TODO: write a nice title here

Thomas Hybel

Aarhus University
October 2017

Abstract

TODO: write an abstract here.

Contents

1 Introduction

- introduction - what are we trying to do - how are we doing it - following the course from MIT - why are we doing it - overview of the coming chapters

2 Lab 1

In this lab we wrote initialization code for our operating system. This code sets up a rudimentary page table and switches the processor from 16-bit real mode to 32-bit protected mode. We also wrote a boot loader, which is a small program that loads the main kernel from disk and transfers control to it.

2.1 The boot process

To understand the purpose of a boot loader, it is first necessary to have an overview of the process which an x86 machine goes through upon startup.

When an x86 machine starts, its BIOS code is run. The BIOS initializes some of the system's hardware components (e.g., keyboard, graphics card, and hard drive). The BIOS will then load one sector (512 bytes) from the boot medium into memory at a hard-coded address (0x7C00). Once this first sector is loaded into memory, the BIOS will transfer execution to the loaded code.

The first sector will typically contain a small program known as the boot loader. Its purpose is to load the main kernel from disk and transfer execution to it.

Before the boot loader loads the kernel, we first run some initialization code which sets up a more comfortable environment for the boot loader and kernel to work in.

2.2 Initialization code

When the BIOS jumps into our code, the processor is running in 16-bit real mode. However the code produced by a modern compiler expects to run in 32-bit protected mode. We therefore needed to write code which performs this switching of processor modes.

The main difference between real mode and protected mode is how address resolution is performed. In real mode, accessing a physical address is done using a segment selector register and a general-purpose register. In contast, protected mode allows (but does not require) the use of *paging*, i.e., the mapping from virtual from physical addresses using a page table.

Since we want our operating system to use virtual memory, we certainly want to switch to protected mode. We do not immediately enable paging, however.

To switch to protected mode, a bit needs to be set in the control register `cr0`. We do so with the following assembly code:

```
# protected mode enable flag
.set CR0_PE_ON, 0x1
```

```
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0
```

To switch to 32-bit mode, we must update the `cs` (code segment) register. This register is an offset into a table called the Global Descriptor Table (GDT) which contains a number of descriptors. Each descriptor has a bit which determines whether the chosen segment is a 32-bit or a 16-bit segment. We use the `ljmp` instruction to update the `cs` register.

Once our code has switched to 32-bit protected mode, it executes the boot loader. At this point the processor is in a state where it can execute compiled C code, which means that we no longer need to hand-write assembly.

The file `boot/boot.S` contains the code that performs the described tasks.

2.3 The boot loader

It is the task of the boot loader to load the main kernel and transfer execution to it. The boot loader must do this using no more than 512 bytes of code, minus the bytes used by the initialization code.

Our kernel will be compiled into an ELF file. The boot loader will parse this ELF file, using the information therein to load the code and data of the kernel to the right physical addresses.

Once the boot loader has finished loading the kernel, it determines the entry point address from the ELF file and jumps there, transferring execution to the kernel.

2.4 Minimal kernel code

At this point our kernel can finally run. However it does not yet have much functionality. Unlike user-mode programs, the kernel does not have access to a C standard library, unless we write one ourselves.

As a start, we would like to have the ability to input and output text. Our kernel can use the `inb` and `outb` instructions to communicate with the outside world via a serial connection. We used this to write out a message and confirm that the kernel runs.

Using instructions such as `inb` and `outb` to communicate with an input/output device is known as Programmed Input/Output (PIO). It is a technique we will be using throughout our operating system development.

3 Lab 2

The goal of this lab was to write code which sets up the page table, so that our operating system can use virtual memory. Before we could set up the page table, we first needed to implement a subsystem which manages the physical memory of the system.

3.1 Physical page allocation

A given system has a limited amount of physical memory, depending on how much RAM the machine has. This memory is split up into a number of pages. On x86, a page is 4096 bytes of memory. In hex, 4096 is 0x1000, which means that pages are always aligned on 0x1000-byte boundaries.

To find out how much memory is available on the system, we query a memory area called the CMOS. The CMOS is an area of memory which holds the amount of system RAM. We can read from the CMOS using PIO to figure out how many pages of physical memory are available to the kernel.

