Project 4: “PebPeb” Paravirtualization
Spring 2022 COVID-19 Special Edition

15-410 Operating Systems

April 15, 2022

Contents

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 10 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 slightly-modified 15-410 kernels.

1.1 Virtualization Approaches

As you may recall from lecture, there are multiple top-level approaches to implementing virtualization, including instruction-by-instruction emulation, binary code translation, "trap and emulate," and paravirtualization. Historically speaking, 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. However, Intel's mechanisms for virtualization, known as "VT-x" and "VT-d", are complicated due to the presence of many features intended to accelerate performance. As a result, this semester we will not follow the "trap and emulate" approach, but will instead pursue paravirtualization.

1.1.1 "Trap and Emulate"

A brief summary of the similarities and differences between the "trap and emulate" and paravirtualization approaches is in order. The key similarity is that both approaches run a guest kernel in user mode. This works for many instructions, e.g., `ADDL $1,%EAX` or `PUSHL %EBP`, because those instructions have exactly the same effects in kernel mode and user mode. However, kernels need to communicate to I/O devices (`OUT`), disable interrupts (`CLI`), switch address spaces (`MOVL %EAX,%CR3`), and carry out other environment-changing operations (writing to console memory, updating page-table entries). The "trap and emulate" approach relies on the hypervisor somehow finding out each time the guest kernel attempts a privileged operation, so it can safely and appropriately emulate the effects. If a guest kernel, running in user mode, executes a privileged instruction (`OUT`, `CLI`, `MOVL %EAX,%CR3`), the hardware will declare a general protection fault. The hypervisor will obtain control via the general-protection-fault handler, at which point it can inspect the instruction the guest kernel tried to run, figure out what to do, advance the guest kernel's `%EIP` past the instruction, and restart the guest. If a guest kernel, running in user mode, writes to an area of memory with special significance (console memory, page-table entries), the hypervisor will obtain control via the page-fault handler. In the "trap and emulate" approach, fault handlers in the hypervisor must disassemble faulting instructions to determine which action the guest kernel was trying to achieve.

[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*!

1.1.2 Paravirtualization

The paravirtualization approach, on the other hand, is based on the observation that most privileged operations a kernel carries out are deliberate and located in a small fraction of the overall kernel code; if it is possible to change the source code of the guest kernel in a small number of places, the guest can *explicitly* indicate which privileged operation it needs, saving substantial overhead. For example, in the "trap and emulate" approach, `disable_interrupts()` contains a `CLI` instruction, which will cause a GPF, at which point the hypervisor will disassemble the instruction pointed at by `%EIP` to figure out what needs to happen. In the paravirtualization approach, the kernel's author replaces the call to `disable_interrupts()` with a call to the `hv_disable_interrupts()` hypercall. A hypercall is an explicit system call to the hypervisor—in this case, asking that the guest kernel not receive virtual interrupts for a while. Because the hypercall interface is explicit, the hypervisor doesn't need to disassemble instructions to figure out what it is being asked to do, resulting in less coding for the authors of the hypervisor—meaning you. Also, in some cases an explicit hypercall interface makes it possible for multiple guest-kernel actions to be bundled up into a single hypercall, in which case the sequence can be carried out with one trap to the hypervisor instead of literally hundreds or thousands of traps. For example, recall the code you wrote in Project 1 to program the timer to run at a specific rate. With "trap and emulate," each `OUT` instruction would cause a trap and the hypervisor would need to decode the meaning of each instruction; with paravirtualization, this painful sequence of trapping and decoding could be replaced by a single "set timer rate" hypercall.

1.2 Guest Kernel vs. Guest User

Note that paravirtualization involves changes to the guest *kernel* code. Guest *user* code is not modified. When a guest user program wishes to grow its address space, it will call its `new_pages()` stub routine, which will invoke `INT $NEW_PAGES_INT`. Your hypervisor will switch the guest virtual machine from guest user mode to guest kernel mode and the guest kernel will start running a system-call handler for `new_pages()`. That handler code will probably allocate a guest physical frame, install the frame into the currently-active guest virtual address space, invoke the `hv_adjustpg()` hypercall, and resume execution in guest user mode via the `hv_iret()` hypercall. This is an important feature of paravirtualization: changes must be made to the guest kernel, but guest *applications*, including legacy commercial applications for which source code is not available, run without modification.

1.3 Other Considerations

Because a kernel consumes the entire keyboard and screen while it's running, there is no sensible way to run multiple guest kernels at the same time unless the hypervisor provides multiple keyboards and screens. Thus Project 4 also includes implementing virtual consoles.

Because writing a hypervisor is a very difficult task, multiple levels of achievement will result in passing grades. In other words, substantial credit will be given for partially completing the assignment, as long as the completed parts work well and achieve interesting results (see Section 6).

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),
- a bit about "I/O virtualization," by multiplexing one console/keyboard pair across multiple kernels,
- more about segmentation, both as an x86-specific artifact and as a general tool.

