TODO: write a nice title here

Thomas Hybel

Aarhus University
November 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 Booting

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 Memory management

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

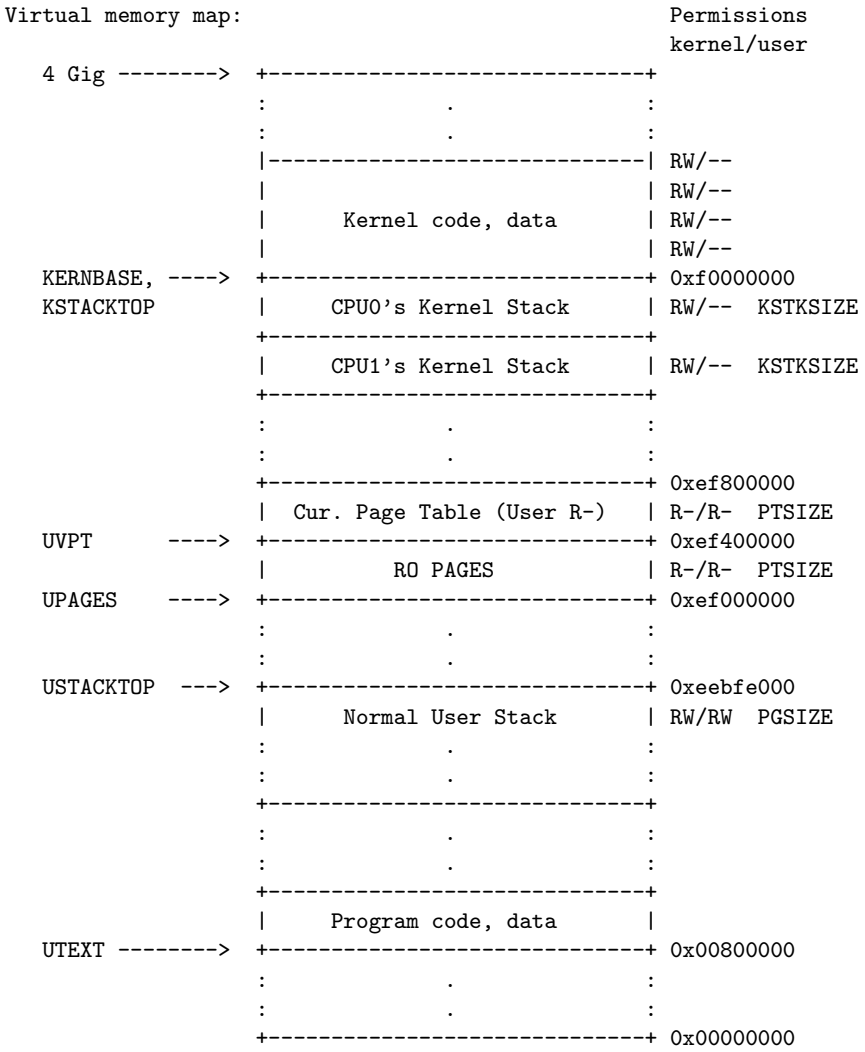

Figure 1: The virtual address space of the kernel

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 Process management

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 information

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 Multiprocessing

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 timer 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

The scheduler has as its main responsibility to pick a new process to run whenever the current process is scheduled out. We opted for round-robin scheduling.

That is, the scheduler keeps a circular queue of all processes. To find the next process, the scheduler retrieves the next process from the queue until it finds one that is runnable, at which point the scheduler is done.

The scheduler also needs a way to preempt each process when its time slice runs out. To accomplish this, during kernel initialization the kernel asks the LAPIC to raise a timer interrupt periodically, waiting some fixed amount of bus cycles between each raised interrupt. Our `trap` function then recognizes this timer interrupt and reacts by asking the scheduler to schedule in a new process.

Our scheduler has much room for improval. Round-robin scheduling is not as performant as more advanced algorithms. Additionally, it does not allow

adjustment of process priorities. Scheduling being a linear-time operation is potentially worrying.

