CS3500: Operating Systems

Lab 4: Stacks and the Kernel Context Calls

Abishek S (EE18B001)

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:
Note: For this question alone, I changed *-O* flag to *-O0* flag in Makefile when compiling .c programs to get the registers containing the arguments for function call `f(8)`. Otherwise the compiler optimizes it and replaces entire `f()` call by the result.
Function calls with arguments :

- `f(8)`
  - *a0* - 8
- `g(x)`
  - *a0* - x
- `printf("%d %d\n", f(8)+1, 13)`
  - *a0* - `char*` pointer to first character of `"%d %d\n"`
  - *a1* - 12 (`f(8) + 1`)
  - *a2* - 13

Without the *-O0* flag, the `printf(..)` and `g(x)` alone will be evident from the *.asm* file.
This is consistent with the RISC-V calling convention that uses register *a0* - *a7* as function arguments.

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:
```
26:   45b1                    li   a1,12
```
The above line is where the call to `f(8)` is hidden in `main()` (it is inlined by the compiler - the code is so much optimized that the entire expression `f(8)+1` is replaced by the result `g(8)+1` $= 8 + 3 + 1 = 12$).

```
14:   250d                    addiw   a0,a0,3
```
The above line is where the call to `g(x)` is inlined in `f(x)` - optimised by compiler to return `x+3` directly.

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

Solution:
```
0000000000000628 <printf>:
```
The above line indicates `printf()` is located at address `0x628`.

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

> **Solution:**
> `34:	5f8080e7	jalr	1528(ra) # 628 <printf>`
> `jalr` instruction stores the `PC` (before jumping) `+ 0x4` in ra and jumps, so the value in register `ra` just after the `jalr` will be `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:**
> **Output:**
> `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:**
> I would set: `unsigned int i = 0x726c6400` in a big endian system to yield the same output, since `%s` specifier will typecast the pointer to `char*` and read 1 byte at a time, starting from the most significant byte in case of big-endian, and terminate at null (`0x00`).
>
> `%x` specifier will read the whole integer (57616) and then convert it to a hexadecimal value for printing, so changing endian-ness won't affect this, hence we need not change 57616.

(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 junk value will be printed after '`y=`'.
>
> *a0* will contain the `char*` pointer for the format string `"x=%d y = %d"`.
> *a1* will contain the value 3.
> Now, when the first `%d` is encountered (for `x`), `va_arg(fmt, int)` will read the first variadic argument (3) from register *a1* according to the calling convention. When the second `%d` is encountered (for `y`), the `va_arg(fmt, int)` will look at register *a2* and will print it, and it will be a junk value.

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:

   - Following the hints, `void backtrace(void);` declaration is added to `kernel/defs.c`.
   - `r_fp()` is added to `kernel/riscv.h` to read content of *s0* register.
   - In `kernel/printf.c`, `backtrace()` is defined as follows:
     • Start from current frame pointer obtained using `r_fp()`.
     • Set `stack_top` to `PGROUNDUP(current frame pointer)`.
     • Looping till the current frame pointer hits the `stack_top`:
       – print the return address from content of memory location 8 bytes below current frame pointer
       – set current frame pointer to content of memory location 16 bytes below current frame pointer
   - Added the `backtrace()` call in `kernel/sysproc.c/sys_sleep()` just before `return` statement.

Output:

```
$ bttest
backtrace:
0x0000000080002cd6
0x0000000080002b48
0x0000000080002832
```

(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 commands along with the outputs:

```
$ riscv64-unknown-elf-addr2line -e kernel/kernel 0x0000000080002cd6
/Users/abishek_programming/xv6-riscv/kernel/sysproc.c:74
```

The Line: `release(&tickslock);`

```
$ riscv64-unknown-elf-addr2line -e kernel/kernel 0x0000000080002b48
/Users/abishek_programming/xv6-riscv/kernel/syscall.c:140 (
    discriminator 1)
```

