Advanced Operating Systems - HomeWork 1

Question 1:

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:

Switching Process: If thread completes his quantum or because of any hardware or software interrupt, the thread calls the “schedule()” function and “schedule()” calls the “switch_to()” function where actually the context switch happens like saving the registers(stack pointers) of the current executing thread into the PCB and loading the saved registers and stack pointer of the next thread into memory.

a. A user kills his/her process using kill -9 while that process is undergoing context switching while being preempted from the CPU.

Ans: I thought about these problems in the two angels, 1. Hardware constraint where, the CPU has only a single core. At that point of time switching causes the problem.
2. If all the switching process is not atomic, and thread calls “schedule()” function and in between if the kill signal kills the thread before the context switch then kernel has to take care of the context switch.

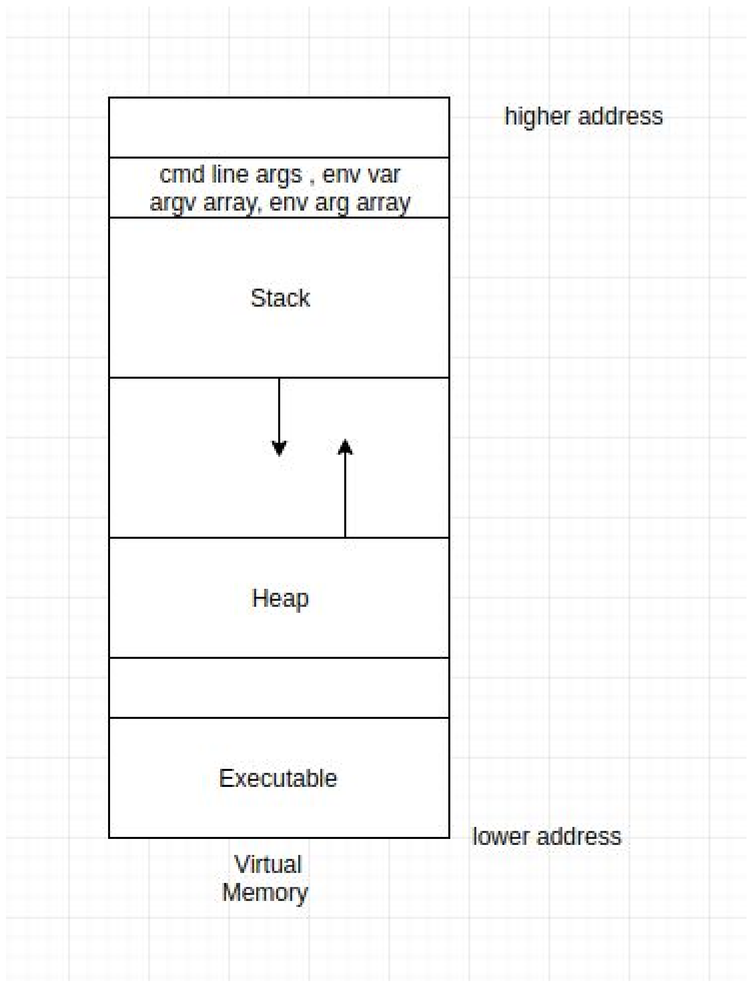
b. A currently running process completes its execution.

If the current process completes it execution i.e completes even the exit routine, then the kernel takes control of the switch process and schedules the next process appropriately.

c. A new process is assigned the CPU for the first time.

As the new process will not have any registers and stack, tge CPU loads all the data and starts the execution.

Question 3: Virtual memory layout of a process:



• Identify the locations where argv and argc are stored.
When I printed out the pointer addresses of the **argc** and **argv** variables, I found their addresses on top of the address space of the stack when I looked at the /proc/maps file.

• In what directions do the stack and the heap grow?
Stack is growing in the direction from top to bottom. To demonstrate this, I created a recursive function and created variables in it and observed the address locations for the same.
Heap grows from bottom to up, To demonstrate this, I created the memory allocations using malloc and printed the address location for the same.

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?
Executable code is divided into three parts, code segment, data segment and bss part. To demonstrate the location for the same, I printed the address of the main function , created static

variable with initialization and without initialisation. I observed intialised variables on the “” and uninitialised static variables on “” of the virtual memory.

• Your allocated memory (from malloc( )) may not start at the very beginning of the heap. Explain why this is so.
In various articles it is explained that it happens because of the address space randomisation.

• 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?
When I run it different times, The starting address memory is allocated at various locations and I’m guessing that it is because of address space randomisation to protect the process from attacks.

Question 4:

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 ( )?
When we create a child process using “fork()” system call, the address space of the parent is replicated as child’s address space. And the memory layout will be entirely similar until and unless we change any page, If we change any page, OS using the Copy On Write(COW) mechanism, it creates a separate copy for the child.

Specify address ranges etc in your observations.
The address ranges are similar for both the parent and the child.

• 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.
The address spaces are different for the processes run through the other command line option.Once the execv() is executed the address space of child and parent will be entirely different.