SMP_4 : Symmetric Multiprocessing for Pebbles

15-410 Operating Systems

April 13, 2013

Contents

1 Introduction

Symmetric multiprocessing ("SMP") makes it possible for a system to take advantage of multiple processors to get work done on behalf of applications. On a multi-processor machine an application's threads run genuinely simultaneously, instead of merely being interleaved from time to time by timer interrupts.

This semester, the "special Project 4" consists of adding symmetric multiprocessing to your Pebbles kernel. Though this may sound daunting, the course staff has prepared the necessary infrastructure support, and we will allow certain simplifications which should save you some time.

1.1 Overview

This document consists of the following parts:

- An overview of Intel's hardware support for SMP
- A summary of the provided base code
- Attack plan, including suggestions

This document does *not* contain all information necessary to complete the project. In particular, it will be necessary to carefully study the lecture material and documentation in some source files we provide you with.

Good luck!

1.2 Grading guidance

- Be sure to review the design guidance provided in Section 2.2 and especially the locking-related guidance in Section 2.2.5.
- While it is obviously important that your kernel runs well on multi-processor machines, that should not mean it fails on single-processor machines. Your kernel should "just work" if `smp_num_cpus()` returns 1 and also if `smp_init()` fails. Ideally your kernel will work not only *correctly* but also *reasonably* on single-processor machines.
- Some points will be deducted if your kernel doesn't build and boot correctly "out of the box."
- We will carefully read your README, which will represent at least 10% of your final project grade. In particular:
    1. Please be sure to discuss your old and new locking architectures.

2. You should provide us with a brief analysis of what you believe the longest spin-wait cases are and how reasonable you believe they are.
3. You should also be sure to discuss any changes you made to TLB management.
4. What was the hardest problem you were able to solve? What unsolved problem are you most frustrated about not having been able to solve?
5. Do you have any suggestions about how the SMP base code should be changed?

1.3 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 `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 Hardware Support for Multiple Processors

The core idea of a multi-processor system is that multiple processors share access to memory and I/O devices. Figure 1 below depicts a system with 2 processors and a standard set of PC I/O devices. In this system, one thread of a multi-threaded application might be on CPU 0 running user-space code to compute the next frame of a movie while another thread might be on CPU 1 in kernel mode copying the previously-generated frame to the graphics adaptor. In this figure, the processors are connected to the North Bridge (the root of the I/O system) via the thick bus, as you would expect. This bus is essentially the same as would be used in a single-processor system. In Intel multi-processor systems, processors are also connected by a second bus which is used to communicate information relevant to multiple processors; this bus is the thinner line at the top of the figure.

2.1 Intel MultiProcessor Specification

The Intel MultiProcessor Specification ("MP Spec") describes how to construct a multi-processor system using Intel processors. The major concepts are these:

1. When a multi-processor system is powered on, logic on the motherboard "elects" one processor to be the "bootstrap processor" ("BSP"). This processor is responsible for

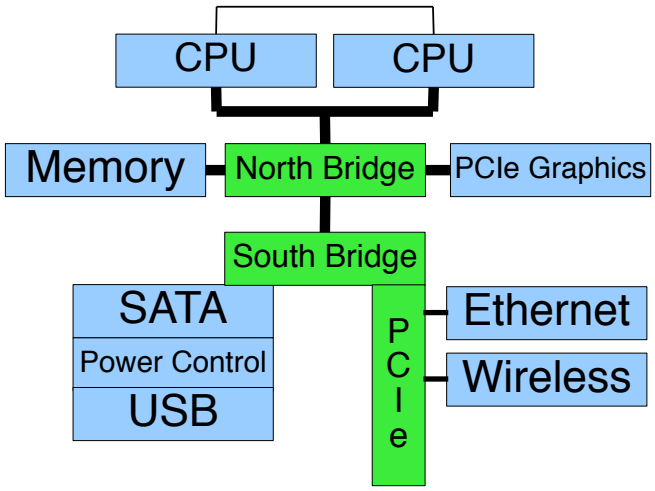

Figure 1: A 2-CPU System

running the BIOS tests, locating the operating system, and booting it. During this time the other processors, called "Application Processors" ("APs"), are halted. If you ran your Project 1, Project 2, or Project 3 code on the dual-processor crash box in the GHC 5 cluster, the BSP was doing all the work and the AP was doing nothing at all.