For each page, the kernel must keep track of whether it is in use or not, and if so, how many references there are to the given page. To accomplish this, metadata about each page is stored in a `PageInfo` struct. For example, the `PageInfo` struct holds the reference count of each page. The kernel has an array of `PageInfo` structs, called `pages`. Each entry of the `pages` array directly corresponds to one physical page of memory, such that the first entry in `pages` holds metadata about the first page of physical memory, and so on.

Free pages are additionally stored in a linked list of `PageInfo` structs called the `page_free_list`. Using a linked list lets the kernel return a free page in constant time.

We wrote the following functions to manage physical pages:

- `page_alloc` is used to allocate a page of physical memory
- `page_free` is used to put a page on the free list
- `page_decref` and `page_incref` are used to manage reference counts of pages

With this infrastructure in place, we were ready to set up the page table so that we could use virtual memory.

3.2 Page table theory

To explain how our operating system implements virtual memory, it is necessary to introduce some theory about the page table.

The page table is a two-level table whose main purpose is to let the processor translate a virtual address to a physical address. Conceptually, the x86 page table is a 1024-ary tree of height 2. The physical address of the page table – the root of the tree – can be found in the `cr3` register.

The first level of the page table is called the Page Directory. It contains 1024 Page Directory Entries (PDEs). Each PDE points to a second-level node in the page table. Each second-level table holds 1024 Page Table Entries (PTEs). It is possible to map a virtual address to a PTE. The PTE then holds the physical address to which the virtual address should map. It also holds some status bits.

The status bits of PTEs and PDEs determine such features as whether the page is writable, and whether it is accessible to user-mode code. There is also

a "present" bit, which determines whether the page is virtual address maps to a physical page at all, or if accesses should cause a page fault instead.

To translate from a virtual to a physical address, it is necessary to walk the page table. For sake of illustration, assume that the processor is instructed to access a virtual address $v = 0x11223344$. The processor first looks in the `cr3` register to find the root of the page table. It then uses the higher-order 10 bits of v as an index into the page table. In this case, we have:

$$v = 0x11223344 = 0b10001001000100011001101000100$$

So the 10 higher-order bits are:

$$0b1000100100 = 548$$

At index 548 into the page table, the processor finds a PDE. It extracts a physical address from the PDE to find a page directory. It then uses the next 10 high-order bits of v as an index into the page directory. We have:

$$0b0100011001 = 281$$

So the processor looks at index 281 into the page directory and finds a PTE. The PTE holds some status bits, which the processor can use to check whether the page is present, and whether access permissions are appropriate. If not, a page fault is generated. If all goes well, the PTE holds the physical address of the page that should be accessed. The lower-order 12 bits of the virtual address are used to index into the physical page, and the processor is finally done with address translation.

This is a costly process, and in practice the job is done by specialized hardware called a Memory Management Unit (MMU). Additionally, a cache called the Translation Lookaside Buffer (TLB) holds the results of recent translations, to avoid having to walk the page table too often.

3.3 Page table management

We wrote the following functions to manage the page table:

- `pgdir_walk` is the main workhorse of the subsystem, since it is called by most of the other functions. It has the same function as the MMU; it is given a page table and a virtual address, and it walks over the table to find the corresponding PTE, allocating new levels of the table if needed, using `page_alloc` from the previous section.
- `page_insert` is used to insert a physical page into a page table at a given virtual address. In other words, it finds a PTE for a virtual address and stores the physical address there.
- `page_lookup` finds the physical address of a page, given a virtual address.
- `page_remove` invalidates a PTE in a page table.

The kernel uses these functions to set up the page table. This involves allocating pages of physical memory and inserting them into appropriate places in the page table.

