Sam Detor

Lab 1

Part 1

I followed this tutorial on how to write and insert a basic module into the kernel. I first installed the required packages needed for kernel development. I then learned that you need to write your kernel init and exit functions and tell the kernel which functions to call when loading or unloading the kernel using the module_init and module_exit functions. The init and exit functions simply used printk() to print "Hello World" and "Goodbye World" respectively. The most difficult part was getting the module to compile, as I didn't realize that it wouldn't compile unless module documentation is added. After I compiled and successfully inserted the module into the kernel, the kernel reported that it was "tainted". Upon further research I found that it is just a warning message saying non-vendor approved software had been inserted into the kernel module.

Source code used in part 1:

I used the following source code from Rober Oliver's "Writing a Simple Kernel Module" tutorial:

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


MODULE_LICENSE(“GPL”);
MODULE_AUTHOR(“Robert W. Oliver II”);
MODULE_DESCRIPTION(“A simple example Linux module.”);
MODULE_VERSION(“0.01”);


static int __init lkm_example_init(void) {
 printk(KERN_INFO “Hello, World!\n”);
 return 0;
}


static void __exit lkm_example_exit(void) {
 printk(KERN_INFO “Goodbye, World!\n”);
}


module_init(lkm_example_init);
module_exit(lkm_example_exit);
```

I changed the name of the init and exit functions from lkm_example_init and lkm_example_exit to helloWord and goodbyeWorld. I also changed the module description, author, and version.
I also used the makefile from the same tutorial:

```
obj-m += lkm_example.o

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

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

I only changed the name of the module from lkm_example to hello.

Part 2

At first, I thought we were supposed to create a sysfs file at the location of /sys/module/<module_name>/parameters/<parameter_name> using k_objects. Once I realized that we were just supposed to use module parameters, the problem became simpler. To implement the enable logging parameter, I followed another online tutorial to learn how to use the module_param macro. I created an integer parameter called enable_logging and made each printk() in the module code statement dependent on the value of enable_logging. After this, I ran into a permission issue. The sudo echo 0 > /sys/module/hello/parameters/enable_logging was not able to actually edit the parameter. I tried setting the write and read privileges for the module to global, but the module wouldn't compile. I then learned that you are not allowed to set the write privileges to a kernel module to global. I then found that the "sudo" in the previous command was only "working" on the echo and the redirection did not get sudo privileges. I then used sudo su to act as the root and set the module parameters to user read and write, and the tests worked as expected.

For the double_me parameter, I modified some code from an online tutorial on how to communicate with device drivers. The most difficult part of this assignment was finding the correct macro—module_param_cb—and online tutorial to use as moduleparam.h was difficult to understand. From the tutorial, I learned that module_param_cb macro takes an argument of a kernel_param_ops struct. This tells the macro what methods to call when the parameter is set or accessed by the user. So, I created a method that would be called when the parameter was set by the user, which checked if the parameter value given by the user was an integer —using the macro param_set_int—and then doubled the value set. This method was then put in the kernel_param_ops struct as the .set method and the standard param_get_int macro was used for the .get method in the struct. The rest of the set up for the double_me module parameter was similar to the enable_logging parameter.

Source code used in part 2:

For the enable logging parameter, I used code from Lirhan B.H.'s blog post on kernel module parameters. I used and modified the following two lines from the post:

```
static int irq=10;
module_param(irq,int,0660);
```

I changed the name from irq to enable_logging and changed the permissions from 0660 to S_IRUGO | S_IRUSR.

For the double_me parameter I used and modified some of the following code from an EmbeTronicX tutorial on passing arguments to device drivers.

```
int notify_param(const char *val, const struct kernel_param *kp)
{
        int res = param_set_int(val, kp); // Use helper for write variable
        if(res==0) {
                printk(KERN_INFO "Call back function called...\n");
                printk(KERN_INFO "New value of cb_valueETX = %d\n", cb_valueETX);
                return 0;
        }
        return -1;
}

const struct kernel_param_ops my_param_ops =
{
        .set = &notify_param, // Use our setter ...
        .get = &param_get_int, // .. and standard getter
};

module_param_cb(cb_valueETX, &my_param_ops, &cb_valueETX, S_IRUGO|S_IWUSR );
```

I changed the name of the parameter, kernel_param_ops struct, and the .set method to double_me, double_me_ops, and double_val respectively. In my version of the notify_param method, I doubled the double_me variable and printed a message to the kernel log instead. I also set my permissions to user read and write instead of global read and user write. Lastly, I returned the recommended EINVAL instead of -1 if the input given by the user was not an integer.