2. Each processor includes a co-processor called a "local advanced programmable interrupt controller" ("local APIC"). The local APICs are memory-mapped devices; each CPU communicates with its personal local APIC by making memory accesses to a certain physical page. Even though the address of this page is a system-wide constant, each CPU is contacting a different local APIC when it accesses that physical page. The local APICs communicate with each other and with the system's master interrupt controller, called the I/O APIC, over the APIC bus, formally known as the "Interrupt Controller Communication Bus." See Figure 2 below.

3. An MP-aware kernel can detect that the BSP it is running on has APs available. If it wishes, it can launch one or more of the available APs. Each AP begins execution in a primeval 16-bit mode which can access only one megabyte of memory, but through diligent game play can acquire enough experience points to be awarded 32-bit protected-mode execution, virtual memory, etc.

4. Some system resources have one copy per processor. For example, each processor has its own `%cr0`, so CPU 0 might have paging enabled while CPU 1 has it disabled. Likewise, each processor has its own `%cr3`, so their virtual-to-physical mappings can be entirely different (though typically they will agree to some extent). Because each processor has its own `%eflags`, some processors may be ignoring interrupts while others are accepting them.

5. I/O devices are not cloned: there is still only one keyboard controller, one graphics adaptor, etc.

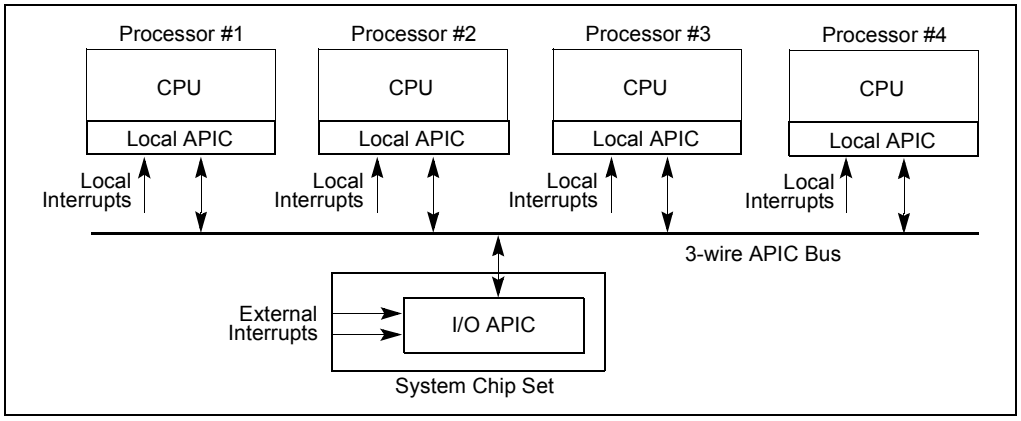

Figure 2: Layout of the the Local APICs and the I/O APIC

Note: When processors refer to each other it is in terms of "local APIC ID numbers." It would seem logical that in a system with N CPUs the local APICs would be numbered 0 to N-1, but in the real world this is not the case. For example, when running Simics with a 4-processor configuration, the local APICs are assigned IDs 0, 1, 6, and 7. Real hardware behaves this way, too. For the sake of convenience, we provide code that maps between these APIC IDs and "logical" CPU numbers. The `smp_get_cpu()` function returns the current CPU number.

2.2 Multi-processor Kernel Design Notes

Most of your uni-processor kernel code should work with little modification in a multi-processor world. However, some kinds of code will need to be re-thought because implicit underlying assumptions will no longer be true.

2.2.1 Interrupt disabling is CPU-local

The `disable_interrupts()` and `enable_interrupts()` functions manage interrupts *only on the invoking processor*. If one processor executes `disable_interrupts()`, it will temporarily not receive interrupts, but all other processors will continue to be eligible to have interrupts delivered to them. If there are places in your kernel where you disable interrupts during a code sequence and expect that as a consequence no other threads can execute that code sequence, you will need a new approach to ensuring exclusive access to those critical sections.

