Lab 4: traps

1950787 杨鑫

1. 实验目的：

- 熟悉 trap，了解其执行过程
- 通过系统调用使用 trap 以实现一些功能

2. 实验步骤：

首先切换分支至 traps branch，以获取本次实验的内容

同时清理文件，以获得纯净的初始文件系统

```
git checkout traps
make clean
```

a. RISC-V assembly

实验目的：

了解一些 RISC-V 程序集很重要，您在 6.004 中接触过这些程序集。 在您的 xv6 存储库中有一个文件 user/call.c。 make fs.img 对其进行编译，并在 user/call.asm 中生成程序的可读汇编版本。

阅读 call.asm 中函数 g、f 和 main 的代码。 RISC-V 的说明手册在参考页上。 以下是您应该回答的一些问题：

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

   Answer：函数参数放在a0~a7，在这里，13这个参数被放在a2中。

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.)**

   Answer：没有函数调用的代码，在编译后函数直接内联到了相应地方。

3. **At what address is the function printf located?**

   Answer：printf的地址是0x630。

4. **What value is in the register ra just after the jalr to printf in main?**

   Answer：ra的值为0x38。

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 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.

   Answer：输出是He110 World。如果是大端存储的话，57616不变，i改为0x726c6400即可。

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);
   ```

   Answer：y=后面输出的值取决于当前a2寄存器的内容。因为3存放在a1寄存器，输出的第二个值应当放在a2寄存器，但是这里没有第二个参数，所以就取决于当前a2寄存器的值了。

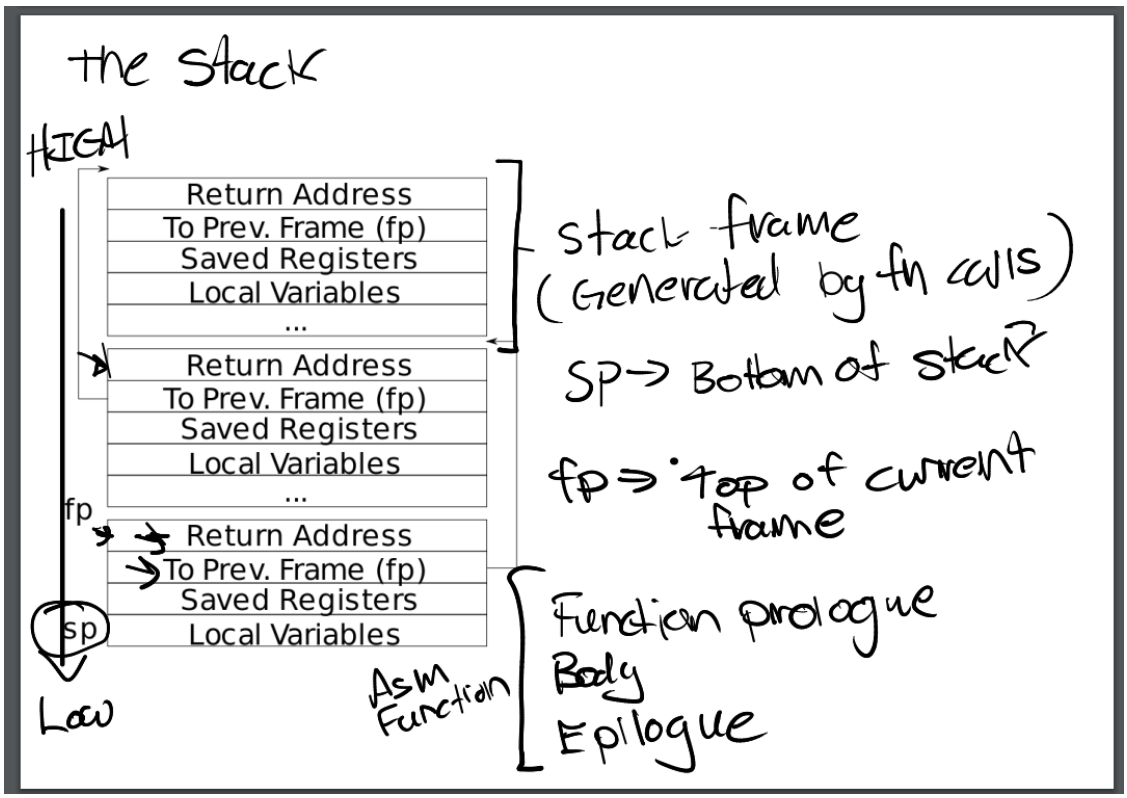
b. Backtrace

实验目的：

对于调试来说，回溯通常很有用：堆栈上发生错误的点上方的函数调用列表。
编译器在每个堆栈帧中放入一个帧指针，该指针保存调用者帧指针的地址。 您的回溯应该使用这些帧指针向上遍历堆栈并在每个堆栈帧中打印保存的返回地址。

实验分析：

返回地址位于距fp的固定偏移量 (-8) 处，而保存的fp（调用者的fp）位于距fp的固定偏移量 (-16) 处（如下图所示）。根据这个条件可以得到堆栈中的所有返回地址。



实验核心代码：

riscv.h

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

printf.c

