CS3500: Operating Systems

Lab 5: Signals

Abishek S EE18B001

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:**
> The feature can be used in places where there is a constraint wrt time or a cost corresponding to the time elapsed/used. For example,
>
> - The feature can be used by OS for security purposes to monitor if a critical process is running without problems by polling it periodically.

- It can be used to ensure that a process runs for a given time only and kill it on timing out, like setting timeout for bash processes and program executions.
- It can be used by a process which requires a periodically recurring event to be executed, like notifications for software updates.
- It can be used for executing something after a given time, like how a photo is clicked with timer.
- It can be used by cloud service providers to check that a container or virtual compute is not idle for too long. It can also be used to charge based on the compute time taken.

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:
   The approach I used is as follows:

   - The system call function definitions are added to `user/user.h` and entries to `user/usys.pl`.
   - The two system calls' index are added to `kernel/syscall.h` and prototypes to `kernel/syscall.c`.
   - In the `struct proc` of `kernel/proc.h`, three new entries are added :
     - `alarm_nticks` (type `int`): number of ticks passed to the `sigalarm()` denoting periodicity of invoking the handler
     - `alarm_ticks_passed` (type `int`): number of ticks elapsed since the last time handler was invoked

- alarm_handler_addr (type uint64): the user space address of the handler function to be invoked

- alarm_nticks and alarm_ticks_passed are initialized to invalid values -1 in kernel/proc.c/allocproc() which can be used to check whether an alarm is active for a process in usertrap().

- In the system call sys_sigalarm(), the entries alarm_nticks and alarm_handler_addr in the process structure are initialized to the first two arguments respectively passed to the system call. alarm_ticks_passed is initialized to 0, so that it can start counting the number of ticks or timer interrupts that has passed for the process.

- The hardware sends a timer interrupt after every tick. This is identified in kernel/trap.c/usertrap() where we check if an alarm is active for the interrupted process through the absence of invalid value in alarm_nticks. If an alarm is active alarm_ticks_passed is incremented and when it reaches alarm_nticks the handler function is made to be invoked by setting p->trapframe->epc to alarm_handler_addr.

- The system call sys_sigreturn() simply returns 0 as of now.

Register containing user space return address:
The **sepc** register stores the user space address to which the CPU must return after the interrupt handler in kernel exits. In usertrap() this value is accessible through the p->trapframe->epc entry to which content of **sepc** is stored in trampoline.S/uservec() and from which restored in trampoline.S/userret().

(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:
The approach is as follows:

- In order to resume normal program execution after signal handler finishes executing, the trapframe contents is saved to a new entry in struct proc called alarm_tf (type struct trapframe *) just before setting p->trapframe->epc to alarm_handler_addr in usertrap().

- A page is allocated to alarm_tf in kernel/proc.c/allocproc() using kalloc() and the page is de-allocated using kfree() in kernel/proc.c/freeproc().

- In sys_sigreturn() the original trapframe contents from p->alarm_tf are restored to p->trapframe. Thus the process will move to the state it was originally in, just before the signal handler was invoked.

- For avoiding re-entrants to signal handler, an entry handler_in_progress (type int) is added to struct proc. It is a flag which indicates whether a handler is in execution for a process. This bit is set when p->trapframe->epc is set to alarm_handler_addr in usertrap(). It is turned off in sys_sigreturn() since every handler is guaranteed to call sigreturn() system call at the end of execution. Hence when required ticks of another alarm has elapsed, we check that this bit is not set in usertrap() before setting p->trapframe->epc to invoke the handler.

- Finally, the alarm_ticks_passed counter is reset in sys_sigreturn() so that subsequent ticks can be processed and the alarm recurs.

Registers saved:
All registers x_0 to x_31 (in RISC-V) should be saved. This is because the alarm can invoke the handler at any point during the execution of the process and any of these registers might be used during by the process at that time instant. The handler is invoked via usertrap() after a timer interrupt, and timer interrupts can occur at any instant of time.
This is not like a function call, where there is no need to save the caller-saved registers before returning.
Also, we restore the epc in trapframe to the user space return address to which CPU must return after handling the interrupt.

(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.