In Project 3 you were forbidden to spin in a tight loop awaiting an interrupt, because this is vastly unproductive. This restriction still continues to be generally true. However, in a multi-processor system, it may make sense for a processor to spin for a *brief, bounded*

period of time *while another processor finishes some particular work*. Note that when multiple processors wish to execute code protected by such a scheme, only one processor, running one thread, can make any progress at a time, so care should be taken that such a scheme is used only when other solutions are not viable. Depending on the situation, it may even be reasonable for a processor to spin *with interrupts disabled*—as long as there is a good justification, the period of spinning is brief, and there is good reason to believe that spinning is rare. Also, as a special case, you are *allowed* to spin while calibrating the LAPIC timer (Section 2.4.2). Finally, note that any time there are more CPUs than runnable threads some CPUs will have nothing useful to do. Real systems use elaborate mechanisms to shut down CPUs which seem likely to be unused in the immediate future. We will not require you to do anything along these lines, so it is ok with us if CPUs spin for a while if there is genuinely nothing useful for them to do.[1]

2.2.2 Per-processor TLBs

Because at any given time the processors in a system may be executing code on behalf of different address spaces, the translation lookaside buffer (TLB) is per-processor hardware. This means that setting `%cr3` and using the `INVLPG` instruction flush TLB entries only on the processor invoking these operations.

If code running on one CPU needs to flush a TLB entry, it is possible that the same invalid entry is present in the TLBs of other CPUs. To ensure that threads running on other processors do not incorrectly modify memory that is no longer in their task's address space, you will need to set up a system for ensuring that all processors executing in an address space take appropriate notice when a previously-valid mapping changes or is removed. There are multiple reasonable approaches, which probably involve some mixture of warning other processors that a change has occurred and delaying until other processors have become aware of a change.

Intel processors automatically handle TLB misses (by creating a new TLB entry) when a valid page mapping is found. Because of this, adding new mappings to an address space inherently requires less coordination with other processors than removing or changing an existing mapping.

2.2.3 "Who am I?"

Lots of kernel code needs to know which thread it is running on behalf of. A uni-processor kernel can handle this by declaring a single global "current thread" variable, but that approach clearly won't work for a multi-processor system. Somehow you will need a way for kernel code running on a particular processor to fairly cheaply determine which thread (e.g., the address of its thread control block) is currently running on that processor. One approach leverages the fact that the binding between threads and processors is defined by the scheduler. Alternatively, you may wish to re-visit the thread-library writeup's "efficient

[1] That said, there may be creative ways to do something useful even if no user threads are runnable... consider this an optional challenge if you wish.

`thr_getid()` challenge"—you may find that a solution to that problem is relevant to this one.

Note that the `smp_get_cpu()` function is "dangerous" with respect to the scheduler: if a thread invokes this function and is told which processor it is running on, it must finish using the answer before it is moved to another processor!

2.2.4 Selected structural implications

The above issues and others will require you to go over your code base and look for implicit assumptions which will be invalidated the instant your kernel activates its first AP. Below we list a few higher-level issues you will need to consider.

- Your locks will probably need to be "re-optimized." For one thing, the lock code itself may well behave in ways that are correct but silly on multi-processor machines. However, there is a more-subtle issue you should consider as well: some of your kernel data structures may use a kind of lock that is arguably just as good as some other kind on a uni-processor machine, but the "just as good as" relation may no longer hold on multi-processor machines.
- Your scheduler will need to "understand" that multiple threads can be "checked out" or otherwise running (also that multiple processors may simultaneously try to find a new thread to run and/or return their current thread to the scheduler).
- Multiple interrupt handlers can be executing completely simultaneously: CPU 0 can be handling a keyboard interrupt while CPU 1 is taking a timer tick.

2.2.5 Coping mechanisms

Our grading process will consider your locking code, how and when various locking objects are used, and also how you reason about locking, as evidenced by material in your README.

