# ECE 468/668 Linux Kernel Lab

Peng Xu

In this lab you will learn how to create and use a virtual machine in VirtualBox and build a new Linux kernel image for this virtual machine. You will also learn how to modify the kernel and add a custom kernel function into it which can be called from your program.

Since operations on Linux kernel many cause system failures, such operations can only performed by system administrators. To prevent accidental operation by unauthorized users, changing most system files in Linux system needs a root privilege. However, we as students do not have root privilege on the lab PCs. A reasonable solution is using a virtual machine. Using virtual machine will not affect most steps in building and modifying kernel and thus your experience gained from this lab can be applied to physical PCs.

## A. Create a virtual machine with given initial image

1. Installing a system on virtual machine usually takes a long time. The time will be longer on virtual machines since it is a simulation. For this reason, a base virtual machine disk image is prepared for you so you can get started in a few minutes. This disk image contains a Linux system with applications for compiling a Linux kernel.
2. Create a new directory for storing the image file and copy the base virtual machine image to that directory.
    a. mkdir ~/lab668-4
    b. cp /users/pxu/lab668/lab668base.vdi ~/lab668-4
3. Set up a new virtual machine
    a. Start Virtual Box from the Application menu or by typing "VirtualBox" in terminal.
    b. Click "New" button in the Oracle VM VirtualBox Manager. Use the setting listed below
        i. VM Name and OS Type
            1. Name: Choose one you like. e.g. lab668
            2. OS Type:
                a. Operating System: Linux
                b. Version: Ubuntu
        ii. Memory
            1. Base memory Size:
                a. (default)
        iii. Virtual Hard Disk
            1. [x] Start-up Disk
            2. [x] Use existing hard disk
            3. Choose the virtual machine image file (the one you copied) in your VirtualBox virtual machine directory.

iv. Now you can see a virtual machine in Virtual Machine Manager. We still need to change some settings to make it work

4. Change hard disk attachment
    a. By default the virtual hard disk is attached under IDE controller in some VirtualBox. However, the virtual machine is built with the hard disk under SATA controller and the kernel does not have the necessary IDE controller driver. So it will not boot correctly.
    b. Right click on the virtual machine you just created and select **Settings**.
    c. In the list on the left, choose **Storage**.
    d. If you find the hard disk lab668base.vdi is under **IDE controller**, do the following steps, else just skip to 5.
    e. Right click on the hard disk and **Remove Attachment**.
    f. Add a disk on to **SATA Controller** by right click and select **Add Hard Disk**.
    g. Use the existing file in VirtualBox virtual machine directory.
    h. OK to finalize change.
5. Start your virtual machine by click "**Start**" on VirtualBox Manager.
    a. You will see the login screen after some kernel messages.
    b. The user name is "user" and password is "password" (both without quote).
    c. It is not a good idea to shut down the virtual machine by close window. The file system changes in memory will not be able to write back into the disk and loss or damage of files will be the consequence.
        i. To shut down, type `sudo init 0` in command line interface. When asked for password, just type "password" (without quote).
        ii. To reboot, type `sudo init 6`.
        iii. Some background knowledge. The init command will change the system *runlevel* (0/6 here) which is a notion inherited from Unix SysV style initialization scripts. It is a standard for 0 to be shutting down system and 6 to be rebooting system. However, the definition of number between 1 and 5 varies from distribution to distribution. There is a Wiki page that gives a good explanation if you search for "runlevel".
6. If you broke something in the virtual machine that cannot be recovered. Just repeat the steps above to get a new one. However, the work done already will be lost.
7. A trick here is setting up snapshot for your virtual machine regularly so you do not have to restart over from the beginning. For details about how to set up snapshot, consult the Help file of VirtualBox.

## B. Build a trimmed down kernel

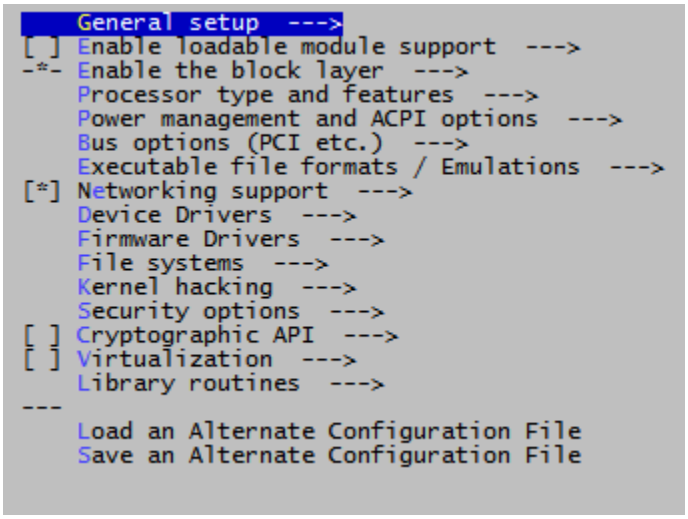
1. In this section, you will go through the steps below and build a kernel image following a minimalist's approach.
    a. Here kernel trimming means turning off unnecessary options a kernel rebuild. The kernel image file size and the initialization ram disk file size can be greatly reduced. In this lab you will see the kernel image file reduced to around 1M (usually a few MB). Moreover, after kernel trimming there is no initialization ram disk file (usually tens of

