Lab3 作业报告

介绍

这个lab实现了进程所需的基本内核环境。

1. 在kernel里面设置数据结构来跟踪用户环境，创建单个用户环境，加载程序并运行。
2. 处理进程的system call和exceptions.

准备工作

1. debug
   除了之前的make qemu-gdb, gdb
   lab3新增了make run-hello与make run-hello-nox来运行user/hello.c
2. 安装expect
   老师给的ubuntu里面没有安装，需要自己手动安装。
   sudo apt install expect 即可。

Part A

Part A主要实现了进程相关的数据结构以及处理exception。 我们先看看inc/env.h,这个文件里面实现了进程的数据结构struct Env，注意它也定义了一个常量NENV，这是Jos能同时支持的最大进程数。

我们再看看kern/env.c，这个文件的结构跟我们之前lab2的pmap.c很类似。env.c实现了进程的init，create，run等功能。

Exercise 1:
给envs分配空间并映射。 这个操作跟pages很类似，直接改一下就好。

```
envs = boot_alloc(NENV * sizeof(struct Env));
memset(envs, 0, NENV * sizeof(struct Env));
```

然后映射一下

```
boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR(envs), PTE_U | PTE_P);
```

接下来我们修改kern/env.c来运行进程。 Exercise 2:

env_init() 初始化envs队列里的每一个进程，把他们全加进到env_free_list里面。

```
void
env_init(void)
{
    // Set up envs array
    // LAB 3: Your code here.

        int i;
```

```
    for (i = NENV-1; i >=0; i--) {
            envs[i].env_status = ENV_FREE;
            envs[i].env_link = env_free_list;
            envs[i].env_id = 0;
            envs[i].env_break = 0;
            env_free_list = &envs[i];
    }

    // Per-CPU part of the initialization
    env_init_percpu();
}
```

env_setup_vm() 给一个进程设置虚拟地址空间。首先分配一个页作为pgdir,然后把kern_pgdir里面用户空间映射给env_pgdir，最后map UVPT的权限。

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

    // Allocate a page for the page directory
    if (!(p = page_alloc(ALLOC_ZERO)))
        return -E_NO_MEM;

    // Now, set e->env_pgdir and initialize the page directory.
    //
    // Hint:
    //    - The VA space of all envs is identical above UTOP
    //  (except at UVPT, which we've set below).
    //   See inc/memlayout.h for permissions and layout.
    //   Can you use kern_pgdir as a template?  Hint: Yes.
    //  (Make sure you got the permissions right in Lab 2.)
    //    - The initial VA below UTOP is empty.
    //    - You do not need to make any more calls to page_alloc.
    //    - Note: In general, pp_ref is not maintained for
    //   physical pages mapped only above UTOP, but env_pgdir
    //   is an exception -- you need to increment env_pgdir's
    //   pp_ref for env_free to work correctly.
    //    - The functions in kern/pmap.h are handy.

    // LAB 3: Your code here.
        e->env_pgdir = (pde_t *)page2kva(p);
        p->pp_ref++;
        //map (UTOP,UVPT)
        for(i=PDX(UTOP);i<NPDENTRIES;i++){
            e->env_pgdir[i] = kern_pgdir[i];
        }

    // UVPT maps the env's own page table read-only.
    // Permissions: kernel R, user R

    e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
```

```
    return 0;
}
```

region_alloc() 给进程分配物理地址空间。

```
//
// Allocate len bytes of physical memory for environment env,
// and map it at virtual address va in the environment's address space.
// Does not zero or otherwise initialize the mapped pages in any way.
// Pages should be writable by user and kernel.
// Panic if any allocation attempt fails.
//
static void
region_alloc(struct Env *e, void *va, size_t len)
{
    // LAB 3: Your code here.
    // (But only if you need it for load_icode.)
    //
    // Hint: It is easier to use region_alloc if the caller can pass
    //   'va' and 'len' values that are not page-aligned.
    //   You should round va down, and round (va + len) up.
    //   (Watch out for corner-cases!)
        uint32_t start=(uint32_t)ROUNDDOWN(va,PGSIZE);
    uint32_t end=(uint32_t)ROUNDUP(va+len,PGSIZE);
        struct Page* tmp = NULL;
        for(uint32_t i=start;i<end; i+=PGSIZE){
            tmp = page_alloc(ALLOC_ZERO);
            if(!tmp){
               panic("Allocation attempt fails!");
            }
            else{
               if(page_insert(e->env_pgdir,tmp,(void *)i,PTE_W|PTE_U)){
                  panic("page table couldn't be allocated, failed in region_alloc");
               }
            }
        }

