COMP 530H
Fall 2014

Honors Assignment 2
Module to implement 'getpinfo' (extended)
Date assigned: September 12, 2014
Date due: September 26, 2014

Modify and extend your getpinfo service module and calling program from assignment 1 to provide additional information from the task_struct itself and from other resource descriptor structures specified with pointers in the task_struct.

In the example program, an inline function, task_pid_nr() for accessing process pids was compiled into the program from the file include/linux/sched.h. The kernel also has EXPORTED symbols for kernel functions that produce the correct pid value depending on the context of the process. These functions provide a more robust way to get the pid. Replace the task_pid_nr() call with functions get_task_pid() and pid_vnr(). These functions are contained in the source file kernel/pid.c Note that these functions work with a struct of type pid instead of pid_t.

The task_struct has a field (comm) giving the file name of the program executed by the process. A helper function get_task_comm() is provided for access to the field. Use this function to add the program name executed to the list of information displayed.

Much of a process's total state is given in resource descriptors linked with pointers to the task_struct. Use the file system information (struct fs_struct *fs) to get and display the current working directory of the program using the helper function d_path(). The relevant data structures and function definitions can be found in <include/linux/fs.h>, <include/linux/fs_struct.h>, <include/linux/path.h>, <linclude/inux/dcache.h>, and <fs/dcache.c>

Use the memory management information (struct mm_struct *mm) to get and display the following values: number of virtual memory areas (VMAs) , the total VM (virtual memory), and amount of VM that is of the three types: shared, executable, and stack (all these values are counts of the VM pages of 4096 bytes each). There no helper functions are needed to access these values, but note that mm_struct contains a field of type rw_semaphore (a read-write semaphore) that should be used to protect read access from concurrent updates. The relevant semaphore calls are down_read() and up_read(). Note: semaphores will be covered in the 530 class on Sept. 15, 17, and 22. The slides and videos are already on the course web page.

Provide all the information specified in assignment 1 and this assignment ***for each process*** that is a sibling process to the process that called getpinfo (siblings are the other child processes of the real parent of the calling process). All the information returned to the calling program should be formatted as a character string the caller can display on stdout and having a format like that used in assignment 1. Be careful in choosing and monitoring your file buffer sizes since the amount of data displayed depends on the number of sibling processes.

Your module should deal with any errors that can occur (but not be burdened with tests for errors that cannot occur). The calling program should receive sufficient information so that it can determine if the call has been successful or has failed.

This assignment will require you to understand and use the Linux list representation and list-handling functions, specifically the structure list_head and list processing macro list_for_each_entry() (see the include file <linux//list.h>). A tutorial and examples of Linux list handling can be found in the book by R. Love, Linux Kernel Development, 3rd Ed., pp. 88-94.

Linux has a special locking mechanism “read-copy-update” (RCU) For this exercise, you should use it to protect traversing task_struct lists and reading task_struct fields. The lock/unlock calls are simply rcu_read_lock() and rcu_read_unlock() and the required include file is <linux/rcupdate.h>. The lock and unlock calls require no parameter and simply result in disabling kernel preemption.

You will need to write a testing tool that gives you a way to control some of the process state that is to be displayed. I suggest you consider reusing the fork(), exec(), and wait() code from the regular COMP 530 assignment 2 to create a program (parent process) that forks a number of child processes (the siblings) that run concurrently. Each child should exec a different executable file, including one that exec’s the caller program used to invoke the getpinfo service module (so caller is the last one of the siblings created). You can then create a number of small 4-5 line programs with different names that simply block on input from stdin after they are exec’d. If each of these programs contains a static array of 1, 2, 4, 8, … Megabytes, you should be able to partially verify that your display of VM information is reasonable. An example of this type of program can be found in /home/smithfd/530H/examples/sample_code/getpinfo_test.c. Also use the ps command to check pid values, etc. Each child process should have a unique and recognizable pid and the same parent pid.

***Submitting your program*:**

Follow the instructions from assignment 1.