Project 1: Initializing PC Hardware

Due Thursday, February 14, 2013

Introduction

In this project, you will start building a kernel called PIOS, for *Parallel Instructional Operating System*. PIOS uses some of the same source code as xv6, and is also derived from JOS, the 6.828 kernel, but is structured quite differently from either xv6 or JOS. Like xv6, PIOS supports multiprocessor and multicore systems. You will learn more about the design of PIOS gradually as you work through this course.

This first project focuses on getting a skeleton PIOS kernel up and running to a point where it provides a basic, functional kernel development environment. The project contains four parts:

1. Making the kernel produce useful output: i.e., printing to the console. Most of this somewhat boring I/O code is done for you; you will just have to fill in a couple **short** missing pieces, and sometimes **correct** a couple of lines with error.
2. Setting up the processor so that any traps the kernel causes, such as an illegal instruction or memory accesses (usually indicating a bug in the kernel) will be caught by the kernel and yield a useful "panic" message instead of just causing the machine to reboot mysteriously.
3. Implement *protected control transfer*, enabling the kernel to run application code in a less privileged mode and safely recover control when the application makes a system call or a trap occurs during its execution.
4. Determining the amount of physical memory available and implementing a dynamic memory allocator for it, so that code in future projects will be able to allocate and free memory.

Software Setup

The files you will need for this and subsequent project assignments in this course are available on SOCS. In the following projects, we will use GIT (it is a version control system) to manage different versions of our projects. We will discuss GIT with more details in project 2. For this project, you do not need to use GIT.

```
$ cd ~/os
```

```
$ mkdir pios
$ cd pios
$
```

Please download project1.tar to directory ~/os/pios. Save the file as project1.tar.

```
$ tar –xvf project1.tar
```

```
$ cd project1
```

The same compilers and simulators you are already using to explore xv6 will work for the projects. Once you have the extract the file from SOCS, you can build the PIOS skeleton by typing `make`, and you can run or debug it under QEMU by typing `make qemu` or `make qemu-gdb`, respectively, just as with xv6. The skeleton won't get very far, of course: it will quickly panic in one of the functions you will need to implement.

Inline Assembly

In this project you may find GCC's inline assembly language feature useful, although it is also possible to complete the project 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 SOCS.

Hand-In Procedure

Tar your lab1 directory into file lab1.tar. Submit this file to SOCS under project 1 category. You do not need to turn in answers to any of the questions in the text of the project. (Do answer them for yourself though! They will help with the rest of the project.)

I will be grading your solutions with a grading program. You can run `make grade` to test your solutions with the grading program.

Part 1: Printing to the Console

We will now start to examine the minimal PIOS kernel in a bit more detail. PIOS uses essentially the same boot loader code as xv6, which you examined in the lecture homeworks, so we will skip that part. Once the boot loader loads the kernel's ELF image, it jumps to the kernel entrypoint, namely the `start` label in `kern/entry.s`. This assembly language entrypoint code just performs basic

initialization and then calls the C function `init()` in `kern/init.c`. After zeroing out the kernel's uninitialized data area (which is expected by C program loading conventions but the boot loader didn't bother to do when it was loading the kernel), the first thing the kernel does is initialize the console device driver so that your kernel can produce visible output.

Formatted Printing to the Console

Most people take functions like `printf()` for granted, sometimes even thinking of them as "primitives" of the C language. But in an OS kernel, we have to implement all I/O ourselves.

Read through `kern/cons.c`, `lib/cprintf.c`, `lib/printfmt.c`, `dev/kbd.c`, and `dev/video.c`, and make sure you understand their relationship. It will become clear in later projects why `printfmt.c` and `cprintf.c` are located in the separate `lib` directory.

Exercise 1. We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment.

Be able to answer the following questions:

1. Explain the interface between `vcprintf()` in `lib/cprintf.c` and `vprintfmt()` in `lib/printfmt.c`. Specifically, how does `vcprintf()` tell `vprintfmt()` how to display a character?
2. Explain the following from `dev/video.c`:

```
1       if (crt_pos >= CRT_SIZE) {
2               int i;
3               memcpy(crt_buf, crt_buf + CRT_COLS,
                  (CRT_SIZE - CRT_COLS) *sizeof(uint16_t));
4               for (i = CRT_SIZE - CRT_COLS;i<CRT_SIZE; i++)
5                   crt_buf[i] = 0x0700 | ' ';
6                   crt_pos -= CRT_COLS;
7       }
```

3. For the following questions you might wish to consult the lecture slides. These slides cover GCC's calling convention on the x86.

