Contents

1 Introduction 2
  1.1 Overview ................................................................. 2
  1.2 Simplifications ......................................................... 3
  1.3 Build Infrastructure .................................................. 4
  1.4 Hand-in ................................................................. 4

2 Guest Kernel Execution 4
  2.1 Core Requirements ..................................................... 4
  2.2 The Address-Space Collision .......................................... 5
  2.3 The “LIDT Problem” ..................................................... 6
  2.4 The Multiboot Information Structure ................................ 6
  2.5 Launch State ............................................................ 7
  2.6 Interrupt/Exception/Trap Delivery .................................... 8
  2.7 Specific Surprises ...................................................... 8
  2.8 Guest Shutdown ........................................................ 9

3 diStorm 9

4 Virtual Consoles 9
  4.1 System Call ............................................................. 10
  4.2 Utility Program ......................................................... 10
  4.3 Usage Scenarios ....................................................... 10
  4.4 Architectural considerations ......................................... 11

5 Plan of Attack 12
1 Introduction

This semester, over the course of a week, you will turn your kernel into a hypervisor capable of hosting 64-bit Linux and Windows 8 guests. Just kidding, that would not be possible, not even for Eric Faust.\(^1\) The actual P4 assignment for this semester is to turn your kernel into a hypervisor capable of hosting 15-410 Project 1 game “kernels.”

As you may recall from lecture, the authors of CP-40, the first environment for virtualizing a hardware platform, found that the IBM S/360 architecture wasn’t fully virtualizable, resulting in the S/370 architecture. A similar thing happened with the x86 platform, which was for many years not fully virtualizable but now is. Our P4 assignment will involve a mixture of architecturally defensible techniques and tasteful hacks so that you achieve some amount of virtualization experience without resorting to the large, complicated x86 virtualization mechanisms (“VT-x” and “VT-d”).

Because a guest P1 kernel consumes the entire keyboard and screen while it’s running, there is no sensible way to run multiple guests at the same time without multiple keyboards and screens. Because of this, we will specify an interface for implementing virtual consoles, though you are not required to implement it—it’s just for fun.

Through the course of this assignment, we expect that you will learn:

• the scope of tasks a hypervisor needs to accomplish (CPU modeling, I/O-device modeling, time management),

• more about x86 privileged instructions, I/O devices, and CPU data structures, and

• more about segmentation, both as an x86-specific artifact and as a general tool

1.1 Overview

This document consists of the following parts:

• Discussion of necessary simplifications

• Specification of the “PebP1” semi-virtualization environment

• Discussion of the “diStorm” disassembler

• A specification of virtual consoles (optional)

• Attack plan, including suggestions

Good luck!

\(^1\)On a dare, Eric, a former OS TA, has gone from a blank screen to p3ck2 in under 48 hours. But Bruce Schneier could write a complete hypervisor in 15,410 seconds!
1.2 Simplifications

Due to the scope of the project and the time available, certain simplifications are reasonable.

- One massive simplification is that we will require you to implement only a tiny sliver of the vast array of I/O device hardware contained in even an original 1980 PC. In particular, since P1’s use only the Intel 8253 interval timer, the Intel 8042 PS/2 keyboard/mouse controller, the Intel 8259 programmable interrupt controller, and the IBM VGA-compatible graphics controller, that is all you will need to emulate. Since P1’s use only a small fraction of the capabilities of each of those devices, your implementation burden is further reduced.

- Because keyboard input will be multiplexed between line-oriented consumers (\texttt{readline()}) and character-oriented consumers (virtual keyboard interrupts delivered to guest kernels), you are authorized to restructure your \texttt{readline()} implementation so that invoking threads essentially loop calling \texttt{getchar()}. Blocking should still happen, inside \texttt{readchar()}, but you do not need to avoid scheduling and running a \texttt{readline()} caller once per available character.

- Guest kernels will have an inaccurate notion of time, i.e., their clocks will not “tick” at the expected rate due to other threads competing for CPU time. For this assignment, this is not a problem (in general it is an unsolvable set of problems [2]).

