MIT 6.828: Operating System Engineering

Lab Report

Lab 3: User Environments

CSNLP
csnlp16@126.com
https://csnlp.github.io/

Contents

1 Introduction

1.1 Useful Flags of PTE

Each PTE contains flag bits that tell the paging hardware how the associated virtual address is allowed to be used.

1. **PTE_P**: indicates whether the PTE is present.
2. **PTE_W**: controls whether instructions are allowed to issue writes to the page.
3. **PTE_U**: controls whether user programs are allowed to use page.

Hardware wants attention, so software must set aside current work and respond. Why does hardware want attention now?

- MMU cannot translate address.
- User program divides by zero.
- User program wants to execute kernel code (INT).
- Kernel CPU-to-CPU communication, e.g., to flush TLB.

These traps, generally speaking, fall into 3 classes:

1. Exceptions (page fault, divide by zero).
2. System calls (INT, intended exception).
3. Interrupts (devices want attention).

How does trap() know which device interrupted?

1. Kernel tells LAPIC/IOAPIC what vector number to use.
2. IDT associates an instruction address with each vector number.
3. Execute the associated instruction.
4. Each vector jumps to alltraps.

2 XV6 Chapter 3: Traps, interrupts, and drivers

3 Part A: User Environments and Exception Handling

The kernel maintains three main global variables pertaining to environments:

```
struct Env *envs = NULL;            // All environments
struct Env *curenv = NULL;          // The current env
static struct Env *env_free_list;   // Free environment list
                                    // (linked by Env->env_link)
```

Listing 1: Three global variables pertaining environments defined in kern/env.c

For this three environment variables:

1. **envs** is a pointer pointing to an array of **Env** structures representing all the environments in the system.

2. **curenv** is used to keep track of the currently executing environment.

3. **env_free_list** is a array which keeps all of the inactive **Env** structures.

3.1 Environment State

The **Env** structure is defined in **inc/env.h** as follows:

```
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
};
```

1. **env_tf**: This structure, defined in **inc/trap.h**, holds the saved register values for the environment while that environment is not running: i.e., when the kernel or a different environment is running. The kernel saves these when switching from user to kernel mode, so that the environment can later be resumed where it left off.

2. **env_link**: This is a link to the next **Env** on the **env_free_list**. **env_free_list** points to the first free environment on the list.

3. **env_id**: The kernel stores here a value that uniquely identifiers the environment currently using this Env structure (i.e., using this particular slot in the envs array). After a user environment terminates, the kernel may re-allocate the same Env structure to a different environment - but the new environment will have a different env_id from the old one even though the new environment is re-using the same slot in the envs array.

4. **env_parent_id**: The kernel stores here the env_id of the environment that created this environment. In this way the environments can form a family tree, which will be useful for making security decisions about which environments are allowed to do what to whom.

5. **env_type**: This is used to distinguish special environments. For most environments, it will be ENV_TYPE_USER. We'll introduce a few more types for special system service environments in later labs.

6. **env_status**: the status of this environment.

7. **env_pgdir**: holds the kernel virtual address of this environment's page directory.

3.2 Allocating the Environments Array

Modify **mem_init()** further to allocate a similar array of **Env** structures, called **envs**.

3.2.1 Exercise 1

*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.*

Firstly, let's check the hints in **mem_init()**.

1. Hint 1:

```
        //////////////////////////////////////////////////////////////////////
        // Make 'envs' point to an array of size 'NENV' of 'struct Env'.
        // LAB 3: Your code here.
```

2. Hint 2:

```
        //////////////////////////////////////////////////////////////////////
        // Map the 'envs' array read-only by the user at linear address UENVS
        // (ie. perm = PTE_U | PTE_P).
        // Permissions:
        //    - the new image at UENVS  -- kernel R, user R
        //    - envs itself -- kernel RW, user NONE
        // LAB 3: Your code here.
```

For the above hints, we have our solutions for exercise 1.

```
        //////////////////////////////////////////////////////////////////////
        // Make 'envs' point to an array of size 'NENV' of 'struct Env'.
        // LAB 3: Your code here.
        envs = (struct Env*) boot_alloc(NENV*sizeof(struct Env));
        memset(envs, 0, NENV*sizeof(struct Env));
```

```
eee
```

3.3 Creating and Running Environments

3.3.1 Exercise 2

In the file ***env.c****, finish coding the following functions:*

1. ***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).*

2. ***env_setup_vm()****: Allocate a page directory for a new environment and initialize the kernel portion of the new environment's address space.*

3. ***region_alloc()****: Allocates and maps physical memory for an environment*

4. ***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.*

5. ***env_create()****: Allocate an environment with* ***env_alloc*** *and call* ***load_icode*** *to load an ELF binary into it.*

