# Lab4 Traps

## RISC-V assembly

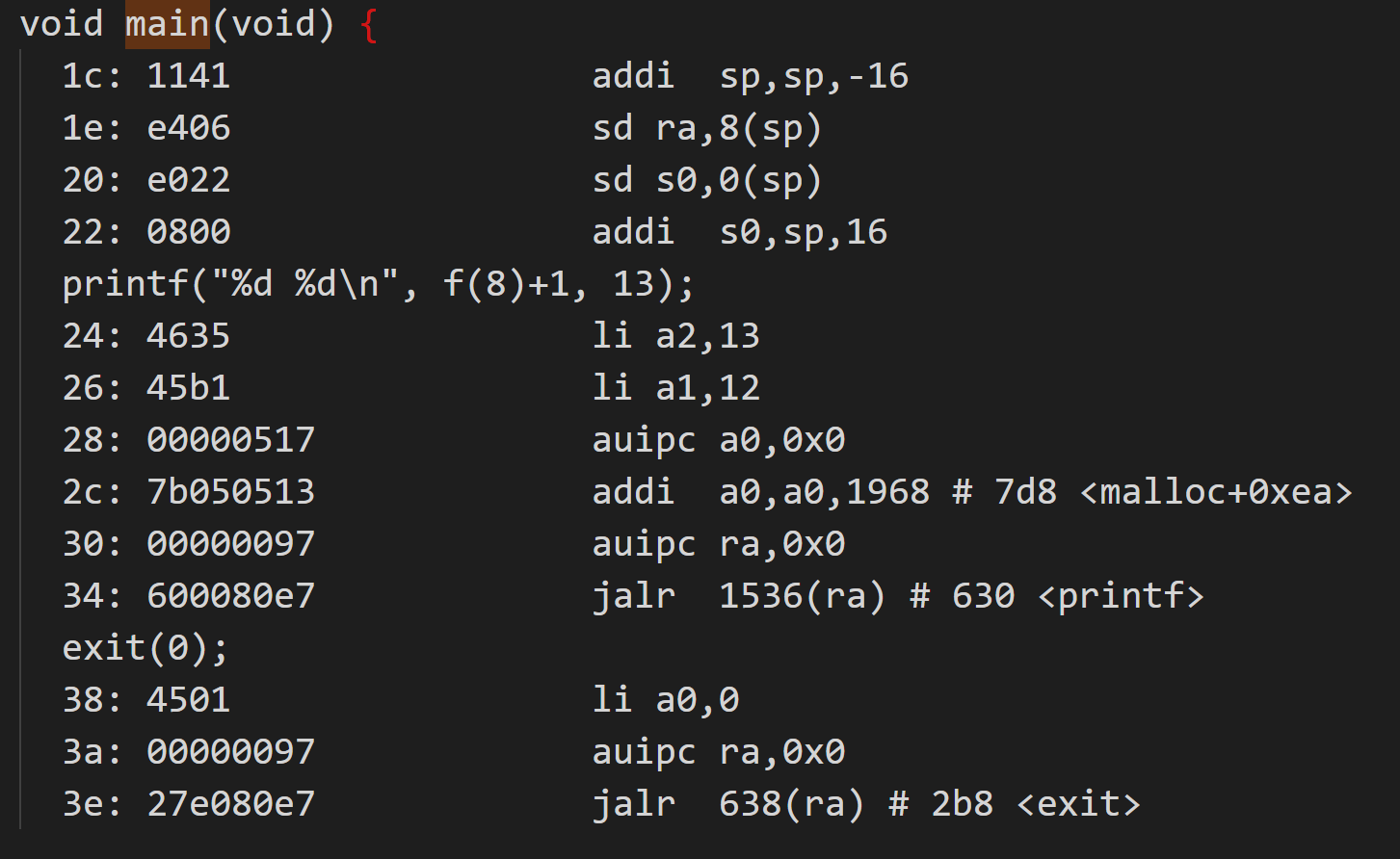
实验目的：

探讨如何使用陷阱实现系统调用。 您将首先对堆栈进行热身练习，然后实现用户级陷阱处理的示例。

实验步骤：

1.make fs.img

查看生成的call.asm中的f,g,main函数的代码



回答以下问题：

1）哪些寄存器包含函数的参数？ 例如，哪个寄存器在main对printf的调用中保留13？

a0 -a7保留了函数的参数，其中a0,a1还保留函数的返回值；

a2寄存器保留13。

2）在main的汇编代码中对函数f的调用在哪里？对函数g的调用在哪里？ （提示：编译器可以内联函数。）

没有这样的汇编代码，因为函数h被内联到函数f中，而函数f又进一步被内联到main函数中。

3)函数printf位于哪个地址？

0x630

4)在main中jalr到printf后，ra寄存器的值多少？

0x38

ra=pc+4=0x34+4=0x38

5)运行下面的代码,回答问题

unsigned int i = 0x00646c72;

printf("H%x Wo%s", 57616, &i);