- Because P1’s do not switch from one privilege level to another, and do not think about what privilege level they are executing at, you are not required to virtualize privilege levels. That is, all execution of your P1 guests will take place at PL3, even though guests will assume they are running at PL0 because privileged instructions will appear to run. That is, guests will run at “physical PL3” but “virtual PL0.”

- Because P1’s do not use virtual memory, you are not required to virtualize virtual memory. The performance and complexity costs of virtualizing virtual memory (e.g., page-table compilers) are a key reason why the second generation of x86 hardware support for virtualization includes nested page-table hardware. Each guest will execute inside a single address space of a fixed size, with read/write/execute access to its whole address space at all time.

- If time is short, one thing to skimp on is correctly handling the cleaning up of crashed or exited guests. It is probably a good idea to print out a somewhat detailed message indicating why a guest is exiting, but in an emergency you could get away with just dropping a broken guest from the run queue.

- Neither your kernel nor guest kernels are expected to use floating-point instructions.

Be sure to document your approach and design decisions before you submit your project; also be sure to mention any shortcuts you were forced to take. We are interested in understanding what about the project did and didn’t work out in case it is used in the future.
1.3 Build Infrastructure

We will provide you with some test P1 payloads. In addition, it is likely that you will wish to run your own P1's.

To include a 410-provided payload, found in 410user/files, add its name to 410FILES in your config.mk. Likewise, to include your own payload, place it in user/files and add its name to STUDENTFILES. Note that you might want to update your .gitignore file if you will have many versions of a given payload.

1.4 Hand-in

Please remember to make veryclean.

When handed in, your kernel must be runnable! This means that it must, upon being built and booted, start running idle, init, and shell without user intervention. In particular, it must not drop into the Simics debugger. When we run the test suite, there will not be a human present to continue execution. Thus, the test harness will declare your kernel to have failed the entire suite.

Also, your kernel should not generate reams of lprintf() debugging messages while running. Ideally you should adjust the setting of your trace facility so that it generates no messages, but in any case the normal loading, execution, and exiting of a program should not generate more than 20 lines of kernel.log output.

It is very important that your README explains which parts of the project you have got working, and which are working solidly.

2 Guest Kernel Execution

Guest kernels are launched with the standard Pebbles exec() system call. Your host kernel distinguishes between regular user-space executables and guest-kernel executables by checking whether the beginning of the text region is at versus below USER_MEM_START. After the thread that invoked exec() loads the executable and establishes the execution environment, it enters user space to begin execution of the guest P1 kernel. When something goes wrong with user-space execution, the thread will figure out what went wrong, and then will typically change some emulated-machine state and resume execution (perhaps at a later time).

2.1 Core Requirements

To achieve P1 virtualization you will need to support these key components of virtualization:

Privileged Instructions P1 kernels expect to be operating in PL0 and will thus attempt to use privileged instructions. Your hypervisor will need to be able to handle the general protection faults and do “the right thing” in each case.
**Keyboard Interrupts** Keyboard interrupts (and PS/2 scan codes) will need to be delivered to guest kernels.

**Timer Interrupts** It will be necessary for your hypervisor to deliver a timer interrupt to each guest kernel from time to time. You will need to determine (and document!) an appropriate strategy.

**Console (Memory-Mapped I/O)** Guest kernels will attempt to write into the memory-mapped I/O region designated to hold the CGA text console (`CONSOLE_MEM_BASE` and will expect those updates to be made visible to human users of the system. There are multiple approaches to making this work that have different complexity/performance tradeoffs.

### 2.2 The Address-Space Collision

At this point you may be worrying about an address-space collision. If your kernel is linked to run below `USER_MEM_START` and the P1 guest kernels are also linked to run below `USER_MEM_START`, how can that work? While virtual memory can be used to map a given virtual address to any physical address, mapping the same virtual address to two different places requires an address-space change, which requires some code to run, and the code can’t generally be in a part of the address-spaces that isn’t mapped the same way.