My part 2 code is also a modified version of my part 1 code, and sources for my part 1 code can be found in the part 1 section of this lab report.

Part 3

Looking back on the work needed for parts 1-5 of this lab, part 3 was definitely the most challenging. The first challenge was figuring out how to create the device file. Chapter 3 of the Linux Device Drivers textbook gave a good overview of how to allocate a character drive major/minor numbers and how to handle the character driver struct, as did many other sources. However, figuring out how to make a device file in the code without calling something like mknod in the shell was difficult. However, after some googling, I found an EmbeTronicX tutorial on how to create a device class and a device, so that the device file would show up when the module was inserted.

The next challenge was writing the methods for the file_operations struct: llseek, read, write, ioctl, open, close. The ioctl method and open method were probably the two most difficult methods to code. Open was deceptively simple. Since the open system call returns a file descriptor, I thought that the open method I was writing was supposed to return a file descriptor. Only after some more research did I realize that the open method is called to notify the driver that the device file has been opened and that the kernel handles returning file descriptors. After realizing that, I modeled my open, close, read, write, and llseek methods off of the example ones described in Chapter 3 of the Linux Device Driver. My biggest debugging challenge with these four methods was with the read and write method. I assumed that copy_from/to_user returned the amount of bytes copied, when in actuality it just returns 0 on success.

The ioctl method was definitely the most challenging one to write. I had two big issues while writing the ioctl methods. First, I didn't know I needed to define the ioctl commands using #define and the specified syntax. Second, I didn't realize that the unsigned long arg argument in the ioctl command was from user space, thus if I gave ioctl a user space pointer, I had to use copy_from_user to get its real value. Once I figured out these two bugs, fixing other minor problems became a lot easier. I found these well explained in another EmbeTronicX tutorial about ioctl, which was very helpful.

In implementing the sysfs files, the allocated parameter was quite similar to module parameters we had implemented in the past, so it was not difficult to use the

module_param macro to define an integer parameter with read only permissions. The regions parameter was a bit more difficult since I had to figure out how to return a string version of my linked list of allocations when the user requested it. After I realized that you can use sprintf in kernel code—which I read in this tutorial on sysfs—I was able to write a method to traverse my linked list of allocated regions and return the relevant information to the user in string form.

Lastly, there were a few issues I faced when compiling my kernel module and running my test program. Similar to the issue the student faced in the Ed discussion post, I also struggled with the fact that when compiling a module spread over multiple files, the module name can not be the same as the name of any of the source files. Additionally, I also didn't realize for a while that the test program needs to be run with sudo privileges, or else it doesn't have permission to open any of the driver files.

Data:

The values below were averages over 15 trials. Data is in seconds.

Time it takes to write to every byte in a 512KB region: 0.386421s

Time it takes to read every byte in a 512KB region: 0.370846s

Source code for part 3:

For the main module init and exit functions found in "MemManager.c", I used code from an EmbeTronicX tutorial on the creation of device files: The source code is as follows:

The init function:

```
81  static int __init etx_driver_init(void)
82  {
83          /*Allocating Major number*/
84          if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
85                  pr_err("Cannot allocate major number\n");
86                  return -1;
87          }
88          pr_info("Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
89
90          /*Creating cdev structure*/
91          cdev_init(&etx_cdev,&fops);
92
93          /*Adding character device to the system*/
94          if((cdev_add(&etx_cdev,dev,1)) < 0){
95              pr_err("Cannot add the device to the system\n");
96              goto r_class;
97          }
98
99          /*Creating struct class*/
100         if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
101             pr_err("Cannot create the struct class\n");
102             goto r_class;
103         }
104
105         /*Creating device*/
106         if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
107             pr_err("Cannot create the Device 1\n");
108             goto r_device;
109         }
110         pr_info("Device Driver Insert...Done!!!\n");
111       return 0;
112
113 r_device:
114         class_destroy(dev_class);
115 r_class:
116         unregister_chrdev_region(dev,1);
117         return -1;
118 }
```

