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:

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 00000000000005c0.

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 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 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 corresponding argument register which in this case is the value in a2 at run-time.

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:
0x0000000080002fd6
0x0000000080002e38
0x0000000080002b22
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 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 0x0000000080002fd6
/xv6-riscv/kernel/sysproc.c:102
```

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 0x0000000080002e38
/xv6-riscv/kernel/syscall.c:145
```

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 0x0000000080002b22
/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

scause 0x000000000000000d
sepc=0x000000008000513e stval=0x0000000000000001
backtrace:
0x0000000080000608
0x0000000080002c68
0x00000000800060e4
0x0000000080005f44
0x0000000080002e40
0x0000000080002b2a
panic: kerneltrap

These corresponded to the following output on using the addr2line
utility.
/xv6-riscv/kernel/printf.c:122
/xv6-riscv/kernel/trap.c:204 (discriminator 1)
??:?
/xv6-riscv/kernel/sysfile.c:455
/xv6-riscv/kernel/syscall.c:145
/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:204 (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 jalra,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:455 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:145 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.

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.

```
                             .
                             .
                             .
        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:

```
PID of the process is 3 which is present in proc structure of
the process(attack.c).
```

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:

```
This is the output by running the attack.c:-

Process ID: 4
Name of the Process: attack
killed: 0
xstate: 0
TrapFrame: 0x0000000087f49000
Current directory address: 0x000000008001ff08
Parent proc structure: 0x0000000080011fd0
size of the Process: 12288
Bottom of kernel stack: -36864
PageTable address: 0x0000000087f75000
Context of the process: 0x0000003fffff7f30
```

```
Process ID: 3
Name of the Process: attack
killed: 0
xstate: 0
TrapFrame: 0x0000000087f64000
Current directory address: 0x000000008001ff08
Parent proc structure: 0x0000000080011e68
size of the Process: 12288
Bottom of kernel stack: -28672
PageTable address: 0x0000000087f48000
Context of the process: 0x0000003fffff9f30

Similarities between parent and child’s PCB:-

1)  Name of the process
2)  killed(If non-zero, have been killed)
3)  xstate
4)  Current directory
5)  Size of the program

Differences between parent and child’s PCB:-

1) Trapframe (Each process has it’s own trapframe).
2) Parent reference (child process has parent process as parent)
3) Bottom of kernel stack
4) PageTable address (Each process has it’s own pagetable directory).
5) Context of the process
```

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:

```
output obtained by running the program:-
process ID: 3
process name: attack
contents of register a0 : 4
contents of register a1 : 12256
contents of register a2 : 5103
contents of register a3 : 16032
contents of register a4 : 4992
contents of register a5 : 238
contents of register a6 : 84215045
contents of register a7 : 1
contents of register s0 : 12256
contents of register s1 : 81744
contents of register s2 : 99
contents of register s3 : 32
contents of register s4 : 5099
contents of register s5 : 4976
contents of register s6 : 84215045
contents of register s7 : 84215045
contents of register s8 : 84215045
contents of register s9 : 84215045
contents of register s10 : 84215045
contents of register s11 : 84215045
contents of register t0 : 84215045
contents of register t1 : 84215045
contents of register t2 : 84215045
contents of register t3 : 84215045
contents of register t4 : 84215045
contents of register t5 : 84215045
contents of register t6 : 84215045
kernel page table: 557055
Top of process's kernel stack: -24576
usertrap() function address: -2147472780
user program counter: 584
kernel tp: 2
ra: 44
sp: 12240
gp: 84215045
tp: 84215045
process ID: 4
process name: attack
contents of register a0 : 0
contents of register a1 : 12256
contents of register a2 : 5103
contents of register a3 : 16032
contents of register a4 : 4992
contents of register a5 : 238
contents of register a6 : 84215045
contents of register a7 : 1
contents of register s0 : 12256
contents of register s1 : 81744
contents of register s2 : 99
contents of register s3 : 32
contents of register s4 : 5099
contents of register s5 : 4976
contents of register s6 : 84215045
contents of register s7 : 84215045
contents of register s8 : 84215045
contents of register s9 : 84215045
contents of register s10 : 84215045
contents of register s11 : 84215045
contents of register t0 : 84215045
contents of register t1 : 84215045
contents of register t2 : 84215045
contents of register t3 : 84215045
contents of register t4 : 84215045
contents of register t5 : 84215045
contents of register t6 : 84215045
kernel page table: 557055
Top of process's kernel stack: -32768
usertrap() function address: -2147472780
user program counter: 584
kernel tp: 1
ra: 44
sp: 12240
gp: 84215045
tp: 84215045
Process ID: 4
Name of the Process: attack
killed: 0
xstate: 0
TrapFrame: 0x0000000087f49000
Current directory address: 0x000000008001ff08
Parent proc structure: 0x0000000080011fd0
size of the Process: 12288
Bottom of kernel stack: -36864
PageTable address: 0x0000000087f75000
Context of the process: 0x0000003fffff7f40

Process ID: 3
Name of the Process: attack
killed: 0
xstate: 0
TrapFrame: 0x0000000087f64000
Current directory address: 0x000000008001ff08
Parent proc structure: 0x0000000080011e68
size of the Process: 12288
Bottom of kernel stack: -28672
PageTable address: 0x0000000087f48000
Context of the process: 0x0000003fffff9f40

Differences between the trapeframes:-
1) a0
2) kernel_sp
3) kernel_hartid
4) pid of the process
```

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:

```
Arguments are passed through a0 to a7 registers in RISCV model.

sleep system call:-
process ID: 3
contents of register a0 : 10
contents of register a1 : 12256
contents of register a2 : 5103
contents of register a3 : 16032
contents of register a4 : 4992
contents of register a5 : 238
contents of register a6 : 84215045
arguments of sleep system call:- 10
Explanation:-
since this system call contains only one argument it
passes through the a0.
 a0 contains 10 which matches with input argument.

exit system call:-
process ID: 3
contents of register a0 : 0
contents of register a1 : 12256
contents of register a2 : 5103
contents of register a3 : 16032
contents of register a4 : 4992
contents of register a5 : 238
contents of register a6 : 84215045
arguments of exit system call:- 0
Explanation:-
since this system call contains only one argument it
passes through the a0.
 a0 contains 0 which matches with input argument.

testing system call:-

process ID: 3
contents of register a0 : 1
contents of register a1 : 2
contents of register a2 : 3
contents of register a3 : 4
contents of register a4 : 5
contents of register a5 : 238
contents of register a6 : 84215045
```

```
arguments of testing system call:- 1 2 3 4 5

since system call has five arguments it passes through a0,a1,a2,a3,
a4,a5.

     a0 = 1
     a1 = 2
     a2 = 3
     a3 = 4
     a4 = 5.
 Therefore arguments are passed through registers from above results.
```

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.