Report for lab3, Yichen Mao

This Complete The Lab.

Part A : User Environments and Exception Handling

Exercise 1

Task: 修改 kern/pmap.c 中的 mem_init() 函数实现为 envs 分配空间以及地址映射。

```
用 boot_alloc 函数申请至少 sizeof(struct Env) * NENV 空间并清空。
envs = (struct Env*)boot_alloc(sizeof(struct Env) * NENV);
memset(envs, 0 ,sizeof(struct Env) * NENV);

用 boot_map_region 函数实现 envs 映射到 UENVS，权限为用户只读
boot_map_region(kern_pgdir,UENVS,PTSIZE,PADDR(envs),PTE_U);
```

Exercise 2

Task:完善 env.c 中部分函数。

```
env_init 函数主要是初始化 env 列表，将所有的 env_id 设置为 0 ，env_status 设置为 FREE 。同时构建空闲环境的链表结构，且将 env_free_list 设置为链表头。
注：特别要求链表顺序要与 env 结构体顺序相同。
void
env_init(void)
{
  assert(env_free_list == NULL);
  for (int i = NENV - 1 ; i>=0; i--){
    envs[i].env_id = 0;
    envs[i].env_status = ENV_FREE;
    envs[i].env_link = env_free_list;
    env_free_list = &envs[i];
  }
  env_init_percpu();
}
```

```
env_setup_vm 函数就是设置环境的地址映射，首先申请一个页面作为当前环境的 page director，根据提示我们知道 UTOP 以上的部分多数其实是相同的，直接把 kern_pgdir 直接拿过来用就可以了，特别注意的是要修改 UVPT 起始对应页面 的地址映射到当前的 page director 地址。
注：一般只有 UTOP 以上的虚拟页面映射的物理页一般不用维护引用数，（应该是这些页面一般会一直存在），但是当前环境的 page director 可能会因为环境切换而删除，所以仍要维护当前页面的引用数。
static int
```

```
env_setup_vm(struct Env *e)
{
  int i;
  struct PageInfo *p = NULL;

  if (!(p = page_alloc(ALLOC_ZERO)))
    return -E_NO_MEM;

  e->env_pgdir = (pde_t*)page2kva(p);
  p->pp_ref++;

  for (int i = 0; i < PDX(UTOP); i++){
    e->env_pgdir[i] = 0;
  }
  for (int i = PDX(UTOP); i < NPDENTRIES; i++){
    e->env_pgdir[i] = kern_pgdir[i];
  }
  e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
  return 0;
}
```

```
region_alloc 函数 为环境建立地址映射，注意点不多
1.手动对齐
2.清空 page_alloc 的 alloc_flag，保证不对申请的页面清空。
3.设置权限位。

static void
region_alloc(struct Env *e, void *va, size_t len)
{
  void *start = ROUNDDOWN(va,PGSIZE);
  void *end =ROUNDUP(va+len,PGSIZE);
  for (void *address = start; address < end; address += PGSIZE){
    struct PageInfo *page = page_alloc(0);
    if (page == NULL)panic("region_alloc: page_alloc failed!");
    if (page_insert(e->env_pgdir,page,address,PTE_W|PTE_U))
      panic("region_alloc: page_insert failed!");
  }
}
```

```
load_icode 函数 从ELF二进制映像中的所有可加载段加载到环境的用户内存中

首先要判断给定的 binary 地址对应的数据是一个合法的 ELF Header
```