}
```

load_icode() 读取elf文件来初始化进程的program binary, stack, and processor flags。

```
//
// Set up the initial program binary, stack, and processor flags
// for a user process.
// This function is ONLY called during kernel initialization,
// before running the first user-mode environment.
//
// This function loads all loadable segments from the ELF binary image
// into the environment's user memory, starting at the appropriate

// virtual addresses indicated in the ELF program header.
```

```
// At the same time it clears to zero any portions of these segments
// that are marked in the program header as being mapped
// but not actually present in the ELF file - i.e., the program's bss section.
//
// All this is very similar to what our boot loader does, except the boot
// loader also needs to read the code from disk.  Take a look at
// boot/main.c to get ideas.
//
// Finally, this function maps one page for the program's initial stack.
//
// load_icode panics if it encounters problems.
//  - How might load_icode fail?  What might be wrong with the given input?
//
static void
load_icode(struct Env *e, uint8_t *binary, size_t size)
{
    // Hints:
    //  Load each program segment into virtual memory
    //  at the address specified in the ELF section header.
    //  You should only load segments with ph->p_type == ELF_PROG_LOAD.
    //  Each segment's virtual address can be found in ph->p_va
    //  and its size in memory can be found in ph->p_memsz.
    //  The ph->p_filesz bytes from the ELF binary, starting at
    //  'binary + ph->p_offset', should be copied to virtual address
    //  ph->p_va.  Any remaining memory bytes should be cleared to zero.
    //  (The ELF header should have ph->p_filesz <= ph->p_memsz.)
    //  Use functions from the previous lab to allocate and map pages.
    //
    //  All page protection bits should be user read/write for now.
    //  ELF segments are not necessarily page-aligned, but you can
    //  assume for this function that no two segments will touch
    //  the same virtual page.
    //
    //  You may find a function like region_alloc useful.
    //
    //  Loading the segments is much simpler if you can move data
    //  directly into the virtual addresses stored in the ELF binary.
    //  So which page directory should be in force during
    //  this function?
    //
    //  You must also do something with the program's entry point,
    //  to make sure that the environment starts executing there.
    //  What?  (See env_run() and env_pop_tf() below.)

    // LAB 3: Your code here.
        struct Proghdr *ph, *eph;
        struct Elf *elf;
    // cast type
    elf = (struct Elf *)binary;

    // is this a valid ELF?
    if (elf->e_magic != ELF_MAGIC)

        panic("Not ELF_NAGIC in load_icode()");
```

```
    // load each program segment (ignores ph flags)
    ph = (struct Proghdr *) ((uint8_t *) elf + elf->e_phoff);
    eph = ph + elf->e_phnum;

        lcr3(PADDR(e->env_pgdir));
    for (; ph < eph; ph++){
             if(ph->p_type == ELF_PROG_LOAD){
                region_alloc(e, (void *)ph->p_va, ph->p_memsz);
                memset((void *)ph->p_va,0,ph->p_memsz);
                if(ph->p_va + ph->p_memsz > e->env_break)
                      e->env_break = ph->p_va + ph->p_memsz;
                memmove((void *)ph->p_va,binary + ph->p_offset,ph->p_filesz);
             }
        }
        e->env_tf.tf_eip = elf->e_entry;
        lcr3(PADDR(kern_pgdir));

    // Now map one page for the program's initial stack
    // at virtual address USTACKTOP - PGSIZE.
        region_alloc(e, (void *)(USTACKTOP - PGSIZE), PGSIZE);
    // LAB 3: Your code here.