6. ***env_run()****: Start a given environment running in user mode.*

As you write these functions, you might find the new cprintf verb %e useful – it prints a description corresponding to an error code. For example, ***r = -E_NO_MEM; panic("env_alloc: %e", r);*** *will panic with the message "env_alloc: out of memory".*

3.3.2 The Implementation of env_init() Function

Let's firstly see the hint:

```
// Mark all environments in 'envs' as free, set their env_ids to 0,
// and insert them into the env_free_list.
// Make sure the environments are in the free list in the same order
// they are in the envs array (i.e., so that the first call to
// env_alloc() returns envs[0]).
//
void
env_init(void)
{
	// Set up envs array
	// LAB 3: Your code here.

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

Next, see my implementation:

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

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

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

3.3.3 The Implementation of env_setup_vm() Function

This is for allocating a page directory for a new environment and only initialize the **kernal portion** of the new environment address space. Firstly, see the original code and its hints:

```
//
// Initialize the kernel virtual memory layout for environment e.
// Allocate a page directory, set e->env_pgdir accordingly,
// and initialize the kernel portion of the new environment's address space.
// Do NOT (yet) map anything into the user portion
// of the environment's virtual address space.
//
// Returns 0 on success, < 0 on error.  Errors include:
//      -E_NO_MEM if page directory or table could not be allocated.
//
static int
env_setup_vm(struct Env *e)
{
        int i;
        struct PageInfo *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.

        // 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;
}
```

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

        // Allocate a page for the page directory
        if (!(p = page_alloc(ALLOC_ZERO)))
                return -E_NO_MEM;
        // LAB 3: Your code here.
        // Step 1: set e->env_pgdir
        e->env_pgdir = (pde_t *)page2kva(p);
        p->pp_ref++;

        // Step 2 and Step 3 is to initialize the page directory of this envir.
        // Step 2: map the directory below UTOP
        //          the initial VA below UTOP is empty.
        for(i=0; i<PDX(UTOP); i++) {
                e->env_pgdir[i] = 0;
        }
        // Step 3: map the directory above UTOP
        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;
}
```

3.3.4 The Implementation of region_alloc() Function

region_alloc() function is used to allocate and map physical memory to environment.

```
//
// 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!)
```

```c
	struct PageInfo *p = NULL;
	void *i;
	int r;
	void *start = (void *)ROUNDDOWN(va, PGSIZE);
	void *end = (void *)ROUNDUP(va+len, PGSIZE);
	for(i=start; i<end; i+=PGSIZE) {
		// page allocation
		p = page_alloc(0);
		if(p == NULL) {
			panic("region alloc: page alloc failed!");
		}
		// page insert
		// page insert
		r = page_insert(e->env_pgdir, p, i, PTE_W | PTE_U);
		if(r != 0)
			panic("region alloc: pgdir modification failed");
	}
}
```

3.3.5 The Implementation of load_icode() Function

The initial hints:

```c
//
// 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)
{
	// Hints:
	//  Load each program segment into virtual memory
	//  at the address specified in the ELF segment 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.
```

```c
        //  (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.

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

        // LAB 3: Your code here.
}
```

It's difficult for me!

Two steps:

1. Parse an ELF binary image.
2. Load the ELP binary image content into the user address space of the new environment.

3.3.6 The Implementation of env_create() Function

```c
// 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, enum EnvType type)
{
        // LAB 3: Your code here.
}
```

There are mainly three steps:

1. Allocate a new env with env_alloc().
2. Load the named elf binary into this env with load_icode().
3. Set the env type.

It seems quite easy now:

```
void
env_create(uint8_t *binary, enum EnvType type)
{
        // LAB 3: Your code here.
        struct Env *env;
        int rc;
        // step 1: allocates a new env with env_alloc
        rc = env_alloc(&e, 0);
        if(rc != 0)
                panic("env_create: env_alloc failed");
        // step 2: loads the named elf binary with load_icode
        load_icode(env, binary);
        // step 3: set env's env_type
        env->env_type = type;

}
```


3.3.7 The Implementation of env_run() Function

Let's firstly revisit its hints:

```
//
// 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.

        panic("env_run not yet implemented");
}
```

Just follows the instructions:

```
void
env_run(struct Env *e)
{
      // LAB 3: Your code here.
```

- start (kern/entry.S)
- i386_init (kern/init.c)
  - cons_init
  - mem_init
  - env_init
  - trap_init (still incomplete at this point)
  - env_create
  - env_run
    - env_pop_tf

Figure 1: Call graph

```
        // Step 1: change the curenv's env_status from ING to ABLE.

        if(curenv != NULL && curenv->env_status == ENV_RUNNING)
                curenv->env_status = ENV_RUNNABLE;

        // Step 2: switch to the new env
        curenv = e;
        // Step 3: set status
        curenv->env_status = ENV_RUNNING;
        // Step 4: update env_runs counter
        curenv->env_runs++;
        // Step 5: use lcr3() to switch to its address space
        lcr3(PADDR(curenv->env_pgdir));

        // Step 6: use env_pop_tf() to restore the env's register
        //         and drop into user mode in the environment.
        env_pop_tf(&curenv->env_tf);


        panic("env_run not yet implemented");
}
```

Let's check what's lcr3() function is for: It's defined in inc/x86.h

```
static inline void
lcr3(uint32_t val)
{
        asm volatile("movl %0,%%cr3" : : "r" (val));
}
```

3.4 The Call Graph

3.5 Basics of Protected Control Transfer

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 environments.

