CS61 Scribe Notes
Date: 11.6.14
Topic: Fork, Advanced Virtual Memory

Scribes:

Mitchel Cole
Emily Lawton
Jefferson Lee
Wentao Xu

Administrivia:

- Final - likely less of a time constraint
- What can we do during reading period:
    - Turn in problem set updates for improved grades
    - Problem Set 6 can be turned in during this time
    - Opportunity to sign up for meetings with Prof. or TFs to talk through code = code style feedback

Look at os02 directory:

- p-alloc.c - page allocator process much like in the problem set.
    - 1. locate address that is first heap page end - last address in the globals
    - 2. call sys_page_alloc(data) - allocates a page of memory at that address.
    - 3. write on that address
    - 4. check that writes worked
- Key Implementation - in the kernel:
    - INT_SYS_PAGE_ALLOC - find an unused page and map it at the address the user asks for. Permissions are PTE_P, PTE_W, PTE_U. If successfully allocated, return 0. If error, return -1.
- What happens if we get rid of PTE_W?
    - Will fault on first address for allocated page => write protection problem.
- How can we get the kernel to crash?
    - Edit p-alloc.c. We can add one or two characters and cause p-alloc.c to crash the kernel.
        - Think about the invariants that need to be maintained:
        - Method 1: Allocate a really low address and mess with kernel memory.
            - Provide a weird address to sys_page_alloc. Provide the page of memory say containing kernel itself. What happens when that runs? We get a panic within the kernel. Virtual_memory_map() is getting lost. Because we allocated a zero page over the kernel's code, crazy stuff happened.
            - add *int r = sys_page_alloc((char*) 0x4000*
            - Results: deputy is now confused because 0 page is allocated to kernel memory:

        - Process page table is enforced when exception happens.
    - Method 2: Pass in a large memory into virtual address for virtual memory mapping.
    - Method 3: Pass in page address that is not aligned. *int r = sys_page_alloc(data + 1)*
        - Results: page panic
        - Validate the data that is passed in very carefully. Process isolation saves you from having to consider everything in the world from being malicious. The kernel is in a different situation. It's the kernel's job to keep the processes isolated. Make sure the inputs are not crazy. If the kernel does something wrong with crazy inputs, BOOM!!!
        - Prevention:
            - *if (page && wantva >= (char*) PROC_START_ADDR && wantva < MEMSIZE_VIRTUAL && wantva % PAGESIZE == 0)*
        - If we screw up the virtual address that contains the exception handlers then the next time we take an exception, BOOM!!!
    - How can you avoid a memory leak?
        - Check the user argument before doing anything. Free it.
            - *current->p_registers.reg_eax = -1;*
            - *break;*

CVE: Common Vulnerabilities and Exposures

- Database of serious security bugs
    - "serious" => how easy it is to exploit
- Privileged vulnerabilities:
    - unprivileged process can still access privileged process
- Example exploitation:
    - Win some addresses that the user can write - write information into that space. By calling another system call, triggers vulnerability - accesses the code the user has shoved into the kernel. That code gives the running process root privilege.

fork:

- How do we start a new process?
- How do we exit a process?
- How do processes communicate?
- Fork takes one process and turns it into 2. All of the process's resources are copied.

pid_t fork():

- [in operating systems derived from Unix tradition]
- A function that returns twice

- Copies the current process
- Returns 0 to the copy
- Returns <ID of the new process> to the original

Hypothetical thinking:

- What if we had a system call called *new process* which returned the process ID of a new process?
    - What are the consequences of this idea? Why don't we just start up an empty process?
        - You need some mechanism for the old process to put stuff into the new process's memory. Not only into its memory but into its initial registers. the goal of process isolation is to keep processes separate. Process isolation is a construct - something that we could define. That feels more dangerous. Rather have to jump through at least one hoop to get control.
        - Before the setup, child process is not runnable and lacks any memory.
    - This introduces a new idea: *un-isolated process*
- We don't allocate memory by copying. We do, however, allocate a new process by copying.

