

# Inside Coyotos

Version 0.1

Jonathan S. Shapiro, Ph.D.  
*The EROS Group, LLC*

September 10, 2007

Copyright © 2007, The EROS Group, LLC. Verbatim copies of this document may be duplicated or distributed in print or electronic form for non-commercial purposes.

THIS GUIDE IS PROVIDED "AS IS" WITHOUT ANY WARRANTIES, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

# Contents

<b>Preface</b>	<b>iii</b>
<b>I A Working Tour of the Kernel</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Why This Approach . . . . .	1
<b>2 Peculiarities of the Coyotos Design</b>	<b>3</b>
2.1 Coyotos is a Microkernel . . . . .	3
2.2 Coyotos is a Capability System . . . . .	4
2.3 Coyotos is Persistent . . . . .	4
2.4 Coyotos is Transactional . . . . .	5
<b>3 From Power-On to kernel_main</b>	<b>7</b>
3.1 Postcards from the Boot Loader . . . . .	7
3.2 Zeroing the BSS Region . . . . .	8
3.3 Processor Identification and PAE Mode . . . . .	8
3.4 Primordial Kernel Map, Switch to Linked Address . . . . .	9
3.5 Transition to C . . . . .	9
3.6 kernel_main: Starting the First Process . . . . .	9
<b>4 Architecture-Specific Initialization</b>	<b>11</b>
4.1 Console Initialization . . . . .	11
4.2 CPU Construction . . . . .	12
4.3 Memory Management . . . . .	12
<b>5 Memory Map Management</b>	<b>13</b>
5.1 The IA-32 Physical Memory Map . . . . .	13
5.2 The PhysMem Allocator . . . . .	14
5.2.1 Initial Setup . . . . .	14
5.2.2 Allocating Physical Memory . . . . .	15
5.3 The Kernel Virtual Map . . . . .	15
5.3.1 Theory of Operation . . . . .	16

5.3.2	The Kernel Map . . . . .	17
5.4	The TransMap . . . . .	18
5.4.1	Purpose of the Transmap . . . . .	18
5.4.2	Lazy TLB Consistency . . . . .	19
5.4.3	Data Cache Locality . . . . .	19
5.4.4	Other Implementations . . . . .	19
5.5	Implementation: Configuring Memory . . . . .	19

# Preface

These notes are an attempt to document the Coyotos system and its theory of operation "from the inside". The approach is to walk through the kernel from power-on through capability invocation, tracing what the kernel does at each step.

I'm sure that these pages will get printed. For many readers (myself included) that is the best way to read a long, basically linear document. However, the book is an evolving document. It's official home is at <http://www.coyotos.org>.



## **Part I**

# **A Working Tour of the Kernel**



This part describes the internals of the Coyotos kernel, beginning at power-on and proceeding through the first inter-process communication operation.



# Chapter 1

## Introduction

These notes document the internal structure of the Coyotos microkernel. They proceed by “visiting” the system from power-on to execution of the first user instruction, and continue to consider the first kernel capability invocation and then the first invocation of a server-implemented capability. The approach is loosely inspired by a one-week lecture series on the Multics system given at MIT by Elliott Organick. Each day he opened his seminar with “Today, we will run the first user-mode instruction, but before we can do that...” The first day started with turning on the power. By the end of the week the lecture series had covered essentially all of the Multics hardware and its operating system kernel.

The “follow the code” approach gives a very different feel for a system than the conventional operating system textbook. It is closer to the approach used in John Lions’ *Commentary on UNIX 6th Edition, with Source Code*. Mainly, the order of presentation works out a bit differently. We will see if it is better, worse, or merely different.

Coyotos is actively under development.. In consequence, this book will quickly become out of date. If you find an issue of that nature, please add comments to the pages identifying the problem, and if possible, the fix. Also, please do not hesitate to comment on any points that confuse you.

As with any microkernel, Coyotos incorporates a fair amount of machine-dependent code. In order to walk through the system, we need to choose an architecture. With some hesitation, I have chosen to describe the internals of the IA-32 (i386) implementation. I have chosen IA-32 because it will certainly be the most common architecture on which Coyotos is run. I hesitate because IA-32 is a complicated architecture, and there are places where this will force me to diverge into discussion of IA-32’s (alleged) features. There is no such thing as a free lunch.

Where appropriate, I will also interject discussion about considerations for non-IA32 architectures.

To give a sense of scale, Coyotos remains a relatively small kernel. Coyotos remains a fairly small kernel. At this writing, the IA-32 implementation is 12,956 source lines of code, of which 4,738 lines are machine-dependent. Of those, 615 are assembly lines. There are people who say that human programmers can keep roughly 10,000 lines of code in their head, and beyond that things get too big to understand. This, of course, is a rough number that is heavily influenced by code organization (is it well-modularized?), idiomatic conventions (assertions and “boiler plate” do not count), the complexity of code interdependencies, and the tool used to count the lines. Empirically, the current Coyotos kernel is of a size that mortals can comprehend.

### 1.1 Why This Approach

Why is this an interesting — or even useful — way of looking at an operating system? Operating system textbooks have come to have a fairly well-established structure. What is better or different about this one?

Let me answer with an example. Every conventional operating system textbook includes a chapter on memory management — which is to say, memory allocation policy. If you look at the better textbooks, you will find that they all end up saying the same thing: the buddy system is the best memory allocator to use. I defy you to find even one production kernel that uses a buddy system allocator for kernel memory. These days they all use something called the slab allocator, but even before that nobody used the buddy system. So why did the textbooks say this obviously wrong

thing? Well, the buddy system really is the best allocator, *provided* you are handling an infinite stream of allocation requests consisting of random allocation sizes. What's wrong with this picture is that operating systems have a small, well-known set of *particular* sizes that they allocate, and they allocate lots of those. The buddy allocator claim is right, but for an irrelevant set of assumptions. When you have actually built an operating system, or looked at it from the inside, it is perfectly apparent that this assumption is irrelevant.

There are some lessons to take away from this:

1. Always ask what the assumptions are behind a conclusion.
2. Always reality-check the assumptions.
3. Don't believe that a statement is right just because it comes from somebody famous. In that case, the culprit was probably Avi Silberschatz. Avi certainly *has* built real operating systems, but he sometimes lets his understanding of theory override what he knows from reality.

As the old joke goes: the difference between theory and practice is that in theory there is no difference between theory and practice, but in practice there is.

