-generic'
  CC [M]  /home/work/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/work/hello.mod.o
  LD [M]  /home/work/hello.ko
make[1]: Leaving directory `/usr/src/linux-headers-2.6.35-22-generic'
work@ubuntu:~$
```

2.6.4 List down the files

ls –l

```
work@ubuntu:~$ ls -l
total 67784
-rwxrwxrwx 1 work work     2303 2011-04-22 11:09 compile_kernels.sh
drwxr-xr-x 2 work work     4096 2011-04-22 16:37 Desktop
drwxr-xr-x 2 work work     4096 2011-04-07 17:36 Documents
drwxr-xr-x 2 work work     4096 2011-04-22 12:40 Downloads
-rw-r--r-- 1 work work      179 2011-04-07 17:32 examples.desktop
-rw-r--r-- 1 work work      233 2011-04-22 16:30 hello.c
-rw-r--r-- 1 work work     2429 2011-04-22 16:34 hello.ko
-rw-r--r-- 1 work work      690 2011-04-22 16:34 hello.mod.c
-rw-r--r-- 1 work work     1808 2011-04-22 16:34 hello.mod.o
-rw-r--r-- 1 work work     1150 2011-04-22 16:34 hello.o
drwxr-xr-x 3 work work     4096 2011-04-10 21:41 Leena
-rwxrwxrwx 1 work work 69331459 2011-04-22 12:39 linux-2.6.35.12.tar.bz2
-rw-r--r-- 1 work work      182 2011-04-22 16:34 Makefile
-rw-r--r-- 1 work work       27 2011-04-22 16:34 modules.order
-rw-r--r-- 1 work work        0 2011-04-22 16:34 Module.symvers
-rw-r--r-- 1 work work      234 2011-04-15 12:49 morning.c
drwxr-xr-x 2 work work     4096 2011-04-07 17:36 Music
drwxr-xr-x 2 work work     4096 2011-04-07 17:36 Pictures
drwxr-xr-x 2 work work     4096 2011-04-07 17:36 Public
drwxr-xr-x 2 work work     4096 2011-04-07 17:36 Templates
drwxr-xr-x 2 work work     4096 2011-04-07 17:36 Videos
work@ubuntu:~$
```

2.6.5 Please note that hello.ko is kernel module file. Modinfo command gives us the information about the module that has been just created

modinfo hello.ko

```
work@ubuntu:~$ modinfo hello
filename:       /lib/modules/2.6.35.11-custombuild/hello.ko
license:        GPL
srcversion:     317F6D050B699C4DA4E7193
depends:
vermagic:       2.6.35.11-custombuild SMP mod_unload modversions 686
work@ubuntu:~$
```

2.6.6 Installs the hello module into the kernel it using the privileged user or as root

sudo insmod hello.ko

2.6.7 List down the loaded modules

lsmod

```
work@ubuntu:~$ lsmod
Module                  Size  Used by
hello                    618  0
binfmt_misc             6599  1
rfcomm                 33811  4
sco                     7998  2
bnep                    9542  2
l2cap                  37008  16 rfcomm,bnep
parport_pc             26058  0
ppdev                   5556  0
joydev                  8735  0
snd_hda_codec_analog   59649  1
arc4                    1165  2
snd_hda_intel          22107  2
iwl3945                85550  0
snd_hda_codec          87552  2 snd_hda_codec_analog,snd_hda_intel
snd_hwdep               5040  1 snd_hda_codec
```

2.6.8 View the log files to verify if creation of hello module shows up

tail -f /var/log/messages