1.4 Simplifications

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

- Because the size of "kernel memory" is limited, it is acceptable for your hypervisor to support only a small number of guest kernels. In particular, mapping a complete 32-bit address space can require 4 megabytes of RAM, which could occupy one fourth of the available kernel memory. While some VM implementations may make it easy to support a large number of guest kernels,[2] you will receive nearly full credit even if you can't support more than two. Try to avoid being able to support only one, but in an emergency one will be better than none.
- Because keyboard input will be multiplexed between line-oriented consumers (`readline()`) and character-oriented consumers (virtual keyboard interrupts delivered to guest kernels), it will make sense for your `readline()` implementation to be structured so that invoking threads essentially loop calling `readchar()`. Blocking should still happen appropriately inside `readchar()`, so that regular host-kernel programs block when they call `readline()` (guest kernels don't have a way to block).
- 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. The notion of time in virtual machines is subtle,[3] and you are not expected to definitely solve it within the time available for this project.
- It may seem to you that a lot of execution time is spent building page tables which are used only briefly. This is true, and page-table caching substantially improves hypervisor performance, but it is outside the scope of this project.

[2]Supporting eight is genuinely feasible.

[3]See "Timekeeping in VMware Virtual Machines,"
https://www.vmware.com/files/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf

- 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.

1.5 Additional Spring 2022 Pandemic Simplifications

The majority of this document describes an assignment which we have used in previous semesters. However, the course staff is aware that this is not a typical semester, that students may be more burned-out, and that your time may be more limited than typical due to external events or the need to catch up in other classes. However, we still want to provide a timely and learning-intensive project. To that end, we are attempting to reduce the functionality which we will grade, focusing on the parts we think involve the greatest ratio of conceptual learning to time, while still making the entire project available for students who happen to have extra time to spend. Note that "merely" coming to a full understanding of the material in this handout and in the Virtualization lectures is a high-density learning activity.

You will receive nearly full credit for the project (see Section 6) if you adopt the following simplifications:

- Virtual console support is optional (`new_console()` is allowed to always return an error). This means that running multiple guests is technically possible but impractical if they rely on keyboard input and/or do any substantial amount of console I/O.
- `hv_setidt()` can reject all IDT entry numbers above 33, and can ignore the "privileged" parameter.
- `hv_setpd()` and `hv_adjustpg()` can be treated as illegal (i.e., it is ok to just crash the guest).
- `hv_iret()` can crash the guest if esp0 is not 0.
- Steps in the attack plan after "flayrod" and "cliff" can be regarded as aspirational rather than required.

1.6 Build Infrastructure

The Project 4 infrastructure is designed to build dual-purpose kernel binaries which can be booted on real hardware or launched inside a PebPeb hypervisor. This means that whether you are trying to build a PebPeb guest or a PebPeb host you will use the same tarball, update script, makefile, etc.

The startup code that runs before `kernel_main()` has been modified to check whether it has been launched as a kernel by the GRUB boot-loader on real hardware or as a PebPeb guest kernel by a PebPeb hypervisor host. In the guest-kernel case, the GDT/TSS/IDT-setup phase will be skipped; in either case, the LMM memory allocator will be configured appropriately and `machine_phys_frames()` will return the relevant memory-size information. In either case, `hv_isguest()` will return whether the kernel image is running in guest or host (hypervisor) mode.

The net effect is that it is possible to build a single guest binary which can be booted on hardware or launched as a paravirtualized guest. While dual-nature kernel binaries are possible, *this is not a requirement for Project 4.* You will submit a version of your kernel that supports PebPeb guests via paravirtualization; you are not required to also modify your kernel so that it can run as a paravirtualized guest. That might be fun (you could boot your kernel inside your kernel), but it is strictly optional.

We have provided you with guest kernels in two forms. First, there are various `.bin` files in the `410guests` directory. You may include any or all of these in your kernel's RAM disk by listing them in a `410GUESTBINS` directive, which you will need to add to your `config.mk` file. Second, most of the guests are provided in source form, also in the `410guests` directory. This source code can't be compiled into a guest kernel "in place"; in particular, it is missing hypervisor stubs. Directions in `guests/README` explain how to build these guests from source, which can be useful if you want to add debugging code or slightly modify the behavior of a guest. Once you have set up a given guest to be built from source you will move its name from `410GUESTBINS` to `STUDENTGUESTS`.

1.7 Debugging Warning

You will need to be creative during debugging. In particular, the Simics debugger isn't as organized as it might be when it comes to debugging code when two symbol tables lay claim to the same part of the address space. You may need to set breakpoints carefully, and you may wish to disassemble payloads for reference purposes (using, e.g., `objdump`).

1.8 Hand-in

Please remember to `make veryclean`. This will clean up not only the object files in your kernel tree but will also clean up the build trees of any guests you are building from source.

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.

1.9 Document Roadmap

This document consists of the following parts:

- Introductory material about paravirtualization
- Description of the "PebPeb" guest execution environment
- Specification of the "PebPeb" hypercalls
- A specification of virtual consoles
- Attack plan, including suggestions

This document does *not* contain all information necessary to complete the project. In particular, we expect it will be necessary to carefully study the lecture material, the source code of the guest payloads we provide, and documentation provided in the new header files in the `spec/` directory. Also, we expect that you will need to read this document top-to-bottom at least three times.

2 Guest Kernel Execution

2.1 The Address-Space Collision

At this point you might be worrying about an address-space collision. If your kernel is linked to run below `USER_MEM_START` and the guest kernels you will be hosting 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. For example, when your hypervisor kernel is booted, the first page of your code region is located in physical memory at `0x100000`; when your kernel enables virtual memory it will "direct-map" that page so that virtual address `0x100000` maps to physical address `0x100000`. When you launch a guest kernel, it will expect that the first page of *its* code region will be located in physical memory at `0x100000` (obviously it can't be), and it will try to establish a direct mapping from virtual address `0x100000` to physical address `0x100000` (obviously your hypervisor can't allow it to do that).

