Created: April 18 2018 11:07:00 Last Updated: Apr 20 2018 18:07:35

Lab 3 Report

Shuyang Cao

Apr 20 2018

User Enviroments and Exception Handling

In JOS, the kernel uses `struct Env` to keep track of each user enviroment. More specifically, a `struct Env` contains an enviroment of a process. JOS can support `NENV` enviroments at most. All enviroments are stored at `envs` , an array of `struct Env` . The JOS kernel keeps all of the inactive `Env` structures on the `env_free_list` . Codes about enviroments are displayed and explained below.

```
/*
 * struct Env is defined in inc/env.h
 */

struct Env {
    struct Trapframe env_tf;    // Saved registers
    struct Env *env_link;       // Next free Env
    envid_t env_id;             // Unique environment identifier
    envid_t env_parent_id;      // env_id of this env's parent
    enum EnvType env_type;      // Indicates special system environments
    unsigned env_status;        // Status of the environment
    uint32_t env_runs;          // Number of times environment has run

    // Address space
    pde_t *env_pgdir;           // Kernel virtual address of page dir
};
```

Exercise 1

> Q : Modify mem_init() in kern/pmap.c to allocate and map the envs array. This array consists of exactly NENV instances of the Env structure allocated much like how you allocated the pages array. Also like the pages array, the memory backing envs should also be mapped user read-only at UENVS (defined in inc/memlayout.h) so user processes can read from this array.
> You should run your code and make sure check_kern_pgdir() succeeds.

There are three things we need to do:

1. Allocate pages for `envs` array.
2. Zero allocated pages.
3. Insert newly allocated pages to kernel page table.

Codes for the three things are shown below.

```
diff --git a/kern/pmap.c b/kern/pmap.c
index 082a495..2a37451 100644
--- a/kern/pmap.c
+++ b/kern/pmap.c
@@ -164,6 +164,8 @@ mem_init(void)
        //////////////////////////////////////////////////////////////////////
        // Make 'envs' point to an array of size 'NENV' of 'struct Env'.
        // LAB 3: Your code here.
+       envs = (struct Env *) boot_alloc(sizeof(struct Env) * NENV);
+       memset(envs, 0, sizeof(struct Env) * NENV);

        //////////////////////////////////////////////////////////////////////
        // Now that we've allocated the initial kernel data structures, we set
@@ -196,6 +198,7 @@ mem_init(void)
        //    - the new image at UENVS  -- kernel R, user R
        //    - envs itself -- kernel RW, user NONE
        // LAB 3: Your code here.
+       boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR((void *)envs), PTE_U);

        //////////////////////////////////////////////////////////////////////
        // Use the physical memory that 'bootstack' refers to as the kernel
```

Creating and Running Environments

To test our implementation for creating and running enviroments, JOS kernel needs to load a user program and run it. Since we do not yet have a filesystem, JOS kernel loads a static binary image that is embedded within the kernel itself. During compilation, a user program is first compiled to an ELF file, then the ELF file is inserted into JOS kernel wihtout modification when JOS kernel is linked.

When JOS kernel want to load a program, it will parse the content in memory starting from a pointer assigned by linker. Then JOS kernel copy some of the content to a user space memory when necessary. After all of these have been done, a user program resides at the proper place in memory and waits for execution.

Exercise 2

> Q : In the file env.c, finish coding the following functions:
>
> `env_init()`
>
> > Initialize all of the Env structures in the envs array and add them to the env_free_list. Also calls env_init_percpu, which configures the segmentation hardware with separate segments for privilege level 0 (kernel) and privilege level 3 (user).
>
> `env_setup_vm()`
>
> > Allocate a page directory for a new environment and initialize the kernel portion of the new environment’s address space.
>
> `region_alloc()`
>
> > Allocates and maps physical memory for an environment
>
> `load_icode()`

> > You will need to parse an ELF binary image, much like the boot loader already does, and load its contents into the user address space of a new environment.
>
> `env_create()`
>
> > Allocate an environment with env_alloc and call load_icode to load an ELF binary into it.
>
> `env_run()`
>
> > Start a given environment running in user mode.

What we need to do in `env_init()` is similar to what we do in `page_init()` . We go through an array and establish a free list. It's required that the environments are in the free list in the same order they are in the `envs` array. So we need to go through the `envs` array backwards.

After an enviroment is allocated, we need to initialized its page table. Since JOS don't switch to a kernel page when execution is trapped into kernel, the page table of an enviroment needs to contain the content of kernel page table. So we need to copy kernel page table to a user page table. Furthermore, kernel accesses memory using only the last 256MB of 4GB memory space. We only need to copy content indexed from `UTOP` . In addition, JOS use a 2-level page table. So we only need to copy the page directory.

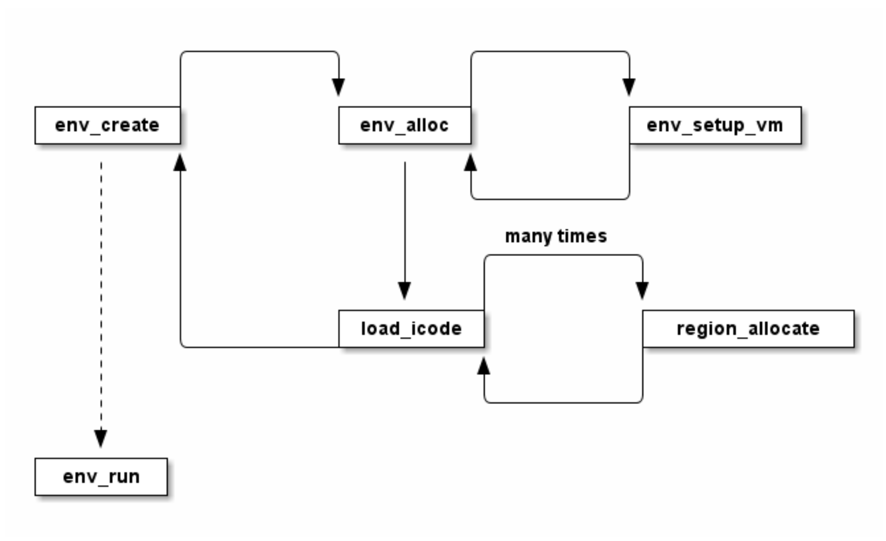
`region_alloc()` is a wrapper for a bunch of page allocation and inserting in a user page table. It need to align addresses passed to it by page.

`load_icode()` is the function that load a user program. In lab 3, it loads the user program in a strange way that we've explained above.

The task of `env_create()` is very simple. It allocates a new enviroment, load the program. It should panic if the allocation failed.

The task of `env_run()` is even simpler. It first substitue the enviroment passed to it it `curenv` . Then it replace `kern_pgdir` with enviroment page directory. Last it pops out register value stored in `struct Env` and jump to excuting the user program using `iret` . Of cousre, it should check the sanity of the arguments at the very beginning.

These functions cooperate in the way shown below.