   Trace the execution of the following code step-by-step:

```
int x = 1, y = 3, z = 4;
cprintf("x %d, y %x, z %d\n", x, y, z);
```

- In the call to `cprintf()`, to what does `fmt` point? To what does `ap` point?
- List (in order of execution) each call to `cons_putc`, `va_arg`, and `vcprintf`. For `cons_putc`, list its argument as well. For `va_arg`, list what `ap` points to before and after the call. For`vcprintf` list the values of its two arguments.

4. Run the following code.

```
unsigned int i = 0x00646c72;
cprintf("H%x Wo%s", 57616, &i);
```

   What is the output? Explain how this output is arrived at in the step-by-step manner of the previous exercise. Here's an ASCII table that maps bytes to characters.

   The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set `i` to in order to yield the same output? Would you need to change `57616` to a different value?

   Here's a description of little- and big-endian and a more whimsical description.

5. In the following code, what is going to be printed after `'y='`? (note: the answer is not a specific value.) Why does this happen?

```
cprintf("x=%d y=%d", 3);
```

6. Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change `cprintf` or its interface so that it would still be possible to pass it a variable number of arguments?

The Stack

We will now explore in more detail the way the C language uses the stack on the x86, and in the process write a useful function that computes a *backtrace* of the stack: a list of the saved Instruction Pointer (**IP**) values from the nested `call` instructions that led to the current point of execution.

Exercise 2. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

The x86 stack pointer (`esp` register) points to the lowest location on the stack that is currently in use. Everything *below* that location in the region reserved for the stack is free. Pushing a value onto the stack involves decreasing the stack pointer and then writing the value to the place the stack pointer points to. Popping a value from the stack involves reading the value the stack pointer points to and then increasing the stack pointer. In 32-bit mode, the stack can only hold 32-bit values, and esp is always divisible by four. Various x86 instructions, such as `call`, are "hard-wired" to use the stack pointer register.

The `ebp` (base pointer) register, in contrast, is associated with the stack primarily by software convention. On entry to a C function, the function's *prologue* code normally saves the previous function's base pointer by pushing it onto the stack, and then copies the current `esp` value into `ebp` for the duration of the function. If all the functions in a program obey this convention, then at any given point during the program's execution, it is possible to trace back through the stack by following the chain of saved `ebp` pointers and determining exactly what nested sequence of function calls caused this particular point in the program to be reached. This capability can be particularly useful, for example, when a particular function causes an `assert` failure or `panic` because bad arguments were passed to it, but you aren't sure *who* passed the bad arguments. A stack backtrace lets you find the offending function.

Exercise 3. To become familiar with the C calling conventions on the x86, set a breakpoint at `debug_trace` in `kern/debug.c`, and examine the state of the stack at the point it gets called. Get GDB to give you a backtrace via the **`bt`** command, compare GDB's backtrace with the "raw" contents of the stack, and make sure you understand how GDB got that backtrace. How many 32-bit words does each nested function push on the stack, and what are those words?

The above exercise should give you the information you need to implement `debug_trace()`, which collects into an array the EIPs from up to `DEBUG_TRACEFRAMES` (10) stack frames starting at the EBP specified in the argument. This function will be used in multiple ways in the kernel, helping to provide meaningful context information for debugging when something goes seriously wrong with the system.

Note that if there are fewer than 10 frames on the stack, `debug_trace` should set the rest of the EIPs to NULL. By studying `kern/entry.s` you'll find that there is an easy way to tell when to stop.

Besides collecting the EIPs in the array passed by the caller, you may find it useful (at least for the moment) to make `debug_trace` print not only the EIP from each

frame but the frame pointer itself and a few of each function's arguments. For example:

```
Stack backtrace:
ebp f0109e58  eip f0100a62  args 00000001 f0109e80 f0109e98 f0100ed2
00000031
ebp f0109ed8  eip f01000d6  args 00000000 00000000 f0100058 f0109f28
00000061   ...
```

Within each line of this example, the `ebp` value indicates the base pointer into the stack used by that function: i.e., the position of the stack pointer just after the function was entered and the function prologue code set up the base pointer. The listed `eip` value is the function's return instruction pointer, where control will return when the function returns. The return instruction pointer typically points to the instruction after the `call` instruction (why?). Finally, the five hex values listed after `args` are the first five arguments to the function in question, which would have been pushed on the stack just before the function was called. If the function was called with fewer than five arguments, of course, then not all five of these values will be useful. (Why can't the backtrace code detect how many arguments there actually are? How could this limitation be fixed?)