A final issue is that the time slice assigned to each process is always of the same duration. It would be useful to assign shorter time slices and more frequent schedulings to processes which need to feel responsive (such as graphics-based processes introduced later) and longer time slices to processes that perform heavy computations.

5.5 A simple fork mechanism

So far, every process was directly spawned by the kernel. However, processes should also be able to spawn more processes. We therefore needed to implement a mechanism to let a process fork. In this section we describe our initial, simple implementation of fork, and in the following section we improve it by introducing a copy-on-write mechanism.

Since our kernel is an exokernel, we prefer to do as much work as possible outside of kernel land. We therefore wrote a number of system calls which can be combined to implement a user-mode fork. Specifically, we wrote the following system calls:

- `sys_exofork`: creates a non-runnable child process with an empty address space.
- `sys_env_set_status`: can mark a process as runnable.
- `sys_page_alloc`: allocates an empty page in the address space of a process.
- `sys_page_map`: maps a page from the current process into a child process.
- `sys_page_unmap`: unmaps a page from the current process or a child process.

Besides these system calls, it is also necessary for a process to have access to information about the layout of its own address space. This is already the case; in section 3.3 on page table management, we set up the page table of a process such that part of the address space contains the page table itself.

To fork, a parent process goes through the following steps:

- The parent process calls `sys_exofork` to create a new child process with an empty address space. The child process is not initially runnable.
- The parent walks over its page table, and for each mapped page, it does the following:
  - The parent uses `sys_page_map` to create a temporary page at a temporary address.
  - The parent copies the contents of the current page into the temporary page.

  - The parent uses `sys_page_map` to insert the temporary page into the address space of the child process at the original address.
  - The parent uses `sys_page_unmap` to remove the temporary page from its own address space.

- The parent marks the child as runnable using `sys_env_set_status`.

At this point the fork is complete, and since the child is marked as runnable, the scheduler will eventually schedule it in.

Note that for security reasons, the system calls are coded to ensure that a process cannot modify pages in other, non-child processes.

5.6 Copy-on-write fork

The simple version of fork described in the previous section is slow and memory-inefficient, because it indiscriminately copies every page of the parent into the child process. This involves a lot of copying, and it means that if the parent used n physical pages of memory, then after a fork, $2n$ physical pages will be used.

However the same physical page can transparently be mapped into both the parent and child process, as long as it is never modified. In fact, *all* the pages in the child process can initially be shared with the parent. It is only once a write happens that a page must be copied. We have implemented such a copy-on-write fork mechanism almost entirely in user land, following the exokernel design philosophy.

To perform as much work in user land as possible, we implemented a mechanism which lets a process handle its own page faults. By default, the kernel will terminate the running process if a page fault occurs. However we have implemented a system call, `sys_env_set_pgfault_upcall`, which lets a process set a handler function. If a handler is set, the kernel will handle a page fault by modifying the saved `eip` and `esp` registers of the process, pushing the old register values onto an exception stack, and switching the process back in.

Now, when a process forks, *all* pages are shared between the parent and child process. However writable pages have their writable bit removed from their page table entry (PTE). Instead we set another bit which marks the page as copy-on-write.

If either the child or parent process attempts to write to said page, a page fault will occur, since the page is not writable anymore. The kernel delegates to the registered user-mode handler function. The handler then uses the same method as in the simple fork implementation to map a new writable page, copy the contents of the old page onto it, and replace the copy-on-write page with the new writable page.

5.7 Inter-process communication

Since our kernel is an exokernel, many of the features implemented in later labs will reside in user land. Two examples are a file system daemon, and a

daemon implementing a network stack. Other processes need a way to interact with these daemons to make use of their services. We therefore have need of an inter-process communication (IPC) mechanism.

We therefore implemented two system calls, `sys_ipc_recv` and `sys_ipc_try_send`, which enable IPC. When a process calls `sys_ipc_recv`, it will hang waiting for a process to send it data. `sys_ipc_try_send` will send data to a process in a non-blocking fashion. By default, a 32-bit integer is sent between processes, but for efficiency an extra argument to the system calls allows the sender to share a full page of memory with the recieving process.