Depending on how your Project 3 kernel did locking, it may be unsafe for multiple CPUs be executing at the same time in some parts of the kernel. The best Project 4 solutions will find and fix these problems by replacing non-scalable approaches with better ones; this will have the side-effect that you will gain additional experience in working with and reasoning about concurrency.

That said, it is possible that in order to complete the project you may need to submit a kernel in which some parts rely on less-than-ideal locking practices. There is a rough hierarchy you may wish to consider.

1. The point of an operating system is to run application code, so there would be little point in "supporting" multiple processors if there were no way for user-space code to take advantage of them. Passing solutions should definitely be capable of running

multiple threads in user space simultaneously on different processors. Note that this includes being able to correctly run multiple threads belonging to a single task.

Kernels which can support multiple threads executing in user space but not kernel space are often said to have a "big kernel lock" or a "giant lock"; if you need to do this for emergency reasons, keep in mind that this is *not* a high-quality approach and should be scheduled for an upgrade if at all possible.

2. "Reasonable" solutions should generally use a carefully-chosen mix of tasteful locking techniques, but we realize that some groups may need to submit solutions including some special-case code related to the scheduler. Try to avoid "special scheduler locking" becoming *too* pervasive in your code base, because there is a risk that most kernel activity could end up serialized on it.

3. "Good" solutions will have either no special cases or a small number with clearly described motivations and reasonable, analyzed costs.

2.3 APIC Inter-Processor Interrupts

A processor has the ability to interrupt another processor by sending an "interprocessor interrupt" ("IPI"). A processor generates an IPI by writing to the "interrupt command register" of its local APIC, which will deliver the interrupt to the target processor's APIC over the APIC bus. Each IPI can be sent to one target processor or to a group of processors. We provide utility functions to make sending IPIs more straightforward. See Section 3.1.4 of this document for more details.

When considering how to use IPIs in your SMP kernel design, you should probably keep these notes in mind:

1. IPIs are interrupts, not messages. One implication is that they don't queue. If CPU 0 and CPU 1 both send an IPI to CPU 2 "at the same time," CPU 2 will receive only one interrupt. Likewise, if CPU 0 sends CPU 2 two IPIs "back to back," CPU 2 is unlikely to receive two IPIs.

2. The operations we provide, such as `apic_ipi_cpu()`, enable one CPU to "post" an IPI for another CPU. However, the target CPU may not respond to that IPI right away—for example, it may have interrupts disabled! The sending CPU cannot assume that just because `apic_ipi_cpu()` has returned this means the target CPU has completed—or even begun—responding to the IPI. If you consult Intel's APIC documentation, you will see that there are mechanisms for a sending CPU to find out whether a target CPU has received an IPI, and you may use those mechanisms if you wish.

All in all, keep in mind that IPIs are a tool which can be part of solving a problem, not themselves a solution. They are good for "getting the attention" of a CPU, but you will probably need additional logic before and after that point. Finally, there is at least one approach to the problem which does not use IPIs at all!

2.4 The APIC Timer

The venerable programmable interval timer (PIT) which you have used in your Project 1 and Project 3 code is a system-wide resource, i.e., there is only one PIT even when there are multiple CPUs. While it is possible to program the I/O APIC to distribute interrupts from the PIT across multiple CPUs, this is not only delicate but would also be counter-productive. For one thing, the I/O APIC considers an interrupt to be delivered once some single CPU is picked to handle it. For scheduling purposes, we generally want *every* CPU to be interrupted at the same rate, so it is counter-productive that the I/O APIC arranges for each clock tick to be delivered to exactly one CPU chosen semi-randomly. Furthermore, to reduce locking contention on the scheduler's data structures, we would not want the I/O APIC to "broadcast" a clock interrupt to every CPU at the same time.

Because of these concerns, the Intel MP designers arranged for each CPU in a multi-processor system to have its own independent interval timer. Because each CPU has its own local APIC, it made sense for these timers to be local APIC features.

Our base code uses a simplified interrupt delivery model in which system-wide devices connected to the legacy "PIC" interrupt controller are handled by CPU 0; this includes the legacy PIT and the keyboard. Your kernel *could* handle scheduling by taking PIT interrupts on the BSP and "re-broadcasting" them to the APs, and this may be a useful step while you are developing your kernel. However, a more "professional" approach is to program each processor's local APIC timer so that every processor receives interrupts at the same rate and the interrupts received by the various processors are evenly distributed in time. We will cover the details below in Section 2.4.3.

