Operating System Implementation

Thomas Hybel
Aarhus University
November 2017

Abstract

TODO: write an abstract here.

CONTENTS

1 INTRODUCTION 1

2 BOOTING 2
- 2.1 The boot process 2
- 2.2 Initialization code 2
- 2.3 The boot loader 2
- 2.4 Minimal kernel code 3

3 MEMORY MANAGEMENT 4
- 3.1 Physical page management 4
- 3.2 Page table theory 4
- 3.3 Page table management 5

4 USER SPACE 7
- 4.1 Managing process metadata 7
- 4.2 ELF loading 7
- 4.3 Context switching 7
- 4.4 Handling of exceptions and interrupts 8
- 4.5 Handling of system calls 8

5 MULTIPROCESSING 9
- 5.1 The process scheduler 9
- 5.2 Activating more processors 9
- 5.3 Ensuring mutual exclusion 9
- 5.4 Interacting with the LAPIC 10
- 5.5 A simple fork mechanism 10
- 5.6 Copy-on-write fork 11
- 5.7 Inter-process communication 12

6 FILE SYSTEM 13
- 6.1 File system design 13
- 6.2 File system daemon 13
- 6.3 Shell 14

7 NETWORKING 15
- 7.1 The network card 15
- 7.2 The network card driver 15
- 7.3 The network daemon 16
- 7.4 Web server 16

8 GRAPHICS 17
- 8.1 Drawing pixels 17
- 8.2 Graphics stack design 17
- 8.3 Details of the display server 18
- 8.4 Sample graphical applications 19

9 HARDWARE 20
- 9.1 Booting from USB 20
- 9.2 No serial connection 20
- 9.3 No USB driver 20
- 9.4 No SATA support 21
- 9.5 A better boot loader 21
- 9.6 Finding the LAPIC 21
- 9.7 A page table bug 22
- 9.8 Broken mouse driver 22
- 9.9 The final result 22

10 FUTURE WORK 23

11 CONCLUSION 24

Abbreviations 26

1 INTRODUCTION

This report documents our journey through writing an operating system from scratch. Our primary goal was to learn about operating systems internals and design, and to investigate the difficulty of the task. Neither usability nor efficiency were significant goals throughout the process, except when their consideration facilitated learning.

To get started on kernel development we made extensive use of material from the MIT 2016 Operating System Engineering course, which is accessible online. The MIT course is divided into six major "labs", with each lab encapsulating a set of kernel functionality. For example, lab 4 involves functionality needed for multitasking and concurrency, while lab 5 comprises the implementation of a custom file system.

The kernel we wrote is for the 32-bit x86 architecture. The kernel is written in C, with some parts having to be hand-written in assembly. We have used GCC as our compiler and Git for version control. Additionally, it was convenient to develop the kernel on emulated hardware rather than a real machine, since this eased debugging and increased development speed. It also made it trivial to change the amount of system RAM, and to add new processors or an extra hard drive. We have used the full-system emulator QEMU for this task.

Each MIT lab has an associated web page which describes what needs to be done and provides links to resources like manuals and specifications. Since we opted to follow the MIT course, this also meant that some design decisions were made for us. One example of this is us following an exokernel philosophy, which meant that we have pushed large parts of the kernel functionality into user space.

During kernel development, occasionally some functionality was important, but not technically interesting to implement. In these cases the MIT lab often provided the code for us, with enough code missing that we had to understand the concept to complete it, while still saving significant amounts of time. An example of this is the code for communicating with the outside world over a serial connection, which is used throughout the labs.

In addition to the six labs of the MIT course, we have decided upon two labs on our own. In lab 7 we designed and implemented a graphical user interface for the operating system. In lab 8 we made the operating system run on real hardware instead of QEMU. Since we were given zero guidance nor code, these labs were significantly more difficult and time-consuming than previous ones.

Each of the following sections corresponds directly to one lab. They describe the development process, from first booting into a minimal kernel, until the end when we run a full graphical operating system on a real machine.

2 BOOTING

In this lab we wrote initialization code and a boot loader for our kernel. The initialization code switches the processor to 32-bit protected mode. The boot loader loads the rest of the kernel into memory and jumps to it.

2.1 THE BOOT PROCESS

To understand the purpose of a boot loader, it helps to have an overview of the startup process of an x86 machine. When an x86 machine starts, its BIOS code runs. The BIOS initializes some of the hardware, and then it loads the first sector (512 bytes) from the boot medium into a hard-coded address which it jumps to.

This 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, the kernel should 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. The initialization code should therefore be written in assembly and should switch processor modes.

The most important difference between real and protected mode is that real mode does not support the use of a page table to implement virtual memory.

We switch to protected mode by setting a bit in the control register `cr0`. Enabling virtual memory similarly works by setting another bit in `cr0`. We do not immediately enable virtual memory, though, since we do not yet have the infrastructure to set up a proper page table. This happens in section 3. Until then, all addressing is physical.

To switch to 32-bit mode, we must update the `cs` (code segment) register. The `cs` register is an offset into a table called the Global Descriptor Table (GDT) which is an array of descriptors. Each descriptor describes a segment of memory; a bit in the descriptor determines whether the segment contains 16-bit or 32-bit code.

