Group-35
Ruijie Ji 20108157
William Du 20100436

Project Description

The purpose of lab2 is to write a Linux module to generate a /proc file to achieve communication between the user and the kernel space. Each line of the file contains the information of each process stored in the task_struct data structure, including process id, the user id, the total amount of virtual memory used by the process, and the amount of virtual memory that is in RAM. To retrieve the required information of the processes, after setting up the module, the user-level process should first make read calls to read the file created in the proc entry, copy the data to the kernel buffer, and then the proc system returns the data to the user process. The code is divided into 3 functions, including init_module for initializing module, my_read_proc for the whole read procedure and cleanup_module for removing the /proc file.

The first function executed after the module is loaded is init_module, which returns 0 if the module is initialized successfully, otherwise, it returns a non-zero value and the module will be deleted from kernel memory. The first step in this function is to declare a pointer to a proc_dir_entry struct. Next, it creates a new proc entry using the create_proc_entry function. The create_proc_entry function returns a pointer to a proc_dir_entry data structure allocated by the kernel and returns NULL if fails. The create_proc_entry function takes three parameters, the first is the name of the proc file to be generated, the second is the permission of the file, we used 0444 which means that the file is readable by everyone, but no one is allowed to write or execute it. The third one gives the directory, and we use NULL for the /proc directory. After the initialization of the module, the read_proc entry of the proc_entry is set to read proc function.

The read procedure is implemented using the my_read_proc function that takes 6 arguments and returns the number of character that was read per time. The page parameter is a pointer to a kernel buffer where the returned information about the user process is stored. The start parameter is used to set its pointer to the page buffer and the blen parameter is the length of the buffer. The parameter fpos shows the current position in the file which starts from 0 for the first read, and it will be equal to the size of data that has been returned after the first read. The parameter eof is used to indicate the end of the file if it is equal to 0. The data parameter is used when multiple processes read the file at once.

If fpos is equal to zero, it indicates that this is the first read of data, and the function will return the data to the kernel and shows that more data is available. In the if statement, the sprintf function prints four headers of the information of each process. The sprintf function is used to print data into the buffer and return the number of characters written per call(numChars). The sprintf function takes several parameters including a pointer to the buffer

where the data is stored and the data to be printed. The sprintf function also support % modifiers and we used %8d in this lab to print the data in a 8 character wide field for good formatting. After the first print, we have to add numChars to the first parameter of sprintf so that the next write will start from the previous write left off. Next, set theTask pointer to the variable init_task which contains the PCB with 0 PID value. The first valid task is found by searching for the first task with a non-zero PID value. To record the starting position in the process list, we copy the pointer in theTask to the variable firstTask. After finding the first valid task, sprintf is used to write the process id and user id to the page buffer. Since the virtual memory information about the process is stored in the field mm, if mm is NULL, we add two 0s to the buffer. Otherwise, we add the total_vm and rss field of the mm field multiplied by the page size to the buffer. The final step is to find the next valid task. After completing the if statement, the my_read_proc function set the memory pointed to by eof to 1, set the pointer given by start to the buffer, and return the number of characters that was written.

The else part of the read procedure is to check whether the end of the process list has been reached, if so, set the memory pointed to by eof to 0, set the pointer given by start to the buffer, and return 0 to show the end of the file. If not, it continues writing the information of one task and then finds the next valid task. After finishing the else part, set the memory pointed to by eof to 1, set the pointer given by start to the buffer, and return the number of characters that was written. Since the process control blocks are stored in a circular linked list, the end of the list is reached if the firstTask is the same as theTask, and we can also easily traverse the list of the processes by move the pointer to the next element.

When the module is removed, the /proc file can also be deleted by calling the cleanup_module function. Inside the cleanup_module function, the remove_proc_entry function is invoked which takes two arguments, the first one is the name of the file to be removed and the second is for the directory, and we use NULL for the /proc directory.