TODO: write a nice title here

Thomas Hybel

Aarhus University
October 2017

Abstract

TODO: write an abstract here.

Contents

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

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.

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.

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.

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 Executable and Linkable Format (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.

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.

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.

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.

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 contains 1024 Page Directory Entries (PDEs). This name makes sense because each PDE points to a second-level table, which is called a Page Directory. Each PDE additionally holds some status bits. In other words, the first level of the page table contains 1024 pointers to second-level tables, along with some status bits.

Each second-level table, i.e., each Page Directory, 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.

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

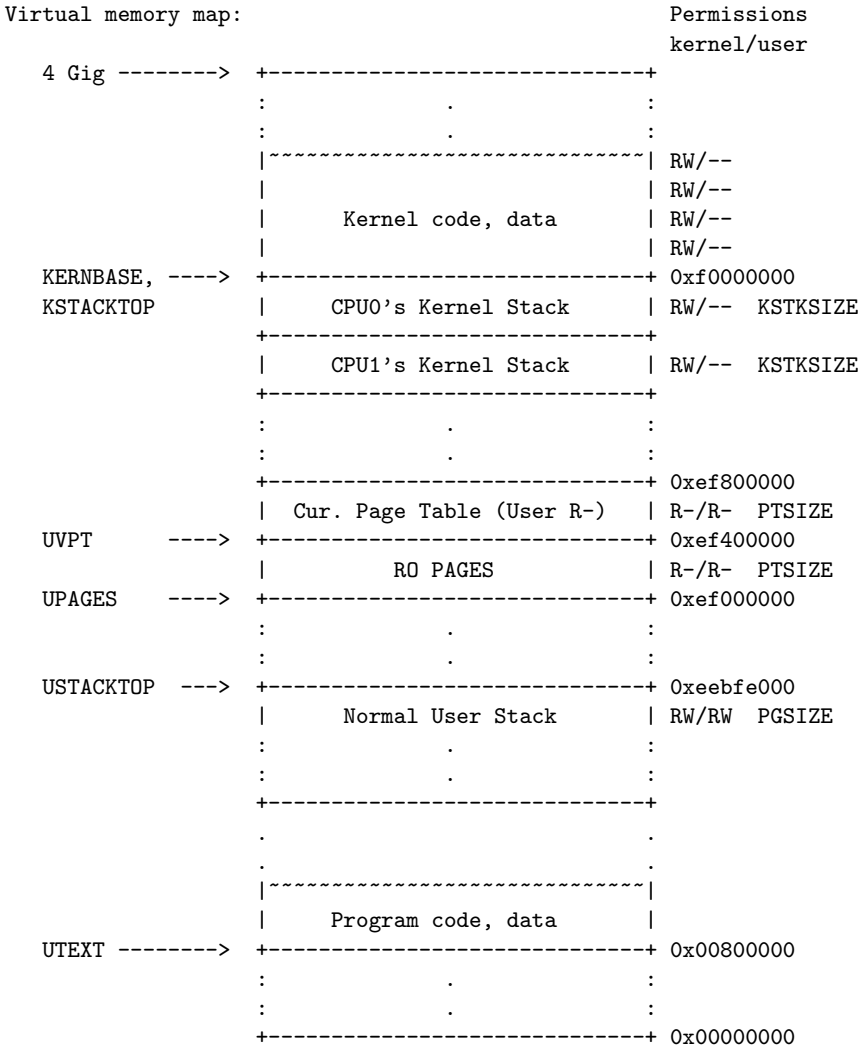

Figure 1: The virtual address space of the kernel

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

Lab 3

- lab 3 - goal: make user-space process run - functions for keeping track of proceses - ELF loader - context switching - handling exceptions - handling syscalls

Lab 4

- lab 4 - preemptive multitasking - starting multiple processors - simple scheduler with yields - preemption (clock signals) - process forking with CoW - simple fork - CoW fork - IPC

Lab 5

- lab 5 - file system - shell

Lab 6

- lab 6 - networking: writing an e1000 card driver

Graphics Lab

- graphics

Hardware Lab

- hardware
    - conclusion

References