6.828 Fall 2003 Lab 3

Handed out Wednesday, October 1, 2003
Due Thursday, October 16, 2003

1. Introduction

In this lab, you will also write the code for environment creation and scheduling. In order to pre-empt running environments, you'll be required to support clock interrupts.

In this lab, the terms environment and process are interchangeable--they have roughly the same meaning. We introduce the term environment to stress the point that environments do not provide the same semantics as UNIX processes.

2. Background: Environments

This section will give you an overview of the environment code. You should consider this a specification which you'll implement for exercise 1.

Env management

In lab 2, you allocated memory for the `envs[]` array. This NENV sized array holds the state of all the possible environments. This means your OS cannot run more than NENV concurrent environments. If an attempt is made to create more than NENV environments, the system will return an `E_NO_FREE_ENV` error.

Typically, your OS will be running many fewer environments. The remaining environments should be inserted on to the `env_free_list`. This supports efficient allocation and deallocation of environments, as they just have to be added to/removed from the free list.

The kernel also maintains `curenv`, which points to the currently executing environment. During boot up, before the first environment is run, this pointer is set to `NULL`.

From env.c:

```
struct Env *envs = NULL;              /* All environments */
struct Env *curenv = NULL;             /* the current env */
static struct Env_list env_free_list; /* Free list */
```

Env state

The figure below shows the state kept by the kernel for each environment.

- **env_tf** -- the register values for the environment (see `trap.h`). The kernel saves these when switching from user to kernel mode, so that the environment can be resumed.
- **env_link** -- pointer which allows the env to be placed on the `env_free_list`. (see `queue.h` for details)
- **env_id** -- unique identifier
- **env_parent_id** -- env_id of the environment which created this environment.
- **env_status** -- one of `ENV_FREE, ENV_RUNNABLE, ENV_NOT_RUNNABLE`
- **env_pgdir** -- virtual address of this env's page directory
- **env_cr3** -- physical address of this env's page directory
- **remaining fields used in future labs**

From env.h:

```
struct Env {
  struct Trapframe env_tf;
  LIST_ENTRY(Env) env_link;
  u_int env_id;
  u_int env_parent_id;
  u_int env_status;
  Pde  *env_pgdir;
  u_int env_cr3;

  /* (below here: not used in lab 3) */
  u_int env_ipc_value;            /* IPC state */
  u_int env_ipc_from;
  u_int env_ipc_blocked;

  u_int env_pgfault_handler;      /* user page fault handler */
};
```

Like a process, an environment couples the concepts of thread and address space. The thread is defined by the registers (the `env_tf` field), and the address space is defined by the linear address mapping in `env_pgdir` and `env_cr3`. To run an environment, the kernel must restore both the registers and the address space.

Our `struct Env` is analogous to `struct user` in V6 UNIX. The key difference between the two is that `struct Env` holds the environment's (i.e., process's) user-mode register state within the `env_tf` substructure. In V6, this saved user-mode register state is stored on the process's kernel stack whenever the CPU is executing kernel code. While in V6 UNIX each process's "user segment" starts with a `struct user`, the process's kernel stack is not part of `struct user` itself but is located immediately *above* this structure.

In our x86-based operating system, individual environments do not have their own kernel stacks; instead, all kernel code runs on a single kernel stack and the kernel saves user-mode register state explicitly in each environment's `struct Env` rather than implicitly on the kernel stack.

3. Background: Interrupts and Exceptions

The follow sections highlight a few key concepts and mechanisms. At first read, the connection between them might not be clear. However, the coding exercises should show you how all the pieces fit together.

Terminology

This lab adopts the terminology defined in IA-32 Intel Architecture Software Developer's Manual, Volume 3: System programming guide. However, be aware that terms such as exceptions, traps, interrupts, faults and aborts have no standardized meaning. When you see these terms outside of this lab, the meanings might be slightly different.

You should read chapter 5 of the above reference (sections 5.7, 5.8.2, 5.8.3, 5.9 and 5.12.2 are not particularly relevant).

Interrupt discipline

In your operating system, you will make a key simplification compared to V6 UNIX. External (ie. device) interrupts are always disabled when in the kernel and always enabled when in user space. External interrupts are controlled by the `FL_IF (see mmu.h)` bit of the `%eflags` register. When this bit is set, external interrupts are enabled. While the bit can be modified in several ways, because of our simplification, we will handle it solely through the `%eflags` register.