There is difference between **exceptions** and **interrupts** 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 synchronously by the currently running code, for example due to a divide by zero or an invalid memory access

There are two mechanism working together to provide the control transfer protection:

1. **The Interrupt Descriptor Table (IDT):**
2. **The Task State Segment:** 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 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.

3.6 Types of Exceptions and Interrupts

1. **Synchronous exceptions (the processor exception)**: All of the synchronous exceptions that the x86 processor can generate internally use interrupt vectors between 0 and 31, and therefore map to IDT entries 0-31.
2. **Software interrupts and Asynchronous hardware interrupts**: 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.

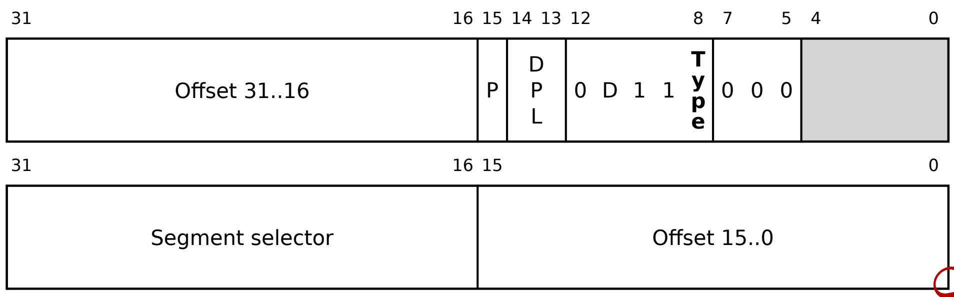
3.7 Setting Up the IDT

Let's see the details:

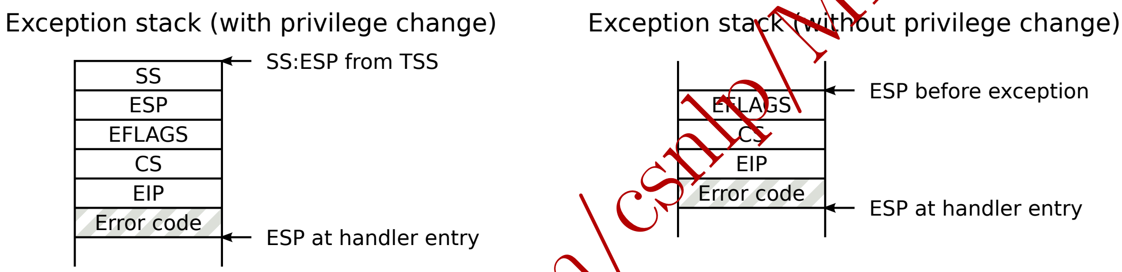
1. The **trap_init()** function initialize the IDT with the address of exception or interrupt handlers.
2. These handlers are implemented in **trapentry.S**.
3. Each of these handlers should build a **struct Trapframe** (see in **inc/trap.h**) on the stack and call **trap()** with a pointer to the Trapframe.
4. **trap()** then handles the exception/interrupt or dispatches to a specific handler function.

3.8 IDT of x86

Interrupt/trap gate

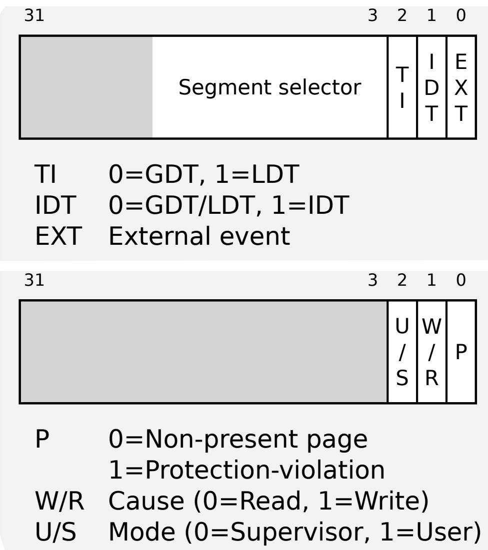


Type	0=Interrupt gate	P	Present
	1=Trap gate	DPL	Descriptor privilege level
Selector	Destination CS		(CPL required to invoke gate)
Offset	Destination IP or EIP	D	Size of gate (0=16-bits, 1=32-bits)



