CS3500: Operating Systems

Lab 5: Traps and System Calls

September 18, 2020

Introduction

In the previous labs, we became familiar with system calls and traps. We also learnt the paging mechanism in xv6. This lab will put all those pieces together. Firstly, we will look at a debugger called **qemu-gdb** and get some insights into RISC-V assembly. Thereafter, we will design a **tracing and alert mechanism** in xv6.

Resources

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 Avengers, Assemble! (20 points)

For this section, it will be important to understand a bit of RISC-V assembly.

There is a file named `user/call.c` as part of the provided xv6 repo. Modify the `Makefile` suitably to allow `user/call.c` to be compiled as a user program in xv6. Run the command `make fs.img`, which compiles `user/call.c` (among other files) and produces a readable assembly version of the program in `user/call.asm`. Read the assembly code in `user/call.asm` for the functions `g()`, `f()`, and `main()`. Here are some questions that you should answer:

1. (3 points) Which registers contain arguments to functions? For example, which register holds `13` in `main()`'s call to `printf()`?

Solution: In general, arguments to functions are stored in registers x10 to x17 also known as a0 - a7. In this example, a0, a1, a2 are used to pass arguments to `printf()` from main. a2 register holds `13` in `main()`'s call to `printf()`.

2. (2 points) Where is the function call to `f()` from `main()`? Where is the call to `g()`? (**HINT**: the compiler may inline functions.)

Solution: Due to compiler optimization, the call to f() and then the call to g() in f() is replaced directly with the result of the call : 12. Therefore in the asm we are doing li a1,12.

3. (2 points) At what address is the function `printf()` located?

Solution: The function `printf()` is located at 0x5b0.

4. (2 points) What value is in the register `ra` just after the `jalr` to `printf()` in `main()`?

Solution: value in the register `ra` just after the `jalr` to `printf()` in `main()` is current PC + 4 which is 0x38.

5. (11 points) Run the following code.

```
unsigned int i = 0x00646c72;
printf("H%x Wo%s", 57616, &i);
```

(a) (3 points) What is the output? Here's an ASCII table that maps bytes to characters.

Solution: The output is "HE110 World". This is because 57616 in hexadecimal is E110 and i is being printed as a string (char)0x72 = r, (char)0x6c = l, (char)0x64 = d and (char)0x00 is like end of string.

(b) (5 points) The above output depends on that fact that the RISC-V is little-endian. If the RISC-V were instead big-endian, what would you set `i` to in order to yield the same output? Would you need to change 57616 to a different value? Here's a description of little- and big-endian.

Solution: `i` needs to be set to 0x726c6400 in order to yield the same output. Theres no need to change 57616 to a different value as its numeric value is what matters.

(c) (3 points) In the following code, what is going to be printed after `'y='`? (Note: the answer is not a specific value.) Why does this happen?

```
printf("x=%d y=%d", 3);
```

Solution: The value stored in a2 register will be printed and hence a garbage value will be printed.

2 The Retreat (30 points)

When something goes wrong, it is often helpful to look back and see what events led to the current predicament. In debugging terminology, we call this introspection a ***backtrace***. Consider a code that dereferences a null pointer, which means it cannot execute any further due to the resulting kernel panic. While working with xv6, you may have encountered (or will encounter) such panics.

In each stack frame, the compiler puts a frame pointer that holds the address of the caller's frame pointer. We can design a `backtrace()` function using these frame pointers to walk the stack back up and print the saved return address in each stack frame. The GCC compiler, for instance, stores the frame pointer of the currently executing function in the register `s0`.

1. (30 points) In this section, you need to implement `backtrace()`. Feel free to refer to the hints provided at the end of this section.
   (a) (20 points) Implement the `backtrace()` function in `kernel/printf.c`. Insert a call to this function in `sys_sleep()` in `kernel/sysproc.c` just before the `return` statement (you may comment out this line after you are done with this section). There is a user program `user/bttest.c` as part of the provided xv6 repo. Modify the `Makefile` accordingly and then run `bttest`, which calls `sys_sleep()`. Here is a sample output (you may get slightly different addresses):

   ```
   $ bttest
   backtrace:
   0x0000000080002c1a
   0x0000000080002a3e
   0x00000000800026ba
   ```

   What are the steps you followed? What is the output that you got?

   Solution: - Added prototype for backtrace to kernel/defs.h so that it can be called in kernel space.
   - Added a function r_fp() to read frame pointer from s0.
   - Implemented backtrace in kernel/printf.c. Read the frame pointer using r_fp(). As long as the frame pointer is within the stack page and is not the top of the stack page, the return address is printed after obtaining it from fp with an offset of 8. Now the frame pointer is reset to the value stored at an offset of 16 and the loop continues.
   Output:
   backtrace:

```
0x0000000080002b54
0x00000000800029b6
0x00000000800026a0
```

(b) (5 points) Use the `addr2line` utility to verify the lines in code to which these addresses map to. Please mention the command you used along with the output you obtained.

