OS Assignment 2: Documentation

November 30, 2013

Group No.12

Group members: Ma Xuan(1110580), Wang Shaoyao(1110539)
Wu Zhengwei(1110548), Yuan Zhaozheng(1110556), Zhang Chao(1110557)

Exercise 1

Description

在 kern/pmap.c 文件中找到i386_vm_init(), 为 envs 分配和映射内存。 这个数组保存着 NENV 个 Env 结构。 从内核的虚拟地址 UENVS 开始存放。当然, 我们的物理页就不再需要是连续的了。因为内核只是会使用虚拟地址来进行 envs 数组的访问。 你可以参考 pages[ ] 的方式来处理。

Creating and Running Environments

由于我们现在的操作系统没有实现文件系统, 所以只能先设置内核载入一个静态的二进制镜像。 JOS 将以真正的可执行的 ELF 格式将这些二进制文件嵌入到内核。Lab2 的 GNUmakefile 在目录 obj/user 产生了一系列的二进制文件。 如果你查看 kern/Makefrag, 你将会发现这些二进制文件是被如何直接链接到内核中去的。 链接选项中的-b binary 设置的目的是将这些文件按照原始的二进制文件而不是普通的由编译器生成的 .o 文件进行链接。在 i386_init() 和 kern/init.c 中, 你将会发现直接运行这些二进制文件镜像的代码, 然而这些关键的函数可能没有完全的实现,你需要完成他们的具体功能。

Solution

```
void
i386_vm_init(void)
{
    ...
    size_t spages = ROUNDUP(npage * sizeof(struct Page),
                            PGSIZE);
    pages = (struct Page*) boot_alloc(spages, PGSIZE);
    physaddr_t ppages = PADDR(pages);
    boot_map_segment(pgdir, UPAGES, spages, ppages, PTE_U);

    // Make 'envs' point to an array of size 'NENV' of
    // 'struct Env'.
    // Map this array read-only by the
    // user at linear address UENVS
    // (ie. perm = PTE_U | PTE_P).

    // LAB 3: Your code here.
    envs = boot_alloc(NENV * sizeof(struct Env), PGSIZE);
    boot_map_segment(pgdir,
                     UENVS,
                     ROUNDUP(NENV * sizeof(struct Env),
                             PGSIZE),
                     PADDR((uintptr_t) envs),
                     PTE_U);
    ...
}
```

Exercise 2

Description

在文件 env.c 中,你需要完成如下函数:

. env_init:初始化所有的 Env 数据结构,然后将它们加入到 env_free_list 中去。

. env_setup_vm: 为一个新的环境分配一个页目录, 并且初始化新环境地址空间中的内核部分。

. segment_alloc: 分配并且映射到相关的物理内存空间。

. load_icode: 你需要分析 ELF 文件格式, 就像 boot loader 中做的那样, 然后加载它的具体内容到新环境中的用户地址空间中。

. env_create: 调用 env_alloc 来创建一个新的环境, 调用 load_icode 载入 ELF 二进制文件镜像。

. env_run:在用户态运行这个指定的环境

Solution

env init() 可类比pages对应的 page init() 实现：

```
void
env_init(void)
{
    // LAB 3: Your code here.
    LIST_INIT(&env_free_list);
    int i = NENV;
    while(i--) {
        envs[i].env_id = 0;
        envs[i].env_status = ENV_FREE;
        LIST_INSERT_HEAD(&env_free_list,
                         &envs[i],
                         env_link);
    }
    return;
}
```

env-setup-vm.c：因为在UTOP之上的所有映射对于任何一个地址空间都是一样的(无论是对于内核地址空间还是对于任意一个用户地址空间而言),他们都和在lab2中对于内核地址空间设置的静态映射一样(静态映射就是没有实际分配物理页,即映射是通过boot map segment()而非 page_insert(), 所以这里我们能直接拷贝系统页目录 boot_pgdir中的内容。

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

    // Allocate a page for the page directory
    if ((r = page_alloc(&p)) < 0)
        return r;

```

```
    // Now, set e->env_pgdir and e->env_cr3,
    // and initialize the page directory.