}
```

env_create() 调用env_alloc来建立一个新进程，用load_icode来初始化。

```
//
// Allocates a new env with env_alloc, loads the named elf
// binary into it with load_icode, and sets its env_type.
// This function is ONLY called during kernel initialization,
// before running the first user-mode environment.
// The new env's parent ID is set to 0.
//
void
env_create(uint8_t *binary, size_t size, enum EnvType type)
{
    // LAB 3: Your code here.
    struct Env *newenv_store ;
    int value = env_alloc(&newenv_store, 0);
    if(value==-E_NO_FREE_ENV){
      panic("NO FREE ENV in env_create()");
      return;
    }
    if(value==-E_NO_MEM){
      panic("NO FREE ENV in env_create()");
      return;
    }
    newenv_store->env_type = type;
    load_icode(newenv_store, binary, size);
```

```
}
```

env_run() 运行一个进程，修改进程的状态。

```
//
// Context switch from curenv to env e.
// Note: if this is the first call to env_run, curenv is NULL.
//
// This function does not return.
//
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!=e){
           if(curenv != NULL){
               if(curenv->env_status == ENV_RUNNING)
                  curenv->env_status = ENV_RUNNABLE;
           }
           curenv = e;
           curenv->env_status == ENV_RUNNING;
           curenv->env_runs++;
           lcr3(PADDR(curenv->env_pgdir));
        }
        env_pop_tf(&curenv->env_tf);

}
```

Exercise 2到此结束，此时qemu可以显示 `[00000000] new env 00002000`

我们运行make qemu-gdb, gdb调试一下，如下

```
(gdb) b env_pop_tf
Breakpoint 1 at 0xf0103836: file kern/env.c, line 490.
(gdb) b *0xf0104119
```

```
Breakpoint 2 at 0xf0104119: file kern/syscall.c, line 97.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0xf0103836 <env_pop_tf>: push   %ebp

Breakpoint 1, env_pop_tf (tf=0xf01b6000) at kern/env.c:490
490 {
(gdb) si
=> 0xf0103837 <env_pop_tf+1>:   mov    %esp,%ebp
0xf0103837  490 {
(gdb)
=> 0xf0103839 <env_pop_tf+3>:   sub    $0xc,%esp
0xf0103839  490 {
(gdb)
=> 0xf010383c <env_pop_tf+6>:   mov    0x8(%ebp),%esp
491     __asm __volatile("movl %0,%%esp\n"
(gdb)
=> 0xf010383f <env_pop_tf+9>:   popa
0xf010383f  491     __asm __volatile("movl %0,%%esp\n"
(gdb)
=> 0xf0103840 <env_pop_tf+10>:  pop    %es
0xf0103840 in env_pop_tf (
    tf=<error reading variable: Unknown argument list address for `tf'.>)
    at kern/env.c:491
491     __asm __volatile("movl %0,%%esp\n"
(gdb)
=> 0xf0103841 <env_pop_tf+11>:  pop    %ds
0xf0103841  491     __asm __volatile("movl %0,%%esp\n"
(gdb)
=> 0xf0103842 <env_pop_tf+12>:  add    $0x8,%esp
0xf0103842  491     __asm __volatile("movl %0,%%esp\n"
(gdb)
=> 0xf0103845 <env_pop_tf+15>:  iret
0xf0103845  491     __asm __volatile("movl %0,%%esp\n"
(gdb)
=> 0x800020:    cmp    $0xeebfe000,%esp
0x00800020 in ?? ()
(gdb) c
Continuing.
=> 0xf0104119 <syscall>:    push   %ebp