2.4.1 Registers

Four APIC registers are relevant to the timer. These entries are defined in `smp/apic.h`.

- `LAPIC_LVT_TIMER` - Specifies the timer mode (`LAPIC_ONESHOT` or `LAPIC_PERIODIC`) and the IDT vector to which the LAPIC timer interrupt will be delivered. For the bit layout of this register, see figure 7-9 on page 248 of `intel-sys.pdf`.
- `LAPIC_TIMER_INIT` - Contains the initial counter value. Writing a 0 to this register disables the timer.
- `LAPIC_TIMER_CUR` - Contains the current value of the counter. This counts down from the initial value at the bus frequency divided by the value specified in the `LAPIC_TIMER_DIV` register.
- `LAPIC_TIMER_DIV` - Specifies the frequency divider for the counter, using a defined constant in the `LAPIC_Xnnn` family.

The LAPIC timer essentially has three "inputs": the bus clock rate (unknown, see below), the initial counter value (chosen by you) and the divider (also chosen by you). The two inputs under your control should be chosen so that the LAPIC timer generates

an interrupt every 10 milliseconds (the same as for Project 1 and Project 3). Various combinations of the two inputs you control can result in the appropriate output interrupt rate. One thing to be careful of is that the encoding of the divider value is "odd" (the bits are oddly placed).

2.4.2 Determining the APIC Bus Frequency

The LAPIC timer doesn't count at the same rate as the legacy PIT on the motherboard. In particular, it counts down at a rate determined by the operation of the inter-APIC bus. This rate varies from system to system and could even change on a single system if the BIOS were updated or reconfigured.

To determine the bus frequency, we will use a clock source operating at a known frequency: the PIT. The BSP's APIC timer should be configured in periodic mode with the desired divider. The PIT should then be configured to generate interrupts as normal. After setting the APIC timer's initial count to a large value such as `0xffffffff`, use the timer ticks generated by the PIT to wait for some reasonable amount of time (say, around 100ms). After this time has elapsed, read the current count from the APIC timer. Using the difference between this count and the initial count, we can determine the bus frequency and initial count can be chosen to interrupt the processors at the desired frequency. At this point, the PIT can be disabled and the APs can be booted.

2.4.3 Initializing the Timers

Each AP should configure its timer in periodic mode and use the same divider. For the first interrupt, however, the initial counter values should be skewed across the processors: the nth CPU's initial count is n divided by the number of processors, multiplied by the desired initial count. The APIC timer handler should arrange for the initial count register to be reset to the original, non-skewed value the first time a timer interrupt is recieved.

At this point, each processor is set to be interrupted at the same frequency, and the interrupts are skewed so that (ideally) no two processors will receive timer ticks at the same time, thus reducing contention in the scheduler.

3 The Programming Environment

In addition to the programming environment provided by Project 3, we add base code to facilitate multi-processor programming.

3.1 Base code

3.1.1 Reserved Memory Addresses

The base code reserves the page `SMP_INIT_PAGE`, which we guarantee to be 0x2000,[2] as the page containing startup code for the application processors. Your kernel must arrange for this page to be direct-mapped. The base code furthermore reserves virtual address `LAPIC_VIRT_BASE`, which must be mapped to the local APIC, as described in section 3.1.2.

3.1.2 MP Configuration

When the system first boots, the BIOS constructs data structures describing the system's configuration and places them in memory for the kernel's edification. There are, in general, two such structures. One is the "MP Floating Pointer Structure," which is required to exist. This structure will either indicate some default processor configuration, or it will point to an "MP Configuration Table" with details about the system's processor configuration. The in-memory layout of these structures is specified in `intel-mp.pdf`, and described in `smp/mptable.h`.

As part of your kernel's initialization, prior to enabling paging, invoke the `smp_init(mbinfo_t *)` function. Pass `smp_init()` the multiboot struct pointer which was given to you as an argument to `kernel_main()`. The function will return 0 if the system has multiple processors in some supported configuration, and a negative error code otherwise. Once this call completes, you are free to enable paging.

