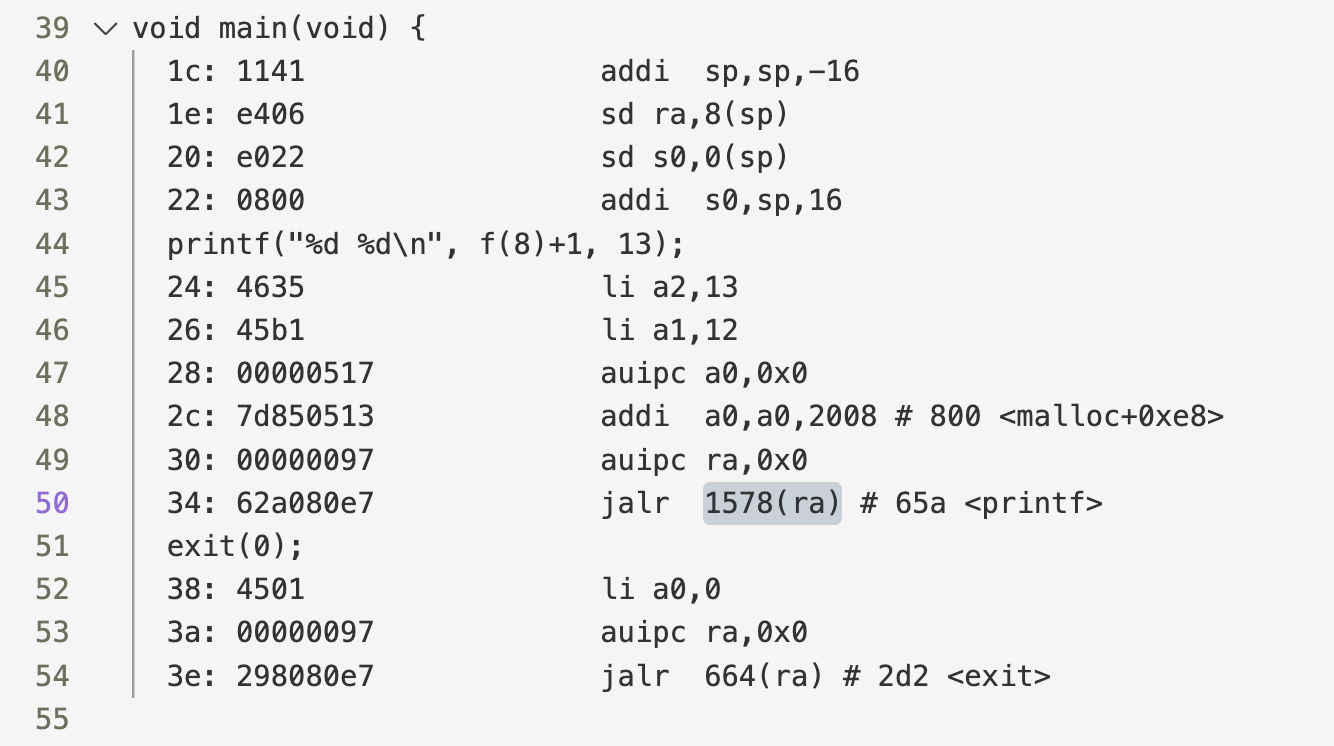
**Lab 4**

**Part A: RISC-V assembly**

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?**



参数存储在寄存器a0-a7中。在上述实验中，13被存储在寄存器a2中。

1. **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)**

如上图所示，Main函数不对f、g进行调用，直接引用f(8)+1的结果12。

1. **函数 printf 位于哪个地址?**

1578(ra) = 0x30 + 0x65a = 0x68a。

1. **在jalr到main中的printf之后，寄存器ra中存储的值是?**

ra存储return address，在以上实验中即存储main函数中断后代码的地址，为0x38。

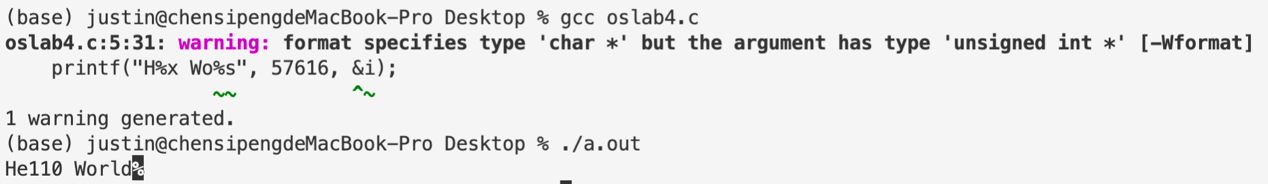
1. **运行以下代码:**

unsigned int i = 0x00646c72;

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

**输出是什么?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i, 57616 为不同的值?**

输出结果如下图所示，为H(e110) Wo(rld)。其中，e110为57616的十六进制表示，rld为i按照little-endian对应的ascii序列：little-endian中最低位字节存储在地址最低位处，因此与通常的阅读习惯相反，打印出的对应顺序为(72)(6c)(64)，即rld。

