236376 OSE Lab 3: User Environments

Due date:
: See Webcourse.

TA in charge:
: See Webcourse

Please note: You should **not** publish your lab solutions in any publicly accessible site such as github.

Lab Q&A

We encourage you to ask questions on course's Piazza forum. If no help provided on Piazza forum then try to email TA in charge. E-mails regarding this lab (such as administrative issues) should be sent with the subject "OSE, lab3".

Introduction

In this lab you will implement the basic kernel facilities required to get a protected user-mode environment (i.e., "process") running. You will enhance the JOS kernel to set up the data structures to keep track of user environments, create a single user environment, load a program image into it, and start it running. You will also make the JOS kernel capable of handling any system calls the user environment makes and handling any other exceptions it causes.

Note: In this lab, the terms *environment* and *process* are interchangeable - both refer to an abstraction that allows you to run a program. We introduce the term "environment" instead of the traditional term "process" in order to stress the point that JOS environments and UNIX processes provide different interfaces, and do not provide the same semantics.

Getting Started

Use Git to commit your changes after your Lab 2 submission (if any), fetch the latest version of the course repository, and then create a local branch called `lab3` based on our lab3 branch, `origin/lab3`:

```
$ git commit -am 'changes to lab2 after handin'
Created commit 734fab7: changes to lab2 after handin
 4 files changed, 42 insertions(+), 9 deletions(-)
$ git pull
Already up-to-date.
$ git checkout -b lab3 origin/lab3
Branch lab3 set up to track remote branch refs/remotes/origin/lab3.
Switched to a new branch "lab3"
$ git merge lab2
Merge made by recursive.
 kern/pmap.c |   42 +++++++++++++++++++
 1 files changed, 42 insertions(+), 0 deletions(-)
$
```

Lab 3 contains a number of new source files, which you should browse:

inc/	env.h	Public definitions for user-mode environments
	trap.h	Public definitions for trap handling
	syscall.h	Public definitions for system calls from user environments to the kernel
	lib.h	Public definitions for the user-mode support library
kern/	env.h	Kernel-private definitions for user-mode environments
	env.c	Kernel code implementing user-mode environments
	trap.h	Kernel-private trap handling definitions
	trap.c	Trap handling code
	trapentry.S	Assembly-language trap handler entry-points
	syscall.h	Kernel-private definitions for system call handling
	syscall.c	System call implementation code
lib/	Makefrag	Makefile fragment to build user-mode library, `obj/lib/libuser.a`
	entry.S	Assembly-language entry-point for user environments
	libmain.c	User-mode library setup code called from `entry.S`
	syscall.c	User-mode system call stub functions
	console.c	User-mode implementations of `putchar` and `getchar`, providing console I/O
	exit.c	User-mode implementation of `exit`
	panic.c	User-mode implementation of `panic`
user/	*	Various test programs to check kernel lab 3 code

In addition, a number of the source files we handed out for lab2 are modified in lab3. To see the differences, you can type:

```
$ git diff lab2
```

You may also want to take another look at the lab tools guide, as it includes information on debugging user code that becomes relevant in this lab.

Lab Requirements

As in lab 2, you will need to do all of the regular exercises described in the lab and *at least one* challenge problem. Write brief answers to the questions posed in the lab and a one or two paragraph description of what you did to solve your chosen challenge problem in the questionary. (If you implement more than one challenge problem, you only need to describe one of them in the write-up.)

Hand-In Procedure

When you are ready to hand in your lab (including the filled `lab3-questionary.txt`), run **`make handin`** in the source directory. This will make a tar file for you, which you can then submit via webcourse site. You can list the contents of the tar file with **`tar -tvzf lab3-handin.tar.gz`** or unpack it (in another directory) with **`tar -xzf lab3-handin.tar.gz`**.

As before, we will be grading your solutions with a grading program. You can run **`make grade`** in the `lab` directory to test your kernel with the grading program (no test for the questionary is provided). The grading program may rely on some in-kernel code for the check. Needless to say, that altering this code or otherwise deceiving automatic testing is considered severe cheating.

Inline Assembly

In this lab you may find GCC's inline assembly language feature useful, although it is also possible to complete the lab without using it. At the very least, you will need to be able to understand the fragments of inline assembly language ("`asm`" statements) that already exist in the source code we gave you. You can find several sources of information on GCC inline assembly language on the class reference materials page.

