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: The arguments for the function calls are passed through the RISC-V argument registers i.e. `a0`, `a1`, `a2`, `a3` etc respectively for the first, second, third arguments and so on. For example, in the case of `main()`'s call to `printf()`, as `13` is the 3rd argument passed to `printf()`, it is passed through the `a2` register. This is also evident from the line

```
li a2,13
```

in the disassembled code.

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: The compiler has performed aggressive inlining optimizations, so there are no explicit calls to `g()` and `f()`.
These are evident from the lines

```
li a1,12
```

In the above line, the compiler has directly loaded the second argument register with the result of the computation `f(8) + 1` without any explicit calls.

```
addiw a0,a0,3
```

Similarly, here for the `return g(x)` statement's conversion, no calls to `g()` are performed, instead, the `a0` register is incremented by `3` directly which would have been the end result of these explicit calls (if they had been implemented naively) as `a0` acts as both the return value and first argument register.

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

Solution: A simple search of the `call.asm` disassembly shows that the `printf()` function is located at the virtual address `0x00000000000005b0`.

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

Solution: The `jalr` or `jump-and-link-register` instructions sets `ra` with `PC + 4` (the next instruction after the `jalr` instruction), as it is usually required for returns after a function call. So the expected value in `ra` is the address of the next line i.e. `0x0000000000000038`.

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:

`HE110 World`

The above line is the output when run on the RISC-V compiler used by xv6. Although the `gcc` compiler has a difference based on whether `%x/%X` are used and correspondingly either `e/E` is printed.
The explanation is as follows.
`%x` format specifier is used for hexadecimals and `57616` in hexadecimal is `0xE110`. This explains the `HE110`.
`%s` format specifier prints the null terminated string starting from the address in the corresponding `printf()` argument. In this case, that address is the address of `unsigned int i` which causes `printf()` to print the string corresponding to the value of `unsigned int i`. Note that as RISC-V hardware encoding is little-endian, the hexadecimal is stored in hardware as

`&i -> 72 | 6c | 64 | 00`

where the split refers to each byte and each byte contains the character's ASCII encoding. From the ASCII table, we see that

`72` $\rightarrow$ `r`
`6c` $\rightarrow$ `l`
`64` $\rightarrow$ `d`
`00` $\rightarrow$ `NULL`

Therefore, this explains the `World`.

(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: If RISC-V were to follow the big-endian system, we could follow a left-to-right byte pattern for the string. Concretely, `i` would now have to be

`unsigned int i = 0x726c6400`

There would be no need to change `57616` to a different value as the `%x` format specifier does a direct conversion of the corresponding `printf()` argument to hexadecimal and prints it as such.

(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: `printf()` of course prints the entire string with the respective format specifiers filled in, but the value of the corresponding argument (non-existant in the C code) used depends on the run-time value in the correspond-

ing argument register which in this case is the value in `a2` at runtime.

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: The below is the output that I got on running the `bttest` user program

```
backtrace:
0x0000000080002c54
0x0000000080002ab6
0x00000000800026a2
```

The steps I followed were as follows. I initially read the current value of the frame pointer from the `s0` register through an inline assembly function. Then, I had a while loop which checked whether the running frame pointer variable was within bounds of the original page by using `PGROUNDDOWN` and `PGROUNDUP`. In the while loop, we first obtain the return address which is at an 8 byte offset from the frame pointer and print it. Finally, we can update the frame pointer with the previous frame pointer which is present at a 16

byte offset.
The C code for backtrace that I implemented is as follows,

```
void
backtrace(void)
{
  uint64 ifp;
  ifp = r_fp();
  uint64 upperLimit = PGROUNDUP(ifp);
  uint64 lowerLimit = PGROUNDDOWN(ifp);
  uint64 fp = ifp;
  uint64 ra;
  printf("backtrace:\n");
  while(fp >= lowerLimit && fp < upperLimit)
  {
    ra = *((uint64*)fp - 1);
    printf("%p\n", ra);
    fp = *((uint64*)fp - 2);
  }
}
```

Then, we call `backtrace()` from `sys_sleep()`. We also need to make a couple of cosmetic changes such as adding the signature for `backtrace()` in `defs.h` etc.

(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: The `addr2line` command was used in the `path_to_xv6-riscv/kernel` directory as follows,

```
$ addr2line -e kernel <hex-address>
```

The following input output pairs were obtained

```
$ addr2line -e kernel 0x0000000080002c54
/path_to_xv6-riscv/kernel/sysproc.c:74
```

The above output corresponded to the line just after the call to `backtrace()` in the body of the `sys_sleep` function which is correct as it is the return address after the most recent kernel function call i.e. to `backtrace()` .

```
addr2line -e kernel 0x0000000080002ab6
/path_to_xv6-riscv/kernel/syscall.c:144
```

The above output corresponded to the line `p->tf->a0 = syscalls[num]();` in the body of the `syscall` function. This is correct as it's the return address after the function call to the `sys_sleep` function.

```
$ addr2line -e kernel 0x00000000800026a2
/path_to_xv6-riscv/kernel/trap.c:76
```

The above output corresponded to the line just after returning from the function call to `syscall()` in the body of function `usertrap()`. This is also the last function call return in the kernel and therefore no more return

addresses are printed.
Therefore, all three outputs are justified.