The main change I made was removing the gotos and replacing them with the relevant lines of code that were going to be called anyway to increase readability. I also changed variable names to be more relevant in the context of the assignment. Additionally, I added these three lines of code from chapter 3 of the Linux Device Driver textbook:

```
cdev_init(&dev->cdev, &scull_fops);
dev->cdev.owner = THIS_MODULE;
dev->cdev.ops = &scull_fops;
```

I again changed variable names to be more relevant to the project.

The exit function:

```
/*
** Module exit function
*/
static void __exit etx_driver_exit(void)
{
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        cdev_del(&etx_cdev);
        unregister_chrdev_region(dev, 1);
        pr_info("Device Driver Remove...Done!!!\n");
}
```

I changed variable names to fit the context of the assignment and removed the print statement

For the open method, I modeled my open method off of the "scull" driver open method from Chapter 3 of the Linux Device Driver textbook (linked above):

```
int scull_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev; /* device information */

    dev = container_of(inode->i_cdev, struct scull_dev, cdev);
    filp->private_data = dev; /* for other methods */

    /* now trim to 0 the length of the device if open was write
    if ( (filp->f_flags & O_ACCMODE) =  = O_WRONLY) {
        scull_trim(dev); /* ignore errors */
    }
    return 0;          /* success */
}
```

I changed variable names and removed the trim to length zero part of this method as it was unnecessary for this assignment.

I also modeled my llseek method after the scull_llseek implementation in the linux device driver textbook (chapter 6).

```
loff_t scull_llseek(struct file *filp, loff_t off, int whence)
{
    struct scull_dev *dev = filp->private_data;
    loff_t newpos;

    switch(whence) {
      case 0: /* SEEK_SET */
        newpos = off;
        break;

      case 1: /* SEEK_CUR */
        newpos = filp->f_pos + off;
        break;

      case 2: /* SEEK_END */
        newpos = dev->size + off;
        break;

      default: /* can't happen */
        return -EINVAL;
    }
    if (newpos < 0) return -EINVAL;
    filp->f_pos = newpos;
    return newpos;
}
```

I also changed variable names to fix the context of this assignment. I also added an if statement to check if the new offset was less than the size of the allocated region.

For the module parameters, I used the source code from Part 2 of this lab. I changed variable names to be more relevant to this assignment. Additionally, for the "regions"

parameter, I wrote a custom get method instead of a custom set method. See the "source code for part 2" for the citation of that code.

Part 4

The first step to creating the smart module was to use the EXPORT_SYMBOLS() macro to make functions from my part 3 module accessible to my new module. I had a bit of difficulty here because I didn't realize at first that I had to declare the functions I was using from the simple module in the header file used in the smart module before I could correctly use them. Additionally, I had some significant difficulty compiling the smart module code, before I realized that the easiest way to do that was to compile both modules together. Apparently, using a makefile to compile files from two different folders is complex and not the best idea. For that reason, I put a copy of my part 3 code in my part 4 folder for easy compilation.

My implementation of the init and exit functions from the smart module was almost the same as my init and exit function for part 3, so that wasn't too difficult. I did have to change a few names and get rid of references to the module parameter and kobject, as they were handled in the part 3 module. Finally, the new read and write methods (smart_read and smart_write), had a few small changes to the original read and write methods from part 3. Instead of checking if each request was for one byte, the smart methods now check if the amount of bytes requested plus the current offset is outside of the current region. Additionally, the copy_to_user and copy_from_user methods now depend on count for the number of bytes to transfer.

Data from Part 4:

The data for this section was collected in bytes and milliseconds. All values shown below were averaged over 15 trials

Table 1. Average values of time to read/write to every byte in the 512KB region.

