Lab4 Traps 实验要求

一、实验简介

本次实验目的是探索系统调用是如何使用 trap 机制实现的。实验首先使用堆栈来进行一些简单的练习，之后需要实现一个用户态的 trap handling 实例。

二、实验准备

[1] 阅读 xv6 book 第 4 章 Traps and system calls，并阅读相关源代码：

- $kernel/trampoline.S$：从用户空间切换到内核空间并返回的汇编程序
- $kernel/trap.c$：处理所有中断的代码

[2] 准备 xv6 Lab4 源码：

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

三、实验要求

3.1 RISC-V assembly

首先需要了解一些 RISC-V 汇编语言。xv6 中有一个文件$user/call.c$。编译后，它在$user/call.asm$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：

https://pdos.csail.mit.edu/6.S081/2022/reference.html

以下是需要回答的问题：

（1） 函数的参数包含在哪些寄存器中？例如在 main 对 printf 的调用中，哪个寄存器保存 13？

（2） Main 的汇编代码中对函数 f 的调用在哪里？对 g 的调用在哪里？（Hint：编译器可能内联函数）

（3） 函数 printf 位于哪个地址？

（4） 在 jalr 到 main 中的 printf 之后，寄存器 ra 中存储的值是？

（5） 运行以下代码：

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

输出是什么？注：https://www.asciitable.com/是一个将字节映射到字符的 ASCII 表。

输出取决于 RISC-V 是 little-endian 的。如果 RISC-V 是 big-endian，怎样设置来产生相同的输出？是否需要更改 i57616 为不同的值？

参考链接：http://www.webopedia.com/TERM/b/big_endian.html；https://www.rfc-editor.org/ien/ien137.txt

（6） 在下面的代码中，会打印出什么？（注意：答案不是特定值）为什么会发生这种情况？

```
printf("x=%d y=%d", 3);
```

3.2 Backtrace

backtrace 对调试非常有用：它是 stack 中位于发生错误的点的上方的函数调用列表。编译器在每个 stack frame 中放入一个指向上一个 stack frame 的帧指针，即该指针保存 caller 的帧指针。寄存器 s0 保存当前 stack frame 的指针（实际上指向堆栈上保存的返回地址加上 8 的地址）。backtrace 应该使用这些帧指针向上遍历 stack 并在每个 stack frame 中打印保存的返回地址。

TODO：

在$kernel/printf.c$中实现 backtrace()函数。在 sys_sleep 中插入对该函数的调用。然后运行 bttest 来调用 sys_sleep。输出应该是如下格式的一串返回地址（值可能有区别）：