Here are a few specific points you read about in K&R Chapter 5 that are worth remembering for the following exercise and for future projects.

- If `int *p = (int*)100`, then `(int)p + 1` and `(int)(p + 1)` are different numbers: the first is `101` but the second is `104`. When adding an integer to a pointer, as in the second case, the integer is implicitly multiplied by the size of the object the pointer points to.
- `p[i]` is defined to be the same as `*(p+i)`, referring to the i'th object in the memory pointed to by p. The above rule for addition helps this definition work when the objects are larger than one byte.
- `&p[i]` is the same as `(p+i)`, yielding the address of the i'th object in the memory pointed to by p.

Although most C programs never need to cast between pointers and integers, operating systems frequently do. Whenever you see an addition involving a memory address, ask yourself whether it is an integer addition or pointer addition and make sure the value being added is appropriately multiplied or not.

Exercise 4. Implement the `debug_trace` function as specified above. When you think you have it working right, run **`make grade`** to see if the checking code agrees, and fix it if it doesn't.

Part 2: Trap Handling

One of the most important functions of any OS kernel is to handle processor traps that occur during execution of either its own code or application code. For this part of the project, the first thing you should do is thoroughly familiarize yourself with the x86 interrupt and exception mechanism.

Exercise 5. Read Chapter 5, Interrupt and Exception Handling, in the IA-32 System Programming Guide, if you haven't already.

There are many types and classifications of traps, and the terminology used to describe them is unfortunately have no fully standard meanings, even within a single processor architecture such as the x86. We will distinguish between two main classes of traps: *exceptions* such as illegal instruction or divide by zero, whose cause is directly related to the code the processor is running; and *interrupts*, which are generated asynchronously by external I/O devices or other CPUs and whose cause may have nothing to do with the code that happens to be running at the time the interrupt arrives. We will generally use the term *trap* as a generic term referring to both exceptions and interrupts, but note that the Intel manuals uses the term differently, to refer to a specific subcategory of exceptions. When you see these terms outside of this project, the meanings may differ further.

This part of the project focuses on handling traps that occur while the processor is running in the x86 processor's most privileged mode, ring 0, which by convention we call "kernel mode". The next part of the project will use traps to implement *protected control transfer*, allowing the kernel to enter user mode (ring 3) and later recover control safely from traps that occur in user mode.

Handling Kernel Traps

Like any nontrivial software, most OS kernels have bugs - especially during development, but typically even after deployment. And kernels typically have no "guard-rails" to keep their bugs contained: an application bug typically only causes that application to crash because the kernel is there to protect applications from each other, but a kernel bug typically causes the whole machine to crash because there is no other software "under" the kernel protecting the kernel from itself. (The obvious exception is when running the kernel in a virtual machine of some kind, such as QEMU, where QEMU prevents guest kernel bugs from escaping the virtual machine.)

When a kernel bug does manifest itself, there is no guarantee anything can be done - the bug could have caused the system to go into practically any state, from which no recovery of any kind might be possible. But often the bug causes the processor to take a trap at some point, and it may be possible to recover from the bug at least enough to collect and output some potentially useful information about what happened. But the processor does not know how to generate a "kernel bug report": it only knows how to take a trap; the kernel itself must *handle* the trap and produce the bug report. We will now create a trap handler for PIOS to help catch and report on such bugs. (You will later extend this same trap handling mechanism to handle protected control transfer, virtual memory faults, and external device interrupts).

Every widespread operating system seems to develop an entire subculture surrounding its kernel trap handling mechanism: this is what a "blue screen of death" (Windows), a "Sad Mac", or a "guru meditation" (Amiga) is. Now it's time to implement your own screen of death.