Amount of Data transferred per call	Time to Read (ms)	Time to Write (ms)
1B	390.549713	391.501495
64B	7.2412	6.776801
1KB	0.593933	0.607667
64KB	0.0326	0.034667
512KB	0.034667	0.034667

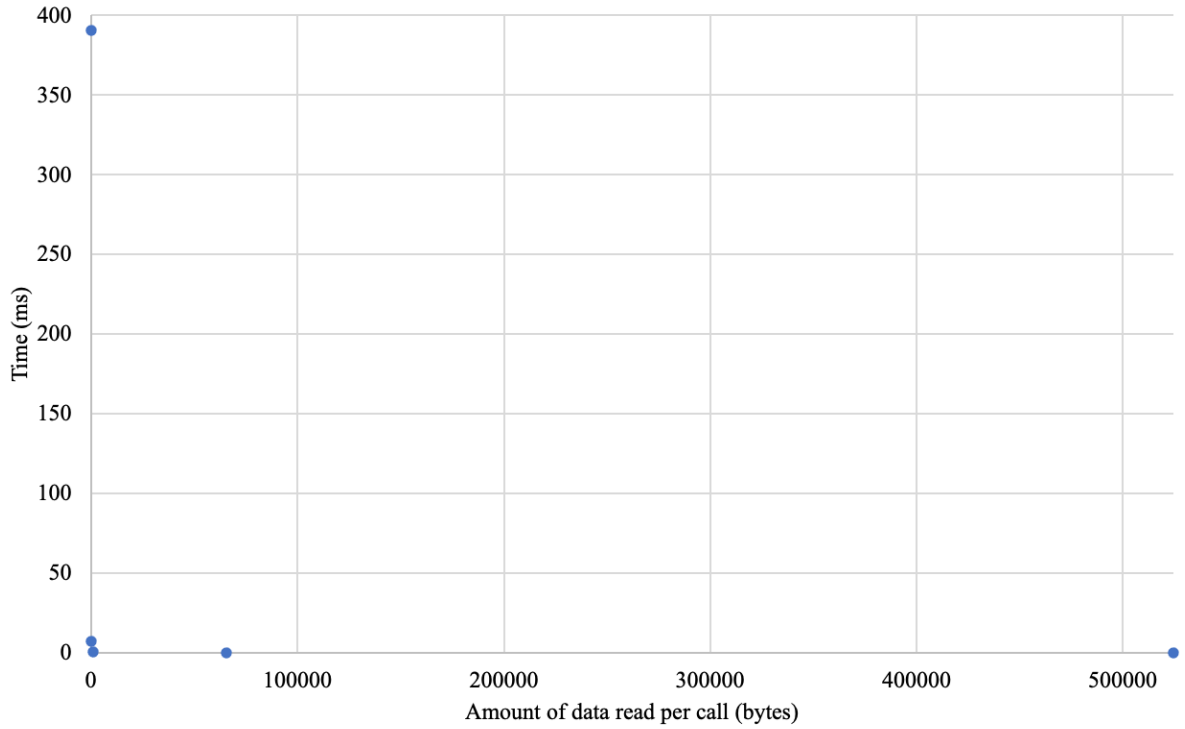

Figure 1. Bytes read per read function call vs. time take to read 512KB

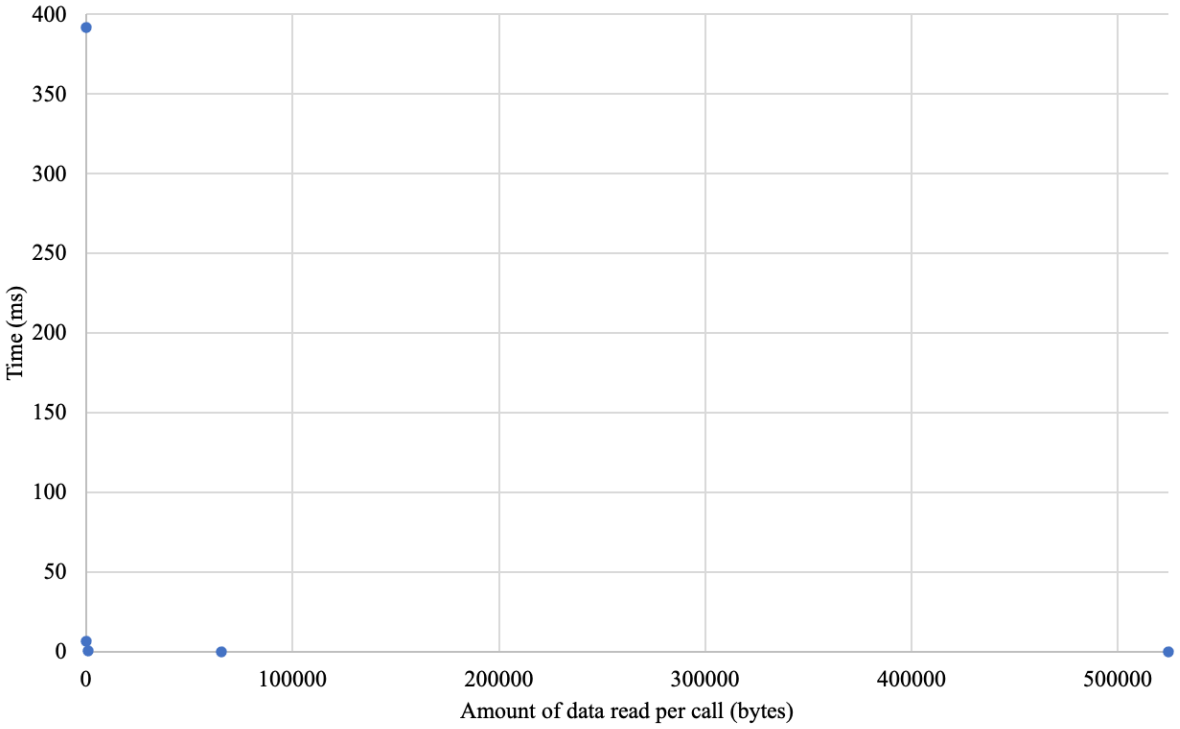

Figure 2. Bytes written per write function call vs. time take to write to 512KB

Analysis:

There is definitely a difference in throughput between reading/writing smaller amounts of bytes at a time vs larger amounts. In fact, there is such a large difference the data points can not be very nicely displayed in graph form. For example, when writing one byte at a time, you can write about $1.3 \cdot 10^6$ bytes/sec. When writing all 512KB in one call, you can write around $1.5 \cdot 10^{10}$, this is 10000x faster than writing one byte per call.

One of the reasons for this increase in speed is the amount of memory references involved in reading and writing the 512KB region. The methods copy_to_user and copy_from_user are responsible for reading and writing to memory in the user space. Accessing memory is a slow process, as the user virtual address has to be translated into a physical address. In the 1 bytes per write call version, there needs to be roughly $5 \cdot 10^5$ of

these copy_from_user calls. In the 512KB per write call version, there is only one call to copy_from_user necessary.

Additionally, calling a system call, which generates a kernel exception, is also a slow process, as the kernel has to find the exception handler, then find the right system call to execute, and then execute the system call. The fewer system calls possible, the faster the program will run. The methods that require writing fewer bytes per system call will then be slower because more system calls are needed to write the required amount of bytes.

Source code used in part 4