We use the `ljmp` instruction to update the `cs` register and use a 32-bit segment descriptor. Next, the boot loader is executed; since the processor is in 32-bit protected mode, the rest of the kernel code can be written in C rather than assembly.

2.3 THE BOOT LOADER

It is the task of the boot loader to load the main kernel and transfer execution to it. Since the boot loader resides on the first sector of the hard drive, it must load the kernel using no more than 512 bytes of code, minus the bytes used by the initialization code.

Our kernel is compiled into an Execute and Linkable Format (ELF) file. The ELF file specifies the physical addresses into which the code and data of the kernel must be loaded. The boot loader must thus parse the ELF file to load the kernel.

Once the boot loader has loaded the kernel, it determines the entry point address from the ELF file and jumps there.

2.4 MINIMAL KERNEL CODE

At this point we were able to run kernel code. However we did not yet have any 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 wanted the ability to input and output text. Our kernel uses the `inb` and `outb` instructions to communicate with the outside world via a serial connection. We used this to print a "hello, world" message and confirm that the kernel runs.

Using instructions such as `inb` and `outb` to communicate with an input/output device is quite common. The method is known as Programmed Input/Output (PIO); it will be used often in the following sections.

3 MEMORY MANAGEMENT

The goal of this lab was to enable virtual memory after setting up a page table. To set up the page table, we first needed to implement a subsystem for allocating and freeing pages of physical memory.

3.1 PHYSICAL PAGE MANAGEMENT

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, or `0x1000` in hexadecimal. Thus pages always aligned on `0x1000`-byte boundaries. The kernel determines the amount of RAM by using PIO to query a memory area called the CMOS.

The physical page management subsystem will keep a reference count for each page. If the count is zero, the page is free and can be allocated. This metadata is stored in an array of `PageInfo` structs. Each entry in this array directly corresponds to one physical page of memory, such that the first `PageInfo` struct holds metadata about the first page of physical memory, and so on.

The `PageInfo` of free pages are additionally stored in a linked list, such that the kernel can 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

These functions provide critical infrastructure needed by other kernel features.

3.2 PAGE TABLE THEORY

It is necessary to introduce some theory before we can explain how our kernel initializes its page table.

The x86 page table is a two-level table whose main purpose is to let the processor translate a virtual address to a physical address. The page table can be thought of as a 1024-ary tree with two levels.

A pointer to the first level of the page table can be found in the `cr3` register. The first level is called the Page Directory. It contains 1024 Page Directory Entries (PDEs). Each PDE points to a second-level node, which contains 1024 Page Table Entries (PTEs). A PTE specifies a page of physical memory and its permissions, including whether it is writable and whether it is accessible to user-mode code.

To translate from a virtual to a physical address, it is necessary to walk the page table. Say that the process wishes to access a virtual address v. It first looks in the `cr3` register to find the Page Directory. It uses the 10 higher-order bits of v to specify a PDE. It uses the next 10 bits of v to specify a PTE. The PTE contains the address of a physical page. The last 12 bits of v are used as an offset into this page, and the address translation is complete. The processor also validates the permissions of the page before the access, and generates a page fault if these are inappropriate.

This is a costly process, and in practice the job is done by specialized hardware called a Memory Management Unit (MMU). Additionally, recent translations are cached in the so-called Translation Lookaside Buffer (TLB).

```
Virtual memory map:                                 Permissions
                                                    kernel/user
   4 Gig -------->  +------------------------------+
                    :              .               :
                    :              .               :
                    |------------------------------| RW/--
                    |                              | RW/--
                    |      Kernel code, data       | RW/--
                    |                              | RW/--
   KERNBASE, ---->  +------------------------------+ 0xf0000000
   KSTACKTOP        |     CPU0's Kernel Stack      | RW/--  KSTKSIZE
                    +------------------------------+
                    |     CPU1's Kernel Stack      | RW/--  KSTKSIZE
                    +------------------------------+
                    :              .               :
                    :              .               :
                    +------------------------------+ 0xef800000
                    |  Cur. Page Table (User R-)   | R-/R-  PTSIZE
   UVPT      ---->  +------------------------------+ 0xef400000
                    |           RO PAGES           | R-/R-  PTSIZE
   UPAGES    ---->  +------------------------------+ 0xef000000
                    :              .               :
                    :              .               :
   USTACKTOP  --->  +------------------------------+ 0xeebfe000
                    |      Normal User Stack       | RW/RW  PGSIZE
                    :              .               :
                    :              .               :
                    +------------------------------+
                    :              .               :
                    :              .               :
                    +------------------------------+
                    |     Program code, data       |
   UTEXT -------->  +------------------------------+ 0x00800000
                    :              .               :
                    :              .               :
                    +------------------------------+ 0x00000000
```

Figure 3.1: The virtual address space of the kernel

3.3 PAGE TABLE MANAGEMENT

We wrote the following functions to manage the page table:

- `pgdir_walk` is called by most of the other functions to walk the page table, finding the PTE corresponding to a given virtual address. It allocates new levels of the page table as 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 using `pgdir_walk` 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.