Luckily (for our current mission) x86 hardware has an additional mapping feature, namely segmentation! So far in this class we have stuck to a “flat” model where every segment starts at 0x00000000 and is 0xFFFFFFFF bytes long, but for P4 we will revoke that simplifying assumption. In particular, you will set up one or more segments with a base address of `USER_MEM_START` and a size much smaller than 0xFFFFFFFF. P1 guests will run using these segments instead of the standard segments for Pebbles user-space code (`SEGSEL_USER_CS`, `SEGSEL_USER_DS`). Thus when a guest starts execution with an `%eip` value like 0x001009cf, instead of the regular Pebbles user code segment performing the identity map and yielding a linear virtual address of 0x001009cf, `%cs` will contain a segment selector for a segment that turns 0x001009cf into a linear virtual address of 0x001009cf+0x1000000=0x011009cf. Because that address is outside the direct-mapped area your kernel inhabits, it will appear to be a regular user-space address and everything should be ok.

To accomplish this segment-based mapping, you will need to construct some segment descriptors and install them in the GDT. We have added four blank entries to the GDT, with slot indices `SEGSEL_SPARE0_IDX` through `SEGSEL_SPARE3_IDX`. We have also added to `seg.h` the corresponding *partial* segment selectors, `SEGSEL_SPARE0` through `SEGSEL_SPARE3`. You will need to carefully study the documentation in `intel-sys.pdf` on segmentation, segment descriptors, and segment selectors in order to accomplish the segmentation part of your mission. Don’t forget that segmentation is also discussed in the Project 1 handout and summarized on a web page on our “Projects” page.
2.3 The “LIDT Problem”

Guest kernels which use interrupts or handle exceptions (most of them!) will need to set up an interrupt-descriptor table (IDT). On the x86 platform, the location of the IDT is up to the kernel, which informs the hardware where it is by executing LIDT, a privileged instruction. Code which wants to install an IDT entry can use the SIDT instruction to store the base address of the IDT and then compute the location of the entry to be installed. If the x86 were fully virtualizable, both the LIDT and SIDT instructions would trap if executed in user mode. Unfortunately, as discussed in the “Virtualization” lecture, the classical x86 architecture is not fully virtualizable. One example of this is that the x86 SIDT instruction is not privileged—in other words, if user-mode code executes it, it will learn the address of the kernel’s IDT.

This poses a problem for a hypervisor that runs kernel code in user mode. When the guest kernel executes the LIDT instruction to set up its IDT, the processor will trap into the kernel; the hypervisor can record the location of the guest’s IDT, skip past the instruction, and resume the guest. However, when code in the guest kernel later wants to install an IDT entry, if it invokes SIDT to find out where its IDT is, it will be given the address of the actual kernel’s actual IDT. If the hypervisor has picked one address for its IDT and the guest kernel has picked a different address, the guest kernel will try to install IDT entries in its address space but relative to the base address of the hypervisor’s IDT. This installation may succeed, but the hypervisor won’t be able to find and decode the IDT entry properly—at least not if it indexes off of the base address that the guest previously specified in the LIDT instruction. Even worse, guest kernel code isn’t required to use SIDT to “remember” where its IDT is located; instead, it could store the base address in a variable, invoke LIDT with that value to inform the hardware, and then thereafter store into the IDT based on the value in the variable. These two options mean that a hypervisor can’t reliably know where the guest has stored each IDT entry.

For P4 we will rely on the following tasteful hack: your kernel will run our head.S startup code which places the IDT at a specific address using LIDT; your kernel will also store IDT entries using the idt_base() function, which invokes SIDT. P1 kernels executed by your hypervisor will run the same version of head.S, which will invoke LIDT with the same value, and will store IDT entries using idt_base()/SIDT. Because both the host and the guest will specify the same address to LIDT, both the host and the guest will receive the same answer when they invoke SIDT. Segmentation will ensure that while the two tables have “the same address” they will be located at different places in memory, and everything will work out.

To rely on this hack, please make sure that you rebuild your P1 after “make update” downloads the new head.S into your P1 build tree.