I used an EmbeTronicX tutorial to understand how to use EXPORT_SYMBOL and compile code which relies on it. I did not directly use any of the source code from the tutorial, but I wanted to acknowledge the tutorial's help.

All other source code used in part 4 was cited in the source code section of part 3.

Part 5

To create the multithreaded program, I used the pthreads library. I created an array of threads of size W, and had them run a for loop that consisted of reading and writing to the "counter" in the file managed by the Smart module. One of the difficulties I experienced during this part was figuring out how to get the parent to wait for all the threads before closing the file, reading, and printing the counter. After some research, I then found a stack overflow post showing how you could use the method pthreads_join in a for loop to have the parent wait for all the children. Additionally, I was originally using fork() to create my threads, but then found that it was a lot easier with pthreads to implement locks, which seemed easier to implement than something like atomic types when dealing with file read and writes. So I switched to implementing my solution with pthreads.

When originally running the unsafe program, I observed errors in the counter at very low values of W for large Ns. Specifically, for N > 100000, if there were more than 1 thread, there were errors. When running the program with 10 and 15 threads, there was overlap between the values. Some of the trials with 10 threads created a larger counter than some trials of the 15 thread program and vice versa. When running the program with 20 threads, the values of the counter usually ended up being smaller than the ones run with 15 threads. As the amount of threads increases, the difference in the counter value between trials with more threads and less threads—ie the difference in counter between trials with 30 threads and 25 threads—becomes proportionally smaller.

