Lab 7: Linux Kernel: Built-in Modules

Samuel Huang
ECEN 449

1. Introduction

The purpose of this lab is to continue working on kernel drivers, in particular adding device drivers to the Linux kernel to load during boot. The lab covers how to add and disable different drivers to the Linux kernel to customize the services available by the kernel upon load in. The lab illustrates how removing different drivers that may not be needed for the system, reduces the size of the kernel image.

2. Procedure

The first step of creating a kernel with a built in driver upon load in is to create a PetaLinux project just like the past few labs. After ensuring the multiply peripheral is configured to the project, the kernel is configured. Importing source code from Linux, to place the source within the project allows for edits to add drivers. Since entries are influenced by dependencies, a multiplier driver is created in the Petalinux project inside of the external Linux source code. The multiplier.c, xparameters.h, and xparameters_ps.h are added to the multiplier driver folder, along with a Makefile and Kconfig. The Makefile runs the multiplier.c, and Kconfig defines the multiplier module. The Makefile and Kconfig of all the drivers is modified as well to include the multiplier driver, and ensure it is added to the final kernel. With the multiplier driver complete, the modified Linux source is linked to the PetaLinux project and the multiplier driver is then enabled to be compiled as built-in to the Linux. Finally the project is built, and the new image.ub is loaded with the previous BOOT.bin and boot.scr.

The new linux project is uploaded to the FPGA, and the resulting kernel is checked to see the built in driver, along with its functionality. The process is then repeated, but during the final project configurations, three unnecessary drivers that do not affect the multiplier functionality are removed from the final kernel configurations. The resulting image.ub is checked and compared to the original to see the effect of removing not critical features has on the size of the file.

3. Results

The continues to expand on previous labs, by providing the same multiplier feature but with a different implementation. This implementation allowed for the multiplier driver to be included in the build, without having to load it later. The functionality of the driver remains unchanged, and continues to communicate with the kernel and userspace, to utilize the multiplication peripheral from lab 4 in user space software. The lab builds upon previous work, but illustrates how to configure custom kernels to include necessary drivers and features. The result of removing unnecessary features from the final kernel configurations, showed the image.ub size to decrease from an initial 18MB to around 16.5 MB.

4. Conclusion

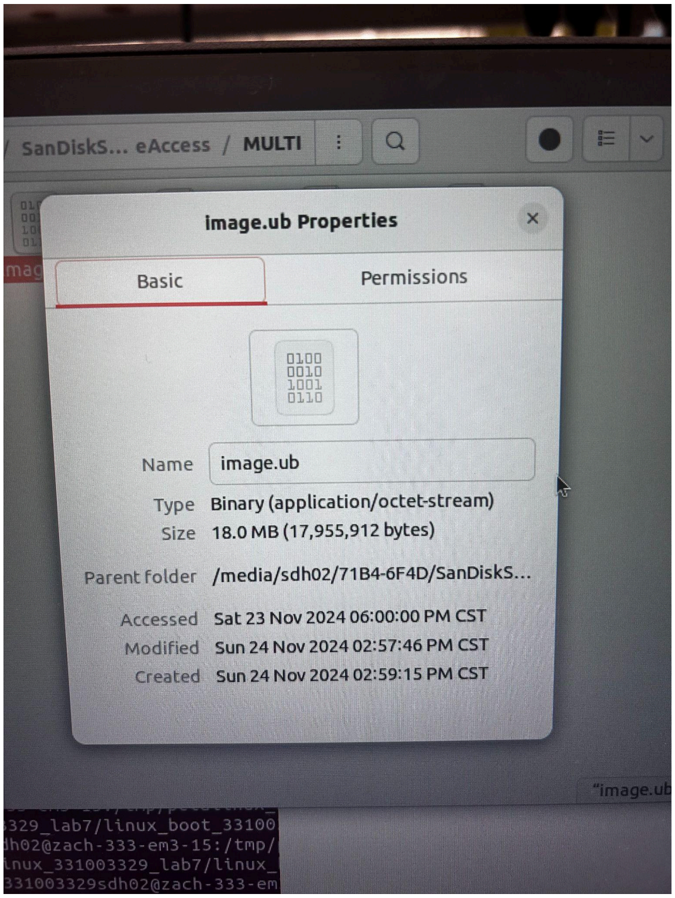
This lab gave practical experience on configuring a Linux kernel. It showed how to add created drivers to the Linux kernel, so that it can be included during the build. The lab also showed how to enable and disable certain features. This part of the lab has the most application for future use, since not every built in kernel feature is necessary for each project. By not including it in the Kernel configurations more space can be saved by excluding the unnecessary drivers. The biggest challenge to this lab was the build process, and getting it to work properly. Ensuring that each step was made to properly set up the kernel and its configurations had a huge impact on the success of the project build. It was not the only factor though, other technical issues with the computers and system had a large impact on the success of the build which was outside of my control. The lab process took a while to ensure everything was properly set up for the build, but the end result showed great potential for different applications of kernel configuration.