This marks the end of the multiprocessing lab. Our kernel can now run user-mode processes in a truly concurrent fashion. A scheduler manages the running processes, preempting them when necessary. Processes can efficiently fork and communicate via IPC.

6 File system

The goal of this lab was to implement a custom file system and a shell.

6.1 Custom file system

So far, our kernel has embedded any user-mode programs inside itself as binary blobs. This is highly undesirable; it requires recompilation of the full kernel to modify user-mode programs, and it makes it impossible for programs to store data persistently on disk. We therefore implemented a custom file system, which we now describe.

A file system exists on a disk. A disk can be thought of as a large amount of writable space. We can partition the disk's space into blocks, where each block has a specific size. For our file system, the block size will be 4096 bytes.

One of these blocks is special; it's called the "superblock". The superblock holds any metadata needed for the file system, such as the disk size, where to find the root folder, etc.

Our file system is laid out as follows. The disk is partitioned into blocks as mentioned. Block 0, the first one, is not used by our FS. (This way it can hold the boot loader.) Block 1 is, by convention, the superblock.

The next blocks, starting at block 2, hold a bitmap of free blocks; bits correspond to blocks on disk, and each bit is 1 if, and only if, the corresponding block is free. This way we can allocate and free blocks.

The remaining blocks are used to store the concrete files and folders.

We have decided to keep our file system as simple as possible for ease of implementation. Thus there is no concept of permissions, symbolic and hard links, timestamps, and so on.

A file is represented in C as a `struct file`, which is stored in a block. Such a `file` struct contains metadata, such as the file name and size. The struct also has 10 pointers to the blocks that hold the raw file data. If the data cannot fit in 10 blocks, the `file` struct has a pointer to a block which holds another

1024 pointers to data blocks. Thus our file system has a maximum file size of $(10 + 1024) * 4096 = 4235264$ bytes, i.e., around 4 MB.

Folders are represented as a `struct folder`, which is exactly identical to a `struct file`, except that the $10 + 1024$ block pointers no longer point to raw data, but to other blocks holding `file` or `folder` structs. There is a type flag allowing us to distinguish between files and folders.

We used a small script to create a raw disk image with an initialized file system of the described format. The script let us add files, such as sample programs, to the file system. We then attached this raw disk to QEMU.

6.2 File system daemon

In accord with exokernel design, we let all file system interaction go through a single privileged process which we call the file system daemon.

The daemon interacts with the disk using PIO. However normal user land processes cannot use the `inb` and `outb` instructions to perform PIO, so our kernel needs to give the daemon I/O privileges. It does so by setting a bit in the `eflags` register for the daemon process.

The file system daemon spends its time looping, waiting for other processes to contact it via IPC. Processes can send requests to open, read, write, and stat files. The IPC details are hidden behind our implementation of a C standard library, such that processes have the usual interface with functions such as `open`, `read` and `write`.

We have further improved the efficiency of the file system daemon by implemented a block caching system. When a block is first read, its contents are stored in RAM, and subsequent reads do not need to interact with the disk until the cache entry is invalidated through a write.

To summarize, when a process wants to open a file, the following happens. The process calls the `open` library function. The library uses IPC to contact the file system daemon. The daemon uses the file system superblock to find the root folder. It walks over the folder structure by following block pointers, until it finds the block that holds the data for the correct file. This location on disk is stored in a file descriptor, and the number of the file descriptor is returned to the first process via IPC. When a subsequent `read` is performed, the daemon uses the pointers in the `file` struct to find the raw data block and sends the requested data back via IPC.

6.3 Shell

We implemented a simplistic shell which lets us load and execute programs from disk. The shell also features input redirection ("¡"), output redirection ("¿"), and pipes ("—"). It is now possible to write to the disk and have the changes persist across reboots.

7 Networking

The goal of this lab was to connect our kernel to the internet by writing a network card driver.

7.1 The e1000 network card

The operating system is connected to network via a network card. The network card has as its main responsibility to send and receive packets by interacting with the physical layer, such as a wire.