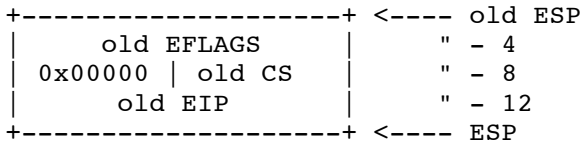
The Interrupt Descriptor Table

The x86 processor uses a table known as the *interrupt descriptor table* (IDT) to determine how to transfer control when a trap occurs. 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 256. 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 IDT, which the kernel sets up in kernel-private memory of the kernel's choosing, much like the GDT. From the appropriate entry in this table the processor loads:

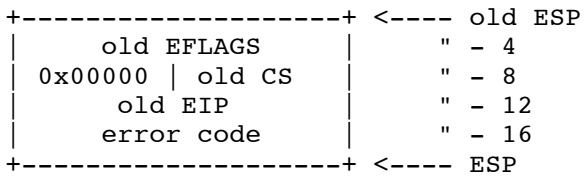
- the value to load into the instruction pointer (`EIP`) register, pointing to the kernel code designated to handle that type of exception.
- the value to load into the code segment (`cs`) register, which includes in bits 0-1 the privilege level at which the exception handler is to run. In PIOS, all exceptions are handled in kernel mode, privilege level 0.

Entering and Returning from Trap Handlers

When an x86 processor takes a trap while in kernel mode, it first pushes a *trap frame* onto the kernel stack, to save the old values of certain registers before the trap handling mechanism modifies them. The processor then looks up the CS and EIP of the trap handler in the IDT, and transfers control to that instruction address. The following diagram illustrates the format of the basic kernel trap frame, defining the state of the kernel stack on entry to the trap handler:



For certain types of x86 exceptions, in addition to the basic three 32-bit 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 x86 manuals 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 trap handler:



The x86 processor provides a special instruction, `iret`, to return from trap handlers. It expects the kernel's stack to look like the *first* figure above, with ESP pointing to the old EIP. When the processor executes an `iret` instruction, pops the saved values of EIP, CS, and EFLAGS off the stack and back into the corresponding registers, and resumes instruction execution at the popped EIP.

Note that when returning from a trap, the processor doesn't actually know or care whether the "old" values it is popping off the stack are really the exact same values that it originally pushed onto the stack on entry to the trap handler. Think about what would happen - for better or worse - if the kernel trap handler changes these values during its execution.

Nested Traps

Because the processor handles a trap by pushing the old EIP (and some other state) on the stack and then transferring control to a designated instruction address, trap handling in the x86 works very analogously to procedure calls, and like ordinary procedure calls, traps can be nested. That is, if the processor is already in the process of handling one trap, and the trap handler itself causes an exception or is interrupted by an external device, then the processor merely pushes another trap frame onto the stack and transfers control to the appropriate trap handler. The processor itself doesn't maintain any state reflecting the fact that the second trap handler is nested inside the first: as far as the processor is concerned it is simply pushing a standard frame onto the stack and transferring control in a standard way whenever a trap occurs. All of the state of both interrupted activities remains on

the stack, and can later be "unwound" by returning from the two trap handlers in reverse order, again analogously to ordinary procedure returns.

Nested trap handling is often extremely important in kernel development, both for its utility and its danger: it is often highly useful to allow one trap handler to be interrupted by another, but *unintentional* nesting can cause extremely subtle and difficult bugs. There is also one important caveat to the processor's nested trap handling capability. If the processor takes a trap 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.