2.4 The Multiboot Information Structure

In the “Bootstrapping” lecture we discussed how a traditional PC BIOS finds and invokes a boot loader, and also how the “Multiboot standard” [1] was developed to provide a
framework so that multiple boot loaders could load multiple operating systems.

The Multiboot standard revolves around two blocks of information stored in memory. The Multiboot header is part of kernel executable files and explains how each kernel should be loaded into memory. In this P4 we will be ignoring the Multiboot header because you have already written `exec()` code which knows how to extract the relevant information from the ELF header contained in kernel executables built using our infrastructure. The other block, the Multiboot information block, is produced by boot loaders and provided to kernels when they are launched. Since you will be loading kernels and launching them, you will be responsible for producing multiboot information blocks (Section 3.3, “Boot information format,” of [1]).

You can find a declaration of the Multiboot information block in 410kern/boot/multiboot.h.

We recommend that you place a multiboot information block at address 0x15410, which happens to be in the “real mode free memory” area of the legacy BIOS execution environment memory map.

The MULTIBOOT_MEMORY flag should be on in the “flags” field. The mem_lower field, which is supposed to indicate the amount of “lower memory” (in a legacy BIOS execution environment) can be set to the value 637 (to indicate 637 kilobytes). The mem_upper field, which is supposed to indicate the amount of “upper memory,” should be set to the size of the virtual machine in kilobytes, minus 1024, because “upper memory” doesn’t count the bottom megabyte of physical memory.

It should be safe to leave the remainder of the Multiboot information block blank.

### 2.5 Launch State

When your host kernel launches a guest kernel, the execution state is as follows.

- Some number of frames have been allocated to the guest. You can launch every guest with the same fixed size (we recommend 20 or 24 megabytes), or you can optionally parse `argv[1]` as a decimal integer indicating a number of megabytes.

- A virtual address space has been prepared which maps those frames starting from USER_MEM_START.

- Segment registers will have appropriate values so that the guest virtual memory located above USER_MEM_START in the address space is reachable via `%eip`, `%esp`, etc., values which are under USER_MEM_START.

- The guest’s ELF image (text, rodata, data, BSS) are all loaded into the guest’s address space at the indicated addresses (relative to the re-mapping segments).

- `%EAX` contains 0x2BADB002 (to indicate a Multiboot launch)

- `%EBX` contains a pointer to the multiboot information structure (Section 2.4).
• %EIP contains the entry point of the executable.

• Other general-purpose registers (including %ESP) should contain 0.

• Guest virtual interrupts must be disabled.

2.6 Interrupt/Exception/Trap Delivery

If a “surprise” happens while a guest kernel is executing, control will transfer to your host kernel. In some cases, your host kernel will examine the guest instruction that triggered the surprise and simulate the effects that the guest was hoping for—for example, this will happen most of the time that the guest executes an OUT instruction. Once the simulation of the instruction is complete, the guest’s %eip value will be adjusted to skip the instruction and guest execution can be resumed.

However, sometimes a guest event, or even a non-guest event, may require the host kernel to deliver a surprise to the guest code, i.e., rewrite its execution state in such a fashion that it starts running a handler. You are already familiar with how this plays out, though you should keep in mind that, because your guest will always believe it is executing in PL0, stack switching will not be part of delivering a surprise to a guest.

A key thing to keep in mind is that your host kernel must not deliver an interrupt to a guest kernel if the guest kernel has disabled interrupts. Thus your host kernel must track the interrupt-enable status of each guest—and, in appropriate circumstances, the host kernel must modify the guest’s interrupt-enable state.

Note that it is possible that, as your host kernel is attempting to deliver a surprise to a guest, some necessary step cannot be carried out. If that happens, the guest is responsible for a double fault, which should be handled accordingly. If a double fault cannot be handled, this is a triple-fault condition, which should be handled appropriately—meaning that we expect you to reflect on how your kernel handles a non-guest Pebbles thread crashing.

