**Project 2 Report**
**Group Members**

George Barroso
Stephan Belizaire
Ricardo Casilimas

**PART 1 Professor emulator**

The purpose of this program is to run an emulation of a professors lab hours. The program runs as follows.

We a start the program by using the command
“./profemulator <Number Of Students> <Office Student limit>".

The emulator then does some input validation in order to ensure that no unwanted inputs are entered by the user. Along with this is a function call isNumber that checks if the input is numerical

```
int totalStudents, i;

if(argc <= 2)
{
    printf("Invalid Number of Arguments (./profemulator <Num Students> <Office Limit>)\n");
    exit(0);
}
else
{
    if(isNumber(argv[1]) && isNumber(argv[2]))
    {
        totalStudents = atoi(argv[1]);
        officeLimits = atoi(argv[2]);
        if(totalStudents <= 0 || officeLimits <= 0)
        {
            printf("Invalid Number of Students or Office Limit(Number must be > 0)\n");
            exit(0);
        }

    }
    else
    {
        printf("Invalid Entry Found (./profemulator <Num Students> <Office Limit>)\n");
        exit(0);
    }
}
```

Then we start the professor helper function and run the student help function according to how many students the user entered as an argument. We also start the CloseOfficeCheck thread which checks if the office is empty for a period of time then closes the office if it is.

```
Professor(); //starts the professor helper function

pthread_create(&closeOffice, NULL, CloseOfficeCheck, NULL); //starts the CloseOfficeCheck Thread

for(i = 0 ; i < totalStudents; i++)
{ //Calls the Student helper function totalStudents times

    Student(i);

}
pthread_exit(NULL);
```

The professor thread then starts and waits to receive a question. At the same time all the student threads start and attempt to enter the office. If the office contains more students then the allowed max it is locked using a mutex and the student that is next in line then sits in a loop waiting for a student to leave so they may enter the office

```c
void *StartStudent(void * student) {

    struct student * std = student;


    pthread_mutex_lock(&office_lock); // Mutex locks the office
    while(inOffice >= officeLimits); //One student thread waits for another student to leave beffore enteringe

    EnterOffice(std);
    inOffice++;
    pthread_mutex_unlock(&office_lock); //Mutex unlocks letting another student enter position 1 of the queue

    printf("Students in office currently: %d\n", inOffice);
```

A student who is in the office then enters a loop where they can ask questions as long as the professor has told them they can ask it. They then ask their question by sending by receiving the readyForQuestion signal and then they ask. Once the question is asked they signal the professor that the question is done, and wait to receive their response signal from the professor. Once this process is done they unlock the question mutex and someone else in the office can ask another question.

```c
while(std->totalQuestions > std->answered) {

    sem_wait(&readyForQuestion); //Waits to receive the signal to ask a question
    pthread_mutex_lock(&question_lock); //locks other students so they can't ask a question at the same time

    currentStudent = std->studentNum;
    QuestionStart(std);

    sem_post(&questionWait); //signals the professer that the question was asked
    sem_wait(&answerWait); //waits to receive the answer


    QuestionDone(std);
    pthread_mutex_unlock(&question_lock);//unlocks the question lock when done
    std->answered++;
```

```c
void * StartProfessor() {
    while(1) {

        sem_post(&readyForQuestion); //signals students that the professor is ready for a question
        sem_wait(&questionWait);

        AnswerStart(); //Answers the question
        AnswerDone();
        sem_post(&answerWait); //informs the student that the question was answerd

    }
}
```

Once a student has finished with all of their questions and has received their final response they leave the office and the student that was next comes into the office.

```c
if(std->totalQuestions == std->answered) { //if the student is done asking questions, leaves the office

    LeaveOffice(std);
    inOffice--;
    printf("Students in office currently: %d\n", inOffice);

}
```

This process continues untill all students have asked all of their questions and no more students are left in the office. While all of this happens, a thread is running checking that the office is continuously in use. When that thread notices that no students have entered the office for more than 4 seconds it then stops the program, therefore closing the office.

**PART 2 LKM**

In this second part we run an LKM that will print out information about the proccess and including the number of threads it has and its parent PID.

First we run our program from part 1 and we insert the LKM using the command.
“sudo insmod proflkm.ko”
This program starts in the task_lister_init function that then runs the DFS function. This is a depth first search through all processes running on the computer. We stop searching when we find our process running. We search for it by name ("profemulator").

```c
void DFS(struct task_struct *task)
{
    struct task_struct *child;
    struct list_head *list;

    if(strcmp(task->comm, "profemulator") == 0) //when we find a process called profemulator we save a pointer to it in found_task
    {
      found_task = task;
      return;
    }

    list_for_each(list, &task->children) { //recursively loop through all tasks depth first
        child = list_entry(list, struct task_struct, sibling);
        DFS(child);
    }
}
```

We then return back to the task_lister_init function in order to print the info from that process. We make sure that before running the next code we found the process. If we did we calculate the number of threads that are running with the while_each_thread loop and then we print out the information to the kernel with printk.

```c
int task_lister_init(void) //init function
{
    printk(KERN_INFO "Loading profemulator info module...\n");
    DFS(&init_task);

    int threadCount = 0;

    if(strcmp(found_task->comm, "profemulator") == 0) //Checking to make sure the proccess was found
    {
        printk("KERN:\n process name: %s\n pid: [%d]\n parent pid: [%d]\n", found_task->comm, found_task->pid, found_task->parent->pid); //info print

        struct task_struct *me = found_task;
        struct task_stuct *t = me;
        do {
            threadCount += 1;
        }while_each_thread(me, t); //counting the number of threads

        printk(" thread count: %d\n", threadCount);
    }

    return 0;
```

We then run the dsmeg command and it results in an output that looks like this.

```
[ 2410.346673] KERN:
                process name: profemulator
                pid: [3742]
                parent pid: [1187]
[ 2410.346689]  thread count: 184
```