You will have to ensure that the `FL_IF` flag is set in user processes when they run so that when an interrupt arrives, it gets passed through to the processor and handled by your interrupt code. Otherwise, it will be masked: this is the case when the processor is reset.

Interrupt Mapping

The interrupt descriptor table (IDT) tells the processor how to handle each possible interrupt. The internal processor exceptions map to IDT entries 0-31. For example, the information about the page fault handler is in entry 14. When a page fault occurs, the processor transfers control to the `CS:EIP` defined in `IDT[14]`. These values define the address of the kernel's page fault handler routine.

External interrupts (i.e., device interrupts) are refered as IRQs. There are 16 possible IRQs, numbered 0 through 15. The mapping from IRQ number to IDT entry is not fixed. `Pic_init` in `picirq.c` maps IRQs 0-15 to interrupts `IRQ_OFFSET` through `IRQ_OFFSET+15`.

In picirq.h `IRQ_OFFSET` is defined to be decimal 32. Thus the IDT entries 32-44 correspond to the IRQs 0-15. For example, the clock interrupt is IRQ 0. Thus, IDT[32] contains the address of the clock's interrupt handler routine.

The `IRQ_OFFSET` is chosen so that the device interrupts don't overlap with the processor exceptions.

Stack switching -- the TSS

When kernel executes an instruction which generates an exception (e.g., the kernel divides by zero, deferences a NULL pointer, etc), the processor pushes exception parameters on the current stack. If there was ever not enough space on the stack, the CPU would reset itself.

When a user process is executing and an exception/interrupt occurs, the processor does not push the parameters on the current stack, but rather it switches to the stack defined by the SS0 and ESP0 fields of the TSS (see `idt_init()` in `trap.c`). The OS should always guarantee that these are valid addresses, so that the exception can be handled.

An Example

Let's put these pieces together and trace through an example. Let's say a user process is running in a loop, when a clock interrupt occurs.

1. The processor switches to the stack defined by the TSS SS0 (GD_KD) and ESP0 (KSTACKTOP).
2. The processor pushes the exception parameters on the kernel stack, starting at addresss 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 IRQ 0, the clock interrupt, the processor reads IDT entry IRQ_OFFSET+0 andsets `%cs:%eip` to the handler function defined there. to the handler function defined there.
4. The handler function takes control and can handle the exception/interrupt.

4. Getting Started

Download the reference code lab3.tar.gz, located in the labs section for this course, and untar it into your 6.828 directory as before. You will then need to merge the changes between our lab 2 and lab 3 source code trees into your own kernel code resulting from completing lab 2. As we mentioned before, the `diff` and `patch` utilities can be very useful for this purpose, as well as their "big brother" `cvs`.

Lab 3 contains the following new source files, which you should browse through as your merge them into your kernel:

- `kern/env.c`
- `kern/sched.c`
- `kern/sched.h`
- `kern/trap.c`
- `kern/trap.h`
- `kern/picirq.c`
- `kern/lab3.S`

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

```
$ diff -ur lab2 lab3
```

You can try using the `patch` command to incorporate these changes into your source tree automatically. But *back up your source tree* before you try this!!! The `patch` command usually does a fairly good job but is not infallible; it can occasionally insert patches in the wrong place. Be sure you check the resulting source files to make sure they look right. Additionally, if one of our changes between lab2 and lab3 is textually overlapping or just too close in a single source file to one of your changes, `patch` will report a conflict and write the (unmerged) patch to a separate file in the directory, leaving you to merge that change manually. For example:

```
$ diff -ur lab2 lab3 >lab2-3.pch
$ cp -r mysrc mysrc-backup
$ cd mysrc
$ patch -p1 <../lab2-3.pch
```

Debugging tips

For all its faults, `bochs` is still a much more hospitable debugging environment than a real processor. Put it to work for you!

- If you include `kern/bochs.h`, then you can call the pseudo-function `bochs()` to cause Bochs to stop executing your kernel and return to the debug prompt. This is useful for setting long-lived "breakpoints", or for stopping your program after a tricky section of code so that you can inspect its work before letting it continue.
- The command `info idt` will print the current IDT. This is useful for checking whether you set it up correctly.

