Lecture 19: OS03

Thursday, November 07, 2013 2:41 PM

IMPORTANT!: Pull code fix from handout code.

- Instructions:
  1. Make sure you committed your changes
     i. git add --all
     ii. git commit -m "<COMMIT MESSAGE>"
  2. Pull from handout
     i. git pull handout master
  3. Push your commit to origin master
     i. git push origin master

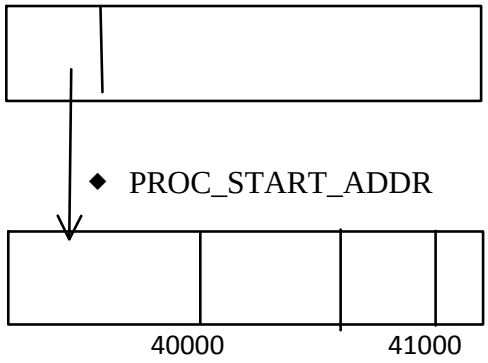
Kernel Memory:

- Kernel should be able to run all of memory,
  - Kernel code is mapped at expected address
  - Kernel data is mapped at expected address
  - Kernel stack is mapped at expected address

Let's do fork!

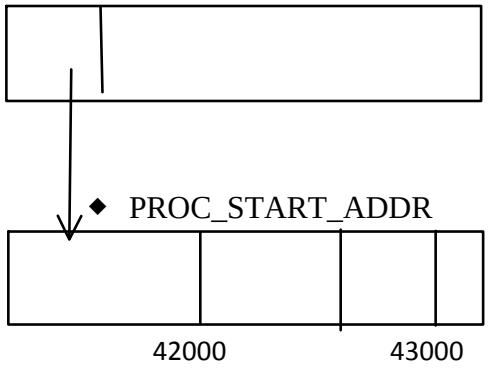
- In os02, let's implement fork!
  - First off, what does fork do?
    - A process is a program in execution; it believes that it has control of all of the OS' memory
    - A process can write bytes to a file -> analogy accessing a hardware device
    - A process can execute instructions
  - Fork is the mechanism we are given to create new processes
    - Analogy: duplicates the current process and all of its resources
    - In a real OS: are files duplicated or shared? SHARED
      - Memory (registers, process_state) resources are duplicated
      - I/O device resources (files) are shared
    - Returns twice (only thing that differentiates the processes)
      - Return to parent: child's id
      - Return to child: 0
  - Interesting historical fact:
    - Why fork? Originally, back in the 70s, the original UNIX had support for one process at a time.
    - There was a shell you could type commands into it; they could hack it into supporting 1.5 processes
      - Overloaded the exit command to put that state in memory
      - Took 20 lines of assembly language that allowed the place that the shell was saved into a process descriptor that was logically running at the same time
  - Let's edit kernel.c (note it's not implemented in lecture os02, branch v12)
    - Need to copy the registers
      - memcpy(&processes[2].p_registers, current->p_registers, sizeof(current->p_registers));
    - Return 0 to child process (via eax)
      - processes[2].p_registers.reg_eax = 0
    - Return pid to parent process (via eax)
      - current->p_registers.reg_eax = 2
    - Set state to runnable
      - Processes[2].p_state = P_RUNNABLE

        - Set the pagetable to the correct value (need 2 free pages and calls to virtual_memory_map)
            - page_alloc_unused();
        - ***In the lecture, only process 2 is free
    - Eddie decided to skip to the end!
        - git checkout v09
        - Parent:
    - L1



    - L2

- Child
    - L1



    - L2
    - Copy the kernel portion of the address space (DO NOT COPY DATA HERE)
    - For user portion, copy every page
        - Assumption is that the physical addresses are mapped
        - Assumes that the virtual address maps to the same physical address
        - Kernel uses virtual address (EVERY ADDRESS IN CODE IS VIRTUAL!)
        - Physical addresses can only be put in pagetables

- What happens when a process tries to access a page it doesn't have access to?
    - PAGEFAULT!
        - i.e. write to kernel memory
    - When this happens, the kernel takes control, and the a register tracks the error address
        - In this case, cr2 register tells us the address that faulted
        - Restores the state before the fault, which allows the operating system to restart the process so that it can identify the fault
    - A process' code is usually mapped PTE_U but NOT writable
        - Process' stack is PTE_U | PTE_W
        - Why? To prevent the process from rewriting its own process

- What is copy-on-write?? (Step 6)
    - When you copy a process' physical page, you have to mark BOTH the parent's permission as READ-ONLY and child's permission as READ-ONLY

- Why? Because you don't want anyone to make a change to it. If they do, the other process will not know so it would be bad
- When you mark as READ-ONLY, you let the process handle it later when the process tries to write into it, it'll copy it and then write to new physical address