Part A: User Environments and Exception Handling

The new include file `inc/env.h` contains basic definitions for user environments in JOS. Read it now. The kernel uses the `Env` data structure to keep track of each user environment. In this lab you will initially create just one environment, but you will need to design the JOS kernel to support multiple environments; lab 4 will take advantage of this feature by allowing a user environment to `fork` other environments.

As you can see in `kern/env.c`, 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
```

Once JOS gets up and running, the `envs` pointer points to an array of `Env` structures representing all the environments in the system. In our design, the JOS kernel will support a maximum of `NENV` simultaneously active environments, although there will typically be far fewer running environments at any given time. (`NENV` is a constant `#define`'d in `inc/env.h`.) Once it is allocated, the `envs` array will contain a single instance of the `Env` data structure for each of the `NENV` possible environments.

The JOS kernel keeps all of the inactive `Env` structures on the `env_free_list`. This design allows easy allocation and deallocation of environments, as they merely have to be added to or removed from the free list.

The kernel uses the `curenv` symbol to keep track of the *currently executing* environment at any given time. During boot up, before the first environment is run, `curenv` is initially set to `NULL`.

Environment State

The `Env` structure is defined in `inc/env.h` as follows (although more fields will be added in future labs):

```
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
};
```

Here's what the `Env` fields are for:

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.

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.

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.

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.

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.

env_status:
: This variable holds one of the following values:

  `ENV_FREE`:
  : Indicates that the `Env` structure is inactive, and therefore on the `env_free_list`.

  `ENV_RUNNABLE`:
  : Indicates that the `Env` structure represents an environment that is waiting to run on the processor.

  `ENV_RUNNING`:
  : Indicates that the `Env` structure represents the currently running environment.

  `ENV_NOT_RUNNABLE`:
  : Indicates that the `Env` structure represents a currently active environment, but it is not currently ready to run: for example, because it is waiting for an interprocess communication (IPC) from another environment.

  `ENV_DYING`:
  : Indicates that the `Env` structure represents a zombie environment. A zombie environment will be freed the next time it traps to the kernel. We will not use this flag until Lab 4.

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

Like a Unix process, a JOS environment couples the concepts of "thread" and "address space". The thread is defined primarily by the saved registers (the `env_tf` field), and the address space is defined by the page directory and page tables pointed to by `env_pgdir`. To run an environment, the kernel must set up the CPU with *both* the saved registers and the appropriate address space.

Our `struct Env` is analogous to `struct proc` in xv6. Both structures hold the environment's (i.e., process's) user-mode register state in a `Trapframe` structure. In JOS, individual environments do not have their own kernel stacks as processes do in xv6. There can be only one JOS environment active in the kernel at a time, so JOS needs only a *single* kernel stack.

Allocating the Environments Array

In lab 2, you allocated memory in `mem_init()` for the `pages[]` array, which is a table the kernel uses to keep track of which pages are free and which are not. You will now need to modify `mem_init()` further to allocate a similar array of `Env` structures, called `envs`.

> **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.
>
> You should run your code and make sure `check_kern_pgdir()` succeeds.

Creating and Running Environments

You will now write the code in `kern/env.c` necessary to run a user environment. Because we do not yet have a filesystem, we will set up the kernel to load a static binary image that is *embedded within the kernel itself*. JOS embeds this binary in the kernel as a ELF executable image.

The Lab 3 `GNUmakefile` generates a number of binary images in the `obj/user/` directory. If you look at `kern/Makefrag`, you will notice some magic that "links" these binaries directly into the kernel executable as if they were `.o` files. The `-b binary` option on the linker command line causes these files to be linked in as "raw" uninterpreted binary files rather than as regular `.o` files produced by the compiler. (As far as the linker is concerned, these files do not have to be ELF images at all - they could be anything, such as text files or pictures!) If you look at `obj/kern/kernel.sym` after building the kernel, you will notice that the linker has "magically" produced a number of funny symbols with obscure names like `_binary_obj_user_hello_start`, `_binary_obj_user_hello_end`, and `_binary_obj_user_hello_size`. The linker generates these symbol names by mangling the file names of the binary files; the symbols provide the regular kernel code with a way to reference the embedded binary files.

In `i386_init()` in `kern/init.c` you'll see code to run one of these binary images in an environment. However, the critical functions to set up user environments are not complete; you will need to fill them in.

> **Exercise 2.** 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` load an ELF binary into it.
>
> `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".

Below is a call graph of the code up to the point where the user code is invoked. Make sure you understand the purpose of each step.