Well, enough rambling. On to Coyotos and its code.

## Chapter 2

# Peculiarities of the Coyotos Design

As you read through the Coyotos code, there are some things about its structure that you will need to know. This section briefly documents them.

### 2.1 Coyotos is a Microkernel

Coyotos is a microkernel. There is a great deal of conventional operating system function that is *not* implemented by the kernel. This function is critical to have a complete, functional operating system, and it is certainly critical to understanding the behavior of any Coyotos system as a whole. For the most part, however, there is no reason for this code to run in the hardware’s “supervisor” mode. Supervisor mode allows code to do absolutely *anything*. That is too much power for a large body of code to wield safely. Indeed, the existing Coyotos kernel is teetering at (some would say past) the limit on the amount of code that can safely be run in supervisor mode — simply because it is teetering at the limits of what the human (well, the software developer, which is only approximately the same thing) head can manage at one time.

Some things that you will *not* find in the Coyotos kernel:

- A file system. All management of higher-level abstractions is handled by user-level code.
- Device drivers. All device handling is done at user level. This, by the way, is a debatable design choice. There are some compelling reasons to implement drivers in the kernel, primarily motivated by issues of *copy elimination*. We will discuss this late in the book.
- Security policy. The kernel implements protection primitives (in the form of capabilities), but not security policy. Security policy is implemented by user-level code.
- Networking support. Coyotos has a TCP/IP stack. By now you can probably figure out for yourself where that is implemented...

What you *will* find in the kernel:

- Implementations of the system’s “primitive” objects. By “objects,” we mean objects in the sense of operating systems: memory pages, processes, and so forth.
- The implementation of capabilities, which provides both the primitive mechanism for naming and the primitive mechanism for protection.
- Interprocess communications. In most microkernels this is called IPC. In Coyotos this is folded in to the the capability mechanism.

## 2.2 Coyotos is a Capability System

Coyotos is a capability system. Its basic mechanism for naming and protection is something called a *capability*. In contrast to many other capability systems, Coyotos is a “pure” capability system, because capabilities are the *only* mechanism for naming and invoking objects or services. Conceptually, the kernel implements only one system call: **invoke capability**. In practice, the kernel implements two additional system calls **load capability** and **store capability**. These simulate instructions that general-purpose computing hardware does not provide.

In the literature, you will often find a capability described as an (object, permissions) pair. This is unfortunate, because we want the term “object” to mean something else, and we would like “permissions” to be something that we can relate more directly to programming languages and the lambda calculus. In the programming languages world, an object consists of some representation state and a set of methods that perform operations which optionally reference that representation state. The intuition we want is that a capability names an object in the programming languages sense of the word “object.” Unfortunately, the operating system community is equally sure about what an “object” means. It is a page, a process, a file, and so forth. Both views are right; they simply speak at different levels of abstraction. The problem is that the bridge between these levels of abstraction is the capability. As a compromise, we recommend the view that a capability consists of a (representation-object, method-set) pair. This is impossibly cumbersome to say, so we will often fall back on the (object, permissions) intuition. When we do, remember that *permissions* is really a short-hand for “a set of permitted methods.” Also, be careful about the term *object*. When we speak of “invoking an object,” we mean an object in the programming languages sense. When we speak of a “capability designating an object,” we may mean *either* an object in the operating system sense *or* an object in the programming languages sense. Hopefully it will be clear from context. The important point is to be aware that the term is overloaded.

We have tried various ways to disentangle this term. None were very satisfactory. The confusion of overloading the term *object* seems to be less than the confusion of trying to introduce a new term.

The reason that Coyotos uses capabilities as its primitive mechanism of protection has to do with security: it allows applications to limit the flow of authority in the system. We use the term *permission* to mean “the authorization to directly perform an operation on an object.” We use the term *authority* to mean “the transitive effect that a process might have on a system, including all of the operations that it might perform in the future based on the permissions that it currently wields *or can transitively obtain*.” In most systems, permissions are determined primarily by who is running a program (the *principal*). In capability systems, permission is determined by the capabilities that a process possesses. Because of this, capability system designs are greatly concerned with both the initial arrangement of capabilities and objects and the ways in which that arrangement can evolve. Coyotos is a pure capability system. The kernel simply doesn’t implement any notion of principal at all. All access decisions are made on the basis of capabilities. Possessing a capability is a necessary and sufficient proof of authority to perform the operations permitted by that capability on the object that it designates.

## 2.3 Coyotos is Persistent

While the current implementation does not (yet) implement persistence, Coyotos is designed to be a persistent system. Like its predecessor EROS, Coyotos has the ability to save the entire running state of the system to disk at any time. This creates an inversion of responsibilities in comparison to a conventional kernel. For example, the *process* structure of a conventional kernel is a kernel data structure that lives in main memory. It is owned by the kernel, created by the kernel, managed by the kernel, and reclaimed by the kernel.

In Coyotos, a process is managed by the kernel, but it is a *disk-based* data structure that is created, destroyed, and owned by some user (more precisely: by a storage allocator called by a program). In order to execute a process, the Coyotos kernel “borrows” its state from the disk-based object. At any time, the kernel may be called on to put that state back into its on-disk form so that it can be written out to the disk. This is true for every other major kernel object type as well. This imposes some constraints on how the kernel is allowed to manipulate objects, and on the dependency relationships that it is required to maintain. You will find that every kernel object that has state (some kernel services have no state, but are considered “objects by courtesy”) divides cleanly into two parts: the persistent state, which lives

within the `state` field of the object, and other fields that are used by the kernel to keep track of the object at run time. These other fields are not saved when the object is written out.

Because the Coyotos kernel is required to be able to write out all of the major kernel objects on demand, we often say that Coyotos kernel memory is a cache of the “official” state, which lives on the disk.

## 2.4 Coyotos is Transactional

The EROS system, which was the predecessor of Coyotos, was strictly transactional. Every EROS system call happens as a single unit of operation. Control enters the kernel, resources are gathered, a “commit point” is reached, and the operation proceeds without interruption until it completes. After the commit point occurs, the operation is required to complete or to panic the system. Failures after the commit point indicate a deep kernel logic error. Prior to the commit point, no modifications to externally visible system state are required. The kernel is entitled to load and clear caches before the commit point. It can mark objects dirty (thereby allowing them to be modified after the commit point). It can convert their representation from one format to another. It cannot actually modify them in any semantically observable way.