```
work@ubuntu:~$ tail -f /var/log/messages
Apr 22 15:49:56 ubuntu kernel: [   32.131139]     (5735000 KHz - 5835000 KHz @ 4
0000 KHz), (300 mBi, 3000 mBm)
Apr 22 15:49:56 ubuntu kernel: [   32.131144] cfg80211: Regulatory domain: 98
Apr 22 15:49:56 ubuntu kernel: [   32.131146]     (start_freq - end_freq @ bandw
idth), (max_antenna_gain, max_eirp)
Apr 22 15:49:56 ubuntu kernel: [   32.131149]     (2402000 KHz - 2472000 KHz @ 4
0000 KHz), (300 mBi, 2700 mBm)
Apr 22 15:49:56 ubuntu kernel: [   32.131194] cfg80211: Current regulatory domai
n updated by AP to: US
Apr 22 15:49:56 ubuntu kernel: [   32.131196]     (start_freq - end_freq @ bandw
idth), (max_antenna_gain, max_eirp)
Apr 22 15:49:56 ubuntu kernel: [   32.131199]     (2402000 KHz - 2472000 KHz @ 4
0000 KHz), (300 mBi, 2700 mBm)
Apr 22 16:46:14 ubuntu kernel: [ 3411.021099] hello: module license 'unspecified
' taints kernel.
Apr 22 16:46:14 ubuntu kernel: [ 3411.021106] Disabling lock debugging due to ke
rnel taint
Apr 22 16:46:14 ubuntu kernel: [ 3411.021254] hello.init_module()
```

2.6.9 List all modules loaded in the kernel

cat /proc/modules

```
work@ubuntu:~$ cat /proc/modules
aes_i586 7280 1 - Live 0xf8235000
aes_generic 26875 1 aes_i586, Live 0xf8229000
rfcomm 32768 4 - Live 0xf8217000
sco 7739 2 - Live 0xf815e000
bnep 9305 2 - Live 0xf8130000
binfmt_misc 6413 1 - Live 0xf812c000
l2cap 35806 16 rfcomm,bnep, Live 0xf81dd000
parport_pc 25956 0 - Live 0xf80f0000
```

