CSCI 5573: Advanced Operating Systems
Fall 2019

Homework One

Due Date and Time: 11:55 PM, Wednesday, September 25, 2019

You may work in teams of size two on this homework. For the programming part, please work with a virtual machine. You will need super user privileges to do some parts of this assignment.

1. We discussed in class about how context switching is implemented in Linux kernel using *schedule*( ) and *switchto*( ) functions when a running process is preempted and another process is assigned the CPU to run. For each of the following scenarios, explain what issues may arise and how they are handled:

    a. A user kills his/her process using *kill -9* while that process is undergoing context switching while being preempted from the CPU.
    b. A currently running process completes its execution.
    c. A new process is assigned the CPU for the first time.

2. <u>Monitoring the kernel state through /proc</u>

    Write a program to report the behavior of the Linux kernel. Your program should run in two different versions.

    The default version should print the following values on stdout:
    Processor type
    Kernel version
    The amount of memory configured into this computer
    Amount of time since the system was last booted

    A second version of the program should run continuously and print lists of the following dynamic values (each value in the lists is the average over a specified interval):
    The percentage of time the processor(s) spend in user mode, system mode, and the percentage of time the processor(s) are idle
    The amount and percentage of available (or free) memory
    The rate (number of sectors per second) of disk read/write in the system
    The rate (number per second) of context switches in the kernel
    The rate (number per second) of process creations in the system

    If your program (compiled executable) is called proc_parse, running it without any parameter should print out information required for the first version. Running it with two parameters "proc_parse <read_rate> <printout_rate>" should print out information required for the second version. read_rate represents the time interval between two consecutive reads on the /proc file system. printout_rate indicates the time interval over which the average values should be calculated. Both read_rate and printout_rate are in seconds. For instance, proc_parse 2 60 should read kernel data structures once every two seconds. It should then print out averaged kernel statistics once a minute (average of 30 samples). The second version of your program doesn't need to terminate.

3. <u>Virtual memory layout of a process</u>

    Write a C program that includes multiple calls to *malloc( ), mmap( )* as well as several function calls. Include some command line arguments. Using the proc filesystem, print out the virtual memory address layout of this process during different execution phases – before and after calling *malloc( )*, before and after calling *mmap( )*, before calling a function, while a function has been invoked, and after the function returns. You will need to look at */proc/[pid]/mem* and */proc/[pid]/*maps. The layout should clearly show all the virtual memory allocated to the process and the address ranges of the executable code, heap and stack. Answer the following questions:

- Identify the locations where *argv* and *argc* are stored.
- In what directions do the stack and the heap grow? Explain your answer by using the virtual memory layouts at different execution phases.
- Your executable code will be "divided" into three different regions with different permissions. What are all these regions?
- Your allocated memory (from *malloc( )*) may not start at the very beginning of the heap. Explain why this is so.
- The program break is the address of the first location beyond the current end of the data region of the program in the virual memory. *sbrk()* with an increment of 0 can be used to find the current location of the program break. In general, that should be the start of the heap. However, you will see that this is NOT the case. You will notice that there is a gap between the program break and the start of the heap. Run your program several times and make observations about this gap. What can you say about the size of this gap? Can you make a guess as to why this gap exists?

4. Virtual memory layout of a child process

Write a C program that includes a function *foo ( )*, which creates a child process to run another C program (this could be the same program that you wrote in problem 3). Answer the following questions:

- How does the virtual memory layout of the child process compare to the virtual memory layout of the parent process before the child process executes *exec ( )*? Specify address ranges etc in your observations.
- How does the virtual memory layout of the child process compare to the virtual memory layout of another process that runs the other C program from command line? Again, specify address ranges etc in your observations.

5. Overwriting the virtual memory of a process

Write a program that creates a copy of the string "This is my initial string " and prints this string repeatedly (indefinitely) with an interval of 5 seconds between each print out. Using a scripting language such as Python or bash, write a script that locates the virtual address of the string (first program) using proc filesystem and replaces that string with another string "This is a different string". Demonstrate your program by first running the first program and then running the script. You must run your script as root.