    // LAB 3: Your code here.

    e->env_pgdir = page2kva(p);
    e->env_cr3 = page2pa(p);

    memmove(e->env_pgdir, boot_pgdir, PGSIZE);
    memset (e->env_pgdir, 0, PDX(UTOP) * sizeof(pde_t));
    p->pp_ref++;
    // set the va above UTOP to be the same

    // VPT and UVPT map the env's own page table,
    // with different permissions.
    e->env_pgdir[PDX(VPT)]  = e->env_cr3 | PTE_P | PTE_W;
    e->env_pgdir[PDX(UVPT)] = e->env_cr3 | PTE_P | PTE_U;
    return 0;
}
```

这个函数的作用是在e代表的用户虚拟地址空间中从va开始的地址分配出len长度的区域,准备写入数据。对实际的物理页面分配映射到当前用户的虚拟地址空间中。

```
static void
segment_alloc(struct Env *e, void *va, size_t len)
{
    // LAB 3: Your code here.
    // (But only if you need it for load_icode.)
    //
    // 'va' and 'len' values that are not page-aligned.
    // You should round va down, and round len up.
    void* end = ROUNDUP((char*)va + len, PGSIZE);
    va = ROUNDDOWN(va, PGSIZE);
    if (va == NULL)
        va += PGSIZE;
    len = (char*)end - (char*)va;
    struct Page *pp;
    for (; len > 0; len -= PGSIZE, va += PGSIZE) {
        int r;
        if ((r = page_alloc(&pp)) < 0)
            panic("segment_alloc: %e", r);
        if ((r = page_insert(e->env_pgdir,
                             pp, va,
                             PTE_W | PTE_U)) < 0)
            panic("segment_alloc: %e", r);
    }
}
```

这个函数的作用就是将嵌入在内核中的用户程序取出释放到相应链接器指定好的用户虚拟空间里。这里的 binary 指针,就是用户程序在内核中的开始位置的虚拟地址。

```
static void
load_icode(struct Env *e, uint8_t *binary, size_t size)
{
  // LAB 3: Your code here.
  struct Elf *elf = (struct Elf *) binary;
  uint8_t* va = NULL;
  uint8_t* origin_va = NULL;
  int page_num = 0;
  int byte_num = 0;
  int copyed_byte = 0;
  int filesz = 0;
  int memsz = 0;
  int offset = 0;

  struct Proghdr *ph, *eph;
  int i,j;
  struct Page *p;
  pte_t* pte;

  if (elf->e_magic != ELF_MAGIC)
    panic("elf->e_magic erro\n");
  // program header
  ph = (struct Proghdr *)(binary + elf->e_phoff);
  // one after last program header
  eph = ph + elf->e_phnum;
  // For each program header, load it into memory,
  // zeroing as necessary
  for (; ph < eph; ph++) {
    if (ph->p_type == ELF_PROG_LOAD) {
    // map segment
      segment_alloc(e,
                    (uintptr_t *)(ph->p_va),
                    ph->p_memsz);
      pte = (pte_t*)KADDR(
            PTE_ADDR(e->env_pgdir[PDX(ph->p_va)]));
      origin_va = (uint8_t*)ROUNDDOWN(
                  ph->p_va, PGSIZE);
      offset = ph->p_va - (int)origin_va;
      page_num = ROUNDUP(ph->p_filesz + offset,
                         PGSIZE) / PGSIZE;
      copyed_byte = 0;
      filesz = ph->p_filesz;
      for (j = 0; j < page_num; j++) {
        va = (uint8_t *)
             (KADDR(PTE_ADDR(
              pte[PTX(origin_va)])) + offset);
        origin_va += PGSIZE;
        if ((filesz + offset) > PGSIZE) {
          filesz -= PGSIZE - offset;
          byte_num = PGSIZE - offset;
        }
        else {
          byte_num = filesz;
```

```
        }

        offset = 0;
        memcpy(va,
               binary + ph->p_offset + copyed_byte,
               byte_num);
        copyed_byte += byte_num;
      }

      if (copyed_byte != ph->p_filesz)
        panic("Load_icode failed\n");
    }
  }

  // Set up the environment's trapframe to
  // point to the right location;
  // Other values for the trap frame as assigned
  // in env_alloc

  e->env_tf.tf_eip = elf->e_entry;

  // Now map one page for the program's initial
  // stack at virtual address USTACKTOP - PGSIZE.