It is up to us where the different things should go. Figure 1 contains a diagram which gives a simplified overview of the address space. For a more complete version, see the file `inc/memlayout.h`. The diagram shows that the

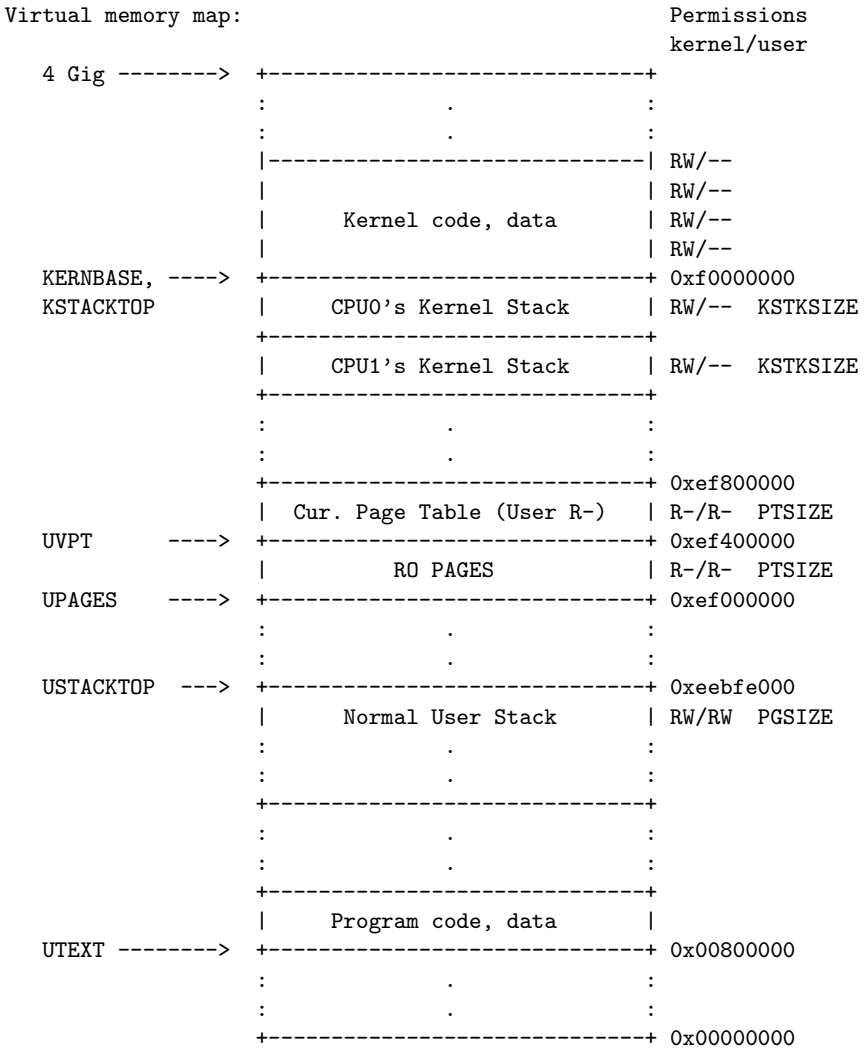

Figure 1: The virtual address space of the kernel

kernel code resides starting at virtual address 0xf0000000. The kernel stacks, used by processors when running kernel-mode code, reside just below, be-

tween 0xefc00000 and 0xf0000000. Further down in the address space, between 0xef400000 and 0xef800000, we have the User Virtual Page Table (UVPT) area, which gives user-mode programs read-only access to the page table, enabling certain exokernel-style programs to work. The stack of the user-mode program starts at 0xeebfe000 and grows toward lower addresses. The user-mode program code and data resides at 0x00800000.

After allocating a page table and updating it to reflect all these conventions, we update the `cr3` register to atomically update the page table. Finally we set a bit in `cr0` to enable paging.

Our system now has virtual memory. This means that user-mode programs cannot access kernel memory, since we have not set the user bit in the PTEs for this memory. Furthermore, user-mode programs cannot interfere with each other; they will each be given their own separate page table, where none of the physical addresses overlap by default.

4 Lab 3

The goal of this lab was to get a user-mode program running as a process in its own virtual address space. To accomplish this, we needed to write functions to manage processes and another ELF file loader. If the process triggers an exception, such as a division by zero, our kernel additionally needs to handle this. Finally we implemented a system call mechanism to e.g. let programs perform input and output.

4.1 Process management

Each process has some associated information. This includes its state (running, runnable, killed, etc.), its process ID, its parent process ID, its page directory, and so on. All this information is stored in a struct `Env`. The process subsystem is similar to the page subsystem; an array, `envs`, holds the `Env` struct of each process on the system, and free environments are stored in a linked list called `env_free_list`. We implemented functions for creating, initializing, and destroying a process.

