Lab4: Traps

- 2351289周慧星

目录

实验跑分

- 最终在traps分支下跑分：

```
make grade
```

- 得分：

```
== Test answers-traps.txt ==
answers-traps.txt: OK
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (6.6s)
== Test running alarmtest ==
$ make qemu-gdb
(5.9s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test   alarmtest: test3 ==
  alarmtest: test3: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (119.6s)
== Test time ==
time: OK
Score: 95/95
```

实验准备

本实验围绕 **陷阱（traps）** 机制展开，探索系统调用的实现原理，包含栈操作热身练习和用户级陷阱处理实现。陷阱是用户空间与内核空间交互的核心机制（如系统调用、异常、中断），通过本实验可深入理解 xv6 的陷阱处理流程。

1. **关键概念**

- **陷阱（Trap）**：用户程序触发的异常或系统调用，导致 CPU 从用户态切换到内核态。
- **中断（Interrupt）**：外部设备（如定时器）触发的事件，同样会进入内核处理。
- ** trampoline **：跳板页，用于在用户态和内核态之间安全切换（保存/恢复寄存器、页表切换）。

2. **核心文件**

- `kernel/trampoline.S`：用户态与内核态切换的汇编代码（关键跳板逻辑）。
- `kernel/trap.c`：所有陷阱和中断的处理逻辑（分发到系统调用或异常处理）。
- `kernel/trapframe.h`：`trapframe` 结构体定义，保存陷阱发生时的寄存器状态。
- `user/uservec.S`/`user/usertrap.S`：用户态陷阱入口代码。

1. **切换到实验分支**：

```
git fetch
git checkout traps
make clean
```

2. **阅读文档**：
   - xv6 书籍第 4 章（Trap），理解陷阱处理的整体流程。
   - 浏览 `trampoline.S` 和 `trap.c`，标记关键函数（如 `usertrap()`、`usertrapret()`）。

实验1：RISC-V assembly(easy)

了解一些 RISC-V 汇编很重要。在 xv6 repo 中有一个文件 `user/call.c`。`make fs.img` 会对其进行编译，并生成 `user/call.asm` 中程序的可读汇编版本。 阅读 `call.asm` 中的 `g`，`f`，和 `main` 函数。（参考这些材料：reference page） 请回答下列问题：

问题1

> Which registers contain arguments to functions? For example, which register holds 13 in main's call to `printf`?

在 RISC-V 调用约定中，函数参数通过特定寄存器传递：

`a1`, `a2`, `a3` 等通用寄存器将保存函数的参数。

查看call.asm文件中的main函数可知，在 main 调用 printf 时，由寄存器 `a2` 保存 13。

```
void main(void) {
  24:   1141                addi    sp,sp,-16
  26:   e406                sd      ra,8(sp)
  28:   e022                sd      s0,0(sp)
  2a:   0800                addi    s0,sp,16
  printf("%d %d\n", f(8)+1, 13);
  2c:   4635                li      a2,13
  2e:   45b1                li      a1,12
  30:   00001517            auipc   a0,0x1
  34:   86050513            addi    a0,a0,-1952 # 890 <malloc+0xf6>
  38:   6aa000ef            jal     6e2 <printf>
  exit(0);
  3c:   4501                li      a0,0
  3e:   2a2000ef            jal     2e0 <exit>
```

问题2

> Where is the call to function `f` in the assembly code for main? Where is the call to `g`? (Hint: the compiler may inline functions.)

查看`call.asm`文件中的`f`和`g`函数可知，函数 `f` 调用函数 `g`； 函数 `g` 使传入的参数加 3 后返回。

```
int f(int x) {
  12:   1141                addi    sp,sp,-16
  14:   e406                sd      ra,8(sp)
  16:   e022                sd      s0,0(sp)
  18:   0800                addi    s0,sp,16
  return g(x);
}
  1a:   250d                addiw   a0,a0,3
  1c:   60a2                ld      ra,8(sp)
  1e:   6402                ld      s0,0(sp)
  20:   0141                addi    sp,sp,16
  22:   8082                ret
```

此外，编译器会进行内联优化，即一些编译时可以计算的数据会在编译时得出结果，而不是进行函数调用。查看 `main` 函数可以发现，`printf` 中包含了一个对 `f` 的调用。但是对应的会汇编代码却是直接将 `f(8)+1` 替换为 `12` 。