Codes for exercise 2 are shown below.

```
diff --git a/kern/env.c b/kern/env.c
index db2fda9..3d5f35a 100644
--- a/kern/env.c
+++ b/kern/env.c
@@ -116,6 +116,23 @@ env_init(void)
 {
        // Set up envs array
        // LAB 3: Your code here.
+       // In current JOS configuration, we actually don't need to
+       // set env_id = 0, env_status = ENV_FREE (0). Because we
+       // zeroed envs array when we allocated it. However, to reserve
+       // the freedome to change initialized values of env_id and
+       // ENV_FREE as well as to add new tests in mem_init() that
+       // would set nonzero values to envs array, we'd better to
+       // initialize env_id and env_status again here. By the way,
+       // this function will be invoked only once when booting
+       // computers and will never be invoked since then. So
+       // redundant code won't affect the runtime performance of JOS.
+       env_free_list = NULL;
+       for (int i = NENV - 1; i >= 0; --i) {
+               envs[i].env_link = env_free_list;
+               envs[i].env_id = 0;
+               envs[i].env_status = ENV_FREE;
+               env_free_list = &envs[i];
+       }

        // Per-CPU part of the initialization
        env_init_percpu();
@@ -179,7 +196,14 @@ env_setup_vm(struct Env *e)
        //    - The functions in kern/pmap.h are handy.

        // LAB 3: Your code here.
-
+       ++(p->pp_ref);
+       e->env_pgdir = (pde_t *) page2kva(p);
+
+       // We set ALLOC_ZERO flag when we page_alloc() a page directory.
+       // So VA below UTOP is empty already.
+       // can be improved
+       memcpy(e->env_pgdir+PDX(UTOP), kern_pgdir+PDX(UTOP), PGSIZE-PDX(UTOP)*sizeof(pde_t));
+
        // 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;
@@ -267,6 +291,19 @@ region_alloc(struct Env *e, void *va, size_t len)
        //   'va' and 'len' values that are not page-aligned.
        //   You should round va down, and round (va + len) up.
        //   (Watch out for corner-cases!)
+       void *start = (void *) ROUNDDOWN((uint32_t) va, PGSIZE);
+       void *end = (void *) ROUNDUP((uint32_t)(va + len), PGSIZE);
+       struct PageInfo *pp_ptr;
+       int err_code;
+       while (start < end) {
+               if ((pp_ptr = page_alloc(0)) == NULL)
+                       panic("region alloc, allocation failed.");
+
+               if ((err_code = page_insert(e->env_pgdir, pp_ptr, start, PTE_U | PTE_W)) != 0)
+                       panic("region alloc: %e", err_code);
+
+               start += PGSIZE;
+       }
 }

 //
@@ -323,11 +360,36 @@ load_icode(struct Env *e, uint8_t *binary)
        //  What?  (See env_run() and env_pop_tf() below.)

        // LAB 3: Your code here.
+       struct Elf *elfhdr = (struct Elf *) binary;
+
+       if (elfhdr->e_magic != ELF_MAGIC)
+               panic("the binary to be loaded is not a elf file.");
+
+       lcr3(PADDR(e->env_pgdir));      // program should be mapped at user page directory.
+
+       struct Proghdr *ph = (struct Proghdr *) ((uint8_t *)elfhdr + elfhdr->e_phoff);
+       struct Proghdr *eph = (struct Proghdr *) (ph + elfhdr->e_phnum);
```

```
+
+       for (; ph < eph; ++ph) {
+               if (ph->p_type == ELF_PROG_LOAD) {
+                       if (ph->p_filesz > ph->p_memsz)
+                               panic("file size is great than memory size");
+
+                       region_alloc(e, (void *)(ph->p_va), ph->p_memsz);
+                       memcpy((void *)(ph->p_va), binary + ph->p_offset, ph->p_memsz);
+                       memset((void *)(ph->p_va) + ph->p_filesz, 0, ph->p_memsz - ph->p_filesz);
+               }
+       }
+
+       e->env_tf.tf_eip = elfhdr->e_entry;

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

        // LAB 3: Your code here.
+       region_alloc(e, (void *)(USTACKTOP - PGSIZE), PGSIZE);
+
+       lcr3(PADDR(kern_pgdir));  // load kern_pgdir back
 }

 //
@@ -341,6 +403,13 @@ void
 env_create(uint8_t *binary, enum EnvType type)
 {
        // LAB 3: Your code here.
+       struct Env *e;
+       int err_code;
+       if ((err_code = env_alloc(&e, 0)) != 0)
+               panic("env_alloc: %e", err_code);
+
+       load_icode(e, binary);
+       e->env_type = type;
 }

 //
@@ -458,6 +527,17 @@ env_run(struct Env *e)

        // LAB 3: Your code here.

-       panic("env_run not yet implemented");
+       // curenv != NULL means this is a context switch
+       if (curenv != NULL && curenv->env_status == ENV_RUNNING)
+               curenv->env_status = ENV_RUNNABLE;
+
+       curenv = e;
+       curenv->env_status = ENV_RUNNING;
+       ++(curenv->env_runs);
+       lcr3(PADDR(e->env_pgdir));
+
+       env_pop_tf(&(curenv->env_tf));
+
+       // panic("env_run not yet implemented");
 }
```

Test Allocating and Running Enviroments

We need to confirm that JOS kernel can run as far as the `int $0x30` now. The `int $0x30` resides at 0x800a1c in “hello” program. The gdb log is listed below.

```
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) break env_pop_tf
Breakpoint 1 at 0xf0102f3f: file kern/env.c, line 490.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0xf0102f3f <env_pop_tf>:     push   %ebp

Breakpoint 1, env_pop_tf (tf=0xf01b2000) at kern/env.c:490
490     {
(gdb) si
=> 0xf0102f40 <env_pop_tf+1>:   mov    %esp,%ebp
0xf0102f40      490     {
(gdb)
=> 0xf0102f42 <env_pop_tf+3>:   sub    $0xc,%esp
0xf0102f42      490     {
(gdb)
=> 0xf0102f45 <env_pop_tf+6>:   mov    0x8(%ebp),%esp
491             asm volatile(
(gdb)
=> 0xf0102f48 <env_pop_tf+9>:   popa
0xf0102f48      491             asm volatile(
(gdb)
=> 0xf0102f49 <env_pop_tf+10>:  pop    %es
0xf0102f49 in env_pop_tf (
    tf=<error reading variable: Unknown argument list address for `tf'.>)
    at kern/env.c:491
491             asm volatile(
(gdb)
=> 0xf0102f4a <env_pop_tf+11>:  pop    %ds
0xf0102f4a      491             asm volatile(
(gdb)
=> 0xf0102f4b <env_pop_tf+12>:  add    $0x8,%esp
0xf0102f4b      491             asm volatile(
(gdb)
=> 0xf0102f4e <env_pop_tf+15>:  iret
0xf0102f4e      491             asm volatile(
(gdb)
=> 0x800020:    cmp    $0xeebfe000,%esp
0x00800020 in ?? ()
(gdb) b *0x800a1c
Breakpoint 2 at 0x800a1c
(gdb) c
Continuing.
=> 0x800a1c:    int    $0x30

Breakpoint 2, 0x00800a1c in ?? ()
(gdb) c[Ksi
The target architecture is assumed to be i8086
[f000:e05b]    0xfe05b: cmpl   $0x0,%cs:0x6c48
0x0000e05b in ?? ()
```

Handling Interrupts and Exceptions

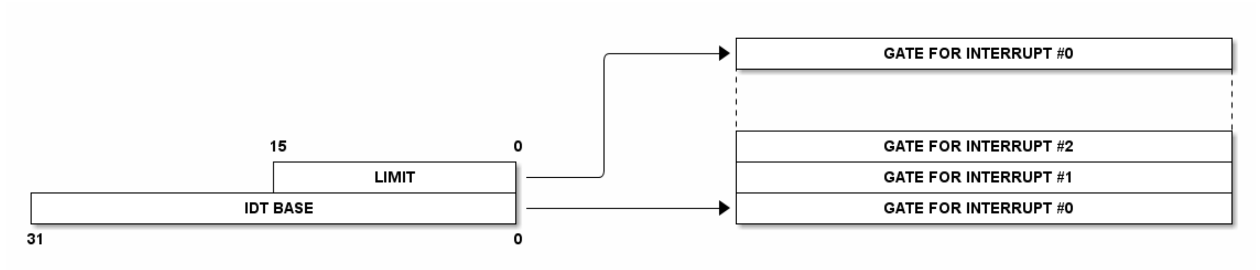
Basics of Protected Control

Exceptions and interrupts are both “protected control transfers”, which cause the processor to switch from user to kernel mode (CPL=0) without giving the user-mode code any opportunity to interfere with the functioning of the kernel or other enviroments. In Intel’s terminology, an interrupt is a protected control transfer that is caused by an asynchronous event usually external to the processor, such as notification of external device I/O activity. An exception, in contrast, is a protected control transfer caused sysnchronously by the currently running code, for example due to a divide by zero or an invalid memory access.(Ref: Lab 3). The processor uses the same mechanism to handle exceptions and interrupts except that interrupt is disabled when handling interrupts while interrupt remains enabled when handling exceptions.

