# Lab BSY IO - II

## Introduction & Prerequisites

This laboratory is to learn how to:

- Learn how to create and install a custom kernel.
- Learn how to create and install a custom kernel module.

The following resources and tools are required for this laboratory session:

- Any modern web browser,
- Any modern SSH client application
- OpenStack Horizon dashboard: https://ned.cloudlab.zhaw.ch
- OpenStack account details: please contact the lab assistant in case you already have not received your access credentials.

Important access credentials:

- Username to login with SSH into VMs in ned.cloudlab.zhaw.ch OpenStack cloud from your laptops
    - **`ubuntu`**

## Time

The entire session will take 90 minutes.

## Task 0 – Setup

In order to get started, create a VM instance from the standard image with the largest flavor size available. Access the VM with SSH using the username `ubuntu`. Once you have SSH'ed into the VM, change your shell to a root shell (`sudo -i`). This will not require you run '`sudo`' in front of every command that requires root/system level privilege[1]. Most of the tools should be installed on the VM, missing ones can be installed using "`apt`".

[1] This is for convenience and not recommended to do on production systems

# Task 1 – Compile and Install a Custom Kernel

In this task you will compile and install a custom Linux kernel and re-do an experiment from the IO-I lab, namely Task 2, Shared Access, experimentation with “ionice”. In this lab you tried to setup priority levels for IO scheduling but these levels were not supported by the default (deadline) IO scheduler of Linux. A different schedule, one that supports priorities, is the BFQ scheduler. It can be loaded as a module or it can be made available as part of the Kernel, if the kernel has been compiled to include it statically. This is exactly the task that you are supposed to achieve.

Download and extract the sources of the Linux kernel from the official website. Choose a version that suits your overall system configuration and the task to be completed. Justify your choice.

Prepare your local build environment, what do you expect to be required? Start from here: https://www.kernel.org/doc/html/latest/admin-guide/README.html#readme

Note that the tools required on Ubuntu can be installed via
`sudo apt-get install build-essential gcc bc bison flex libssl-dev libncurses5-dev libelf-dev`

Configure, build and install a kernel that supports the BFQ scheduler as an integral part, that is compiled statically into the kernel. Start from a config that represents your current running (run-time) system (kernel plus modules in use); mind, that is not the configuration that was used for building (compile-time) the default kernel.

Give your kernel a custom name (the parameter is called CONFIG_LOCALVERSION) that indicates the inclusion of the BFQ scheduler.

Build the kernel and package it with Debian-style packages, then install it on your VM (see slides).

Reboot your system into your custom kernel and execute the following task (copied from the IO-I Lab with minor modifications).

**Shared IO Access**
As soon as access to IO is shared, performance becomes a real issue. There are several tools to analyse IO performance, like iostat and iotop. Familiarize yourself with these two by running them and understanding their outputs.

Now focus on the “dd” tool (man dd). Run a scenario in which you run several parallel instances of dd, each creating a file of 10G each. Before that make sure that you have enough disk space. On your disk, create in parallel a number of files, using the following command. “`dd if=/dev/zero of=/mnt/iotest/testfile_n bs=1024 count=1000000 &`” What exactly is this command doing? What do you have to modify in order to use this command in parallel? Perhaps you want to run this as a shell script, simply put this command line-by-line into a text file and execute the text file (aka bash script file) on the shell. Once load is created, check IO performance. Discuss the results and verify your understanding.

Now experiment with the command “ionice”. Add different schedulers with different priorities to your operations and verify the result. Can you actually run one process faster than another? If yes, explain, if not explain too. Hint, “`less /sys/block/vXXX/queue/scheduler`” and `https://wiki.ubuntu.com/Kernel/Reference/IOSchedulers`

## Task 2 – Compile and Install a Custom Kernel Module

Read the documentation about compiling a Kernel module available here:
https://tldp.org/LDP/lkmpg/2.6/html/index.html

In particular take a look at the section concerning character device drivers:
https://tldp.org/LDP/lkmpg/2.6/html/x569.html

A zip file of the above documentation is also available on Moodle.

