# 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:**

In xv6 the registers a0-7 are responsible for passing arguments to functions any arguments over and above 8 are passed through stack. For floating point numbers registers fa0-7 are responsible.

In this particular .asm file, the arguemtn 13 is passed through register *a2* in mains call to printf. The value of **f(8) + 1** is passed through *a1*

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 function call to **g()** in **f()**. The compiler copied the function definition of **g()** directly into **f()** since **g()** is a really small function. This is an optimization the compiler does to save the overhead of creating a stack frame and passing arguments.

Also, the compiler evaluated **f(8)** when it is compiling the code. The compiler does simple calculations like these to save runtime overhead.

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

**Solution:**

The function `printf()` is located at 00000000000005b0

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

**Solution:**

The Register `ra` contains the address of the next instruction after the `jalr` instrcution to *printf*. In this case it is 38

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 of the given code is **HE110 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:**
> For a big-endian based system to get the same output, the value of `i` to be used is `0x726c6400`. The value `57616` need not be changed

(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:**
> Observe that in the `.asm` file there is an instrction
>
> `li a1, 3`
>
> This instrcution loads `a1` register with the value of 3. But when printf function is called. It expects two arguemnts. So, it takes whatever value there is in `a2` register.

## 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 a function *r_fp* to *defs.h* file that returns the frame pointer of calling function
- Call *r_fp* from **backtrace()** function.
- The return address is stored at an offset of 8 from the *fp*. Print it
- The previous *fp* is stored at an offset of 16 from the *fp* load it into the variable.
- continue the process until fp reaches the page boundary. This can be checked by using *PGROUNDOWM* and *PGROUNDUP*

  **output** backtrace:
  0x0000000080002b44
  0x00000000800029a6
  0x0000000080002690

(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:** addr2line -e kernel 0x000000080002b44 0x0000000800029a6 0x000000080002690
**Output:**
*/home/joel/Documents/xv6-riscv-Lab4/xv6-riscv/kernel/sysproc.c:75*
*/home/joel/Documents/xv6-riscv-Lab4/xv6-riscv/kernel/syscall.c:140*
*/home/joel/Documents/xv6-riscv-Lab4/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:**
0x0000000080000632
0x00000000800027d6
0x0000000080005964
0x00000000800057be

0x00000000800029ae
0x0000000080002698
Again use the *addr2line* line command to get the names and linenumbers of the traceback.

Doing so gives back the following output

```
addr2line -e kernel 0x0000000080000632 0x00000000800027d6
0x0000000080005964 0x00000000800057be
0x00000000800029ae 0x0000000080002698
xv6-riscv-Lab4/xv6-riscv/kernel/printf.c:126
xv6-riscv-Lab4/xv6-riscv/kernel/trap.c:153 (discriminator 1)
??:?
xv6-riscv-Lab4/xv6-riscv/kernel/sysfile.c:441
xv6-riscv-Lab4/xv6-riscv/kernel/syscall.c:140
xv6-riscv-Lab4/xv6-riscv/kernel/trap.c:76
```

As can be seen, when we tried to dereference a *NULL* pointer from exec. The first process will throw an exception. So, the execution is transferred to *trap.c* file which corresponds to **kerneltrap()** function. Which calls a **sys_exec()** function in *sysfile.c* file. Which then systemcalls and it finally again goes to **trap.c**

## 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 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:**

   The PID of the process is **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:**

   They have differnt *PIDs*. Child has a *PID* one more than the Parent. They have same size and name. But they have differnt Trapframes, contexts, kernel stacks

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:**

   The child and the parent have different kernel_sp, that is they have differnt kernel stacks. They have differnt *hart_id* because they run on differnt cpus. Apart from that all the registers have the same values. So that the execution of child and parent resume with the same values in registers. The value of a0 also differs because in child process the fork returns 0 while in parent it returns the *pid* of the process. So a0 also differs.

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:** *a0* to *a6* registers are used to pass the arguements to system call. If *sleep(10)* is used then the argument *10* is passed through *a0*.

## 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.