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`. When your hypervisor kernel runs a standard Pebbles executable such as the shell, the program will run with the traditional segments for Pebbles user-mode code (`SEGSEL_USER_CS`, `SEGSEL_USER_DS`). However, when your hypervisor kernel runs a "PebPeb" guest, both the guest kernel code and the guest user code will use the new user-mode segments you will set up.

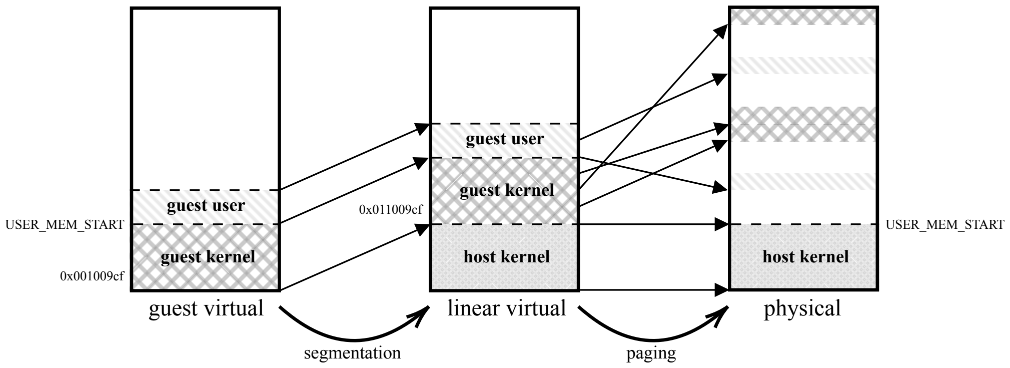

Figure 1: Using segmentation to "lift" guest address spaces

Let's assume that the very first instruction a guest kernel will execute is located at guest address `0x001009cf`.[4] If we launched the guest using the standard `USER_CS` segment, the segment descriptor would identity-map code-segment address `0x001009cf` to linear virtual address `0x001009cf`, which the kernel would have mapped as supervisor-only, and the instruction would receive a page fault. However, you will launch the first instruction of the guest kernel using a different code segment, with a base of `USER_MEM_START`. Thus when the guest issues an instruction fetch for code-segment address `0x001009cf`, the segmentation system will turn that into linear virtual address `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. In other words, the guest will think it is executing an instruction at virtual/physical address `0x001009cf`, but the instruction will actually be located at virtual address `0x011009cf`. If the guest kernel eventually launches some guest user code, that code will appear to guest kernel code and guest user code to be located at virtual address `USER_MEM_START`, but in terms of the

[4]The guest kernel launches in a direct-mapped address space, so the first instruction is located at guest-virtual address `0x001009cf` *and* guest-physical address `0x001009cf`; this will be discussed further in Section 2.2).

address space set up by your hypervisor, the code will actually be located at virtual address `USER_MEM_START+USER_MEM_START` (see Figure 1).

The main downside to this segmentation trick is that neither guest kernel code nor guest user code will be able to refer to addresses in the last sixteen megabytes (`0xFFFFFFFF−USER_MEM_START=0xFEFFFFFF`). Your hypervisor will inform each guest of its maximum virtual address at launch time (see Section 2.3). This maximum virtual address might be `0xFEFFFFFF`, or some smaller value if your hypervisor wishes to reserve some virtual address space for its internal needs.

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` corresponding *partial* segment selectors, `SEGSEL_SPARE0` through `SEGSEL_SPARE3`. In addition, we have added a function `gdt_base()` which returns the base address of the GDT. 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.

2.2 The "Boot VM" Address Space

When a Pebbles kernel boots on PC hardware (or Simics), `kernel_main()` begins in protected mode with paging off. In this mode, the kernel has access to all physical memory without any mapping.[5] Most Pebbles kernels eventually set up page tables and turn paging on, but some (e.g., P1 games) do not, and even the ones that do use paging need to run setup code in order to set up page tables.

When your hypervisor launches a guest, obviously paging will be on because your hypervisor will have turned it on, and paging, once on, is never turned off. But the guest will need to run substantial amounts of code before it makes a `hv_setpd()` call to define a virtual address space—if, indeed, it does so at all. Before launching a guest, you will have allocated some number of actual physical frames for that guest to use, and you will have set up some numbering system so that the guest will believe it has access to frame 0, frame 1, etc.—these will be "guest physical" frames in the sense that the guest will put those numbers into the page tables it creates. Before you launch the guest you should set things up so that if the guest accesses address `0x00000000` it refers to the frame that the guest thinks of as frame 0, address `0x00001000` refers to the frame that the guest thinks of as frame 1, etc.; with all of guest memory being read/write. This is referred to as the "boot VM." If a guest ever issues the `hv_setpd()` hypercall to define a virtual address space, the boot VM mapping can be discarded—there is no way for a guest to "disable paging" and thus no way for it to return to the boot VM operating environment.

[5]Actually, a few "secret" mappings are in effect; if you are interested, look up "BIOS shadowing," "System Management Mode," etc.