- The `vb` command sets a breakpoint at a particular %cs:%eip address. Since the kernel code segment selector is 8, `vb 8:0xf0101234` sets a breakpoint at the given kernel address. Similarly, since the user segment selector is 0x1b, `vb 0x1b:0x80020` sets a breakpoint at the given user address.

Finally, note that passing all the `gmake grade` tests does not mean your code is perfect. It may have subtle bugs that will only be tickled by future labs. In a perfect world, `gmake grade` would find all your bugs, but no one builds operating systems in a perfect world anyway. Keep in mind that debugging an operating system is a very holistic task -- there are abstraction boundaries, but you can't necessarily place much trust in them since nothing is really enforcing them. If you get all sorts of weird crashes that don't seem to be explainable by a single bug in the layer you're working on, it's likely that they're explainable by a single bug in a different layer.

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. For the "definitive" reference to GCC inline assembly language, type `info gcc`, select the "C Extensions" chapter, and then the "Extended Asm" section. Other links you might find useful:

- Brennan's Guide to Inline Assembly
- x86 Inline Assembly Programming
- Inline assembly for x86 in Linux

5. Hand-in Procedure

As before, you can test your code against our test scripts by running `gmake grade`. When you are done, run `gmake handin` to tar up and hand in your source tree. You should get an automatic e-mail confirmation shortly thereafter.

6. Exercises

Exercise 1: Creating and Running Environments

In this exercise you will write the code necessary to run a user process (i.e., a `struct Env`). Because we do not yet have a filesystem, we will set up the kernel to load a static binary image much like V6 does with `icode`. However, there are some functions that will need to be completed before this process can run.

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

```
env_init()
env_setup_vm()
env_alloc() -- you just need to set the tf_eip field
load_icode()
env_create()
env_run()
```

As you write these, you might find the new printf 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".

Once you are done you should compile your kernel and run it under Bochs (remember that `bochs-nogui` doesn't require X11). You should see:

```
...
  Setup timer interrupts via 8259A
  unmasked timer interrupt
```

If you see this type Control-c into the Bochs debugger, then single step a couple of time. What you should see is listed below. Note that the t=??????? values will be different for you.

```
Next at t=0
(0) f000:fff0: e968e0: jmp +#e068
 c
Next at t=9355127
(0) 001b:00800020 (unknown context): ebfe: jmp +#fe
 s
Next at t=9355128
(0) 001b:00800020 (unknown context): ebfe: jmp +#fe

Next at t=9355129
(0) 001b:00800020 (unknown context): ebfe: jmp +#fe

Next at t=9355130
(0) 001b:00800020 (unknown context): ebfe: jmp +#fe
```

If you don't see this output, then you've made a coding mistake. Fix it before continuing.

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` (`locore.S`)
- `i386_init`
    - `cons_init`
    - `i386_detect_memory`
    - `i386_vm_init`

- `ppage_init`
- `idt_init`
- `pic_init`
- `kclock_init`
- `env_init`
- `env_create`
- `sched_yield`
  - `env_run`
    - `env_pop_tf`

Exercise 2: Clock Interrupts

In exercise 1, after the environment was started (with `env_run()`), it just spun in a loop. The kernel never got control back. We need to generate and handle clock interrupts. Interrupts will force control back to the kernel. In a later lab, we will also add system calls, with which a user program can ask the processor to transfer control to the kernel.

The calls to `pic_init` and `kclock_init` (from `i386_init` in `init.c`) set up the clock and the interrupt controller to generate interrupts. However, the user env is not interrupted because the CPU is ignoring device interrupts. You need to correct this. Modify the code in `env_alloc()` so that user environments are run with interrupts enabled.

You might want to re-read sections 5.8 and 5.8.1 of IA-32 Intel Architecture Software Developer's Manual, Volume 3: System programming guide at this time.

Next, you need to setup the IDT and interrupt handler. The handler has been written for you (see `clock_interrupt` in `locore.S`. You should edit `idt_init` in `trap.c` at this time. Use the `SETGATE` macro to set the IDT entry corresponding to the clock interrupt. Note that in order to obtain a useful C symbol, you can add the declaration `extern void clock_interrupt(void);` to your program.