4.2 ELF loading

The kernel also needs a way to load a program into an address space. Programs are represented as ELF files. This let us reuse our ELF loading code from our boot loader.

To load a program, the kernel first allocates a fresh page table. The kernel then walks over each section in the ELF file, figures out the virtual address at which the section should go from the ELF file, allocates corresponding physical pages, inserts them into the page table, and copies the code or data from the program into the physical pages.

Note that we have not yet introduced a file system, so it is not immediately clear where the kernel can find the programs which it should load. To solve this problem we embed each user-mode program into the kernel as a blob of binary data. In a later section we describe our implementation of a proper file system for holding programs and data.

4.3 Context switching

To actually run a process, we need to perform a context switch. The `Env` struct includes the metadata for a process, including its registers. To perform a context switch, we first restore the saved general-purpose registers by using the `popal` instruction. We also load the page table of the process into `cr3`.

We then issue the `iret` instruction, which restores the saved `eip`, `esp`, `eflags`, and `cs` registers. This transfers execution to user-mode code. The lower two bits of the `cs` register determine the Current Privilege Level (CPL) of the processor. This was previously 0, since the kernel runs in ring 0. By setting this to 3 during the `iret`, the processor switches to user-mode operation, i.e., ring 3.

At this point we were able to run user-mode code in its own address space. Unfortunately the code had no way to give control back to the kernel, so it simply ran forever, or at least until it triggered an exception or interrupt, which caused the whole system to crash.

4.4 Theory of exceptions and interrupts

While executing an instruction, the processor may trigger an exception. This could e.g. happen due to a division by zero, or due to an illegal memory access. When an exception occurs, the processor performs a context switch to enter kernel mode. Then it runs some handler code specific to the exception.

The processor may also occasionally trigger an interrupt. This often happens for asynchronous reasons; examples could be receiving a new network packet or key press. Interrupts can also happen for synchronous reasons; for example, an interrupt is raised as a result of executing the `int 0x80` instruction which is commonly used for system calls. The behavior is the same for exceptions; a context switch occurs to run a handler in kernel mode.

We now describe how the processor knows which code to run when an interrupt or exception is raised. Exceptions and interrupts have numbers. Exceptions are numbered from 0 to 31, and interrupts are numbered from 32 to 255.

These numbers specify an index into a table called the Interrupt Descriptor Table (IDT). The physical address of the IDT is stored in the IDT Register (IDTR), which can be read and set using the `lidt` and `sidt` instructions, respectively.

The IDT table entry describes the address at which the handler for the given interrupt resides.

As noted, an exception or interrupt triggers a context switch. Therefore the processor needs to know on which stack it should save the registers of the

faulting process. To find this `esp` value, the processor reads the Task Register (TR)) which is an index into the GDT. The GDT entry contains the address of a data structure called the Task State Segment (TSS). The processor reads the new `esp` value from the TSS.

To sum up: when an exception or interrupt occurs, the corresponding number is looked up in the IDT to find the new `eip` value. The TR, GDT and TSS are used to find the new `esp` value. The processor uses this information to store the registers of the faulting process onto the new stack and execute the relevant handler code.

4.5 Handling exceptions and interrupts

We wrote a common function, `trap`, which is called whenever an exception or interrupt occurs. For each exception and interrupt, we filled in its IDT entry with a small stub which passes the number of the exception or interrupt as the first argument in a call to `trap`.

The typical result of an exception is that the kernel terminates the running process. However exceptions can also occur in kernel mode, in which case there is a bug in the kernel. This results in a kernel panic, making the kernel print the exception and hang.

4.6 Handling system calls

A process often needs to ask the kernel to perform some task for it. This could e.g. involve printing text onto the console, reading a character from the keyboard, or spawning a new child process. We needed to implement a mechanism for triggering system calls in our kernel.

On the x86 architecture, the typical approach is to use the `int` instruction, which triggers an interrupt when executed. On Linux, interrupt 0x80 is used for system calls, but we are free to use any interrupt, so we arbitrarily chose number 0x30. This means that a program uses the `int 0x30` instruction to perform a system call. The interface was up to us, but we kept it fairly standard; the `eax` register holds the system call number, while arguments go in registers `ebx`, `ecx`, etc.

The `trap` function recognizes interrupt number 0x30 and calls the `syscall` function, which uses a large `switch` to delegate each system call to a specific handler.