  // LAB 3: Your code here.
  segment_alloc(e,
                (uintptr_t*)(USTACKTOP - PGSIZE),
                PGSIZE);
}
```

在该函数中, 接受内核传入的用户程序的所在地址 binary(内核地址), 然后为其创建用户进程空间, 并且将其载入到相应的虚拟地址上。接下来的 env_run() 就可以开始真正的运行一个程序了。

```
void
env_create(uint8_t *binary, size_t size)
{
    // LAB 3: Your code here.
    struct Env *env;
    env_alloc(&env, 0);
    load_icode(env, binary, size);
    return;
}
```

```
void
env_run(struct Env *e)
{
    // LAB 3: Your code here.
    curenv = e;
    curenv->env_runs++;
    lcr3(curenv->env_cr3);
    env_pop_tf(&(curenv -> env_tf));
}
```

Exercise 3

Description

编辑 trapentry.S 和 trap.c 文件, 实现如下功能要求。 其中文件中的 trapentry.S 中的宏 TRAPHANDLER 和 TRAPHANDLER_NOEC 和 inc/trap.c 中所定义的 T_* 系列的宏定义可能会对你有所帮助。

. 你需要在 trapentry.S 中为在 inc/trap.h 中所定义的每一个陷阱添加一个入口点(可能需要用到上面介绍的宏)。 除此之外, 你还需要修改 idt_init() 来初始化 IDT 的指针, 将其余这些入口点关联起来, 宏 SETGATE 可能会有所帮助。你的代码应该完成如下工作:

```
.c
push values to make the stack look like a struct Trapframe
load GD_KD into %ds and %es
pushl %esp to pass a pointer to the Trapframe as an
argument to trap()
call trap
pop the values pushed in steps 1-3
iret
```

相关提示: 可以使用 pushal 和 popal 指令, 他们和 Trapframe 的结构配合的很好。 在本次作业中, 还可能需要阅读 Intel®64 and IA-32 Architectures Software Developer's Manual 3A 的5.3 章节。

完成之后, 使用 user 目录下的测试程序进行测试, 例如 user/divzero。 运行的方法是 make run-divzero, 如果要运行其他的程序, 只需要替换 devzero 即可。 如果使用 make grade 进行测试, 那么应该可以通过 divzero、 softint 和 badsegment 这几个测试了(15/60)。

Solution

该练习分为两步:

- 在 kern/trapentry.S 中定义好每个中断对应的中断处理程序;
- 在 kern/trap.c 的 idt init() 中将那些第一步定义好的中断处理程序安装进IDT。

在 kern/trapentry.S 中 JOS 提供了以下两个宏, 其功能是接受一个函数名和对应处理的中断向量编号, 然后定义出一个相应的以该函数名命名的中断处理程序。这样的中断向量程序的执行流程就是向栈里压入相关错误码和中断号, 然后跳转到 alltraps 来执行共有的部分。

```
/* The TRAPHANDLER macro defines a globally-
 * visible function for handling a trap.
 * It pushes a trap number onto the stack,
 * then jumps to _alltraps.
 * Use TRAPHANDLER for traps where the CPU
 * automatically pushes an error code.
 */

#define TRAPHANDLER(name, num)
  // define global symbol for 'name'
```

```
  .globl name;
  // symbol type is function
  .type name, @function;
  // align function definition
  .align 2;
  //function starts here
  name:
  pushl $(num);
  jmp _alltraps

/* Use TRAPHANDLER_NOEC for traps where the
 * CPU doesn't push an error code.
 * It pushes a 0 in place of the error code,
 * so the trap frame has the same
 * format in either case.
 */

#define TRAPHANDLER_NOEC(name, num)
  .globl name;
  .type name, @function;
  .align 2;
  name:
  pushl $0;
  pushl $(num);
  jmp _alltraps
```

压栈时注意数据的长度大小选择对应的指令; 中断处理程序定义:

Listing 1: kern/trapentry.S

```
.text

    // Lab 3: Your code here for generating
    // entry points for the different traps.