这就说明编译器对这个函数调用进行了优化，所以对于 `main` 函数的汇编代码来说，其并没有调用函数 `f` 和 `g`，而是在运行之前由编译器对其进行了计算。

问题3

> At what address is the function `printf` located?

```
void
printf(const char *fmt, ...)
{
 6e2:  711d                addi    sp,sp,-96
 6e4:  ec06                sd      ra,24(sp)
 6e6:  e822                sd      s0,16(sp)
 6e8:  1000                addi    s0,sp,32
 6ea:  e40c                sd      a1,8(s0)
 6ec:  e810                sd      a2,16(s0)
 6ee:  ec14                sd      a3,24(s0)
 6f0:  f018                sd      a4,32(s0)
 6f2:  f41c                sd      a5,40(s0)
 6f4:  03043823            sd      a6,48(s0)
 6f8:  03143c23            sd      a7,56(s0)
  va_list ap;
```

查阅得到其地址在 `0x6e2`。

问题4

> What value is in the register `ra` just after the `jalr` to `printf` in `main`?

```
void main(void) {
  24:   1141                addi    sp,sp,-16
  26:   e406                sd      ra,8(sp)
  28:   e022                sd      s0,0(sp)
  2a:   0800                addi    s0,sp,16
  printf("%d %d\n", f(8)+1, 13);
  2c:   4635                li      a2,13
  2e:   45b1                li      a1,12
  30:   00001517            auipc   a0,0x1
  34:   86050513            addi    a0,a0,-1952 # 890 <malloc+0xf6>
  38:   6aa000ef            jal     6e2 <printf>
  exit(0);
  3c:   4501                li      a0,0
  3e:   2a2000ef            jal     2e0 <exit>
```

问题5

> Run the following code.
>
> ```
> unsigned int i = 0x00646c72;
> printf("H%x Wo%s", 57616, &i);
> ```
>
> What is the output? [Here's an ASCII table](ASCII Table - ASCII Character Codes, HTML, Octal, Hex, Decimal) that maps bytes to characters.
>
> The 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 and a more whimsical description.

运行结果：打印出了 `He110 World`。

57616 转换为十六进制是 0xe110（因为 0xe110 = 57616）。

i 是 4 字节无符号整数，值为 0x00646c72。在 小端（little-endian） 系统（如 RISC-V）中，内存中字节的存储顺序是反向的：0x72、0x6c、0x64、0x00。

这些字节对应的 ASCII 字符依次为：'r'（0x72）、'l'（0x6c）、'd'（0x64）、'\0'（0x00，字符串结束符），因此字符串为 "rld"。

综上，输出为：He110 World

大端（big-endian）调整：

在大端系统中，字节按原值顺序存储。要得到相同输出，i 的值需设为 0x726c6400（确保内存中字节顺序为 0x72、0x6c、0x64、0x00）。

57616 是数值，与字节顺序无关，因此无需修改。

问题6

> 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);
> ```

输出中 y= 后面会是 不确定的值（如随机数或垃圾值）。

原因：printf 格式字符串要求两个 %d 参数，但实际只传递了一个（3）。第二个参数会从栈上未初始化的位置或寄存器中读取，导致结果不确定（这是未定义行为）。

实验2：Backtrace(moderate)

一、实验目的

1. 实现内核回溯（backtrace）功能，通过遍历栈帧获取函数调用链，辅助调试。
2. 理解 RISC-V 架构的栈帧结构，掌握帧指针（frame pointer）在函数调用中的作用。
3. 学会在 xv6 内核中插入调试功能，并通过工具（如 `addr2line`）解析回溯结果。

二、实验步骤

1. 添加帧指针读取函数

在 `kernel/riscv.h` 中添加读取寄存器 `s0`（帧指针）的函数，用于获取当前栈帧的起始地址：

```
#ifndef __ASSEMBLER__
static inline uint64
r_fp()
{
  uint64 x;
  asm volatile("mv %0, s0" : "=r" (x));
  return x;
}
#endif
```