- `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`

Once you are done you should compile your kernel and run it under QEMU. If all goes well, your system should enter user space and execute the `hello` binary until it makes a system call with the `int` instruction. At that point there will be trouble, since JOS has not set up the hardware to allow any kind of transition from user space into the kernel. When the CPU discovers that it is not set up to handle this system call interrupt, it will generate a general protection exception, find that it can't handle that, generate a double fault exception, find that it can't handle that either, and finally give up with what's known as a "triple fault". Usually, you would then see the CPU reset and the system reboot. While this is important for legacy applications (see this blog post for an explanation of why), it's a pain for kernel development, so with the 6.828 patched QEMU you'll instead see a register dump and a "Triple fault." message.

We'll address this problem shortly, but for now we can use the debugger to check that we're entering user mode. Use **`make qemu-gdb`** and set a GDB breakpoint at `env_pop_tf`, which should be the last function you hit before actually entering user mode. Single step through this function using **`si`**; the processor should enter user mode after the `iret` instruction. You should then see the first instruction in the user environment's executable, which is the `cmpl` instruction at the label `start` in `lib/entry.S`. Now use **`b *0x...`** to set a breakpoint at the `int $0x30` in `sys_cputs()` in `hello` (see `obj/user/hello.asm` for the user-space address). This `int` is the system call to display a character to the console. If you cannot execute as far as the `int`, then something is wrong with your address space setup or program loading code; go back and fix it before continuing.

Handling Interrupts and Exceptions

At this point, the first `int $0x30` system call instruction in user space is a dead end: once the processor gets into user mode, there is no way to get back out. You will now need to implement basic exception and system call handling, so that it is possible for the kernel to recover control of the processor from user-mode code. The first thing you should do is thoroughly familiarize yourself with the x86 interrupt and exception mechanism.

> **Exercise 3.** Read Chapter 9, Exceptions and Interrupts in the 80386 Programmer's Manual (or Chapter 5 of the IA-32 Developer's Manual), if you haven't already.

In this lab we generally follow Intel's terminology for interrupts, exceptions, and the like. However, terms such as exception, trap, interrupt, fault and abort have no standard meaning across architectures or operating systems, and are often used without regard to the subtle distinctions between them on a particular architecture such as the x86. When you see these terms outside of this lab, the meanings might be slightly different.

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. 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.

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 the x86, two mechanisms work together to provide this protection:

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.)

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.

Types of Exceptions and Interrupts

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. For example, a page fault always causes an exception through vector 14. 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.

In this section we will extend JOS to handle the internally generated x86 exceptions in vectors 0-31. In the next section we will make JOS handle software interrupt vector 48 (0x30), which JOS (fairly arbitrarily) uses as its system call interrupt vector. In Lab 4 we will extend JOS to handle externally generated hardware interrupts such as the clock interrupt.

An Example

Let's put these pieces together and trace through an example. Let's say the processor is executing code in a user environment and encounters a divide instruction that attempts to divide by zero.

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`:

```
                     +--------------------+ KSTACKTOP
                     | 0x00000 | old SS   |     " - 4
                     |      old ESP       |     " - 8
                     |     old EFLAGS     |     " - 12
                     | 0x00000 | old CS   |     " - 16
                     |      old EIP       |     " - 20 <---- ESP
                     +--------------------+
```

3. Because we're handling a divide error, which is interrupt vector 0 on the x86, the processor reads IDT entry 0 and sets `CS:EIP` to point to the handler function described by the entry.
4. The handler function takes control and handles the exception, for example by terminating the user environment.

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*. The page fault exception, number 14, is an important example. See the 80386 manual to determine for which exception numbers the processor pushes an error code, and what the error code means in that case. 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:

```
                     +--------------------+ KSTACKTOP
                     | 0x00000 | old SS   |     " - 4
                     |      old ESP       |     " - 8
                     |     old EFLAGS     |     " - 12
                     | 0x00000 | old CS   |     " - 16
                     |      old EIP       |     " - 20
                     |     error code     |     " - 24 <---- ESP
                     +--------------------+
```

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. This capability is an important tool in implementing protection, as we will see later in the section on system calls.

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. For exception types that do not push an error code, the kernel stack therefore looks like the following on entry to the exception handler:

