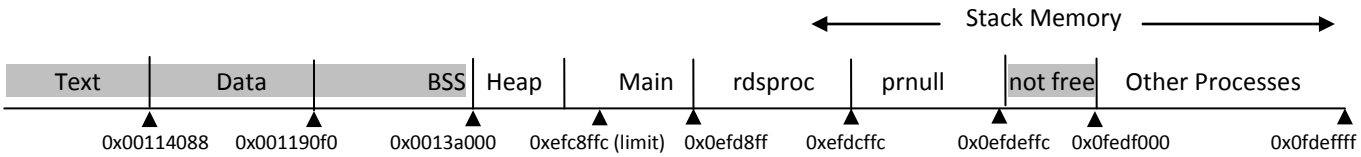
Ayush Patwari

Operating Systems [CS503]

Lab1 Answers and Discussion

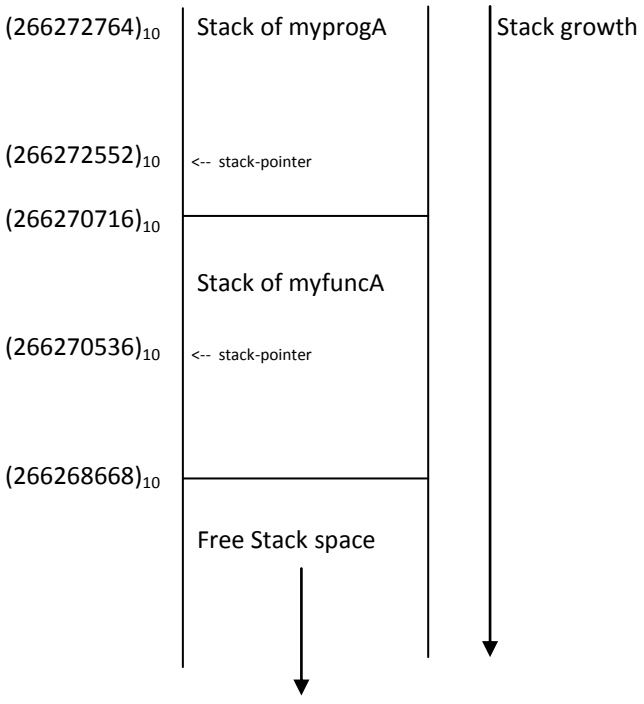
---

4. XINU Process Memory Layout



Observation:

- There is a some gap between text, data and bss sections
- As seen from xinu boot results, there are two free memory pools with a unsused space in between. When processes other than main are spawned they are allocated stacks beyond this region.
- There are two processes always running namely : prnull and rdsproc which have stacks allocated in the first free region

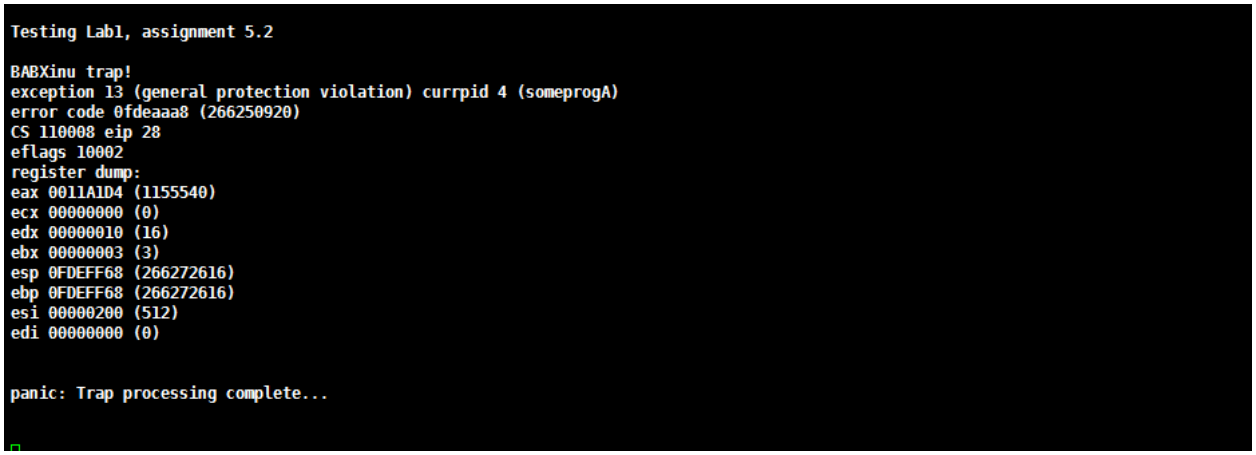


Note: Addresses are shown in decimal for better understanding

5. Intentional Stack Overflow

When process *rogueB* is able to cause a stack overflow, it actually overwrites onto the runtime stack of the process *someprogA* since they are stacked adjacent to each other (as seen above) and XINU does not impose any memory protection (any process can access any location). The stack of *someprogA* stores the values of the base-address, registers, register flags etc before being context switched out – all of these are overwritten. The return address is also overwritten such that when the processor looks for a valid instruction at some garbage address (See below: here eip has value 28) which causes a XINU trap since it tries to read from inaccessible memory address. Therefore *someprogA's* stack state is lost and it no longer can print the characters 'a' and 'A'.

As a part of the trap routine the values of the registers are dumped on the console before the operating system cleans up the process.



6. 'Hacking' into the stack of process someprogA

Since the XINU system doesn't impose protection in memory accesses and the process stacks are laid out adjacent to each other it is possible to overflow the stack of someprogA without engaging in nested function calls. To do that one would have to observe the current stack state of someprogA when it is switched out. This can be done by using the proctab[] and testing for the process whose base address coincides with the address of rogueB. The stack pointer of someprogA can then be accessed. During the context switch the return address is saved at the top of the stack and when it exits it calls the ret instruction to return to a new process. To inject new code the return address could be overwritten to that of address of the code of a new function which would then start executing. Another way could be to write an address which has the instructions to call that function and save the original return address. This would also cause that function to be executed. Immediately after that a ret instruction could be executed to return to the original return address and someprogA could continue execution as normal.