A corollary to this is that any EROS system call can be abandoned prior to the commit point without any damage to kernel integrity. Think of this as a form of exception handling: any kernel path can “throw” before the commit point, but not after. In practice, locks must be released and the current transaction ID must be abandoned, but that is it. In the EROS implementation, this is accomplished by cleaning up a small number of global variables, re-setting the kernel stack pointer to point to the top of the kernel stack, and branching to a well-known “figure out what to do next” entry point. From a code-reading perspective, there are two implications to this:

- No state should ever go on the stack that requires any sort of cleanup, because if procedure  $f()$  calls procedure  $g()$ , control may never return to  $f()$  and cleanup may never occur.
- The nominal return values of procedures indicate what happens if things succeed. If things fail and the current transaction is aborted or restarted, procedure returns will not happen at all.

A second corollary to this is that any system state can be reconstructed by some purely sequential sequence of system calls and user-mode instructions. That is, the system evolution is serializable. Unfortunately, it is impossible to preserve this model in pure form on a multiprocessor. As an example, consider a string copy. When address spaces are suitably shared, it is possible for a second processor to observe a string that has been partially copied by a first processor. As a result, serializability is lost.

Once the purely serializable transactional model is lost, it is desirable for performance reasons to look for ways to optimize. In some cases, it is good enough to say that the results are not fully defined until the operation completes, and that partial results may appear from operations that *never* complete. We will look at some examples of this in the discussion of capability invocation. Coyotos preserves the EROS model in spirit, but it aggressively exploits these sorts of “partially defined” operations for performance. In each case, it is necessary to specify why the optimizing short cut is safe with respect to the original model, and we will do so. If you catch us *failing* to do so, it is a mistake, and you should ask about it by posting a comment on the appropriate page.

From the kernel perspective, there is an important consequence of a transactional kernel design: there is no per-process kernel stack. When a process becomes blocked in the kernel it does not hold resources. On resumption, it will restart its current system call from scratch. Because of this, it has no state on the stack that needs to be preserved. Coyotos requires a stack for each CPU, but that stack is owned by the CPU, not the process.



## Chapter 3

# From Power-On to kernel\_main

Traditionally, the first instruction of a UNIX binary is labeled by the symbol `_start`. Actually, the leading `'_'` may or may not be present, depending on your linker. In any case, a great deal happens between the time that the hardware powers on and the time we arrive at the start symbol of the Coyotos kernel:

- The BIOS runs, sizing your memory, initializing the system memory controller and interrupt controller (we hope, but not always) to sane states, possibly running the network boot ROM to load your kernel, or possibly loading the first sector of your boot loader from disk (or these days, from a USB key, CF card, or SD card).
- The boot loader (in our case, *grub*) executes, collecting various information from the BIOS that may or may not turn out to be useful later. It is *grub* that establishes most of the initial conditions that apply when Coyotos is started.

Within the Coyotos source tree (`coyotos/src`), the kernel source tree lives in the `sys/` subdirectory. The kernel build makes use of the IDL compiler, so it is necessary to build the `ccs/` subtree before building the `sys/` subtree. Most of what we are about to discuss occurs in the file `arch/i386/kernel/boot.S`. You may find it convenient to have a copy of that ready to hand while you read.

### 3.1 Postcards from the Boot Loader

By the time control reaches the kernel at `_start`, the *grub* boot loader has placed the IA-32 processor in protected mode, and has established 4 gigabyte segment limits for both code and data. Grub has loaded the kernel to its linker-specified load address (on IA-32: `0x100000`, or 1 megabyte). It has also loaded a second file using the *grub* “module” directive. This second file contains the *initial system image*, which is a serialized form of all the objects which exists when the system initially starts running.

At system startup on a multiprocessor, only one CPU has been enabled. The BIOS has elected this CPU as the “master” CPU. Any other CPUs that are present on the system are quiescent. We will refer to this CPU as “cpu0.” Of course, on a uniprocessor `cpu0` is the only CPU that there is.

The situation bequeathed to us by *grub* is delicate. We know that it has established various initial conditions and collected various information. Unfortunately, we don’t know where *grub* has placed this information in memory. This means that we cannot allocate any memory until we figure out where *grub* has put things. In particular, we cannot reserve space for either a kernel stack or an initial kernel memory mapping structure. We know that *grub* has established a map for us somewhere, but the *grub* specification does not tell us where this might be. The Coyotos kernel linker script `arch/i386/kernel/ldscript.S` has pre-reserved space for these things. We need to get out from under these unknowns as quickly as we can, but there are some things that we need to do first.

To make matters really fun, there is some delicacy that we have imposed on ourselves. Long-standing convention on IA-32 is that the kernel resides in the uppermost portion of the address space, typically starting at `0xC000000` (3

gigabytes). Most IA-32 machines don't have that much physical memory, so we cannot load the kernel at that address directly. What we do instead is to *link* the kernel starting at 0xC0100000 (3 gigabytes plus 1 megabyte), but *load* the kernel starting at 0x10000 (1 megabyte). This has implications:

- Coyotos will not run on an IA-32 machine with less than 1 megabyte of memory (or at least, not the way we normally compile it).
- Until we can build a virtual memory map for the kernel, we need to restrict ourselves to purely position-independent instructions, because any attempt to reference an absolute address is going to try to make that reference at the official address above 3 gigabytes, and we don't yet have a mapping there.

At various points in `arch/i386/kernel/ldscript.S` you will see the symbol `KVA` being subtracted from a symbol address. `KVA` is a macro that tells us what the linker's notion of the base virtual address was. Subtracting it allows us to correct from linked addresses to loaded addresses where we need to. Confusing? Definitely. We are going to get out of this confused as quickly as possible, but we need to build a kernel virtual map before we can do that.

## 3.2 Zeroing the BSS Region

The first thing we need to do is to zero the `bss` region. When applications run, it is the responsibility of the paging subsystem (or the application loader) to make sure that the `bss` region is zeroed. A kernel must do this for itself. Further, we want to do this before we touch any global variables, because some of those variables may live in the `bss` section. If we write to them and *then* zero the `bss` section, we will end up zeroing anything we have written. To zero the `bss` region we need to use the `%eax` register. Unfortunately `grub` has left some information there that we do not yet want to lose, so we need to move that value out of the way first. Immediately after clearing the `bss` area, we can save the information provided by `grub` into global variables (being careful to make the `KVA` adjustment). We will use this information later.