```
                     +--------------------+ <---- old ESP
                     |     old EFLAGS     |     " - 4
                     | 0x00000 | old CS   |     " - 8
                     |      old EIP       |     " - 12
                     +--------------------+
```

For exception types that push an error code, the processor pushes the error code immediately after the old `EIP`, as before.

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.

Setting Up the IDT

You should now have the basic information you need in order to set up the IDT and handle exceptions in JOS. For now, you will set up the IDT to handle interrupt vectors 0-31 (the processor exceptions). We'll handle system call interrupts later in this lab and add interrupts 32-47 (the device IRQs) in a later lab.

The header files `inc/trap.h` and `kern/trap.h` contain important definitions related to interrupts and exceptions that you will need to become familiar with. The file `kern/trap.h` contains definitions that are strictly private to the kernel, while `inc/trap.h` contains definitions that may also be useful to user-level programs and libraries.

Note: Some of the exceptions in the range 0-31 are defined by Intel to be reserved. Since they will never be generated by the processor, it doesn't really matter how you handle them. Do whatever you think is cleanest.

The overall flow of control that you should achieve is depicted below:

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

Each exception or interrupt should have its own handler in `trapentry.S` and `trap_init()` should initialize the IDT with the addresses of these handlers. Each of the handlers should build a `struct Trapframe` (see `inc/trap.h`) on the stack and call `trap()` (in `trap.c`) with a pointer to the Trapframe. `trap()` then handles the exception/interrupt or dispatches to a specific handler function.

> **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.

> *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`.

> **Questions**
>
> Answer the following questions in your `lab3-questionary.txt`:
>
> 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)?

This concludes part A of the lab.

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

Now that your kernel has basic exception handling capabilities, you will refine it to provide important operating system primitives that depend on exception handling.

Handling Page Faults

The page fault exception, interrupt vector 14 (`T_PGFLT`), is a particularly important one that we will exercise heavily throughout this lab and the next. When the processor takes a page fault, it stores the linear (i.e., virtual) address that caused the fault in a special processor control register, `CR2`. In `trap.c` we have provided the beginnings of a special function, `page_fault_handler()`, to handle page fault exceptions.

> **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`, `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`**.

You will further refine the kernel's page fault handling below, as you implement system calls.

The Breakpoint Exception

The breakpoint exception, interrupt vector 3 (`T_BRKPT`), is normally used to allow debuggers to insert breakpoints in a program's code by temporarily replacing the relevant program instruction with the special 1-byte `int3` software interrupt instruction. In JOS we will abuse this exception slightly by turning it into a primitive pseudo-system call that any user environment can use to invoke the JOS kernel monitor. This usage is actually somewhat appropriate if we think of the JOS kernel monitor as a primitive debugger. The user-mode implementation of `panic()` in `lib/panic.c`, for example, performs an `int3` after displaying its panic message.

> **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.

> *Challenge!* Modify the JOS kernel monitor so that you can 'continue' execution from the current location (e.g., after the `int3`, if the kernel monitor was invoked via the breakpoint exception), and so that you can single-step one instruction at a time. You will need to understand certain bits of the `EFLAGS` register in order to implement single-stepping.
>
> *Optional:* If you're feeling really adventurous, find some x86 disassembler source code - e.g., by ripping it out of QEMU, or out of GNU binutils, or just write it yourself - and extend the JOS kernel monitor to be able to disassemble and display instructions as you are stepping through them. Combined with the symbol table loading from lab 2, this is the stuff of which real kernel debuggers are made.

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

System calls

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. The exact details of how the user process gets the kernel's attention and how it specifies which call it wants to execute vary from system to system.

In the JOS kernel, we will use the `int` instruction, which causes a processor interrupt. In particular, we will use `int $0x30` as the system call interrupt. We have defined the constant `T_SYSCALL` to 48 (0x30) for you. You will have to set up the interrupt descriptor to allow user processes to cause that interrupt. Note that interrupt 0x30 cannot be generated by hardware, so there is no ambiguity caused by allowing user code to generate it.