```
void
backtrace(void)
{
  uint64 fp = r_fp();
  uint64 up = PGROUNDUP(fp);
  uint64 ra;
  printf("backtrace:\n");
  while(fp < up) {
    ra = *(uint64*)(fp - 8);
    printf("%p\n", ra);
    fp = *(uint64*)(fp - 16);
  }
}
```

sysproc.c

```
uint64
sys_sleep(void)
```

```
{
  int n;
  uint ticks0;

  if(argint(0, &n) < 0)
    return -1;
  acquire(&tickslock);
  ticks0 = ticks;
  while(ticks - ticks0 < n){
    if(myproc()->killed){
      release(&tickslock);
      return -1;
    }
    sleep(&ticks, &tickslock);
  }
  release(&tickslock);
  backtrace();
  return 0;
}
```

测试：

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ bttest
backtrace:
0x0000000080002e80
0x0000000080002ce2
0x0000000080002a3e
$
```

```
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ addr2line -e kernel/kernel
0x0000000080002e80
0x0000000080002ce2
0x0000000080002a3e
/home/yangxin/桌面/xv6-labs-2020/kernel/sysproc.c:74
/home/yangxin/桌面/xv6-labs-2020/kernel/syscall.c:144
/home/yangxin/桌面/xv6-labs-2020/kernel/trap.c:39
```

可以看到，依次打印出了调用 sys_sleep 的这些函数返回地址，说明测试成功

评分：

```
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ ./grade-lab-traps backtrac
e
make: “kernel/kernel”已是最新。
== Test backtrace test == backtrace test: OK (1.0s)
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$
```

可以通过所有测试

c. Alarm

实验目的：

在本练习中，您将向 xv6 添加一个功能，该功能会在进程使用 CPU 时间时定期提醒它。这对于想要限制它们占用多少 CPU 时间的计算绑定进程，或者对于想要计算但又想要采取一些定期操作的进程可能很有用。更一般地说，您将实现一种原始形式的用户级中断/故障处理程序；例如，您可以使用类似的东西来处理应用程序中的页面错误。

您应该添加一个新的 sigalarm(interval, handler) 系统调用。如果应用程序调用 sigalarm(n, fn)，那么在程序消耗的每 n 个 CPU 时间“滴答”之后，内核应该调用应用程序函数 fn。当 fn 返回时，应用程序应该从中断的地方继续。在 xv6 中，tick 是一个相当任意的时间单位，由硬件定时器产生中断的频率决定。如果应用程序调用 sigalarm(0, 0)，内核应停止生成定期警报调用。

您将在 xv6 存储库中找到文件 user/alarmtest.c。将其添加到 Makefile。在您添加 sigalarm 和 sigreturn 系统调用之前，它不会正确编译。

实验分析：

实验要求通过 trap 实现系统调用 sigalarm, sigreturn。可以实现在固定时间间隔不断执行一个函数，并且保证执行完指定的函数后可以正常回到原程序状态以继续之前的代码。可以考虑在每次 timer interrupt 的时候检测是否到达间隔时间，然后判断是否执行对应函数；然后还需要添加一些保存和恢复寄存器的代码，用于确保能够在函数执行完后可以正常回到原状态继续执行代码。

test0: invoke handler

实验核心代码：

proc.h

```
  uint64 interval;             // interval for sys_sigalarm
  void (*handler)();           // handler function for sys_alaram
  uint64 rest;                 // rest of interval
```

sysproc.c

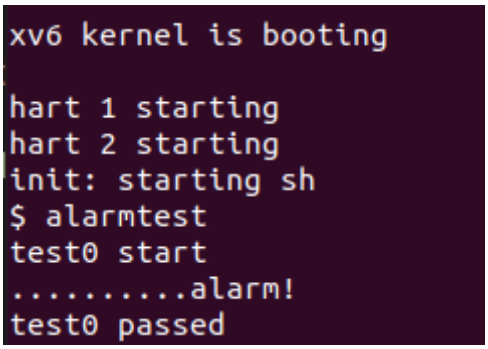
```
uint64
sys_sigalarm(void)
{
  struct proc *p = myproc();
  int n;
  uint64 handler;
  if (argint(0, &n) < 0) return -1;
  if (argaddr(1, &handler) < 0) return -1;
  p->interval = n;
  p->handler = (void(*)())handler;
  p->rest = n;
  return 0;
}

uint64
sys_sigreturn(void)
{
  return 0;
}
```

usertrap.c

```
//
// handle an interrupt, exception, or system call from user space.
// called from trampoline.S
//
void
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(p->killed)
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sstatus &c registers,
    // so don't enable until done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    if (which_dev == 2) {
      if (p->interval != 0) {
        p->rest--;
        if (p->rest == 0) {
          p->rest = p->interval;
          p->trapframe->epc = (uint64)p->handler;
        }
      }
    }
  } else {
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }

  if(p->killed)
    exit(-1);
```

```
  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret();
}
```