The reason the incorrect results occur is because in the time one thread reads increments and then writes the counter back to memory, another thread could have read the counter for its own incrementation process. Once the second thread increments the value it has read from file and writes it back, it will have overwritten the increment that that thread one just performed, as it was using an old value of the counter. In the test program, this is especially exaggerated because of the time needed to move the file offset back to zero between a read and write command. These errors cause the value of the counter to be smaller than expected. Another issue that can occur is when one thread is reading while another is writing. This would cause the data read to be incorrect, and potentially the some or all of the data to be written to the wrong location because read and write depend on the same offset.

To fix this error, I implemented mutex locks using the pthreads library. I locked the code segment in my for loop—lines 31-35—which allowed only one thread to read, increment, and write the counter to the file at any one time. This prevented the earlier errors as the value of the counter was not changing in between when a thread read it and wrote to it. Programmatically, using the pthreads library, it was pretty simple to implement. Additionally, it's implementation didn't depend on any properties of what was happening in the "locked" code, which makes it very versatile. One issue with using locks, per this source, is that it makes threads wait for locks, which wastes CPU time that could be spent doing something else. Additionally, if something happens to the thread

while it is in a locked section of the code and it doesn't finish the segment, the segment will never be unlocked. Meaning, the program will not finish.

Assuming that I could eliminate the data race from the user space, I could implement a semaphore in my smart kernel module. I found this approach in Chapter 5 of the Linux Device Drivers textbook. This would function similarly to the userspace locking approach. In my "myMem_struct", I would put a semaphore struct object (which is defined by the kernel) and initialize it before the character device was "added" and available to take requests. Whenever a thread needed to access the memory that my smart module was used for, it would first have to check, with kernel defined methods, that it had permission to access the memory. In the context of the smart module, I would only want one thread at a time being allowed to access the memory, so it would function like a mutex lock. Once the thread was done with it's memory access, then it would unlock the semaphore, and another thread could acquire a lock on accessing the memory. This would make sure that no two threads are accessing the smart module's memory or parameters (like the offset) at the same time.

As is, the smart module allows a lot of unsafe behavior beyond what was explored in the experiment to happen. For example, if two threads were trying to allocate a region at the same time, they could potentially end up with the same region ID. This would cause data loss issues as well as potential over allocation of kernel memory, if both processes allocate a "region" struct and data for the "same" region. This could be solved with the global counter used for region IDs to be moved into the "myMem_struct", so it could be protected by the semaphore like the other memory parameters.

To address some of the examples in the Lab 1 handout, if you deallocate a region before a worker thread is done, many things could happen. If the worker was mid read or write, it would be writing to freed memory space, which could have already been allocated to another program. This could have any number of consequences. This could also be solved by only letting one thread access any memory parameters or resources in the "myMem_struct", as described above.

Currently, if a worker thread tries to access a byte of memory outside of the current region's bounds, the module will return an error, granted that the region_size parameter is accurate, and has not been messed up by data races.

Many of the unsafe behaviors allowed by the smart module are related to not controlling what threads can access global variables or memory regions managed by the module. These problems can be avoided by making the access point to these types of resources a struct, "like myMem_struct", and protecting that struct with a semaphore. There are even optimizations like a read/write semaphore that the kernel provides to try to reduce the amount of time wasted through waiting for access.

Source code used in part 5:

I used the following code from this stack overflow post to create the original multi-threaded program:

```
int main()
{
    pthread_t tid[2];
    for (int i = 0; i < 2; i++) {
        pthread_create(&tid[i], NULL, routine, NULL);
    }
    for (int i = 0; i < 2; i++)
       pthread_join(tid[i], NULL);
    return 0;
}
```

I changed the size of the pthreads array, and my "routine", which is called "forThreads" is a for loop reading, incrementing, and writing to the counter "N" times. I also changed the names of the variables to reflect the context of this assignment.