```
serio_raw 4022 0 - Live 0xf8137000
soundcore 880 1 snd, Live 0xf80be000
video 18287 0 - Live 0xf80a6000
agpgart 31489 3 ttm,drm,intel_agp, Live 0xf8076000
snd_page_alloc 7088 2 snd_hda_intel,snd_pcm, Live 0xf8033000
i2c_algo_bit 4960 1 radeon, Live 0xf864f000
output 1883 1 video, Live 0xf8608000
hello 638 0 - Live 0xf816c000
usbhid 36220 0 - Live 0xf8110000
hid 66453 1 usbhid, Live 0xf80fd000
e1000e 130562 0 - Live 0xf8084000
work@ubuntu:~$
```

2.7 Loading the modules

The list of kernel modules can be found out at **/etc/modules** that are loaded during the boot time.

2.7.1 Create a directory for hello module

sudo mkdir -p /lib/modules/$(uname -r)/kernel/drivers/hello

2.7.2 Copy hello module

sudo cp hello.ko ../../lib/modules/$(uname -r)/kernel/drivers/hello/

2.7.3 Edit /etc/modules file to add an entry

sudo vi /etc/modules

2.7.4 Add hello module to the list of kernel modules that will be loaded at boot time:

Hello

2.7.5 Reboot and use lsmod to ensure if the module is loaded or not.

cat /proc/modules

3 Challenges and Learned Lessons

During the course of this project, we as team faced some challenges that required both time and effort to be resolved. Some examples of those challenges are:

- **Downloading a source code that matches the kernel version:**
  One of the issues we faced in this project is downloading a separate source code that matches our kernel version which is ubuntu 10.10. Since we are well aware that downloading the inappropriate source code will prevent us from compiling the new kernel to-be installed and the built-user module, we began searching. During our search, we found many versions of Linux source code, but we decided to download the latest version of Linux source code, which we found at http://kernel.org/.

- **Using commands that run in our installed Ubuntu 10.10 Linux:**
  While working with the ubuntu environment and terminal, we found that not all Linux commands run at ubuntu 10.10. Some commands run only in older versions of Linux; some examples of which are make menuconfig, make xconfig, and make gconfig. Therefore, we had to search for alternative commands or different variations of the non-working commands that could run in ubuntu and accomplish the same job. For instance, we found that 'make config' configures the kernel in a similar way like the above three commands, so we applied it in our project. In addition, we found that some commands may work independently in some Linux versions; on the other hands, those same commands only work, in ubuntu 10.10, by adding to them another commands to help them execute and run. For example, we tried to run commands like 'apt-get' and 'make modules', but we could not have them run until we added the executing command 'sudo' to the beginning commands.

- **During compilation, only the kernel and system modules got compiled without the built-user module:**
  The first time we encountered this issue, it was because the built-user module was still not inserted into the kernel; therefore, it did not get linked to other modules in the kernel. Also, even when it is inserted, it needs to be in a particular place in order to be located.

- **The compilation process crashed/stopped and did not finish successfully:**
  This issue of compilation process crashing after around an hour of lunching kernel compilation happened twice during this project. The first time, it was because we did not have enough allocated disk space in the ubuntu partition. So, we learned that since compilation includes preprocessing, compiling, assembly, and linking, it does need enough disk space. The second time we faced this issue, it was because we chose the default configuration and did not change anything in the settings. We were able to resolve these issues by conducting the compilation on

a different laptop that has more disk space specified for ubuntu partition. Also, we used the package manager for Debian Linux (dpkg) which installed configuration packets with settings.

- **After reboot, the new installed kernel does not boot:**
  We found that the main reason behind this issue is because we used some commands that were not ubuntu specific in the compilation. But, once we reviewed our steps and used ubuntu specific instructions, we were able to see and boot our installed kernel.

- **After reboot, when the new installed kernel boots, the built-user module does not load:**
  Another problem that we encountered was that after we successfully had the new kernel boots, we found out that our built-user module did not load. After a careful review of the project and search, we found that one needs to make sure that the module name is added to /etc/modules file, which contains the names of all the modules that are loaded at boot time. We also need to make sure that the object file (e.g. hello.ko) generated from the module file (e.g. hello) is added to /lib/modules dir, which is the kernel source tree. Also, when adding the module, one needs to be logged in as root to module in kernel which requires one to enter his password. After inserting the module into the kernel, we used 'insmod' command to list all the modules inside the kernel and confirm the correct insertion of the user module.

- **Setting the kernel configuration properties:**
  Before compiling the kernel, we had to set the kernel configuration. When we entered the commands to compile the kernel, a long set of yes/no configuration questions were displayed for us to either change the configuration by typing yes or not for each question, or going along with the default configuration settings by pressing the enter key.

  When we tried to go with the default configuration and not modify any settings, the kernel compilation did not proceed successfully and we got many displayed errors before compilation failure.

  After some search, we found out that the configuration settings that need to be changed differ based on the kernel version. In our project, the package manager for Debian Linux (dpkg) took care of configuration by managing and installing individual configuration packets.

Dealing with the challenges of this project have taught us The following lessons about the project as a whole and about working as a team:

- We learned to be patient. For example, when we were working on the compilation step, it took us around two hours to finish the compilation successfully and without any errors.

- We also learned to try different approaches and variations of commands when something is not running correctly or not giving us the expected results. For example, when we faced some problems in configuration, we tried different options to make it work.
- We learned that conducting a search and reading articles related to our project are necessary to retrieve information about the ubuntu environment, ubuntu kernel, and encountered issues.
- As a group, we learned that communication is essential to get the project done. Holding meetings, discussing the different phases of the project, and dealing with the encountered problems and working as a team to resolve is a key for the project's success.

References

http://www.ibm.com/developerworks/linux/library/l-linuxboot/

http://en.wikipedia.org/wiki/Linux_startup_process

http://en.wikipedia.org/wiki/Dpkg

http://en.wikipedia.org/wiki/Advanced_Packaging_Tool

http://www.gratisoft.us/sudo/intro.html

http://en.wikipedia.org/wiki/Make_%28software%29

http://en.wikipedia.org/wiki/Topological_sorting

http://www.wlug.org.nz/MakefileHowto

http://linuxdevcenter.com/pub/a/linux/2002/01/31/make_intro.html

http://www.gnu.org/software/make/manual/make.html

http://en.wikipedia.org/wiki/GNU_GRUB

http://www.dedoimedo.com/computers/grub.html

https://help.ubuntu.com/community/Grub2#Creating%20the%20Custom%20Menu

http://tldp.org/LDP/tlk/sources/sources.html

http://tldp.org/LDP/khg/HyperNews/get/tour/tour.html

http://tldp.org/HOWTO/Linux-i386-Boot-Code-HOWTO/init_main.html