2. 实现 backtrace 函数

在 `kernel/printf.c` 中实现回溯功能，遍历栈帧并打印返回地址：

```
void
backtrace(void)
{
  printf("backtrace:\n");
```

```
  uint64 fp = r_fp();  // 获取当前帧指针（s0 的值）
  uint64 stack_page = PGROUNDDOWN(fp);  // 栈所在页的起始地址（页对齐）
  while (fp >= stack_page && fp < stack_page + PGSIZE) {
    uint64 ra = *(uint64*)(fp - 8);  // 返回地址在帧指针 -8 处
    printf("%p\n", (void*)ra);              // 打印返回地址
    uint64 prev_fp = *(uint64*)(fp - 16);  // 上一级帧指针在帧指针 -16 处
    if (prev_fp <= fp)  // 防止循环（栈帧应向上生长）
      break;
    fp = prev_fp;  // 移动到上一级帧指针
  }
}
```

3. 声明函数原型

在 `kernel/defs.h` 中添加 `backtrace` 函数的声明，使其可在其他文件中调用：

```
void backtrace(void);
```

4. 在 sys_sleep 中调用 backtrace

修改 `kernel/sysproc.c` 的 `sys_sleep` 函数，插入 `backtrace` 调用以验证功能：

```
  backtrace();  // 调用回溯函数
```

5. 在 panic 中添加 backtrace

修改 `kernel/printf.c` 的 `panic` 函数，使其在系统崩溃时自动打印回溯信息：

```
  backtrace();  // 崩溃时打印调用链
```

6. 编译并测试

```
make clean
make qemu
# 在 xv6 命令行中运行测试
bttest
```

运行 bttest 之后退出 qemu。在终端中:运行 addr2line -e kernel/kernel并剪切粘贴上述地址.

三、实验结果

1. **运行 `bttest` 输出**：

```
hart 1 starting
hart 2 starting
init: starting sh
$ bttest
backtrace:
0x0000000080001d86
0x0000000080001ca8
0x0000000080001a32
$
```

2. **使用 `addr2line` 解析地址**：

```
xing@xing-VMware-Virtual-Platform:~/桌面/xv6-labs-2024$ addr2line -e kernel/kernel
0x0000000080001d86
/home/xing/桌面/xv6-labs-2024/kernel/sysproc.c:59
0x0000000080001ca8
/home/xing/桌面/xv6-labs-2024/kernel/syscall.c:141 (discriminator 1)
0x0000000080001a32
/home/xing/桌面/xv6-labs-2024/kernel/trap.c:76
```

四、遇到的问题与解决方案

1. **栈帧遍历无限循环**
   - **问题**：未正确判断栈帧终止条件，导致 `backtrace` 无限循环。
   - **解决方案**：通过 `PGROUNDDOWN(fp)` 获取栈页基地址，确保只遍历当前栈页内的帧（`fp` 必须在 `[stack_page, stack_page + PGSIZE)` 范围内）。

2. **返回地址解析错误**
   - **问题**：错误计算返回地址在栈帧中的偏移，导致打印无效地址。
   - **解决方案**：根据 RISC-V 栈帧结构，返回地址位于帧指针 `-8` 处，上一级帧指针位于 `-16` 处（需严格匹配内存布局）。

3. **函数未声明导致编译错误**
   - **问题**：在 `sys_sleep` 中调用 `backtrace` 时，编译器提示“未声明的函数”。
   - **解决方案**：在 `kernel/defs.h` 中添加 `backtrace` 的函数原型，确保跨文件调用有效。

4. **`r_fp` 函数无法读取 s0**
   - **问题**：内联汇编语法错误，导致无法正确读取 `s0` 寄存器。
   - **解决方案**：使用正确的 RISC-V 汇编指令 `mv %0, s0`（移动 `s0` 到输出寄存器），并通过约束 `"=r"` 指定输出变量。

五、实验心得

1. **栈帧结构的重要性**：RISC-V 的栈帧通过帧指针 `s0` 形成链式结构，每个帧保存上一级帧指针和返回地址，这是回溯功能的核心基础。理解栈帧布局是调试内核的关键。
2. **硬件与软件的协作**：回溯功能依赖于 CPU 寄存器（`s0` 保存帧指针）和编译器生成的栈帧布局，体现了硬件架构与软件约定的紧密配合。
3. **调试工具的作用**：`addr2line` 工具能将二进制地址转换为源码位置，展示了调试信息（如符号表）在软件开发中的重要性，为定位复杂问题提供了高效手段。
4. **内核调试的实践意义**：在 `panic` 中添加回溯，能在系统崩溃时快速定位错误调用链，大幅简化内核调试流程。这种"主动暴露信息"的思路在实际开发中非常有用。

