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 $\cdots$ (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. A user process might want to perform some action at periodic
intervals apart from whatever computation it otherwise performs.

2. A user process could also use the alarm feature to perform an
analysis of the user code i.e. to check which sections of the user
code take how much CPU time.
```

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:

First we make all the necessary changes i.e. the adding the signature of the system calls to defs.h, adding the user stubs etc for both sigalarm(interval, handler) and sigreturn(). The corresponding syscall table entries and declarations also need to be done for these two system calls in syscall.c . We need to define these system call functions in sysproc.c

We also need to add entries to the proc structure to hold the alarm handler and the tick intervals passed to the sigalarm() system call. We also need a entry which will count the number of ticks since the last time the handler was called.
The tick intervals are stored in p->interval, the VA of the handler is stored in p->handler and the number of ticks that have passed since the last call to the handler is stored in the p->ticks_passed.

These entries are initialized in allocproc() to -1, 0 and -1 respectively. We will use the check if(p->interval != -1) as a check for whether this alarm feature has been enabled for that process or not.

In the implementation of the sys_sigalarm() system call we need to do the following.
1. Fill in the proc structure with the arguments passed to the system call.
This can be done using the argint() and argaddr() functions,

```
which are used to fetch the arguments.
2. Set p->ticks_passed = 0.
The code that triggers invocation of the handler when
p->ticks_passed % p->interval = 0 needs to be added in usertrap()
in trap.c.

Here we need to make the following additions
1. There will be an if(which_dev == 2) block in usertrap function
after the function call syscall(). This block yields the CPU on
a timer interrupt which in turn invokes the scheduler for context
switch. Our code needs to be added before this if block.
2. Add an if(which_dev == 2) block which checks if the device
causing the interrupt is a timer. Here we add another sanity
check, i.e. we check
if (p->interval != -1) which as told above, indicates that the
alarm  feature has been enabled for this process. Inside this
if block, we first increment p->ticks_passed, and
if p->ticks_passed % p->interval == 0, then
            set p->tf->epc = p->handler.
This ensures that if the timer interrupt is a critical timer
interrupt (one which causes p->ticks_passed % p->interval == 0),
then  after the trap handling exits and jumps back to user mode,
the user process will resume execution from the handler function.

The register sepc holds the user space address of the instruction
to which the system calls return after being handled.
This register will be indirectly modified above through the means
of the trapframe entry p->tf->epc.
Later,when trampoline.S is invoked to jump back to user space,
the sepc register is filled with this entry.
```

(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:

```
    Here, we need to complete the implementation of the
sys_sigreturn() system call and add additional code and checks in
```

our newly added if block in usertrap().

This is done to ensure that once the user space handler finishes execution and calls sigreturn() which in turn invokes the sys_sigreturn() system call,
we need to somehow reload the context of the original user code which was interrupted when p->ticks_passed % p->interval == 0. For this, we first need to store the context somehow in usertrap(), when the event p->ticks_passed % p->interval == 0 occurred.
We can save this context in the proc structure. The question arises as to how many registers we need to save. As the user process could have been interrupted at any time, and given that we do not have sufficient information if the compiler follows the RISC-V calling conventions,
it would be safe to save all the registers when the critical timer interrupt occurs which invokes the handler. We must therefore add the proc entries required to save all the registers.
So we need to make the following changes to our newly added if(which_dev== 2) block in usertrap().

1. Inside the nested if block which checks if p->ticks_passed % p->interval == 0, we need to add instructions which update the proc entries corresponding to the registers with the user space context just before entering the kernel mode for this interrupt. These registers would have been saved in the trapframe.
So, we need to add lines like p->x = p->trapframe->x for all 32 registers x.
We also need to save the epc register from the trapframe into another entry p->epc in the proc structure. This is required to restart execution from the interrupted user code once the handler returns.

2. We also need to implement the system call function sys_sigreturn() in sysproc.c as follows. First, we need to change the trapframe to ensure that we will be restarting execution from the stored context. For this,we add lines like

p->trapframe->x = p->x

for all 32 registers. And also set p->tf->epc = p->epc. We also need to set p->ticks_passed= 0, this is required to sort of "re-arm" the alarm handler so that it can continue this cycle.
All these changes ensure that after the sys_sigreturn() system call finishes and eventually comes to trampoline.S to jump back to the user mode, it jumps to the interrupted user code with the correct context and can potentially invoke another alarm handler call again.
As test2 requires us to check ensure that we prevent reentrant

```
calls to the handler i.e. ensure that once in the handler,
irrespective of the number of ticks it takes, we do not reinvoke
the handler again until it finishes execution. To do this, we can
add another proc entry called p->handler_visited.
In the sanity check performed earlier, we also add an additional
condition if(p->handler_visited != 1).
Also, we need to set p->handler_visited = 1 inside the
if(p->ticks_passed % p->interval == 0) block and
reset p->handler_visited = 0  in  sys_sigreturn().
```

(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.