Once you are done coding, compile and run your kernel. If you see asterisks print out one after another you can continue.

Make sure you can answer the following questions:

1. How many instruction of user code are executed between each interrupt?
2. How many instruction of kernel code are executed to handle the interrupt?

   Hint: use the `vb` command mentioned earlier.

   When using objdump to dump the kernel pass `--adjust-vma=0xf00ff000`. *It's not clear why this is required, `--adjust-vma=0xf0100020` would seem more logical, but nevertheless it is.*

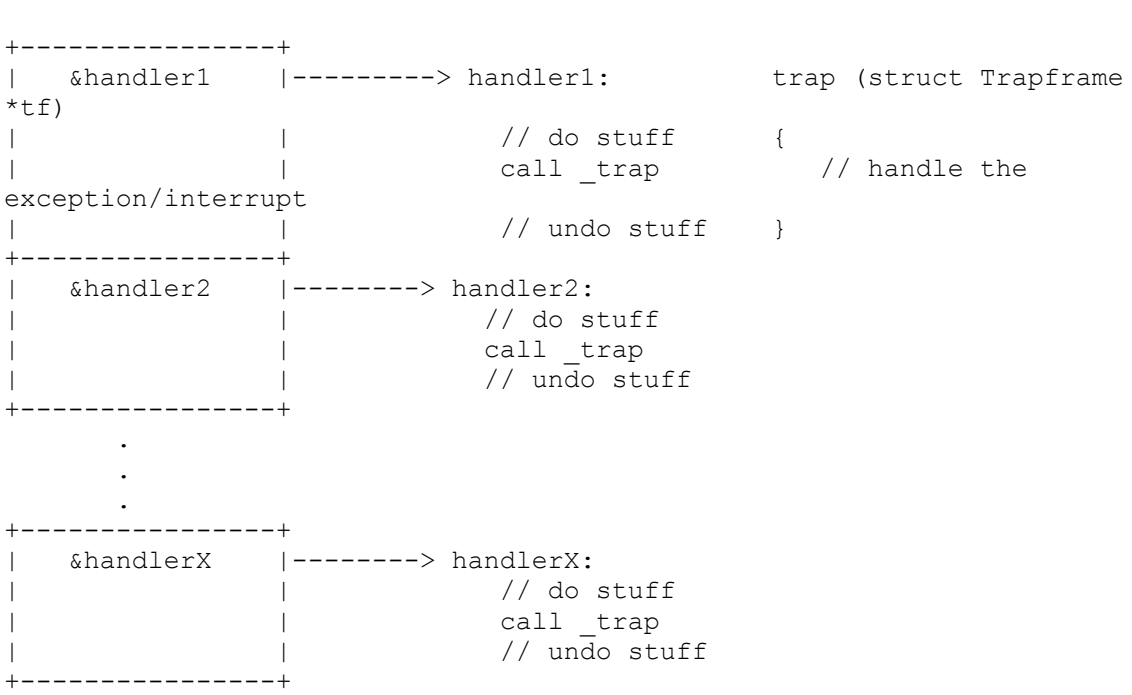
Exercise 3: Generalized interrupt/exception handling

In this exercise, you will set up the IDT to handle all the interrupts and exceptions that we expect to see. Although 256 interrupts/exceptions are possible, your code only needs to handle interrupts 0-31 (the processor exceptions) and interrupts 32-47 (the device IRQs). We may add additional interrupts later.

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

Note: Some of the exceptions in the range 0-31 are defined by Intel to be reserved. Since they will never be generated, 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:



Each exception or interrupt has its own handler in `locore.S` and the IDT is initialized with the address of these handlers. Each of the handlers should build a `struct Trapframe` (see `trap.h`) on the stack and call into `trap()` (in `trap.c`) with a pointer to the Trapframe.

Even the clock interrupt should fit into this model. This means you will no longer be using the `clock_interrupt` handler.

After control is passed to `trap()`, that function handles the exception/interrupt or dispatches the exception/interrupt to a specific handler function.

1. Edit `locore.S` and `trap.c` and implement what has been described above. The macros `IDTFNC` and `IDTFNC_NOEC` in `locore.S` should help you, as well as the T_* defines in `trap.h`.

   You can use the code from `_clock_interrupt` as a starting point, but remember that the code must be modified to build a `struct Trapframe`.