MB) at all. Although hard disk is getting cheap these days and MB size space probably means nothing to your laptop, embedded hardware may still prefer a smaller kernel footprint.

b. Trimming down helps the kernel run faster since a bigger portion of the kernel binary can reside in the L2 cache of CPU.
c. A trimmed down kernel can be built in significantly less time.
d. The process of trimming down kernels helps familiarizing the purpose of each component in the kernel.

2. Let's start. After login, go to the kernel source directory prepared for you. Shaded text marks the command to type.
   a. `cd ~/kernel/linux-2.6.32.55-1`
3. Initialize the directory structure (clean everything)
   a. `make mrproper`
4. Since we prefer to build a minimal kernel, it is not so bad to start with a configuration that has nothing selected.
   a. `make allnoconfig`
   b. Other options of make can be viewed by `make help`.
5. Start a menu style configuration system, and check all items listed below.
   a. `make menuconfig`

   You will see something like this:




   b. Below is a list of options you need to check on ([*]) unless it specifies unchecking something explicitly. Ellipses are used to shorten the list item in case there is no ambiguity.
   c. General Setup
      i. Local version (Fill in your customized version here. The content should be a valid file name, so you cannot put in "*", "?", "|", etc .)
      ii. Automatically append….
      iii. Kernel compression …
         1. Bzip2 for better compression rate.

iv. System V IPC
v. UTS Namespace
vi. IPC Namespace
vii. Kernel performance events and counters
    1. Kernel performance events and counters

d. Enable the block layer
    i. IO scheduler
        1. Deadline ….
        2. Default …
            a. Deadline…

e. Processor Type and Feature
    i. Single Depth WCHAN
    ii. Generic x86 support
    iii. Preemption Model
        1. Voluntary ….
    iv. Low address space to prevent…
        1. Fill 65536
    v. Reserve Low 64K …
    vi. Enable seccomp

f. Bus options
    i. PCI support
        1. PCI access mode (any)
    ii. Enable deprecated pci_find_*API

g. Executable file format ….
    i. Kernel support for ELF binaries

h. Networking support (remember to select this at top level)
    i. Networking options
        1. Packet socket
        2. Unix domain socket
        3. TCP/IP Networking
        4. Uncheck everything not listed above
    ii. Uncheck wireless if it is checked automatically.

i. Device Drivers
    i. Generic driver options
        1. Create a kernel maintained …
           (if you did not see this option, come back later)
        2. Automount devtmpfs …
           (if you did not see this option, come back later)
        3. Prevent firmware from …
        4. Include in-kernel firmware blobs…
    ii. SCSI device support
        1. SCSI Device support

2. Legacy /proc …
3. SCSI Disk support
4. Probe all LUNS
5. Asynchronous SCSI scanning
6. SCSI low level driver

iii. Serial ATA and Parallel …
1. AHCI …
2. Uncheck everything else

iv. Multi-device driver support
1. Raid support
2. Autodetect raid array…

v. Network device support
1. Ethernet (1000 Mbit)
a. Intel Pro 1000 Gigabit…
2. Uncheck other items

vi. Character device
1. Support for binding and unbinding …
2. Support multiple instances of devpts
3. Hardware Random Number generator…
4. Uncheck everything else on this screen.

vii. HID Devices
1. Generic …

viii. Real time clock …
1. Allow for the default options

ix. DMA engine support

j. File systems
i. The extend 4 (ext4) file system
ii. Uncheck the “ext4 extended attributes”
iii. Dnotify support
iv. Inotify support for userspace
v. Pseudo file system
1. /proc/kcore …
2. Virtual memory file system

vi. Uncheck “network file system”

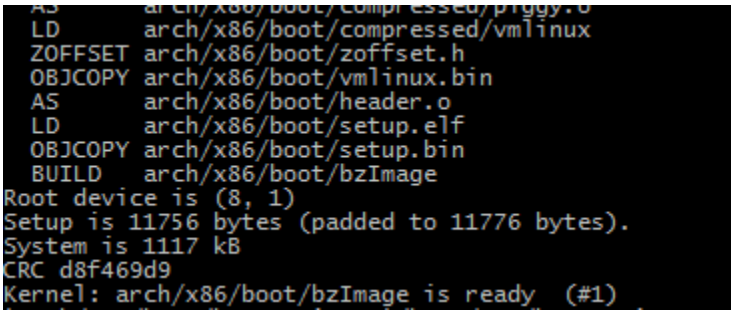
k. Kernel hacking
i. Show timing information on printks
ii. Filter access to /dev/mem
iii. Allow gcc to …

l. Exit

m. Your settings will be saved to .config file.

