Lab4 Traps

Sublab1 RISC-V assembly

阅读完 `user/call.asm`，问题回答:

```
1. 哪些寄存器用于传递函数参数？
   RISC-V 中使用 a0 到 a7 寄存器传递函数参数。
   对于 printf("H%x Wo%s", 57616, &i)，参数情况如下：
   a0 = 格式字符串 "H%x Wo%s"
   a1 = 57616
   a2 = &i（指向变量 i 的地址）

2. main 函数中 f 和 g 的调用在哪里？
   在 call.asm 中，没有看到对 f() 或 g() 的 jal 或 call 指令。
   这是因为编译器将这两个函数“内联”到了 main 中，优化掉了真正的函数调用。

3. printf 函数位于哪个地址？
   在 call.asm 中，printf 函数的定义位置为：
   0x0000000000001140
   所以 printf 位于地址 0x1140。

4. main 中 jalr 调用 printf 之后 ra 寄存器中的值是多少？
   ra 是 return address（返回地址）寄存器。
   在调用 printf 的 jalr 之后，ra 中保存的是“下一条指令的地址”，也就是返回到 main 中 printf 调用之后的位置。

5. 执行以下代码输出什么？
     unsigned int i = 0x00646c72;
     printf("H%x Wo%s", 57616, &i);

   输出是：He110 World

   解释：
   57616 以 16 进制打印为 e110，所以 H%x 打印为 He110。
   &i 指向的内存内容（小端）是：0x72 0x6c 0x64 0x00，即字符串 "rld"
   所以 printf 输出 "World"

6. 如果系统为大端（big-endian），该如何设置 i 才能得到相同输出？
   要让内存中的顺序为：0x72, 0x6c, 0x64, 0x00（即 "rld"）
   在大端系统中，需要将 i 设置为：
   i = 0x726c6400

   不需要修改 57616，因为整数在寄存器中传递，不受大小端影响。

7. 以下代码中 'y=' 后会输出什么？为什么？
     printf("x=%d y=%d", 3);

   输出是：x=3 y=（随机值，垃圾值）
   解释：
   printf 要求两个参数（两个 %d），但只传入了一个（3）
```

```
45    printf 会尝试访问第二个参数，但该值未定义，因此输出为垃圾值（未定义行为）
```

Sublab2 Backtrace

一、环境搭建

本实验在 xv6-labs-2021 的 `traps` 分支中进行。首先通过 `git fetch` 和 `git checkout trap` 切换至对应分支，然后执行 `make clean` 清理旧的构建文件，确保环境干净。实验在 Ubuntu 22.04 系统上进行，所需交叉编译工具链、QEMU 等配置与前几个实验一致。

二、实验目的

本实验旨在实现一个内核栈回溯（backtrace）函数，用于调试时打印函数调用栈信息。通过该实验，理解函数调用过程中栈帧的组织结构、帧指针（frame pointer）的作用，并掌握如何在内核中获取和遍历调用栈信息，为调试内核错误提供有效手段。

三、实验内容

- 在 `kernel/riscv.h` 中添加获取帧指针的函数：
- 在 `kernel/defs.h` 中声明 `void backtrace();`，以便在其他文件中调用。
- 在 `kernel/printf.c` 中实现 `backtrace()` 函数，使用如下逻辑：
  - 通过 `r_fp()` 获取当前帧指针；
  - 使用 `PGROUNDDOWN(fp)` 和 `PGROUNDUP(fp)` 限定栈的范围；
  - 通过遍历栈帧，逐级读取返回地址（位于当前帧指针的 `-8` 处），并打印；
  - 同时更新当前帧指针为其上一层（位于 `-16` 处保存的前一个帧指针），直到越界。
- 在 `kernel/sysproc.c` 的 `sys_sleep()` 函数中插入 `backtrace()` 调用，确保运行 `bttest` 时可以触发栈回溯。
- 编译并执行 `make qemu`，在 QEMU 中运行 `bttest` 测试程序，验证输出。