Breakpoint 2, syscall (syscallno=2, a1=0, a2=0, a3=0, a4=0, a5=0)
    at kern/syscall.c:97
97  {
(gdb)
```

可见程序能够正常syscall。

接下来我们来实现对中断跟异常的处理。 异常跟中断都是protected control transfers,都是让处理器从 用户态(CPL=3) 转为 内核态(CPL=0)同时也不会给用户态代码任何干扰内核运行的机会,在intel的术语中interrupt通常为处理器外部异步事件引起的保护控制传输,比如外部I/O活动,作为对比exception为同步事件引起的保护控制传输,例如除0,访问无效内存。

为了保证中断跟异常到内核态的传输是受保护的，x86提供了两种机制：IDT跟TSS。

IDT： kernel对特定的中断，有特定的入口点，而不会继续执行错误的代码。,x86允许256个不同的interrupt/exception 入口点，也就是interrupt vector(也就是0~255的整数),数值由中断类型决定,CPU用interrupt vector的值作为index在IDT中找值放入eip,也就是指向内核处理该错误的到函数入口,加载到代码段（CS）寄存器中的值,其中第0-1位包括要运行异常处理程序的权限级别。 （在JOS中,所有异常都在内核模式下处理,权限级别为0）

简单的说就是，不同的错误(interrupt/exception)会发出不同的值(0~255)然后cpu再根据该值 在IDT中找处理函数入口,所以我们的任务要去配置IDT表 以及实现对应的处理函数。

TSS： 在中断前 需要保存当前程序的寄存器等 在处理完后回重新赋值这些寄存器 所以保存的位置需要不被用户修改 否则在重载时可能造成危害

因此x86在 处理interrupt/trap时 模式从用户转换到内核时,它还会转换到一个内核内存里的栈(一个叫做TSS(task state segment)的结构体),处理器把SS, ESP, EFLAGS, CS, EIP, and an optional error code push到这个栈上,然后它再从IDT的配置 设置CS和EIP的值,再根据新的栈设置esp和ss

虽然 TSS很大并有很多用途,但对于lab对于jos我们只用它来定义处理器在从用户模式 转换到内核模式时,应切换的堆栈,因为x86上JOS在kernel态的权限级别为0,在进入内核模式时,处理器用TSS的ESP0 和SS0两个字段来定义内核栈 ,JOS不使用其它的TSS字段

我们接下来要处理IDT里面0-31的异常，IDT里面大于31的都是软件中断，我们将在Part B里面处理IDT里面编号48的syscall。

设置IDT

```
                  IDT                   trapentry.S         trap.c

       +----------------+
       |   &handler1    |---------> handler1:          trap (struct Trapframe *tf)
       |                |             // do stuff      {
       |                |             call trap          // handle the exception/interrupt
       |                |             // ...           }
       +----------------+
       |   &handler2    |--------> handler2:
       |                |            // do stuff
       |                |            call trap
       |                |            // ...
       +----------------+
                  .
                  .
                  .
       +----------------+
       |   &handlerX    |--------> handlerX:
       |                |             // do stuff
       |                |             call trap
       |                |             // ...
       +----------------+
```

最终实现的结构如上。

Exercise 4: 修改trapentry.s 跟 trap.c 来实现上述操作。
trapentry.s里面定义了两个宏：
TRAPHANDLER(name, num)： 给trap号为num的trap定义一个trap_handle函数，会push error code
TRAPHANDLER_NOEC(name, num)：
不push error code 至于IDT里面哪个会push error code，哪个不会，就需要自己google了，或者参考这里。

trapentry.s

```
/*
 * Lab 3: Your code here for generating entry points for the different traps.
 */
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_NOEC(handler_align, T_ALIGN)
TRAPHANDLER_NOEC(handler_mchk, T_MCHK)
TRAPHANDLER_NOEC(handler_simderr, T_SIMDERR)
```

这里的函数名与挂钩IDT的操作实在trap.c中的trap_init来实现的。

下面我们实现_alltraps

```
_alltraps:
    pushl %ds
    pushl %es
    pushal
    movl $GD_KD, %eax
    movw %ax,%ds
    movw %ax,%es
    pushl %esp
    call trap
```

我们看Trapframe的结构：

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
```

```
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

我们之前压入了tf_trapno,还剩ds与es两个要压栈，其余的可以用pushall来操作。然后load GD_KD into %ds and %es，push %esp， call trap即可。

trap.c

```
void
trap_init(void)
{
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.

        extern void handler_divide();
        extern void handler_debug();
        extern void handler_nmi();
        extern void handler_brkpt();
        extern void handler_oflow();
        extern void handler_bound();
        extern void handler_illop();
        extern void handler_device();
        extern void handler_dblflt();
        extern void handler_tss();
        extern void handler_segnp();
        extern void handler_stack();
        extern void handler_gpflt();
        extern void handler_pgflt();
        extern void handler_fperr();
        extern void handler_align();
        extern void handler_mchk();
        extern void handler_simderr();