2.7 Specific Surprises

If a guest P1 kernel runs into a problem, your host kernel will need to classify it and react accordingly. Here we will provide brief hints on selected issues.

• CLI - suspend delivery of virtual interrupts to the guest (of course, continue to deliver exceptions/faults/traps, which are not interrupts)

• STI - if a virtual interrupt is pending, deliver it; later, if a virtual interrupt arrives for this guest, it can be delivered immediately

• JMP FAR/LJMP - sanity-check the segment selector; the address should “make sense”

• LGDT - probably safe to skip (see head.S)

• LIDT - see Section 2.3
2.8 Guest Shutdown

If a guest kernel executes the HLT instruction while it has virtual interrupts disabled, execution has by definition concluded, so the kernel should clean up the thread which was executing the guest and then clean up the task. Whatever value was in %eax when the guest terminated should be reported as the exit status of the task.

3 diStorm

In order to determine how to react to a guest-caused surprise, you will need to know which instruction the guest was executing when things went wrong. To help with this, we have provided you with a copy of the “diStorm” x86 disassembly package, in 410kern/distorm. The diStorm package is quite powerful and very complicated. However, we have provided you with a one-function wrapper, disassemble(), which is declared in distorm/disassemble.h. Serious hypervisors don’t work by disassembling instructions into strings and then running strcmp() because the performance of doing that is terrible, but this is a reasonable thing for you to do for P4.

4 Virtual Consoles

What “virtual consoles” means is that the single physical screen and keyboard are multiplexed by the operating system (that’s you) so that it looks as if there are multiple instances of each one. Output routines such as print() will paint a virtual screen, which may or may not be visible, and keyboard scan codes will result in characters being available on various keyboard queues at various times.

From the user’s point of view, there should be multiple independent text consoles, and pressing the Tab key on the keyboard should switch the screen and keyboard “promptly” from one virtual console to another (by “promptly” we mean that this should happen promptly—not after the next readline() completes, etc.).

Threads belonging to a task share a virtual console. Each newly created task will begin using the same virtual console as its parent task.

If you wish to use a different color combination for each virtual console, they must all be reasonable.
Once a virtual console has no more threads or processes associated with it, it should be deleted from the Tab key rotation after the user has had a chance to view the contents once (it might be friendly for the kernel to somehow indicate to the user that this is the last viewing of the console and optionally for the kernel to require a confirmatory keypress).

4.1 System Call

To activate virtual consoles, one new system call is added:

- int new_console(void) - If it is possible to create a new console, do so and switch all further console I/O of the calling task to the new virtual console.

If too many virtual consoles are in use, new_console() may fail. You should support at least four virtual consoles.

If any thread in a thread family has a console I/O operation pending or in progress, new_console() should fail.

If nobody ever calls new_console() the operation of the console/keyboard system should be as it was for P3. Also, user code should have no way of observing it is not running on a P3 kernel without virtual consoles (aside from the behavior of new_console()).

4.2 Utility Program

The update area contains a program called new_shell which uses the new_console() system call to launch a shell in a new console window.

4.3 Usage Scenarios

Here is a usage scenario which hopefully clarifies how virtual consoles should work.

1. A Pebbles kernel implementing virtual consoles is booted and init launches the shell. The shell is running in the system’s sole virtual console.

2. The shell prints a prompt and invokes the readline() system call.

3. The kernel blocks the shell.

4. The user types new_shell and hits Return.

5. The kernel unblocks the shell, which invokes fork() and (in the child) exec().

6. The shell blocks on wait().
7. The new shell program invokes the fork() system call and (in the child) the new console() system call. The kernel creates a second virtual console, updates the task data structure so it references the new virtual console, and switches to displaying the new (blank) virtual console. The new shell child invokes the shell via exec(). The shell prints a prompt (in the second virtual console) and invokes the readline() system call.

8. The kernel blocks the shell.


10. The first shell, running in the first virtual console, which is not visible, completes the wait() system call, prints a program-completion message, prints a prompt, and blocks in readline(). All of this output (including cursor motion) is stored in the first virtual console’s data structures so it can be viewed later, but none of these changes are visible on on the screen.