Solution: command used :
riscv64-unknown-elf-addr2line -e kernel/kernel
0x0000000080002b54
0x00000000800029b6
0x00000000800026a0
output obtained:
/home/nischith/Documents/os lab 5/xv6-riscv/
kernel/sysproc.c:74 /home/nischith/Documents/os lab 5/xv6-riscv/
kernel/syscall.c:140 /home/nischith/Documents/os lab 5/xv6-riscv/
kernel/trap.c:76

(c) (5 points) Once your `backtrace()` is working, invoke it from the `panic()` function in `kernel/printf.c`. Add a null pointer dereference statement in the `exec()` function in `kernel/exec.c`, and then check the kernel's backtrace when it panics. What was the output you obtained? What functions/line numbers/file names do these addresses correspond to? (Don't forget to comment out the null pointer dereference statement after you are done with this section.)

Solution:
Output:
0x0000000080000642
0x00000000800027e6
0x0000000080005964
0x00000000800057c2
0x00000000800029be
0x00000000800026a8

functions/line numbers/file names:
kernel/printf.c:125
kernel/trap.c:153 (discriminator 1)
kernel/kernelvec.S:51
kernel/sysfile.c:441
kernel/syscall.c:140
kernel/trap.c:76

Additional hints for implementing `backtrace()`

- Add the prototype `void backtrace(void)` to `kernel/defs.h`.
- Look at the inline assembly functions in `kernel/riscv.h`. Similarly, add your own function, `static inline uint64 r_fp()`, and call this from `backtrace()`

to read the current frame pointer. (**HINT**: The current frame pointer is stored in the register `s0`.)

- Here is a stack diagram for your reference. The current frame pointer is represented by `$fp` and the current stack pointer by `$sp`. Note that the return address and previous frame pointer live at fixed offsets from the current frame pointer. (What are these offsets?) To follow the frame pointers back up the stack, brush up on your knowledge of pointers.

```
                             .
                             .
                             .
        0x2fe0 +-> +-------------------+   |
        0x2fd8 |   | ret addr          |   |
        0x2fd0 |   | 0x2ff8 (prev fp)  ----+
        0x2fc8 |   |       ...         |
        0x2fc0 |   |       ...         |
$fp --> 0x2fb8 |   +-------------------+ <-+
        0x2fb0 |   | ret addr          |   |
$sp --> 0x2fa8 +---- 0x2fe0 (prev fp)  |   |
                   +-------------------+   |
                             .
                             .
                             .
```

- You may face some issues in terminating the backtrace. Note that xv6 allocates one page for each stack in the xv6 kernel at PAGE-aligned address. You can compute the top and bottom address of the stack page by using `PGROUNDUP(fp)` and `PGROUNDDOWN(fp)` (see `kernel/riscv.h`). These are helpful for terminating the loop in your `backtrace()`.

2. (30 points) **[OPTIONAL]** Print the names of the functions and line numbers in `backtrace()` instead of numerical addresses.

3 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:** This will be useful for processes that want to limit the amount of computation they are doing are the amount of CPU time they are hogging. This will also be useful for processes that want to take same action periodically along with doing some other computation.

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.) For the time being, create a simple `sigreturn()` system call with a `return 0;` statement.

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: My approach:
- Added alarmtest.c to the makefile to be compiled as a user program.
- Added user-space stubs for these system calls by adding declarations to sigalarm and sigreturn in user/user.h and adding stubs to user/usys.pl.
- Added signatures of the system calls in kernel/syscall.c and assigned system call numbers in kernel/syscall.h.
- Defined sys_sigreturn to return 0 for now in kernel/sysproc.c.
- Added 3 new fields to struct proc in kernel/proc.h - ticks, interval, handler. ticks is to count the number of ticks passed since last call, the other two are self explanatory.
- Initialized ticks to 0 in allocproc() in kernel/proc.c.
- Defined sys_sigalarm() in kernel/sysproc.c. Here I copy the arguments from user space into processes structure. Also i set interval to -1 if the arguments are both zero, this is just an extra measure to not call program handler.
- In usertrap() in kernel/trap.c I check for timer interrupt (which_dev = 2) and increase ticks accordingly. If ticks is equal to the interval, i reset ticks and set p->tf->epc to p->handler so that when sret is called it jumps to the handler. p->handler was set in sys_sigalarm().
sepc register contains the user-space instruction address to which system calls return.

(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: My approach:
- I added alarmtf in kernel/proc.h.
- For simplicity I am saving the whole trapframe in usertrap() in kernel/trap.c. It is then restored in sys_sigreturn.
- Added sigret a flag to tell if the handler has returned.
- sigret is initialised to 1 in allocproc(). It is set to 0 before setting epc in usertrap() and later set to 1 again in sys_sigreturn.

All registers need to be saved and restored to resume the interrupted code correctly. Since an interrupt can happen anytime we need to save all registers. Also there is no information whether a consistent calling convention is followed by the compiler so it is better to save all the registers.

(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
$
```

3.1 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.