OS实验3 traps&lazy_alloction 实验报告

吕佳鸿 10235501436

Trap

Risc-V assembly

首先执行`make fs.img`得到call.asm文件如下：

```
int g(int x) {
   0: 1141                  addi  sp,sp,-16 // 将栈顶下移16个byte,为栈桢移出空间
   2: e422                  sd  s0,8(sp)
   4: 0800                  addi  s0,sp,16
  return x+3;
}
   6: 250d                  addiw a0,a0,3 // a0 + 3
   8: 6422                  ld  s0,8(sp)
   a: 0141                  addi  sp,sp,16 // 恢复sp的初始位置
   c: 8082                  ret // 返回a0

000000000000000e <f>:

int f(int x) { // 逻辑同g(x) 编译器可能做了优化，因为f(x)的逻辑就是调用g(x)
   e: 1141                  addi  sp,sp,-16
  10: e422                  sd  s0,8(sp)
  12: 0800                  addi  s0,sp,16
  return g(x);
}
  14: 250d                  addiw a0,a0,3
  16: 6422                  ld  s0,8(sp)
  18: 0141                  addi  sp,sp,16
  1a: 8082                  ret

000000000000001c <main>:

void main(void) {
  1c: 1141                  addi  sp,sp,-16 // 将栈顶下移16个byte,为栈桢移出空间
  1e: e406                  sd  ra,8(sp)
  20: e022                  sd  s0,0(sp)
  22: 0800                  addi  s0,sp,16
  printf("%d %d\n", f(8)+1, 13);
  24: 4635                  li  a2,13 // 将13 移到a2寄存器
  26: 45b1                  li  a1,12 // 将12移到a1 寄存器
  28: 00000517            auipc a0,0x0
  2c: 7a850513            addi  a0,a0,1960 # 7d0 <malloc+0x102>
```

```
  30: 00000097            auipc ra,0x0 // 设置返回地址 // ra = pc = 30
  34: 5e6080e7            jalr  1510(ra) # 616 <printf>  // 调用printf ra + 1510 = 0x616
  exit(0);
  38: 4501                  li  a0,0
  3a: 00000097            auipc ra,0x0
  3e: 274080e7            jalr  628(ra) # 2ae <exit>
```

Q1:哪些寄存器用来存放函数所用的参数？比如在main函数中调用printf时13这个参数是在哪个寄存器中传递?

a0-a7用来存放函数所用的参数 ， 在main函数中13由寄存器a2保存

Q2 `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?

在C的代码中，主函数main调用f，f调用g。而在生成的汇编中，main函数进行了内联优化处理。

从代码 `li a1,12` 可以看出，main直接计算出了结果并储存

Q3 `printf` 函数位于哪个地址?

在0x616, ra = 0x30 , ra + 1510 = 0x30 + 0x5e6 = 0x616

Q4在 `main` 中 `printf` 的 `jalr` 之后的寄存器 `ra` 中有什么值?

0x38

Backtrace

这个函数就是实现曾经调用函数地址的回溯

首先根据提示将 `r_fp` 添加到riscv.h中，GCC编译器将当前正在执行的函数的帧指针保存在 `s0` 寄存器这个函数使用内联汇编来读取 `s0`

返回地址位于栈帧帧指针的固定偏移(-8)位置，并且保存的帧指针位于帧指针的固定偏移(-16)位置

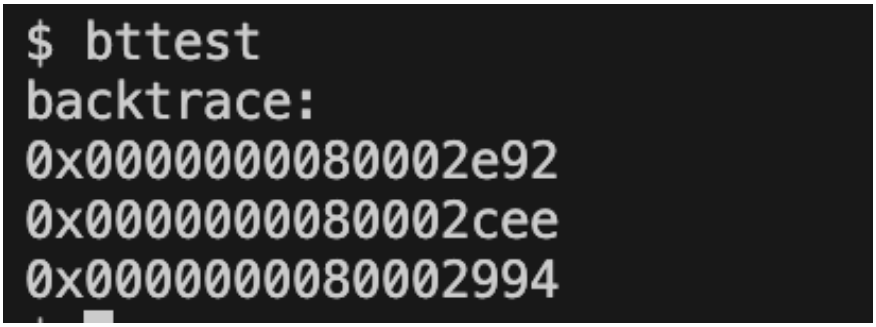
并且注意XV6在内核中以页面对齐的地址为每个栈分配一个页面。可以通过 `PGROUNDDOWN(fp)` 和`PGROUNDUP(fp)来计算栈页面的顶部和底部地址