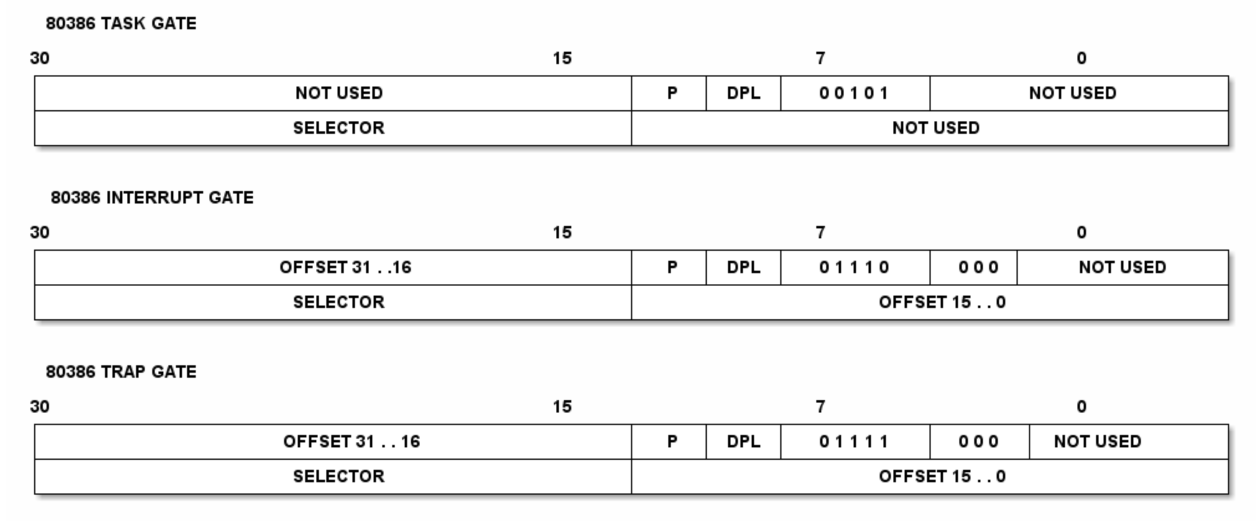
In order to ensure that these protected control transfers are actually protected, the processor’s interrupt/exception mechanism is designed so that the code currently running when the interrupt or exception occurs does not get to choose arbitrarily where the kernel is entered or how. Instead, the processor ensures that the kernel can be entered only under carefully controlled conditions. On x86, two mechanisms work together to privide this protection (Ref: Lab 3):

1. ***The Interrupt Descriptor Table*** The processor ensures that interrupts and exceptions can only cause the kernel to be entered at a few specific, well-defined entry-points determined by the kernel itself, and not by the code running when the interrupt or exception is taken.
   The x86 allows up to 256 different interrupt or exception entry points into the kernel, each with a different interrupt vector. A vector is a number between 0 and 255. An interrupt's vector is determined by the source of the interrupt: different devices, error conditions, and application requests to the kernel generate interrupts with different vectors. The CPU uses the vector as an index into the processor's interrupt descriptor table (IDT), which the kernel sets up in kernel-private memory, much like the GDT. From the appropriate entry in this table the processor loads:
   - the value to load into the instruction pointer (EIP) register, pointing to the kernel code designated to handle that type of exception.
   - the value to load into the code segment (CS) register, which includes in bits 0-1 the privilege level at which the exception handler is to run. (In JOS, all exceptions are handled in kernel mode, privilege level 0.)

   All of the sysnchrouus exceptions that the x86 processor can generate internally use interrupt vectors between 0 and 31, and therefore map to IDT entries 0-31. Interrupt vectors greater than 31 are only used by software interrupts, which can be generated by the int instruction, or asynchronous hardware interrupts, caused by external devices when they need attention.
   On x86, a IDT register (48 bits) stores the entry number and the address of IDT, which is shown below.



   The IDT may contain any of three kinds of descriptor:
   - Task gates
   - Interrupt gates
   - Trap gates
     Their format are shown below.



2. ***The Task State Segment*** The processor needs a place to save the old processor state before the interrupt or exception occurred, such as the original values of EIP and CS before the processor invoked the exception handler, so that the exception handler can later restore that old state and resume the interrupted code from where it left off. But this save area for the old processor state must in turn be protected from unprivileged user-mode code; otherwise buggy or malicious user code could compromise the kernel.
   For this reason, when an x86 processor takes an interrupt or trap that causes a privilege level change from user to kernel mode, it also switches to a stack in the kernel's memory. A structure called the task state segment (TSS) specifies the segment selector and address where this stack lives. The processor pushes (on this new stack) SS, ESP, EFLAGS, CS, EIP, and an optional error code. Then it loads the CS and EIP from the interrupt descriptor, and sets the ESP and SS to refer to the new stack.
   Although the TSS is large and can potentially serve a variety of purposes, JOS only uses it to define the kernel stack that the processor should switch to when it transfers from user to kernel mode. Since "kernel mode" in JOS is privilege level 0 on the x86, the

processor uses the ESP0 and SS0 fields of the TSS to define the kernel stack when entering kernel mode. JOS doesn't use any other TSS fields.
In JOS, TSS is defined as below.

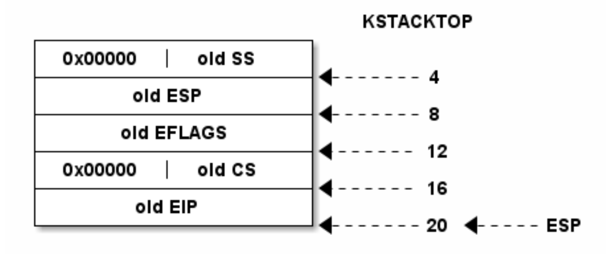
```
// Task state segment format (as described by the Pentium architecture book)
struct Taskstate {
    uint32_t ts_link;           // Old ts selector
    uintptr_t ts_esp0;          // Stack pointers and segment selectors
    uint16_t ts_ss0;            //   after an increase in privilege level
    uint16_t ts_padding1;
    uintptr_t ts_esp1;
    uint16_t ts_ss1;
    uint16_t ts_padding2;
    uintptr_t ts_esp2;
    uint16_t ts_ss2;
    uint16_t ts_padding3;
    physaddr_t ts_cr3;          // Page directory base
    uintptr_t ts_eip;           // Saved state from last task switch
    uint32_t ts_eflags;
    uint32_t ts_eax;            // More saved state (registers)
    uint32_t ts_ecx;
    uint32_t ts_edx;
    uint32_t ts_ebx;
    uintptr_t ts_esp;
    uintptr_t ts_ebp;
    uint32_t ts_esi;
    uint32_t ts_edi;
    uint16_t ts_es;             // Even more saved state (segment selectors)
    uint16_t ts_padding4;
    uint16_t ts_cs;
    uint16_t ts_padding5;
    uint16_t ts_ss;
    uint16_t ts_padding6;
    uint16_t ts_ds;
    uint16_t ts_padding7;
    uint16_t ts_fs;
    uint16_t ts_padding8;
    uint16_t ts_gs;
    uint16_t ts_padding9;
    uint16_t ts_ldt;
    uint16_t ts_padding10;
    uint16_t ts_t;              // Trap on task switch
    uint16_t ts_iomb;           // I/O map base address
};
```

Overall Interrupt/Exception Handling Process

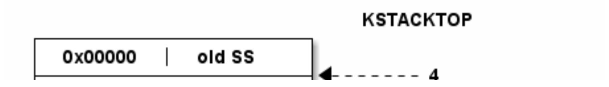
Interrupted in User Environment

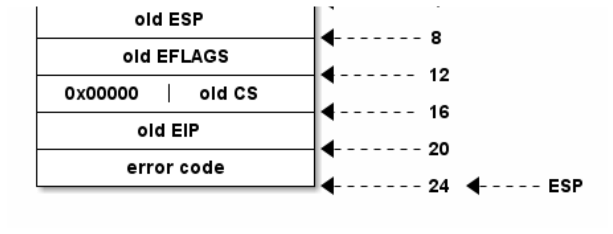
Let's say the processor is executing code in a user enviroment and encounters an interrupt/exception.