2.3 Launching a Guest Kernel

A guest kernel is launched when a user program such as the shell, running on the host kernel, invokes the standard Pebbles `exec()` system call on an ELF binary which contains a guest kernel. A 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`.

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

- Some number of hypervisor (actual) physical frames have been allocated to the guest. You can launch every guest with the same fixed size (which must be at least 20 megabytes), or you can optionally parse `argv[1]` as a decimal integer indicating a number of megabytes. The guest will refer to these hypervisor physical frames (which may well be non-contiguous) using contiguous guest physical frame numbers counting upward from zero.
- A "boot VM" virtual address space has been prepared which direct-maps those frames (guest virtual page X maps to guest physical frame X, which in turn is mapped to the appropriate hypervisor physical frame). The "boot VM" address space closely resembles hardware protected mode with paging disabled, though of course paging will be on for every instruction executed by the guest.
- The guest's ELF image (text, rodata, data, BSS) are all loaded into the guest's address space at the indicated addresses. As is the case when a regular Pebbles kernel begins execution, these guest virtual pages are all read/write.
- `%EAX` contains the magic value `0x15410DE0U` (`GUEST_LAUNCH_EAX`).
- `%EBX` contains the maximum legal guest physical frame number
- `%ECX` contains the maximum legal guest virtual address
- `%CS` contains a selector for a 32-bit ring-3 read/execute code segment with an offset of `USER_MEM_START` and a limit less than `0xFFFFFFFF`.
- `%DS`, `%ES`, `%FS`, `%GS`, and `%SS` each contain a selector for a 32-bit ring-3 read/write data segment with an offset of `USER_MEM_START` and a limit less than `0xFFFFFFFF`.
- All other registers (including `%ESP`!) are zeroed.
- The virtual IDT contains no valid entries.
- Virtual interrupts are disabled.

2.4 The Hypervisor Call Interface

PebPeb hypervisor calls use a different calling convention than Pebbles system calls.

To make a hypervisor call, a PebPeb guest kernel places the call number into `%eax` and invokes `INT $0xDE` (`$HV_INT`—all values are documented in `spec/hvcall_int.h`).

If the hypervisor call in question takes parameters, they are found in "reverse order" at `(%esp)`, `4(%esp)`, etc. For example, a guest kernel can move the cursor on its screen by invoking `hv_cons_set_cursor_pos(int row, int col)`. In that case, the address of `row` will be `(%esp)` and the address of `col` will be `4(%esp)`. When the hypercall returns, the return value will be found in `%eax`.

If a hypervisor call requests an action which is invalid, such as providing an invalid pointer or attempting to map nonexistent frames into its address space, the hypervisor should immediately crash the guest, i.e., there are no "failure return codes" in the specification. A reasonable hypervisor implementation will print a message somewhere describing what went wrong. "Crashing the guest" means pretending the guest had invoked `hv_exit(0xDEADC0DE)` (`GUEST_CRASH_STATUS`).

Note that the hypervisor must be careful when validating addresses specified by guest kernels. With the exception of page directories, which are guest-physical addresses, addresses specified by guests are guest-virtual, meaning that the hypervisor must translate them carefully in order to fetch or store the data.

2.5 Delivery of Virtual Interrupts/Exceptions/Traps

If a "surprise" happens while a guest kernel is executing, control will transfer to your host kernel. Some surprises will be due to an action taken by the guest, such as a hypercall; other surprises will be due to physical hardware interrupts. In addition, guest code (either guest user code or guest kernel code), might result in an exception or fault. Some surprises will result in resuming execution of the guest, but others, such as the physical timer interrupt, may result in a temporary suspension of guest execution.

There are many cases; here are some to consider.[6]

- A guest user program might divide by zero.
- A guest *kernel* might divide by zero.
- The physical timer might generate a physical interrupt while the guest is running kernel code.
- A physical key might be released while the guest is running user code.

In some situations a guest event, or even a non-guest event, may require the host kernel to deliver a surprise to the guest kernel, i.e., rewrite the guest virtual machine execution state so that the guest suddenly starts running a handler in guest kernel mode. A key

[6] "Consider" means answering "What should the result be?"

thing to keep in mind is that your host kernel must not deliver a virtual interrupt to a guest kernel if the guest kernel has disabled virtual interrupts. Thus your host kernel must track the virtual interrrupt-enable status of each guest.

From the point of view of the guest, virtual interrupt/exception/trap delivery occurs as follows.

1. Delivery of virtual interrupts is suspended (every vIDT entry is an "interrupt gate")
2. If the guest is not already in guest kernel mode, then:
   - The guest enters guest kernel mode.
   - The value of esp0 from the most recent `hv_iret()` invocation is loaded into `%ESP`.
   - The old value of `%ESP` is pushed onto the stack.
   - The value of `%EFLAGS` is pushed on the stack. The value that should be pushed is the *virtual/logical* `%EFLAGS` before the event. For example, if the event being delivered is a virtual interrupt, the `IF` bit in the pushed `%EFLAGS` should be 1, because if the guest had virtual interrupts off we wouldn't be delivering a virtual interrupt. However, if the guest had virtual interrupts disabled and encountered an *exception*, the `IF` bit in the pushed `%EFLAGS` should be 0, so that the right thing will happen if the guest passes this saved `%EFLAGS` value to `hv_iret()`.
   - The value `0x13370000` (`GUEST_INTERRUPT_UMODE`) is pushed onto the stack.