11. Meanwhile, the user types “cho” at the second shell’s prompt, but does not hit Return. Instead, the user presses the Tab key.

12. The kernel switches to the first virtual console, which involves displaying its contents on the screen. The user sees the completion message from the shell and the shell prompt. Now the user hits the Tab key again.

13. The kernel switches to the second virtual console, where the user sees the shell prompt and “cho” following it.

14. The user hits the Return key.

15. Now the kernel unblocks the second shell, readline() completes, the shell launches cho, and the user can see output scrolling by on the console.

16. If the user hits the Tab key, the screen goes back to the first virtual console, where nothing is happening, since the first shell is blocked in readline(). Hitting the Tab key again will display the contents of the second virtual console, where cho may still be running, or may have finished.

4.4 Architectural considerations

As you can see, virtual consoles complicate the implementation of readline() and the keyboard interrupt handler. In particular, when a scancode arrives, the data structure for some virtual console needs to be updated, a character may be need to be printed to a virtual console, and changes made to the virtual console need to be reflected on the actual screen. Given the increased complexity of handling input, it is permitted for your readline() implementation to awaken blocked threads more often than once per completed line.
Guest virtual kernels complicate the situation further in two ways. First, guest kernels want scan codes, not characters. Second, guest kernels may run with virtual interrupts off. Scan codes that arrive while the guest has virtual interrupts off, up to a reasonable limit, should be queued for when the guest re-enables virtual interrupts. Third, when a guest kernel exits, the virtual console in question, which probably contains a shell which is about to complete a \texttt{wait()} system call and launch a \texttt{readline()}, needs to stop queueing scan codes for the guest and resume processing scan codes into characters. This may serve as a further reason for \texttt{readline()} to awaken blocked threads before a line has been completed.

5 Plan of Attack

A recommended plan of attack has been developed. While you may not choose to do everything in this order, it will provide you with a reasonable way to get started.

If you find yourself embarking on a plan which is \textit{dramatically} different from this one, or a kernel architecture which is dramatically different from what we’ve discussed in class, you should probably consult a member of the course staff. It is quite possible that your approach contains a known-to-be-fatal flaw.

1. Unpack the tarball and follow the directions to upgrade your P3 build tree to a PebP1 build tree. Make sure that your kernel builds and runs without incident.

2. Carefully read this entire handout top to bottom and form a todo list. For example, it is likely that you will want to restructure your \texttt{readline()} so that it has more “loop around \texttt{getchar()} nature,” especially if you plan to support virtual consoles. Regardless, some restructuring will be necessary.

3. Read up on segmentation (using the materials listed above). Examine the new material in 410kern/x86/seg.h. Install one or more segment descriptors as appropriate, and figure out which segment selectors you will use. Potentially consult our web site’s list of useful Simics commands to see if anything there is useful.

4. Decide on key data structures. We encourage you to avoid adding random fields into random structs. Instead, we recommend clustering virtualization-related information together in an appropriate way.

5. Decide on key utility functions. How will you handle the need to read from and write to guest logical addresses? Is there an opportunity to encapsulate this in a clean way?

6. Write code to set up a P1 guest kernel environment, including kernel data structures and some user-space memory.

7. Write an “ELF lookaside loader”: Guest kernels should be loaded so that guest logical address 0x00000000 is loaded at hypervisor virtual address 0x01000000 \texttt{USER_MEM_START}. The guest kernel executable should be loaded into memory so that
when it is executed it believes its address space begins at 0x00000000. It may be helpful to draw a picture of the the address space as seen by the guest, the hypervisor, and the physical frame allocation.

8. Test your guest loader by invoking one of the “micro-guest” payloads we provide. Set a breakpoint on the first guest instruction (this is harder than it looks)—or maybe set a breakpoint on all guest instructions. Your guest exec() code should activate the right address space, disable interrupts, and enter the special guest user mode at the correct entry point. When the breakpoint trips, make sure that various parts of the guest are located at the correct address, in the eyes of the guest. For a while you will work in a mode where you are able to switch into a guest and do some things on its behalf, but perhaps not switch out of the guest and run something else. This is deliberate.