如果是合法的，设置程序的入口并将并通过 lcr3 指令切换当前环境所对应的页目录。

类似 boot/main.c 通过 Header 找到起始段和最后一个段的地址。对于每一个段 memsz 存储的是段的大小，filesz 存储的是文件大小，va 是虚拟地址。
要求 段是可加载的且段的大小要大于文件大小，将 [binary + ph->p_offset,binary + ph->p_offset + ph->p_filesz) 复制到
虚拟内存 [ph->p_va,ph->p_va + ph->p_filesz)，并将其余清空。
最后再申请一个栈的空间。

```
static void
load_icode(struct Env *e, uint8_t *binary)
{
  struct Elf *ELFHDR = (struct Elf *)binary;
  if (ELFHDR->e_magic != ELF_MAGIC)
    panic("load_icode: ELFHDR->e_magic != ELF_MAGIC\n");

  e->env_tf.tf_eip = ELFHDR->e_entry;
  lcr3(PADDR(e->env_pgdir));

  struct Proghdr *ph = (struct Proghdr *)((uint8_t *)ELFHDR + ELFHDR->e_phoff);
  struct Proghdr *eph = ph + ELFHDR->e_phnum;
    for (; ph < eph; ph++){
    #ifdef DEBUG
      cprintf("memory size: %x\nfile size: %x\nvirtual address: %x\noffset: %x\n\n",ph->p_memsz,ph->p_filesz,ph->p_va,ph->p_offset);
    #endif
    if (ph->p_type == ELF_PROG_LOAD){
      assert(ph->p_memsz >= ph->p_filesz);
      region_alloc(e,(void *)ph->p_va,ph->p_memsz);
      memset((void *)ph->p_va, 0, ph->p_memsz);
      memcpy((void *)ph->p_va, binary + ph->p_offset, ph->p_filesz);
    }
  }
  region_alloc(e,(void *)(USTACKTOP-PGSIZE), PGSIZE);
}
```

env_alloc(struct Env **newenv_store, envid_t parent_id) 函数会申请一个新的环境将其地址放在 newenv_store，父环境设为parent_id。

函数 env_create 先申请新的环境，并加载 elf 文件和设置环境类型。

```
void
env_create(uint8_t *binary, enum EnvType type)
{
  struct Env *e;
    if (env_alloc(&e, 0) != 0)
          panic("env_create: fail!\n");

     load_icode(e, binary);
     e->env_type = type;
}
```

```
为什么会跳到 0x00800020

void
env_run(struct Env *e)
{
  // Step 1: If this is a context switch (a new environment is
running):
  //     1. Set the current environment (if any) back to
  //        ENV_RUNNABLE if it is ENV_RUNNING (think about
  //        what other states it can be in),
  //     2. Set 'curenv' to the new environment,
  //     3. Set its status to ENV_RUNNING,
  //     4. Update its 'env_runs' counter,
  //     5. Use lcr3() to switch to its address space.
  // Step 2: Use env_pop_tf() to restore the environment's
  //     registers and drop into user mode in the
  //     environment.

  // Hint: This function loads the new environment's state from
  //  e->env_tf.  Go back through the code you wrote above
  //  and make sure you have set the relevant parts of
  //  e->env_tf to sensible values.

  // LAB 3: Your code here.
  if(curenv != NULL && curenv->env_status == ENV_RUNNING)
        curenv->env_status = ENV_RUNNABLE;
  // cprintf("env_run: %x %x\n",curenv,curenv->env_tf.tf_eip);
    curenv = e;
    curenv->env_status = ENV_RUNNING;
    curenv->env_runs++;
    lcr3(PADDR(curenv->env_pgdir));
```

```
31    // cprintf("env_run: %x %x\n",curenv,curenv->env_tf.tf_eip);
32      env_pop_tf(&curenv->env_tf);
33    panic("env_run not yet implemented");
34  }
```

Exercise 3

学习 Exceptions and Interrupts。

9.1 Interrupts and Exceptions

异常和中断是控制转移方式，中断是用来处理异步事件 (external to the processor)。而异常用来处理情况 (detected by the processor)。

1. 中断
    - 可屏蔽中断，signalled via the INTR pin
    - 不可屏蔽中断，signalled via the NMI pin

2. 异常
    - 处理器检测到的，faults traps aborts。
    - 编程的， INT

9.2 Enabling and Disabling Interrupts

NMI 处理过程中会忽略所有 NMI pin 上的信号，直到 IRET 指令。

IF 标志位控制 INTR，IF = 0 INTR中断屏蔽。用 CLI 和 STI 设置 IF 标志位。

9.3 Priority Among Simultaneous Interrupts and Exceptions

优先级表

```
HIGHEST    Faults except debug faults
Trap instructions INTO, INT n, INT 3
Debug traps for this instruction
Debug faults for next instruction
NMI interrupt
LOWEST     INTR interrupt
```