1. The processor switches to the stack defined by the `SS0` and `ESP0` fields of the TSS, which in JOS will hold the values GD_KD and KSTACKTOP, respectively.
2. The processor pushes the exception parameters on the kernel stack, starting at address KSTACKTOP:



3. For certain types of x86 exceptions, in addition to the "standard" five words above, the processor pushes onto the stack another word containing an error code. When the processor pushes an error code, the stack would look as follows at the beginning of the exception handler when coming in from user mode:





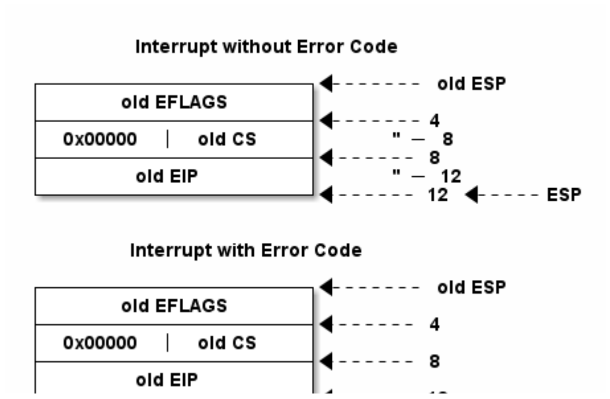
Description	Interrupt Number	Error Code
Divide error	0	No
Debug exception	1	No
Breakpoint	3	No
Overflow	4	No
Bounds check	5	No
Invalid opcode	6	No
Coprocessor not available	7	No
System error	8	Yes (always 0)
Coprocessor Segment Overrun	9	No
Invalid TSS	10	Yes
Segment not present	11	Yes
Stack exception	12	Yes
General protection fault	13	Yes
Page fault	14	Yes
Coprocessor error	16	No
Two-byte SW interrupt	0-255	No

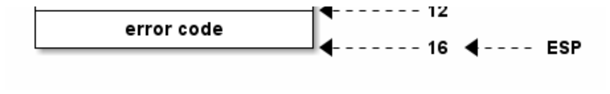
4. The processor reads IDT entry N (N is the interrupt vector) and sets CS:EIP to point to the handler function described by the entry.
5. The handler function takes control and handles the exception, for example by terminating the user environment.

Nested Exceptions and Interrupts

The processor can take exceptions and interrupts both from kernel and user mode. It is only when entering the kernel from user mode, however, that the x86 processor automatically switches stacks before pushing its old register state onto the stack and invoking the appropriate exception handler through the IDT. If the processor is already in kernel mode when the interrupt or exception occurs (the low 2 bits of the CS register are already zero), then the CPU just pushes more values on the same kernel stack. In this way, the kernel can gracefully handle nested exceptions caused by code within the kernel itself.

If the processor is already in kernel mode and takes a nested exception, since it does not need to switch stacks, it does not save the old SS or ESP registers. The kernel stack therefore looks like the following on entry to the exception handler:





There is one important caveat to the processor's nested exception capability. If the processor takes an exception while already in kernel mode, and cannot push its old state onto the kernel stack for any reason such as lack of stack space, then there is nothing the processor can do to recover, so it simply resets itself. Needless to say, the kernel should be designed so that this can't happen.

Exercise 4 & Challenge 1

> Q : Edit `trapentry.S` and `trap.c` and implement the features required. The macros `TRAPHANDLER` and `TRAPHANDLER_NOEC` in `trapentry.S` should help you, as well as the `T_*` defines in `inc/trap.h` . You will need to add an entry point in `trapentry.S` (using those macros) for each trap defined in `inc/trap.h` , and you'll have to provide `_alltraps` which the `TRAPHANDLER` macros refer to. You will also need to modify `trap_init()` to initialize the idt to point to each of these entry points defined in `trapentry.S` ; the `SETGATE` macro will be helpful here.
> Your _alltraps should:
>
> 1. push values to make the stack look like a struct Trapframe
> 2. load GD_KD into %ds and %es
> 3. pushl %esp to pass a pointer to the Trapframe as an argument to trap()
> 4. call trap (can trap ever return?)
>
> Consider using the `pushal` instruction; it fits nicely with the layout of the struct Trapframe.
> Test your trap handling code using some of the test programs in the user directory that cause exceptions before making any system calls, such as user/divzero. You should be able to get make grade to succeed on the divzero, softint, and badsegment tests at this point.
>
> **Challenge** : You probably have a lot of very similar code right now, between the lists of TRAPHANDLER in trapentry.S and their installations in trap.c. Clean this up. Change the macros in trapentry.S to automatically generate a table for trap.c to use. Note that you can switch between laying down code and data in the assembler by using the directives .text and .data.

In `trapentry.S` , We define entries for each interrupt/exception with `TRAPHANDLER` if there is an error code or `TRAPHANDLER_NOEC` if there isn't. Then we establish an array that stores entries address so that we can iterate over it in `trap_init()` to reduce tedious work. We set null entries that don't belong to a valid interrupt vector to zero. So some invalid IDT entries would have handle entries with zero address. The array will end at the last entry with a valid interrupt vector. So IDT entries without valid interrupt vectors will not be initalized. Hardwares assure that they won't generate an interrupt with an invalid interrupt vector. All invalid IDT entries have a zero-value DPL. Any attempt to generate an invalid exception from user programs will end up generating a "General Protection Exception".

Codes for setting up the IDT is shown below.

```diff
diff --git a/inc/trap.h b/inc/trap.h
index c3437af..7884372 100644
--- a/inc/trap.h
+++ b/inc/trap.h
@@ -12,13 +12,13 @@
 #define T_ILLOP      6         // illegal opcode
 #define T_DEVICE     7         // device not available
 #define T_DBLFLT     8         // double fault
-/* #define T_COPROC  9 */      // reserved (not generated by recent processors)
+#define T_COPROC        9              // reserved (not generated by recent processors)
 #define T_TSS       10         // invalid task switch segment
 #define T_SEGNP     11         // segment not present
 #define T_STACK     12         // stack exception
 #define T_GPFLT     13         // general protection fault
 #define T_PGFLT     14         // page fault
-/* #define T_RES    15 */      // reserved
+#define T_RES                  15              // reserved
 #define T_FPERR     16         // floating point error
 #define T_ALIGN     17         // aligment check
 #define T_MCHK      18         // machine check
diff --git a/kern/trap.c b/kern/trap.c
index e27b556..e94cb5b 100644
--- a/kern/trap.c
+++ b/kern/trap.c
@@ -65,6 +65,20 @@ trap_init(void)
        extern struct Segdesc gdt[];

        // LAB 3: Your code here.
+       extern uint32_t idt_entries[];
+       extern uint32_t idt_entries_end[];
+       int entry_num = idt_entries_end - idt_entries;
+       for (int i = 0; i < entry_num; ++i) {
+               switch (i) {
+                       case T_BRKPT :
+                       case T_SYSCALL :
+                               SETGATE(idt[i], 0, GD_KT, idt_entries[i], 3);
+                               break;
+                       default :
+                               SETGATE(idt[i], 0, GD_KT, idt_entries[i], 0);
+                               break;
+               }
+       }

        // Per-CPU setup
        trap_init_percpu();
diff --git a/kern/trapentry.S b/kern/trapentry.S
index 22fc640..ea24a22 100644
--- a/kern/trapentry.S
+++ b/kern/trapentry.S
@@ -46,10 +46,68 @@
 /*
  * Lab 3: Your code here for generating entry points for the different traps.
  */
+       TRAPHANDLER_NOEC(trap_divide_error, T_DIVIDE)                           # divide error
+       TRAPHANDLER_NOEC(trap_debug_exception, T_DEBUG)                                         # debug exception
+       TRAPHANDLER_NOEC(trap_nmi, T_NMI)                                                       # non-maskable interrupt
+    TRAPHANDLER_NOEC(trap_breakpoint, T_BRKPT)                                 # breakpoint
+       TRAPHANDLER_NOEC(trap_overflow, T_OFLOW)                                                # overflow
+       TRAPHANDLER_NOEC(trap_bounds_check, T_BOUND)                                            # bounds check
+       TRAPHANDLER_NOEC(trap_illegal_opcode, T_ILLOP)                                  # illegal opcode
+       TRAPHANDLER_NOEC(trap_device_not_available, T_DEVICE)           # device not available
+       TRAPHANDLER(trap_double_fault, T_DBLFLT)                                                # double fault
+       TRAPHANDLER_NOEC(trap_coprocessor_segment_overrun, T_COPROC)    # reserved (not generated by recent processors)
+       TRAPHANDLER(trap_invalid_TSS, T_TSS)                                                    # invalid task switch segment
+       TRAPHANDLER(trap_segment_not_present, T_SEGNP)                          # segment not present
+       TRAPHANDLER(trap_stack_exception, T_STACK)                                              # stack exception
+       TRAPHANDLER(trap_general_protection_fault, T_GPFLT)                             # general protection fault
+       TRAPHANDLER(trap_page_fault, T_PGFLT)                                           # page fault
+       TRAPHANDLER_NOEC(trap_reserved, T_RES)                                          # reserved
+       TRAPHANDLER_NOEC(trap_float_point_error, T_FPERR)                       # floating point error
+       TRAPHANDLER(trap_alignment_check, T_ALIGN)                                      # alignment check
+       TRAPHANDLER_NOEC(trap_machine_check, T_MCHK)                                    # machine check
+       TRAPHANDLER_NOEC(trap_SIMD_float_point_error, T_SIMDERR)                # SIMD floating point error
+       TRAPHANDLER_NOEC(trap_system_call, T_SYSCALL)                           # system call

-
+.data
```

```
+.globl idt_entries
+idt_entries:
+       .long trap_divide_error
+       .long trap_debug_exception
+       .long trap_nmi
+       .long trap_breakpoint
+       .long trap_overflow
+       .long trap_bounds_check
+       .long trap_illegal_opcode
+       .long trap_device_not_available
+       .long trap_double_fault
+       .long trap_coprocessor_segment_overrun
+       .long trap_invalid_TSS
+       .long trap_segment_not_present
+       .long trap_stack_exception
+       .long trap_general_protection_fault
+       .long trap_page_fault
+       .long trap_reserved
+       .long trap_float_point_error
+       .long trap_alignment_check
+       .long trap_machine_check
+       .long trap_SIMD_float_point_error
+       .fill 29, 4, 0                                  # 29 null entries
+       .long trap_system_call
+.global idt_entries_end
+idt_entries_end:

 /*
  * Lab 3: Your code here for _alltraps
  */
-
+.text
+_alltraps:
+       push %ds
+       push %es
+       pushal
+
+       movw $GD_KD, %ax
+       movw %ax, %ds
+       movw %ax, %es
+
+       pushl %esp
+       call trap
\ No newline at end of file
```