实现如下：

```
void
backtrace(void) {
  printf("backtrace:\n");
  // 读取当前帧指针
  uint64 now_fp = r_fp();
  while (PGROUNDUP(now_fp) - PGROUNDDOWN(now_fp) == PGSIZE) {
    // 返回地址偏移量是8
    uint64 ret_addr = *(uint64*)(now_fp - 8);

    printf("%p\n", ret_addr);
    // 帧指针保存在-16偏移的位置
    now_fp = *(uint64*)(now_fp - 16);
```

```
  }
}
```



Alarm

这个练习是向XV6添加一个特性，在进程使用CPU的时间内，XV6定期向进程发出警报。这对于那些希望限制CPU时间消耗的受计算限制的进程，或者对于那些计算的同时执行某些周期性操作的进程可能很有用。更普遍的来说，是实现用户级中断/故障处理程序的一种初级形式

首先根据提示，添加了`sigalarm`和`sigreturn`系统调用后才能正确编译

在`user.h`中添加函数声明

```
int sigalarm(int ticks, void (*handler)());
int sigreturn(void);
```

在`def.s`中添加系统调用号

```
#define SYS_sigalarm    22
#define SYS_sigreturn   23
```

在`usys.pl`中添加接口

```
entry("sigalarm");
entry("sigreturn");
```

在`syscall.c`中添加声明

```
[SYS_sigalarm]    sys_sigalarm,
[SYS_sigreturn]   sys_sigreturn,

extern uint64 sys_sigalarm(void);
extern uint64 sys_sigreturn(void);
```

并将`alarmtest.c`添加到Makefile中去

```
$U/_alarmtest\
```

然后在`proc.h`中添加字段，作用如注释所示，并在`allocproc`中初始化为0,并在`freeproc`中释放

```
int alarm_interval;          // 报警间隔
  void (*alarm_handler)();     // 报警处理
  int ticks_num;               // 两次报警间的滴答计数
  int is_alarm   ;             // 是否警告
  struct trapframe* alarm_trapframe;  // 警告陷阱帧
```

```
if((p->alarm_trapframe = (struct trapframe*)kalloc()) == 0) {
    freeproc(p);
    release(&p->lock);
    return 0;
  }
  p->is_alarm = 0;
  p->alarm_interval = 0;
  p->alarm_handler = 0;
  p->ticks_num = 0;
```

没有alarm时trap运行的大致过程

1. 进入内核空间，保存用户寄存器到进程陷阱帧
2. 陷阱处理过程
3. 恢复用户寄存器，返回用户空间

而当添加了alarm后，变成了以下过程

1. 进入内核空间，保存用户寄存器到进程陷阱帧
2. 陷阱处理过程
3. 恢复用户寄存器，返回用户空间，但此时返回的并不是进入陷阱时的程序地址，而是处理函数`handler`的地址，而`handler`可能会改变用户寄存器