## 3.3 Processor Identification and PAE Mode

The beauty of the IA-32 family of processors is that there are so many variants to choose from. Later processors support features that earlier machines do not. There are four features that we want to use if we can:

- Large page support will allow us to map the kernel with a much smaller number of TLB entries. This has a very significant impact on kernel performance, so we want to use this feature if it is available.
- Global page support allows mappings for the kernel to remain resident in the TLB across TLB flushes. This also has a significant impact on kernel performance.
- Later processors implement a “no-execute” hardware protection. The IA-32 family supports *two* memory mapping models: legacy (the original two-level scheme) and PAE (a three-level scheme). The no-execute feature is only available in PAE mode, and then only on later processors. Unfortunately, it is not simple to switch between legacy and PAE mode later, so we need to figure out early whether PAE mode is available and use it from the beginning.
- We want to enable the extended hardware debugging features if present. Later, this will allow us to use hardware watchpoints and breakpoints for debugging.

To figure out which subset of these features is available, we need to do a moderately convoluted exercise in processor feature identification. IA-32 processor identification is now a subject for a long Intel technical note. At this stage, we want to do the bare minimum. If you are following along in `boot.S`, the instruction marking the end of this sequence is the `jnz setup_pae_map`. At that point, we have made a decision about the availability of the PAE features, and we will proceed to build the appropriate type of mapping table.

Note that the current startup code enables PAE mode without checking whether the no-execute (NX) feature is present on the hardware. This is a bug. The PAE mapping mode is slower and more complicated than the legacy mapping mode. While it *does* provide some new features, none of these are used by Coyotos, and we probably should not be enabling PAE unless we are actually going to get some value out of it.

### 3.4 Primordial Kernel Map, Switch to Linked Address

Now that the processor has been identified, we can begin weaning ourselves from the initial configuration provided by *grub*. The first step is to build a memory map of our own at a known location. The kernel has pre-reserved space for a single page table, a single page directory, and a page directory page table (PDPT). The last is used only in PAE mode. We will initialize a full page table of mappings. In the legacy mapping mechanism, this will map the lowest 4 megabytes of memory. In the PAE mapping mechanism, this will map the lowest 2 megabytes of memory. We will construct two aliases for this region: one at 0x00000000 (which is where we are presently running) and the other at 0xC0000000 (which is where we are linked). In the process, we take care to ensure that the text-mode console frame buffer is mapped non-cacheable, so that writes to that area will appear on the physical display immediately. Once the map is build, we load it onto the processor and enable the virtual address translation hardware.

This is a very sloppy mapping. We have no idea whether the machine actually has 2 or 4 megabytes of memory, but we are going to be re-building this map later in any case. The reason we want this early mapping is so that we can start running at our properly linked address.

Once the mapping is established, we perform a jump register to `drop_low_map`. Note that for the first time we do *not* correct the destination address by subtracting KVA from it. The purpose of this jump is to change the program counter to be running at our proper virtual address. Having done so, we remove the low-memory mapping at virtual address 0x0, reload the hardware mapping table pointer to flush the hardware TLB, and re-initialize the kernel stack pointer. After all of this we can *finally* call C code!

Eventually, the primordial kernel map will serve as our fall-back mapping structure — the one we use when we cannot identify an appropriate user-mode mapping to run. This mapping will contain only kernel mappings.

### 3.5 Transition to C

Having established a mapping structure that allows us to do so, we transition to C code by calling `kernel_main()`.

### 3.6 kernel\_main: Starting the First Process

In application programs, the `main()` procedure surrounds the entire execution of the program. If `main()` exits, so does the application. This is not true in many kernels, and it is not true in Coyotos. Operating system kernels are event-driven systems. They respond to interrupts and exceptions (system calls are generally a special form of exception), do something, and return control to some user process. Because of this, the role of the kernel's equivalent to `main()` is to get the system initialized and start the first process.

In Coyotos, we do not use the traditional function name `main()` because its signature is not appropriate. The `kernel_main()` procedure accepts no arguments and returns no results — in fact, it never returns at all. If you look at the code in `sys/kernel/kern_main.c` you will find that the last action is to call `sched_dispatch_something()`. As we will see in detail later, the `sched_dispatch_something()` procedure never returns. If necessary, it will run an idle loop until some process is ready to run. In practice, the call from `kernel_main()` is the *first* call to `sched_dispatch_something()`, and the initial system image contains many processes that are ready to run.

#### Note

When you see a statement like “the initial system image contains many processes that are ready to run,” you should immediately ask: well, what if it doesn't? Perhaps the initial system image is broken. How

does the system behave if there is nothing to do? Is this statement a requirement, or does it merely describe the expected case?

In fact it is not a requirement. If there are no processes to run, the call to `sched_dispatch_something()` will simply idle the current CPU and loop forever. Arguably, the `kernel_main()` procedure should complain in this unlikely case. The existing behavior is correct, but without a diagnostic it may be mysterious.

The `kernel_main()` procedure calls, in sequence, `arch_init()`, `obhdr_stallQueueInit()`, `cache_init()`, and `arch_cache_init()` before calling `sched_dispatch_something()`. These procedures respectively perform architecture-dependent initialization, initialize the object stall queues, set up the data structures for the kernel object cache, give the architecture-specific code a chance to do additional cache setup, and then dispatch the first process. We will need to look at each of these steps in turn, but first we need to pause to describe some of the internal logic of the Coyotos kernel.

## Chapter 4

# Architecture-Specific Initialization

By the time it is completed, the architecture-specific initialization routines will do several things:

- It will initialize the console, if any, for output.
- It will determine what physical memory is present, and where it is found.
- It will determine how many processors are attached to the machine, and initialize them.
- It will construct a proper virtual memory map for the kernel to use.
- It will set up the necessary data structures so that interrupts, system calls, and exceptions can be processed.
- It will reserve space for use as application page tables and page directories.
- When all this is done it will enable interrupts for the first time.

While most of this activity is *extremely* architecture dependent, the same basic requirements must be satisfied for each architecture, and there are a surprising number of common elements in the management structures for these activities.

All of these actions are driven by procedure calls made from the `arch_init()` procedure in `arch/i386/BSP/multiboot/bsp_mem_config.c`.

