# 2023/09/18~2023/09/20

## XV6第四章

总的内容为枯燥且乏味的trap流程描述。但Trampoline是一个比较令人眼前一亮的设计。这也解答了我在pagetable实验时的困惑：内核和用户空间使用的trampoline在物理地址上确实是相同的，甚至他们的虚拟地址也是相同的；这使得在内核和用户态的切换不会导致任何错误。这也许解释了trampoline（蹦床）名字的由来，用户代码好像击中了蹦床一样，它会先在蹦床上完成一系列的力的交换，然后再被反弹到内核空间。

后续还有对lazy allocation，cow和mmap的讨论，但这是下一个lab的内容了。

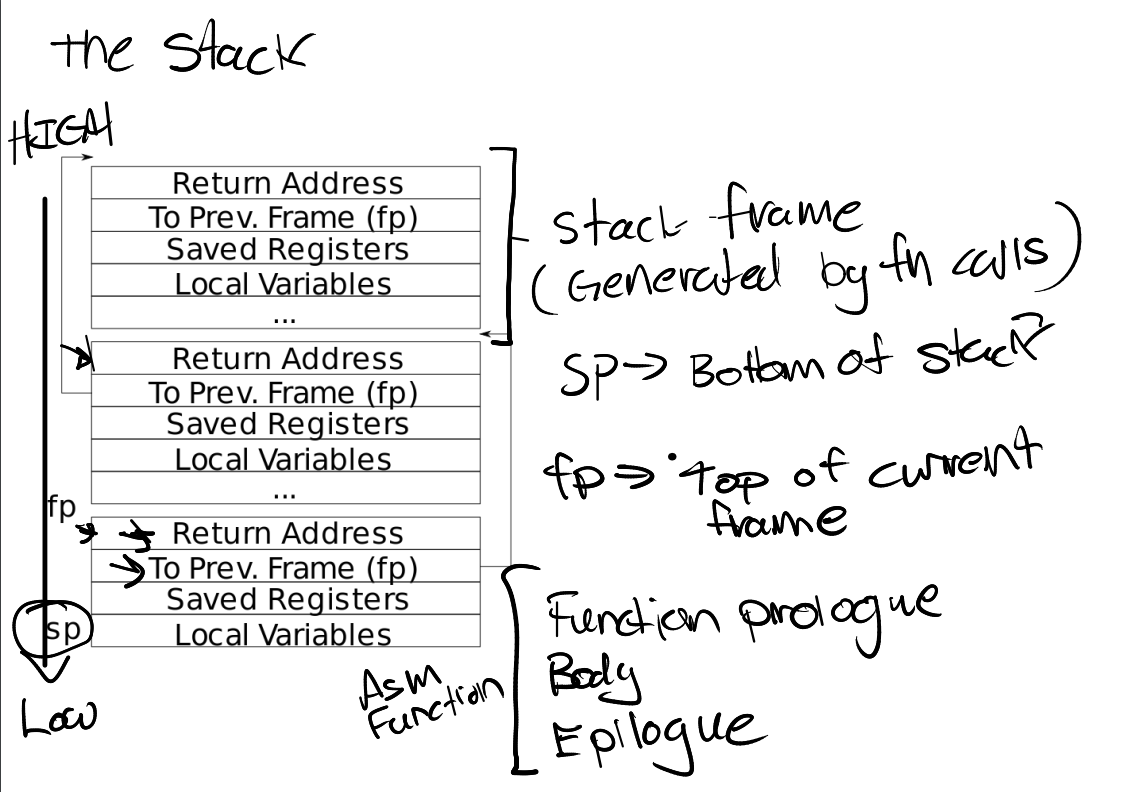
## Lab4 Traps

## RISC-V ASM

让回答一些简单的问题，查阅call.asm，阅读一点汇编代码即可回答。

## Backtrace

需要了解函数压栈的过程，其中两个寄存器存放的值需要注意，一个是ra中存放着return adress，也就是该函数被调用前的pc地址；一个是fp，存放着上一级函数的栈顶地址。其中ra位于栈顶，fp位于ra下方，分别占八个字节。在CPU寄存器中，fp寄存器中的值指向当前函数栈的栈顶。