6. Make the kernel
a. `make bzImage`

b. Wait for a while. After compilation is finished, it will say something like this




c. Screen above means the compiled kernel image size is 1117kB and the path is arch/x86/boot/bzImage. This screen shot is not taken with the setting above, so your mileage may vary on kernel image size.

d. If anything goes wrong in the build process, try to figure out what is going wrong and fix it. If you want to do it over again, clean the directory by `make clean` and go back to step (2) if you want to redo the configure step or (4) if you just want to rebuild again cleanly.

7. Copy the kernel image to /boot and name it appropriately.
    a. As by default the kernel is named bzImage, you need to give it a new name to distinguish it from other kernels. Linux kernel file naming convention are listed below
        i. Kernel files is usually named like vmlinuz-[official_version][local_version].img. The vmlinuz means it is a virtual memory enabled (*vm*-) linux (-*linu*-) compressed (-*z*) kernel. The official version is determined by the Linux source file version and the local version is usually determined by people how build the kernel to distinguish it from other kernel with the same official version.
        ii. Official version of the source used in this lab is 2.6.32.55.
        iii. Local version is what you put in “General Setup/Local Version” in the configuration part.
        iv. E.g. if I have the local version to be “-lab668”(without quote), the Linux image file name should be: vmlinuz-2.6.32.55-lab668.img
    b. `sudo cp arch/x86/boot/bzImage /boot/vmlinuz-2.6.32.55[local_version].img`
        i. The `sudo` is added because privilege is needed to perform copy operation into system directory /boot.
        ii. Remember to replace the [local_version] with what it should be.
8. Setting up bootloader. Now the kernel image is in the right place. However, the bootloader (grub in our set up) did not know it yet.
    a. The setting file for grub is at `/boot/grub/menu.lst`. By editing this file, you can change the boot menu and add your kernel into the menu.
    b. Try figuring out how to edit this file
        i. Available editors are `nano` and `vi`.
        ii. The original setting in the file and some Google search will be of great help.
        iii. Root privilege is needed as well.
    c. Creating backup or making snapshot before this step is recommended if this is the first time you do it.

9. Reboot the virtual machine by `sudo init 6` and enjoy your new kernel.
    a. During reboot you will see another boot option appear in the menu. Select it.
    b. If everything goes well, you should see the login prompt at this time.
    c. Login and type in the command `uname -a`, you will see the kernel version has your own local version on it.
    d. You just finished configure, build and install the kernel. **Cheers!**

## C. Adding a kernel function into kernel

1. In this part of the lab, you will add a custom function into the kernel that outputs strings into the kernel messages. Kernel message can be viewed by typing command `dmesg`.
2. Background for Linux kernel function:
    a. Modern operating system all adopted the notion of privilege level to keep a multi-process, multi-user system maintainable. The user application, which include the desktop environment, firefox, terminal window, gcc compiler and the program you wrote for previous three labs, etc., only runs in user mode, while most of the underlying service related to system memory and device management is in the kernel. Most user mode programs need these services to perform ordinary jobs. For example, fopen, fread and fwrite a file on hard disk needs the ability to access the hard disk controller. This means the user program needs to delegate these operations to kernel. Ideally, the user program can directly link to kernel function either statically or dynamically and call into kernel functions to perform these operations in runtime.
    b. However, there are two factors stopping the user function from doing so
        i. Development of kernel and user program is usually independent. Changing one or the other can be made individually in most cases. A standard single point interface for function call will be beneficial in that it greatly simplifes system integration.
        ii. User program run in non-privileged level do not have access to kernel memory, and thus cannot directly call into kernel.
    c. Solution in Linux: Linux has a call gate called system call (syscall) implemented with software interrupt. You may think syscall is formed by a big table of function entries. Each index in the table corresponds with a kernel function. In C library most of the syscall are encapsulated in functions like fopen and fclose, so you do not have to deal with software interrupt or syscall index. The Linux headers provide a syscall function so you can call to an arbitrary kernel function with an index.
3. First, you need a function that outputs the message. This function will be compiled in to Linux kernel.
    a. Create a C source file that contains your new kernel function. Name the file printmsg.c, and put it into the “kernel” directory inside the Linux kernel source directory, i.e. the directory “~/kernel/linux-2.6.32.55-1/kernel”. All the future reference to directory in kernel modification and compiling context assume a current path at “~/kernel/linux-2.6.32.55-1/”, thus “kernel” directory means “~/kernel/linux-2.6.32.55-1/kernel” and “include” directory means “~/kernel/linux-2.6.32.55-1/include”, etc.