    TRAPHANDLER_NOEC(routine_divide, T_DIVIDE)
    TRAPHANDLER_NOEC(routine_debug, T_DEBUG)
    TRAPHANDLER_NOEC(routine_nmi, T_NMI)
    TRAPHANDLER_NOEC(routine_brkpt, T_BRKPT)
    TRAPHANDLER_NOEC(routine_oflow, T_OFLOW)
    TRAPHANDLER_NOEC(routine_bound, T_BOUND)
    TRAPHANDLER_NOEC(routine_illop, T_ILLOP)
    TRAPHANDLER_NOEC(routine_device, T_DEVICE)
    TRAPHANDLER(routine_dblflt, T_DBLFLT)
    TRAPHANDLER(routine_tss, T_TSS)
    TRAPHANDLER(routine_segnp, T_SEGNP)
    TRAPHANDLER(routine_stack, T_STACK)
    TRAPHANDLER(routine_gpflt, T_GPFLT)
    TRAPHANDLER(routine_pgflt, T_PGFLT)
    TRAPHANDLER_NOEC(routine_fperr, T_FPERR)
    TRAPHANDLER(routine_align, T_ALIGN)
    TRAPHANDLER_NOEC(routine_mchk, T_MCHK)
    TRAPHANDLER_NOEC(routine_simderr, T_SIMDERR)
```

```
    TRAPHANDLER_NOEC(routine_system_call, T_SYSCALL)

    // Lab 3: Your code here for _alltraps

     _alltraps:
    pushl %ds;
    pushl %es
    pushal
    movl $GD_KD, %eax
    movw %ax,%ds
    movw %ax,%es
    pushl %esp
    call trap
    popl %esp
    popal
    popl %es
    popl %ds
    addl $8, %esp
    iret
```

定义好中断服务程序以后, 开始安装 IDT 表 :

Listing 2: kern/trap.c

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

    // LAB 3: Your code here.
    extern void routine_divide ();
    extern void routine_debug  ();
    extern void routine_nmi    ();
    extern void routine_brkpt  ();
    extern void routine_oflow  ();
    extern void routine_bound  ();
    extern void routine_illop  ();
    extern void routine_device ();
    extern void routine_dblflt ();
    extern void routine_tss    ();
    extern void routine_segnp  ();
    extern void routine_stack  ();
    extern void routine_gpflt  ();
    extern void routine_pgflt  ();
    extern void routine_fperr  ();
    extern void routine_align  ();
    extern void routine_mchk   ();
    extern void routine_simderr();
    extern void routine_syscall();

    SETGATE (idt[T_DIVIDE], 0, GD_KT,
    routine_divide, 0);
    SETGATE (idt[T_DEBUG],  0, GD_KT,
    routine_debug,  0);
```

```
    SETGATE (idt[T_NMI],    0, GD_KT,
    routine_nmi,    0);

    // break point needs no kernel mode privilege
    SETGATE (idt[T_BRKPT], 0, GD_KT,
    routine_brkpt, 3);

    SETGATE(idt[T_OFLOW], 0, GD_KT,
            routine_oflow, 0);
    SETGATE(idt[T_BOUND], 0, GD_KT,
            routine_bound, 0);
    SETGATE(idt[T_ILLOP], 0, GD_KT,
            routine_illop, 0);
    SETGATE(idt[T_DEVICE], 0, GD_KT,
            routine_device, 0);
    SETGATE(idt[T_DBLFLT], 0, GD_KT,
            routine_dblflt, 0);
    SETGATE(idt[T_TSS], 0, GD_KT,
            routine_tss, 0);
    SETGATE(idt[T_SEGNP], 0, GD_KT,
            routine_segnp, 0);
    SETGATE(idt[T_STACK], 0, GD_KT,
            routine_stack, 0);
    SETGATE(idt[T_GPFLT], 0, GD_KT,
            routine_gpflt, 0);
    SETGATE(idt[T_PGFLT], 0, GD_KT,
            routine_pgflt, 0);
    SETGATE(idt[T_FPERR], 0, GD_KT,
            routine_fperr, 0);
    SETGATE(idt[T_ALIGN], 0, GD_KT,
            routine_align, 0);
    SETGATE(idt[T_MCHK], 0, GD_KT,
            routine_mchk, 0);
    SETGATE(idt[T_SIMDERR], 0, GD_KT,
            routine_simderr, 0);

    extern void routine_system_call();
    SETGATE(idt[T_SYSCALL], 0, GD_KT,
            routine_system_call, 3);

    // Setup a TSS so that we get the right stack
    // when we trap to the kernel.
    ts.ts_esp0 = KSTACKTOP;
    ts.ts_ss0 = GD_KD;

    // Initialize the TSS field of the gdt.
    gdt[GD_TSS >> 3] = SEG16(STS_T32A,
                        (uint32_t) (&ts),
                        sizeof(struct Taskstate), 0);
    gdt[GD_TSS >> 3].sd_s = 0;

    // Load the TSS
    ltr(GD_TSS);
```

```
85      // Load the IDT
        asm volatile("lidt idt_pd");
}
```