### 4.1 Console Initialization

The *very* first action taken by the architecture-specific initialization code is to initialize the console logic. This subsystem is responsible for the actual display of characters that are passed to the kernel `printf()` and `fatal()` functions. The implementation of `printf_putc()` in `kernel/kern_printf.c` calls two functions: one puts each output character into a ring buffer containing all diagnostic output. The other emits the same character to the display if one is available. On machines that have an attached display, we would like to be able to see what is going on as early as possible. On machines that do *not*, we would at least like to be able to store those messages into a holding buffer so that they can be retrieved by a hardware debugging probe. Both of these are good reasons to enable the console logic as early as possible.

#### Implementation Deficiency

The architecture-dependent configuration parameters include a C preprocessor macro `HAVE_HUI` that is intended to indicate whether any sort of human interface exists on the platform. If none exists, the entire implementation of `printf()` is compiled out.

This is excessively draconian, since the circular ring buffer can usefully be read using an in-circuit emulator or debug monitor probe even if no display is present. The current implementation arguably ought to implement the ring buffer even when `HAVE_HUI` is not defined.

On IA-32, the console implementation relies on the fact that the legacy MDA graphics adaptor (the text-mode display) is memory mapped from 0xb8000 to 0xb8fff. Note that this address range falls within the 2 megabyte initial memory region that has been mapped for us by the bootstrap code. Because of this, the IA32 console implementation is actually very simple. All it needs to do is write new characters to the memory region with an appropriate background and foreground color and deal with scrolling when needed. The bootstrap code in `boot.S` has mapped the display region as non-cacheable, so we do not need to worry about update delays that might result from use of the hardware's write-back cache.

A tricky point arises here that might not be obvious at first. A goal of the Coyotos design is for drivers to execute as application code. This means that we will later have some application that believes it owns the display. This application may write to display memory, or it may even change the display mode in such a way that kernel writes in the 0xb8000-0xb8fff region are meaningless (or even harmful). The kernel must be careful to avoid getting in the way of the console management application. We ensure this by means of the `console_detach()` function. This is called just before the process scheduler is called from `kernel_main()` to dispatch the first piece of application code. Once `console_detach()` is called, `console_putc()` will simply do nothing when it is called. In consequence, the kernel will no longer update the display and the application-level console manager will be able to manipulate it without worry of interference.

As a very transient measure, there is a bring-up expedient `console_shrink()`. This is an early debugging tool in which the kernel and the console management application divide the screen by hard-coded agreement. It is the kind of mechanism that is very helpful during early system bring-up, but tends to do more harm than good in the long term. These sorts of “hidden contracts” between the kernel and external code are very easy to forget later.

## 4.2 CPU Construction

The next action taken by `arch_init()` is to construct the per-CPU data structures. At this point we don't actually know how many CPUs there are, and we haven't started any other CPUs, so strictly speaking we only need to initialize the data structures for CPU0 (the bootstrap CPU). In fact, we cannot even fully initialize CPU0, because we haven't yet fully determined the feature set of this CPU.

Unfortunately, we need one field of the CPU structure initialized early: the lock word field. This is needed because we will soon start calling the physical memory manager, and the physical memory manager uses the locking subsystem. Constructing the per-CPU structures is done here so that the per-CPU lockword field will have a non-zero value.

## 4.3 Memory Management

We now turn to matters having to do with managing the physical and virtual memory map, but these requires enough background information that it is best provided by a separate chapter.

## Chapter 5

# Memory Map Management

The next step of `arch_init()` is to initialize the *transmap*. We have a previously reserved (at link time) page table. We zero it and install it at a known address (`TRANSMAP_WINDOW_VA`). The question is: what is a *transmap* and why do we need one? To answer that, we first need to talk about the kernel's virtual and physical memory maps and our strategies for managing them.

### 5.1 The IA-32 Physical Memory Map

The IA-32 has been through a fair amount of evolution. Early PCs were limited to 640 kilobytes of RAM, so the BIOS ROM was placed just below the 1 megabyte addressable limit. As later machines came to support more memory, logic was introduced in the memory controller hardware to “skip” the BIOS ROM region. Some controllers have that logic, others do not. Other portions of the physical memory map are taken up by the PCI bus and the video subsystem. How much depends on your video card. If you go out and buy two of the latest and greatest video cards with 512 megabyte graphics memories, you are going to devote more than a gigabyte of your *physical* address space to mapping those regions across the PCI bus. In addition, you will lose part of the physical address space to the PCI configuration areas. Install 4 gigabytes of memory on a machine like that and on most hardware you will actually get to use a bit under 3 gigabytes — the rest will be hidden behind PCI card memory.

Or maybe not. As of late 2007, a few of the very latest motherboards have memory controllers and BIOS logic that will relocate that memory to appear above 4 gigabytes. Note that you cannot address that memory unless you are using PAE mode mappings. The problem here is that there is a lot being done behind the scenes by the BIOS as it brings up your machine, and we really don't want to have to fool around with the low-level memory controller chipset if we can avoid it. Fortunately the BIOSes that can do these tricks provide interfaces we can query to find out what happened. We also want to support earlier BIOSes. There are a surprising number of people out there who are still running i486 processors or first-generation Pentiums. Coyotos runs just fine on these systems. We do not support the i386, because it lacks a memory protection feature that we really need.

In any case, the point is that the physical map is not something that we can hard-code. We need to find out how it is laid out from the BIOS, and then we need to take care to allocate physical memory from regions that are actually populated on the current machine. Finally, we want to mark certain physical memory locations as “reserved”, because they contain information generated by the BIOS that we will later want to use from application level. Examples of such locations include the basic and extended BIOS data area, the APM and ACPI configuration data areas, the various locations where pieces of the BIOS ROM are mapped into the physical address space, and so forth. Finally, we want to make sure that the locations where the kernel code, data, and stack live are not allocated later by, say, the kernel heap management code. It would not be good to overwrite the kernel!

To keep track of all this complexity, Coyotos implements a physical memory management subsystem.

## 5.2 The PhysMem Allocator

The Coyotos physical memory manager is implemented by `kernel/kern_PhysMem.c` and `kerninc/PhysMem.h`. The physical memory allocator has three functions:

1. It tracks which portions of the physical address space are *addressable*, given our choice of hardware mapping options. If we are using the legacy virtual memory mapping structure, we cannot address anything above address `0xffffffff` (4 gigabytes) in the physical memory address space. If we are using the newer PAE mapping mechanism, we can address up to `0xffffffff` (64 gigabytes).
2. It tracks which portions of the physical address space are *populated*. That is: where each RAM, ROM, and NVRAM or equivalent device memory region can be found and what type of memory resides at that location.
3. It tracks the kernel's *use* of these memory regions. For example, some RAM will be used for mapping tables, some will be used to store the kernel heap, and some will be used as frames for page objects. Each of these is a distinct type of use that is tracked by the physical memory allocator.