The network card emulated by QEMU is the Intel e1000. We therefore need to write a driver for this card. The task of the driver is to discover the network card, initialize and enable it, drain incoming packets from the card, and deliver outgoing packets to the card.

We found all the details needed for the driver in the manual. It describes in detail how the card works, how it should be initialized, and so on.

The e1000 card is connected through PCI. Our kernel uses PIO to scan the PCI bus, iterating over all connected devices. Each device has numbers representings its class, subclass, vendor ID and device ID. The kernel uses these numbers to recognize the e1000 card if it is connected. The PCI interface lets us determine a physical address at which the e1000 card is mapped.

This completes the discovery phase; the kernel can now use MMIO to communicate with the card. Concretely, this means that the various registers of the network card can be found at specific offsets from the base address found via PCI. In other words, the registers of the card can be read from and written to by simply reading or writing at certain memory addresses.

The next step is initializing the card. We next describe the data structures which the card uses, since these are what need to be initialized.

7.2 Packet queues

Overall, the e1000 network card uses two circular queues for holding packets.

One queue, the transmit queue, is for packets which are yet to be sent. When the kernel wants to transmit a packet, it places it into this queue. The e1000 periodically drains this queue and puts the packets on the wire.

The other queue, the receive queue, is for packets which the e1000 has received but which have not yet been claimed by the kernel. Thus when a new packet arrives on the wire, the e1000 picks it up and deposits it into the receive queue. The kernel then periodically drains this queue.

Each queue is implemented as an array of descriptor structs. A descriptor contains a physical address and a size, and thus it describes a memory area which can hold one packet. Each queue also has a tail index register and a head index register which are used during insertion and removal of packets.

This means that when a packet arrives from the network, the e1000 card finds the descriptor at the tail of the receive queue, copies the raw packet data into the described memory area, and advances the tail register. The kernel takes

a packet out of the queue by copying the raw packet out of the descriptor at the head of the queue, and then incrementing the head register. Transmission of packets is analogous.

This raises the question of what should happen if one of the queues is full. According to the manual, if the receive queue is full, the e1000 card will simply drop further packets. However if the transmit queue is full, it is up to the kernel what should be done. The kernel could potentially store packets until the network card has drained the queue. However for simplicity we have decided to simply drop packets if the transmission queue is full. We are free to do this since the protocols on higher levels of the network stack handle packet loss.

7.3 The e1000 driver

Before the card can be enabled, our driver must initialize these queues. That is, it must allocate the physical pages required to hold the queues themselves. It must also allocate pages for the raw packet contents, and fill out the addresses and lengths into the descriptors in the queues. The head and tail registers of each queue must also be reset.

Once our driver has done this, it writes into a register to enable the card. From this point it is possible to transmit and receive packets. Our driver implements the code for taking packets from the receive queue and inserting packets into the transmit queue. User-mode programs have access to this driver functionality through two system calls, `sys_transmit` and `sys_receive`.

7.4 The network daemon

With the driver written, our kernel was capable of transmitting and receiving packets. However most processes only know of the data they wish to send; they are not capable of constructing packets.

We therefore needed a TCP/IP stack. Allegedly, writing such a network stack from scratch is an immense task. We therefore used the open-source stack lwIP ("lightweight IP"). lwIP acts as a black box for our purposes, taking raw data as input, and producing packets as output.

Rather than adding lwIP to the kernel, we embedded it in a network daemon. The daemon is responsible for managing sockets, just as the file system daemon was responsible for managing file descriptors. The network daemon takes in `send` requests via IPC, produces packets, and hands these over to the operating system via the `sys_transmit` system call. Likewise, it takes in `receive` requests and uses `sys_receive`, parses the resulting packets, and hands over the data to the other process.

The IPC communication is hidden away in the C standard library, such that user-mode programs have access to the usual `connect`, `send` and `receive` interface.

7.5 Web server

To test the new functionality, we wrote a simple web server which can serve files from the file system. The web server thus makes use of both the file system daemon and the network daemon.