练习2: 请尝试回答如下问题:

1. What is the purpose of having an individual handler function for each exception/interrupt? (i.e., if all exceptions/interrupts were delivered to the same handler, what feature that exists in the current implementation could not be provided?)

现在 JOS 的中断处理程序在处理之前要将其中断号放入内核栈以组织成 Trapframe 的结构; 如果所有中断都跳转到同一个 handler, 那么就无法区分是哪类中断调用进来的, 也就无法正确设置它们的中断号。

2. Did you have to do anything to make the user/softint program behave correctly (i.e., as the grade script expects)? Why is this the correct behavior? What happens if the kernel actually allows softint's int $14 instruction to invoke the kernel's page fault handler (which is interrupt number 14)?

现在我们在中断向量里设置的14号 Page fault 的调用权限是0, 即只能内核抛出, 所以直接在softint中用int指令调用肯定产生的 General Protection Fault 权限错误。

Exercise 4

Description

缺页中断, 也叫页面失效, 中断号为 14(T_PGFLT), 是十分重要的一个中断。当处理器遇到缺页中断时, 它将导致这个中断的线性地址保存在一个特殊的寄存器 cr2 中。 在文件 trap.c 中提供了函数 page_fault_handler 来处理缺页中断。

修改 trap_dispatch() 将缺页中断交由 page_fault_handler 来处理。

相关提示:完成之后,进行 make grade 进行测试此时faultread, faultreadkernel, faultwrite 和 faultwritekernel 都可以通过了. 如果有任何一个无法正常完成请正确的修改完成再往下执行。

The Breakpoint Exception

断点异常, 中断号为 3(T_BRKPT), 通常是用来使调试器可以在程序的任意位置设置断点, 调试器临时将该处的指令替换为一字节的int3 软中断指令。JOS 中, 在我们扩大它的使用范围,将其设置为原始的伪系统调用(primitive pseudo-system call), 使得任何用户都可以调用它来引用 JOS 的内核监视器(JOS kernel monitor)。 在用户模式下实现的 panic() 函数在输出信息之后就会使用 int 3 中断。

Solution

```
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);
    ...
}
```

Exercise 5

Description

修改 trap_dispatch() 函数, 使得断点异常能够触发内核的监视器(kernel monitor)。

相关提示:完成之后,进行make grade 进行测试此时就可以通过break point 测试了,可以通过(40/60).

Solution

Listing 3: kern/trap.c: trap dispatch()

```
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);
    if (tf->tf_trapno == T_BRKPT)
        monitor (tf);
    if (tf->tf_trapno == T_DEBUG)
        monitor (tf);
    int r;
    if (tf->tf_trapno == T_SYSCALL) {
        r = 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 (r < 0)
            panic("trap_dispatch:\
                  The System Call number is invalid");

        tf->tf_regs.reg_eax = r;
```

```
25          return;
        }

        // Unexpected trap: The user process or the kernel
        // has a bug.
30      print_trapframe(tf);
        if (tf->tf_cs == GD_KT)
            panic("unhandled trap in kernel");
        else {
            env_destroy(curenv);
35          return;
        }
}
```

练习3: 请尝试回答如下问题:

1. The break point test case will either generate a break point exception of a general protect fault depending on how you initialized the break point entry in the IDT (i.e., your call to SETGATE from idt_init). Why? How did you need to set it in order to get the breakpoint exception to work as specified above?