The application will pass the system call number and the system call arguments in registers. This way, the kernel won't need to grub around in the user environment's stack or instruction stream. The system call number will go in `%eax`, and the arguments (up to five of them) will go in `%edx`, `%ecx`, `%ebx`, `%edi`, and `%esi`, respectively. The kernel passes the return value back in `%eax`. The assembly code to invoke a system call has been written for you, in `syscall()` in `lib/syscall.c`. You should read through it and make sure you understand what is going on.

> **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 systems calls listed in `inc/syscall.h` by invoking the corresponding kernel function for each call. You may also find it helpful to read `inc/syscall.h`.
>
> 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.

> *Challenge!* Implement system calls using the `sysenter` and `sysexit` instructions instead of using `int 0x30` and `iret`.
>
> The `sysenter/sysexit` instructions were designed by Intel to be faster than `int/iret`. They do this by using registers instead of the stack and by making assumptions about how the segmentation registers are used. The exact details of these instructions can be found in Volume 2B of the Intel reference manuals.
>
> The easiest way to add support for these instructions in JOS is to add a `sysenter_handler` in `kern/trapentry.S` that saves enough information about the user environment to return to it, sets up the kernel environment, pushes the arguments to `syscall()` and calls `syscall()` directly. Once `syscall()` returns, set everything up for and execute the `sysexit` instruction. You will also need to add code to `kern/init.c` to set up the necessary model specific registers (MSRs). Section 6.1.2 in Volume 2 of the AMD Architecture Programmer's Manual and the reference on SYSENTER in Volume 2B of the Intel reference manuals give good descriptions of the relevant MSRs. You can find an implementation of `wrmsr` to add to `inc/x86.h` for writing to these MSRs here.
>
> Finally, `lib/syscall.c` must be changed to support making a system call with `sysenter`. Here is a possible register layout for the `sysenter` instruction:
>
> ```
> eax                - syscall number
> edx, ecx, ebx, edi - arg1, arg2, arg3, arg4
> esi                - return pc
> ebp                - return esp
> esp                - trashed by sysenter
> ```
>
> GCC's inline assembler will automatically save registers that you tell it to load values directly into. Don't forget to either save (push) and restore (pop) other registers that you clobber, or tell the inline assembler that you're clobbering them. The inline assembler doesn't support saving `%ebp`, so you will need to add code to save and restore it yourself. The return address can be put into `%esi` by using an instruction like `leal after_sysenter_label, %%esi`.
>
> Note that this only supports 4 arguments, so you will need to leave the old method of doing system calls around to support 5 argument system calls. Furthermore, because this fast path doesn't update the current environment's trap frame, it won't be suitable for some of the system calls we add in later labs.
>
> You may have to revisit your code once we enable asynchronous interrupts in the next lab. Specifically, you'll need to enable interrupts when returning to the user process, which `sysexit` doesn't do for you.

User-mode startup

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.

Page faults and memory protection

Memory protection is a crucial feature of an operating system, ensuring that bugs in one program cannot corrupt other programs or corrupt the operating system itself.

Operating systems usually rely on hardware support to implement memory protection. The OS keeps the hardware informed about which virtual addresses are valid and which are not. 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 can fix it and let the program continue running. If the fault is not fixable, then the program cannot continue, since it will never get past the instruction causing the fault.

As an example of a fixable fault, consider an automatically extended stack. In many systems the kernel initially allocates a single stack page, and then if a program faults accessing pages further down the stack, the kernel will allocate those pages automatically and let the program continue. By doing this, the kernel only allocates as much stack memory as the program needs, but the program can work under the illusion that it has an arbitrarily large stack.

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 of these reasons the kernel must be extremely careful when handling pointers presented by user programs.

You will now solve these two problems with a single mechanism that scrutinizes all pointers passed from userspace into the kernel. 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. If the kernel does page fault, it should panic and terminate.

> **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.

Note that the same mechanism you just implemented also works for malicious user applications (such as `user/evilhello`).

> **Exercise 10.** 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
> ```

This completes the lab. Make sure you pass all of the **`make grade`** tests and don't forget to write up your answers to the questions and a description of your challenge exercise solution in `lab3-questionary.txt`. Commit your changes and type **`make handin`** in the `lab` directory to submit your work.

Before handing in, use **`git status`** and **`git diff`** to examine your changes and don't forget to **`git add lab3-questionary.txt`**. When you're ready, commit your changes with **`git commit -am 'my solutions to lab 3'`**, then **`make handin`** and follow the directions.