https://pdos.csail.mit.edu/6.828/2021/labs/traps.html

RISC-V assembly

1

Arguments go in registers a0 through a7

13 is stored in `a2`, see the code:

```
  printf("%d %d\n", f(8)+1, 13);
  24: 4635                    li  a2,13
  26: 45b1                    li  a1,12
  28: 00000517              auipc a0,0x0
  2c: 7a850513              addi  a0,a0,1960 # 7d0 <malloc+0xea>
  30: 00000097              auipc ra,0x0
  34: 5f8080e7              jalr  1528(ra) # 628 <printf>
```

2

at line

```
  26: 45b1                    li  a1,12
```

it seems that the inline optimization already computed `f(8) + 1 = 12`

3

see main:

```
000000000000001c <main>:

void main(void) {
  1c: 1141                    addi  sp,sp,-16
  1e: e406                    sd  ra,8(sp)
  20: e022                    sd  s0,0(sp)
  22: 0800                    addi  s0,sp,16
  printf("%d %d\n", f(8)+1, 13);
  24: 4635                    li  a2,13
  26: 45b1                    li  a1,12
  28: 00000517              auipc a0,0x0
  2c: 7a850513              addi  a0,a0,1960 # 7d0 <malloc+0xea>
  30: 00000097              auipc ra,0x0
  34: 5f8080e7              jalr  1528(ra) # 628 <printf>
  exit(0);
  38: 4501                    li  a0,0
  3a: 00000097              auipc ra,0x0
  3e: 276080e7              jalr  630(ra) # 2b0 <exit>
```

at 30:
ra = pc = 0x000000000000001c + 30 = 0x000000000000003a
location of `printf` = 1528 + ra = 0x0000000000000632

4

ra should be the next instruction to executed after `jalr`, so it is
0x000000000000001c + 38 = 0x0000000000000042

5

if we run the following program:

```
int main() {
  unsigned int i = 0x00646c72;
  printf("H%x Wo%s", 57616, &i);
}
```

res: `He110 World`

If RISC-V is big endian, we need to reverse the order of bits within each byte.

so for 57616

6

whatever value that happens to be on the stack above 3 will be printed.
assembly code:

```
  .file "testprint.c"
  .option nopic
  .attribute arch, "rv64i2p0_m2p0_a2p0_f2p0_d2p0_c2p0"
  .attribute unaligned_access, 0
  .attribute stack_align, 16
  .text
  .section  .rodata
  .align  3
.LC0:
  .string "x=%d y=%d"
  .text
  .align  1
  .globl  main
  .type main, @function
main:
  addi  sp,sp,-16
  sd  ra,8(sp)
  sd  s0,0(sp)
  addi  s0,sp,16
  li  a1,3
  lui a5,%hi(.LC0)
  addi  a0,a5,%lo(.LC0)
```

```
    call  printf
    li  a5,0
    mv  a0,a5
    ld  ra,8(sp)
    ld  s0,0(sp)
    addi  sp,sp,16
    jr  ra
    .size main, .-main
    .ident  "GCC: () 10.2.0"
```

since we do not set value for a2, so a2 is a random value and whatever value stored in that address will be printed, if that is valid address.

code explain

.file "testprint.c"

- This directive indicates the name of the source file being compiled (in this case, testprint.c).

.option nopic

- This tells the assembler not to generate position-independent code (PIC). PIC is typically used for shared libraries, and since it’s not required here, this option disables it.

.attribute arch, "rv64i2p0_m2p0_a2p0_f2p0_d2p0_c2p0"

- This specifies the architecture and features of the RISC-V target. It indicates a 64-bit RISC-V machine (rv64), with certain extensions enabled (e.g., integer, multiplication, floating point, etc.).

.attribute unaligned_access, 0

- This specifies that unaligned memory access is not allowed. In RISC-V, memory accesses should generally be aligned to avoid performance penalties.

.attribute stack_align, 16

- This ensures that the stack pointer is aligned to 16-byte boundaries, which is required for efficient data access and compatibility with certain instructions.

.text

- This starts the section of the program that contains executable instructions (code).

.section .rodata

- This starts the section for read-only data (such as string literals).

.align 3

- This ensures that the .LC0 string is aligned to a 3-byte boundary, which is a common alignment for string literals.

.LC0: .string "x=%d y=%d"

- This defines a string constant (used in printf) to format the output. It will print two integers, labeled x and y.