The physical memory allocator is a special-purpose allocator. It assumes that we are subdividing an initially contiguous region (the addressable physical range) into contiguous, non-overlapping subregions (RAM, ROM, NVRAM, and “holes”). A defined region may be further subdivided as uses for it are determined, but — with the notable exception of device memory — the physical memory allocator assumes that these regions will be allocated once and will never be released. The goal is to support an initial partitioning of physical memory, rather than to support a general-purpose allocation scheme. For the same reason, the allocator assumes — again with the exception of device memory — that two adjacent memory regions having the same class and use can be coalesced in the allocation book-keeping structures.

### A Limitation

While there is a paper design, the current implementation does not adequately address the use of hot-plug memory boards. Hot-plug memory is different from other device memory because the use of the memory is general-purpose.

Taken together, these assumptions allow us to implement the allocator very simply. It is stored in a statically allocated array. Regions do not overlap. The vector is kept sorted in order of start address, with unallocated entries at the end of the array. For each region, the allocator keeps track of the memory region *class* and the memory region *use*. The *class* tells us what type of memory exists at a location (ROM, RAM, NVRAM, valid but undefined). The *use* tells us how this memory is currently being used (if at all). For example, `pmu_KMAP` tells us that the memory region consists of pages that define the kernel virtual map. Both of these are documented in the header file. The `pmc_UNUSED` allocation class is reserved to mean that the table entry is not presently being used for anything. Entries in the `pmem.table` are maintained sorted by start address, with all entries of class `pmc_UNUSED` kept at the end of the table. Adjacent entries having the same class and use are coalesced into a single entry whenever possible. The PhysMem allocator tracks both the *existence* and the *allocation* of physical memory.

Device memory (`pmc_DEV`) is an exception to the coalescing rule. Among the initial applications will be drivers. These will inform us about memory regions that were not visible to the BIOS, such as memory on PCI cards or AGP graphics controllers. Some of these devices — notably CardBus adaptors — can be inserted and removed from the system, with the consequence that their memory regions can appear and disappear. If we allowed these regions to be coalesced, we might find that *removing* a memory region from the physical memory allocator's list would require *allocating* one or more tracking structures in order to split an existing region up so that part of it could be converted to a hole. This operation could fail if we run out of entries in the physical memory range table. To ensure that this failure cannot occur in practice, we do not allow device regions to be coalesced.

### 5.2.1 Initial Setup

Initially, the allocator has no knowledge of physical memory structure. Very early in the `arch_init()` procedure we will initialize the allocator by calling `pmem_init()`. This tells it what the addressable physical range is. If

we are using the legacy mapping hardware, we can address up to 4 gigabytes of physical memory. If we are using the PAE mapping mechanism, we can address up to 64 gigabytes of physical memory. The call to `pmem_init()` establishes an initial physical memory allocation record whose class is `pmc_ADDRESSABLE` and whose current use is `pmu_AVAIL`.

Once the addressable range is established, the architecture-dependent initialization code will start to define what portion of the physically addressable range is actually backed by physical memory, where various ROM chips live, and so forth. This is done by performing calls to `pmem_AllocRegion()`, specifying the appropriate memory class and use for each call (see `config_physical_memory()` in `arch/i386/kernel/init.c`). In addition to adding regions for each memory device identified by the BIOS, this code adds regions for the BIOS data area and the extended BIOS data area. Some BIOSes already report these regions. Others do not. If the physical memory allocator is asked to allocate a region that is already allocated with the same class and use, it ignores the second allocation.

Earlier, I stated that *grub* had collected various information for us. We need to allocate storage so that we have a place to put that information, but we would like to avoid the misfortune of allocating that storage from the memory areas that *grub* has set up. We therefore go through and allocate these physical regions as well. We will release them later after we have copied the *grub*-supplied information to a safe place. Dealing with that is the purpose of `arch_cache_init()`.

## 5.2.2 Allocating Physical Memory

The other parts of the physical memory allocator interface support memory allocation (as opposed to definition). In order to be allocatable, a memory region must have class `pmc_RAM` and use `pmu_AVAIL`. The `pmem_Available()` function determines how many units of physical memory having a particular size and alignment and falling between a specified base and bound address are available to be allocated. The caller may optionally specify that *contiguous* units are required. The `pmem_AllocBytes()` procedure can be used to allocate storage that `pmem_Available()` has indicated is available. Regions returned by `pmem_AllocBytes()` are always contiguous. These calls will be used by the kernel heap manager to allocate backing store for the kernel heap.

### Bug

at present, the code in `kern_PhysMem.c` is not multiprocessor-safe, because no locks are taken. This is a bug, because driver code may need to define new physical memory regions later, and those calls may be made after multiprocessing is enabled. The current logic assumes that the entire valid physical memory map can be defined at system startup. Because drivers are not in the kernel, we are unable to determine the requirements of attached cards and peripherals.

By the end of physical memory initialization, we have defined the entire physical map that is available for allocation by the kernel. We may later add device memory, but the kernel should not use that memory for general-purpose allocation. It would be an awkward mistake if, say, GPT structures got allocated from the frame buffer!

### Note

There is an exception to this statement: hot-plug memory. The current implementation does not have enough infrastructure to support hot-pluggable memory cards such as are seen on high-end multiprocessors. Adding support for that is straightforward, but it requires some logic in the physical memory allocator that we do not currently provide.

## 5.3 The Kernel Virtual Map

Now that we have a picture of what is happening at the physical memory layer, we need to look at the *virtual* memory organization. As part of that, we need to look at the “theory of operation” for how Coyotos handles memory mapping in general.

### 5.3.1 Theory of Operation

There are several challenges that a virtual mapping design needs to address. The two biggest ones are virtual crowding and multiprocessing. Neither of these issues is architecture specific.

#### Virtual Crowding

On IA-32, the application owns the `[0x0,0xBFFFFFFF]` region of the address space. The kernel owns, at most, the `[0xC0000000,0xFFFFFFFF]` range (which is 1 gigabyte). On implementations that support the “small spaces” optimization, the kernel may have a bit less than that. Machines with more than a gigabyte of main memory are pretty common these days, so it is obvious that the kernel cannot just map the entire physical memory into the kernel portion of the virtual address space. Actually, kernel virtual memory is very constrained on this platform, because there is a second factor to consider.