We configured QEMU to forward requests at port 80 to the emulated machine. At this point we were able to point a browser at the machine and be served a working web page.

8 Graphics

intro goes here

8.1 Drawing pixels

Before our kernel could render a full GUI, it first needed the ability to render a single pixel at a time. From this primitive is it straightforward to implement rendering of bigger shapes like rectangles, lines, and so on.

To draw pixels, it is necessary to set a video mode. A video mode describes the width, height, and depth of the screen. The video mode can be set by querying the BIOS. Once the kernel selects a video mode, the BIOS maps a buffer called the Linear Frame Buffer (LFB) into RAM.

The LFB represents the pixels of the screen; it can be thought of as a two-dimensional array, where each value is a 32-bit integer representing the RGB value of a pixel on the screen. Thus to write a pixel to the screen, the kernel must simply calculate the correct offset into the LFB and write a 32-bit integer there.

To set the video mode, the BIOS can be queried with the `int 0x10` instruction. This transfers execution to the code of the BIOS. However since the BIOS is written as 16-bit real mode code, our kernel must switch the processor back to 16-bit real mode before it can issue the interrupt.

We therefore wrote assembly code which switches the processor mode, queries the BIOS to enumerate the valid video modes, and selects one with a satisfactory resolution.

At this point our kernel was able to color the screen by writing all over the LFB. We were then ready to design the graphics stack.

8.2 Graphics stack design

To design our graphics stack, we first read up on how graphics work in common operating systems and used this as a basis for our design. We decided to have a central privileged process, called the display server, which is responsible for most of the work.

The display server is responsible for spawning other graphical applications, assigning a portion of the screen to each application. The display server is the

only process with access to the LFB, so it is responsible for drawing all pixels to the screen.

Each graphical application is spawned by the display server. When this happens, the two set up an area of shared memory through IPC. This memory is called the canvas of the application. The application can write pixels into the canvas, and the display server will periodically read the pixels from the canvas and write them to the LFB.

Graphical applications wait in a so-called event loop; they continuously wait for input events to arrive from the display server. When an input arrives, the application handles it and can make changes to its canvas on this basis.

The kernel keeps a queue for input events. When raw input packets arrive from the mouse or keyboard, a driver handles these packets by putting them into an event queue. The display server periodically drains this queue through a system call. It then forwards each event to the appropriate application.

A graphics library provides common functionality needed by the user applications and the display server. The library provides functions for rendering rectangles, straight lines, fonts, and windows.

The following diagram shows an overview of the different components and how they interact with each other:

Thus imagine that a user sits in front of the machine with a terminal emulator application open and active. If the user presses the 'A' key, the following series of events will occur:

- The user presses the 'A' key.
- The keyboard generates an interrupt to signal to the kernel that input is available.
- The kernel keyboard driver reads the pressed key using PIO.
- The driver puts the event into the events queue.
- The display server eventually drains the events queue and finds the key press event.
- The display server finds that the terminal application is active and therefore forwards the event to it using IPC.
- The terminal application receives the event and adds the 'A' to a buffer which holds the current input of the user. This buffer will eventually be used to run a command once the user presses the enter key.
- The terminal application also wants to display the 'A' on the screen, so that the user can see what is written so far. The application therefore asks the graphics library to render an 'A' on its canvas using the default font.
- The display server writes the pixels from the canvas into part of the LFB.
- The user sees the 'A' appear on screen.

8.3 Details of the display server

The display server needs to have write access to the LFB, which is mapped at a physical address by the BIOS. To accomplish this, the display server uses a new system call, `sys_map_lfb`, which causes the kernel to add the LFB to the page table of the display server.

It is highly important that the display server is efficient; if it runs too slowly, the system will have a low frame rate, and interaction will feel choppy. We had to carefully optimize the code for the display server before we got an acceptable frame rate inside QEMU.

One of the optimizations is the following. The display server takes the content of each canvas and writes it into the LFB. However if canvases overlap, there is no need to first write the lower canvas into the LFB, and then immediately overwrite it with the canvas that is on top. We therefore introduced a buffer held in RAM. The display server first constructs the final canvas in this buffer, and then copies each pixel once and for all into the LFB.