Vector	Description	Type	Error code
0	Divide error	Fault	No
1	Reserved		
2	Non-maskable interrupt	Interrupt	No
3	Breakpoint	Trap	No
4	Overflow	Trap	No
5	BOUND range exceeded	Fault	No
6	Invalid/undefined opcode	Fault	No
7	No math coprocessor	Fault	No
8	Double fault	Abort	Zero
9	Reserved		
10	Invalid TSS	Fault	Yes
11	Segment not present	Fault	Yes
12	Stack-segment fault	Fault	Yes
13	General protection	Fault	Yes
14	Page fault	Fault	Yes
15	Reserved		No
16	x87 FPU error	Fault	No
17	Alignment check	Fault	Zero
18	Machine check	Abort	No
19	SIMD FP exception	Fault	No
20-31	Reserved		
32-255	User defined interrupts	Interrupt	No

Exception types	
Fault	Faulting instruction not executed CS:EIP is the faulting instruction
Trap	Trapping instruction executed CS:EIP is the next instruction
Abort	Location is imprecise; cannot safely resume execution



```
        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
|                |              // ...
+----------------+
```

Figure 2: Setting up the IDT

Exercise 4. Edit `trapentry.S` and `trap.c` and implement the features described above. 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.

Figure 3: Exercise 4

3.9 Exercise 4

3.9.1 The Hints of kern/trapentry.S

```
/* See COPYRIGHT for copyright information. */

#include <inc/mmu.h>
#include <inc/memlayout.h>
#include <inc/trap.h>



###################################################################
# exceptions/interrupts
###################################################################

```

```
/* TRAPHANDLER 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.
 *
 * You shouldn't call a TRAPHANDLER function from C, but you may
 * need to _declare_ one in C (for instance, to get a function pointer
 * during IDT setup).  You can declare the function with
 *   void NAME();
 * where NAME is the argument passed to TRAPHANDLER.
 */
#define TRAPHANDLER(name, num)            \
  .globl name;    /* define global symbol for 'name' */ \
  .type name, @function;  /* symbol type is function */   \
  .align 2;   /* align function definition */   \
  name:     /* function starts here */    \
  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

.text

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



/*
 * Lab 3: Your code here for _alltraps
 */
```

We should notice that macros **TRAPHANDLER** and **TRAPHANDLER_NOEC** (no error code needs to be pushed) are used for traps. That's enough.

Next, let's see the content in file **inc/trap.h**

```
#ifndef JOS_INC_TRAP_H
#define JOS_INC_TRAP_H

// Trap numbers
// These are processor defined:
#define T_DIVIDE     0    // divide error
#define T_DEBUG      1    // debug exception
#define T_NMI        2    // non-maskable interrupt
#define T_BRKPT      3    // breakpoint
#define T_OFLOW      4    // overflow
#define T_BOUND      5    // bounds check
```

```
#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_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_FPERR     16    // floating point error
#define T_ALIGN     17    // aligment check
#define T_MCHK      18    // machine check
#define T_SIMDERR   19    // SIMD floating point error

// These are arbitrarily chosen, but with care not to overlap
// processor defined exceptions or interrupt vectors.
#define T_SYSCALL   48    // system call
#define T_DEFAULT   500   // catchall

#define IRQ_OFFSET  32  // IRQ 0 corresponds to int IRQ_OFFSET

// Hardware IRQ numbers. We receive these as (IRQ_OFFSET+IRQ_WHATEVER)
#define IRQ_TIMER        0
#define IRQ_KBD          1
#define IRQ_SERIAL       4
#define IRQ_SPURIOUS     7
#define IRQ_IDE         14
#define IRQ_ERROR       19

#ifndef __ASSEMBLER__

#include <inc/types.h>

struct PushRegs {
  /* registers as pushed by pusha */
  uint32_t reg_edi;
  uint32_t reg_esi;
  uint32_t reg_ebp;
  uint32_t reg_oesp;    /* Useless */
  uint32_t reg_ebx;
  uint32_t reg_edx;
  uint32_t reg_ecx;
  uint32_t reg_eax;
} __attribute__((packed));

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));


#endif /* !__ASSEMBLER__ */

#endif /* !JOS_INC_TRAP_H */
```