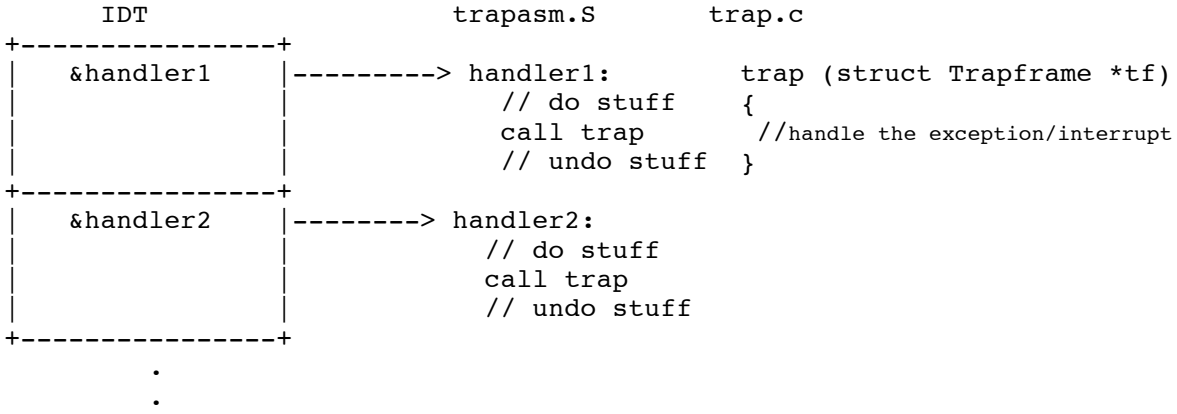
Setting Up the IDT

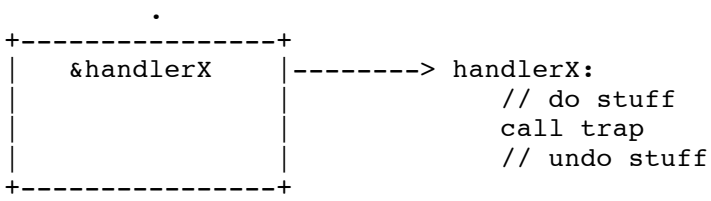
You should now have the basic information you need in order to set up the IDT and handle exceptions in PIOS. For now, you will set up the IDT to handle interrupt vectors 0-31 (the processor exceptions).

The header files `inc/trap.h` and `kern/trap.h` contain important definitions 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. The skeleton source code for PIOS's trap handling mechanism is in `kern/trap.c` (the C part) and `kern/trapasm.s` (the assembly language part).

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:





Each exception should have its own handler in `trapasm.s` and `trap_idt_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 that trapframe.

`trap()` handles the exception/interrupt or dispatches to a specific handler function. If and when `trap()` returns, the code in `trapasm.s` restores the old CPU state saved in the Trapframe and then uses the `iret` instruction to return from the exception.

Exercise 6. Edit `trapasm.s` and `trap.c` and implement the features described above. The macros `TRAPHANDLER` and `TRAPHANDLER_NOEC` in `trapasm.s` should help you, as well as the T_* defines in `inc/trap.h`. You will need to add an entry point in `trapasm.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_idt_init()` to initialize the IDT to point to each of these entry points defined in `trapasm.s`; the `SETGATE` macro will be helpful here.

Hint: your `_alltraps` should:

1. push values to make the stack look like a struct trapframe
2. load `CPU_GDT_KDATA` 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` and `popal` instructions; they fit nicely with the layout of the `struct trapframe`.

We have provided a function `trap_check()` to test your trap handling code for a variety of traps (though by no means all the exceptions the processor can generate). Make sure it reports success: You should be able to get `make grade` to succeed on the trap handler test at this point.

Returning from Traps

You will now need to implement the complement of the trap entry code: namely the code to return from a trap.

In many kernels, including xv6, the high-level trap handling code written in C returns from a trap simply by returning from the C procedure corresponding to PIOS's `trap()`. PIOS follows a different convention, however. You might notice the `gcc_noreturn` macro (see `inc/gcc.h`) in the definition of the `trap()` function, which declares that this function "never returns", at least as far as GCC is concerned. In PIOS, kernel code returns from a trap by performing *another call*, to the function `trap_return()`, which is also declared in `trap.h` never to return. See `trap_check_recover()` for an example of how it is used. Those familiar with programming language implementation will recognize this as *tail-call* style programming.

Exercise 7. Implement the `trap_return()` function in `trapasm.S`. (Alternatively, you may implement it in `kern/trap.c` using inline assembly language if you prefer.) The `trap_return()` function takes a pointer to a `trapframe` structure as an argument, and first "unwinds" the stack to the point where that `trapframe` was pushed, discarding any information that has been pushed onto the stack since that point. The code then restores all the saved register state from the `trapframe` and returns from the trap via the `iret` instruction.

We have provided a function `trap_check()` to test your trap handling code for a variety of traps (though by no means all the exceptions the processor can generate). Make sure it reports success: You should be able to get `make grade` to succeed on the trap handler test at this point.