We wrote system call handlers for input and output of a single character. At this point we were able to run simple user-mode programs and have them interact with the user. The programs could terminate themselves with a specific system call, or by triggering any exception.

5 Lab 4

In the previous lab, we reached a point where our kernel could run a single user-mode process. The current lab concerns itself with running multiple processes concurrently, and having them interact. We implemented preemptive multitasking, process forking, and inter-process communication.

Before we could do this, however, we needed to implement code for interacting with a device called the LAPIC.

5.1 Interacting with the LAPIC

A Programmable Interrupt Controlled (PIC) is a device which is responsible for managing interrupts for the processor. For example, if multiple interrupts are generated simultaneously, the PIC can prioritize the interrupts and deliver them one at a time. When Intel updated their PIC standard to include new features, the conforming device was called an Advanced PIC (APIC). The APIC has a component called the Local APIC (LAPIC) which is local to each processor. Thus modern systems have one LAPIC per processor.

In section 5.2 we need to query the LAPIC to activate more processors than just the initial one. Furthermore, in section 5.4 we need to ask the LAPIC to generate periodic clock interrupts which our scheduler will use to preempt the running process. Therefore it is relevant to describe how our kernel communicates with the LAPIC.

The processor can communicate with its LAPIC using Memory-Mapped I/O (MMIO). This means that the LAPIC is mapped into memory at a specific, system-dependent physical address. Reading at certain offsets will correspond to reading from certain registers in the LAPIC, and likewise for writing.

This means that to communicate with the LAPIC, the kernel merely needs to read and write to certain addresses. The difficult part is figuring out the physical address where the LAPIC resides.

There are multiple ways to find the LAPIC. For now, our kernel uses the method described in Intel's multi-processor specification. We will refer to this as the mpconfig method.

We leave out the details for brevity, but the method boils down to the following. The kernel searches for a structure called the MP floating pointer structure. It does so by looking in certain parts of physical memory for the string "_MP_", validating a checksum on the memory to ensure that this was not a false positive. This structure points to a table called the MP configuration table, which contains the physical address of the LAPIC.

5.2 Activating more processors

So far we have been running our kernel on an emulated machine with a single processor. A system with n processors can be emulated by passing the `-smp n` option to QEMU.

When a system with multiple processors boots, the hardware dynamically selects one of the processors to be the Bootstrap Processor (BP). The other processors are called Application Processors (APs). At first, only the BP runs. It is up to the BP to start up the remaining APs once the system is ready.

Up to this point, only the BP had been running our kernel so far; APs were never activated. We therefore needed to write code which starts the APs. This is done by asking the LAPIC of the BP to send an Inter-Processor Interrupt (IPI) to each of the APs. This IPI causes the APs to start executing code. The address of the code to run is sent as part of the IPI.

The APs start in 16-bit real mode, just as the BP did. They therefore need to switch to 32-bit real mode mode. After doing so, each AP calls into the scheduler to run a new process.

5.3 Ensuring mutual exclusion

With multiple processors running concurrently, all the typical issues of concurrency arose. Specifically, multiple processors could modify internal kernel structures simultaneously, leading to race conditions.

We prevented this in the trivial but inefficient way: we make sure that at most one processor is running kernel code at the same time. In other words, our kernel is not truly concurrent. Still, user-mode processes *can* run truly concurrently.

The "big kernel lock" is a spinlock. It is essentially a global variable which determines whether the kernel is locked. A processor repeatedly uses the `lock` and `xchg` instructions to atomically exchange the global variable with the value 1. If the global value was zero, the processor now holds the lock and may enter the kernel. Otherwise it must retry.

We added calls to lock and unlock the kernel in the right places. At this point our kernel was capable of running multiple user-mode processes concurrently. However these processes ran forever, so we needed a scheduler next.