9. Write more of your guest-exec() code. For example, create, fill out, and install an mbinfo_t struct, at which point you should be able to initialize all guest registers to usable values.

10. Extend your GPF handler so that it does something “minimally supportive.” For example, it might MAGIC_BREAK and then yield(-1). Running guests will generate lots of GPF’s, so you’ll need this.

11. Write some code that, for virtualization guests, figures out the opcode of the faulting instruction.

12. Implement handlers for privileged instructions. To get started, you can read through 410kern/boot/head.S to see what order the first few of them will arrive in. In some cases, a privileged instruction attempting a particular operation can be successfully “emulated” by skipping over it. For example, if a P1 guest kernel attempts to change %cs in a way which would be legal if it were running in PL0 on hardware, you can’t let it carry out the change, but in order for it to keep running you will need to move on to the next instruction. For fun, you could implement a GPF handler which, for virtualization guests, always skips the faulting instruction and moves on (meaning: restores the execution state and returns to user space). Later, though, In many cases, you will need to update virtualization-related state, possibly after decoding instruction parameters and examining saved register state.

13. If you wish to run your personal P1, you can copy or symbolic-link one or another binary (kernel.strip is smaller than kernel) into user/files. Because of the “LIDT problem” (Section 2.3) you must ensure that your P1 is freshly built using the latest version of head.S (“make update”).

14. At some point you will be able to launch a P1 guest kernel and “run” it, albeit without being able to see what it’s doing and without it getting any interrupts. Note that at various points in your development process it may be useful to run two instances of Simics, one running your P1 on “hardware” and one running your P1 on top of your hypervisor, and to single-step them so they run in lock step.
15. Because most P1 guest kernels display some output before handling interrupts, you may wish to figure out how to make guest console output visible, at least in an initial-hack sort of fashion. It is ok to defer this until later if you would prefer. Note that there is more than one way to make this work, and some ways may be much easier than others.

16. Plan how to deliver interrupts in a general sense.

   • If an interrupt delivery is successful, which registers, which memory, and which non-register/memory state should be modified, in which ways?

   • Note that your kernel must not crash if some address is invalid while you are attempting to deliver an interrupt. As just one example, if the guest kernel’s `%esp points “somewhere bad” when your hypervisor wants to deliver an interrupt to the guest, you must crash the guest instead of crashing your kernel. There are many other corner cases you must be wary of.

   • Carefully review the conditions that must be true before an interrupt can be delivered. For example, recall that a second interrupt from a particular source won’t be delivered until the interrupt controller believes the CPU is done processing the previous interrupt from that source.

   • Note that if an interrupt is delivered to the guest and the relevant IDT entry is invalid then the guest should receive a double fault, unless of course the double-fault entry in the IDT is invalid...

17. Deliver one kind of interrupt (keyboard or timer). Each one has some associated issues to work out, including “of all interrupts of this kind that arrive, which ones belong to this guest kernel, and what do I do with the other ones?” You will probably refer to these files: `410kern/x86/keyhelp.h`, `410kern/x86/timer_defines.h`, and `410kern/x86/video_defines.h`. When working on keyboard interrupts, you may wish to consult some of the virtual-console documentation in Sections 4.3 and 4.4 (even if you do not implement virtual consoles).

18. Now deliver the other kind of interrupt.

19. If you’ve left anything out (e.g., guest crashing and/or guest cleanup), go back and add it.

20. Think about things that evil guest code might do. In theory, a guest being run by a hypervisor shouldn’t be able to do affect anything outside of its virtual machine.

21. Clean up code.

22. Document! Make sure your README contains what it should.

23. If you have time and energy, you can:

   (a) Implement virtual consoles to show multiple kernels running at once
(b) Implement a key sequence (Control-C?) to force-exit a guest kernel

24. Celebrate!

References