(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: The null pointer dereference was added just before the call to `beginop()` in `exec.c` and the `backtrace()` call was added just before the `printf("panic: ");` line in the body of the `panic()` function. This generated the following backtrace output

```
backtrace:
0x0000000080000608
0x00000000800028e6
0x0000000080005be4
0x0000000080005a44
0x0000000080002abe
0x00000000800026aa
```

These corresponded to the following output on using the `addr2line` utility.

```
/path_to_xv6-riscv/kernel/printf.c:122
/path_to_xv6-riscv/kernel/trap.c:198 (discriminator 1)
??:?
/path_to_xv6-riscv/kernel/sysfile.c:441
/path_to_xv6-riscv/kernel/syscall.c:144
/path_to_xv6-riscv/kernel/trap.c:76
```

These return addresses correspond to the following kernel code instructions.

1. `/path_to_xv6-riscv/kernel/printf.c:122` corresponds to the line after the `backtrace()` call in the body of the `panic()` function.

2. `/path_to_xv6-riscv/kernel/trap.c:198 (discriminator 1)` corresponds to the line after the `panic("kerneltrap")` function call in the body of the `kerneltrap()` function.

3. This address is not printed as this VA corresponds to the kernel code written in the RISC-V assembly for which `addr2line` does not have any support. We can manually search for this line in the address annotated disassembly `kernel.asm`. This search shows that this address corresponds to the next instruction after the `jal ra,8000283e <kerneltrap>` instruction in `kernelvec` in `kernelvec.S`, which jumps to the kernel trap handler `kerneltrap()` in `trap.c` after saving all the regisers.

4. `/path_to_xv6-riscv/kernel/sysfile.c:441` corresponds to the line `int ret = exec(path, argv)` in the body of the `sys_exec()` function.

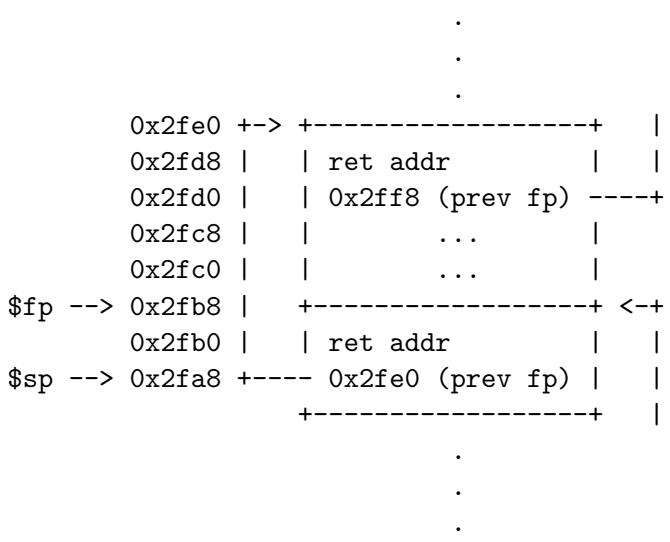
5. `/path_to_xv6-riscv/kernel/syscall.c:144` corresponds to the line `p->tf->a0 = syscalls[num]();` in the body of the `syscall()` function.

6. `/path_to_xv6-riscv/kernel/trap.c:76` corresponds to the line after the `syscalls()` function call in the body of the `usertrap()` function.

As this is the first function call once in the kernel mode with the kernel page tables, no other higher stack frames are available.

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.



- 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:

   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.) 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: First we make all the necessary cosmetic 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` similar to Assignment 1. We then 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. In my implementation, the tick intervals are stored in `p->ticks`, 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->ticks != -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` == `p->ticks_passed` 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 a 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->ticks != -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->ticks`, then set `p->tf->epc = p->handler`. This ensures that if the timer interrupt is a crictical timer interrupt (i.e. one which causes `p->ticks_passed == p->ticks`), 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 the user space, the `sepc` register is filled with this entry.
Please refer to the actual code in the `xv6-riscv` repo attached with the submission.

(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->ticks`.
For this, we first need to store the context somehow in `usertrap()`, when the event `p->ticks_passed == p->ticks` 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->ticks`, 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->inhandler`. In the sanity check performed earlier, we also add an additional condition to make it `if(p->ticks != -1 && p->inhandler != 1)`. Also, we need to set `p->inhandler = 1` inside the `if(p->ticks == p->ticks_passed)` block and reset `p->inhandler = 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
$
```

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.