WeensyOS:

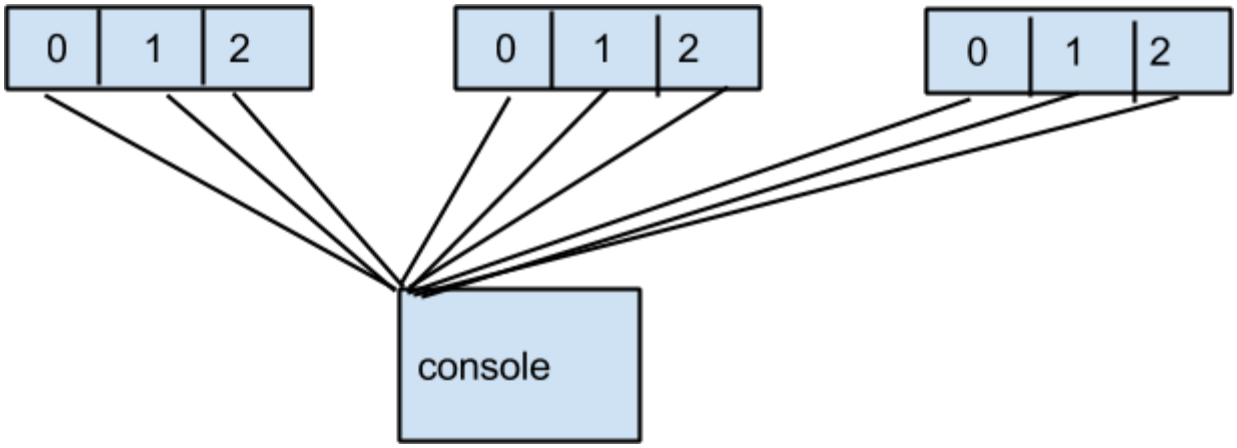
- Parent process and child process are usually running on different physical pages. Because they are now isolated, changes to one should not affect the other. The processes can't tell what physical page they're running on. We don't allow the process even to look at its own pagetable.
- copy process memory(stack, heap, data, text)
- copy page table, registers
- different physical pages, but they can't tell the difference
- also different - pid and eax register
- So what is different between parent process and child process?
    - Parent process and child process are running on different address.
- The classic way to see fork used in a process:
    - (Unix or WeensyOS)

```
pid_t p = fork();
if(p==0 {
        /*child*/;
}
else {/*parent */
        if (p < 0) /*error*/
        if (p > 0) /*child*/
}
```

- Process ID's are in sequential order. Take a look in l18 directory. Run fork1.c:
    - What does this do?

        - One prints, finishes, then the other prints.
        - Processes share I/O devices - a way that processes can communicate. Best example - files.
    - outputs:
        - Hello from pid 13629
        - Hello from pid 13630
    - Both processes printed. The process IDs look like they were allocated sequentially. The second thing we notice is that the message from the first process is always printed before we get a new prompt. The second process message is always printed after we get a new prompt. Did we get a new terminal? No.
    - The goal of process isolation: gives process illusion it has full control of the machine. If we were to enumerate the resources a machine has, a line is drawn across those resources - own cpu time, etc. but share the same i/o devices. I/O devices are shared - way the processes communicate. They communicate over devices. The model for those devices is files. Files are the main model for I/O devices in Unix operating systems.
    - Examples:
        - stdin
        - stdout
        - stderror
    - What actually happens when we type and call ./fork1?
        - examine using strace
        - reads fork - shell creates child, then waits for the child to finish - child exec fork - fork calls fork
        - the first fork finishes before the second fork is booted - we don't have this weird behavior if we do ./fork1 > f, where f is an output file

K file descriptor table



0 = stdin
1 = stdout
2 = stder

exec:

- replaces memory with a fresh program image from disk
- Kernel needs to have some version of exec in it because kernel loads the program from disk.

Let's take a look at some other things the shell can do:

- ./fork1 > f
- What do we notice?
    - Nothing is printed to the console. It's because the file descriptor changed. f has messages from both children.
- How did the shell actually do this?
    - The shell read a command. Next, shell calls open() which creates the output file. This adds an entry to our file descriptor table.
    - Skip ahead to clone() - which creates the child.
    - dup(3, 1) makes new standard output point to where 3 already points.
    - In the other process, it erases the third file descriptor (which is irrelevant now) and then runs. Redirects output to file f. Now we can tell what happens once the shell running fork1 forks again. The second fork creates a copy of the file descriptor table and that table is a copy of fork1's so its standard output also points to f. This table provides a universe in which the process runs.
- What is cool about this?
    - The code for setting up the universe is the code that happens between the fork and exec. This is where the magic of unix really happens!

Linkers (*digression)*:

- elf format: executable linking format

- Look in the link directory and see script that tells the linker how to smash a bunch of executables/object files together into a executable. This is so we can build a single binary out of a bunch of object files.
- -etext - symbol that is defined at that point
- -put read data together
- -read/write data should be on a page-aligned offset.