3.9.2 The Hints of kern/trap.c

Actually, we should also see the code in **kern/trap.c**.

```
#include <inc/mmu.h>
#include <inc/x86.h>
#include <inc/assert.h>

#include <kern/pmap.h>
#include <kern/trap.h>
#include <kern/console.h>
#include <kern/monitor.h>
#include <kern/env.h>
#include <kern/syscall.h>

static struct Taskstate ts;

/* For debugging, so print_trapframe can distinguish between printing
 * a saved trapframe and printing the current trapframe and print some
 * additional information in the latter case.
 */
static struct Trapframe *last_tf;

/* Interrupt descriptor table.  (Must be built at run time because
 * shifted function addresses can't be represented in relocation records.)
 */
struct Gatedesc idt[256] = { { 0 } };
struct Pseudodesc idt_pd = {
  sizeof(idt) - 1, (uint32_t) idt
};


static const char *trapname(int trapno)
{
  static const char * const excnames[] = {
    "Divide error",
    "Debug",
    "Non-Maskable Interrupt",
    "Breakpoint",
    "Overflow",
    "BOUND Range Exceeded",
    "Invalid Opcode",
    "Device Not Available",
    "Double Fault",
    "Coprocessor Segment Overrun",
    "Invalid TSS",
    "Segment Not Present",
```

```c
    "Stack Fault",
    "General Protection",
    "Page Fault",
    "(unknown trap)",
    "x87 FPU Floating-Point Error",
    "Alignment Check",
    "Machine-Check",
    "SIMD Floating-Point Exception"
  };

  if (trapno < ARRAY_SIZE(excnames))
    return excnames[trapno];
  if (trapno == T_SYSCALL)
    return "System call";
  return "(unknown trap)";
}


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

  // LAB 3: Your code here.

  // Per-CPU setup
  trap_init_percpu();
}

// Initialize and load the per-CPU TSS and IDT
void
trap_init_percpu(void)
{
  // 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;
  ts.ts_iomb = sizeof(struct Taskstate);

  // Initialize the TSS slot of the gdt.
  gdt[GD_TSS0 >> 3] = SEG16(STS_T32A, (uint32_t) (&ts),
          sizeof(struct Taskstate) - 1, 0);
  gdt[GD_TSS0 >> 3].sd_s = 0;

  // Load the TSS selector (like other segment selectors, the
  // bottom three bits are special; we leave them 0)
  ltr(GD_TSS0);

  // Load the IDT
  lidt(&idt_pd);
}

void
print_trapframe(struct Trapframe *tf)
{
  cprintf("TRAP frame at %p\n", tf);
  print_regs(&tf->tf_regs);
  cprintf("  es   0x----%04x\n", tf->tf_es);
```

```
	cprintf("  ds   0x----%04x\n", tf->tf_ds);
	cprintf("  trap 0x%08x %s\n", tf->tf_trapno, trapname(tf->tf_trapno));
	// If this trap was a page fault that just happened
	// (so %cr2 is meaningful), print the faulting linear address.
	if (tf == last_tf && tf->tf_trapno == T_PGFLT)
		cprintf("  cr2  0x%08x\n", rcr2());
	cprintf("  err  0x%08x", tf->tf_err);
	// For page faults, print decoded fault error code:
	// U/K=fault occurred in user/kernel mode
	// W/R=a write/read caused the fault
	// PR=a protection violation caused the fault (NP=page not present).
	if (tf->tf_trapno == T_PGFLT)
		cprintf(" [%s, %s, %s]\n",
			tf->tf_err & 4 ? "user" : "kernel",
			tf->tf_err & 2 ? "write" : "read",
			tf->tf_err & 1 ? "protection" : "not-present");
	else
		cprintf("\n");
	cprintf("  eip  0x%08x\n", tf->tf_eip);
	cprintf("  cs   0x----%04x\n", tf->tf_cs);
	cprintf("  flag 0x%08x\n", tf->tf_eflags);
	if ((tf->tf_cs & 3) != 0) {
		cprintf("  esp  0x%08x\n", tf->tf_esp);
		cprintf("  ss   0x----%04x\n", tf->tf_ss);
	}
}

void
print_regs(struct PushRegs *regs)
{
	cprintf("  edi  0x%08x\n", regs->reg_edi);
	cprintf("  esi  0x%08x\n", regs->reg_esi);
	cprintf("  ebp  0x%08x\n", regs->reg_ebp);
	cprintf("  oesp 0x%08x\n", regs->reg_oesp);
	cprintf("  ebx  0x%08x\n", regs->reg_ebx);
	cprintf("  edx  0x%08x\n", regs->reg_edx);
	cprintf("  ecx  0x%08x\n", regs->reg_ecx);
	cprintf("  eax  0x%08x\n", regs->reg_eax);
}

static void
trap_dispatch(struct Trapframe *tf)
{
	// Handle processor exceptions.
	// LAB 3: Your code here.

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

void
trap(struct Trapframe *tf)
```

```
{
  // The environment may have set DF and some versions
  // of GCC rely on DF being clear
  asm volatile("cld" ::: "cc");

  // Check that interrupts are disabled.  If this assertion
  // fails, DO NOT be tempted to fix it by inserting a "cli" in
  // the interrupt path.
  assert(!(read_eflags() & FL_IF));

  cprintf("Incoming TRAP frame at %p\n", tf);

  if ((tf->tf_cs & 3) == 3) {
    // Trapped from user mode.
    assert(curenv);

    // Copy trap frame (which is currently on the stack)
    // into 'curenv->env_tf', so that running the environment
    // will restart at the trap point.
    curenv->env_tf = *tf;
    // The trapframe on the stack should be ignored from here on.
    tf = &curenv->env_tf;
  }

  // Record that tf is the last real trapframe so
  // print_trapframe can print some additional information.
  last_tf = tf;

  // Dispatch based on what type of trap occurred
  trap_dispatch(tf);

  // Return to the current environment, which should be running.
  assert(curenv && curenv->env_status == ENV_RUNNING);
  env_run(curenv);
}


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.

  // 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);
}
```

kern/trap.c has these functions:

1. static const char *trapname(int trapno) : get trap name.
2. void trap_init(void): trap initialization.
3. void trap_init_percpu(void): Initialize and load the per-CPU TSS and IDT, called from trap_init.
4. void print_trapframe(struct Trapframe *tf): as what the name suggests.
5. void print_regs(struct PushRegs *regs): print registers' value.
6. static void trap_dispatch(struct Trapframe *tf): dispatch the traps according to the trapframe.
7. void trap(struct Trapframe *tf): the main trap function.
8. void page_fault_handler(struct Trapframe *tf)

3.9.3 The Implementation of kern/trapentry.S

```
/*
 * Lab 3: Your code here for generating entry points for the different traps.
 */
TRAPHANDLER_NOEC(DIVIDE, T_DIVIDE)
TRAPHANDLER_NOEC(DEBUG, T_DEBUG)
TRAPHANDLER_NOEC(NMI, T_NMI)
TRAPHANDLER_NOEC(BRKPT, T_BRKPT)
TRAPHANDLER_NOEC(OFLOW, T_OFLOW)
TRAPHANDLER_NOEC(BOUND, T_BOUND)
TRAPHANDLER_NOEC(ILLOP, T_ILLOP)
TRAPHANDLER_NOEC(DEVICE, T_DEVICE)
TRAPHANDLER(DBLFLT, T_DBLFLT)
TRAPHANDLER(TSS, T_TSS)
TRAPHANDLER(SEGNP, T_SEGNP)
TRAPHANDLER(STACK, T_STACK)
TRAPHANDLER(GPFLT, T_GPFLT)
TRAPHANDLER(PGFLT, T_PGFLT)
TRAPHANDLER_NOEC(GPFLT, T_GPFLT)
TRAPHANDLER(ALIGN, T_ALIGN)
TRAPHANDLER_NOEC(MCHK, T_MCHK)
TRAPHANDLER_NOEC(SIMDERR, T_SIMDERR)
/*
 SYSCALL: 48
*/
TRAPHANDLER_NOEC(SYSCALL, T_SYSCALL)


/*
 * Lab 3: Your code here for _alltraps
 */
_alltraps:
  pushl %ds
  pushl %es
  pushall

```

```
  movl $GD_KD, %eax
  movw %ax, %ds
  movw %ax, %es

  push %esp
  call trap
```

3.9.4 The Implementation trap_init() in kern/trap.c

Let's firstly see the macros **SETGATE** in **inc/mmu.h**

```
// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
    //   see section 9.6.1.3 of the i386 reference: "The difference between
    //   an interrupt gate and a trap gate is in the effect on IF (the
    //   interrupt-enable flag). An interrupt that vectors through an
    //   interrupt gate resets IF, thereby preventing other interrupts from
    //   interfering with the current interrupt handler. A subsequent IRET
    //   instruction restores IF to the value in the EFLAGS image on the
    //   stack. An interrupt through a trap gate does not change IF."
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//        the privilege level required for software to invoke
//        this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, dpl)                    \
{                                                               \
        (gate).gd_off_15_0 = (uint32_t) (off) & 0xffff;         \
        (gate).gd_sel = (sel);                                  \
        (gate).gd_args = 0;                                     \
        (gate).gd_rsv1 = 0;                                     \
        (gate).gd_type = (istrap) ? STS_TG32 : STS_IG32;        \
        (gate).gd_s = 0;                                        \
        (gate).gd_dpl = (dpl);                                  \
        (gate).gd_p = 1;                                        \
        (gate).gd_off_31_16 = (uint32_t) (off) >> 16;           \
}
```

3.10 Questions Following Exercise 4

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?)*

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)?*

4 Part B: Page Faults, Breakpoints Exceptions, and System Calls

4.1 Handling Page Faults

The page fault exception: interrupt vector 14(T_PGFLT). When the processor takes a page fault, it stores the linear (virtual address) that caused this fault and store it in a special register: **CR2**.

4.1.1 Exercise 5

*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, faultread-kernel, 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.*

The hints of original trap_dispatch()

```
static void
trap_dispatch(struct Trapframe *tf)
{
        // Handle processor exceptions.
        // LAB 3: Your code here.

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

The implementation is quite simple:

```
static void
trap_dispatch(struct Trapframe *tf)
{
        // Handle processor exceptions.
        // LAB 3: Your code here.
        switch(tf-tf_trapno) {
                case T_PGFLT:
                        page_fault_handler(tf);
                default:
                        break;
        }

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

```
        }
}
```

4.2 The Breakpoint Exception

4.2.1 Exercise 6

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.