评分：



可以通过所有测试

test1/test2(): resume interrupted code

实验核心代码：

proc.h

```
  uint64 interval;             // interval for sys_sigalarm
  void (*handler)();           // handler function for sys_alaram
  uint64 rest;                 // rest of interval
  struct trapframe *trapframeInterrupt;
  int allowedCall;
```

allocproc.c

```
// Look in the process table for an UNUSED proc.
// If found, initialize state required to run in the kernel,
// and return with p->lock held.
// If there are no free procs, or a memory allocation fails, return 0.
static struct proc*
allocproc(void)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == UNUSED) {
      goto found;
    } else {
      release(&p->lock);
    }
  }
  return 0;
```

```
found:
  p->pid = allocpid();
  p->rest = p->interval;
  p->allowedCall = 1;

  // Allocate a trapframe page.
  if((p->trapframeInterrupt = (struct trapframe *)kalloc()) == 0){
    release(&p->lock);
    return 0;
  }

  if((p->trapframe = (struct trapframe *)kalloc()) == 0){
    release(&p->lock);
    return 0;
  }

  // An empty user page table.
  p->pagetable = proc_pagetable(p);
  if(p->pagetable == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }

  // Set up new context to start executing at forkret,
  // which returns to user space.
  memset(&p->context, 0, sizeof(p->context));
  p->context.ra = (uint64)forkret;
  p->context.sp = p->kstack + PGSIZE;

  return p;
}
```

trap.c

```
//
// handle an interrupt, exception, or system call from user space.
// called from trampoline.S
//
void
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();
```

```
  if(r_scause() == 8){
    // system call

    if(p->killed)
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sstatus &c registers,
    // so don't enable until done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    if (which_dev == 2 && p->allowedCall == 1) {
      if (p->interval != 0) {
        p->rest--;
        if (p->rest == 0) {
          switchTrapframe(p->trapframeInterrupt, p->trapframe);
          p->rest = p->interval;
          p->trapframe->epc = (uint64)p->handler;
          p->allowedCall = 0;
        }
      }
    }
  } else {
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }

  if(p->killed)
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret();
}

void
switchTrapframe(struct trapframe* trapframe, struct trapframe
*trapframeInterrupt)
{
  trapframe->kernel_satp = trapframeInterrupt->kernel_satp;
  trapframe->kernel_sp = trapframeInterrupt->kernel_sp;
  trapframe->epc = trapframeInterrupt->epc;
  trapframe->kernel_hartid = trapframeInterrupt->kernel_hartid;
  trapframe->ra = trapframeInterrupt->ra;
  trapframe->sp = trapframeInterrupt->sp;
  trapframe->gp = trapframeInterrupt->gp;
  trapframe->tp = trapframeInterrupt->tp;
  trapframe->t0 = trapframeInterrupt->t0;
  trapframe->t1 = trapframeInterrupt->t1;
  trapframe->t2 = trapframeInterrupt->t2;
```

```
  trapframe->t3 = trapframeInterrupt->t3;
  trapframe->t4 = trapframeInterrupt->t4;
  trapframe->t5 = trapframeInterrupt->t5;
  trapframe->t6 = trapframeInterrupt->t6;
  trapframe->s0 = trapframeInterrupt->s0;
  trapframe->s1 = trapframeInterrupt->s1;
  trapframe->s2 = trapframeInterrupt->s2;
  trapframe->s3 = trapframeInterrupt->s3;
  trapframe->s4 = trapframeInterrupt->s4;
  trapframe->s5 = trapframeInterrupt->s5;
  trapframe->s6 = trapframeInterrupt->s6;
  trapframe->s7 = trapframeInterrupt->s7;
  trapframe->s8 = trapframeInterrupt->s8;
  trapframe->s9 = trapframeInterrupt->s9;
  trapframe->s10 = trapframeInterrupt->s10;
  trapframe->s11 = trapframeInterrupt->s11;
  trapframe->a0 = trapframeInterrupt->a0;
  trapframe->a1 = trapframeInterrupt->a1;
  trapframe->a2 = trapframeInterrupt->a2;
  trapframe->a3 = trapframeInterrupt->a3;
  trapframe->a4 = trapframeInterrupt->a4;
  trapframe->a5 = trapframeInterrupt->a5;
  trapframe->a6 = trapframeInterrupt->a6;
  trapframe->a7 = trapframeInterrupt->a7;
}
```

sysproc.c

```
uint64
sys_sigreturn(void)
{
  struct proc* p = myproc();
  switchTrapframe(p->trapframe, p->trapframeInterrupt);
  p->allowedCall = 1;
  return 0;
}
```

