Lab 4 traps

RISC-V assembly (easy)

实验目的

理解 RISC-V 汇编语言，并通过简单的汇编代码来加深对指令和寄存器的理解。

实验步骤

1. 编写一个简单的 `RISC-V` 汇编程序，将两个数相加并将结果存储在寄存器中。
2. 使用 `RISC-V` 工具链编译和链接汇编代码。
3. 运行生成的可执行文件。

```
qemu-riscv64 assembly
```

实验中遇到的问题和解决办法

- 问题：在编写汇编代码时，可能会出现语法错误或者逻辑错误，导致程序无法正确运行。
- 解决办法：可以使用 `RISC-V` 工具链提供的汇编器（`riscv64-unknown-elf-as`）进行汇编代码的检查，以及调试器（例如 `gdb`）来跟踪程序执行过程并定位错误。

实验心得

这个实验帮助我们了解了 `RISC-V` 汇编语言的基本结构和指令。我们编写了一个简单的汇编程序，使用了一些常见的汇编指令来实现两个数相加并将结果存储在寄存器中。通过这个实验，我们对 `RISC-V` 的寄存器和指令有了初步的认识，并了解了如何使用 RISC-V 工具链来编译和运行汇编代码。

Backtrace (moderate)

实验目的

实现一个函数，用于打印当前函数调用栈的情况，即返回地址的链表。

实验步骤

1. 在 `kernel/riscv.h` 中添加获取当前栈指针 fp 的函数。

```
// kernel/riscv.h
static inline uint64 r_fp() {
  uint64 x;
  asm volatile("mv %0, s0" : "=r" (x));
  return x;
}
```

2. 在 `kernel/defs.h` 中声明 `backtrace()` 函数。

```
// kernel/defs.h
void backtrace(void);
```

3. 在 `kernel/printf.c` 中实现 `backtrace()` 函数，遍历当前栈，并打印栈帧中的函数返回地址。

```
// kernel/printf.c
void backtrace(void) {
  printf("backtrace:\n");
  uint64 fp = r_fp();
  while (fp < PGROUNDUP(fp)) {
    printf("%p\n", *(uint64*)(fp-8));  // Get return address and print
    fp = *(uint64*)(fp - 16);
  }
}
```

4. 在 `kernel/sysproc.c` 的 `sys_sleep` 函数中调用 `backtrace()` 函数，以便在进程休眠时打印函数调用栈。

```
// kernel/sysproc.c
uint64 sys_sleep(void) {
  // ...
  backtrace();  // Print function call stack
  return 0;
}
```

实验中遇到的问题和解决办法

- 问题：在实现 `backtrace()` 函数时，可能会由于指针操作不当或者栈帧结构的错误导致无法正确遍历函数调用栈。
- 解决办法：首先，可以通过仔细阅读操作系统的文档和代码来理解栈帧结构和寄存器保存情况。其次，可以在实现过程中使用调试输出语句（例如 `printf`）来观察各个寄存器和栈帧的值，以便更好地理解程序的执行过程和排查错误。

实验心得

这个实验让我们了解了函数调用栈的结构，以及如何通过获取栈指针 `fp` 和栈帧指针来遍历当前函数调用栈。我们实现了一个简单的 `backtrace()` 函数，它可以打印当前函数调用栈的返回地址链表，从而帮助我们在调试过程中追踪函数的调用情况。

Alarm (hard)

实验目的

实现一个周期性执行的函数，并在每次周期执行时调用用户指定的处理函数。

实验步骤

1. 在 `kernel/syscall.h` 中定义 `sigalarm()` 和 `sigreturn()` 系统调用号。

```
// kernel/syscall.h
#define SYS_sigalarm 22
#define SYS_sigreturn 23
```

2. 在 `kernel/syscall.c` 中实现 `sys_sigalarm()` 和 `sys_sigreturn()` 系统调用。

```
// kernel/syscall.c
extern uint64 sys_sigalarm(void);
extern uint64 sys_sigreturn(void);

[SYS_sigalarm] sys_sigalarm,
[SYS_sigreturn] sys_sigreturn,
```