   Note that when a guest switches from guest kernel mode to guest user mode, which pages it has permission to access must change accordingly.
3. Otherwise (if the guest is already in guest kernel mode):
   - The value of `%EFLAGS` is pushed on the stack (as above).
   - The value `0x00001337` (`GUEST_INTERRUPT_KMODE`) is pushed onto the stack.
4. Regardless of whether or not there was a stack switch:
   - The old value of `%EIP` is pushed onto the stack.
   - An interrupt/exception/trap-specific code is pushed onto the stack as follows:
     - If the event is an x86 exception which specifies an error code (#PF, #GP), that code is pushed.
     - If the event is a virtual keyboard interrupt, the augchar[7] in question is pushed.
     - Otherwise, `0x00000000` is pushed

[7] "augmented char"—refer to the Project 1 documentation.

- One more word is pushed onto the stack: if the event is a page-fault exception, the most recent faulting address is pushed onto the stack *in guest-virtual format*; otherwise, a zero is pushed. Note that the x86-32 page-fault exception sets `%CR2` to the *linear virtual address* that caused the fault; please refer to Section 2.1.
- `%EIP` is set to the first instruction of the handler.

Virtual *traps* are delivered to the guest kernel in the same fashion as virtual interrupts and exceptions.

Note that if anything goes wrong while doing these steps (e.g., pushing onto the guest's stack fails), the hypervisor crashes the guest—this is the PebPeb equivalent of an x86-32 triple fault.

2.6 Virtual Interrupt Acknowledgment

The PebPeb environment virtualizes interrupts, but it does not expose an explicit model of the hardware programmable interrupt controller (PIC) to guest kernels. That is, there isn't a "paravirtual `outb()`" that sends an "interrupt acknowledge" command to the "paravirtual PIC."

In PebPeb, virtual device interrupts have been simplified compared to the x86 model. First, all virtual interrupt entries in the virtual IDT are "interrupt gates" in the sense that delivering one virtual interrupt to a guest automatically suspends delivery of other virtual interrupts to that guest. Second, when a guest re-enables virtual interrupt delivery (most naturally via `hv_iret()`), that is a statement to the hypervisor that the guest is ready to receive *all* virtual interrupts, including the next virtual interrupt from the most-recent source, so there is no need for a separate hypercall to explicitly enable further virtual interrupts from a particular virtual device. Skipping a "paravirtual `outb()`" step is a performance win, and one of the goals of paravirtualization is improving the performance of virtual I/O devices compared to physical I/O devices.

3 Hypervisor Calls

- `unsigned int hv_magic(void)` - Verifies that we are running under a version 2 PebPeb hypervisor, by returning the "magic number" `0xC001C0DE` (`HV_MAGIC`).
- `void hv_disable_interrupts(void)` - Suspends delivery of virtual interrupts (of course, virtual exceptions cannot be "suspended").

  Note that a guest invoking `hv_disable_interrupts()` does not mean that the hypervisor invokes `disable_interrupts()` or that physical interrupts stop happening! The hypervisor needs physical clock interrupts to keep arriving so that it can time-slice between guests. When a guest "disables" virtual interrupts, what is suspended is *delivery* to the guest kernel of virtual interrupts that "belong" to it.

  Note that when a guest kernel is launched virtual interrupts are initially disabled.

- `void hv_enable_interrupts(void)` - Begins or resumes delivery of virtual interrupts.

- `void hv_setidt(int irqno, void *eip, int privileged)` - Installs a virtual interrupt/exception/trap handler for the specified virtual-IDT slot—unless the specified `eip` is zero, in which case the handler is uninstalled. If the `privileged` parameter is non-zero, it is illegal for guest user code to invoke this handler via the `INT` instruction, so a virtual general-protection fault should be delivered to the guest kernel.

  If a guest experiences a virtual interrupt, exception, or trap at a time when no handler is installed for that event, the hypervisor crashes the guest—this is the PebPeb equivalent of an x86-32 triple fault.

  Your hypervisor must support at least those virtual IDT slots expected for operation of Pebbles guests, namely:

0..19	Hardware events	`410kern/x86/idt.h`
32..33	Virtual interrupts	`spec/hvcall.h`
65..116	Regular system calls	`spec/syscall_int.h`
128..134	Irregular system calls	`spec/syscall_int.h`

- `void hv_setpd(void *pdbase, int wp)` - Activates a new address space. The value of `pdbase` must be page-aligned; note that `pdbase` specifies a guest *physical* address (in other words, a guest frame), not a guest *virtual* address. If the value of the `wp` parameter is non-zero, guest kernel code should experience a page fault if it attempts to write into a guest virtual page which the guest has marked as both user-level and read-only (see Intel's description of bit 16 of `%cr0`).

  PebPeb page-directory and page-table entries support only a subset of the x86 hardware flags. In particular, only the present, read/write, user/supervisor, base-address, and "available for system programmer's use" bits are supported. All other bits (e.g., global) must be set to zero.

- `void hv_adjustpg(void *addr)` - Re-validates/invalidates/updates the mapping of one guest-virtual-to-guest-physical translation. The hypervisor will examine the guest page-directory and page-table entries which map the guest virtual address specified by `addr` (which must be page-aligned) and will make any appropriate changes to the page tables maintained by the hypervisor. This call may be used to inform the hypervisor that a mapping has been added or removed, that some flag bits have been changed, etc.