The Line: `p->trapframe->a0 = syscalls[num]();`

```
$ riscv64-unknown-elf-addr2line -e kernel/kernel 0x0000000080002832
/Users/abishek_programming/xv6-riscv/kernel/trap.c:76
```

The Line: `if(p->killed)`

(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:

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
scause 0x000000000000000d
sepc=0x0000000080004b44 stval=0x0000000000000000
panic: kerneltrap
backtrace:
0x00000000800005fe
0x0000000080002970
0x0000000080005b44
```

```
0x00000000800059b2
0x0000000080002b48
0x0000000080002832
```

The functions/line numbers/file names:

- `0x00000000800005fe` -
  `/Users/abishek_programming/xv6-riscv/kernel/printf.c:125`
  The line: `panicked = 1; // freeze uart output from other CPUs`
- `0x0000000080002970` -
  `/Users/abishek_programming/xv6-riscv/kernel/trap.c:153`
  `(discriminator 1)`
  The line: `if(which_dev == 2 && myproc() != 0 &&`
  `myproc()->state == RUNNING)`
- `0x0000000080005b44` -
  `/Users/abishek_programming/xv6-riscv/kernel/kernelvec.S:51`
  The line: `ld ra, 0(sp)` (the line after call to `kerneltrap()`)
- `0x00000000800059b2` -
  `/Users/abishek_programming/xv6-riscv/kernel/sysfile.c:444`
  The line: `int ret = exec(path, argv);`
- `0x0000000080002b48` -
  `/Users/abishek_programming/xv6-riscv/kernel/syscall.c:140`
  The line: `p->trapframe->a0 = syscalls[num]();`
- `0x0000000080002832` -
  `/Users/abishek_programming/xv6-riscv/kernel/trap.c:76`
  The line: `if(p->killed)`

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: 3
Output of pcbread system call:

```
$ attack
----------- PROCESS CONTROL BLOCK -----------
-- PID: 3
-- Name: attack
-- Process state: RUNNING (4)
-- Killed ?: no
-- Exit status: 0
-- Parent PID: 2 | Name: sh
-- kstack virtual address: 0x0000003fffff9000
-- Size: 12288 Bytes
-- Pagetable base address: 0x0000000087f49000
-- Context base address: 0x0000000080011a00
-- Trapframe base address: 0x0000000087f65000
-- cwd inode address: 0x000000008001f7e0
-- Open files struct address:
-- -- 0x00000000800213d0
-- -- 0x00000000800213d0
```

```
-- -- 0x00000000800213d0
-- Lock address: 0x0000000080011b20
-- chan: 0x0000000000000000
------------------ END OF PCB ----------------
```

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:
Output:

```
$ attack

FROM CHILD :
------------ PROCESS CONTROL BLOCK -----------
-- PID: 4
-- Name: attack
-- Process state: RUNNING (4)
-- Killed ?: no
-- Exit status: 0
-- Parent PID: 3 | Name: attack
-- kstack virtual address: 0x0000003fffff7000
-- Size: 12288 Bytes
-- Pagetable base address: 0x0000000087f76000
-- Context base address: 0x0000000080011b68
-- Trapframe base address: 0x0000000087f4a000
-- cwd inode address: 0x000000008001f7e0
-- Open files struct address:
-- -- 0x00000000800213d0
-- -- 0x00000000800213d0
-- -- 0x00000000800213d0
-- Lock address: 0x0000000080011b20
-- chan: 0x0000000000000000
------------------ END OF PCB ----------------

FROM PARENT :
------------ PROCESS CONTROL BLOCK -----------
-- PID: 3
-- Name: attack
-- Process state: RUNNING (4)
-- Killed ?: no
-- Exit status: 0
-- Parent PID: 2 | Name: sh
-- kstack virtual address: 0x0000003fffff9000
-- Size: 12288 Bytes
-- Pagetable base address: 0x0000000087f49000
-- Context base address: 0x0000000080011a00
-- Trapframe base address: 0x0000000087f65000
-- cwd inode address: 0x000000008001f7e0
-- Open files struct address:
-- -- 0x00000000800213d0
```