于是ROUNDUP(fp)以获得用户栈的栈顶地址，作为fp溯源的边界。fp-8的值便是上一个函数的ra地址（即当前函数的调用地址），fp-16是上一个函数的fp地址。

代码如下：

void

backtrace(void){

  uint64 fp=r\_fp();

  uint64 stacktop=PGROUNDUP(fp);

  printf("backtrace:\n");

  while(fp<stacktop){

    uint64 ra=fp-8;

    printf("%p\n",\*(uint64\*)ra);

    fp=\*(uint64\*)(ra-8);

  }

}

## Alarm

对于test0而言，题目提供了非常友好详尽的提示。首先根据题目要求定义sigalarm和sigreturn系统调用，最初的sigreturn不实现任何功能，因为现阶段只需要成功“alarm!”即可，而不需要考虑进程alarm之后的运行状态。

在proc.h中添加成员rticks（已运行时间）与cticks（alarm周期时间），以及指向handler的函数指针，因为在sigalarm系统调用中获取指针地址时是以uint64\*的形式传参的，所以将handler的类型选择为uint64，后续调用时转换即可。

现在可以着手实现sigalarm的功能，实际上sigalarm只是起到一个初始化设置的功能，并不参与alarm本身。sigalarm通过获取系统调用传递的参数，分别给进程的cticks和handler赋值。

uint64

sys\_sigalarm(void){

  struct proc \*p=myproc();

  if(argint(0,&p->cticks)<0)

    return -1;

  if(argaddr(1,&p->handler)<0)

    return -1;

  return 0;

}

接下来实现在指定时钟中断时，进程转移到到handler执行的功能。在usertrap处理来自用户空间的trap时，首先会识别trap是否是因为系统调用，其次判断是否是来自于硬件中断，最后对无法归类的trap进行报错。

这里我们需要对时钟中断进行diy。在每次时钟中断的时候为当前进程的rticks++,并进行判断，如果rticks==cticks，那么将改变trapframe中保存的pc值（即中断产生时用户进程的pc位置）为handler的入口地址。

else if((which\_dev = devintr()) != 0){

    if(which\_dev==2){

        struct proc\* p=myproc();

        p->rticks++;

        if(p->rticks==p->cticks&&p->cticks!=0)

          p->trapframe->epc=p->handler;

    }

  }

这样一来，当进程从usertrap返回并回复用户寄存器的值时，pc已经在用户进程完全不知情的情况下被修改到了handler了，用户进程将从handler开始执行。

对于test1，要求在handler执行完毕之后返回到原本进程被时钟中断的位置，并恢复当时的上下文。

题目给了一个非常误导性的提示：*what registers do you need to save and restore to resume the interrupted code correctly? (Hint: it will be many).*

我想了很久，苦于不会用gdb调试时钟中断，而不知道该保存/恢复哪些寄存器。冥思苦想两三天，各种画图，企图用草稿纸验算的方式找出应该保存的寄存器，都没有结果。

直到9.24的早上我才顿悟，既然不知道哪些寄存器该保存，那就全部都保存再全部恢复不就行了吗？

在proc.h中定义一个alarm\_trapframe用于备份trapframe。在每次pc被修改为handler地址之前，都将trapframe拷贝在alarm\_trapframe中。后续在sigreturn时，再用这份备份恢复trapframe就能实现功能。这样一来，既不需要纠结epc，也不需要纠结变量值。真如本实验对于hard lab的描述，往往不需要太多代码，但实现方式都是很”tricky”的。

代码如下：

else if((which\_dev = devintr()) != 0){

    if(which\_dev==2){

        struct proc\* p=myproc();

        p->rticks++;

        if(p->rticks==p->cticks&&p->cticks!=0){

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

          p->trapframe->epc=p->handler;

        }

    }

  }

sys\_sigreturn(void){

  struct proc \*p=myproc();

  p->rticks=0;

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

  return 0;

}

同时，我也用很tricky的方式在一开始就实现了*Prevent re-entrant calls to the handler*的功能，即仅仅在sigreturn时才重置rtick。这样就实现了“只要handler还未返回，就不会有handler调用”，而不需要我去设置一个flag。

### 总结

Tricky，真的非常Tricky。