5. Questions

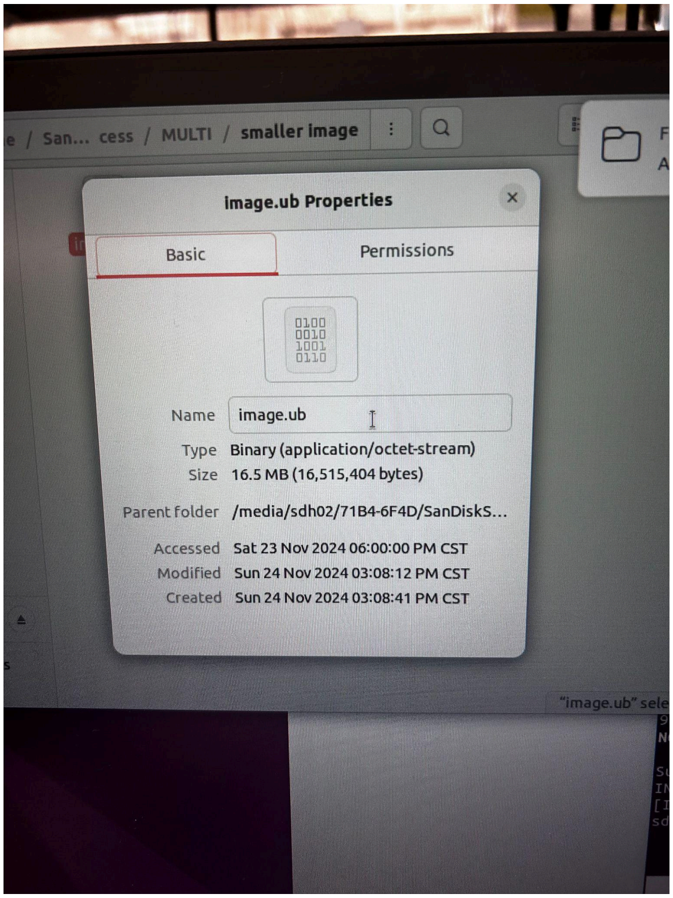
- What are the advantages and disadvantages of loadable kernel modules and built-in modules?
  The advantages of loadable kernel modules is that they can be added into the kernel at runtime without having to load a new kernel, making them convenient for adding new features. They can also be removed easier without having to restart the program. The disadvantage to them is that each one has to be loaded to the system, which can cause overhead when loading. The advantage to built in modules is that they are compiled directly into the kernel, and the user does not need to manage their load in after the kernel is built and implemented. The disadvantages is that they are inflexible and cannot be removed at runtime, they increase the kernel size, and if any changes need to be made the whole kernel needs to be rebooted and customized.