高优先级先处理，忽略低的异常（会回到指令再产生一次），挂起低的中断。

9.4 Interrupt Descriptor Table

IDT ，一个表项 8 byte，第一个条目可以有描述符。

用 IDTR(IDT 寄存器) 找到 IDT ，用异常号 $\times$ 8 作索引

LIDT (特权0) 初始化创建时使用，需要线性基地址和限制。

SIDT (无特权)，把IDTR分到另一个内存地址。

9.5 IDT Descriptors

IDT 描述符有三种任务门 中断门 陷阱门

9.6 Interrupt Tasks and Interrupt Procedures

索引到中断表述符

中断门或陷阱门 handler

任务门 task switch

9.6.1 Interrupt Procedures

流程

1. EFLAGS register & address & (error code) -> stack
2. 返回时 IRET 会将增加 EIP 4 个字节，并改回保护的标志寄存器位。

9.7 Error Code

9.8 Exception Conditions

一些中断发生的条件。

Exercise 4

Task: 对于每个异常或中断

1. 在 trapentry.S 中编写 handler，借用其中的宏 TRAPHANDLER 和 TRAPHANDLER_NOEC 。
2. 在函数 trap_init() 初始化 IDT 设置对应处理程序地址，

```
trapentry.S

从 9.10 Error Code Summary 了解到哪些异常需要error code
在 trap.h 查找对应的异常。

TRAPHANDLER_NOEC(Handler_DIVIDE, T_DIVIDE)
TRAPHANDLER_NOEC(Handler_DEBUG, T_DEBUG)
TRAPHANDLER_NOEC(Handler_NMI, T_NMI)
TRAPHANDLER_NOEC(Handler_BRKPT, T_BRKPT)
TRAPHANDLER_NOEC(Handler_OFLOW, T_OFLOW)
TRAPHANDLER_NOEC(Handler_BOUND, T_BOUND)
TRAPHANDLER_NOEC(Handler_ILLOP, T_ILLOP)
TRAPHANDLER_NOEC(Handler_DEVICE, T_DEVICE)
TRAPHANDLER(Handler_DBLFLT, T_DBLFLT)
TRAPHANDLER(Handler_TSS, T_TSS)
TRAPHANDLER(Handler_SEGNP, T_SEGNP)
TRAPHANDLER(Handler_STACK, T_STACK)
TRAPHANDLER(Handler_GPFLT, T_GPFLT)
TRAPHANDLER(Handler_PGFLT, T_PGFLT)
TRAPHANDLER_NOEC(Handler_FPERR, T_FPERR)
TRAPHANDLER(Handler_ALIGN, T_ALIGN)
TRAPHANDLER_NOEC(Handler_MCHK, T_MCHK)
TRAPHANDLER_NOEC(Handler_SIMDERR, T_SIMDERR)
TRAPHANDLER_NOEC(Handler_SYSCALL, T_SYSCALL)

补充完整 TRAPHANDLER 和 TRAPHANDLER_NOEC
```

