Scribe Notes – 11/07/13

Assignment 4:

- Update pset 4 to get new code (bugs fixed).
    - Possibly commit first to avoid conflicts (git pull will tell you if necessary).
- There are 2^10 (1024) entries in each page table. Each pointer is 4 bytes.
- Direct (identity) mapping maps every virtual address to the same physical address.
- Top 10 bits determine level 1 page table entry and second 10 bits determine level 2 page table entry.
- Permissions are also in the page table entries (0|PTE_P|PTE_W|PTE_U).
- Kernel memory only has permissions 0|PTE_P|PTE_W, so that user processes cannot alter that memory.

Lecture:

- Working in os01, version 12 (separate page tables for hello and welcome processes):
    - Process 1 (hello) page faults because it was trying to modify welcome's code.
    - Initializing permissions to 0 causes constant rebooting because the memory will not be accessible to even the kernel.
    - Process page table must allow at least kernel permissions for the process to run.
- Working in os02:
    - fork()
        - Implemented on a branch of os02; will help with pset implementation.
        - Fork allows process to duplicate the computer.
        - Creates new processes by duplicating current process and its resources (some are shared).
            - memory + registers + process states = duplicated
            - I/O devices (files, etc.) = shared
        - Fork system call returns twice: once for child, once for parent.
        - Return value to parent = child ID; return value to child = 0.
        - Original computers (from the 70s) could run only one process.
            - Was updated to "1.5" processes (a process runs while a shell of the other process is suspended in memory)
            - Now computers allow many processes running concurrently because of fork.
        - This limited version of fork we will create here can only copy into process 2 (process 2 asserted to be free).
            - processes[2].p_registers = current->p_registers;
              processes[2].p_registers.reg_eax = 0;
              current->p_registers.reg_eax = 2;
    - Process_main should print successive numbers doubled (i.e., 0 then 0 then 1 then 1 then 2 then 2 and so on).
        - Printing successive numbers; should be printing successive numbers with doubles (with a bit of interrupt timing error possible).

        - The child process is not running. It must be set to be runnable.
            - processes[2].p_state = P_RUNNABLE;
            - Error – machine instantly reboots.
            - No page table (need two free pages to make both levels of pagetable).
    - Skip to os02, version 09 (fork implementation):
        - proc* copy = &processes[2];
          pagetable_alloc(copy);
          for (uintptr_t va = PROC_START_ADDR; va < MEMSIZE_VIRTUAL; va += PAGESIZE); {
              vamapping vam = virtual_memory_lookup(current->p_pagetable, va);
              if (vam.perm) {
                  void* page = page_alloc_unused();
                  memcpy(page, (void*) vam.pa; PAGESIZE);
                  virtual_memory_map(copy->p_pagetable, va, (uintptr_t) page, PAGESIZE, vam.perm);
              }
          }
            - This code copies the address space.
                - Allocates new pagetables.
                - Starts at the process starting address and copies the memory and permissions from the parent process to the child process for every address that has memory at it (i.e., vam.perm exists).
        - Say physical address of code page is 0x40000 and physical address of stack page is 0x41000.
        - In page tables for child process:
            - Level 1 page table entries must be at different addresses than parents (need more than one page table in order to separate processes memory from each other)
            - In level 2 page table:
                - Kernel should be the same.
                - First page in user memory – say 42000 (code page) [outside of parent process' memory].
                - No page allocations in kernel address space; copy all of user address space from parent process (i.e., share the kernel).
                - Process to follow for memory copying:
                    - Look up, allocate, copy, map.
        - memcpy assumes pages are mapped and that they are identity mapped.
        - Side note: all code memory is actually virtual addresses.
        - Can only memcpy if physical addresses would work as virtual addresses.

- vamapping check_vam = virtual_memory_lookup (current->p_pagetable, vam.pa);
  assert(check_vam.pa == vam.pa);
    - Asserts an identity mapping.

BREAK

- Write to kernel memory causes page fault.
    - cr2 register = address that faults.
    - CPU restores process state to what it was right before fault; allows process to resume again and consider the fault transparent if the fault has been fixed.
    - eip register = faulty instruction address.
- No operating system initializes process code to writeable; instead initializes to user accessible, but not writeable.
- A lack of copying pages is only detectable when a change in one process is detectable by another process.
- Mark pages as read-only in child process; when child writes to pages, we get a fault, and the kernel can examine the fault, copy the page, and exchange a new physical address for the page address with write permission.
- Mark parent addresses as read-only as well.
- If both parent and child try to write to the same page, the page is not shared after the first write because a copy was made, so we now just have to change the page permissions to writeable on the original page for the process that writes second.
    - Original page is no longer shared, so it can be written to without altering other processes' memory.
- mmap() allows for permissions PROT_READ (corresponds to PTE_P|PTE_U), PROT_WRITE (corresponds to PTE_P,PTE_U|PTE_W), MAP_SHARED, and MAP_PRIVATE (copy-on-write).
- How to find how much memory is in active use?
    - Idea: make a copy of each page table, erase the original versions, faults recreate original mapping
        - amount of faults = amount of memory in current use
    - Actual process used:
        - Analogy:
            - There are post office boxes (represent memory) and a rabid wolverine. Put the wolverine in one box per day and see if anyone gets lacerations. The probability of a box being used times the number of boxes ~ number of active boxes (assuming boxes are used at random).
        - Actual implementation: make a copy of only part of the page table, perform copy-fault counting process described earlier, repeat many times.
- How should operating systems (virtual machines) decide when to transfer control of the CPU to and from each other?

    - The CPU detects how much memory is in use by an operating system (virtual machine).
    - Operating systems that use less memory less often are run less often and have their unused memory gifted to other virtual machines.
- Some systems compress the memory of a process by sharing physical memory pages that contain only the value 0 with a system 0-value page.
    - Shared pages are not writeable.