When implementing the mutex locks, I used the Mutex lock for Linux Thread Synchronization article in Geeks for Geeks. I applied their placement and method of locking their certain section of code to the code I had written with the help of the above stack overflow post. Specifically, I used these sections of the code I found in the article:

```c
void* trythis(void* arg)
{
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++)
        ;

    printf("\n Job %d has finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}
```

```c
int main(void)
{
    int i = 0;
    int error;

    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return 1;
    }

    while (i < 2) {
        error = pthread_create(&(tid[i]),
                               NULL,
                               &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created :[%s]",
                   strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
```

I did not use any of the lines in between the calls pertaining to the mutex lock, I substituted code from the unsafe multithreaded program earlier in part 5.

Part 6

Summary:

In order to implement my system call, I changed the overall kernel makefile, the syscall_64.tbl (syscall table) and created a folder in the linux root directory which housed my system call source code and another makefile. The folder I created had the code for my system call and the makefile that specified how to compile that code. In order to tell the kernel that it needed to compile the code that was in the capitalize_syscall folder, I needed to edit the overall kernel makefile. I added "capitalize_syscall/" to the "core-y :=" line to tell the makefile that it needed to visit the capitalize_syscall sub directory and run the makefile inside there according to Anubhav Shrimal's Medium Tutorial. I also added an entry to the syscall table stored at syscall_64.tbl. This entry first specified the system call number, which ended up being 548 in the 5.16.11 kernel. Then it specified what type of system call it was. Based on the St Olaf lab handout, I choose to designate the system call as "common". The name of the system call (capitalize_syscall) and then "sys_*systemcallname*" were the last two items in the syscall table entry.

To effectively deal with the string the user passed to my syscall, I added a length of the string argument to allow the method to allocate and copy the whole string effectively. Without this length, you wouldn't have a good way of knowing how long the string is since you can't access the user pointer directly in kernel space to look for null characters. I also check that the "length of the string" parameter given is greater than 0, to prevent issues like passing a negative number to kmalloc and doing unnecessary processing. After kmalloc is called, I also check to make sure that it didn't return a null pointer, to make sure that the rest of the code doesn't try to dereference a null pointer. I mostly rely on copy_to/from_user to verify that the char* input is valid, and return -1 and free the kernel string if those methods return an error. This would prevent errors when passed invalid user pointers. If one of these invalid user pointers was blindly used (i.e. not through copy_from/to_user), it could cause data to be read or written from locations that it shouldn't be. Lastly, after any error based early exit from the function, any kernel memory allocated is free'd.

The code for the actual system call was fairly simple to write. I used the SYSCALL_DEFINE2 macro to define my system call and its two arguments (the string and the length) as explained in the lab handouts from St Olaf and Albany State. This macro creates the "entry point" for the syscall and provides information about the new system call to other parts of the kernel that might need it. I used the recommended method for parsing lower case letters and changing them to uppercase that was posted on Ed Discussion. My approach basically consisted of copying the strings to a kernel allocated block of memory, parsing the string, and then copying back into the user buffer.

The hardest part of this lab was definitely getting the kernel to compile and installing the kernel. The compilation process was pretty slow, which made it hard to make progress efficiently. In order to combat this issue, I used the .config file from my running kernel so that less unnecessary things would be compiled. The first major issue I ran into was a certification issue connected to the .config file. Since I copied the .config file from my running kernel, there were some odd certification parameters in the .config file that I needed to get rid of. The next issue was the format of adding entries to the syscall table. I did not realize that if you add "__x64_sys_my_call" to the syscall table, it will look for a function called __x64__x64_sys_my_call. I was also originally not using the SYSCALL_DEFINE macro to create my system calls, which also caused some issues. I think the largest issue I ran into was after the compilation of my kernel, when I tried to run "make modules_install", my virtual machine ran out of memory, crashed, and was unable to be booted again. I then had to create a new virtual machine image, reinstall the operating system, and start again.

In response to one of the questions posed in the lab report, one of the reasons why the kernel no longer allows for loadable kernel modules to create new syscalls could be connected to security. There are now many 3rd party loadable kernel modules for all the different types of drivers we have in our computer. Letting these drivers create their own system calls could be a security risk. Additionally, the kernel may want to limit the amount of syscalls it has, per this source, and would prefer developers use other methods—for example, ioctl() via device drivers—to communicate with the user space.