```
-- -- 0x00000000800213d0
-- -- 0x00000000800213d0
-- Lock address: 0x00000000800119b0
-- chan: 0x0000000000000000
----------------- END OF PCB ---------------
```

The observations are as follows:

- The PID of parent and children are different and infact PID of forked child is 1 more than that of parent which is how it is implemented in this version of xv6-riscv.
- The process name used for debugging purpose and size of memory occupied is same for both child and parent since child is an exact replica of the parent process.
- Since a forked process has its own pagetable and trapframe, the pagetable and trapframe addresses are different in child and parent.
- Each process in the process queue has its own kernel stack and so the kstack address is different in child and parent.
- Each struct process has it's own context struct, hence the address of the context of the parent and child is different.
- Since the child is an exact replica of parent process and is formed from the same underlying program, the current directory inode address (it is copied from parent to child after increasing reference count in `fork()`) and the open files' address are same in the parent and child.
- Each process has its own lock in the `struct proc`, hence child and parent has different address of the lock.
- Both the child and parent have `0x0` for `void* chan` since they are not sleeping on chan.

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:

```
$ attack
---------- PARENT PROCESS TRAPFRAME ---------
-- kernel_satp: 0x8000000000087fff
-- kernel_sp: 0x0000003fffffa000
-- kernel_trap: 0x0000000080002b5e
-- epc: 0x00000000000002ee
-- kernel_hartid: 0
-- ra: 0x0000000000000010
```

```
-- sp: 0x0000000000002fd0
-- gp: 0x0505050505050505
-- tp: 0x0505050505050505
-- t0: 0x0505050505050505
-- t1: 0x0505050505050505
-- t2: 0x0505050505050505
-- s0: 0x0000000000002fe0
-- a0: 0x0000000000000004
-- a1: 0x0000000000002fe0
-- a2: 0x00000000000014bf
-- a3: 0x0000000000003ea0
-- a4: 0x00000000000013f8
-- a5: 0x00000000000000ee
-- a6: 0x0000100000000000
-- a7: 0x0000000000000001
-- s2: 0x0000000000000063
-- s3: 0x0000000000000020
-- s4: 0x00000000000014bb
-- s5: 0x00000000000013e8
-- s6: 0x0505050505050505
-- s7: 0x0505050505050505
-- s8: 0x0505050505050505
-- s9: 0x0505050505050505
-- s10: 0x0505050505050505
-- s11: 0x0505050505050505
-- t3: 0x0505050505050505
-- t4: 0x0505050505050505
-- t5: 0x0505050505050505
-- t6: 0x0505050505050505
---------- END OF PARENT TRAPFRAME ----------

---------- CHILD PROCESS TRAPFRAME ----------
-- kernel_satp: 0x8000000000087fff
-- kernel_sp: 0x0000003fffff8000
-- kernel_trap: 0x0000000080002b5e
-- epc: 0x00000000000002ee
-- kernel_hartid: 1
-- ra: 0x0000000000000010
-- sp: 0x0000000000002fd0
-- gp: 0x0505050505050505
-- tp: 0x0505050505050505
-- t0: 0x0505050505050505
-- t1: 0x0505050505050505
-- t2: 0x0505050505050505
-- s0: 0x0000000000002fe0
-- a0: 0x0000000000000000
-- a1: 0x0000000000002fe0
-- a2: 0x00000000000014bf
-- a3: 0x0000000000003ea0
-- a4: 0x00000000000013f8
-- a5: 0x00000000000000ee
-- a6: 0x0000100000000000
-- a7: 0x0000000000000001
-- s2: 0x0000000000000063
```

```
-- s3: 0x0000000000000020
-- s4: 0x00000000000014bb
-- s5: 0x00000000000013e8
-- s6: 0x0505050505050505
-- s7: 0x0505050505050505
-- s8: 0x0505050505050505
-- s9: 0x0505050505050505
-- s10: 0x0505050505050505
-- s11: 0x0505050505050505
-- t3: 0x0505050505050505
-- t4: 0x0505050505050505
-- t5: 0x0505050505050505
-- t6: 0x0505050505050505
----------- END OF CHILD TRAPFRAME ----------