四、实验结果分析

运行 `bttest` 后输出如下内容：

```
backtrace:
0x0000000080002cda
0x0000000080002bb6
0x0000000080002898
```

将这些地址输入 `addr2line -e kernel/kernel` 后显示：

```
kernel/sysproc.c:74
kernel/syscall.c:224
kernel/trap.c:85
```

说明栈回溯功能能够正确输出当前函数的调用路径，地址解析也能够正确定位源文件和行号。

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ bttest
backtrace:
  0x0000000080002192
  0x0000000080001fee
  0x0000000080001cd8
$
$ QEMU: Terminated
sincetoday@LZ:~/xv6-labs-2021$
sincetoday@LZ:~/xv6-labs-2021$ addr2line -e kernel/kernel
0x0000000080002192
0x0000000080001fee
0x0000000080001cd8/home/sincetoday/xv6-labs-2021/kernel/sysproc.c:74
/home/sincetoday/xv6-labs-2021/kernel/syscall.c:140 (discriminator 1)

/home/sincetoday/xv6-labs-2021/kernel/trap.c:76

??:0
sincetoday@LZ:~/xv6-labs-2021$
```

五、实验中遇到的问题及解决方法

实现 backtrace 时未正确处理帧指针更新，导致陷入死循环。调试发现 `fp` 未在每次循环中更新为上层帧指针。修复方法是在每次循环中显式读取 `*(uint64 *)(fp - 16)` 更新帧指针后继续遍历。同时注意加入地址边界判断防止越界读取栈空间。

六、实验心得

本实验深入了解了函数调用过程中栈帧的组织结构，并掌握了如何使用帧指针进行调用路径追踪。在实现 backtrace 函数的过程中，对内核栈布局、内联汇编以及地址计算有了更深刻的认识。该功能对调试内核代码极为有用，为后续复杂功能实现和调试提供了基础能力。

附：实验部分源码

```c
void
backtrace(void)
{
  uint64 fp = r_fp();
  uint64 stack_top = PGROUNDUP(fp);
  uint64 stack_bottom = PGROUNDDOWN(fp);

  printf("backtrace: \n");

  while (fp >= stack_bottom && fp + 16 <= stack_top) {
    uint64 ra = *((uint64*)(fp - 8));
    printf("  %p\n", ra);

    uint64 prev_fp = *((uint64*)(fp - 16));
    if (prev_fp <= fp || prev_fp >= stack_top)
      break;
    fp = prev_fp;
```