测试：

```
xv6 kernel is booting

hart 1 starting
hart 2 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
$
```

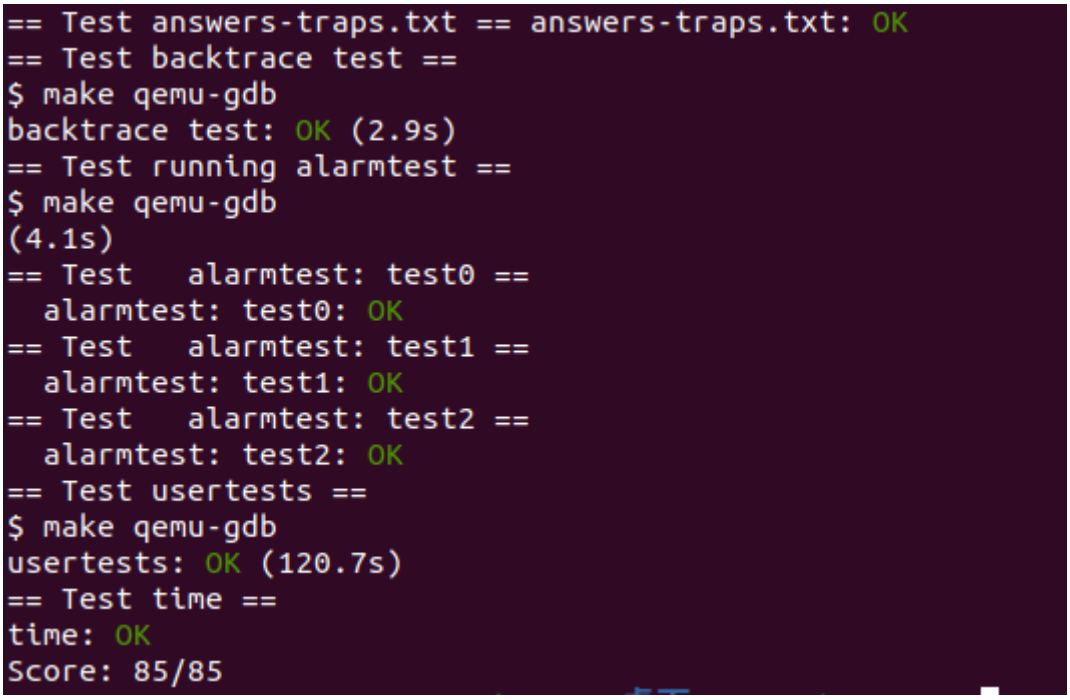
评分：

```
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ ./grade-lab-traps alarm
make: “kernel/kernel”已是最新。
== Test running alarmtest == (4.4s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
```

可以通过所有测试

3. 实验评分：

按照实验要求对本次实验的所有小实验进行评分，结果如图：



通过了所有测试

4. 问题以及解决办法：

a.如何在栈中获取函数的执行顺序以及相应信息

栈在执行函数时保存相应的内容。通过查阅文档，我知道了栈里面分为多个栈帧，且 xv6 的栈是从高到低扩展的，每个栈帧保留着返回地址、前一个栈帧的地址、局部变量和寄存器等信息。并且每个栈帧中，其返回地址和前一个栈帧的地址所在的位置都是固定的（偏移量分别为-8.-16）。于是我们可以根据当前栈帧找到其前方所有的栈帧信息。

b.如何在进程执行时跳转至另一个函数执行并且在执行完后回到原状态继续执行

实验中需要固定间隔执行指定的函数，但是这个函数不能影响正常进程的执行，所以需要确保函数执行完以后能够正常的返回原处及原始状态。可以在 proc 中加入一个数据结构用于保存指定的寄存器状态信息，这样在切换时就能确保不会干扰到原进程的正常执行。

5. 实验心得：

1. 本次实验了解了 trap 的实现机制，加深了理解。
2. 了解了操作系统中函数调用的栈的组织结构和包含的信息，对于函数调用过程有了新体会。
3. 学习了 trapframe 的结构，以及使用它来保存和恢复上下文，从而确保切换时，一些信息不会改变或者丢失。