- `void hv_iret(void *eip, unsigned int eflags, void *esp, void *esp0, unsigned int eax)` - Atomically activates a new privilege level and/or execution context.

The registers `%eip`, `%eflags`, `%esp`, and `%eax` atomically take on the values specified by those parameters. If the `esp0` parameter is zero, execution will remain in guest-kernel mode; otherwise, execution will switch to guest-user mode and the `esp0` value will be used to deliver the next virtual interrupt, exception, or trap. In the post-`hv_iret()` execution context, virtual interrupts will be on or off depending on the value of the `IF` bit in the specified `eflags` parameter.

Note that the hypervisor must carefully examine the specified value of the `eflags` parameter: if it's invalid, it must crash the guest instead of allowing it unwarranted powers.

Also note that when a guest switches between guest kernel mode and guest user mode which pages it has permission to access must change accordingly.

- `void hv_print(int len, unsigned char *buf)` - Prints to the console. The maximum length is `HV_PRINT_MAX`.
- `void hv_cons_set_term_color(int color)` - Changes the color used for future printing on the console.
- `void hv_cons_set_cursor_pos(int row, int col)` - Sets the cursor position, if valid; otherwise, crashes the guest.
- `void hv_cons_get_cursor_pos(int *rowp, int *colp)` - Retrieves the cursor position, if the pointers are valid; otherwise, crashes the guest.
- `void hv_print_at(int len, unsigned char *buf, int row, int col, int color)` - This hypercall may be used to increase efficiency. It moves the cursor, changes the print color, prints, and then restores the cursor position and print color. The maximum length is `HV_PRINT_MAX`.
- `void hv_exit(int status)` - Ceases execution and passes the indicated status to the parent of the host task that launched the guest with `exec()`.

4 Virtual Consoles

Note: The functionality described in this section is *optional* for Spring 2022. That said, if you want to use the Tab key to multiplex keyboard events among guests without doing screen management or `readline()` buffer management, you may. Also, reading carefully through how virtual consoles are expected to work may be instructive.

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 to 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 proceses 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).

As a practical matter, it isn't clear there is a sensible usage model for a single console to be shared between multiple guests, or between a guest and a non-guest program. It is not even crystal clear what should happen with pending keyboard input when a guest is launched or exits in a console. Thus, you are not expected to ensure that "the single right thing" happens in such scenarios. Of course you shouldn't crash, but you have flexibility to set reasonable policy.

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. Try to 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

You have been issued a new user program, `new_shell`, which invokes the `new_console()` system call to launch a shell in a new virtual console. Please consult the source code in `new_shell.c` as you read the descriptive text below.

4.3 Usage Scenario

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 second shell prints a prompt (in the second virtual console) and invokes the `readline()` system call.
8. The kernel blocks the second shell.
9. The `new_child` parent task exits.
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 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 physical keyboard interrupt handler. In particular, when a scan code 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 augmented characters, not scan codes or characters. Second, guest kernels may run with virtual interrupts off. Key presses/releases that arrive while the guest has virtual interrupts off, up to a reasonable limit, should be queued for delivery 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 `wait()` system call and launch a `readline()`, needs to stop queueing augmented characters for the guest and resume processing them into characters for `readline()`. This may serve as a further reason for `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 *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. If you are not generally pressed for time, consider testing your P3 kernel on the crash box. While you are *not* required to get your P3/P4 working on the crash box, it is quite gratifying to see your kernel running on real hardware and hosting another kernel. If you want to try to get there, it is much easier to make sure your P3 kernel runs on the crash box before you add hypervisor support—it is generally easier to debug smaller and simpler programs than larger and more-complicated programs.

2. Review the P4 lecture material.

3. Carefully read this entire handout top to bottom and form a todo list.

4. Unpack the tarball and follow the directions to upgrade your P3 build tree to a P4 PebPeb build tree. Make sure that your kernel builds and runs without incident.

5. Read through the new files in `spec/` and also the explanatory material in `guests/README`.

6. Read the source code of key guests—at least `magic` and `console`, but arguably also `station`. It is probably wise to talk through and write down a detailed list of the events that must happen for `magic` to carry out its mission. If you are unclear on the meaning of specific parts of the specification, reading the code for further guests may be useful.

7. Read up on segmentation (using the materials listed above). Examine the new material in `410kern/x86/seg.h`.

8. Plan the virtual-console work. Depending on your group, it may make sense for one person to start work on this right away, or it may make sense to defer this part entirely until everything else works. Whenever you do it, here are some steps.

    - Consider rewriting `readline()` to have more "loop around `readchar()`" nature. Don't forget that `readchar()` should block.
    - Implement console-switching when `readchar()` is about to "process" the Tab key
    - Test with our "launch shell in new console" program

    Note that it is possible to *significantly* defer some of the virtual-console work. For example, nothing truly terrible will happen if virtual consoles are not deleted and instead remain viewable indefinitely even after the task(s) using them have exited. Also, it is entirely possible to run one guest even if your kernel supports only one console.

9. You may wish to go through the steps to build one of the guest payloads from source (this includes writing hypercall stubs). If for some reason you can't make this work, it would be wise to seek help from the course staff quickly.