6. Appendix

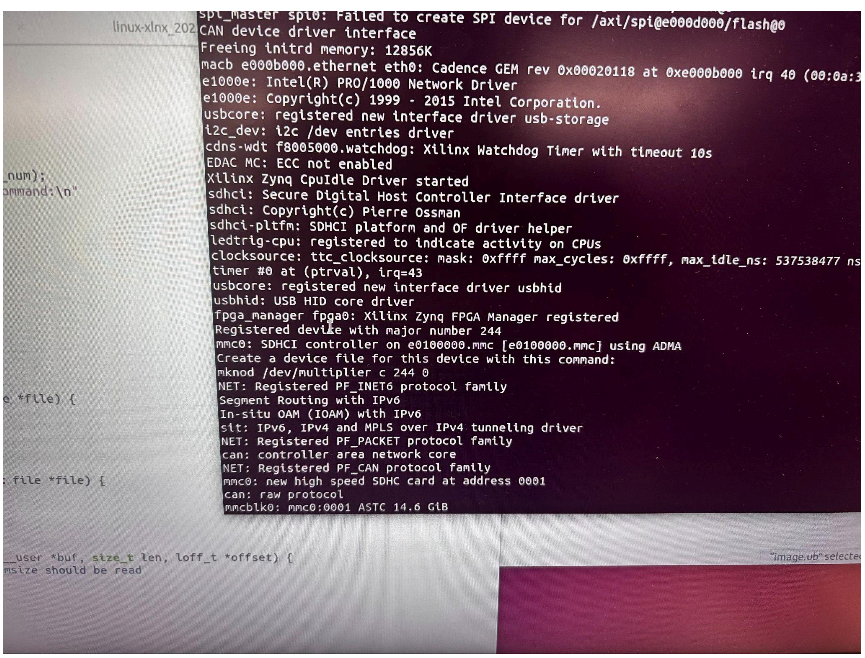
- image.ub



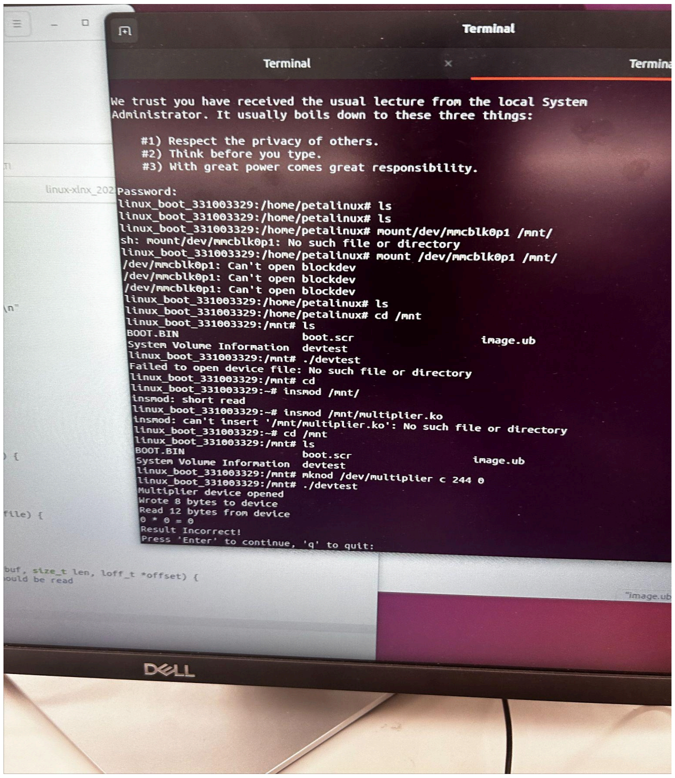
- image.ub after removing unnecessary features

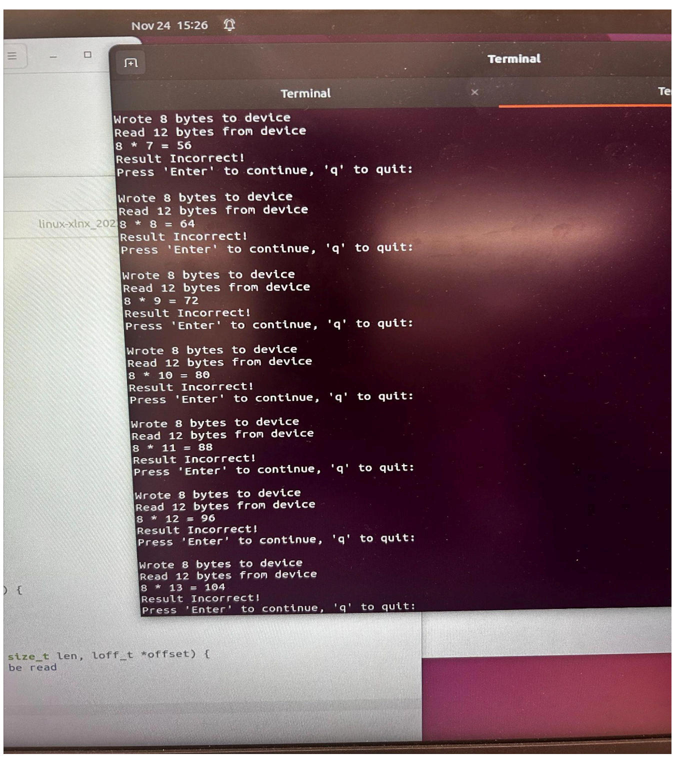


- Multiplier kernel being included upon build



- Multiplier driver functioning properly (print statement is inaccurate but program is properly functioning)





