CS3500: Operating Systems

Lab 5: Signals

October 1, 2021

Introduction

In this lab we will use system calls and xv6 paging to a **tracing and alert mechanism** in xv6.

Resources

Similar to the previous assignment, please go through the following resources before beginning this lab assignment:

1. The **xv6 book**: **Chapter 4** (**Traps and System Calls**): sections **4.1**, **4.2**, **4.5**
2. Source files: `kernel/trampoline.S` and `kernel/trap.c`

Note

As part of this assignment, we have provided a clean version of the xv6 repo, with the required files included in it. Please implement your solutions in this repo only. We have also attached the LaTeXtemplate of this document. Please write your answers in this file and submit the generated PDF (NOT the `.tex`).

1 Wake me up when Sep ⋯ (40 points)

From emails to WhatsApp notifications, we often rely on alerts for certain events. In this section, you will add such an alarm feature to xv6 that alerts a process as it uses CPU time.

1. (2 points) Think of scenarios where such a feature will be useful. Enumerate them.

Solution:
1) The most basic scenario is alarm/timers on electric gadgets.
2) Alerting user after fetching mails from server/messages from sim carrier network and any time-restricted utilities
3) Automated Traffic lights show a similar behavior.
4) Pushing notifications/ Rendering frames on a display

2. (38 points) More generally, you'll be implementing a primitive form of user-level interrupt/fault handlers. You could use something similar to handle page faults in the application, for example. Feel free to refer to the hints at the end of this section.

   (a) (10 points) 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 the 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.)

   Also create a simple `sigreturn()` system call which does nothing but returns 0 for the time being. Inoke `sigreturn` at the end of the alarm handler function `fn`.

   HINT: You need to make sure that the handler is invoked when the process's alarm interval expires. You'll need to modify `usertrap()` in `kernel/trap.c` so that when a process's alarm interval expires, the process executes the handler. To this end, you will need to recall how system calls work from the previous labs (i.e., the code in `kernel/trampoline.S` and `kernel/trap.c`). Mention your approach as the answer below. Which register contains the user-space instruction address to which system calls return?

   Solution:
   1) For system call sigalarm(), protocol definition have been added in the corresponding files (just as any other systemcall). Where, in the definition, alarm_handler & timer_interval are read using helper functions the values are then assigned to proc struct.
   2) Since the call sigalarm() passes the arguments for interval handler, corresponding fields alarm_handler, time_interval, ticks_counter have been added to struct proc in kernel/proc.h to store the address of function handler, the interval for that timer, counter to update ticks respectively. And are initiated in proc.c (allocproc) to 0, all except ticks_counter, which is set to -1 to distinguish the unique state.
   3) In trap.c file, under the which_dev == 2, which corresponds to a timer interrupt, ticks_counter is increased by 1, each time it's invoked, and after a certain stage (series of increments), when the ticks_counter equals the time_interval, user program counter is pointed to the alarm_handler, as the required time interval's been met.

   (b) (8 points) Complete the `sigreturn()` system call, which ensures that when the function `fn` returns, the application resumes where it left off.

   As a starting point: user alarm handlers are required to call the `sigreturn()` system call when they have finished. Have a look at the `periodic()` function in `user/alarmtest.c` for an example. You should add some code to `usertrap()` in `kernel/trap.c` and your implementation of `sys_sigreturn()` that cooperate to cause the user process to resume properly after it has handled the alarm.

   Your solution will require you to save and restore registers. Mention your approach as the answer below. What registers do you need to save and restore to resume the interrupted code correctly? (**HINT**: it will be many).

Solution:
Method 1 (saving the entire trapframe):
1) A new field entry save_state has been added to struct proc to save the trapframe before serving the interrupt, so as to retrieve it after the interrupt's been processed.
2) So, correspondingly, in kernel/trap.c, before program counter is modified, the trapframe is loaded on to the save_state field to latter return to the interrupted program. And to prevent call being repeated, a status field has been added, so when it's not null, it means that the interupt's currently running, so the block will not be executed.
3) In the sigreturn() system call, the trapframe is loaded from the save_state field, which was used to store trapframe earlier when the interrupt occurred. Thus, the interrupted program will be resumed.
Method 2 (saving only the required registers):
1) All the steps are same as what was done in Method 1 except for save_state being an entire trapframe, which can be alternatively stored in a a new structure (struct context) that stores only the required registers, instead of all. Namely, s0, s1, s2, - - -, s11 besides ra, sp.
2) In this case, when copying restoring these registers, instead of loading the whole tf, only callee registers are copied onto therefore restored later, both processes are equivalent, per se, just that you don't store the whole trapframe, but rather just use swtch(), which is a context switch assembly program to save current registers in old, in case of kernel context switches (interrupts here).