After `smp_init()` has successfully completed you may invoke `smp_num_cpus()` to find out how many processors the current system has[3] and you may obtain the physical address of the local APIC by calling `smp_lapic_base()`. However, you should not invoke most other functions in the base code's SMP library until your kernel has turned paging on and established a translation from the virtual address `LAPIC_VIRT_BASE`, defined in `smp/apic.h`, to the local APIC's physical address. Note that the page-table entry for this translation should have the "cache disable" bit turned on (we want writes to the local APIC's page to hit the device right away instead of lingering in the cache, and we want reads from that page to return the most up-to-date information from the device).

3.1.3 SMP boot code

Once the local APIC is mapped into virtual memory, the other processors in the system can be booted.

The uni-processor base code we provided you for Project 3 assumed a global descriptor table (GDT) with a handful of entries for user-mode and kernel-mode code and data segments. In addition, due to a quirk of x86 processor design, the base-code GDT must include a "task state segment" (TSS), because, as an unfortunate implementation detail,

[2]The MP Specification requires this address to be page-aligned; because the APs begin execution in real mode, the address must also be under 1MB.

[3]The total returned includes the BSP.

`esp0` is a field in the TSS instead of being a processor register. In order for each processor to have its own `esp0`, it must have its own TSS. While it would be possible for a single system-wide GDT to contain a TSS for each CPU, our base code follows an alternate approach in which each CPU has its own GDT; this makes it easy for each CPU to manipulate its own `esp0` because every CPU's TSS has the same segment index and segment selector, which are relative to its personal GDT.

The base code knows how to clone the static part of the GDT, i.e., the code and data segments, for each AP. The base code will also create a valid TSS for each AP. But each AP's GDT will need a segment descriptor which specifies the location and size of the TSS and, of course, various flag bits. You will need to write a function, `uint64_t tss_desc_create(void *tss, size_t tss_size)` which will form and return a segment descriptor for a given TSS as a function of its address and size (there will be no need for your code to examine the *contents* of the TSS, since the descriptor indicates only type, location, and size information).

Figure 6-3 on page 203 of `intel-sys.pdf` specifies the layout for a TSS descriptor. There are relevant defines for segment descriptors in `x86/seg.h`. The Simics commands `cpu`*n*`.print-gdt` and `cpu`*n*`.print-tss` will prove useful for debugging.

When you decide to launch the APs, you will specify a function that they will all execute. This "AP entry-point function" is roughly analogous to `kernel_main()` in that the processor will be in 32-bit protected mode, interrupts will be disabled, and paging will be off. Unlike `kernel_main()`, the entry-point function will receive its CPU number as a single integer parameter. To launch the APs, invoke `smp_boot(void (*entry)(int))` when you are ready.

Because the base code configures all the processors to share a single IDT, there is no need for the APs to "re-register" IDT entries.

3.1.4 APIC Operations

Base code is provided to read and write to the local APIC, declared in `smp/apic.h`. Functions are provided to read and write APIC registers, and utility functions are provided to handle interprocessor interrupts.

The `uint32_t lapic_read(int reg)` function will return the 32-bit value of the specified APIC register. Similarly, `uint32_t lapic_write(int reg, uint32_t data)` will write to an APIC register.

In order to receive inter-processor interrupts, first pick an appropriate interrupt number (which must be above `IDT_USER_START` as defined by `x86/idt.h`). We have reserved some slots for your use as indicated by `spec/syscall_int.h`. Next, install an IDT entry[4] for that interrupt number. Then you can transmit a specific interrupt number to a specific CPU with `apic_ipi_cpu(int cpuno, uint8_t vector)` or broadcast a specific interrupt number to all CPUs (except that of the invoker) with `apic_ipi_others(uint8_t vector)`. Because the target CPUs will receive just an interrupt, without any associated message-like

[4]Trap gate, interrupt gate, task gate—your decision!

information, you may wish to use multiple IPI vectors for various purposes.

Finally, each CPU receiving an IPI should acknowledge it using `apic_eoi(void)`. This will enable the delivery of future IPIs to that CPU from its local APIC.

