CS3500: Operating Systems
Lab 4: Implementing User Space Traps

2^{nd} September 2025

1 Introduction

In this lab, you will implement an example of user-level trap handling.

Before you start coding:

- Read Chapter 4 of the xv6 book.
- Review the following `xv6 source files`:
  - `kernel/trampoline.S`: the assembly involved in changing from user space to kernel space and back.
  - `kernel/trap.c`: code handling all interrupts.

2 Lab Setup

You have been provided with the `alarmtest.c` file in the zipped folder. Add this file to your `xv6` repository in the `user/` directory. Add it to the Makefile. It won't compile until you have solved this lab.

3 Problem

In this lab, you will add a feature to `xv6` that periodically alerts a process as it uses CPU time in the form of user-level interrupt/fault handlers. Your solution is correct if it passes `alarmtest` and `'usertests -q'`.

(a) To achieve this, you need to add a new `sigalarm(interval, handler)` system call. If an application calls `sigalarm(n, fn)`, then after every n "ticks" of CPU time that the program consumes, the kernel should cause application function `fn` to be called. (A "tick" is a fairly arbitrary unit of time in xv6, determined by how often a hardware timer generates interrupts.)

(b) When `fn` returns, the application should resume where it left off. Implement a `sigreturn()` system call to achieve this. User alarm handlers are required to call the `sigreturn` system call when they have finished (look at periodic in alarmtest.c for an example). You should add code to `usertrap()` and `sys_sigreturn()` that cooperate to cause the user process to resume properly after it has handled the alarm.

Note: If an application calls `sigalarm(0, 0)`, the kernel should stop generating periodic alarm calls.

`alarmtest` calls `sigalarm(2, periodic)` in `test0` to ask the kernel to force a call to periodic() every 2 ticks, and then spins for a while. Your solution is correct when `alarmtest` produces the following output and `usertests -q` also runs correctly:

```
$ alarmtest
test0 start
........alarm!
test0 passed
test1 start
...alarm!
..alarm!
...alarm!
..alarm!
...alarm!
..alarm!
...alarm!
..alarm!
...alarm!
..alarm!
test1 passed
test2 start
................alarm!
test2 passed
test3 start
test3 passed
$ usertest -q
...
ALL TESTS PASSED
$
```

Steps:

1. Modify the Makefile to cause `alarmtest.c` to be compiled as an xv6 user program.
2. Add declarations for `sigalarm` and `sigreturn` in `user/user.h`:

    ```
    int sigalarm(int ticks, void (*handler)());
    int sigreturn(void);
    ```

3. Update `user/usys.pl`, `kernel/syscall.h`, and `kernel/syscall.c` to allow `alarmtest` to invoke the `sigalarm` and `sigreturn` system calls.
4. Your `sys_sigalarm()` should store the alarm interval and the pointer to the handler function in new fields in the proc structure (in `kernel/proc.h`).
5. Add definitions of `sys_sigalarm` and `sys_sigreturn` to the kernel in `sysproc.c`.
6. You will need to keep track of how many ticks have passed since the last call (or are left until the next call) to a process's alarm handler; you will need a new field in `struct proc` for this too. You can initialize proc fields in `allocproc()` in `proc.c`.
7. Every tick, the hardware clock forces an interrupt, which is handled in `usertrap()` in `kernel/trap.c`. You only want to manipulate a process's alarm ticks if there is a timer interrupt; you want something like

    ```
        if(which\_dev == 2) ...
    ```

8. You will need to modify `usertrap()` so that when a process's alarm interval expires, the user process executes the handler function.

Hints:

1. Your solution will require you to save and restore registers to correctly resume the interrupted code.
2. Have `usertrap` save enough state in `struct proc` when the timer goes off that `sigreturn` can correctly return to the interrupted user code.
3. Prevent re-entrant calls to the handler: if a handler has not returned yet, the kernel should not call it again. (test2 tests this)
4. Make sure to restore `a0`. (`sigreturn` is a system call, and its return value is stored in a0).

4 Submission

- You are required to zip and submit the full folder and a report highlighting the changes you made in the relevant files. Note that only your contributions/modifications to the code are required.
- In the report, very briefly explain the logic/reasoning of the code you implemented for each question. We are looking for the core concept and not a line-by-line explanation of your code.
- Your zipped folder should be named $< Roll\_No >$_Lab4.zip.