已经知道 TRAPHANDLER 和 TRAPHANDLER_NOEC 已经将 异常号推入栈中，因此只剩下ds，es和一些基础寄存器。
使用 pushw 和 pushl将其压入栈中（） pushw 是 padding
再将 GD_KD 移到 ds 和 es 上，注意到移动会报错，所以通过 %eax 寄存器过渡一下。
最后把栈指针push到栈中，并调用 trap

```
_alltraps:
  pushw $0x0
  pushw %ds
  pushw $0x0
  pushw %es
  pushal
  movl $GD_KD, %eax
  movw %ax, %ds
  movw %ax, %es
  push %esp
  call trap
```

```
    void Handler_*();
    SETGATE(idt[T_DIVIDE],0,GD_KT,Handler_DIVIDE,0);
    SETGATE(idt[T_DEBUG],0,GD_KT,Handler_DEBUG,3);
    SETGATE(idt[T_NMI],0,GD_KT,Handler_NMI,0);
    SETGATE(idt[T_BRKPT],1,GD_KT,Handler_BRKPT,3);
    SETGATE(idt[T_OFLOW],1,GD_KT,Handler_OFLOW,0);
    SETGATE(idt[T_BOUND],0,GD_KT,Handler_BOUND,0);
    SETGATE(idt[T_ILLOP],0,GD_KT,Handler_ILLOP,0);
    SETGATE(idt[T_DEVICE],0,GD_KT,Handler_DEVICE,0);
    SETGATE(idt[T_DBLFLT],0,GD_KT,Handler_DBLFLT,0);
    SETGATE(idt[T_TSS],0,GD_KT,Handler_TSS,0);
    SETGATE(idt[T_SEGNP],0,GD_KT,Handler_SEGNP,0);
    SETGATE(idt[T_STACK],0,GD_KT,Handler_STACK,0);
    SETGATE(idt[T_GPFLT],0,GD_KT,Handler_GPFLT,0);
    SETGATE(idt[T_PGFLT],0,GD_KT,Handler_PGFLT,0);
    SETGATE(idt[T_FPERR],0,GD_KT,Handler_FPERR,0);
    SETGATE(idt[T_ALIGN],0,GD_KT,Handler_ALIGN,0);
    SETGATE(idt[T_MCHK],0,GD_KT,Handler_MCHK,0);
    SETGATE(idt[T_SIMDERR],0,GD_KT,Handler_SIMDERR,0);
    SETGATE(idt[T_SYSCALL],0,GD_KT,Handler_SYSCALL,3);
```

先声明函数，再修改 idt 对应表项，

```
#define SETGATE(gate, istrap, sel, off, dpl)      \
```

```
{               \
  (gate).gd_off_15_0 = (uint32_t) (off) & 0xffff;   \
  (gate).gd_sel = (sel);           \
  (gate).gd_args = 0;          \
  (gate).gd_rsv1 = 0;          \
  (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32;  \
  (gate).gd_s = 0;           \
  (gate).gd_dpl = (dpl);           \
  (gate).gd_p = 1;           \
  (gate).gd_off_31_16 = (uint32_t) (off) >> 16;   \
}

istrap 表示是否为 trap, 查 9.9 Exception Summary 表知
Breakpoint,Overflow,Two-byte SW Interrupt 是 trap。
dpl: Descriptor Privilege Level。
```

Question：

1. 为什么对于每个 异常/中断 都要单独设计 handler

   答：1.不同异常 输入的参数不同，有的需要error code，而有的并不需要。2.不同异常所需要的权限不同。3.不同异常处理过程中执行的操作流程也不同。

2. 我们把 idt 表项中第 14 个标箱 dpl 设置为 0。 在 user/softint 中，代码中产生一个 中断 14，但是期望产生 13 为什么，需要怎么做。如果能随意产生 中断 14 会发生什么。

   答：首先系统正在运行在用户模式下，特权级为3，但是特权级为0。特权级在调用 int 14 过程 所需要特权级为 3 此时，如果处理器在当前特权级大于0的情况下遇到这样的指令，将产生一个通用保护异常。（见 6.3），如果可以随意产生 Page Fault， 可能会导致内存泄漏。

PART B：Page Faults, Breakpoints Exceptions, and System Calls

Exercise 5

```
完善 trap_dispatch 处理 Page_fault
static void
trap_dispatch(struct Trapframe *tf)
{
  if (tf->tf_trapno == T_PGFLT)
    page_fault_handler(tf);
  print_trapframe(tf);
  if (tf->tf_cs == GD_KT)
    panic("unhandled trap in kernel");
  else {
    env_destroy(curenv);
    return;
  }
}
```