Answer to Q1 & Q2

> 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?)

We can’t push ***Interrupt Vector*** on to the stack for `trap_dispatch()` to choose the right handler accordingly if we all exceptions/interrupts were delivered to the same handler.

> 2. Did you have to do anything to make the user/softint program behave correctly? The grade script expects it to produce a general protection fault (trap 13), but softint’s code says int $14. Why should this produce interrupt vector 13? What happens if the kernel actually allows softint’s int $14 instruction to invoke the kernel’s page fault handler (which is interrupt vector 14)?

We don’t need to do anything to make user/softint program behave correctly. DPLs of all IDT entries except syscall and break point are zero. Therefore, any attempt from user/softint program to generate an interrupt vector that are not allowed will end up in “General Protection Fault”.

Page Faults, Breakpoints Exceptions, and System Calls

Exercise 5 & 6 & 7

> Q : Modify `trap_dispatch()` to dispatch page fault exceptions to `page_fault_handler()`. You should now be able to get make grade to succeed on the faultread, faultreadkernel, faultwrite, and faultwritekernel tests. If any of them don’t work, figure out why and fix them. Remember that you can boot JOS into a particular user program using make run-x or make run-x-nox. For instance, make run-hello-nox runs the hello user program.

> Q : Modify trap_dispatch() to make breakpoint exceptions invoke the kernel monitor. You should now be able to get make grade to succeed on the breakpoint test.

> Q : Add a handler in the kernel for interrupt vector T_SYSCALL. You will have to edit kern/trapentry.S and kern/trap.c's trap_init(). You also need to change trap_dispatch() to handle the system call interrupt by calling syscall() (defined in kern/syscall.c) with the appropriate arguments, and then arranging for the return value to be passed back to the user process in %eax. Finally, you need to implement syscall() in kern/syscall.c. Make sure syscall() returns -E_INVAL if the system call number is invalid. You should read and understand lib/syscall.c (especially the inline assembly routine) in order to confirm your understanding of the system call interface. Handle all the system calls listed in inc/syscall.h by invoking the corresponding kernel function for each call.
>
> Run the user/hello program under your kernel (make run-hello). It should print "hello, world" on the console and then cause a page fault in user mode. If this does not happen, it probably means your system call handler isn't quite right. You should also now be able to get make grade to succeed on the testbss test.

It's easy to finish exercise 5 & 6. All we need to do is to use a `switch` structure to invoke different interrupt handler function according to the interrupt vector passed to `trap_dispatch()` . As you can see in ***Exercise 4***, we already add system call entry address in `idt_entries` array in `trapentry.S` . So we don't need to set up modify codes for IDT trap initialization any more (I miscalculate the place of syscall entry in `idt_entries` by one. I correct it in this exercise).

When it comes to syscall, we need to know that user program passes its parameters using register when invoking a system call. So in `trap_dispatch()` , parameters for system call are stored in the `TrapFrame` passed to `trap_dispatch()` . The calling convetion is that system call number and the return value stored in `%eax` . You can read `lib/syscall.c/syscall()` to understand how JOS arrange parameters among general registers.

In `kern/syscall.c/syscall()` , we dispatch syscall request according to syscall number in the same way that we did in `trap_dispatch()` .

Codes for these exercise are shown below.

```
diff --git a/kern/syscall.c b/kern/syscall.c
index 414d489..4e7b70f 100644
--- a/kern/syscall.c
+++ b/kern/syscall.c
@@ -70,11 +70,15 @@ syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4,
        // Return any appropriate return value.
        // LAB 3: Your code here.

-       panic("syscall not implemented");
+       // panic("syscall not implemented");

        switch (syscallno) {
-       default:
-               return -E_INVAL;
+               case SYS_cputs: sys_cputs((const char *)a1, a2); return 0;
+               case SYS_cgetc: return (int32_t)sys_cgetc();
+               case SYS_getenvid: return (int32_t)sys_getenvid();
+               case SYS_env_destroy: return (int32_t)sys_env_destroy((envid_t)a1);
+               default:
+                       return -E_INVAL;
        }
 }

diff --git a/kern/trap.c b/kern/trap.c
index e94cb5b..6e6b1b7 100644
--- a/kern/trap.c
+++ b/kern/trap.c
@@ -79,7 +79,7 @@ trap_init(void)
                                break;
                }
        }
-
+       cprintf("entry number: %d\n", entry_num);
        // Per-CPU setup
        trap_init_percpu();
 }
@@ -159,6 +159,20 @@ trap_dispatch(struct Trapframe *tf)
        // Handle processor exceptions.
        // LAB 3: Your code here.

+       switch(tf->tf_trapno) {
+               case T_BRKPT: monitor(tf);
+                                         return;
+               case T_PGFLT: page_fault_handler(tf);
+                                         return;
+               case T_SYSCALL: 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);
+                                         return;
+       }
+
        // Unexpected trap: The user process or the kernel has a bug.
        print_trapframe(tf);
        if (tf->tf_cs == GD_KT)
diff --git a/kern/trapentry.S b/kern/trapentry.S
index ea24a22..ca457ee 100644
--- a/kern/trapentry.S
+++ b/kern/trapentry.S
@@ -91,7 +91,7 @@ idt_entries:
        .long trap_alignment_check
        .long trap_machine_check
        .long trap_SIMD_float_point_error
-       .fill 29, 4, 0                                    # 29 null entries
+       .fill 28, 4, 0                                    # 29 null entries
        .long trap_system_call
 .global idt_entries_end
 idt_entries_end:
```

Answer to Q3 & Q4

> 3. The break point test case will either generate a break point exception or a general protection fault depending on how you initialized the break point entry in the IDT (i.e., your call to SETGATE from trap_init). Why? How do you need to set it up in order

> to get the breakpoint exception to work as specified above and what incorrect setup would cause it to trigger a general protection fault?

We need to set the DPL = 3 for break point trap handler to give user program the privilege to invoke it. Otherwise a program generating break point exception will trigger a general protection fault.

> 4. What do you think is the point of these mechanisms, particularly in light of what the user/softint test program does?

These mechanisms prevents user program from invoking arbitrary interrupt handlers that gives user program a chance to interfere with kernel or other programs unconsciously or deliberately.

Exercise 8

> Q : Add the required code to the user library, then boot your kernel. You should see user/hello print "hello, world" and then print "i am environment 00001000". user/hello then attempts to "exit" by calling sys_env_destroy() (see lib/libmain.c and lib/exit.c). Since the kernel currently only supports one user environment, it should report that it has destroyed the only environment and then drop into the kernel monitor. You should be able to get make grade to succeed on the hello test.

The user program has access to `envs` . It have access to its `envid` by using `sys_getenvid()` , too. So it's easy for a user program to get the pointer to it's own enviroment. By the way, I fixed a bug left in last exercise.