   Hint: you code should perform the following steps:

   1. push values on the stack in the order defined by struct Trapframe
   2. load GD_KD into %ds and %es
   3. pushl %esp -- pass pointer to Trapframe which is built on the stack
   4. call _trap
   5. pop the values pushed in steps 1-3
   6. iret

   Consider using the `pushal` and `popal` instructions; they fit nicely with the layout of the `struct Trapframe`.

2. In the `trap` function, dispatch clock interrupts to the `clock` function. Run your kernel. You should see the asterisks, as you did in the previous exercise. Do not continue until you see this.

   If you get stuck, remember the debugging tips given earlier.

3. The trap handling is going to help you clean up after errant user envs. `lab3.S` contains a number of test cases which simulate what an errant user env might execute.

   To configure your code for a test case edit the following line from `i386_init`:

   ```
   env_create(&spin_start, &spin_end - &spin_start);
   ```

   Replace `spin_start` and `spin_end` with the analogous names for the test case. You'll also need to declare the symbols for the test case, of course.

   Then, recompile and boot your kernel. In trap(), you should call `env_destroy()` to cleanup after the errant environment. You should expect the following assert in `yield` to fire:

```
assert(envs[0].env_status == ENV_RUNNABLE);
```

Make sure your kernel passes all of the test cases.

Make sure you can answer these questions:

1. How do you know when you pass a test case? What output did you see and why?
2. The break point test case will either generate a break point exception of a general protect fault depending on how you initialized the break point entry in the idt (i.e., your call to `SETGATE` from `idt_init`). Which did you see? Change your code to generate the other.
3. What do you think is the point of these mechanisms? (hint: consider what would happen if the user code executed "int $0x20")
4. 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 functionality that exists in the current implementation could not be provided?)

Challenge!!! You probably have a lot of very similar code right now, between the lists of `IDTFNC` in `locore.S` and their installations in `trap.c`. Clean this up. Change the macros in `locore.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`.)

Exercise 4: Pre-emptive Mulitasking

In this exercise you'll create a second env and use the clock interrupt to switch between the two envs.

1. Edit `i386_init()`. Replace the line that says:

   ```
   env_create(&spin_start, &spin_end - &spin_start);
   ```

   with

   ```
   env_create(&alice_start, &alice_end - &alice_start);
   env_create(&bob_start, &bob_end - &bob_start);
   ```

   (Again, you will need to declare the appropriate symbols.) Now, `envs[0]` will correspond to the "alice" code and `envs[1]` to the "bob" code.

2. Edit `trap()` so that when a clock interrrupt occurs, `sched_yield()` is called.
3. Open `sched.c` and rewrite the function `sched_yield()`. You should rewrite it to be a round-robin scheduler, which loops over the envs in `envs[]` calling

`env_run()` on an env if its status is ENV_RUNNABLE. Be sure that you don't create starvation problems.

4. The last step to making this work requires a small modification to env_run(). If you are switching from one env to another, you need to save the register state of the current environment before running the new env. Add the following to the begining of env_run().

   ```
   if (curenv)
       /* FILL IN ONE LINE OF CODE HERE */
   ```

   Hint: the one line of code uses `curenv->env_tf` and `KSTACKTOP`.

   As an aid to you, "alice" and "bob" check that their registers are saved/restored correctly.

Ensure that your kernel continually switches between the two envs. In env_run(), you might want to print out the env's `envid` when it is run.

Make sure you can answer these questions:

1. Examine the code to "alice" and "bob" in `lab3.S`. What is the purpose of "alice" and "bob" code?
2. If two copies of "alice" were run (instead of running "alice" and "bob"), what mistake in the register saving/restoring code would go undetected?
3. In your implementation of `env_run()` you should have called `lcr3()`. But before and after the call to `lcr3()`, your code makes references (at least it should) to the variable `e`--the argument to `env_run`. How can this work?

   Upon loading the `%cr3` register, the addressing context used by the MMU is instantly changed. The problem is that a virtual address (namely `e`) has meaning relative to a given address context--the address context specifies the physical address to which the virtual address map. Why can the pointer `e` be referenced both before and after the addressing switch?

This completes the lab.

Version: $Revision: 1.6 $. Last modified: $Date: 2003/10/07 19:23:39 $