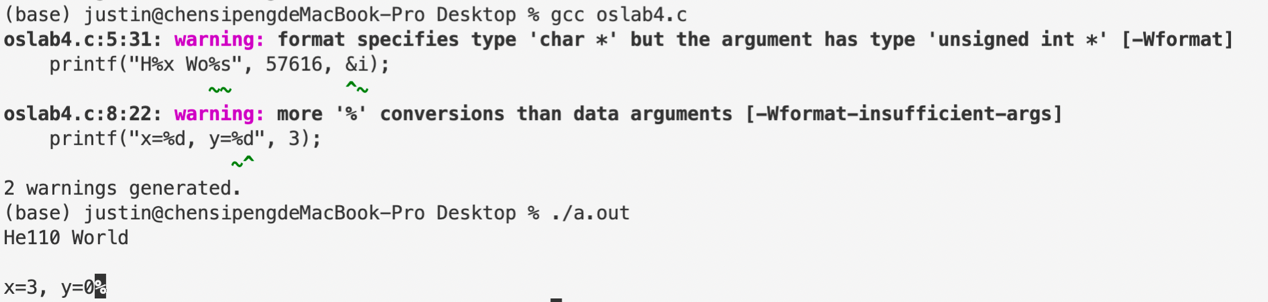


若采用big-endian，57616不需要修改，而i应当被改为0x726c6400。

1. **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);

输出结果如下。



编译器将直接从寄存器中读取先前存储的值。

**Part B: Backtrace**

按照题目提示在riscv.h中加入r\_fp()函数，backtrace()函数通过该函数获得当前帧指针后向上回溯能够分别获得返回地址以及先前保存的帧指针。根据题目要求通过标准输出打印出返回地址，并用保存的帧指针回溯到调用函数的页帧处，之后循环上述过程，直到遍历整个堆栈。代码如下：

void

**backtrace**(void)

{

**printf**("backtrace:\n");

**uint64** fp;

fp = **r\_fp**();

**uint64** top = **PGROUNDUP**(fp), bottom = **PGROUNDDOWN**(fp);

while(fp >= bottom && fp < top){

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

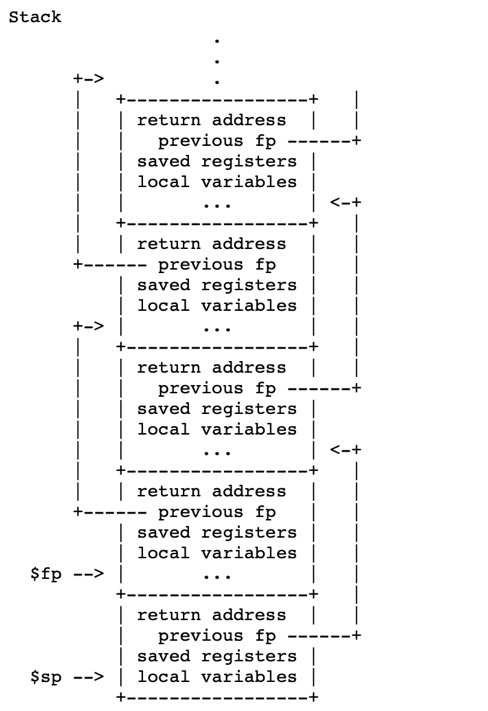
fp = \*((**uint64**\*)(fp - 16));