Exercise 6

```
完善 trap_dispatch 处理 breakpoint
static void
trap_dispatch(struct Trapframe *tf)
{
  switch (tf->tf_trapno){
    case T_PGFLT: page_fault_handler(tf);break;
    case T_DEBUG: monitor(tf):break;
  }
  print_trapframe(tf);
  if (tf->tf_cs == GD_KT)
    panic("unhandled trap in kernel");
  else {
    env_destroy(curenv);
    return;
  }
}
```

Question：

3. break point test case 可能会 断点异常 或 一般保护故障，什么时候会产生断点异常，什么时候会导致一般保护故障，为什么。

答：在初始化 idt 的时候对于 breakpoint 即 第 3 个条目，如果初始化特权级为 3 时正确产生断点一场，如果初始化为 0 则会产生一般保护错误，这还是因为在用户权限下调用权限 0 会导致一般保护错误。

4. 机制的意义是什么。

   答：通过权限分级管理，对于用户而言在一定特定条件下可以执行内核代码，这些代码属于比较常用，可以共享在内核处理特定问题。而对于高权限的限制，可以有效的保护系统，使得系统更安全。

Exercise 7

```
完善 syscall
增加 T_SYSCALL 的情况，系统调用号放在 eax 中，并将参数依次存在 edx...
调用 syscall，返回值为负数，说明执行过程中发生错误，对应为负的error。
static void
trap_dispatch(struct Trapframe *tf)
{
  switch (tf->tf_trapno){
    case T_PGFLT: page_fault_handler(tf);break;
    case T_BRKPT: monitor(tf);break;
    case T_SYSCALL: {
      int32_t ret = syscall(tf->tf_regs.reg_eax,
                    tf->tf_regs.reg_edx,
                  tf->tf_regs.reg_ecx,
                  tf->tf_regs.reg_ebx,
                  tf->tf_regs.reg_edi,
                  tf->tf_regs.reg_esi);
      if (ret < 0 )
        panic("trap_dispatch: system call %d\n",ret);
      tf->tf_regs.reg_eax = ret;
      return;
    }
  }
  print_trapframe(tf);
  if (tf->tf_cs == GD_KT)
    panic("unhandled trap in kernel");
  else {
    env_destroy(curenv);
    return;
  }
```

```
}

以此调用已完成的函数。
注意到 sys_cputs 是没有返回值的，因此要将函数返回值初始为0，最后如果调用值未完善则返回 -E_INVAL 表示该调用不合法。
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
{
  int32_t res = 0;
  switch (syscallno){
    case SYS_cputs:sys_cputs((const char *)a1,a2);break;
    case SYS_cgetc:res = sys_cgetc();break;
    case SYS_getenvid:res = sys_getenvid();break;
    case SYS_env_destroy:res = sys_env_destroy(a1);break;
    default:res = -E_INVAL;
  }
  return res;
  panic("syscall not implemented");

  switch (syscallno) {
  default:
    return -E_INVAL;
  }
}
```

Exercise 8

Task：设置 thisenv

```
首先 lib/entry.S 进行基本设置，然后进入 lib/libmain.c 在这里初始化 thisenv。

sys_getenvid() 是个系统调用获取当且环境 id。找到对应的环境修改即可。
void
libmain(int argc, char **argv)
{
  thisenv = 0;
  thisenv = envs + ENVX(sys_getenvid());
  if (argc > 0)
    binaryname = argv[0];
```

```
11
12    umain(argc, argv);
13
14    exit();
15  }
16
```

