ELEC 377 Lab 2: Description of the Program

Name & Student ID: Shiyan Boxer (20106887) and Arsh Kochhar (20104779)
Date: Oct 13th, 2020

Overview of the Program

This program exposes some kernel internals. It reports on some memory information about the processors and delivers the results through a /proc file. The program has two linux modules that generate a /proc file that contains one line for each process process, with each line containing the PID, the user id (called uid) the total amount of virtual memory used by the process and the amount of virtual memory that is in RAM.

Solution of our Program

Step 1: “Hello World” Module `int init_module(){`

Step 1 implements the init_module function. It declares a pointer to a proc_dir_entry struct shown in the first line of the function.

```
struct proc_dir_entry * proc_entry;
```

Then it creates a new proc entry (“lab2” file) using the create_proc_entry function. The 0444 makes it readable to everyone and NULL gives the directory.

```
proc_entry = create_proc_entry("lab2", 0444, NULL);
```

A if-else statement was added to check if creating the new proc entry was successful. If it was successful, it will return a pointer to a proc_dir_entry data structure that the kernel allocated. Then set the read_proc entry of the proc_entry data structure to the read proc function and return the value 0 from init_module to indicate that the initialization of the module was successful. If there is a problem it will return -1 from init_module to tell the kernel that it could not successfully initialize the module. The kernel will delete the module from kernel memory.

```
   if (proc_entry==NULL){
       return -1;
   }
   else {
       proc_entry->read_proc=&my_read_proc;
       return 0;
   }
```

Step 2: cleanup_module Function `int cleanup_module(){`

Step 2 implements the cleanup_module function. It is called by the kernel when the module is removed.

```
remove_proc_entry("lab2", NULL);
```

This removes the /proc file by taking in the “lab2” parameter, the file created and the NULL parameter.

Step 3: Read Procedure `int my_read_proc(char * page, char **start, off_t fpos, int blen, int * eof, void * data){`

The read procedure function uses an if-else structure. The first if statement checks if this is the first read of the data.

```
if (fpos == 0){
```

The numChars variable was created so that the second write starts from where the first write left off. Since sprintf returns the number of characters written by this call, you have to add the return value to numChars.

```
int numChars=0;
```

The function writes the headers (PID, UID, VSZ and RSS) to the buffer using a sprintf call.

```
numChars+= sprintf(page + numChars, "PID\tUID\tVSZ\tRSS\n");
```

The first section finds the first valid task, assigns the address of the variable init_task to the variable theTask, and then uses a while loop to find the first task who's PID (theTask->pid) is not equal to zero. Then copy the pointer in theTask to the variable firstTask to remember where you started in the list.

```
 while(firstTask->pid == 0){
            firstTask=firstTask->next_task;
```

Add the process id and user id to the buffer page.

```
numChars+= sprintf
(page+numChars,"%d\t%d\t",theTask->pid,theTask->uid);
```

Calculate page size by adding a variable that contains the value (PAGE_ZIZE >>10), the size of the page in K. It also checks if mm is NULL, if so, then add two 0’s to the buffer. Otherwise add the total_vm and the rss fields of the mm field multiplied by tmp variable with the page size to the buffer.

```
if (theTask->mm == NULL){
            numChars+= sprintf (page+numChars,"0\t0\n");
        }
        else{
            int tmp= PAGE_SIZE >> 10;
            theTask->mm->rss*=tmp;
            theTask->mm->total_vm*=tmp;
            numChars+= sprintf
(page+numChars,"%d\t%d\n",theTask->mm->total_vm,theTask->mm->rss);
```

```
        }
```

The do-while loop was used to move the variable theTask to point to the next valid task.

```
do{
            theTask=theTask->next_task;
        }while(theTask->pid == 0);
```

The else statement checks if the end of the list of processes has been reached, and if so, signals the end of the file. The code within this else statement is the same as the first if statement described above where it prints the data to the buffer, and the code to advance to the next task.

Notes about the C language: The **sprintf function** is used throughout the program. It is like the fprintf function, but instead of printing to a file, it prints into a buffer. The function returns the number of characters that was written.