.text

- This indicates the beginning of another text section. The next part of the code will be executable instructions.

.align 1

- Aligns the following code to a 1-byte boundary.

.globl main

- This makes the main function globally visible, meaning it can be called from other modules or linked into the final program.

.type main, @function

- This declares the symbol main as a function.

main:

- This is the label marking the entry point for the main function.

addi sp, sp, -16

- This instruction adjusts the stack pointer (sp) to allocate 16 bytes of space on the stack.

sd ra, 8(sp)

- This saves the return address (ra) to the stack at position sp + 8.

sd s0, 0(sp)

- This saves the value of register s0 to the stack at position sp.

addi s0, sp, 16

- This sets the s0 register to point to the stack space allocated for local variables. It's used for managing the stack frame.

li a1, 3

- This loads the immediate value 3 into register a1, which is used as an argument to printf.

lui a5, %hi(.LC0)

- This loads the upper 20 bits of the address of the string .LC0 (the format string) into register a5.

addi a0, a5, %lo(.LC0)

- This adds the lower 12 bits of the address of .LC0 to a5 to get the full address and stores it in register a0. This will be passed to printf.

call printf

- This calls the printf function to print the formatted string x=%d y=%d with the value of a1 (which is 3), though the second argument is not loaded here, so only 3 would be printed.

Backtrace

core function is in printf.c:

```
void backtrace()
{
  printf("backtrace:\n");
  uint64 fp = r_fp();
  uint64 top = PGROUNDUP(fp);
  uint64 bottom = PGROUNDDOWN(fp);

  while(fp >= bottom && fp <= top) {
    // location of return address in user virtual memory = fp - 8
    // we need to print the M(fp - 8) - 8, which is the address of calling function
    uint64 ra = *(uint64 *)(fp - 8);

    printf("%p\n", ra - 8);

    fp = *(uint64 *)(fp - 16);
  }
}
```

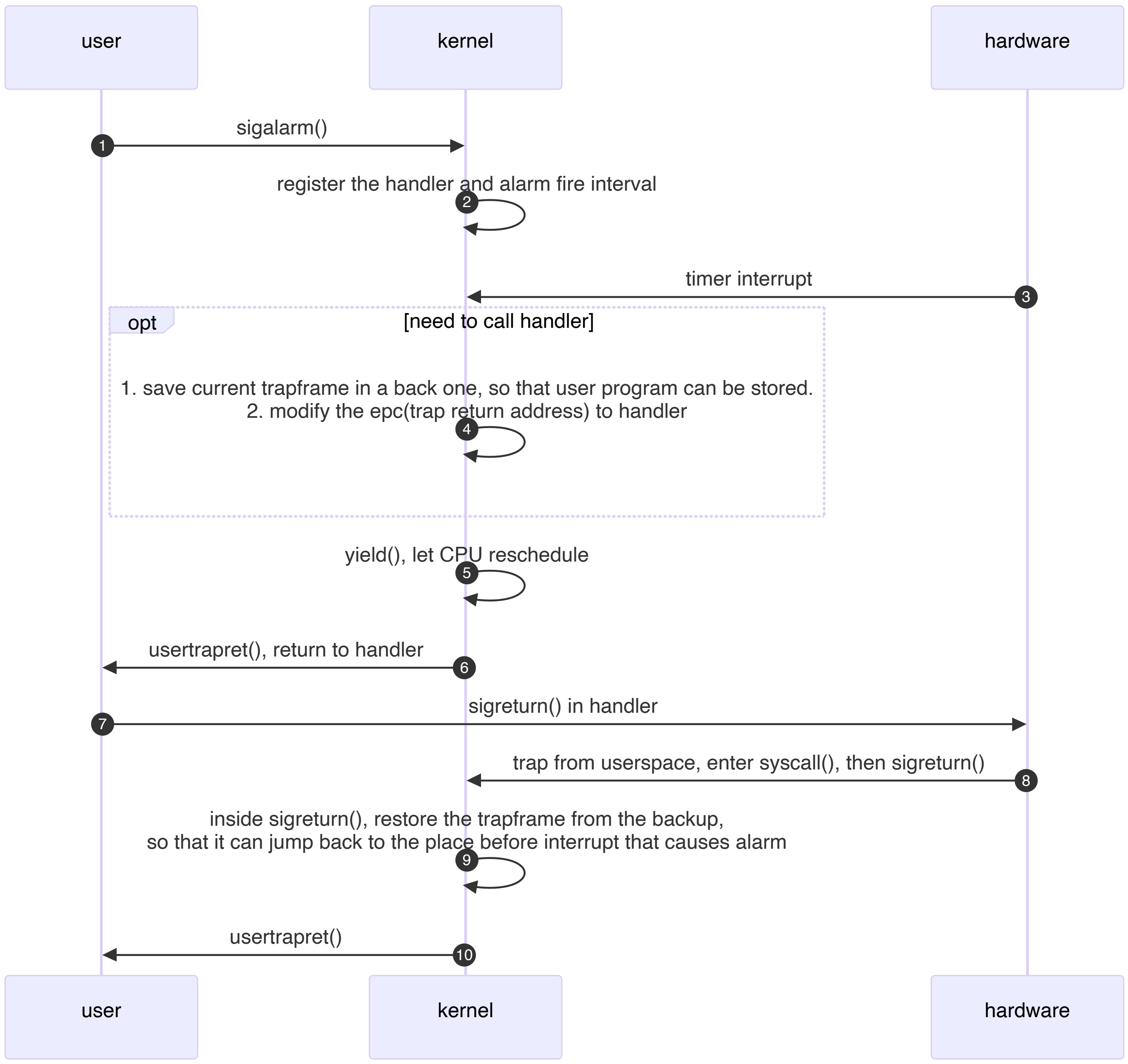
result:

```
$ bttest
backtrace:
0x0000000080002148
0x0000000080001fac
0x0000000080001c96
$ QEMU: Terminated
$ addr2line -e kernel/kernel
0x0000000080002148
/home/toadjiang/Code/OS_study/xv6-labs-2021/kernel/sysproc.c:73
0x0000000080001fac
/home/toadjiang/Code/OS_study/xv6-labs-2021/kernel/syscall.c:139 (discriminator 1)
0x0000000080001c96
/home/toadjiang/Code/OS_study/xv6-labs-2021/kernel/trap.c:67
^Ax??:0
```