Codes for this exercise are shown below.

```
diff --git a/kern/trap.c b/kern/trap.c
index 6e6b1b7..8e99f14 100644
--- a/kern/trap.c
+++ b/kern/trap.c
@@ -164,12 +164,13 @@ trap_dispatch(struct Trapframe *tf)
                                           return;
                case T_PGFLT: page_fault_handler(tf);
                                           return;
-               case T_SYSCALL: 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);
+               case T_SYSCALL: tf->tf_regs.reg_eax = 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);
                                           return;
        }

diff --git a/lib/libmain.c b/lib/libmain.c
index 8a14b29..e42ca80 100644
--- a/lib/libmain.c
+++ b/lib/libmain.c
@@ -2,6 +2,7 @@
 // entry.S already took care of defining envs, pages, uvpd, and uvpt.

 #include <inc/lib.h>
+#include <inc/env.h>

 extern void umain(int argc, char **argv);

@@ -13,7 +14,7 @@ libmain(int argc, char **argv)
 {
        // set thisenv to point at our Env structure in envs[].
        // LAB 3: Your code here.
-       thisenv = 0;
+       thisenv = &envs[ENVX(sys_getenvid())];

        // save the name of the program so that panic() can use it
        if (argc > 0)
```

Page faults and memory protection

OS usually rely on hardware support to implement memory protection. When a program tries to access an invalid address or one for which it has no permissions, the processor stops the program at the instruction causing the fault and then traps into the kernel with information about the attempted operation. If the fault is fixable, the kernel fixes it and let the program continue running. If the fault is not fixable, then the program is destroyed, since it will never get past the instruction causing the fault.

System calls present an interesting problem for memory protection. Most system call interfaces let user programs pass pointers to the kernel. These pointers point at user buffers to be read or written. The kernel then dereferences these pointers while carrying out the system call. There are two problems with this:

1. A page fault in the kernel is potentially a lot more serious than a page fault in a user program. If the kernel page-faults while manipulating its own data structures, that's a kernel bug, and the fault handler should panic the kernel (and hence the whole system). But when the kernel is dereferencing pointers given to it by the user program, it needs a way to remember that any page faults these dereferences cause are actually on behalf of the user program.
2. The kernel typically has more memory permissions than the user program. The user program might pass a pointer to a system call that points to memory that the kernel can read or write but that the program cannot. The kernel must be careful not to be tricked into dereferencing such a pointer, since that might reveal private information or destroy the integrity of the kernel.

For both reasons the kernel must be extremely careful when handling pointers presented by user programs. JOS solves these problems by checking that the address passed by a user program is the user part of the address space and that the page table would allow the memory operation. In this way, the kernel will never suffer a page fault due to dereferencing a user-supplied pointer. If the kernel does page fault, it should panic and terminate.

Exercise 9 & 10

> Q : Change `kern/trap.c` to panic if a page fault happens in kernel mode. Change debuginfo_eip in `kern/kdebug.c` to call `user_mem_check` on `usd` , `stabs` , and `stabstr` .

> Q : Boot your kernel, running user/evilhello. The environment should be destroyed, and the kernel should not panic. You should see:
>
> ```
> [00000000] new env 00001000
> ...
> [00001000] user_mem_check assertion failure for va f010000c
> [00001000] free env 00001000
> ```

Using `pgdir_walk` , it's quite easy to check the memory address. There is a corner case that needs our attention. It's much more effcient to round addresses passed to `user_mem_check` and check the beginning address of each page insteat of checking every address. If the first page we check failed, we should recored `va` as `user_mem_check_addr` instead of the beginning address of the first page because what really caused the fault was the `va` passed to `user_mem_check` from the perspective of a user program.

Codes for these two exercises are shown below.

```
diff --git a/kern/kdebug.c b/kern/kdebug.c
index 38cfc6b..32a2ff1 100644
--- a/kern/kdebug.c
+++ b/kern/kdebug.c
@@ -142,6 +142,8 @@ debuginfo_eip(uintptr_t addr, struct Eipdebuginfo *info)
                // Make sure this memory is valid.
                // Return -1 if it is not.  Hint: Call user_mem_check.
                // LAB 3: Your code here.
+               if (user_mem_check(curenv, (const void *)usd, sizeof(struct UserStabData), 0) < 0)
+                       return -1;

                stabs = usd->stabs;
                stab_end = usd->stab_end;
@@ -150,6 +152,9 @@ debuginfo_eip(uintptr_t addr, struct Eipdebuginfo *info)

                // Make sure the STABS and string table memory is valid.
                // LAB 3: Your code here.
+               if (user_mem_check(curenv, (const void *)stabs, (stab_end - stabs)*sizeof(struct Stab), 0) < 0 ||
+                       user_mem_check(curenv, (const void *)stabstr, (stabstr_end - stabstr)*sizeof(char), 0) < 0)
+                       return -1;
        }

        // String table validity checks
diff --git a/kern/pmap.c b/kern/pmap.c
index 2a37451..78f13a7 100644
--- a/kern/pmap.c
+++ b/kern/pmap.c
@@ -682,6 +682,25 @@ user_mem_check(struct Env *env, const void *va, size_t len, int perm)
 {
        // LAB 3: Your code here.

+       uintptr_t va_start = (uintptr_t) ROUNDDOWN(va, PGSIZE);
+       uintptr_t va_end = (uintptr_t) ROUNDUP(va + len, PGSIZE);
+
+       for (; va_start < va_end; va_start += PGSIZE) {
+               if (va_start < ULIM) {
+                       pte_t *pt_entry = pgdir_walk(env->env_pgdir, (const void *)va_start, false);
+                       if (pt_entry != NULL &&
+                               ((uint32_t)*pt_entry & perm) == perm)
+                               continue;
+               }
+
+               if (va_start < (uintptr_t) va)
+                       user_mem_check_addr = (uintptr_t )va;
+               else
+                       user_mem_check_addr = va_start;
+
+               return -E_FAULT;
+       }
+
        return 0;
 }

diff --git a/kern/syscall.c b/kern/syscall.c
index 4e7b70f..c39afc3 100644
--- a/kern/syscall.c
+++ b/kern/syscall.c
@@ -21,6 +21,7 @@ sys_cputs(const char *s, size_t len)
        // Destroy the environment if not.

        // LAB 3: Your code here.
+       user_mem_assert(curenv, (const void *)s, len, 0);

        // Print the string supplied by the user.
        cprintf("%.*s", len, s);
diff --git a/kern/trap.c b/kern/trap.c
index 8e99f14..ebf2b3c 100644
--- a/kern/trap.c
+++ b/kern/trap.c
@@ -234,6 +234,11 @@ page_fault_handler(struct Trapframe *tf)
        // Handle kernel-mode page faults.

        // LAB 3: Your code here.
+       // the low 2 bits in code stack register (CS) gives the
+       // privilege level of the trapframe. The low 2 bits of kernel
+       // mode is 0 while the low 2 bits of user mode is 3
+       if ((tf->tf_cs & 0x3) == 0)
+               panic("Page fault in kernel.");

        // We've already handled kernel-mode exceptions, so if we get here,
        // the page fault happened in user mode.
```

Challenges

Challenge 1

Challenge 1 has already been implemented in section Exericse 4 & Challenge 1

Challenge 2

> Q : 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.

Since `curenv` doesn’t change when a user program is trapped into kernel monitor, to continue to execute it, we only need to add a command to kernel monitor which will invoke `env_run(curenv)` . To single step the user program, we need to turn on the TF flag bit (bit 8) in EFLAGS which tells the processor enter single stepping mode, namely, execute one instruction and generate a debug exception. We should recall that we need to tell the processor to single step the user program not the kernel monitor so we need to modify `tf->tf_eflags` instead of the real EFLAGS register. In addition, in single step mode, the processor generate debug exception instead of breakpoint exception. Thus we need to modify `trap_dispatch()` to dispacth both break point exception and debug exception to kernel monitor.