Exercise 9

Task：内核中的 Page Fault 相对用户 Page Fault 会严重，内核无法处理。因此要区分 用户产生的 Page fault 还是 内核产生的。

用户程序可能传递指针窃取非法内存，因此内核需要检查所有从用户空间传递到内核的指针。

1. Kern/trap.c：内核页错误->panic(tf_cs)
2. kern/pmap.c: 实现user_mem_check
3. kern/syscall.c: 检测参数
4. kern/kdebug.c:debuginfo_eip

```
1  kern/trap.c
2  通过最低位判断是否处于用户态，若处于内核态遭遇 page_fault 则产生panic。
3
4  void
5  page_fault_handler(struct Trapframe *tf)
6  {
7    uint32_t fault_va;
8    fault_va = rcr2();
9    cprintf("tf_cs:  %x\n",tf->tf_cs);
10   if ( (tf->tf_cs&1)!=1 )
11     panic("page_fault_handler: kernel page fault!\n");
12   cprintf("[%08x] user fault va %08x ip %08x\n",
13     curenv->env_id, fault_va, tf->tf_eip);
14   print_trapframe(tf);
15   env_destroy(curenv);
16 }
```

```
1  kern/pmap.c
2  检查访问的地址权限，首先地址可能不对齐先对齐。
```

对于每一页，首先检查地址是否超出 ULIM ，超出则返回 -E_FAULT，并设置 user_mem_check_addr
再检测权限位，若不符合处理同上。

特别注意：由于对齐，因此在首页时要求 给出的是第一个不满足地址，因此给出的是首地址而不是对齐之后的地址，否则会出错。

```
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
  // LAB 3: Your code here.
  uintptr_t start = (uintptr_t)ROUNDDOWN(va,PGSIZE);
  uintptr_t end = (uintptr_t)ROUNDUP(va+len,PGSIZE);
  perm |= PTE_P;

  for (uintptr_t address = start; address < end; address+= PGSIZE){
    if (address >= ULIM){
      user_mem_check_addr = (address>(uintptr_t)va)?address:(uintptr_t)va;
      return -E_FAULT;
    }
    pte_t *pte = pgdir_walk(env->env_pgdir, (void *)address, 0);
    if ((*pte & perm) != perm){
      user_mem_check_addr = (address>(uintptr_t)va)?address:(uintptr_t)va;
      return -E_FAULT;
    }
  }
  return 0;
}
```

kern/syscall.c: 在 sys_cputs 函数处检测输出的区间地址是否是用户可访问的。

```
static void
sys_cputs(const char *s, size_t len)
{
  user_mem_assert(curenv,(const void *)s,len,PTE_U);
  cprintf("%.*s", len, s);
}
```

kern/kdebug.c
对于申请的空间检测是否是用户可访问的。

```
3
4   int
5   debuginfo_eip(uintptr_t addr, struct Eipdebuginfo *info)
6   {
7     ...
8     if (addr >= ULIM) {
9       stabs = __STAB_BEGIN__;
10      stab_end = __STAB_END__;
11      stabstr = __STABSTR_BEGIN__;
12      stabstr_end = __STABSTR_END__;
13    } else {
14      const struct UserStabData *usd = (const struct UserStabData *)
    USTABDATA;
15
16      if (user_mem_check(curenv, usd,sizeof(struct UserStabData),PTE_U)
    <0)return -1;
17      stabs = usd->stabs;
18      stab_end = usd->stab_end;
19      stabstr = usd->stabstr;
20      stabstr_end = usd->stabstr_end;
21
22      if (user_mem_check(curenv,stabs,stab_end-stabs,PTE_U)<0)return -1;
23      if (user_mem_check(curenv,stabstr,stabstr_end-stabstr,PTE_U)
    <0)return -1;
24    }
25    ...
26  }
```

```
Incoming TRAP frame at 0xeffffe5c
tf_cs:  8
kernel panic at kern/trap.c:264: page_fault_handler: kernel page fault!
```

Exercise 10

Task:测试 user/evilhello

```
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xeffffbc
Incoming TRAP frame at 0xeffffbc
[00001000] user_mem_check assertion failure for va f010000c
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

Challenge

Task: Challenge 2

```
Challenge! Modify the JOS kernel monitor so that you can 'continue'
execution from the current location (e.g., after the int3, if the kernel
monitor was invoked via the breakpoint exception), and so that you can
single-step one instruction at a time. You will need to understand
certain bits of the EFLAGS register in order to implement single-
stepping.

