Indian Institute of Technology Bhilai
Department of Computer Science
CSL301: Operating Systems

Take Home Assignment – 1

Part 1 - FORK

Question 1: Write a C program that declares a global integer initialized to 10 before calling `fork()`. After the fork, have the parent add 5 to the variable and the child add 10. Each process should print only its final value of the variable and a concluding statement about whether their changes affected each other.
Explain why changes made to a global variable in the child process after `fork()` do or do not affect the parent's copy of the variable. Describe how `fork()` impacts process memory and variable states.

Question 2: Write a C program that calls `fork()` twice in a row. Each process should print its own PID and its parent's PID. At the end, print how many processes you observe running your program (from your perspective).
Describe how two successive calls to `fork()` result in multiple processes. Explain the process tree structure that emerges and why the total number of processes is what you observe.

Question 3: Write a C program where the parent opens a file for writing, then calls `fork()` to create a child. Both parent and child should write their own distinct messages to that open file descriptor. Add your name in the file written by the parent and for child write your roll number. Each process should print a message indicating it wrote to the file.
Explain how file descriptors and the file offset behave between a parent and child process after a `fork()`. Discuss what happens when both processes write to the file through a shared file descriptor and what the expected outcome is.

Question 4: Write a program where the parent forks two child processes sequentially. Each process prints its PID, its parent's PID, and a role message ("I am {your name}", "I am first child", or "I am second child"). End with a print summarizing the total number of processes running this code.
Discuss the roles of the parent and two child processes created sequentially in a `fork()` program. Explain how the process creation order and PID relationships are reflected in their output and what that implies about process hierarchy and execution.

Part 2 - XV6 : Add System Calls

Add syscalls to "Set" and "Get" a User-Defined Flag in the Process Table

These syscalls allow each process to save and retrieve an integer flag (`userflag`) in the process structure through two syscalls.

Step 1: Update the process structure (`proc.h`)

Add a new integer variable to the structure `struct proc` (often at the end). This will act as a per-process integer variable accessible via syscalls.

Step 2: Assign syscall numbers (`syscall.h`)

Assign unique syscall numbers for `setflag` and `getflag`. Add two `#define` statements with two unused syscall numbers (for example, 26 and 27) in the file.

Step 3: Declare handlers and update syscall table (`syscall.c`)

Declare the kernel handler functions for `setflag` and `getflag` using `extern`. Add their references to the syscall dispatch table at their respective syscall numbers.

Step 4: Implement the kernel handlers (`sysproc.c`)

Implement two handler functions that interact with the user-defined flag stored in each process's structure:

- `sys_setflag`:
  - Use `argint()` to safely fetch an integer argument passed from the user program.
  - Call `myproc()` to get a pointer to the current process's `proc` structure.
  - Assign the fetched integer value to the process's `userflag` field.
  - Return 0 on successful assignment.
  - Return -1 if `argint()` fails (invalid argument).
- `sys_getflag`:
  - Call `myproc()` to get the pointer to the current process's `proc` structure.
  - Access and return the `userflag` field value from that structure.

Using `myproc()->userflag` in these functions ensures that each process reads and writes its own flag safely.

Step 5: Update user prototypes (`user.h`)

Declare the prototypes for both syscalls so user programs can call them.

Step 6: Add syscall assembly stubs (`usys.S`)

Add SYSCALL macros for `setflag` and `getflag` to this file so user calls are translated into actual syscalls.

Step 7: User Program Example (`testflags.c`)

Write a small user program that:

- Calls `setflag` with one value, retrieves it with `getflag`, and prints it.
- Calls `setflag` again with a different value, retrieves and prints again.

Use `printf` to display the outputs.

Final step: Add `_testflags` to your Makefile's `UPROGS` line so the test program is built and available in xv6. Use last four digits of your roll number to set the flag.