10. You may wish to port a Project 1 game kernel to the PebPeb paravirtualized environment (see below). It isn't necessary to do this right away, but it is another way to firm up your understanding of the guest environment and what it requires, so you might want to plan now to do it later.

11. Decide on key data structures.

    - We encourage you to avoid adding random fields into random structs throughout your kernel. Instead, we recommend clustering virtualization-related information together in an appropriate way.

- Figure out how to track which hypervisor frames are allocated to the guest (for verification, mapping, and release).
- Figure out how to represent guest IDTs.

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

13. Install one or more segment descriptors as appropriate, and figure out which segment selectors you will place in various segment registers in order to access those segments. Potentially consult our web site's list of useful Simics commands to see if anything there is useful.

14. Write code to set up a guest "boot VM" (allocate a bunch of hypervisor frames for a guest, record the allocations somewhere, "direct-map" the guest frames).

15. Provide your kernel with "ELF lookaside loader" functionality: an ELF loader capable of loading into a different address than execution will take place at. Guest kernels should be loaded so that guest logical address `0x00000000` is loaded at hypervisor virtual address `0x01000000` (`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 address space as seen by the guest, the hypervisor, and the physical frame allocation.

16. Use the ELF lookaside loader to load a guest into the guest's boot VM. Disable physical interrupts, `set_cr3()` into the boot VM, and drop into the debugger. Check that everything is where it needs to be.

17. Write more of `exec_guest()`, to the point where you can launch the "hello" guest payload.

    - Form up a set of register values
    - Write code to launch the guest, probably involving `set_cr3()` and `IRET`. Be sure to enter the guest with physical interrupts disabled since neither the guest nor the host is really ready to take interrupts yet. For a while you may work in a mode where you are able to switch into a guest and do some things on its behalf, but not switch out of the guest and run something else. That is fine.

18. It should be easy to get the "dumper" payload running now. It is ok if for the next several steps launching a guest permanently disables physical interrupts.

19. Implement `hv_magic()` and test it with the "magic" payload. At some point you will want some sort of dispatch framework for the various hypercalls.

20. Now that your guest can make hypervisor calls, implement `hv_exit()` and the console-output calls, and run the "console" payload. Note that the guest specifies parameters

using guest-virtual addresses (based on %esp, which is a guest-virtual address); all of those addresses must be translated before/as your kernel uses them to fetch or store guest data. If guest launch has been permanently disabling physical interrupts, you might want to hack `hv_exit()` to turn them back on. Now you have reached "checkpoint 1"!

21. Implement `hv_disable_interrupts()` and `hv_enable_interrupts()` - start with a fake version that just sets/clears a flag.

22. Now don't disable physical interrupts as part of guest launch. Verify (using, e.g., "bg hello") that you can leave guest execution via a physical timer tick, run other things for a while, and resume guest execution. Don't worry about delivering virtual timer interrupts—as far as the guest is concerned, it is always running, but sometimes it runs slower than other times. At this point you have returned to having a "working system": launching a guest doesn't mean that your system becomes useless for running other things. This is good.

23. Write an initial version `hv_setidt()` that just registers handlers. If you launch the "station" guest, it should sit there doing nothing.

24. Write an initial version of virtual-interrupt injection. You can deploy it in a trial fashion: when you resume the guest after a context switch, if its virtual interrupts are off, merely resume it, but if they are on, mutate its state according to the exception-delivery protocol so it experiences a virtual timer interrupt. Now the "station" guest should do something.

25. Now you can write an initial version of `hv_iret()`: don't worry about validation or esp0, since for now we are concerned with going from guest-kernel mode to guest-kernel mode after handling a virtual interrupt. This should enable you to run the "tick_tock" payload. Now you have reached "checkpoint 2"!

26. Now that you have virtual clock interrupts doing something, you might try getting a hacked-up version of virtual keyboard interrupts working as well—for example, if a physical keyboard interrupt arrives while a guest is running with virtual interrupts on, deliver it to the guest, otherwise do something else. The "tick_tock" guest can provide a simple check of your work.

27. Now is probably a good time to think through the design and policy issues related to virtual interrupts, which might make sense to break down in terms of interrupt detection (what is the relationship between physical interrupts and virtual interrupts?) and virtual interrupt delivery (including delivery scheduling). Consider:

    - When is a virtual timer interrupt for a guest *detected*?
    - When should a virtual timer interrupt be *delivered* to a guest?
    - When a physical keyboard interrupt happens, what will it probably be interrupting?

- When should a virtual keyboard interrupt be delivered to a guest?
- Your virtual interrupt implementation might (or might not) involve an event queue or two.
- Hint: some hypercalls not directly related to virtual interrupts may be able to provide some "support" for the delivery of virtual interrupts.

28. If you have guest console output, guest keyboard input, and guest timer interrupts working, it should be possible to run the "gomoku" guest and play a game. This is actually quite a triumph, even if it is hard to explain to your friends and family members exactly what you have achieved.[8] You can also guestify `410kern/p1test/410_test.c` from Project 1—or maybe even your own P1 game.[9] If you have "gomoku" running well, it is possible to clean up your code and submit a successful, if not full-credit, hypervisor project.

29. Use the "flayrod" and "cliff" payloads to upgrade your handling of guest surprises. Now you are ready to tackle guest user mode and run a complete kernel.

