CSE 3430

Unix/Linux fork()

1. To create a process, Unix/Linux uses fork(), which is a system call.
2. When a fork() call is made/executed by a process:
    a. The process executing fork(), called the parent, makes a system call to the OS which causes a new process (a child of the parent) to be created (assuming no error occurs, and they only occur ***very*** rarely).
    b. The fork() system call returns (in the return register, as always):
        i. If an error occurs, -1 is returned to the parent, and no child process is created; else
            a. The pid of the child is returned to the parent; and
            b. 0 is returned to the child.
    c. Notice that the different return values for the parent and child can be used to have the parent and child execute different code after return from the fork system call.
    d. Also notice that the value returned to the child is NOT a pid (only the init process in the OS has pid 0; it is the "ancestor" process of all other processes running in the system each time the system is booted); the 0 returned to the child is A RETURN VALUE from the system call, but it is NOT a pid! The value returned to the parent IS a pid, but it is NOT the pid of the parent (rather, it is the pid of the CHILD).
    e. When the OS creates the child, the child is assigned its own pid by the OS, and its pid will be greater than the pid of its parent (often 1 greater, but it depends on whether or not any other fork system calls were made (by other processes running in the system) between the time the parent was created and the time the fork call is made in the parent).
    f. The child gets its own separate address space (its own space in memory), which INITIALLY contains a copy of the parent's address space (both code and all data).
    g. Linux implements fork() using copy-on-write pages [discussed later], so process creation is quite efficient.

h. As part of creating the child process, the OS will also create a process control block (PCB) for the child, which will be used by the OS to keep track of data needed to manage the process, such as the pid of the process, current process execution state (register values), process state (ready, running, blocked), open files, etc.

i. Notice carefully that the child ALSO gets a copy of the parent's **register values** (PC, PSR, in Intel, rax, rbx, etc.), except the OS will write the pid of the child to the parent's rax (return register in Intel), and write 0 to the child's rax (return register in Intel).

j. In some cases, the child process may subsequently overwrite its address space with a different program to execute. This can be done using exec() system calls, such as execvp() [discussed later].

k. ***Notice carefully***: When the parent process makes the fork call, ***the PC has already been incremented to the address of the next instruction after the fork() instruction when the fork instruction executed***, so the fork() call is only made once ***by the parent*** (that is, the parent will start executing instructions starting with ***the next instruction after the fork call instruction*** when the parent runs again). Also notice carefully that the child gets a copy of all the code and data for the parent, and also all of the parent's registers (context), so ***the child also gets the incremented PC*** (which has the address of the next instruction after the fork() call); therefore, the child will not execute the fork call, but rather, will start executing instructions starting with ***the next instruction after the fork call instruction***.

l. Notice also that, because the child gets a copy of the parent's address space, it also gets a copy of the parent's stack, so if the parent makes the fork call from a function which has been called by the parent, the parent will return to the function from which the function call was made when the currently executing function terminates. This will also, then, be true for the child process!