The implementation is quite simple:

```
                case T_BRKPT:
                        monitor(tf);
                        return;
```

But it seems do not work!!! Why, recall that we use

```
SETGATE(idt[T_BRKPT], 0, GD_KT, t_brkpt, 0);
```

However, the truely breakpoint use **int $3**. The privilege level doesn't match.

So, we change the above command in **trap_init()** to

```
SETGATE(idt[T_BRKPT], 0, GD_KT, t_brkpt, 3);
```

4.3 System Calls: Exercise 7

User processes ask the kernel to do things for them by invoking system calls. When the user process invokes a system call, the processor enters kernel mode, the processor and the kernel cooperate to save the user process's state, the kernel executes appropriate code in order to carry out the system call, and then resumes the user process.

4.3.1 Exercise 7

Exercise 7. 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.

Figure 4: Exercise 7

The whole procedure is as follows:

1. trap_dispatch() handle the T_SYSCALL.
2. trap_dispatch() calls syscall() in kern/syscall.c

See the hints about syscall() in kern/syscall.c:

```
// 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.

        panic("syscall not implemented");

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

4.3.2 The Implementation of syscall() in kern/syscall.c

```
// 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.


        //panic("syscall not implemented");

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

syscall() calls sys_cputs, which also requires implementation.

4.3.3 The Implementation of sys_cputs() in kern/syscall.c

```
// Print a string to the system console.
// The string is exactly 'len' characters long.
```

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

	// LAB 3: Your code here.
	if (curenv->env_tf.tf_cs & 3)
		user_mem_assert(curenv, s, len, 0);

	// Print the string supplied by the user.
	cprintf("%.*s", len, s);
}
```

4.3.4 DO NOT FORGET TRAP_INIT and TRAP_DISPATCH

In trap_init():

```c
	void SYSCALL();
	SETGATE(idt[48], 0, GD_KT, SYSCALL, 3);
```

In trap_dispatch(),

```c
static void
trap_dispatch(struct Trapframe *tf)
{
	// Handle processor exceptions.
	// LAB 3: Your code here.
	switch(tf->tf_trapno) {
		case T_PGFLT:
			page_fault_handler(tf);
			return;
		case T_BRKPT:
			monitor(tf);
			return;
		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;
		default:
			break;
	}

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

NOW: 60/80 score.

4.4 User-mode startup

A user program starts from lib/entry.S. After some setup, this code calls libmain() in lib/libmain.c.

4.4.1 Exercise 8

A user program starts running at the top of `lib/entry.S`. After some setup, this code calls `libmain()`, in `lib/libmain.c`. You should modify `libmain()` to initialize the global pointer `thisenv` to point at this environment's `struct Env` in the `envs[]` array. (Note that `lib/entry.S` has already defined `envs` to point at the `UENVS` mapping you set up in Part A.) Hint: look in `inc/env.h` and use `sys_getenvid`.

`libmain()` then calls `umain`, which, in the case of the hello program, is in `user/hello.c`. Note that after printing "`hello, world`", it tries to access `thisenv->env_id`. This is why it faulted earlier. Now that you've initialized `thisenv` properly, it should not fault. If it still faults, you probably haven't mapped the `UENVS` area user-readable (back in Part A in `pmap.c`; this is the first time we've actually used the `UENVS` area).

> **Exercise 8.** 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.

Figure 5: Exercise 8

4.4.2 The Content in lib/entry.S

```
#include <inc/mmu.h>
#include <inc/memlayout.h>

.data
// Define the global symbols 'envs', 'pages', 'uvpt', and 'uvpd'
// so that they can be used in C as if they were ordinary global arrays.
  .globl envs
  .set envs, UENVS
  .globl pages
  .set pages, UPAGES
  .globl uvpt
  .set uvpt, UVPT
  .globl uvpd
  .set uvpd, (UVPT+(UVPT>>12)*4)


// Entrypoint - this is where the kernel (or our parent environment)
// starts us running when we are initially loaded into a new environment.
.text
.globl _start
_start:
  // See if we were started with arguments on the stack
  cmpl $USTACKTOP, %esp
  jne args_exist

  // If not, push dummy argc/argv arguments.
```

```
  // This happens when we are loaded by the kernel,
  // because the kernel does not know about passing arguments.
  pushl $0
  pushl $0

args_exist:
  call libmain
1:  jmp 1b
```

4.4.3 libmain() Function in lib/libmain.c

```c
// Called from entry.S to get us going.
// entry.S already took care of defining envs, pages, uvpd, and uvpt.

#include <inc/lib.h>

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

const volatile struct Env *thisenv;
const char *binaryname = "<unknown>";

void
libmain(int argc, char **argv)
{
        // set thisenv to point at our Env structure in envs[].
        // LAB 3: Your code here.
        thisenv = &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();
}
```

Now: 65/80.