5.4 Process scheduling

- we don't have a scheduler yet! - we'll do the simplest possible thing: round-robin scheduling - we keep processes in an array, and the scheduler has an index into the array which it increments until it finds a runnable process. - scheduling is voluntary for now. We need preemption. - LAPIC gives us preemption; we tell it to raise a clock interrupt every n (miliseconds? or what?) - our trap function recognizes this and schedules out the current process

5.5 Forking

- simple version - do as much as possible in user land - compose fork of multiple other syscalls - exofork, then fill out the process with identical pages - can read our page table via UVPT - security: only parent can do this - CoW version - motivation: simple fork is inefficient; might as well share non-writable pages,

and even writable ones too, if they're never modified.. - process gets to handle its own page faults, so we can keep code out of the kernel - process can register a page fault handler via a syscall - then page faults will be handled by trap, such that execution is transferred to a special function (and dedicated stack) - works by just overwriting the saved registers and putting the old ones on the new stack - now CoW fork is implemented like this: - all processes register a default page fault handler - to fork, just copy all the pages and set the CoW bit, but not the W bit - writes trigger page faults -¿ trap -¿ page fault handler in user-mode library - the page fault handler makes a new writable page, copies content to it, unmaps old page, moves new page to old one's address, then resumes normal execution

5.6 Inter-process communication

- will need IPC for many things, because this is an exokernel - e.g. file system will be implemented by a user-mode daemon process, and other processes will have to ask it to read/write. - same for networking - sys_ipc_recv and sys_ipc_try_send - recv blocks - send never blocks - a 32-bit integer is sent, and optionally also a page of memory - also wrote user-mode library functions for receiving and sending - send repeatedly calls try_send

6 Lab 5

- lab 5 - file system - shell

7 Lab 6

- lab 6 - networking: writing an e1000 card driver

8 Graphics Lab

- graphics

9 Hardware Lab

- hardware
  - conclusion

References

Abbreviations

`cr0` Control Register 0. It holds bits which determine how the processor operates. It e.g. determines whether paging and protected mode are enabled.. 1, 6

`cr3` Control Register 3. It points to the current page table.. 3, 4, 6, 7

`cs` Code Segment selector register; it holds an index into the GDT. Its lower two bits determine the CPL.. 2, 7

`eflags` A register which holds various flags that mostly reflect the properties of the most recently executed instruction. For example the signed flag is set during a subtraction whose result is below zero.. 7

`eip` Extended Instruction Pointer; a register which holds the address of the next instruction to be executed.. 7, 8

`esp` Extended Stack Pointer register.. 7, 8, 13

AP Application Processor; any processor which is not a BP. The APs only run once they are started by the BP.. 10

APIC Advanced PIC.. 9, 12

BP Bootstrap Processor; the first, and initially only, processor that runs when a system boots.. 10

CPL Current Privilege Level; the current ring in which the processor is executing. CPL=3 means ring, 3, i.e., user-mode, while CPL=0 means ring 0, i.e. kernel-mode.. 7, 12

ELF Executable and Linkable Format; a file format used to store executable programs. An ELF file describes the address at which each section of program code or data should be loaded.. 2, 6

GDT Global Descriptor Table; a table which holds descriptors, each of which describes a segment of memory and its permissions.. 2, 8, 12, 13

IDT Interrupt Descriptor Table; a table which describes the address of the handler that should be run when a given interrupt or exception is triggered.. 7, 8, 12

IDTR A register which holds the physical address of the IDT.. 7

IPI Inter-Processor Interrupt. An interrupt sent by one processor to another using the LAPIC.. 10

LAPIC Local APIC. The LAPIC is the processor-local component of the APIC. Modern processors have one LAPIC per processor.. 9, 10

MMIO Memory-Mapped I/O. If you communicate with a device using MMIO, it means that the registers of the device are mapped into memory at some address, and so communication happens by reading from or writing to memory.. 9

MMU Memory Management Unit.. 4

mpconfig mpconfig is a method for finding information about multiple processors described in Intel's Multi-processor specification.. 9

PDE Page Directory Entry; a 32-bit integer which stores a physical address of a second-level node in the page table, as well as some status bits.. 3, 4

PIC Programmable Interrupt Controller; a hardware device which orders interrupts before delivering them to the processor. 9, 12

PTE Page Table Entry; a 32-bit integer which stores a physical address to which a virtual address maps, as well as some status bits.. 3, 4

TLB Translation Lookaside Buffer.. 4

TR Task Register; a register which is used as an index into the GDT to find the TSS.. 8

TSS Task State Segment; a data structure which, among other things, determines the `esp`-value used during a context switch triggered by an exception or interrupt.. 8, 13