Optional: If you're feeling really adventurous, find some x86
disassembler source code - e.g., by ripping it out of QEMU, or out of
GNU binutils, or just write it yourself - and extend the JOS kernel
monitor to be able to disassemble and display instructions as you are
stepping through them. Combined with the symbol table loading from lab
1, this is the stuff of which real kernel debuggers are made.
```

```
继续执行，将 FL_TF 位清 0，返回一个负数以退出 monitor 。
(FL_TF 为 1 时每执行一条语句就会产生一个异常)

int
mon_continue(int argc, char **argv, struct Trapframe *tf){
    if (!tf){cprintf("mon_continue: No Trapframe!\n");return 0;}
  switch (tf->tf_trapno){
      case T_BRKPT:
            tf->tf_eflags &= ~FL_TF;return -1;
      case T_DEBUG:
```

```
            tf->tf_eflags &= ~FL_TF;return -1;
      default:
          return 0;
    }
}
```

```
单步执行，如果是 breakpoint ，则设置 FL_TF 位。

int
mon_stepi(int argc, char **argv, struct Trapframe *tf){
    if (!tf){cprintf("mon_stepi: No Trapframe!\n");return 0;}
    switch (tf->tf_trapno){
      case T_BRKPT:tf->tf_eflags|=FL_TF;return -1;
      case T_DEBUG:
          if (tf->tf_eflags&FL_TF)return -1;
      default:
          return 0;
    }
}
```

```
反汇编，这里调用了 "disassembler.h" 中函数 disassemble，代码源于项目 [github](https://github.com/btbd/disassembler)，在其基础上进行适当修改（以保证通过编译并使其符合本任务要求）。

mon_disassembler 主要实现了对当前地址语句的反汇编。支持0/1个参数，当参数为 1 时，可以输入参数 n ，表示反汇编语句的句数。否则默认为 1。

1. 由 tf 获取地址。
2. 对于每条指令
  count = disassemble(address, 0x10, 0x0, disassembled);
  表示将 1. address 地址
        2. 至多 10 byte(最大语句长度，只要充分答即可) 的
        3. 偏移量为 0x0 (这里是相对函数头的偏移量，由于在本任务中可能在函数中间开始反汇编，因此不是很容易获取偏移量，因此在该任务中不考虑偏移量(即默认为 0)，因此可能在反汇编 jump 之类的语句时产生的结果还是相对地址)。
        4. 翻译到 disassembled 地址。
3. 然后将结果输出。

int
mon_disassembler(int argc, char **argv, struct Trapframe *tf){
  if(argc>2){
    cprintf("mon_disassembler: The number of parameters is two.\n");
```

```
    return 0;
  }
  int InstructionNumber = 1;
  if (argc == 2){
    char *errChar;
    InstructionNumber = strtol(argv[1], &errChar, 0);
    if (*errChar){
      cprintf("mon_disassembler: The first argument is not a
number.\n");
      return 0;
    }
  }
  cprintf("%d %d\n",argc,InstructionNumber);
    if (!tf){cprintf("mon_disassembler: No Trapframe!\n");return 0;}
  unsigned char* address = (unsigned char*)tf->tf_eip;
  for (int i = 0;i<InstructionNumber;i++){
    char disassembled[0xFF];
    char instruction[0xFF];
    uint32_t count = disassemble(address, 0x10, 0x0, disassembled);
    cprintf("%08x: ", address);
    instruction[0] = 0;
    for (int e = 0; e < count; e++) {
      snprintf(instruction + strlen(instruction),0xf, "%02x ",
address[e]);
    }
    cprintf("%-20s %s\n", instruction, disassembled);
    address = (unsigned char*)((uint32_t)address + count);
  }
  return 0;
}
```

```
具体 disassemble 函数实现 见 kern/disassembler.h
```

```
测试

这里对 breakpoint 函数进行稍微修改,
#include <inc/lib.h>
void
umain(int argc, char **argv)
{
```

```
  int a = 1,b = 2,c=0, d=0, e=0,f=0;
  asm volatile("int $3");
  c = a + b;
  d = c - a;
  e = d / b;
  asm volatile("addl %0,%0":"=r"(c):"0" (a));
  cprintf("%d %d %d %d %d\n",a,b,c,d,e);
  asm volatile("int $3");
  if (c+a>d*e)f=0;else f =1;
  cprintf("%d\n",f);
  return;
}
测试结束后会恢复修改。以保证 make grade 的正确性。

```