```
  }
}
```

Sublab3 Alarm

一、环境搭建

本实验基于 xv6-labs-2021 的 `traps` 分支进行。在 Ubuntu 22.04 环境下，首先切换到 `trap` 分支并执行 `make clean` 确保干净的构建环境。然后将 `user/alarmtest.c` 添加到 `Makefile` 中的 `UPROGS` 列表中，以便能够编译并测试 alarm 相关功能。其余交叉编译工具链与 QEMU 环境配置与之前实验一致。

二、实验目的

本实验旨在实现一种用户态的“定时中断”机制，使用户进程能够注册一个回调函数，在消耗一定 CPU 时间（tick）后被周期性调用。通过实现 `sigalarm` 和 `sigreturn` 两个系统调用，深入理解用户态与内核态之间的上下文切换机制、定时中断处理、用户态中断恢复等内容。该机制模拟了用户态异常处理的基本框架，为处理更复杂的中断或异常提供基础。

三、实验内容

- 实现系统调用接口：
  - 在 `user/user.h` 中声明;
  - 在 `user/usys.pl` 中添加：`entry("sigalarm");` 和 `entry("sigreturn");`
  - 在 `kernel/syscall.h` 中定义 syscall 编号，在 `kernel/syscall.c` 中添加分发逻辑。
- 在 `kernel/proc.h` 中的 `struct proc` 添加以下字段：
  - `int alarm_interval;` // 闹钟周期
  - `uint64 alarm_handler;` // 用户态 handler 地址
  - `int alarm_ticks;` // 距离下次触发还剩几 tick
  - `int in_handler;` // 标志是否已进入 handler
  - `struct trapframe alarm_tf;` // 保存中断时寄存器状态
- 在 `kernel/proc.c` 的 `allocproc()` 中初始化新增字段。
- 实现 `sys_sigalarm()`，将 interval 和 handler 写入 proc 结构体，并初始化 tick 计数。
- 实现 `sys_sigreturn()`，将当前 trapframe 恢复为 handler 触发前的状态，允许用户程序继续执行。
- 在 `kernel/trap.c` 的 `usertrap()` 中：
  - 检查是否为 timer 中断（`which_dev == 2`）；
  - 若当前进程设置了 alarm 且尚未在 handler 中，则判断 tick 是否到达；
  - 保存当前 trapframe 至 `alarm_tf`，将 `trapframe.epc` 设置为 `alarm_handler`；
  - 设置 `in_handler = 1`，并重置 alarm_ticks。
- 在 `user/alarmtest.c` 中编写测试用例，包括 `sigalarm` 调用、定时打印 alarm，并确保 `sigreturn` 能正确恢复执行。

四、实验结果分析

运行 `alarmtest`，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
$
$ QEMU: Terminated
sincetoday@LZ:~/xv6-labs-2021$
sincetoday@LZ:~/xv6-labs-2021$
sincetoday@LZ:~/xv6-labs-2021$
```

说明三组测试用例均成功通过。进一步执行 `make grade` 和 `usertests`，输出结果显示所有测试均已通过，表明系统调用实现正确，用户态程序可以周期性接收中断，并能正确恢复。

五、实验中遇到的问题及解决方法

刚开始未保存中断前的 trapframe，导致 handler 返回后无法正确恢复原用户态状态。通过在 `proc` 中增加 `alarm_tf` 字段并在进入 handler 前完整拷贝当前 `trapframe`，解决了该问题。另一个问题是 handler 被重复进入，在 handler 中再次触发 timer 时未正确检查 `in_handler` 标志，添加判断后避免重复中断。

六、实验心得

通过本实验掌握了内核对用户程序的中断插入机制，并理解了如何保存和恢复用户态上下文。实现 `sigalarm` 的过程加深了对 trapframe 结构、RISC-V 调用约定以及定时器中断处理的理解。同时也感受到设计用户态中断处理机制在操作系统设计中的重要性，对系统调用实现的灵活性和完整性也有了更深刻认识。

附：实验部分源码

```
1  if(which_dev == 2){
2    if (p->alarm_ticks > 0 && p->handling_alarm == 0) {
3      p->ticks_passed++;
4      if (p->ticks_passed >= p->alarm_ticks) {
5        p->alarm_tf = (struct trapframe *)kalloc();
6        if (p->alarm_tf) {
7          memmove(p->alarm_tf, p->trapframe, sizeof(struct trapframe));
8          p->trapframe->epc = p->handler;
```

```
          p->handling_alarm = 1;
          p->ticks_passed = 0;
        }
      }
    }
    yield();
  }
```

以及 `sys_sigalarm` 和 `sys_sigreturn` 函数

```
uint64
sys_sigalarm(void)
{
  int ticks;
  uint64 handler;

  if (argint(0, &ticks) < 0)
    return -1;
  if (argaddr(1, &handler) < 0)
    return -1;

  struct proc *p = myproc();
  p->alarm_ticks = ticks;
  p->handler = handler;
  p->ticks_passed = 0;
  p->handling_alarm = 0;

  return 0;
}

uint64
sys_sigreturn(void)
{
  struct proc *p = myproc();

  // restore trapframe
  memmove(p->trapframe, p->alarm_tf, sizeof(struct trapframe));
  kfree((void *)p->alarm_tf);  // don't forget to free memory
  p->alarm_tf = 0;
  p->handling_alarm = 0;
  return 0;
}
```