实验3：Alarm (hard)

一、实验目的

实现一个周期性闹钟功能，通过 `sigalarm(interval, handler)` 系统调用，让内核在进程消耗指定数量的 CPU 时钟 tick 后，触发用户定义的处理函数 `handler`。当处理函数返回后，进程应从被中断的位置继续执行。具体目标：

1. 新增 `sigalarm` 和 `sigreturn` 两个系统调用。
2. 确保闹钟按指定间隔（`interval` 个 tick）触发，且处理函数执行完成后进程能正确恢复。
3. 避免处理函数重入（若上一次处理未完成，不触发新的处理）。
4. 通过 `alarmtest` 测试和 `usertests -q` 验证。

二、实验步骤

1. 添加系统调用框架

定义系统调用号和原型

- 在 `kernel/syscall.h` 中添加：

```
#define SYS_sigalarm 22
#define SYS_sigreturn 23
```

- 在 `user/user.h` 中添加用户态函数原型：

```
int sigalarm(int ticks, void (*handler)());
int sigreturn(void);
```

- 在 `user/usys.pl` 中添加系统调用入口：

```
  entry("sigalarm");
  entry("sigreturn");
```

- 在 kernel/syscall.c 中添加函数指针和声明：

```
  extern uint64 sys_sigalarm(void);
  extern uint64 sys_sigreturn(void);
  static uint64 (*syscalls[])(void) = {
    // ... 其他系统调用 ...
    [SYS_sigalarm] sys_sigalarm,
    [SYS_sigreturn] sys_sigreturn,
  };
```

2. 扩展进程结构体（struct proc）

在 kernel/proc.h 中添加闹钟相关字段：

```
struct proc {
  uint64 interval;              // 间隔
  void (*handler)();            // 定时处理的函数
  uint64 ticks;                 // 上一次调用函数距离的时间
  struct trapframe *alarm_trapframe;  // 用于恢复 trapframe
  int alarm_goingoff;           // 是否正在alarm，防止嵌套的中断，导致trapframe丢失
};
```

3. 初始化进程闹钟字段

在 kernel/proc.c 的 allocproc 中初始化新增字段：

```
if((p->alarm_trapframe = (struct trapframe *)kalloc()) == 0) {
    freeproc(p);
    release(&p->lock);
    return 0;
  }
  p->ticks = 0;
  p->handler = 0;
  p->interval = 0;
  p->alarm_goingoff = 0;
```

4. 实现 sys_sigalarm 系统调用

在 kernel/sysproc.c 中添加：

```
uint64
sys_sigalarm(void) {
  int n;
  uint64 handler;
  argint(0, &n);
  argaddr(1, &handler);
  return sigalarm(n, (void(*)())(handler));
}
```

5. 实现 `sys_sigreturn` 系统调用

在 `kernel/sysproc.c` 中添加:

```
uint64
sys_sigreturn(void) {
  return sigreturn();
}
```

6. 修改陷阱处理逻辑（`kernel/trap.c`）

在 `usertrap` 中添加闹钟触发逻辑:

```
  if(which_dev == 2){
    if(p->interval != 0) { // 如果设定了时钟事件
      if(p->ticks++ == p->interval) {
        if(!p->alarm_goingoff) { // 确保没有时钟正在运行
          p->ticks = 0;
          *(p->alarm_trapframe) = *(p->trapframe);
          p->trapframe->epc = (uint64)p->handler;
          p->alarm_goingoff = 1;
        }
      }
    }
    yield();
  }
```

7. 释放资源（`kernel/proc.c`）

在进程退出时释放 `alarm_tf` 分配的内存,在`freeproc`添加:

```
if(p->alarm_trapframe)
    kfree((void*)p->alarm_trapframe);
  p->alarm_trapframe = 0;
  p->ticks = 0;
  p->handler = 0;
```

```
    p->interval = 0;
    p->alarm_goingoff = 0;
```