8.4 Sample graphical applications

We have written two sample graphical applications. One is a simple terminal emulator. The other is a simple paint program.

9 Hardware

Up to this point, our kernel had always been running in the QEMU emulator. In this lab, our goal was to get it running on real hardware, specifically a Packard Bell Dot S netbook. The road there was paved with surprisingly many complications. Most of the issues were caused by the fact that QEMU emulates different hardware than that of the netbook. There were also instances where QEMU did not emulate certain aspects of a machine faithfully, causing latent bugs to surface on real hardware.

9.1 Booting from USB

The build process of our kernel produces a raw disk image which QEMU will boot. The first block of this image is the boot loader, which the BIOS loads and transfers execution to as described in section 2. Since QEMU will boot from this image, we figured that so would the netbook. We put the raw disk image on a USB drive and had the netbook boot from it. However the netbook did not recognize the USB as a bootable medium.

It turns out that a USB drive must have a valid data structure called a Master Boot Record (MBR) in its first block, otherwise most machines will not recognize the drive as bootable. The MBR must also contain a valid data structure called the BIOS Parameter Block (BPB).

The reason for this is historical; when USB technology was new, there was disagreement on whether a USB drive should be a raw storage drive or a bootable

medium. Initially the user could decide through a BIOS setting. But for usability reasons, manufacturers eventually instead used heuristics to detect whether a USB is bootable or not. These heuristics are based on the MBR and BPB.

Once we added a valid MBR and BPB to our boot loader, the netbook booted. However the machine got stuck at some point during the boot loader code.

9.2 No serial connection

Previously the kernel printed debugging information through a serial connection which QEMU emulated. However we did not have the hardware necessary to set up a serial connection to the netbook. We therefore had no output, which made it difficult to determine why the boot loader was hanging.

We therefore had to write code which outputs text to the screen instead. Before a video mode has been set as described in section **??**, the machine is in text mode. In text mode, by convention a buffer at physical address 0xb8000 represents the text on the screen; we can write characters directly into this buffer to show text on screen.

9.3 No USB driver

With the ability to print text to the screen, we figured out why the boot loader was hanging. Our boot loader is naively coded such that it loads the kernel from a disk connected with an ATA connection. But currently the kernel resides on a USB drive, which is a completely different interface.

To continue, we needed to write a USB disk driver for the boot loader. However the boot loader is still constrained to 512 bytes, since it is loaded from the first sector of disk by the BIOS. Fitting a USB driver there is tricky.

We therefore opted for a different approach. We booted from USB into a Linux distribution, and used that to overwrite the hard drive of the netbook with our raw kernel image. Now we can boot from a proper hard drive rather than USB.

This approach had the major downside of being slow; we had to boot the weak netbook into a Linux distribution and enter several commands manually every time we wanted to test a new iteration of our kernel. This slowed development speed down significantly.

Additionally, the boot loader continued to get stuck.

9.4 No SATA support

After more debugging we determined the problem: the hard drive in the netbook uses a SATA connection, while the machine emulated by QEMU used ATA. Thus reading the kernel from disk was failing.

Fortunately, ATA and SATA are almost identical. ATA uses PIO to communicate with the disk on fixed ports. SATA uses the same protocol; the only difference is that the ports are machine-specific and must be found with PCI.

So we tried to add PCI support to our boot loader but ran out of space – we only had 512 bytes to work with, after all. As a temporary workaround, we booted into a Linux distribution and used the "lspci" tool to figure out the SATA I/O ports and hardcoded them.

With that, the boot loader finally succeeded in loading the kernel, which promptly broke; the scheduler was failing to switch in new applications.

9.5 A better boot loader

We attempted to debug the scheduler, but our development process was too slow and unwieldy to get anywhere; as mentioned, every time we made a change to the kernel code we had to boot into a Linux distribution and enter commands to write the raw kernel image to the hard disk. We needed to get around this; ideally we would simply insert a USB drive and immediately boot into our kernel.