2. What do you think is the point of these mechanisms, particularly in light of what the user/softint test program does?

Exercise 6

Description

在内核中为T_SYSCALL添加一个处理程序。

. 修改 kern/trapentry 和 kern/trap.c 中的 idt_init();

. 修改 trap_dispatch(), 调用 syscall() 来处理系统调用，并设置参数和返回值;

. 实现 kern/syscall.c 中的 syscall() 函数, 如果系统调用编号不合法时,返回E_INVAL;

相关提示: 文件 lib/syscall.c 和 inc/syscall.h 可能对你有所提示。在你的内核中运行 user/hello 程序 (make run-hello), 它应该可以在控制台上输出 “hello, world”, 然后就会在用户模式下引发缺页中断。 如果没有产生这个中断,请检查你前面的代码。 请确认该程序能够正确执行。

User-mode startup

用户程序在 lib/entry.S 的顶部开始运行。 进行一些设置之后, 这部分代码会调用lib/libmain.c 中的 libmain() 函数, 这个函数负责初始化全局的指向这个程序在 envs[] 数组中的Env 结构的 env 指针 (可查看inc/env.h 和使用 sys_getenvid)。之后, libmain() 调用 umain, 假设当前是运行 hello 程序,那么 umain 就在 user/hello.c 中。在输出”hello, world”之后,它就会访问 env-¿env_id。 这也是它为什么会在上面的实验中出现缺页中断的原因。 现在我们