因此我们要在`usertrap`中再次保存用户寄存器，当`handler`调用`sigreturn`时将其恢复，并且要防止在`handler`执行过程中重复调用

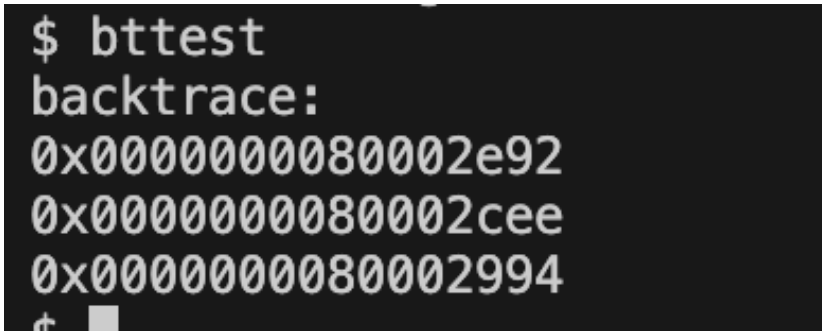
更改usertrap函数，保存进程陷阱帧`p->trapframe`到`p->alarm_trapframe`

```
// give up the CPU if this is a timer interrupt.
if(which_dev == 2) {
  if(p->alarm_interval != 0 && ++p->ticks_count == p->alarm_interval && p->is_alarming == 0)
{
    // 保存寄存器内容
    memmove(p->alarm_trapframe, p->trapframe, sizeof(struct trapframe));
    // 更改陷阱帧中保留的程序计数器，注意一定要在保存寄存器内容后再设置epc
    p->trapframe->epc = (uint64)p->alarm_handler;
    p->ticks_count = 0;
    p->is_alarming = 1;
  }
  yield();
}
```

更改`sys_sigreturn`，恢复陷阱帧

```
uint64
sys_sigreturn(void) {
  memmove(myproc()->trapframe, myproc()->alarm_trapframe, sizeof(struct trapframe));
  myproc()->is_alarming = 0;
  return 0;
}
```

trap's result



```
$ 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
```

Lazy allocation

这个实验是实现一个内存页懒分配机制，在调用 sbrk() 的时候，不立即分配内存，而是只作记录。在访问到这一部分内存的时候才进行实际的物理内存分配。

Eliminate allocation from sbrk()

这个实验需要删除`sys_sbrk`系统调用中的页面分配代码 sbrk系统调用将进程的内存大小增加n个字节，然后返回新分配区域的开始部分（即旧的大小）。新的 sbrk应该只将进程的大小（`myproc()->sz`）增加n，然后返回旧的大小。不应该分配内存——因此应该删除对`growproc()`的调用,但是仍然需要增加进程的大小。

实现如下：

```
uint64
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;

  addr = myproc()->sz;
  // lazy allocation
  myproc()->sz += n;

  return addr;
}
```

Lazy allocation

这个实验需要修改trap.c中的代码以响应来自用户空间的页面错误

修改usertrap函数，使用r_scause(判断是否为页面错误，在页面错误处理的过程中，先判断发生错误的虚拟地址（r_stval (读取）是否位于栈空间之上，进程大小（虚拟地址从0开始，进程大小表征了进程的最高虚拟地址）之下，然后分配物理内存并添加映射

```
else if(r_scause() == 13 || r_scause() == 15) {
    // 处理页面错误
    uint64 fault_va = r_stval();  // 产生页面错误的虚拟地址
    char* pa;                     // 分配的物理地址
    if(PGROUNDUP(p->trapframe->sp) - 1 < fault_va && fault_va < p->sz &&
      (pa = kalloc()) != 0) {
        memset(pa, 0, PGSIZE);
        if(mappages(p->pagetable, PGROUNDDOWN(fault_va), PGSIZE, (uint64)pa, PTE_R | PTE_W |
PTE_X | PTE_U) != 0) {
          kfree(pa);
          p->killed = 1;
        }
      }
```

修改uvmunmap()之所以修改这部分代码是因为lazy allocation中首先并未实际分配内存，所以当解除映射关系的时候对于这部分内存要略过，而不是使系统崩溃

```
for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
    if((pte = walk(pagetable, a, 0)) == 0)
      panic("uvmunmap: walk");
    if((*pte & PTE_V) == 0)
      continue;
```