FROM CHILD :
------------ PROCESS CONTROL BLOCK -----------
-- PID: 4
-- Name: attack
-- Process state: RUNNING (4)
-- Killed ?: no
-- Exit status: 0
-- Parent PID: 3 | Name: attack
-- kstack virtual address: 0x0000003fffff7000
-- Size: 12288 Bytes
-- Pagetable base address: 0x0000000087f76000
-- Context base address: 0x0000000080011b80
-- Trapframe base address: 0x0000000087f4a000
-- cwd inode address: 0x000000008001f9e0
-- Open files struct address:
-- -- 0x00000000800215d0
-- -- 0x00000000800215d0
-- -- 0x00000000800215d0
------------------ END OF PCB ----------------

FROM PARENT :
------------ PROCESS CONTROL BLOCK -----------
-- PID: 3
-- Name: attack
-- Process state: RUNNING (4)
-- Killed ?: no
-- Exit status: 0
-- Parent PID: 2 | Name: sh
-- kstack virtual address: 0x0000003fffff9000
-- Size: 12288 Bytes
-- Pagetable base address: 0x0000000087f49000
-- Context base address: 0x0000000080011a10
-- Trapframe base address: 0x0000000087f65000
-- cwd inode address: 0x000000008001f9e0
-- Open files struct address:
-- -- 0x00000000800215d0
-- -- 0x00000000800215d0
-- -- 0x00000000800215d0
------------------ END OF PCB ----------------
```

The observations are:

- Among the kernel related contents, `kernel_satp` and `kernel_trap` are the same in parent and child while `kernel_sp` is different since each process is allotted a separate kernel stack, but the kernel's satp and trap addresses are constant and do not depend on the process.
- The `kernel_hartid` is different in parent and child since the processes are independent and can execute in different CPU cores in a multi-tasking environment.
- The `epc` is same in parent and child as they both return to the same address, which is where the `fork()` returns in the `attack.c` program.
- All the register entries are the same in parent and child since they are copied from the parent to the child during fork system call, except one register `a0` which stores the return value of `fork()` and is different in the parent and child. In the child process, `a0` is `0x0` and in the parent it is `0x4` which is the PID of the child process, as expected.

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:

user/attack.c

```c
#include "kernel/param.h"
#include "kernel/types.h"
#include "user/user.h"

void main(void) {
  int pid = fork();
  if (pid > 0) {
    wait(0);
    // printf("\nFROM PARENT :\n");
    if (pcbread() < 0) {
      printf("error: during pcbread system call\n");
    }
  }
  else {
    // printf("\nFROM CHILD :\n");
    // if (pcbread() < 0) {
    //   printf("error: during pcbread system call\n");
    // }
    //
    // few additional system calls for Lab 4 section 3 qn4
    sleep(5);
    printf("S");
  }
```

```c
  exit(0);
}
```

```
$ attack
-------- TRAPFRAME a0 to a6 for syscall: fork --------
-- a0: 0x0000000000000001
-- a1: 0x0000000000002fe0
-- a2: 0x00000000000014bf
-- a3: 0x0000000000003ea0
-- a4: 0x00000000000013f8
-- a5: 0x00000000000000ee
-- a6: 0x0000100000000000
---------------------- THE END ---------------------

-------- TRAPFRAME a0 to a6 for syscall: wait --------
-- a0: 0x0000000000000000
-- a1: 0x0000000000002fe0
-- a2: 0x00000000000014bf
-- a3: 0x0000000000003ea0
-- a4: 0x00000000000013f8
-- a5: 0x00000000000000ee
-- a6: 0x0000100000000000
---------------------- THE END ---------------------