30. Write page-table "compiler" code and `hv_setpd()/hv_adjustpg()`. Note that the guest generally needs two address spaces "handy":

    (a) the kernel-mode page table, which can variously read and/or write most or all of the pages mapped by the page directory, and

    (b) the user-mode page table, which can variously read and/or write only the pages which are mapped and not supervisor pages.

    You have multiple options for handling this. For example, you can repeatedly re-translate from the guest page tables into hypervisor page tables for the currently-required mode. Another alternative might be maintaining one *pair* of translated page tables to make it easy to "flip" between guest kernel mode and guest user mode.

    Once guest kernels can manipulate virtual memory, you can use various payloads we provide ("teeny," "vast," "warp") to check your work.

31. Upgrade `hv_iret()`, at least enough to get into guest-user mode. Now you should be able to run the "fondle" payload.

32. Now that a guest kernel can set up virtual memory to run a guest user, the next step is a way for guest user code to make system calls into the guest kernel. You will need to add code to your system-call entry points: check if a guest was running and, if so, package up the state before the `INT` instruction as a virtual exception, mutate the guest's execution state appropriately, and resume execution of the guest. Don't forget about `misbehave()`: if a guest user program and a guest kernel both agree on

[8] "This may look like a simplistic game with low-quality graphics. The important thing is that the machine it is running on does not exist! And it's experiencing time dilation equivalent to the surface of a neutron star! Mwa-ha-ha!"

[9] You may wish to consult and/or adapt `410kern/inc/malloc.h` from Project 1.

what `misbehave()` does, the hypervisor should enable the user code to invoke that system call in the guest kernel even if your kernel has no idea what `misbehave()` does.

33. Now you're ready to try running a real guest kernel! Luckily, we have provided "pathos" (in binary form).

34. Ensure that user code running inside a guest can't make hypercalls, also that guest kernels can't make non-hypercall system calls (it would probably be bad if a guest kernel called `fork()`).

35. If you cut some corners earlier (e.g., in virtual consoles), now might be a good time to re-visit those decisions.

36. Clean up code.

37. Celebrate!

6 Grading

In Spring 2011, the first time we issued a paravirtualization P4, only a small number of groups were able to support a complete guest kernel. Since then we have restructured the project to *substantially* reduce the overall amount of work and also the amount of infrastructure work that must be done before the first payload can be launched. Based on results obtained in several semesters, we believe it is possible for most groups to reach the point of being able to run P1 game kernels (even if their performance is poor—don't worry if you observe that). We plan to award nearly full credit (e.g., approximately 95%) for submissions that can run P1-relevant test payloads and faithfully run the "Gomoku" game payload.

In terms of how to best use your time, we suggest you aim to have an identifiable set of guest payloads that can reliably be launched, run, and shut down (and make sure your README actually identifies them!). Even if you don't have any virtual-console support at all, being able to reliably run some guest payloads indicates that you have understood key virtualization concepts in a hands-on fashion. Virtual consoles enable you to demonstrate running multiple guests at the same time, but we anticipate some groups might want prioritize the ability to run an entire guest kernel+user stack. Either way, please aim for reliability/repeatability, and don't forget to document your design decisions in your README.

Good luck!

A Guide to Guest Payloads

A.1 Guest descriptions

Here is a brief description of what each guest does.

hello This guest invokes `lprintf()` and spins. It is the simplest test of launching a guest.

dumper This guest invokes `lmm_dump()` and spins. It is an easy step after `hello`.

magic This guest invokes one hypercall, `hv_magic()`, uses `lprintf()` to emit diagnostics, and spins.

console This guest lightly tests the various console-output hypercalls.

station This guest uses virtual clock interrupts to print out the lyrics to a song.

tick_tock This guest exercises virtual timer and keyboard interrupts.

gomoku This P1 game guest relies on console output, keyboard input, and timer interrupts.

flayrod This guest lightly tests delivery of virtual page faults in the "Boot VM."

cliff This guest generates a fatal exception.

teeny This guest establishes a small and simple virtual address space.

vast This guest establishes a genuinely large virtual address space, examines its contents, and then exits.

warp This guest establishes a non-direct-mapped virtual address space, makes a hypercall designed to test hypervisor address translation, and then exits.

fondle This guest manipulates its virtual address space in multiple ways, then exits.

pathos This guest is the Project 2 reference kernel, adjusted to run as a PebPeb guest instead of on hardware. It has been built with a hopefully-useful selection of test programs.

A.2 Guest feature table

Here is a table briefly summarizing the features that various guest payloads use and what is expected to happen when each is run.

Payload	lprintf	magic	exit	print	setidt	iret	dis/en	Timer	Kbd	setpd	adjustpg	U-mode	Result	Source?
hello	Y												loops	Y
dumper	Y												loops	Y
magic	Y	Y											loops	Y
console	Y	Y	Y	Y									77	Y
station				Y	Y		Y	Y					0	Y
tick_tock				Y	Y	Y	Y	Y	Y				loops	Y
gomoku				Y	Y	Y	Y	Y	Y				0	No
flayrod					Y								`0`	Y
cliff					Y								`0xdeadcode`	Y
teeny	Y	Y								K			crashes	Y
vast	Y		Y	Y						K			99	Y
warp	Y		Y	Y						K			`0xfaded`	Y
fondle (v2)	Y		Y		Y	Y				K,U	Y	Y	1	Y
pathos	Y		Y	Y	Y	Y	Y	Y	Y	K,U	Y	Y	0	No