An IA-32 machine can have up to 64 gigabytes of physical memory. Very few machines actually have this much, but in principle it is possible. If we want to support a memory this large, we need a book-keeping structure for each of these pages. The book-keeping structure tells us the OID associated with each page frame, the current allocation count, and so forth. The Coyotos per page book-keeping structure (`MemHeader`) is about 48 bytes. 64 gigabytes works out to 16,777,216 pages, each requiring a `MemHeader` structure. While the pages themselves do not need to be mapped into the kernel, the `MemHeader` structures *do* need to be mapped. For a fully loaded system, the `MemHeader` structures alone will require 805,306,368 bytes of virtual memory. The good news is that we have 1,073,741,824 bytes of virtual space available. The bad news is that there are a whole bunch of *other* things that need to go into that space. Realistically, the current Coyotos kernel can cleanly support roughly 32 gigabytes of physical memory or perhaps a bit more. Beyond that we would either need to start using the remaining memory as a disk cache or we would need to run the kernel in its own address space. At present, the implementation simply ignores any physical memory that is above what we can actually handle.

But you begin to see the problem. In addition to the object management structures for pages, we need to keep other kernel objects — processes, GPTs, endpoints, and so forth — in virtual memory as well.

#### Multiprocessing

The other major design concern in designing the memory map is multiprocessing. Can multiple processors use a common virtual map? The answer is driven by several factors:

- CPU-local storage. Each CPU requires some amount of private storage. At a minimum, each CPU needs to be able to build temporary mappings independent of the others, and each requires a separate notion of the “current process.”
- Hardware coupling. On some hardware, sharing memory across processor clusters is expensive. The current implementation is designed to support tightly coupled and closely coupled hardware implementations. An effective implementation for a loosely coupled machine is possible, but it requires a different implementation.
- Hardware bugs. There are bugs in the multiprocessing logic on many IA-32 implementations, and some of them interact with the memory mapping subsystem. In particular, if two processors attempt to update the accessed/used bits of a page table entry at the same time, the results can be entertaining. Fortunately this behavior can be suppressed by software.

Concerning CPU-local storage, there are basically two possible designs:

- All CPUs share the same map. Each somehow retains a pointer to its per-CPU state, all of which is gathered into a single data structure somewhere.
- CPUs have distinct memory maps. These maps agree except for a small number of places that are CPU-local.

Coyotos adopts the “shared map” approach. All CPU’s run out of the same map. Each CPU has a pointer to its CPU-local storage that is maintained at the bottom of the stack.

To be more precise, any CPU can run in any map, and the kernel portion of the map is shared in the sense that all of the page tables for the kernel portion of the mapping are shared in common across all per-process mapping trees.

In the Coyotos IA-32 implementation, each CPU has the following bits of CPU-local storage:

- A per-CPU stack.
- A current process pointer
- A current CPU pointer
- A region of the TransMap that is used exclusively by that CPU. In effect, each CPU has its own TransMap structure, but all of those TransMaps live within a common page table.

To repeat something that I said earlier, Coyotos kernel stacks are per-CPU, not per-Process. This means that the association between a kernel stack and its CPU is long-lived.

#### Note

At present, Coyotos includes the necessary mechanism to declare and reference CPU-local storage, but it does not actually implement the necessary mechanisms to access this storage on a per-CPU basis. The seeds of the support can be seen in `kerninc/ccs.h`, but the implementation is incomplete.

### 5.3.2 The Kernel Map

With all of that as preamble, here is what the IA-32 kernel virtual map looks like in greater detail:

Base	Bound	Start	Description
0x0	0x100000		Flat map of low physical memory. Used only for BIOS access
._start=0x100000	._etext		Kernel code
._extext	._erodata		Kernel read-only data
._syscallpg	._syscallpg	Page boundary	System call trampoline page. This page will be mapped at a published location within upper memory so that applications can use the most efficient system call entry mechanism available on the platform.
__cpu_data_start	__cpu_data_end	Page boundary	Per-CPU data area.
KernPageDir	KernPageDir+0x1000	Page boundary	Kernel page directory
KernPageTable	KernPageTable+0x1000	Page boundary	Kernel page table
TransientMap	._end_maps	Page boundary	Page table for transient maps
kstack_lo	kstack_hi	Page boundary	CPU0 stack
._pagedata	._epagedata	Page boundary	Data region for page-sized data items (presently empty)
._data	._end	Page boundary	Kernel data and BSS
._end	0xFF400000 ( <i>bound</i> )		Kernel heap
0xFF400000	0xFF800000		Additional CPU stacks (not yet implemented)

Base	Bound	Start	Description
0xFF800000	0xFFC00000		TransMap window
0xFFC00000	<i>top of memory</i>		IPC Destination window (not yet implemented)

Everything shown between `_start` and `_end` is determined by the kernel linker script `arch/i386/kernel/ldscript.S`. The reserved regions in the uppermost portion of the map are determined by constants defined in `arch/i386/kernel/target-hal/config.h`.

Of these various locations, only the location of the system call trampoline page is architecturally visible to applications, and we will probably need to rearrange the map when we start to use that. All of the rest can be adjusted without impacting application compatibility.

The table shows a kernel page directory. When we run with the pre-PAE mapping logic, each application will have its own page directory. In pre-PAE execution, the Kernel Page Directory page will be used when there is no per-process directory that is appropriate. In PAE mode, this kernel page directory covers the 1 gigabyte region owned by the kernel. All per-process PDPT structures will reference this directory, and there is a Kernel PDPT that is used when no per-process mapping table is appropriate.

The table shows a single kernel page table. Obviously this will not be enough as the kernel heap grows beyond the 0xC0200000 (PAE: 2 megabyte) or 0xC0400000 (non-PAE: 4 megabyte) boundary. As that occurs, additional pages will be allocated from physical memory to serve as additional page tables. These are not normally mapped into the kernel address space. Coyotos manipulates the kernel mapping tables using the *transmap* mechanism (see below).