        SETGATE(idt[T_DIVIDE],0,GD_KT,handler_divide, 0);
        SETGATE(idt[T_DEBUG],0,GD_KT,handler_debug, 0);
        SETGATE(idt[T_NMI],0,GD_KT,handler_nmi, 0);
        SETGATE(idt[T_BRKPT],0,GD_KT,handler_brkpt, 3);
        SETGATE(idt[T_OFLOW],0,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);

    // Per-CPU setup
    trap_init_percpu();
}
```

现在，Part A已经全部完成，总结一下，Part A初始化了进程，初始化了0-31的IDT。

Part B

在Part B，我们解决了Page Fault，BreakPoint和system call。
Exercise 5：
修改trap_dispatch函数来实现page_fault_handler()。

```
static void
trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.
        if(tf->tf_trapno == T_PGFLT)
            page_fault_handler(tf);
    // Unexpected trap: The user process or the kernel has a bug.
        if(tf->tf_trapno == T_BRKPT || tf->tf_trapno == T_DEBUG)
            monitor(tf);
        /*if(tf->tf_trapno == T_DEBUG){
            tf->tf_eflags = tf->tf_eflags & (~0x100);
            monitor(tf);
        }*/
    print_trapframe(tf);
    if (tf->tf_cs == GD_KT)
        panic("unhandled trap in kernel");
    else {
        env_destroy(curenv);
        return;
    }
}
```

系统调用

用户程序通过使用系统调用来 让内核帮它们完成它们自己权限所不能完成的事情，当用户程序调用系统调用时 处理器进入内核态，处理器+内核合作一起保存用户态的状态，内核再执行对应的系统调用的代码，完成后再返回用户态。但用户如何调用系统调用的内容和过程因系统而异。

在JOS里我们使用sysenter指令，你需要在kern/init.c中配置MSRs，来允许用户调用系统调用 程序会用寄存器传递系统调用号和系统调用参数，系统调用号放在%eax中，参数依次放在%edx，%ecx，%ebx， %edi中，内核执行完后返回值放在%eax中，在lib/syscall.c的syscall()函数中已经写好了汇编的系统调用函数的一部分，你也许需要修改这个函数，如处理返回值或消除冗余的寄存器保存但不要修改sysenter指令。

Exercise 6： 用sysenter和sysexit指令来实现系统调用。 我们调用syscall的流程是：

1. 调用lib/syscall.c 里面的syscall函数，压入返回地址到%esi，把%esp压到%ebp，然后sysenter。

```
//Lab 3: Your code here
                 "leal after_sysenter_label%=, %%esi\n\t"
                 "movl %%esp,%%ebp\n\t"
                 "sysenter\n\t"
                 "after_sysenter_label%=: \n\t"
```

2. 执行kern/trapentry.s里面的sysenter_handler函数，具体就是压入syscall所需的参数，然后调用syscall，最后将返回地址和esp压到%edx和%ecx（为什么是这两个寄存器，这是sysexit规定的）里面，然后sysexit。

```
sysenter_handler:
/*
 * Lab 3: Your code here for system call handling
 */
  pushl %edi
  pushl %ebx
  pushl %ecx
  pushl %edx
  pushl %eax
  call syscall
  movl %ebp, %ecx
  movl %esi, %edx
  sysexit
```

3. 然后进入kern/syscall.c里面补全syscall函数。

```
// Dispatches to the correct kernel function, passing the arguments.
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
{
    // Call the function corresponding to the 'syscallno' parameter.
    // Return any appropriate return value.
    // LAB 3: Your code here.

        int32_t ret = 0;
        switch(syscallno){
        case SYS_cputs:
          sys_cputs((const char*)a1,(size_t)a2);
          break;
```