Now read the code of a character device module taken from the example above (we made minor changes, they are highlighted):

```
/*
 *  chardev.c: Creates a read-only char device that says how many times
 *  you've read from the dev file
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>    /* for put_user */
MODULE_LICENSE("GPL");
/*
 *  Prototypes - this would normally go in a .h file
 */
int init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

#define SUCCESS 0
#define DEVICE_NAME "chardev"    /* Dev name as it appears in /proc/devices   */
#define BUF_LEN 80        /* Max length of the message from the device */

/*
 * Global variables are declared as static, so are global within the file.
 */

static int Major;         /* Major number assigned to our device driver */
static int Device_Open = 0;    /* Is device open?
                           * Used to prevent multiple access to device */
static char msg[BUF_LEN];    /* The msg the device will give when asked */
static char *msg_Ptr;

static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
```

```
};

/*
 * This function is called when the module is loaded
 */
int init_module(void)
{
        Major = register_chrdev(0, DEVICE_NAME, &fops);

    if (Major < 0) {
      printk(KERN_ALERT "Registering char device failed with %d\n", Major);
      return Major;
    }

    printk(KERN_INFO "I was assigned major number %d. To talk to\n", Major);
    printk(KERN_INFO "the driver, create a dev file with\n");
    printk(KERN_INFO "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);
    printk(KERN_INFO "Try various minor numbers. Try to cat and echo to\n");
    printk(KERN_INFO "the device file.\n");
    printk(KERN_INFO "Remove the device file and module when done.\n");

    return SUCCESS;
}

/*
 * This function is called when the module is unloaded
 */
void cleanup_module(void)
{
    /*
     * Unregister the device
     */
    unregister_chrdev(Major, DEVICE_NAME);
    int ret = 0;
    if (ret < 0)
         printk(KERN_ALERT "Error in unregister_chrdev: %d\n", ret);
}

/*
 * Methods
 */

/*
 * Called when a process tries to open the device file, like
 * "cat /dev/mycharfile"
 */
static int device_open(struct inode *inode, struct file *file)
{
    static int counter = 0;

    if (Device_Open)
         return -EBUSY;

    Device_Open++;
    sprintf(msg, "I already told you %d times Hello world!\n", counter++);
    msg_Ptr = msg;
    try_module_get(THIS_MODULE);

    return SUCCESS;
}

/*
 * Called when a process closes the device file.
 */
```

```
static int device_release(struct inode *inode, struct file *file)
{
    Device_Open--;        /* We're now ready for our next caller */

    /*
     * Decrement the usage count, or else once you opened the file, you'll
     * never get get rid of the module.
     */
    module_put(THIS_MODULE);

    return 0;
}

/*
 * Called when a process, which already opened the dev file, attempts to
 * read from it.
 */
static ssize_t device_read(struct file *filp,    /* see include/linux/fs.h   */
                           char *buffer,    /* buffer to fill with data */
                           size_t length,    /* length of the buffer        */
                           loff_t * offset)
{
    /*
     * Number of bytes actually written to the buffer
     */
    int bytes_read = 0;

    /*
     * If we're at the end of the message,
     * return 0 signifying end of file
     */
    if (*msg_Ptr == 0)
         return 0;

    /*
     * Actually put the data into the buffer
     */
    while (length && *msg_Ptr) {

         /*
          * The buffer is in the user data segment, not the kernel
          * segment so "*" assignment won't work.  We have to use
          * put_user which copies data from the kernel data segment to
          * the user data segment.
          */
         put_user(*(msg_Ptr++), buffer++);

         length--;
         bytes_read++;
    }

    /*
     * Most read functions return the number of bytes put into the buffer
     */
    return bytes_read;
}

/*
 * Called when a process writes to dev file: echo "hi" > /dev/hello
 */
static ssize_t
device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
{
    printk(KERN_ALERT "Sorry, this operation isn't supported.\n");
```

```
    return -EINVAL;
}
```

Use the code above (you'll also find the source file on Moodle) to compile the module using the makefile from the kernel you compiled. Have a look at the slides to see the commands and files needed.

HINTS - you'll need to:

- Create a directory for your module
- Put there the source code of your chardev module (chardev.c)
- Write a module Makefile (see slides)
- Issue the module build request with the appropriate make command

Next, insert the module in the kernel you are running.

Verify that the module is initialized correctly. Where will you see the messages concerning module initialization?

Create the dev file associated with the device

Try reading the dev file multiple times, what do you get?

Look at the /sys system directory and find the module you installed. Check its init state

Remove the module. What happens to the module directory in /sys?

# Cleanup!

**IMPORTANT:** At the end of the lab session:

- **Delete** all OpenStack VMs, volumes, security group rules that are no longer needed. You may want to keep some data for exam preparations.

## Additional Documentation

OpenStack Horizon documentation can be found on the following pages:

- User Guide: https://docs.openstack.org/horizon/latest/