-------- TRAPFRAME a0 to a6 for syscall: sleep --------
-- a0: 0x0000000000000005
-- a1: 0x0000000000002fe0
-- a2: 0x00000000000014bf
-- a3: 0x0000000000003ea0
-- a4: 0x00000000000013f8
-- a5: 0x00000000000000ee
-- a6: 0x0000100000000000
---------------------- THE END ---------------------

-------- TRAPFRAME a0 to a6 for syscall: write --------
-- a0: 0x0000000000000001
-- a1: 0x0000000000002edf
-- a2: 0x0000000000000001
-- a3: 0x0000000000003ea0
-- a4: 0x00000000000013f8
-- a5: 0x00000000000000ee
-- a6: 0x0000100000000000
---------------------- THE END ---------------------

S-------- TRAPFRAME a0 to a6 for syscall: exit --------
-- a0: 0x0000000000000000
-- a1: 0x0000000000002edf
-- a2: 0x0000000000000001
-- a3: 0x0000000000003ea0
-- a4: 0x00000000000013f8
-- a5: 0x00000000000000ee
-- a6: 0x0000100000000000
---------------------- THE END ---------------------
```

```
-------- TRAPFRAME a0 to a6 for syscall: pcbread --------
-- a0: 0x0000000000000004
-- a1: 0x0000000000002fe0
-- a2: 0x00000000000014bf
-- a3: 0x0000000000003ea0
-- a4: 0x00000000000013f8
-- a5: 0x00000000000000ee
-- a6: 0x0000100000000000
---------------------- THE END ---------------------

------------ PROCESS CONTROL BLOCK -----------
-- PID: 3
-- Name: attack
-- Process state: RUNNING (4)
-- Killed ?: no
-- Exit status: 0
-- Parent PID: 2 | Name: sh
-- kstack virtual address: 0x0000003fffff9000
-- Size: 12288 Bytes
-- Pagetable base address: 0x0000000087f49000
-- Context base address: 0x0000000080011a10
-- Trapframe base address: 0x0000000087f65000
-- cwd inode address: 0x000000008001f9e0
-- Open files struct address:
-- -- 0x00000000800215d0
-- -- 0x00000000800215d0
-- -- 0x00000000800215d0
------------------ END OF PCB ----------------
-------- TRAPFRAME a0 to a6 for syscall: exit --------
-- a0: 0x0000000000000000
-- a1: 0x0000000000002fe0
-- a2: 0x00000000000014bf
-- a3: 0x0000000000003ea0
-- a4: 0x00000000000013f8
-- a5: 0x00000000000000ee
-- a6: 0x0000100000000000
---------------------- THE END ---------------------
```

The observations are:

- The first system call in *user/attack.c* is `fork()` which takes no argument and the registers `a0-a6` contain some arbitrary values.
- The second system call is `wait(0)` in the parent process which takes an argument of type `int*`. We see that `a0` in the trapframe contains `0x0` which is what we passed to `wait()` in *user/attack.c*. The others registers contain arbitrary values.
- Next, we see `sleep(5)` in the child process and `a0` being `0x5`. Other registers contain arbitrary values.
- Next is the `write()` system call which takes three arguments - `int`, `const void*` (address), `int`. The `printf("S")` call gets translated to a `putc('S')`

call which then invokes `write(1, <address of char variable storing 'S'>, 1)`. And we see that in the trapframe `a0, a1, a2` contain `0x1`, `0x2edf` (arbitrary address but of the char variable actually), `0x1`. Other registers contain arbitrary values.

- There is an `exit(0)` system call in both child and parent, and in both trapframes, `a0` contains `0x0` and other registers contain arbitrary values.
- The `pcbread()` system call in the parent process takes no arguments and all registers in the trapframe contain arbitrary values.
- The arguments to the system called are passed through the registers `a0` to `a7` (the initials ones till it is possible) according to the calling convention in RISC-V. And, they are caller saved registers, so the registers not used for passing arguments can contain junk values across function calls which are not preserved. This is exactly what the outputs we obtained also reflect.

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.