Lazytests and Usertests

处理`sbrk()`参数为负数的情况，参考之前`sbrk()`调用的`growproc()`程序，如果为负数，就调用`uvmdealloc()`函数，但需要限制缩减后的内存空间不能小于0

```
uint64
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
```

```
    return -1;

  struct proc* p = myproc();
  addr = p->sz;
  uint64 sz = p->sz;

  if(n > 0) {
    // lazy allocation
    p->sz += n;
  } else if(sz + n > 0) {
    sz = uvmdealloc(p->pagetable, sz, sz + n);
    p->sz = sz;
  } else {
    return -1;
  }
  return addr;
}
```

uvmunmap是在释放内存时调用的，由于释放内存时，页表内有些地址并没有实际分配内存，因此没有进行映射。如果在 uvmunmap中发现了没有映射的地址，直接跳过就行，不需要 panic：

```
void
uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
  uint64 a;
  pte_t *pte;

  if((va % PGSIZE) != 0)
    panic("uvmunmap: not aligned");

  for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
    if((pte = walk(pagetable, a, 0)) == 0)
      //panic("uvmunmap: walk");
      continue;
    if((*pte & PTE_V) == 0)
      continue;
      //panic("uvmunmap: not mapped");
    if(PTE_FLAGS(*pte) == PTE_V)
      panic("uvmunmap: not a leaf");
    if(do_free){
      uint64 pa = PTE2PA(*pte);
      kfree((void*)pa);
    }
    *pte = 0;
  }
}
```

由于 read/write 等系统调用时，由于进程利用系统调用已经到了内核中，页表已经切换为内核页表，无法直接访问虚拟地址。因此，需要通过 walkaddr 将虚拟地址翻译为物理地址。这里如果没找到对应的物理地址，就分配一个

```
uint64
walkaddr(pagetable_t pagetable, uint64 va)
{
  pte_t *pte;
  uint64 pa;

  if(va >= MAXVA)
    return 0;

  pte = walk(pagetable, va, 0);
  //if(pte == 0)
  //  return 0;
  //if((*pte & PTE_V) == 0)
  //  return 0;
  if (pte == 0 || (*pte & PTE_V) == 0) {
    //pa = lazyalloc(va);
    struct proc *p = myproc();
    if(va >= p->sz || va < PGROUNDUP(p->trapframe->sp)) return 0;
    pa = (uint64)kalloc();
    if (pa == 0) return 0;
    if (mappages(p->pagetable, va, PGSIZE, pa, PTE_W|PTE_R|PTE_U|PTE_X) != 0) {
      kfree((void*)pa);
    return 0;
    }
  return pa;
  }
  if((*pte & PTE_U) == 0)
    return 0;
  pa = PTE2PA(*pte);
  return pa;
}
```

修改usertrap()函数，使用r_scause()判断是否为页面错误，在页面错误处理的过程中，先判断发生错误的虚拟地址（`r_stval()`读取）是否位于栈空间之上，进程大小（虚拟地址从0开始，进程大小表征了进程的最高虚拟地址）之下，然后分配物理内存并添加映射

```
else if((which_dev = devintr()) != 0) {
    // ok
  } else if(cause == 13 || cause == 15) {
    // 处理页面错误
    uint64 fault_va = r_stval();  // 产生页面错误的虚拟地址
    char* pa;                     // 分配的物理地址

    if(PGROUNDUP(p->trapframe->sp) - 1 < fault_va && fault_va < p->sz &&
```

```
    if(PGROUNDDOWN(p->trapframe->sp) - 1 < fault_va && fault_va < p->sz &&
      (pa = kalloc()) != 0) {
        memset(pa, 0, PGSIZE);
        if(mappages(p->pagetable, PGROUNDDOWN(fault_va), PGSIZE, (uint64)pa, PTE_R | PTE_W |
PTE_X | PTE_U) != 0) {
          kfree(pa);
          p->killed = 1;
        }
    } else {
      // printf("usertrap(): out of memory!\n");
      p->killed = 1;
    }
```

Result

```
== Test   usertests: fourteen ==
  usertests: fourteen: OK
== Test   usertests: bigfile ==
  usertests: bigfile: OK
== Test   usertests: dirfile ==
  usertests: dirfile: OK
== Test   usertests: iref ==
  usertests: iref: OK
== Test   usertests: forktest ==
  usertests: forktest: OK
== Test time ==
time: OK
Score: 119/119
```