已经设置了 env, 所以就不会出错了。如果出错了, 也许是你没有将 UENVS 设置为可读, 返回 lab1 中的 pmap.c 文件中仔细的揣摩一下吧。

Solution

在该练习中我们开始处理系统中断, 首先修改 kern/trap.c 中的 idt init() 以及 kern/trapentry.S 添加相应的中断服务程序和中断向量, 然后修改 kern/-trap.c.

```
// Lab 3: Your code here for generating entry points
// for the different traps.

    TRAPHANDLER_NOEC(routine_divide, T_DIVIDE)
    TRAPHANDLER_NOEC(routine_debug, T_DEBUG)
    TRAPHANDLER_NOEC(routine_nmi, T_NMI)
    TRAPHANDLER_NOEC(routine_brkpt, T_BRKPT)
    TRAPHANDLER_NOEC(routine_oflow, T_OFLOW)
    TRAPHANDLER_NOEC(routine_bound, T_BOUND)
    TRAPHANDLER_NOEC(routine_illop, T_ILLOP)
    TRAPHANDLER_NOEC(routine_device, T_DEVICE)
    TRAPHANDLER(routine_dblflt, T_DBLFLT)
    TRAPHANDLER(routine_tss, T_TSS)
    TRAPHANDLER(routine_segnp, T_SEGNP)
    TRAPHANDLER(routine_stack, T_STACK)
    TRAPHANDLER(routine_gpflt, T_GPFLT)
    TRAPHANDLER(routine_pgflt, T_PGFLT)
    TRAPHANDLER_NOEC(routine_fperr, T_FPERR)
    TRAPHANDLER(routine_align, T_ALIGN)
    TRAPHANDLER_NOEC(routine_mchk, T_MCHK)
    TRAPHANDLER_NOEC(routine_simderr, T_SIMDERR)

    TRAPHANDLER_NOEC(routine_system_call, T_SYSCALL)

// Lab 3: Your code here for _alltraps

     _alltraps:
    pushl %ds;
    pushl %es
    pushal
    movl $GD_KD, %eax
    movw %ax,%ds
    movw %ax,%es
    pushl %esp
    call trap
    popl %esp
    popal
    popl %es
    popl %ds
    addl $8, %esp
    iret
```

```
void
idt_init(void)
```

```
{
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.
    extern void routine_divide ();
    extern void routine_debug ();
    extern void routine_nmi ();
    extern void routine_brkpt ();
    extern void routine_oflow ();
    extern void routine_bound ();
    extern void routine_illop ();
    extern void routine_device ();
    extern void routine_dblflt ();
    extern void routine_tss ();
    extern void routine_segnp ();
    extern void routine_stack ();
    extern void routine_gpflt ();
    extern void routine_pgflt ();
    extern void routine_fperr ();
    extern void routine_align ();
    extern void routine_mchk ();
    extern void routine_simderr ();
    extern void routine_syscall();

    SETGATE (idt[T_DIVIDE], 0, GD_KT, routine_divide, 0);
    SETGATE (idt[T_DEBUG],  0, GD_KT, routine_debug,  0);
    SETGATE (idt[T_NMI],    0, GD_KT, routine_nmi,    0);

    // break point needs no kernel mode privilege
    SETGATE (idt[T_BRKPT], 0, GD_KT, routine_brkpt, 3);

    SETGATE (idt[T_OFLOW], 0, GD_KT, routine_oflow, 0);
    SETGATE (idt[T_BOUND], 0, GD_KT, routine_bound, 0);
    SETGATE (idt[T_ILLOP], 0, GD_KT, routine_illop, 0);
    SETGATE (idt[T_DEVICE], 0, GD_KT, routine_device, 0);
    SETGATE (idt[T_DBLFLT], 0, GD_KT, routine_dblflt, 0);
    SETGATE (idt[T_TSS], 0, GD_KT, routine_tss, 0);
    SETGATE (idt[T_SEGNP], 0, GD_KT, routine_segnp, 0);
    SETGATE (idt[T_STACK], 0, GD_KT, routine_stack, 0);
    SETGATE (idt[T_GPFLT], 0, GD_KT, routine_gpflt, 0);
    SETGATE (idt[T_PGFLT], 0, GD_KT, routine_pgflt, 0);
    SETGATE (idt[T_FPERR], 0, GD_KT, routine_fperr, 0);
    SETGATE (idt[T_ALIGN], 0, GD_KT, routine_align, 0);
    SETGATE (idt[T_MCHK], 0, GD_KT, routine_mchk, 0);
    SETGATE (idt[T_SIMDERR], 0, GD_KT, routine_simderr, 0);

    extern void routine_system_call();
    SETGATE(idt[T_SYSCALL], 0, GD_KT, routine_system_call, 3);

    // Setup a TSS so that we get the right stack
    // when we trap to the kernel.
    ts.ts_esp0 = KSTACKTOP;
    ts.ts_ss0 = GD_KD;
```

```
    // Initialize the TSS field of the gdt.
    gdt[GD_TSS >> 3] = SEG16(STS_T32A, (uint32_t) (&ts),
                         sizeof(struct Taskstate), 0);
    gdt[GD_TSS >> 3].sd_s = 0;

    // Load the TSS
    ltr(GD_TSS);

    // Load the IDT
    asm volatile("lidt idt_pd");
}
```

Listing 4: kern/syscall.c

```
uint32_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 r = 0;
    switch (syscallno) {
        case SYS_cputs:
            sys_cputs ((const char*) a1, (size_t)a2);
            break;
        case SYS_cgetc:
            r = sys_cgetc ();
            break;
        case SYS_getenvid:
            r = sys_getenvid ();
            break;
        case SYS_env_destroy:
            r = sys_env_destroy ((envid_t) a1);
            break;
        default:
            r = -E_INVAL;
    }
    return r;
}
```

Exercise 7

Description

添加 user library 中所需要的代码(lib/libmain.c), 然后启动内核(make run-hello)。 首先, 程序 user/hello 输出 "hello, world", 然后输出 "i am environment 00000800"。之后该程序就会调用 sys_env_destroy() 来退出(可查阅

lib/libmain.c 和lib/exit.c)。 由于内核当前只支持一个程序, 所以当前程序退出后, 内核就会显示当前唯一的程序已经退出并且陷入内核监视器。

程序的相关提示: 使用 make grade 命令, 此时就可以通过 hello 测试了,应该显示为 (50/60)。

Page faults and memory protection

存储保护是操作系统的一项至关重要的功能。通过存储保护, 操作系统可以保证一个错误的程序不会影响到其他的程序或者操作系统本身。

Solution

只需在第 6 行更改env，其中 sys_getenvid() 是lib中包装好的函数, 在其内部是使用调用系统中断实现的。可以正常运行 make run-hello.

```
void
libmain(int argc, char **argv)
{
    // set env to point at our env structure in envs[].
    // LAB 3: Your code here.
    env = envs + ENVX(sys_getenvid ());
    // save the name of the program so that panic() can use it
    if (argc > 0)
        binaryname = argv[0];

    // call user main routine
    umain(argc, argv);

    // exit gracefully
    exit();
}
```