To demonstrate our implementation, we modify `breakpoint.c` , add a `nop` instruction and another `int3` to it, then run `make run-qemu-nox` . The log and explanation are listed below.

```
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
entry number: 49
[00000000] new env 00001000
Incoming TRAP frame at 0xefffffbc
Incoming TRAP frame at 0xefffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01b4000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x00000003 Breakpoint                    # break point exception generated by the first "int3"
  err  0x00000000
  eip  0x00800037
  cs   0x----001b
  flag 0x00000082
  esp  0xeebfdfd0
  ss   0x----0023
K> stepi                                        # excute next instruction, namely "nop"
Incoming TRAP frame at 0xefffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01b4000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x00000001 Debug                         # debug exception generated by the processor after executing "nop"
  err  0x00000000
  eip  0x00800038
  cs   0x----001b
  flag 0x00000182
  esp  0xeebfdfd0
  ss   0x----0023
K> stepi
Incoming TRAP frame at 0xefffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01b4000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfd0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x00000003 Breakpoint                    # break point exception generated by the second "int3"
  err  0x00000000
  eip  0x00800039
  cs   0x----001b
  flag 0x00000182
  esp  0xeebfdfd0
  ss   0x----0023
K> stepi
```

```
Incoming TRAP frame at 0xefffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01b4000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdff0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x00000001 Debug                        # debug exception generated by the processor after executing
  err  0x00000000                              # the instruction after the second "int3". We know this exception
  eip  0x0080003a                              # wasn't generated by executing "int3" in single step mode
  cs   0x----001b                              # because it's "eip" is different from last trapframe.
  flag 0x00000182
  esp  0xeebfdfd4
  ss   0x----0023
K> stepi
Incoming TRAP frame at 0xefffffbc
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf01b4000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdff0
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0xeec00000
  es   0x----0023
  ds   0x----0023
  trap 0x00000001 Debug
  err  0x00000000
  eip  0x00800075
  cs   0x----001b
  flag 0x00000182
  esp  0xeebfdfd8
  ss   0x----0023
K> continue                                    # continue to run the program. As we can see, the program
Incoming TRAP frame at 0xefffffbc              # ended ans was destroyed by the kernel normally after "continue".
[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.
K> QEMU: Terminated
```

Codes for this challenge are shown below.

```diff
diff --git a/kern/monitor.c b/kern/monitor.c
index cd43b29..513e2e7 100644
--- a/kern/monitor.c
+++ b/kern/monitor.c
@@ -11,6 +11,7 @@
 #include <kern/monitor.h>
 #include <kern/kdebug.h>
 #include <kern/trap.h>
+#include <kern/env.h>

 #define CMDBUF_SIZE    80      // enough for one VGA text line

@@ -26,6 +27,8 @@ static struct Command commands[] = {
        { "help", CMD_HELP_HELP_STR, mon_help },
        { "kerninfo", CMD_KERNINFO_HELP_STR, mon_kerninfo },
        { "backtrace", CMD_BACKTRACE_HELP_STR, mon_backtrace},
+       { "continue", CMD_CONTINUE_HELP_STR, mon_continue},
+       { "stepi", CMD_STEPI_HELP_STR, mon_stepi},
 };


@@ -130,7 +133,53 @@ mon_backtrace(int argc, char **argv, struct Trapframe *tf)
        return 0;
 }

+int
+mon_continue(int argc, char **argv, struct Trapframe *tf)
+{
+       if (!curenv) {
+               cprintf("No runnable enviroment available.\n");
+               return 0;
+       }
+
+       if (!tf) {
+               cprintf("No trapframe available.\n");
+               return 0;
+       }
+
+       if (tf->tf_trapno != T_BRKPT && tf->tf_trapno != T_DEBUG) {
+               cprintf("Kernel monitor is not invoked by a break point or debug interrupt.\n");
+               return 0;
+       }
+
+       tf->tf_eflags &= ~(FL_TF);
+
+       env_run(curenv);
+       return 0;
+}

+int
+mon_stepi(int argc, char **argv, struct Trapframe *tf)
+{
+       if (!curenv) {
+               cprintf("No runnable enviroment available.\n");
+               return 0;
+       }
+
+       if (!tf) {
+               cprintf("No trapframe available.\n");
+               return 0;
+       }
+
+       if (tf->tf_trapno != T_BRKPT && tf->tf_trapno != T_DEBUG) {
+               cprintf("Kernel monitor is not invoked by a break point or debug interrupt.\n");
+               return 0;
+       }
+
+       tf->tf_eflags |= FL_TF;
+
+       env_run(curenv);
+       return 0;
+}

 /***** Kernel monitor command interpreter *****/

diff --git a/kern/monitor.h b/kern/monitor.h
index 9aea717..1b26be5 100644
```

```
--- a/kern/monitor.h
+++ b/kern/monitor.h
@@ -15,6 +15,8 @@ void monitor(struct Trapframe *tf);
 int mon_help(int argc, char **argv, struct Trapframe *tf);
 int mon_kerninfo(int argc, char **argv, struct Trapframe *tf);
 int mon_backtrace(int argc, char **argv, struct Trapframe *tf);
+int mon_continue(int argc, char **argv, struct Trapframe *tf);
+int mon_stepi(int argc, char **argv, struct Trapframe *tf);

 #define CMD_HELP_HELP_STR       "\
 -SYNOPSIS:\n\
@@ -23,8 +25,17 @@ int mon_backtrace(int argc, char **argv, struct Trapframe *tf);
     list: display all help information of all commands.\n\
     command name: display help information of given name\n"

-#define CMD_KERNINFO_HELP_STR   "Display information about the kernel\n"
+#define CMD_KERNINFO_HELP_STR   "    Display information about the kernel\n"

-#define CMD_BACKTRACE_HELP_STR  "Display the current call stack\n"
+#define CMD_BACKTRACE_HELP_STR  "    Display the current call stack\n"
+
+#define CMD_CONTINUE_HELP_STR   "\
+    Continue executing the program that caused a break point or debug\n\
+    interrupt.\n"
+
+#define CMD_STEPI_HELP_STR      "\
+    Execute next intruction of the program that caused a break point or\n\
+    debug interrupt. Then a debug interrupt is raised and execution is\n\
+    trapped into kernel monitor again.\n"

 #endif // !JOS_KERN_MONITOR_H
diff --git a/kern/trap.c b/kern/trap.c
index ebf2b3c..bd63da7 100644
--- a/kern/trap.c
+++ b/kern/trap.c
@@ -160,6 +160,7 @@ trap_dispatch(struct Trapframe *tf)
        // LAB 3: Your code here.

        switch(tf->tf_trapno) {
+               case T_DEBUG:
                case T_BRKPT: monitor(tf);
                                          return;
                case T_PGFLT: page_fault_handler(tf);
diff --git a/user/breakpoint.c b/user/breakpoint.c
index 47e4cb2..d23469d 100644
--- a/user/breakpoint.c
+++ b/user/breakpoint.c
@@ -6,5 +6,7 @@ void
 umain(int argc, char **argv)
 {
        asm volatile("int $3");
+       asm volatile("nop");
+       asm volatile("int $3");
 }
```

Challenge 3

> Q : Implement system calls using the sysenter and sysexit instructions instead of using int 0x30 and iret.

Ref: Intel$^{®}$ 64 and IA-32 Architectures Software Developer's Manual

`sysenter` give user program a chance to jump directly to system call handler instead of taking a detour to interrupt mechanism for system call, which results in a better performance.

When `sysenter` is executed, the processor:

1. Loads the segment selector from the ***IA32_SYSENTER_CS*** into the ***CS*** register.
2. Loads the instruction pointer from the ***IA32_SYSENTER_EIP*** into the EIP register.
3. Adds 8 to the value in ***IA32_SYSENTER_CS*** and loads it into the ***SS*** register.
4. Loads the stack pointer from the ***IA32_SYSENTER_ESP*** into the ***ESP*** register.
5. Switches to privilege level 0.
6. Clears the ***VM*** flag in the ***EFLAGS*** register, if the flag is set.

7. Begins executing the selected system procedure.

The processor does not save a return ***IP*** or other state information for the calling procedure.

When `sysexit` is executed, the processor:

1. Adds 16 to the value in ***IA32_SYSENTER_CS*** and loads the sum into the ***CS*** selector register.
2. Loads the instruction pointer from the ***EDX*** register into ***EIP*** register.
3. Adds 24 to the value in ***IA32_SYSENTER_CS*** and loads the sum into the ***SS*** selector register.
4. Loads the stack pointer from the ***ECX*** register into the ***ESP*** register.
5. Switches to privilege level 3.
6. Begins executing the user code at the ***EIP*** address.