To facilitate booting from USB, we replaced our boot loader with a better one. GRUB is the gold standard for boot loaders; it supports booting from various hard drive types, USB, and even booting via an ethernet connection. After integrating GRUB into the project, we were finally able to boot directly from USB, and development sped up significantly.

Our custom file system assumes that the boot loader will fit into the first 512 bytes, with the superblock residing in the following sector. However as GRUB uses multiple stages, it needs much more space. We therefore had to modify our file system such that the first 32 MB are reserved for GRUB, and the superblock resides thereafter.

9.6 Finding the LAPIC

Usually the LAPIC generates clock interrupts periodically, and on such an interrupt the scheduler switches in a new process. However no clock interrupts were generated, and thus only one process got any processing time.

In section 5.1 we described how our kernel uses the so-called mpconfig method to find the LAPIC; that is, it looks for an MP configuration table in memory to find the physical address of the LAPIC. This physical address is used to communicate with the LAPIC, asking it to generate clock interrupts. However our kernel failed to find these MP configuration tables. They were simply not present in the memory of the netbook.

It turns out that the mpconfig method is outdated and unsupported by modern hardware. We therefore had to implement a different way to find the LAPIC.

We leave out the details of the method, but in essence newer machines contain so-called ACPI tables. One of these is the APIC table, which contains the physical addres of the LAPIC. The ACPI tables can be found by scanning a region of memory for a data structure called the RSDP, which points at another data structure, the RSDT, which finally points at the ACPI tables.

After implementing this finding and parsing of ACPI tables, the kernel found the LAPIC and clock signals were generated, letting the scheduler work as intended.

Then another bug was uncovered.

9.7 A page table bug

Our kernel was behaving oddly; modifying a PTE seemed to have no effect. After the modification, writing to memory at the virtual address still affected memory at the old physical address rather than the new one.

We figured that this must be related to the TLB, since this seemed to be a caching issue. After much searching we learned that the `invlpg` instruction must be used to invalidate a TLB entry after a PTE has been modified. Otherwise the outdated cached TLB entry will continue to be used until it is evicted.

Interestingly, this bug never surfaced while running the kernel in QEMU. We assume that QEMU does not faithfully emulate the TLB for performance reasons.[1]

9.8 Broken mouse driver

At this point the kernel successfully booted into a graphical interface. However the mouse was behaving oddly, jumping around when moved. The PS/2 mouse driver we wrote during the graphics lab was at fault; the driver uses PIO to read from the mouse. Before each `inb` instruction, it is necessary to wait for the mouse to signal that it has sent another byte of data. The driver did not do this. However in QEMU these waits were not necessary; thus this was another instance of QEMU not emulating hardware perfectly.

9.9 The final result

With all the changes and fixes described so far, the kernel finally ran on our netbook.

10 Future work

11 Conclusion

- conclusion

> Lesson learned: foo
>
> Maybe we can use something like this to describe the various lessons we learned...

[1] VirtualBox behaved similarly to QEMU; we therefore believe that this quirk can be used as a means to detect virtualization/emulation.

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, 14

`eip` Extended Instruction Pointer; a register which holds the address of the next instruction to be executed.. 7, 8, 12

`esp` Extended Stack Pointer register.. 7, 8, 12, 21

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, 20

BP Bootstrap Processor; the first, and initially only, processor that runs when a system boots.. 10, 20

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, 20

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, 20, 21

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, 20

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

IPC Inter-Process Communication. A mechanism which lets processes communicate.. 13, 14, 18

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 systems have one LAPIC per processor.. 9, 10, 20

LFB Linear Frame Buffer, a buffer that repesents the pixels which are drawn to the screen.. 17–19

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, 15, 21

MMU Memory Management Unit.. 4

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

PCI Peripheral Component Interconnect. A type of bus to which devices, such as the network card, can be connected.. 15

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, 20

PIO Programmed Input/Output. A way to read and write data from/to disk (or another device) using instructions such as `inb` and `outb`. Often an alternative to MMIO.. 2, 3, 14, 15, 18

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, 12

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, 21