Part 3: Privilege Levels and Protected Control Transfer

So far in PIOS we have used only *kernel mode* or *ring 0*, the processor's most privileged level. To protect the kernel from unruly applications, and in turn applications from each other, the kernel must be able to protect both its own code and data, and various sensitive parts of the processor's state, from accidental or malicious modification by application code. An x86-based kernel protects its state in memory from applications by using the processor's virtual memory facilities, which we will use later in project 3.

For now, we will focus on the other requirement: protecting sensitive processor state from application code. If any application could run a `lidt` instruction to change the IDT, for example, then the application could wrest control of trap handling away from the kernel and redirect all traps to handlers of the application's

choosing. The x86 processor provides multiple *privilege levels* or *rings* to enable operating systems to protect sensitive processor state. To protect the location of the sensitive IDT, for example, the x86 architecture restricts the use of the `lidt` instruction to code running in ring 0; the kernel can thus prevent applications from executing `lidt` and other sensitive instructions by running application code at a lower privilege level. To do so, however, the kernel must still have a protected way to get *into* and back *out of* lower privilege levels.

Exercise 8. Read Chapter 4, Protection, in the IA-32 System Programming Guide (on SOCS), if you haven't already.

Basics of Protected Control Transfer

In most processor architectures including the x86, the trap handling mechanism doubles as a *protected control transfer* mechanism, enabling the kernel to transfer control safely to and from code running at lower privilege levels. When the processor is running in user mode (ring 3 on the x86) and takes a trap or makes an explicit system call, the processor must transfer control to the kernel, but it is crucial for protection purposes that the user mode code currently running *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.** As you have already seen in the previous part, through the IDT 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. This mechanism effectively protects the instruction pointer (EIP) address used to handle traps.
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 example, one user mode thread could change the kernel state of another thread while the latter is in a system call, or user code could simply point ESP to unmapped or read-only memory, making it impossible for the processor to push the trap frame and causing an immediate reset as described above.

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, PIOS 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 PIOS 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. PIOS doesn't use any other TSS fields.

Combined, the IDT and TSS provide the kernel with a mechanism to ensure that traps are handled only by calling well-defined entrypoints in the kernel (the interrupt vectors in the IDT) and that trap handlers will have a well-defined, protected workspace (the stack pointers in the TSS). Exactly *where* these entrypoints and kernel stacks are located is up to the kernel, however.

Kernel Stack Management

A particularly important kernel design issue is *how many* kernel stacks there are, and which OS abstraction they are associated with. There are basically two common models:

- The xv6 kernel, like most Unix kernels, uses a *process model*: it associates a kernel stack with each user process, so that whenever the kernel is running on behalf of a particular process, it runs on that process's kernel stack, and it switches stacks whenever it switches between processes.
- The PIOS kernel, in contrast, is an *interrupt model* kernel, which means it maintains one kernel stack per *physical CPU*, irrespective of the number of processes. Kernel code running on a given CPU always runs on that CPU's permanently-assigned kernel stack regardless of what user process is running, and kernel code thus never switches kernel stacks once it has booted. The downside of this simple design is that PIOS kernel code cannot use its stack to maintain any state on behalf of processes that are not currently running: if the kernel is working on behalf of a process and needs to put the process to sleep waiting on some event, the kernel must explicitly store all relevant information about what it was doing somewhere else, such

as in the process control structure, or it would be lost when the kernel switches to another process.

You will learn more about process versus interrupt model kernels when we get into process management and scheduling in the next project.

For now all you really need to know is that PIOS's kernel stack is part of the `cpu` structure, defined in `kern/cpu.h`: it grows downwards (like all x86 stacks) from `kstackhi` towards`kstacklo` in the structure. Thus, the ESP field for ring 0 in a given processor's TSS needs to point to `kstackhi` in that processor's `cpu` structure.