Once a system call is a part of the kernel, it has to be supported indefinitely, so kernel developers may want to avoid unnecessarily adding complexity to the kernel code if the same result can be achieved with a loadable module.

When you call syscall() with a bogus system call number that doesn't exist, the function returns -1 and sets errno to 38, which corresponds to "function not implemented."

There are a couple different ways a user program can help prevent this from happening. All official system call numbers are supposed to be listed in unistad.h, so one way to create a user space check is to find the largest system call number in unistad.h and make sure all calls to syscall() are using a system call number that this greater or equal to 0 and less than equal to the largest system call number in unistad_64.h (for 64 bit systems). One difficult thing about implementing a user space solution is that the number of system calls a linux kernel has is based on distribution and compile options, so it would end up being dependent on what kernel the program was running on. You could also run a bash script of some kind like the ones described in this source to determine how many system calls there are, and then use that number minus 1 as the maximum number allowed to be passed as a system call number to syscall(). However, these methods do not seem to perfectly determine the amount of system calls that are present. If the bash script option was chosen, it would probably be more like a rough userspace check to see if the syscall number being passed was not outrageously wrong.

Source Code:

To figure out how to correctly create a folder with a C file and it's corresponding makefile and get the kernel to compile it, I used Anubhav Shrimal's tutorial on how to add a hello world system call. I used the following source code for my makefile:

```
obj-y := hello.o
```

I changed the "hello.o" to "capitalize_syscall.o" to make it relevant to the project.
I also used source code from that tutorial for my syscall table entry:

```
548     64      hello           sys_hello
```

I changed 64 to common per the St Olaf lab handout and changed variable names to better align with the assigment.

I also used another one of Shrimal's tutorials on compiling a custom kernel to figure out the steps to compiling/booting a new version of the kernel

I learned how to use the SYSCALL_DEFINEn macro from the St Olaf lab handout and the Albany State handout, but didn't use any source code directly from them.

I used the recommendation for how to detect lowercase letters and change them to uppercase found on Ed Discussion Post #45:

```
if (c >= 'a' && c <= 'z') {
    ...
}
```

The rest of the capital syscall code and the code for the userspace test were based off of my experience in kernel module coding from part 3, see that part of this report for more relevant sources as well.

Part 7

I used inline extended asm to call capitalize_syscall manually. In order to do this, I had to move the system call number, the string, and length of the string in registers rax, rdi, and rsi respectively. Then I had to call the "syscall" instruction and move the return value found in rax into a variable I could return from my manual syscall function. In the asm syntax, there is also a section to list registers that have been overwritten during the course of the assembly to be executed. So I made sure to list rax, rdi, and rsi as "clobbered" registers in order to make sure that nothing important got overwritten. The biggest challenge during this assignment was getting used to the extended asm syntax and

making sure that I was calling x64 instructions. For example, I was originally trying to put the syscall number in eax and calling int 80h instead of using rax and syscall.

I also know that there were methods of using advanced inline assembly syntax where you would only have to write 1 line of actual assembly code "syscall" to call the system call manually. However, I felt that even though the way I did it may be less efficient, it was more readable and clear.

The max number of arguments a system call can take is 6. The limiting factor is register space, as there are 6 registers system call arguments can be put in: RBX, RCX, RDX, RSI, RDI, RBP. One way to get around this limit is passing a pointer to a struct, which could contain more than 6 items, as one of the designated 6 arguments.

Source Code:

I used the following source code from this "Just a Memo" blog post:

```
#include <string.h>

int main() {
        char* str = "Hello World\n";
        long len = strlen(str);
        int ret = 0;

        __asm__("movq $1, %%rax \n\t"
                "movq $1, %%rdi \n\t"
                "movq %1, %%rsi \n\t"
                "movl %2, %%edx \n\t"
                "syscall"
                : "=g"(ret)
                : "g"(str), "g" (len));

        return 0;
}
```

I only used the code contained in the "__asm__" function call. I changed the system call number to an input variable, and only passed arguments to the rdi and rsi registers. I also changed the names of the variables to make them more relevant to the assignment and added a line to move the value in rax (the return value) into the my "returnVal" variable.