4.5 Page faults and memory protection

4.5.1 Crucial Memory Protection Challenges Brought By System Calls

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.**

How to solve this problem?

1. When a program passes the kernel a pointer, the kernel will check that the address is in the user part of the address space, and that the page table would allow the memory operation. Thus, the kernel will never suffer a page fault due to dereferencing a user-supplied pointer.

2. If the kernel does page fault, it should panic and terminate.

4.5.2 Exercise 9

Exercise 9. Change `kern/trap.c` to panic if a page fault happens in kernel mode.

Hint: to determine whether a fault happened in user mode or in kernel mode, check the low bits of the `tf_cs`.

Read `user_mem_assert` in `kern/pmap.c` and implement `user_mem_check` in that same file.

Change `kern/syscall.c` to sanity check arguments to system calls.

Boot your kernel, running `user/buggyhello`. The environment should be destroyed, and the kernel should *not* panic. You should see:

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

Finally, change `debuginfo_eip` in `kern/kdebug.c` to call `user_mem_check` on `usd`, `stabs`, and `stabstr`. If you now run `user/breakpoint`, you should be able to run `backtrace` from the kernel monitor and see the backtrace traverse into `lib/libmain.c` before the kernel panics with a page fault. What causes this page fault? You don't need to fix it, but you should understand why it happens.

Figure 6: Exercise 9

We have the following steps:

1. Check it is in the kernel mode or user mode: check low bits of the **tf_cs**.

2. Implement **user_mem_assert()** in **kern/pmap.c**.

3. ddd

The following is about the code work:

1. **Check the mode:**

```
void
page_fault_handler(struct Trapframe *tf)
{
        uint32_t fault_va;

```

```c
        // 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 && 0x01 == 0) {
                panic("page fault in kernel mode, fault address %d\n", fault_va);
        }

        // 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);
}
```

Listing 2: page_fault_handler() in kern/trap.c

2. **Check the user_mem_assert()**

```c
//
// Checks that environment 'env' is allowed to access the range
// of memory [va, va+len) with permissions 'perm | PTE_U | PTE_P'.
// If it can, then the function simply returns.
// If it cannot, 'env' is destroyed and, if env is the current
// environment, this function will not return.
//
void
user_mem_assert(struct Env *env, const void *va, size_t len, int perm)
{
        if (user_mem_check(env, va, len, perm | PTE_U) < 0) {
                cprintf("[%08x] user_mem_check assertion failure for "
                        "va %08x\n", env->env_id, user_mem_check_addr);
                env_destroy(env);       // may not return
        }
}
```

It calls user_mem_check()

```c
//
// Check that an environment is allowed to access the range of memory
// [va, va+len) with permissions 'perm | PTE_P'.
// Normally 'perm' will contain PTE_U at least, but this is not required.
// 'va' and 'len' need not be page-aligned; you must test every page that
// contains any of that range.  You will test either 'len/PGSIZE',
// 'len/PGSIZE + 1', or 'len/PGSIZE + 2' pages.
//
// A user program can access a virtual address if (1) the address is below
// ULIM, and (2) the page table gives it permission.  These are exactly
// the tests you should implement here.
//
// If there is an error, set the 'user_mem_check_addr' variable to the first
// erroneous virtual address.
```

```
//
// Returns 0 if the user program can access this range of addresses,
// and -E_FAULT otherwise.
//
int
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
        // LAB 3: Your code here.

        return 0;
}
```

See its implementation:

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

        pte_t *cur_pg = NULL;
        for(; start<end; start+=PGSIZE) {
                cur_pg = pgdir_walk(env->env_pgdir, (void *)start, 0);
                if((int) start > ULIM || cur_pg == NULL ||(((uint32_t)(*cur_pg) & perm) != perm)) {
                        if(start == ROUNDDOWN((char *)va, PGSIZE)) {
                                user_mem_check_addr = (uintptr_t) va;
                        }
                        else {
                                user_mem_check_addr = (uintptr_t) start;
                        }
                        return -E_FAULT;
                }
        }

        return 0;
}
```

Now, we finally finish this lab: 80/80.

5 Reference

1. bysui's github and blog
    - github: https://github.com/bysui/mit6.828
    - blog: https://blog.csdn.net/bysui
2. SmallPond's github and blog
    - github: https://github.com/SmallPond/MIT6.828_OS
    - blog: https://me.csdn.net/Small_Pond
3. SimpCosm's github
    - github: https://github.com/SimpCosm/6.828
4. fatsheep9146's blog
    - blog: https://www.cnblogs.com/fatsheep9146/category/769143.html