From the above description, we can conclude that to use `sysenter/sysexit` mechanism, we need to do:

1. Put the code stack segment selector of system call handler in ***IA32_SYSENTER_CS***.
2. Put the address of the beginning of system call handler in its code segment in ***IA32_SYSENTER_EIP***.
3. Put the stack pointer used by system call handler in ***IA32_SYSENTER_ESP***.
4. The processor use ***IA32_SYSENTER_CS+8*** for system call handler stack segment selector on ***SYSENTER***, ***IA32_SYSENTER_CS+16*** for user code segment selector on ***SYSEXIT***, ***IA32_SYSENTER_CS+24*** for user stack segment selector on ***SYSEXIT***. So we need to design JOS to fulfill this requirement.
5. Put the address of the user instruction to be executed in ***EDX*** before executing ***SYSEXIT***.
6. Put the user stack pointer in ***ECX*** before executing ***SYSEXIT***.

In JOS, we have the following macros for code segment and stack segment of kernel and user. They meet the requirement so we don't need to modify them.

```
#define GD_KT     0x08     // kernel text
#define GD_KD     0x10     // kernel data
#define GD_UT     0x18     // user text
#define GD_UD     0x20     // user data
```

JOS needs to set up ***IA32_SYSENTER_CS***, ***IA32_SYSENTER_EIP***, ***IA32_SYSENTER_ESP*** for user. Since values in these reigster never change after initialization, we initialize them during kernel initialization using macros we defined in `inc/x86.h` . ***SYSENTER/SYSEXIT*** mechanism doesn't store anything for us. Thus we need to pass parameters for system call as well as the returning instuction address and user stack pointer for ***SYSEXIT*** using registers. We arrange registers in the way shown below while preserve the GCC calling convention.

register	usage
eax	syscall number
edx	argument 1
ecx	argument 2
ebx	argument 3
edi	argument 4
ebp	return esp
esp	trashed by sysenter

Because ***SYSENTER*** will change the value in ***ESP***, we can't store anything in it. This leaves us registers enough only for 4 arguments. To pass five arguments, we store a pointer to real argument 4 in ***EDI***. In GCC calling convention, argument 4 and 5 reside consecutively on the stack. Thus system call handler is able to get the real argument 4 by `*(edi)` and argument 5 by `*(edi + 4)` . In the first few instructions of system call handler, we push arguments onto stack then we can reuse the real system call handler function. After the system call handler function return, we set up ***EDX*** and ***ECX*** then execute ***SYSEXIT*** to return to the user program.

After everything is in place, we can pass all tests as we did by using interrupt for system call.

```
divzero: OK (2.2s)
softint: OK (0.9s)
badsegment: OK (1.0s)
Part A score: 30/30

faultread: OK (0.9s)
faultreadkernel: OK (1.6s)
faultwrite: OK (2.3s)
faultwritekernel: OK (2.0s)
breakpoint: OK (1.0s)
testbss: OK (1.9s)
hello: OK (1.8s)
buggyhello: OK (2.1s)
buggyhello2: OK (1.0s)
evilhello: OK (2.0s)
Part B score: 50/50

Score: 80/80
```

Codes for this challenge are shown below.

```
diff --git a/inc/x86.h b/inc/x86.h
index cc15ff4..430341c 100644
--- a/inc/x86.h
+++ b/inc/x86.h
@@ -261,4 +261,25 @@ xchg(volatile uint32_t *addr, uint32_t newval)
        return result;
 }

+/*
+ * Access to machine-specific registers (available on 586 and better only)
+ * Note: the rd* operations modify the parameters directly (without using
+ * pointer indirection), this allows gcc to optimize better
+ */
+
+#define IA32_SYSENTER_CS       0x174
+#define IA32_SYSENTER_ESP      0x175
+#define IA32_SYSENTER_EIP      0x176
+
+#define rdmsr(msr,val1,val2) \
+       __asm__ __volatile__("rdmsr" \
+       : "=a" (val1), "=d" (val2) \
+       : "c" (msr))
+
+#define wrmsr(msr,val1,val2) \
+       __asm__ __volatile__("wrmsr" \
+       : /* no outputs */ \
+       : "c" (msr), "a" (val1), "d" (val2))
+
+
 #endif /* !JOS_INC_X86_H */
diff --git a/kern/init.c b/kern/init.c
index 87ed126..731b635 100644
--- a/kern/init.c
+++ b/kern/init.c
@@ -3,6 +3,7 @@
 #include <inc/stdio.h>
 #include <inc/string.h>
 #include <inc/assert.h>
+#include <inc/x86.h>

 #include <kern/monitor.h>
 #include <kern/console.h>
@@ -35,6 +36,13 @@ i386_init(void)
        env_init();
        trap_init();

+       // initialize Machine Specific Registers to support
+       // system call using sysenter/sysexit
+       extern char _sysenter[];
+       wrmsr(IA32_SYSENTER_CS, GD_KT, 0);
+       wrmsr(IA32_SYSENTER_ESP, KSTACKTOP, 0);
+       wrmsr(IA32_SYSENTER_EIP, _sysenter, 0);
+
 #if defined(TEST)
        // Don't touch -- used by grading script!
        ENV_CREATE(TEST, ENV_TYPE_USER);
diff --git a/kern/trapentry.S b/kern/trapentry.S
index ca457ee..fcfe3a4 100644
--- a/kern/trapentry.S
+++ b/kern/trapentry.S
@@ -110,4 +110,21 @@ _alltraps:
        movw %ax, %es

        pushl %esp
-       call trap
\ No newline at end of file
+       call trap
+
+/*
+ * entry code for system call using sysenter/sysexit
+ */
+.global _sysenter
+_sysenter:
+       pushl 4(%edi)
+       pushl (%edi)
+       pushl %ebx
+       pushl %ecx
+       pushl %edx
+       pushl %eax
+       call syscall
+
+       movl %ebp, %ecx
+       movl %esi, %edx
+       sysexit
diff --git a/lib/syscall.c b/lib/syscall.c
index 8d28dda..9eab046 100644
--- a/lib/syscall.c
+++ b/lib/syscall.c
@@ -3,33 +3,34 @@
 #include <inc/syscall.h>
 #include <inc/lib.h>

-static inline int32_t
+/*
+ * new syscall using sysenter/sysexit
+ *
+ * We use a label (_sysexit) in inline asm that C doesn't know.
+ * So we can't declare syscall() with 'inline'. Otherwise C
+ * preprocessor will copy those asm codes to multiple places then
+ * the compiler will complain that it found the same label
+ * declaration in multiple places.
+ */
+static int32_t
 syscall(int num, int check, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
 {
        int32_t ret;

-       // Generic system call: pass system call number in AX,
-       // up to five parameters in DX, CX, BX, DI, SI.
-       // Interrupt kernel with T_SYSCALL.
-       //
-       // The "volatile" tells the assembler not to optimize
-       // this instruction away just because we don't use the
-       // return value.
-       //
-       // The last clause tells the assembler that this can
-       // potentially change the condition codes and arbitrary
-       // memory locations.
-
-       asm volatile("int %1\n"
-                    : "=a" (ret)
-                    : "i" (T_SYSCALL),
-                      "a" (num),
-                      "d" (a1),
-                      "c" (a2),
-                      "b" (a3),
-                      "D" (a4),
-                      "S" (a5)
-                    : "cc", "memory");
+       asm volatile(
+                       "pushl %%ebp\n\t"
+                       "movl $_sysexit, %%esi\n\t"
+                       "movl %%esp, %%ebp\n\t"
+                       "sysenter\n\t"
+                       "_sysexit:\n\t"
+                       "popl %%ebp"
+                       : "=a" (ret)
+                       : "a" (num),
+                         "d" (a1),
+                         "c" (a2),
+                         "b" (a3),
+                         "D" (&a4)              /* store pointer to a4 in edi to pass a4 and a5 */
+                       : "cc", "memory", "esi");

        if(check && ret > 0)
                panic("syscall %d returned %d (> 0)", num, ret);
```

Grade

Finally, we got our grade.

```
divzero: OK (6.7s)
softint: OK (1.2s)
badsegment: OK (1.1s)
Part A score: 30/30

faultread: OK (2.0s)
faultreadkernel: OK (2.1s)
faultwrite: OK (1.9s)
faultwritekernel: OK (2.0s)
breakpoint: OK (1.1s)
testbss: OK (1.6s)
hello: OK (1.2s)
buggyhello: OK (1.8s)
buggyhello2: OK (2.1s)
evilhello: OK (1.2s)
Part B score: 50/50

Score: 80/80
```

End of Lab 3 Report

Email: caoshuyang@pku.edu.cn GitHub: JOS-Lab