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

运行 bttest 后退出 qemu，在终端中运行 addr2line -e kernel/kernel（或 riscv64-unknown-elf-addr2line -e kernel/kernel），粘贴上面打印出的 backtrace 输出的地址，应该看到如下内容：

```
$ addr2line -e kernel/kernel
0x0000000080002de2
0x0000000080002f4a
0x0000000080002bfc
Ctrl-D

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

Hints：

- 将 backtrace 的函数原型添加到$kernel/defs.h$以便可以在 sys_sleep 中调用 backtrace
- GCC 编译器将当前执行的函数的帧指针存储在寄存器 s0 中。将以下函数添加到$kernel/riscv.h$：

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

  并在 backtrace 中调用此函数以读取当前帧指针。此函数使用内联汇编来读取 s0。
- https://pdos.csail.mit.edu/6.1810/2022/lec/l-riscv.txt 有一张 stack frame 布局的图片。请注意，**返回地址位于距当前 stack frame 的帧指针的固定偏移量（-8）处，而保存的帧指针位于距当前 stack frame 的帧指针的固定偏移量（-16）处。**

- xv6 为 xv6 内核中的每个 stack 在 PAGE 地址对齐处分配一个页面。可以使用 PGROUNDDOWN(fp)和 PGROUNDUP(fp)计算堆栈页面的顶部和底部地址。请参阅$kernel/riscv.h$。这些数字有助于 backtrace 终止其遍历。

一旦 backtrace 工作正常，就可以在$kernel/printf.c$中的 panic 调用它，这样你就可以看到 kernel 在 panic 时的 backtrace。

3.3 Alarm

本练习中将向 xv6 中添加一个功能，该功能会在进程使用 CPU 时定期提醒它。这对于想要限制它们占用 CPU 时间的计算密集型进程，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，你将实现一种原始形式的 user-level interrupt/fault handler；例如，你可以使用类似的东西来处理应用程序中的 page fault。

如果你的测试通过了 **alarmtest** 和'usertests -q'，那么结果就是正确的。

你应该添加一个新的 sigalarm(interval,handler)系统调用。如果一个应用程序调用了 sigalarm(n,fn)，那么在它消耗了 n 个 CPU "ticks"之后（在 n 个时钟中断后），将执行 handler 函数。当 handler 通过调用 sigreturn()返回时，应用程序应该从中断的地方继续。Tick 由硬件计时器产生中断的频率决定。如果应用程序调用 sigalarm(0,0)，则停止产生周期性调用。

xv6 存储库中有一个$user/alarmtest.c$文件。将其添加到 Makefile 中。只有添加了 sigalarm 和 sigreturn 系统调用才能正确编译。

alarmtest 在 test0 中调用 sigalarm(2,periodic)以要求内核在 2 个时钟中断后调用一次 periodic()。可以在$user/alarmtest.asm$中看到 alarmtest 的汇编代码。

Test0: invoke handler

首先修改内核以跳转到用户空间中的 alarm 处理程序，这将导致 test0 打印"alarm!"。

Hints：

1) 修改 Makefile，为 sigalarm 和 sigreturn 系统调用添加代码。
2) 现在，sys_sigreturn 应该只返回零。
3) sys_sigalarm()应该将 n 和 handler 存储在 proc 结构的新字段中（在$kernel/proc.h$中）。
4) 需要跟踪该进程自上次调用 alarm handler 以来已经过去了多少 ticks。为此，需要在 struct proc 中新建一个字段。可以在$proc.c$的 allocproc()中初始化这个字段。
5) 每个 tick 硬件时钟都会强制中断，该中断在$kernel/trap.c$的 usertrap()中处理。
6) 时钟中断是：if(which_dev == 2) ...
7) 需要修改 usertrap()以在进程的警报间隔到期时，让用户进程执行 handler。注意思考当 RISC-V 上的 trap 返回到用户空间时，是什么决定了用户空间代码恢复执行时的指令地址?
8) 可以首先运行 make CPUS=1 qemu-gdb 来告诉 qemu 只使用一个 CPU，这样

使用 gdb 查看 traps 会更容易。

Test1/Test2()/Test3(): resume interrupted code

此前在返回到用户态的时候，我们并未恢复寄存器的内容。因此 handler 完成后，当控制权返回到用户程序最初被中断的指令，必须确保寄存器内容恢复，以及重置警报计数器，以便定期调用 handler。

user alarm handler 需要在完成后调用 sigreturn 系统调用。我们可以将代码添加到 usertrap 和 sys_sigreturn 中，它们会协同工作以使用户进程在处理完警报后正确恢复。

Hints：

1) 确保正确地保存和恢复寄存器。
2) 当计时器到期时，让 usertrap 在 struct proc 中保存足够的状态，以便 sigreturn 可以正确返回到被中断的用户代码。
3) 防止对处理程序的重入调用——如果处理程序尚未返回，内核不应再次调用它。
4) 确保还原 a0。Sigreturn 是一个系统调用，它的返回值存储在 a0 中。

实验结果

全部完成后，输入 alarmtest 以及 usertest -q，结果应如下所示：

```
$ 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
$ usertest -q
...
ALL TESTS PASSED
$
```

五、实验提交

参考 PPT 中提交部分的详细说明