An Example

Let's put the above pieces together and trace through an example. Suppose the processor is executing code in user mode 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 PIOS will hold the values `CPU_GDT_KDATA` and `&cpu->kstackhi`, respectively.
2. The processor pushes the following basic trap frame onto the kernel stack, starting at `kstackhi`:

```
+--------------------+ &cpu->kstackhi
| 0x00000 | old SS   |     " - 4
|      old ESP       |     " - 8
|     old EFLAGS     |     " - 12
| 0x00000 | old CS   |     " - 16
|      old EIP       |     " - 20 <---- ESP
+--------------------+  <---- 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 defined there.
4. The handler function takes control and handles the exception, for example by terminating the user environment.

As you can see, the trap frame the processor pushes on kernel entry from user is similar to the one it pushes when it is *already* in the kernel, except in this case the processor also pushes the SS and ESP registers *before* pushing the frame discussed in the preivous part of the project. You can see this difference reflected in the comments in the definition of `trapframe`in `inc/trap.h`, and on the definitions of the `trapframe_usize()` and `trapframe_ksize()` macros in the same header file.

For those traps for which the processor also pushes an error code when taking a trap from kernel mode, as discussed in the last part, it pushes an error code in the same fashion for traps from user mode. For these traps, therefore, a trap from user mode will leave the kernel stack in the following state:

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

Exercise 9. Modify `cpu_init()` to set up the `tss` field in the `cpu` struct appropriately, and to create a TSS segment descriptor in the GDT pointing to it. The SEGDESC16 macro should make the latter part easy: note that you can use this macro in an assignment statement, as in '`c->gdt[...] = SEGDESC16(...)`'. A GDT entry for the TSS named `CPU_GDT_TSS` has already been reserved in `kern/cpu.h`.

Then, at the end of `cpu_init()`, after the processor's GDT is loaded, load your TSS into the processor using the LTR instruction. For convenience, there is an `ltr()` function in `inc/x86.h`. Make sure your code "accepts" your TSS descriptor: if it doesn't, the LTR instruction will cause a trap, which the kernel probably won't handle very well because `trap_init()` hasn't yet been called at this point.

Entering User Mode

What piece of register state in the processor actually defines which privilege level it is executing in at a given moment? Many architectures use a "kernel mode" flag in a control register of some kind, but x86 processors uses the low two bits of the CS register, effectively treating privilege level as a property of the currently running code segment.

To enter user mode, we might be tempted simply to take the kernel's code segment selector, `CPU_GDT_KCODE`, set the bottom two bits to 1, and do an `ljmp` to that code segment much like`cpu_init()` does to load `CPU_GDT_KCODE` as a kernel-mode code segment. Unfortunately, this doesn't work: *why?*

Exercise 10. Modify the definition of `cpu_boot` in `kern/cpu.c` to create code and data segments for user mode. Other than privilege level, these segments should be

identical to the corresponding kernel segments: we aren't going to do any address translation or memory protection for now.

We described above how the processor *leaves* user mode and enters the kernel via a trap, but how does the kernel *enter* user in the first place? Simple: the kernel "returns" to user mode - even if the processor has never been there before!

When the processor executes an `iret` instruction, it pops its standard trap frame off the stack starting with the old EIP, but it doesn't actually know or care whether *it* actually pushed that frame on the stack or if it got there some other way. Thus, the kernel can always *manufacture* a trap frame representing whatever user mode state it wants to load into the processor, and "return" from it via `iret` to enter user mode. (There are other ways to switch to user mode on the x86, but this is the most general method.)

Exercise 11. Modify the code at the end of `init()` to cause the `user()` function to be run in user mode, instead of just calling it as a procedure like the skeleton code does, which leaves the processor in kernel mode. To do this, you will need to create and set up a `trapframe` struct correctly, and call `trap_return()` to "return" to it. The user mode code will need a stack, which must be separate from the kernel stack since the kernel stack will be used for trap handling: we have defined an array called `user_stack` for this purpose.

Note: You should make sure the processor enters user mode with the I/O privilege level set to 3, by setting the `FL_IOPL_MASK` bits in the EFLAGS register to the value `FL_IOPL_3`. This will allow console output, such as the `cprintf()` at the beginning of `user()`, to work even from user mode for debugging purposes. We would not want to do this for ordinary applications, since that would give them unrestricted access to the PC's I/O space: we will later provide a way for user mode code code to produce output in a controlled fashion via system calls.

You should now see `user()` print '`in user()`' from user mode: verify that you're really in user mode here by setting a GDB breakpoint at `user` and looking at the CS register at that point.

Software Interrupts