(c) (20 points) There is a file named `user/alarmtest.c` in the xv6 repository we have provided. This program checks your solution against three test cases. `test0` checks your `sigalarm()` implementation to see whether the alarm handler is called at all. `test1` and `test2` check your `sigreturn()` implementation to see whether the handler correctly returns to the point in the application program where the timer interrupt occurred, with all registers holding the same values they held when the interrupt occurred. You can see the assembly code for `alarmtest` in `user/alarmtest.asm`, which may be handy for debugging.

Once you have implemented your solution, modify `Makefile` accordingly and then run `alarmtest`. If it passes `test0`, `test1` and `test2`, run `usertests` to make sure you didn't break any other parts of the kernel. Following is a sample output of `alarmtest` and `usertests` if the alarm invocation and return have been handled 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
$ usertests
...
ALL TESTS PASSED
$
```

1.1 Additional hints for test cases

`test0`: Invoking the handler

Get started by modifying the kernel to jump to the alarm handler in user space, which will cause `test0` to print "alarm!". At this stage, ignore if the program crashes after this. Following are some hints:

- The right declarations to put in `user/user.h` are:

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

- Recall from your previous labs the changes that need to be made for system calls.
- `sys_sigalarm()` should store the alarm interval and the pointer to the handler function in new fields in `struct proc` (in `kernel/proc.h`).
- To keep track of the number of ticks passed since the last call (or are left until the next call) to a process's alarm handler, add a new field in `struct proc` for this too. You can initialize `proc` fields in `allocproc()` in `kernel/proc.c`.
- Every tick, the hardware clock forces an interrupt, which is handled in `usertrap()` in `kernel/trap.c`. You should add some code there to modify a process's alarm ticks, but only in the case of a timer interrupt, something like:

```
if(which_dev == 2) ...
```

- It will be easier to look at traps with gdb if you configure QEMU to use only one CPU, which you can do by running:

```
make CPUS=1 qemu-gdb
```

`test1`/`test2`: Resuming interrupted code

Most probably, your `alarmtest` crashes in `test0` or `test1` after it prints "alarm!", or `alarmtest` (eventually) prints "test1 failed", or `alarmtest` exits without printing "test1 passed". To fix this, you must ensure that, when the alarm handler is done, control returns to the instruction at which the user program was originally interrupted by the timer interrupt. You must ensure that the register contents are restored to the values they held at the time of the interrupt, so that the user program can continue undisturbed after the alarm. Finally, you should "re-arm" the alarm counter after each time it goes off, so that the handler is called periodically. Here are some hints:

- Have `usertrap()` save enough state in `struct proc` when the timer goes off, so that `sigreturn()` can correctly return to the interrupted user code.
- Prevent re-entrant calls to the handler: if a handler hasn't returned yet, the kernel shouldn't call it again. `test2` tests this.

Submission Guidelines

1. Implement your solutions in the provided xv6 folder. Write your answers in the attached LaTeXtemplate, convert it to PDF and name it as `YOUR_ROLL_NO.pdf`. This will serve as a report for the assignment.
2. Put your entire solution xv6 folder, and the `YOUR_ROLL_NO.pdf` in a common folder named `YOUR_ROLL_NO_LAB5`.
3. Compress the folder `YOUR_ROLL_NO_LAB5` into `YOUR_ROLL_NO_LAB5.tar.gz` and submit the compressed folder on Moodle.
4. NOTE: Make sure to run `make clean`, delete any additional manual and the `.git` folder from the xv6 folder before submitting.

Submission Guidelines

1. Implement your solutions in the provided xv6 folder. Write your answers in the attached LaTeXtemplate, convert it to PDF and name it as `YOUR_ROLL_NO.pdf`. This will serve as a report for the assignment.
2. Put your entire solution xv6 folder, and the `YOUR_ROLL_NO.pdf` in a common folder named `YOUR_ROLL_NO_LAB5`.
3. Compress the folder `YOUR_ROLL_NO_LAB5` into `YOUR_ROLL_NO_LAB5.tar.gz` and submit the compressed folder on Moodle.
4. NOTE: Make sure to run `make clean`, delete any additional manual and the `.git` folder from the xv6 folder before submitting.