3.2 Testing with Multiple Processors

As is our tradition, we will grade your code using Simics rather than randomly-selected pieces of hardware, so whether your code runs on the crash box or not is up to you. It is probably unwise to turn a uni-processor Pebbles kernel which has not run on hardware into a multi-processor kernel and then try to debug it on hardware: debugging a smaller, simpler, and less-concurrent code base will almost certainly be significantly easier!

3.2.1 Simics

If the value of the `SIMICS_CPU_COUNT` environment variable can be converted to an integer, Simics will attempt to run with this many CPUs. The simulator may not necessarily be well-behaved for CPU counts less than 1 or greater than 16.

When Simics is running with multiple processors, it will contain objects `cpu0` through `cpu`*n*, and corresponding TLB, APIC, and Memory objects for each of them. Many simics commands (e.g., `pregs` and `break`) will attempt to use the "current" processor. You can invoke the commands on a different CPU by explicitly specifying the relevant processor object, for example `cpu3.pregs`. You can change the current CPU under default inspection by using the `pselect` command. Some symbolic debugging commands may not work without an explicit CPU.

3.2.2 The Crash Machine

The crash machine is equipped with two 400 Mhz Pentium II Xeon processors. Please see `https://www.cs.cmu.edu/~410/doc/crashbox.html` for details in testing your kernel on the crash machine.

4 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. **STOP AND THINK** about what parts of your Project 3 kernel will be wrong once you start up your second processor, and about how you will fix them. You will wish

to refer to Section 2.2.4 for guidance on what to watch out for. It is probably *very important* for you to review your code base and write down a list of these structural issues *before* you start writing code.

2. Read through new material.

3. Review the P4 lecture material.

4. Read Intel documentation on the TSS and GDT.

5. Invoke `smp_init(mbinfo_t *info)` and arrange for your VM system to map in the local APIC located at physical address `smp_lapic_base()`.

6. Write the `tss_desc_create()` function. The defines in `x86/seg.h` and the documentation on page 203 of `intel-sys.pdf` will prove useful. We provided you with a non-functional template version of `tss_desc_create()` in `kern/smp_glue.c` in the Project 3 tarball; if you deleted it when you started Project 3, you will want to un-delete it. Before you start editing the file it would probably be wise to delete the obsolete comment block at the top.

7. Write a very simple entry point function for the APs to run. For example, this function might `MAGIC_BREAK` once and then spin forever.

8. Boot the APs and confirm that they begin running your entrypoint code.

9. Extend your AP entrypoint code to enable paging and interrupts.

10. Send a simple IPI to test functionality.

11. Decide how you will handle removing or changing virtual-memory mappings. This part of the design is not easy, in part because there are multiple solutions which trade off various factors such as efficiency and code complexity. You may wish to begin with an easy-to-code approach at first and consider upgrading it later.

12. Initially, so you can quickly get to the point where you can schedule multiple CPUs, hack together a "broadcast timer" implementation in which the BSP receives legacy PIT interrupts and "forwards" them to the APs using IPIs. To reduce contention, it might be helpful to send such an IPI to one processor at a time. Doing this is an ugly hack—every interrupt must cause the BSP to stop running user code, as well as preempt some other processor. So you will wish to upgrade this code, if time permits, as described in Section 2.4.3.

13. Test!

14. Schedule a thread on one processor, then interrupt it and move it to a different one.

15. Consider what will happen if some processors are idle. Suppose there are two threads to schedule on four processors. What should the other processors do?

16. Schedule threads on multiple processors.

17. Fix the `halt()` system call so it stops running user code on *all* processors.

18. Rewrite your timer code to use the local APIC timer. You will probably wish to rewrite your AP startup code so that it shuts down the PIT once you have used it to set up the local APIC timer. One approach is to reconfigure the PIT from the familiar `TIMER_SQUARE_WAVE` mode to `TIMER_ONE_SHOT` mode; this should cause it to issue one final interrupt and then fall silent.

19. Test for any regressions!

20. Clean up code. Make sure documentation is up-to-date.

21. Celebrate!