b. The file content should be like this.

```
#include <linux/kernel.h>   /* for printk */
#include <linux/syscalls.h>  /* for SYSCALL_DEFINE1 macro */


SYSCALL_DEFINE1(printmsg, int, i)
{
    printk(KERN_DEBUG "Hello! This is ECE %d Lab 4", i);
    return 1;
}
```

c. The kernel.h is needed because `printk` is called. The `syscalls`.h is for the `SYSCALL_DEFINE1` macro. This macro will be expand into the definition of function and a few other facility for kernel debugging. The number "1" signifies the function take one parameter. Thus if later you want to create a syscall function that takes two parameters, you need to use `SYSCALL_DEFINE2`.

d. The function `printmsg` just outputs a string and return an interger 1. However, since this is in kernel, it uses printk, another kernel function, instead of printf, which is a C standard library function. The `KERN_DEBUG` macro signifies that this message is for debug purpose. Debug message will be recorded but not be posted on user's screen immediately, thus is less annoying. There are different types of messages defined in the header file "include/linux/kernel.h".

e. Be creative about the message here.

4. You have a source file, however the make utility do not know the existence of this file yet. You have to add it into the dependence tree by editing the make file.
    a. Since your file is under "kernel" directory, edit the "Makefile" in the same directory.
    b. The beginning of file should reads

```
obj-y   = sched.o fork.o exec.o …
          …
obj-y  += groups.o
          …
```

    c. Just after the line "obj-y += groups.o", add a line "obj-y += printmsg.o". The make utility will compile your printmsg.c to get printmsg.o and then add it into the kernel binary.
5. Remember the syscall has a table of function entries? It is necessary to add your function to that table to enable it to be called.
    a. Open the table definition file "arch/x86/kernel/syscall_table_32.S". The ".S" extension means it is an assembly language source file.
    b. You will see a long list of kernel function mostly begin with "sys_". Some of the function has a /* <number> */ comment after it. This number is the index of the kernel function. The index starts from 0. For the function without a number comment, just count up or down from the nearest one.
    c. Go to the end of the file and add one line " .long sys_printmsg". The "sys_" prefix is required as it is added automatically by the SYSCALL_DEFINE1 macro. Moreover, find the index of your function by counting from nearest one. Remember it. If you are using the same kernel source as I do when I write this note, the number should be 337.

d. Please note that this is only for kernel running on x86 architecture CPU (suggested from the path arch/x86/… ). If you need your syscall to be cross-platform, editing the corresponding table for that architectures as well.
e. You have finished modifying the kernel source part.

6. Build the kernel again and add another boot option for it.
7. Restart the computer with newly added kernel with your own kernel function.
8. Now you have to build a program to calls your special kernel function.
    a. This is going to be a user mode program, thus it do not have to be in the kernel source directory. Place the source code anywhere you want.
    b. The recommended source code

```
#include <linux/unistd.h>
#define __NR_printmsg    <the_index>

int printmsg(int i){
    return syscall(__NR_ printmsg, i);
}
int main(int argc, char** argv)
{
    printmsg(668);
    return 0;
}
```

    c. Replace "`<the_index>`" with the index of your kernel function.
    d. Compile this program using gcc.
9. Run the program and validate the effect by typing `dmesg |tail`, which shows the last few lines of the kernel message. You should find your special message among other system messages. Below is an example of results. Notice that my kernel function is slightly different.

```
user@linuxlab668:~$ dmesg|tail
[  459.062064] HELLO NUMBER IS 19
[  459.714493] HELLO NUMBER IS 19
[  460.227283] HELLO NUMBER IS 19
[  462.259129] HELLO NUMBER IS 20
[  466.144298] HELLO NUMBER IS 21
[  467.145658] HELLO NUMBER IS 23
[  468.104750] HELLO NUMBER IS 24
[  469.465539] HELLO NUMBER IS 25
[  470.479499] HELLO NUMBER IS 26
[  471.622629] HELLO NUMBER IS 27
user@linuxlab668:~$ _
```

## D. Summary

In this lab, you learned a lot about Linux and Linux kernel. Started from create a virtual machine for you own experiment, you went through the steps for configuring an extremely trimmed down kernel, building the customized kernel, install the kernel into system and setting up bootloader and finally adding a syscall into the kernel. Now you graduated from Lab 4. Congratulations!