8. 修改 Makefile

在 `Makefile` 的 `UPROGS` 中添加 `alarmtest`：

```
    $U/_alarmtest\
```

三、实验结果

编译并运行测试：

```
make CPUS=1 qemu
alarmtest
usertests -q
```

输出如下：

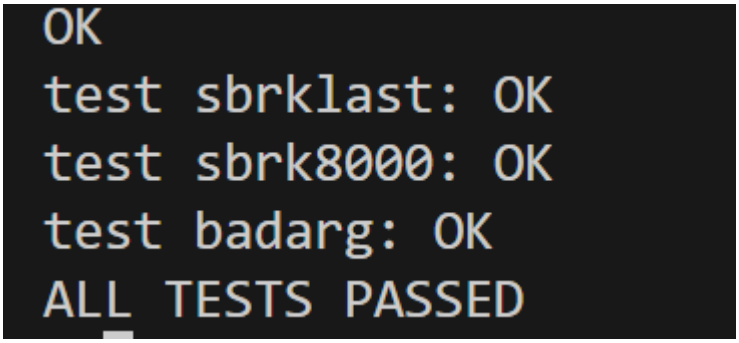
```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ alarmtest
test0 start
.....................alarm!
test0 passed
test1 start
....alarm!
...alarm!
...alarm!
...alarm!
...alarm!
..alarm!
...alarm!
...alarm!
...alarm!
....alarm!
test1 passed
test2 start
............................alarm!
test2 passed
test3 start
test3 passed
$
```

运行 `usertests -q` 来确保你的修改没有影响到内核的其他部分。

- **test0**：验证闹钟能触发处理函数 `periodic`。
- **test1**：验证处理函数返回后进程能继续执行，且闹钟按间隔触发。

- **test2**：验证处理函数未返回时不会重入。
- **test3**：验证 `sigalarm(0, 0)` 能停止闹钟。



当显示上述内容后，说明代码通过测试，没有影响其他部分。

四、遇到的问题与解决方案

1. 进程无法从处理函数返回

- **问题**：处理函数执行后，进程无法回到被中断的位置（`sepc` 未正确恢复）。
- **解决方案**：在 `usertrap` 中触发闹钟时，保存当前 `trapframe` 到 `alarm_tf`，`sigreturn` 时恢复该 `trapframe`，包括 `sepc`（被中断的指令地址）。

2. 处理函数重入

- **问题**：若处理函数执行时间超过 `interval`，会被重复触发。
- **解决方案**：添加 `alarm_active` 标记，当处理函数执行时（`alarm_active=1`），不触发新的处理，直到 `sigreturn` 重置该标记。

3. 寄存器状态不一致

- **问题**：处理函数执行后，寄存器（如 `a0`、`s0-s11`）的值被修改，导致进程后续执行错误。
- **解决方案**：`alarm_tf` 保存了所有寄存器状态，`sigreturn` 恢复整个 `trapframe`，确保寄存器与中断前一致。

4. 闹钟计数错误

- **问题**：`alarm_ticks` 未正确递减，导致闹钟间隔不准确。
- **解决方案**：在每次时钟中断（`which_dev == 2`）时，若闹钟激活且未处理中，则 `alarm_ticks--`，直到为 0 时触发。

5. `sigreturn` 系统调用返回值覆盖 `a0`

- **问题**：`sigreturn` 作为系统调用，会将返回值存入 `a0`，覆盖中断前的 `a0`。
- **解决方案**：`alarm_tf` 保存了中断前的 `a0`，`sigreturn` 恢复整个 `trapframe` 时会覆盖系统调用设置的 `a0`，确保与中断前一致。

五、实验心得

1. **用户态与内核态协作**：闹钟功能需要内核和用户态紧密配合。内核负责计数和触发，用户态提供处理函数，`sigreturn` 作为桥梁恢复进程状态，体现了操作系统分层设计的思想。

2. **状态保存的重要性**：陷阱处理的核心是状态的完整保存与恢复。`trapframe` 结构体保存了所有寄存器和关键状态，确保中断前后进程执行环境一致，这是进程透明性的基础。

3. **并发控制**：通过 `alarm_active` 防止处理函数重入，展示了简单的并发控制逻辑。在多任务环境中，类似的标记（如锁、信号量）是避免资源竞争的关键。

4. **系统调用设计**：`sigalarm` 负责设置参数，`sigreturn` 负责恢复状态，分工明确。这种拆分降低了复杂度，也符合"单一职责"原则。

5. **调试技巧**：使用 `make CPUS=1 qemu-gdb` 单 CPU 调试，结合 `alarmtest.asm` 查看用户态汇编，能快速定位 `sepc` 恢复错误、寄存器不一致等问题。