3. 在 `kernel/proc.h` 中添加与周期执行函数相关的变量。

```
// kernel/proc.h
struct proc {
  // ...
  int alarm_interval;       // Interval for the alarm handler
  void (*alarm_handler)();  // User-defined alarm handler
  int ticks;                // Timer ticks counter
  struct trapframe *saved_trapframe; // Save trapframe when executing alarm
handler
  int alarm_handling;       // Non-zero if alarm handler is being executed
};
```

4. 初始化进程时，对新增的变量进行初始化。

```
// kernel/proc.c
int allocproc(void) {
  // ...
  p->alarm_handler = 0;
  p->alarm_interval = 0;
  p->ticks = 0;
  p->alarm_handling = 0;
  if ((p->saved_trapframe = (struct trapframe *)kalloc()) == 0) {
    freeproc(p);
    release(&p->lock);
    return 0;
  }
  // ...
}
```

5. 修改 `kernel/trap.c` 中的 `usertrap()` 函数，实现周期性执行函数。

```
// kernel/trap.c
void usertrap(void) {
  // ...
  if (r_scause() == 8) {
    // system call
    // ...
    else if ((which_dev = devintr()) != 0) {
      // ok
      if (which_dev == 2) {
        // timer interrupt
        p->ticks++;
        if (p->alarm_interval != 0 && p->ticks % p->alarm_interval == 0) {
          if (p->alarm_handling == 0) {
            p->alarm_handling = 1;
            // save original trapframe
            cpytrapframe(p->saved_trapframe, p->trapframe);
            // set return address to alarm_handler
            p->trapframe->epc = (uint64)p->alarm_handler;
          }
        }
      }
    }
    // ...
  }
  // ...
}
```

6. 实现 `sigalarm()` 和 `sigreturn()` 系统调用。

```
// kernel/sysproc.c
uint64 sys_sigalarm(void) {
  int interval;
  uint64 handler;
  if (argint(0, &interval) < 0 || argaddr(1, &handler) < 0)
    return -1;
  struct proc* proc = myproc();
  proc->alarm_interval = interval;
  proc->alarm_handler = (void (*)(void))handler;
  return 0;
}

uint64 sys_sigreturn(void) {
  struct proc* p = myproc();
  cpytrapframe(p->trapframe, p->saved_trapframe);
  p->alarm_handling = 0;
  return 0;
}
```

实验中遇到的问题和解决办法

- 问题：在实现周期性执行函数时，可能会遇到函数重入的问题，即周期性执行函数自身被递归调用导致程序异常。
- 解决办法：为了避免函数重入，可以在执行周期性执行函数之前设置一个标志位，当函数正在执行时，禁止再次调用该函数。可以在进程结构中添加一个 `alarm_handling` 变量来标志当前是否正在执行周期性执行函数，并在调用函数之前检查该标志位。在执行函数结束时，需要及时将标志位还原。

实验心得

通过这个实验，我们实现了一个周期性执行的函数，并在每次周期执行时调用用户指定的处理函数。这涉及到了对系统调用的设计和实现，以及如何在用户程序和内核之间进行数据传递和共享。我们在实现中注意了避免函数重入，确保每次执行处理函数时都能正确保存和恢复寄存器状态。这个实验对于我们理解操作系统的中断和定时器机制，以及系统调用的设计和实现都有很大的帮助。通过实现这三个小实验，我们对 `Xv6` 操作系统的底层机制和系统调用有了更深入的理解，并锻炼了在操作系统层面编程的能力。

评分

```
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (5.8s)
== Test running alarmtest ==
$ make qemu-gdb
(3.8s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (251.3s)
    (Old xv6.out.usertests failure log removed)
== Test time ==
time: OK
Score: 85/85
```