```
        case SYS_cgetc:
          ret = sys_cgetc();
          break;
        case SYS_getenvid:
          ret = sys_getenvid();
          break;
        case SYS_env_destroy:
          ret = sys_env_destroy((envid_t)a1);
          break;
        case SYS_map_kernel_page:
          ret = sys_map_kernel_page((void*)a1,(void*)a2);
          break;
        case SYS_sbrk:
          ret = sys_sbrk(a1);
          break;
        default:
          return -E_INVAL;
        }
        return ret;

    //panic("syscall not implemented");
}
```

除此之外，我们还要在kern/init.c里面设置MSRs来启用sysenter跟sysexit。 kern/init.c

```
extern void sysenter_handler();
wrmsr(0x174, GD_KT, 0);
wrmsr(0x175, KSTACKTOP, 0);
wrmsr(0x176, (uint32_t)&sysenter_handler, 0);
```

inc/x86.c

```
static inline void
wrmsr(unsigned msr, unsigned low, unsigned high)
{
        asm volatile("wrmsr" : : "c" (msr), "a"(low), "d" (high) : "memory");
}
```

用户模式启动

Exercise 7: 修改lib/libmain.c里面的libmain函数来输出“i am environment 00001000”

```
thisenv = envs + ENVX(sys_getenvid());
```

Exercise 8: 实现sys_sbrk()函数，即实现动态申请内存。 我们先在Struct Env结构中增加`uintptr_t env_break;`来记录heap的位置（注意heap是向下增长的）。 在kern/env.c中给env_break赋值 `region_alloc(e, (void *)(USTACKTOP - PGSIZE), PGSIZE); // LAB 3: Your code here. e->env_break = (uintptr_t)ROUNDDOWN(USTACKTOP - PGSIZE,PGSIZE);`
最后我们在kern/sycall.c里面实现sbrk。

```
static int
sys_sbrk(uint32_t inc)
{
    // LAB3: your code sbrk here...

        region_alloc(curenv, (void *) (curenv->env_break - inc), inc);
        return curenv->env_break = (uintptr_t)ROUNDDOWN(curenv->env_break - inc,PGSIZE);

}
```

断点异常

Exercise 9:
改变trap_dispatch()让断点异常能够调用kernel monitor。然后实现c,si,x三条指令。

```
static void
trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.
        if(tf->tf_trapno == T_PGFLT)
            page_fault_handler(tf);
    // Unexpected trap: The user process or the kernel has a bug.
        if(tf->tf_trapno == T_BRKPT || tf->tf_trapno == T_DEBUG)
            monitor(tf);
        /*if(tf->tf_trapno == T_DEBUG){
            tf->tf_eflags = tf->tf_eflags & (~0x100);
            monitor(tf);
        }*/
    print_trapframe(tf);
    if (tf->tf_cs == GD_KT)
        panic("unhandled trap in kernel");
    else {
        env_destroy(curenv);
        return;
    }
}
```

page fault 和 内存保护

Exercise 10: 修改kern/trap.c 使之若在kernel mode 发生页错误 则panic。 阅读kern/pmap.c中的user_mem_assert()函数并实现user_mem_check()函数. 修改kern/syscall.c以至能健全的检查系统调用的参数. 修改kern/kdebug.c中的debuginfo_eip 让它调用user_mem_check。

kern/trap.c

```
void
page_fault_handler(struct Trapframe *tf)

{
```

```
    uint32_t fault_va;

    // Read processor's CR2 register to find the faulting address
    fault_va = rcr2();

    // Handle kernel-mode page faults.

    // LAB 3: Your code here.
        if (tf->tf_cs == GD_KT){
            panic("page fault in kernel");
        }
    // We've already handled kernel-mode exceptions, so if we get here,
    // the page fault happened in user mode.