输出是什么？

如果 RISC-V 是大端序的，要实现同样的效果，需要将 i 设置为什么？需要将 57616 修改为别的值吗？

输出是He110 World；需要将i设置为0x726c6400；57616 不需要修改。

％x 表示以十六进制数形式输出整数，57616 的16进制表示就是 e110，与大小端序无关。

%s 是输出字符串，以整数 i 所在的开始地址，按照字符的格式读取字符，直到读取到 ‘\0’ 为止。当是小端序表示的时候，内存中存放的数是：72 6c 64 00，刚好对应rld。当是大端序的时候，则反过来了，因此需要将 i 以16进制数的方式逆转一下。

6）在下面的代码中，之后会打印什么 'y='？ （注：答案不是具体值。）为什么会出现这种情况？

printf("x=%dy=%d", 3);

y输出的是一个受调用前代码影响的“随机”的值，因为printf尝试读的参数数量比提供的参数数量多，第二个参数‘3’通过a3传递，而第三个参数对应的寄存器a2在调用前不会被设置为任何具体的值，而是会包含调用发生的任何已经在里面的值。

### 实验中遇到的困难和解决办法

不知道函数中参数对应的意思：查实验提供的文献等资料

### 实验心得

实验指导书上说, 调试的时候指定一个CPU运行会更好一些，并且在其他资料的指导下学会了gdb调试，加快了自己理解汇编知识的速度，并且对于一些重复的指令，自己也学会了运用脚本的操作来节约时间。

## Backtrace

### 实验目的：

对于调试来说，有一个`backtrace`通常是有用的，这个函数可以在程序的某个error发生时，把函数调用栈上的一系列函数调用给打印出来。

本实验要在`kernel/printf.c`中实现`backtrace()`函数。

在`sys\_sleep()`中插入对这个函数的调用，然后运行`bttest`，它调用`sys\_sleep`。编写函数 `backtrace()`, 遍历读取栈帧(frame pointer)并输出函数返回地址.

### 实验步骤：

根据hints：

在def.h头文件下添加以下声明

void            printf(char\*, ...);

void            panic(char\*) \_\_attribute\_\_((noreturn));

void            printfinit(void);

void            backtrace(void);

在kernel/riscv.h中加入以下函数，以便backtrace能返回当前页指针

// add for backtrace

static inline uint64

r\_fp()

{

  uint64 x;

  asm volatile("mv %0, s0" : "=r" (x) );

  return x;

}

在kernel/printf.c中实现backtrace的具体逻辑

void backtrace(void){

  printf("backtrace:\n");

  uint64 fp = r\_fp();

  while (PGROUNDDOWN(fp) < PGROUNDUP(fp)){

    printf("%p\n", \*(uint64\*)(fp - 8));

    fp = \*(uint64\*)(fp - 16);

  }

}

最后加入到kernel/sysproc.c中的sys\_sleep即可

  int n;

  uint ticks0;

  backtrace();

  if(argint(0, &n) < 0)

    return -1;

  acquire(&tickslock);

### 实验中遇到的困难和解决办法

做实验时不明确距堆栈帧的帧指针的固定偏移量 (-8) 处所存数据值的具体含义是什么，后来通过查资料知道了s0放的是fp的地址，该地址-8即是返回地址（打印内容）的地址，-16就是上一个fp的地址。

要注意符号结合性的顺序，例如加号和强制转换

### 实验心得

栈的知识从数据结构就开始接触了，到操作系统能接触到一个更具体的栈，通过指针进行指引，主要记好该操作系统中指针怎么指向栈就能在函数的逻辑实现中游刃有余。

## Alarm

### 实验目的：

In this exercise you’ll add a feature to xv6 that periodically alerts a process as it uses CPU time. This might be useful for compute-bound processes that want to limit how much CPU time they chew up, or for processes that want to compute but also want to take some periodic action. More generally, you’ll be implementing a primitive form of user-level interrupt/fault handlers; you could use something similar to handle page faults in the application, for example. Your solution is correct if it passes alarmtest and usertests.