note that when we call `sys_sleep()`, we are in the kernel stack in supervisor mode with kernel satp. So VM is directly mapped to PM.
We can get return address `ra` at `Memory[fp-8]` and the previous frame pointer `fp` at `Memory[fp-16]`, and we repeatedly print `ra - 8`, which is the instruction jump to the next routine, are update fp with previous fp.

Alarm

basic workflow:



user
kernel
hardware
1
sigalarm()
2
register the handler and alarm fire interval
3
timer interrupt
opt
[need to call handler]
4
1. save current trapframe in a back one, so that user program can be stored.
2. modify the epc(trap return address) to handler
5
yield(), let CPU reschedule
6
usertrapret(), return to handler
7
sigreturn() in handler
8
trap from userspace, enter syscall(), then sigreturn()
9
inside sigreturn(), restore the trapframe from the backup,
so that it can jump back to the place before interrupt that causes alarm
10
usertrapret()
user
kernel
hardware

core code

in `sysproc.c`, add two syscall implementation

```
uint64
sys_sigalarm(void)
{
  if(argint(0, &myproc()->alarminterval) < 0)
    return -1;
  if(argaddr(1, &myproc()->alarmhandler) < 0)
    return -1;
  myproc()->alarmticks = 0;
  printf("sys_sigalarm: alarminterval: %d, alarmhandler: %p\n", myproc()->alarminterval,
myproc()->alarmhandler);
  return 0;
}
```

```
uint64
sys_sigreturn(void)
{
  // restore the trapframe before interrupt that causes handler
  memmove(myproc()->trapframe, myproc()->interptf, sizeof(struct trapframe));
  // set in handler = false
  myproc()->inhandler = 0;
  return 0;
}
```

inside `trap.c`, add extra logic to check if handler need to be executed:

```
  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2){
    // lab4: if alarm interval is set, add alarmticks by 1
    if(p->alarminterval > 0 && p->inhandler == 0){
      printf("alarmticks: %d\n", p->alarmticks);
      p->alarmticks ++;
      if(p->alarmticks == p->alarminterval){
        p->alarmticks = 0;
        // saves user program state
        memmove(p->interptf, p->trapframe, sizeof(struct trapframe));
        // set in handler = true
        p->inhandler = 1;
        // modify the trap return address to handler
        p->trapframe->epc = p->alarmhandler;
        // execute the alarm handler
        usertrapret();
      }
    }
    yield();
  }
```