- Multiplier.c

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

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

```c
#include <linux/uaccess.h>
#include <linux/io.h>
#include "xparameters.h"

#define DEVICE_NAME "multiplier"
#define BASE_ADDR XPAR_MULTIPLY_0_S00_AXI_BASEADDR
#define MEM_SIZE 12

// module variables
static int major_num;
void __iomem *mapped_addr; // mapped address
//dev_t multiplier_dev;
static u8 kbuf[MEM_SIZE];  // kernel buffer of 12 bytes

// function prototypes
static int multiplier_open(struct inode *inode, struct file *file);
static int multiplier_release(struct inode *inode, struct file
*file);
static ssize_t multiplier_read(struct file *file, char __user *buf,
size_t len, loff_t *offset);
static ssize_t multiplier_write(struct file *file, const char __user
*buf, size_t len, loff_t *offset);

// File operations
const struct file_operations multiplier_fops = {
    .read = multiplier_read,
    .write = multiplier_write,
    .open = multiplier_open,
    .release = multiplier_release,
};
```

```c
//initialization routine
static int __init multiplier_init(void) {
    // Register the character device and assign a major number
    major_num = register_chrdev(0, DEVICE_NAME, &multiplier_fops);
    if (major_num < 0) {
        pr_err("Failed to register device\n");
        return major_num;
    }
    mapped_addr = ioremap(BASE_ADDR, MEM_SIZE);
    if (!mapped_addr) {
        unregister_chrdev(major_num, DEVICE_NAME);
        pr_err("Failed to map memory\n");
        return -ENOMEM;
    }


    pr_info("Registered device with major number %d\n", major_num);
    pr_info("Create a device file for this device with this
command:\n"
        "mknod /dev/%s c %d 0\n", DEVICE_NAME, major_num);
    return 0;
}

//Exit routine

static void __exit multiplier_exit(void) {
    iounmap(mapped_addr);
    unregister_chrdev(major_num, DEVICE_NAME);
    pr_info("Unregistered device and unmapped memory\n");
}
```

```c
// Open function
static int multiplier_open(struct inode *inode, struct file *file) {
    pr_info("Multiplier device opened\n");
    return 0;
}

// Release function
static int multiplier_release(struct inode *inode, struct file
*file) {
    pr_info("Multiplier device closed\n");
    return 0;
}

// Read function
static ssize_t multiplier_read(struct file *file, char __user *buf,
size_t len, loff_t *offset) {
    int bytes_to_copy = min(len, MEM_SIZE);  // only memsize should
be read
    int i;

    // data from device memory kbuf
    for (i = 0; i < bytes_to_copy; i++) {
        kbuf[i] = ioread8(mapped_addr + i);
    }

    // copy data from kbuf to user space
    if (copy_to_user(buf, kbuf, bytes_to_copy)) {
        pr_err("Failed to copy %d bytes to user space\n",
bytes_to_copy);
        return -EFAULT;
    }
```

```
    pr_info("Read %d bytes from device\n", bytes_to_copy);
    return bytes_to_copy;
}

// Write function
static ssize_t multiplier_write(struct file *file, const char __user
*buf, size_t len, loff_t *offset) {
    int bytes_to_copy = min(len, 8);  // ensure only 8 are written
      // write data from kbuf to device memory (mapped_addr)

    // copy data from user space to kbuf
    if (copy_from_user(kbuf, buf, bytes_to_copy)) {
        pr_err("Failed to copy %d bytes from user space\n",
bytes_to_copy);
        return -EFAULT;
    }
    int i;
    for (i = 0; i < bytes_to_copy; i++) {
        iowrite8(kbuf[i], mapped_addr + i);
    }

    pr_info("Wrote %d bytes to device\n", bytes_to_copy);
    return bytes_to_copy;
}

// Module macros
module_init(multiplier_init);
module_exit(multiplier_exit);

MODULE_LICENSE("GPL");
```

```
MODULE_AUTHOR("Samuel Huang");
MODULE_DESCRIPTION("Multiplier Device Driver");
```

- devtest.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    unsigned int result;
    int fd;  // File descriptor for the device
    int i, j;  // Loop variables
    char input = 0;

    // open device file for reading and writing
    fd = open("/dev/" DEVICE_NAME, O_RDWR);
    if (fd == -1) {
        perror("Failed to open device file");
        return -1;
    }

    while (input != 'q') {  // Continue unless user enters 'q'
        for (i = 0; i <= 16; i++) {
            for (j = 0; j <= 16; j++) {
                // Write values i and j to the registers using char
device
```

```
        int factor[2] = {i,j} // stores the 2 numbers which
take up 8 bytes
        if (write(fd, result, 8) == -1) {
            perror("Failed to write i to device");
            close(fd);
            return -1;
        }
        int product[3];
        // Read result from the device
        if (read(fd, product, 12) == -1) {
            // loads i,j and the product as the last four
bytes
            perror("Failed to read result from device");
            close(fd);
            return -1;
        }
        // Print result to screen
        printf("%u * %u = %u\n", factor[0], factor[1],
product[2]);

        // Validate result
        if (product[2] == (i * j)) {
            printf("Result Correct!\n");
        } else {
            printf("Result Incorrect!\n");
        }

        // Read from terminal to continue or quit
        printf("Press 'Enter' to continue, 'q' to quit: ");
        input = getchar();
        getchar();  // Consume the newline character left in
the input buffer
```

```
                if (input == 'q') break;

            }

            if (input == 'q') break;

        }

    }



    close(fd);

    return 0;

}
```