```
K> mon_disassembler 20
2 20
0080003c: bb 01 00 00 00                mov ebx,0x1
00800041: 01 db                         add DWORD PTR ebx,ebx
00800043: c7 44 24 14 01 00 00 00       mov DWORD PTR [esp+0x14],0x1
0080004b: c7 44 24 10 02 00 00 00       mov DWORD PTR [esp+0x10],0x2
00800053: 89 5c 24 0c                   mov DWORD PTR [esp+0xc],ebx
00800057: c7 44 24 08 02 00 00 00       mov DWORD PTR [esp+0x8],0x2
0080005f: c7 44 24 04 01 00 00 00       mov DWORD PTR [esp+0x4],0x1
00800067: c7 04 24 c8 0d 80 00          mov DWORD PTR [esp],0x800dc8
0080006e: e8 2d 01 00 00                call 0x132
00800073: cc                            int3
00800074: 43                            inc ebx
00800075: 83 fb 02                      adc DWORD PTR ebx,0x2
00800078: 0f 9e c0                      setle BYTE PTR al
0080007b: 0f b6 c0                      movzx al,BYTE PTR al
0080007e: 89 44 24 04                   mov DWORD PTR [esp+0x4],eax
00800082: c7 04 24 d4 0d 80 00          mov DWORD PTR [esp],0x800dd4
00800089: e8 12 01 00 00                call 0x117
0080008e: 83 c4 24                      adc DWORD PTR esp,0x24
00800091: 5b                            pop ebx
00800092: 5d                            pop ebp
```

首先展示 mon_disassembler ， 可以看到反汇编的结果是正确的。(这里应该在编译的时候进行了优化，导致中间的运算都没有体现在汇编代码中而是在编译过程中就计算了结果)，不过中间的一句汇编代码还是可以在反汇编结果中找到即 语句 [01 db]， （这里顺序改变了），如果要进一步测试可以嵌入更多的汇编代码。

```
K> mon_stepi
Incoming TRAP frame at 0xeffffbc
trap_dispatch 1
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf024e000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xeffffdc
  ebx  0x00000002
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x00000001 Debug
  err  0x00000000
  eip  0x00800043
  cs   0x----001b
  flag 0x00000102
  esp  0xeebfdfa8
  ss   0x----0023
```

```
K> mon_stepi
Incoming TRAP frame at 0xeffffbc
trap_dispatch 1
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf024e000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xeffffdc
  ebx  0x00000001
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x00000001 Debug
  err  0x00000000
  eip  0x00800041
  cs   0x----001b
  flag 0x00000182
  esp  0xeebfdfa8
  ss   0x----0023
```

这里执行两次 mon_stepi 的 结果，主要观察 eip 和 返汇编对应，可以进一步证明反汇编结果正确且每执行一步就陷入内核一次。

```
K> mon_continue
Incoming TRAP frame at 0xeffffbc
trap_dispatch 30
1 2 2 2 1
Incoming TRAP frame at 0xeffffbc
trap_dispatch 3
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf024e000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xeffffdc
  ebx  0x00000002
  edx  0xeebfde78
  ecx  0x0000000a
  eax  0x0000000a
  es   0x----0023
  ds   0x----0023
  trap 0x00000003 Breakpoint
  err  0x00000000
  eip  0x00800074
  cs   0x----001b
  flag 0x00000096
  esp  0xeebfdfa8
  ss   0x----0023
```

```
K> mon_continue
Incoming TRAP frame at 0xeffffbc
trap_dispatch 30
0
Incoming TRAP frame at 0xeffffbc
trap_dispatch 30
[00001000] exiting gracefully
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

这里展示 两次 mon_continue 的 结果，第一次 continue 遇到第二个端点，比较 eip 与 disassembler 一致，输出结果正确。

第二次 continue 直到程序运行结束。