}

}

其中，通过PGROUNDUP和PGROUNDDOWN计算得到当前页的上/下限。另外，需要注意通过fp计算得到的返回地址与上一个帧指针均为指向地址的地址，为二级指针，因此在打印/回溯的过程中需要两次引用指针内容。完成backtrace的编写后，在sys\_sleep中加入命令backtrace()。

堆栈结构如下图所示。



最终运行结果如下。



**Part C: Alarm**

实现alarm的关键在于如何保存/重新获取中断前的信息，我们可以通过在proc结构体中定义新的变量以当前进程信息。在proc.h中添加以下定义信息：

int interval;

int n;

**uint64** handler;

struct **trapframe** \*alarmframe;

**uint64** return\_val;

其中，interval记录从上一次中断起至今的时间间隔；n表示用户输入的中断间隔时间；handler为中断调用函数句柄；alarmframe为指向结构体trapframe的指针，用来保存中断时正在运行的进程的相关寄存器信息；return\_val保存函数返回值。以上变量均在proc.c中allocproc()函数内，亦即进程被分配的时候初始化。

之后在sysproc.c最后添加函数sys\_sigalarm，sys\_sigreturn。sigalarm负责将用户输入的变量从寄存器中取出并存储到proc结构体的相关变量中；sigreturn部分负责将进程切换回中断之前的状态，通过memmove函数将存储在alarmframe中的寄存器信息重新载入到进程的trapframe中，并将计时器interval置零。另外，为了防止sigreturn更改函数原先的返回值，sigreturn函数本身的返回值应当被设为先前存储在进程中的return\_val变量。

**uint64**

**sys\_sigalarm**(void){

**argint**(0, &**myproc**()->n);

**argaddr**(1, &**myproc**()->handler);

return 0;

}

**uint64**

**sys\_sigreturn**(void){

*//extern char alarm\_frame[512];*

struct **proc** \*p = **myproc**();

**memmove**(p->trapframe, p->alarmframe, 512);

*//kfree(p->alarmframe);*

*//memset(p->alarmframe, 0, PGSIZE);*

p->interval = 0;

return p->return\_val;

}

随后，在trap.c中实现陷入过程的实现。首先增加计时器，随后对计时器的值做判断：若已经达到中断时长，则先将当前状态下的trapframe、函数返回值（即a0寄存器中存储值）存储值alarmframe、return\_val中，然后将当前进程的epc，即用户态下的PC切换为中断函数的句柄。

if(which\_dev == 2){

if (p->n != 0){

*//acquire(&p->lock);*

p->interval++;

if(p->interval == p->n){

*//p->alarmframe = (struct trapframe \*)kalloc();*

**memmove**(p->alarmframe, p->trapframe, **PGSIZE**);

p->return\_val = p->trapframe->a0;

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

}

*//release(&p->lock);*

}

完成以上内容后，xv6可以通过alarmtest，但运行usertests -q报错提示lost some free pages，经检查发现是由于释放进程时没有同时释放进程中存储的alarmframe。在proc.c中freeproc()部分加入以下代码后可以正常通过测试。

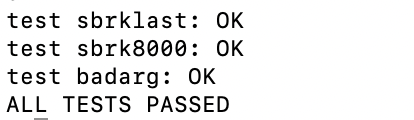
if (p->alarmframe)

**kfree**((void\*)p->alarmframe);

p->alarmframe = 0;

最终运行结果如下。





**Thoughts**

本次试验以对陷入机制的探索为主题，实现了对函数调用的回溯以及以周期性报警为主题的进程切换，加深了对进程、帧指针等概念的理解。