    // Destroy the environment that caused the fault.
    cprintf("[%08x] user fault va %08x ip %08x\n",
        curenv->env_id, fault_va, tf->tf_eip);
    print_trapframe(tf);
    env_destroy(curenv);
}
```

我们先看一下kern/pamp.c里面的user_mem_assert函数，发现它调用了user_mem_check函数。user_mem_check看注释就是检测相应虚拟地址的权限，就是检测相应page的权限。

```
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
    // LAB 3: Your code here.
        void* begin = ROUNDDOWN((void*)va,PGSIZE);
        void* end = ROUNDUP((void*)(va+len),PGSIZE);
        while(begin < end){
           pte_t* pte = pgdir_walk(env->env_pgdir,begin,0);
           if((uint32_t)begin >= ULIM){
                if(begin > va)
                  user_mem_check_addr = (uintptr_t)begin;
                else
                  user_mem_check_addr = (uintptr_t)va;
                return -E_FAULT;
           }
           if(!pte || !(*pte & PTE_P) || ((*pte & perm) != perm)){
                if(begin > va)
                  user_mem_check_addr = (uintptr_t)begin;
                else
                  user_mem_check_addr = (uintptr_t)va;
                return -E_FAULT;
           }
           begin += PGSIZE;
        }
    return 0;
}
```

在kern/syscall.c里面的sys_cputs函数里面加上`user_mem_assert(curenv,s,len,PTE_U);`让函数生效。 最后在kern/kdebug.c里面加上

```
if (user_mem_check(curenv, usd, sizeof(struct UserStabData), PTE_U) < 0) return -1;
```

```
if (user_mem_check(curenv, stabs  , stab_end   -stabs  , PTE_U) < 0) return -1;
                if (user_mem_check(curenv, stabstr, stabstr_end-stabstr, PTE_U) < 0) return -1;
```

Exercise 12: 这里，我需要在用户态，获得 ring0 权限。首先，通过 sgdt，获得 gdt 的 base 地址。 其次，通过系统调用中的 sys_map_kernel_page 函数，将 gdt 所在的物理页映射到我提 供的虚拟地址上。由于映射的是页，因此存在着一定的偏移。由于 va 不一定是 PGSIZE 对齐，因此我们需要将其 PGOFF 设为 0，并加上 base 地址的 PGOFF，这样获得的地址， 才是 gdt 的地址。这之后，需要 GDT 中的用户的一段（我选择了 GD_UT）修改为 CallGateDescriptor，并将原先的 SEG Descriptor 保存起来，用于恢复。因此，在找到 GD_UT 对应的 SEG 的地址后，调用 SETCALLGATE，在 GDT 中存入调用门描述符，并指向一个 wrapper 函数。之后，使用 lcall 指令，调用刚才修改的 CallGateDescriptor。在 wrapper 函数中，调用 evil，并恢复 gdt，pop ebp，再调用 lret，回到原先的函数。这样，在 wrapper 函数中，就以 RING0 的状态，调用了 evil 函数。

user/evilhello2.c

```
char va[PGSIZE];
struct Segdesc* gdt;
struct Segdesc savedGate;
struct Gatedesc* gate;
```

```
void wrapper(){
    evil();
    *((struct Segdesc*)gate) = savedGate;
    asm volatile("popl %ebp");
    asm volatile("lret\n\t");
}

// Invoke a given function pointer with ring0 privilege, then return to ring3
void ring0_call(void (*fun_ptr)(void)) {
    // Here's some hints on how to achieve this.
    // 1. Store the GDT descripter to memory (sgdt instruction)
    // 2. Map GDT in user space (sys_map_kernel_page)
    // 3. Setup a CALLGATE in GDT (SETCALLGATE macro)
    // 4. Enter ring0 (lcall instruction)
    // 5. Call the function pointer
    // 6. Recover GDT entry modified in step 3 (if any)
    // 7. Leave ring0 (lret instruction)

    // Hint : use a wrapper function to call fun_ptr. Feel free
    //        to add any functions or global variables in this
    //        file if necessary.

    // Lab3 : Your Code Here
    struct Pseudodesc gdtd;
    sgdt(&gdtd);
```

```
    sys_map_kernel_page((void*)gdtd.pd_base,(void*)va);
    gdt = (struct Segdesc*)(((PGNUM(va) << PTXSHIFT)) + (PGOFF(gdtd.pd_base)));
    savedGate = *(gdt + (GD_UT >> 3));
    gate = (struct Gatedesc*)(gdt + (GD_UT >> 3));
    SETCALLGATE(*gate,GD_KT,wrapper,3);
    asm volatile("lcall $0x18,$0\n\t");
}
```