CS3500: Operating Systems

Lab 4: Stacks and the Kernel Context Calls

September 17, 2021

Introduction

In the previous labs, we became familiar with system calls. We also learnt the paging mechanism in xv6. This lab will look into the stack management in a process and the kernel's context. Firstly, we will look at a debugger called **qemu-gdb** and get some insights into RISC-V assembly. Thereafter, we will introduce a system call to print the kernel state of a process 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:

Registers a0-a7(x10-x17) contain arguments to functions.
Register a2(x12) holds 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:

There is no explicit call to `f()` from `main()` due to optimisation by compiler. value of `f()+3` is directly computed and stored in a1 register.
Similarly for `g()`. Its definition is directly appplied to the function `f()`. So no explicit call.

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

Solution:

00000000000005b0

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

Solution:

ra will have value 0000000000000038. After jalr instruction, the address for next instruction will be stored as returning address.

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:

   Output - "HE110 World"
   57616 in hexadecimal is "E110"
   0x00646c72 written as string refering to ASCII table - "rld"

   (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 the RISC-V were instead big-endian, we would have to set is as : 0x726c6400 in order to yield same output. We will not have to change 57616 to a different value.

(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:
A random number will be printed. This happens as if we dont give the argument, it will print the random number that is there on the stack

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:
Steps followed:

1. Added r_fp in riscv.h
2. Added prototype for backtrace function in defs.h
3. Added function backtrace in printf.c
4. Added backtrace function call in sys_sleep syscall
5. Added bttest in Makefile

Output :

```
$ bttest
backtrace:
0x0000000080002b60
0x00000000800029c2
0x00000000800026ac
```

(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:

```
$ riscv64-unknown-elf-addr2line -e kernel/kernel
0x0000000080002b60
0x00000000800029c2
0x00000000800026ac
/home/ds/Desktop/CS3500-Lab/xv6-riscv/kernel/sysproc.c:76
/home/ds/Desktop/CS3500-Lab/xv6-riscv/kernel/syscall.c:140
/home/ds/Desktop/CS3500-Lab/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:

```
$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel
kernel/kernel -m 3G -smp 3 -nographic -drive
file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,
drive=x0,bus=virtio-mmio-bus.0
```

```
xv6 kernel is booting

hart 2 starting
scause 0x000000000000000d
sepc=0x00000000800049c2 stval=0x0000000000000000
hart 1 starting
backtrace:
0x0000000080000614
0x00000000800027ea
0x0000000080005974
0x00000000800057d2
0x00000000800029c2
0x00000000800026ac
panic: kerneltrap
QEMU: Terminated
**********************************
```

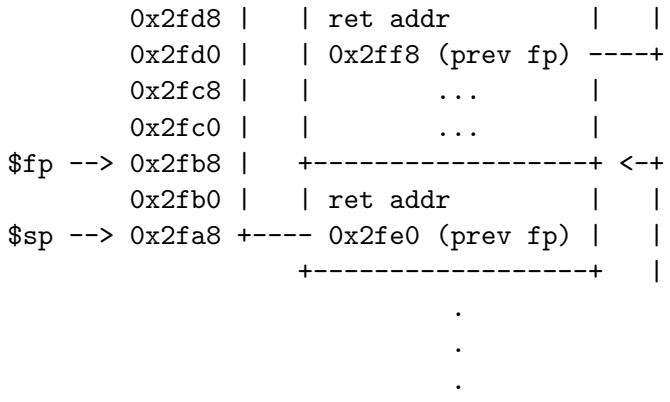
Functions/line numbers/file names these addresses correspond to:

```
$ riscv64-unknown-elf-addr2line -e kernel/kernel
0x0000000080000614
0x00000000800027ea
0x0000000080005974
0x00000000800057d2
0x00000000800029c2
0x00000000800026ac
/home/ds/Desktop/CS3500-Lab/xv6-riscv/kernel/printf.c:121
/home/ds/Desktop/CS3500-Lab/xv6-riscv/kernel/trap.c:153 (discriminator 1)
??:?
/home/ds/Desktop/CS3500-Lab/xv6-riscv/kernel/sysfile.c:441
/home/ds/Desktop/CS3500-Lab/xv6-riscv/kernel/syscall.c:140
/home/ds/Desktop/CS3500-Lab/xv6-riscv/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 +-> +------------------+   |
```



- 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 The Attack . . . (20 points)

A process not just has its own virtual address space but, it also has metadata in the kernel. In this part we will try to understand the contents of these metadata.

1. (5 points) Every process is allocated a Process Control Block entry into the `proc` structure. Introduce a system call `pcbread` to print the contents of the `proc` structure.

   Write a user program `user/attack.c` (similar to question 1). Use this program to invoke and test `pcbread`.

   What is the PID of the process?

   Solution:

   3

2. (5 points) Fork a child process in `attack.c`. Use your system call to find the similarities and differences between the parent and child's PCB. List those differences here.

   Solution:

   1. pid
   2. kstack
   3. pagetable

4. trapframe
5. context
6. file
7. lock
8. state
9. parent

3. (5 points) Just before `usertrapret` returns, print the contents of the trapframe in the parent and child process in `attack.c`. This printing should be done only for the `fork` system call and at no other time. How are the trapframes different?

Solution:

kernel_sp, kernel_hartid, a0 are different in the trapframes of child and parent

4. (5 points) Print the contents of the `a0` to `a6` registers from the trapframe. Compare the contents of these registers with system call arguments passed from the `attack.c`. Test with several different system calls. List your observations here.

Solution:

Registers are inintialised with a value initialy. If the value of registered is not required in next syscall, then the value will remain unchanged. System call Arguements arre passed through these.

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.