Note that the kernel heap is dynamically sized. This presents a potential problem. In non-PAE execution, application page directories are initially created by making a copy of the kernel page directory, thereby incorporating a complete set of kernel mappings. Heap growth can require that an additional page table be allocated to map the heap. This occurs frequently during system initialization, but that is before any per-process page directory is created, so it does not create any problems. Unfortunately, *drivers* can later come along and announce new regions of physical memory to the kernel. These may force the kernel to allocate `MemHeader` structures, which come from the heap. If the heap grows as a result, and a new kernel page table gets allocated, the individual per-process page directories may no longer be up to date. In the current implementation, the kernel avoids this by pre-reserving a bounded number of `MemHeader` structures for later driver use (this number can be adjusted on the kernel command line). A better solution would have the kernel page fault handler “fix” the per-process page directories by copying the missing entries from the master kernel page directory as needed.

## 5.4 The TransMap

Finally we arrive at the *transmap*. The purpose of the *transmap* is to allow the kernel to build mappings dynamically so that it can access data which is not ordinarily mapped into the kernel. Typically this happens when the kernel manipulates pages or capability pages, but it also can happen when the kernel manipulates mapping tables.

### 5.4.1 Purpose of the Transmap

The *transmap* is a page of memory that is mapped simultaneously as a page table and as a data page. Viewed as a page table, it specifies 512 (PAE) or 1024 (non-PAE) mapping entries for the virtual address region beginning at 0xFF800000. Viewed as a data page, it can be found at the address associated with the symbol `TransientMap`. The `TransientMap` symbol has no intrinsic type. Depending on the mapping mode that is currently in use by the hardware, we will cast it to one of the types `IA32_PTE *` or `IA32_PAE *`, which are the types for the respective types of page table entries for the two mapping modes. We will then use this pointer to update the mapping table, which allows us to construct temporary mappings.

Whenever the kernel needs to manipulate the content of a data page, capability page, or mapping table, the protocol is to call `TRANSMAP_MAP(pa, ptr type)`. The supplied physical address must be page aligned, and the return value is the assigned virtual address. When the mapping is no longer needed, `TRANSMAP_UNMAP(va)` is called, passing the previously returned virtual address. The *unmap* operation marks the virtual address as “available for re-mapping,

but not yet cleared from the TLB.”

The *transmap* page is shared across CPUs. Each CPU has a dedicated region of the *transmap* providing 32 slots where pages can be mapped on a transient basis. This allows the current implementation to support up to 16 processors. Obviously, the size of the *transmap* could be extended at need. Our view as designers is that above 16 CPUs the memory hierarchy of the machine tends to switch to a more pronounced NUMA architecture, and the entire model of a shared kernel map needs to be re-examined.

### 5.4.2 Lazy TLB Consistency

The general rule is to flush the TLB when a mapping is removed. For application mappings this is absolutely required. For kernel mappings, the true constraint is more subtle: we must not re-use any transient mapping if it might still reside in the hardware TLB. For kernel mappings, we can impose an invariant requiring that no transient mapping be used after it is unmapped. This creates an opportunity for an optimization that may initially seem like a bug.

The *transmap* implementation can be found in `arch/i386/kernel/transmap.c`. When an `unmap` is performed, the corresponding TLB entry is not flushed immediately. If you examine the implementation of the `transmap_map()`, you will see that it initially attempts to allocate entries from `TransMetaMap`. `TransMetaMap` indicates which entries are available (corresponding bit set to 1) for allocation. In order to be available, an entry must be free and we must know that it cannot be resident in the local TLB. Initially all entries are available.

If the allocator cannot find an entry in `TransMetaMap`, it locates an entry in `TransReleased` and uses that. The `TransReleased` map indicates which entries have been unmapped (corresponding bit set to 1) but not yet flushed from the TLB. The allocator flushes the entry selectively and re-uses it.

The reason for this approach is that we are making a bet. We are betting that an address space switch will be performed soon in any case. When that occurs, any entries in `TransReleased` can be migrated to `TransMetaMap` immediately. Note that some system calls do not flush the TLB at all.

Finally, note that entries in the *transmap* are *never* shared across CPUs. Invalidating TLB entries is expensive, but invalidating them across all CPUs in a multiprocessor is glacial. This is why the *transmap* is divided into separate regions for the entries of each CPU.

### 5.4.3 Data Cache Locality

On some processor implementations, including recent AMD 64-bit processors, TLB loads proceed through the data cache. The data cache is subject to an inter-processor coherency protocol. For this reason, it is desirable to choose the number of per-CPU *transmap* entries in such a way that they will occupy an integral multiple of cache lines. This will ensure that the respective entries will be allocated exclusively to the proper CPU by the hardware cache coherency protocol, and later will avoid contention because no other CPU will reference the entry.

### 5.4.4 Other Implementations

The implementation of the *transmap* is part of the hardware abstraction layer. On machines where the entire physical memory space can be mapped into the kernel virtual region, the *transmap* can be implemented using a constant offset mechanism, and no manipulation of the hardware map is required.

## 5.5 Implementation: Configuring Memory

With the mechanisms now explained, we return to the discussion of `arch_init()`, picking up after the initialization of the *transmap*. The call to `pmem_init()` initializes the physical memory manager by advising it how large the physically addressable region is. The call to `config_physical_memory()` uses the information provided by *grub* to add all of the defined regions of the physical map to the allocator (or at least all of the ones we know about at this

point). It also adds (by hand) a small number of well-known regions that contain BIOS information. Applications may want those regions later, and by reserving them now we ensure that the kernel does not use them for object storage.

Ignoring the memory used by any local APICs and/or IOAPICs, which will not come out of RAM regions, we now have configured all of the RAM memory that we are ever going to know about. Momentarily, we will do temporary allocations of some of the multiboot regions that *grub* has built in order to protect them while we allocate the heap, but before we do that we want to get an estimate of the total number of physical pages that are going to be available when we are done. This is returned by the call to `pmem_Available()`. Finally, the call to `protect_multiboot_regions()` allocates the regions containing the *grub*-provided information that we still need, primarily the command line and the loaded module that defines the initial system image. This prevents these regions from being overwritten as we define the heap.

All of this gets us most of the way to where we need to be, but we still have not reserved the physical memory addresses for any local APICs and/or IOAPICs that happen to be present. The local APIC and IOAPIC logic is discussed in the next chapter. For the present, suffice it to say that we want to mark their locations in the physical map allocated. In order to do that, we need to probe the system to determine how many CPUs there are and where their hardware elements are mapped. The call to `cpu_probe_cpus()` in turn calls `acpi_probe_cpus()`, which reserves the required locations and determines (as a side effect) how many CPUs are actually present.

At this point, we have reserved all of the physical memory locations that it is possible to know about at this point in the execution,