Exercise 8

Description

完成如下内容:

. 修改文件 kern/trap.c, 使得在内核中出现缺页错误时, 就终止(使用panic)内核。 要检验缺页中断是否发生在内核模式可以检查tf_cs的最低几位;

. 阅读文件 kern/pmap.c 中的 user_mem_assert() 并实现同文件中的 user_mem_check();

. 修改文件 kern/syscall.c 来检查传递给内核的参数;

. 运行 user/buggyhello 并启动你的内核 (make run-buggyhello)。 调试你的内核, 然后启动它。 如果这样, 用户进程会被销毁而内核将会中止 (panic)。 你将会出现如下输出:

```
[00001000] user_mem_check assertion failure for va 00000001
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
```

程序的相关提示: 请注意, 如果此时运行 make grade 将出现 buggyhello 可以通过但是 hello 反而无法通过的情况, 可以考虑检查你的内存初始化部分的代码, 有可能在那里出现了权限设置、 物理地址映射错误等问题。

Solution

在 page fault handler() 中处理内核态中抛出页面错误:

Listing 5: kern/trap.c: page fault handler()

```
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 & 3) == 0)
        panic("kernel-mode page faults");
    ...
}
```

检测线性地址的页面是否有效:

Listing 6: kern/pmap.c: user mem check()

```
int
user_mem_check(struct Env *env, const void *va, size_t len,
               int perm)
{
    // LAB 3: Your code here.
    uintptr_t lva = (uintptr_t) va;
    uintptr_t rva = (uintptr_t) va + len - 1;
    perm = perm | PTE_U | PTE_P;
    pte_t *pte;
    uintptr_t idx_va;

    for (idx_va = lva; idx_va <= rva; idx_va += PGSIZE) {
        if (idx_va >= ULIM) {
            user_mem_check_addr = idx_va;
            return -E_FAULT;
        }

        pte = pgdir_walk (env->env_pgdir, (void*)idx_va, 0);

        if (pte == NULL || (*pte & perm) != perm) {
            user_mem_check_addr = idx_va;
            return -E_FAULT;
        }
```

```
25          idx_va = ROUNDDOWN (idx_va, PGSIZE);
    }

    return 0;
}
```

在 sys cputs() 中加入相应对用户空间地址的检查:

Listing 7: kern/syscall.c: sys cputs()

```
static void
sys_cputs(const char *s, size_t len)
{
    // Check that the user has permission to read
5   // memory [s, s+len).
    // Destroy the environment if not.

    // LAB 3: Your code here.
    user_mem_assert (curenv, s, len, 0);
10
    // Print the string supplied by the user.
    cprintf("%.*s", len, s);
}
```

Exercise 9

Description

修改 kern/init.c 然后

. 运行 user/evilhello 并重新编译并且启动你的内核。 用户程序会被销毁而内核仍然不会中止, 你可以看到如下输出:

```
[00000000] new env 00001000
[00001000] user_mem_check assertion failure for va f0100020
[00001000] free env 00001000
```

程序的相关提示: 见代码中相关注释。

至此, 本次作业的代码部分已经结束, 确认你可以通过全部的 make grade 测试 (60/60), 然后提交你的代码。

Solution

```
void
i386_init(void)
{
    ...
5   // Lab 3 user environment initialization functions
    env_init();
```

```
    idt_init();
    ...
}

// user/evilhello.c:
void
umain(void)
{
    // try to print the kernel entry point as a string!
    // mua ha ha!
    sys_cputs((char*)0xf010000c, 100);
}
```

Acknowledgement

Group work

武政伟(1110548) 完成了第 1,2 题，并撰写及整理了全部文档；

袁兆争(1110556)，张超(1110557) 完成了第 3-6 题代码及部分文档；

马璇(1110580)，王少尧(1110539) 完成了第 7-9 题代码及部分文档。