本实验中我们会向 xv6 添加一项功能，该功能会在进程使用 CPU 时间时定期发出警报。这对于想要限制其消耗的 CPU 时间的计算密集型进程，或者想要计算但也想要执行一些定期操作的进程可能很有用。

### 实验步骤：

1. 在makefile文件中加入以下代码

  $U/\_alarmtest\

1. 在user/user.h里添加声明

// add new alarm

int sigalarm(int ticks, void (\*handler)());

int sigreturn(void);

1. 仿照lab2，添加两个系统调用sigalarm与sigreturn：

syscall.h中增加

#define SYS\_sigreturn 23

syscall.c中增加

extern uint64 sys\_sigalarm(void);

extern uint64 sys\_sigreturn(void);

[SYS\_sigalarm] sys\_sigalarm,

[SYS\_sigreturn] sys\_sigreturn,

  // add for alarm

  int alarm\_interval;

  int alarm\_ticks;

  uint64 alarm\_handler;

  struct trapframe alarm\_trapframe;

1. 在kernel/proc.c的allocproc函数中初始化这些属性

// set alarm related parameters

  p->alarm\_interval = 0;

  p->alarm\_ticks = 0;

  p->alarm\_handler = 0;

1. 在kernel/trap.c的usertrap里处理interrupt

// give up the CPU if this is a timer interrupt.

  if(which\_dev == 2){

    if (p->alarm\_interval){

      if (++p->alarm\_ticks == p->alarm\_interval){

        memmove(&(p->alarm\_trapframe), p->trapframe, sizeof(\*(p->trapframe)));

        p->trapframe->epc = p->alarm\_handler;

      }

    }

    yield();

  }

  usertrapret();

1. 最后在kernel/sysproc.c中完善sys\_sigalarm和sys\_sigreturn函数

uint64 sys\_sigalarm(void){

  int interval;

  uint64 handler;

  if (argint(0, &interval) < 0)

    return -1;

  if (argaddr(1, &handler) < 0)

    return -1;

  myproc()->alarm\_interval = interval;

  myproc()->alarm\_handler = handler;

  return 0;

}

uint64 sys\_sigreturn(void){

  memmove(myproc()->trapframe, &(myproc()->alarm\_trapframe), sizeof(struct trapframe));

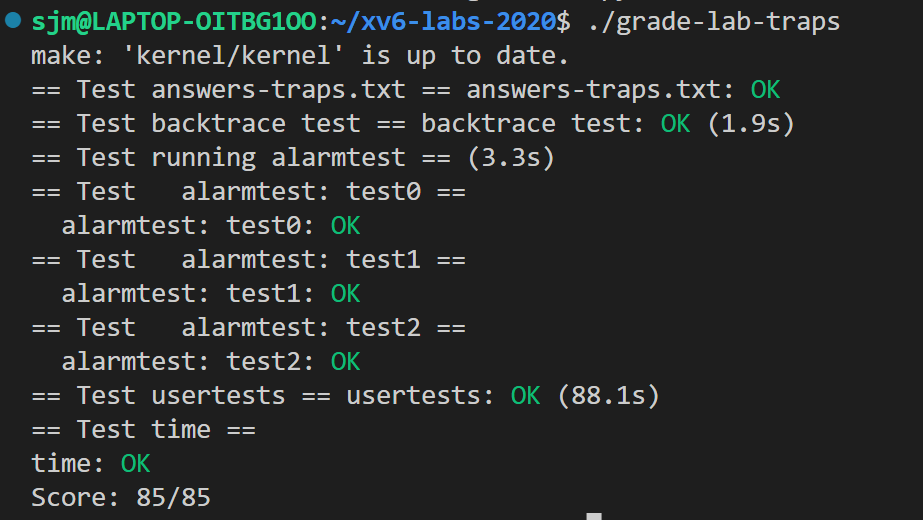
  myproc()->alarm\_ticks = 0;

  return 0;

}

在每次时钟中断的时候，如果进程有已经设置的时钟，则进行倒数。当alarm ticks 倒数到小于等于0的时候，如果没有正在处理的时钟，则尝试触发时钟，将原本的程序流保存起来，然后通过修改 pc 寄存器的值，将程序流转跳到handler中，handler执行完毕后再恢复原本的执行流。这样从原本程序执行流的视角，就是不可感知的中断了。

实验结果如下：



### 实验中遇到的问题和解决方法

当有东西没有被释放掉时usertests会报错丢失free page，查资料并大量printf调试后发现是save\_trapframe与trapframe之间的拷贝造成的。错误情况是直接将两个指针进行复制来拷贝，但只拷贝指针的话，指针之前指向的内容没被释放掉。

### 实验心得

再次实现系统调用，对于前面实验中实现系统调用的方法的印象更加深入了实验不是非常复杂但是以理解概念为重，trap机制、trampoline作用、函数calling convention、调用栈、特权模式、riscv汇编，这些即使都不知道可能依然能完成 lab。但是不代表这些不重要，相反这些才是主要内容，否则 lab 就算跑起来也只是无头苍蝇一般没有有真正达到学习效果，在完成实验后又去看了mit的教学视频lecture5,6，对这些知识的感悟更加深入了。