Now that your kernel has basic exception handling capabilities and can enter user mode, you will refine it to handle traps that user mode code may cause deliberately for various purposes; we refer to such traps as *software interrupts*. There are two traps defined by the x86 processor expressly to serve as software interrupts, and PIOS defines a third one to serve as its system call mechanism:

- `T_BRKPT`: The breakpoint exception, interrupt vector 3, 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.
- `T_OFLOW`: The overflow exception, interrupt vector 4, allows software to generate a trap deliberately via the special `into` software interrupt instruction, if the overflow flag (FL_OF) is set when the instruction executes. The intent is for software to execute an `into` after an arithmetic operation that might overflow: if it doesn't, execution proceeds normally, but if it does, the overflow condition is immediately caught. This appealing idea has never really caught on in high-level languages, however, and is rarely used. Nevertheless, on principle we want to ensure that PIOS can handle `into` instructions in applications properly.
- `T_SYSCALL`: This value defined in `inc/trap.h` is *not* one of the 32 defined by the x86 architecture, but is somewhat arbitrarily assigned by PIOS to be 48 (0x30), one of the higher vectors in the 256-vector space. PIOS application code will invoke this trap vector deliberately with the `int` instruction when it wishes to make an explicit system call to the kernel. This is the classic way to perform system calls on the x86.

We do not want user code to be able to invoke *just any* interrupt vector in the IDT deliberately via `int` instructions, however: that might allow user code to confuse the kernel into thinking that some special event has occurred when it has not. For this reason, all IDT descriptors have a *descriptor privilege level* indicating what privilege level is required for software to invoke that interrupt vector deliberately via a software interrupt instruction. Most of these vectors are normally set to "privileged" (ring 0), but we want the vectors used for software interrupts to be set so user mode code can invoke them.

Exercise 12. Modify `trap_init_idt()` to allow user mode code to invoke the `T_BRKPT` and `T_OFLOW` vectors via software interrupt instructions. Don't worry about `T_SYSCALL` for now; we'll do that in the next project when we start implementing useful system calls.

The call from `user()` to `trap_check()` in user mode should now succeed, and you should be able to get `make grade` to succeed on the user mode test.

Part 4: Physical Page Allocation

The operating system must keep track of which parts of physical RAM are free and which are currently in use. The final part of this project is to construct a physical

memory allocator for the kernel, so that the kernel can dynamically allocate memory and later free it. Before starting this part of the project, be sure you have read and understood the part about memory allocation in xv6 chapter 2 (on SOCS).

Your allocator will operate in units of 4096 bytes, called *pages*. Pages will become highly relevant in project 3, where you will implement virtual memory, but for now take it on faith that pages are useful units of allocation in kernels. Your task will be to maintain data structures that record which physical pages are free and which are allocated, and how many users there are of each allocated page. You will also write the routines to allocate and free pages of memory.

Dividing up physical memory into pages of exactly equal size will make memory allocation in the PIOS kernel substantially simpler than it is in xv6, or any user-level implementation of `malloc()` for that matter, because you won't have to deal with the problems of fragmentation or searching for a sufficiently large free chunk to fill a request for a given amount of memory. This design simplicity has its costs: namely, the rest of the kernel will not have any way to allocate physically contiguous memory for structures larger than a page after the kernel boots. As it turns out, the PIOS kernel will not *need* to allocate any data structures larger than a page after it boots. But even some much more complex kernels - Linux, for example - impose restrictions such as this to keep their physical memory allocators simple and fast, and work around these limitations using virtual memory to make physically discontiguous memory appear virtually contiguous to the kernel code using that memory.

You'll now write the physical page allocator. It keeps track of which pages are free with a linked list of `pageinfo` structures, each corresponding to a physical page.

Exercise 13. In the file `kern/mem.c`, you must implement code for the following functions.

```
mem_init()
mem_alloc()
mem_free()
```

We have provided a function in the same source file, `mem_check()`, which tests your physical page allocator. You should boot PIOS and see whether `mem_check()` reports success. Fix your code so that it passes. You may find it helpful to add your own `assert()`s to verify that your assumptions are correct. Our checking code is certainly not guaranteed to detect every possible bug, and any bugs your code has that it does *not* detect may easily come back to bite you in a future project!

This project, and all the projects in this course, will require you to do a bit of detective work to figure out exactly what you need to do. This assignment does not describe all the details of the code you'll have to add to the kernel. Look for comments in the parts of the source that you have to modify; those comments often contain specifications and hints. You will also need to look at related parts of PIOS, at the Intel manuals, and perhaps at your notes from relevant courses you may have taken previously.

This completes the project.