To set up the page table, the kernel allocates pages of physical memory and uses the newly implemented functions to insert these into the table.

The memory layout of the address space of the kernel is largely up to us. Figure 3.1 gives a simplified overview of the layout we opted for. The diagram shows that the 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 processes read-only access to their page table, enabling certain exokernel-style programs to work. The stack of the user-mode program starts at `0xeebfe000` and grows towards lower addresses. The user-mode code and data reside near the bottom of the address space, around `0x00800000`.

We created a page table according to this layout. We then updated the `cr3` register and set a bit in the `cr0` register to enable the use of the page table. At this point our kernel had proper virtual memory. Once user-mode processes have been implemented, virtual memory guarantees that the processes cannot modify the address space of one another, nor can they corrupt the kernel.

4 USER SPACE

The goal of this lab was to run a user-mode process. This required us to write infrastructure for managing processes and their metadata. We also modified our kernel to handle any exceptions generated by user-mode processes. Finally we implemented a system call mechanism to let processes interact with the kernel.

4.1 MANAGING PROCESS METADATA

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

4.2 ELF LOADING

Programs are represented as ELF files. To launch a process, the kernel first allocates a fresh page table. It then walks over each section in the ELF file, reads at which virtual address the section should go, allocates corresponding physical pages, inserts them into the page table, and copies the code or data 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 section 6 we describe our implementation of a proper file system for storing programs and data.

4.3 CONTEXT SWITCHING

Once a program has been loaded, the kernel must perform a context switch to let the created process run. During a context switch the kernel restores the saved general-purpose registers using the `popal` instruction. It also updates `cr3` to switch to the new address space. Finally it uses the `iret` instruction to restore the saved `eip`, `esp`, `eflags`, and `cs` registers. This transfers execution to user-mode code.

The kernel must also drop its privileges. The lower two bits of the `cs` register determine the privilege level 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, which is ring 3. Since instructions such as `iret` may only be run from privilege level 0, the process cannot simply modify its `cs` register to increase its privileges.

At this point our kernel was able to successfully create a new process, load program code and data into its address space, and let the process run. 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. Since this was not yet handled, any exception caused the whole system to crash.

4.4 HANDLING OF EXCEPTIONS AND INTERRUPTS

The processor may trigger an exception while processing an instruction, e.g., on a division by zero or illegal memory access. The processor may also occasionally trigger an interrupt. This often happens for asynchronous reasons, such as when a key is pressed or a network packet is received. Interrupts can also be raised with the `int` instruction; this mechanism can be used to implement system calls.

The result of an exception or interrupt is that the processor enters kernel mode through a context switch, running exception- or interrupt-specific handler code. This handler code is found as follows. Exceptions and interrupts are numbered. This number specifies an index into a table called the Interrupt Descriptor Table (IDT), which can be found using the IDT register, `idtr`. The IDT entry describes the address of the handler code.

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 and calls `trap`.

The typical result of an exception is that the kernel terminates the running process. If an exception occurs in kernel mode, the result is a kernel panic, where the kernel prints an error message and hangs.

4.5 HANDLING OF SYSTEM CALLS

A process frequently needs to call kernel code, e.g. to print text onto the screen or to perform inter-process communication. This is called a system call.

One mechanism to implement system calls on the x86 architecture is to use the `int` instruction, which triggers an interrupt when executed. We are free to use any interrupt that is not in use, so we arbitrarily chose number `0x30`. Thus a program uses the `int 0x30` instruction to perform a system call. We chose a fairly standard ABI; 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 kernel system call handlers for input and output of a single character, as well as for process termination. User-mode programs did not have a C standard library at this point, so we created one, adding an interface to the system calls to the library. At this point we were able to run a simple user-mode program and have it interact with the user.

5 MULTIPROCESSING

In this lab we added the features necessary for running multiple processes concurrently and having them interact.

We implemented a simply round-robin scheduler. The scheduler can preempt the running process when its time slice is up with the aid of a device called a LAPIC, which can generate periodic timer interrupts.

We also wrote code to activate any processors beyond the first, letting processes run truly concurrently.

We implemented a naive fork mechanism which lets proceses spawn more processes. We then upgraded its efficiency with a copy-on-write mechanism.

Finally, we added an inter-process communication feature, letting processes send values and pages of memory to one another.

5.1 THE PROCESS SCHEDULER

The main purpose of a scheduler is to decide which process to schedule in when the running process is scheduled out. There are many ways to make this decision, and the scheduler has a great impact on system responsitivity and efficiency.

However for simplicity reasons, we opted for a simple round-robin scheduler. That is, the scheduler keeps a circular queue of all processes. The first runnable process in the queue is chosen to be scheduled in.

The scheduler also needs to preempt each process when its time slice is used. To accomplish this, during kernel initialization the kernel asks a device called the LAPIC to raise a timer interrupt periodically, waiting some fixed amount of bus cycles between each raised interrupt. Our `trap` function recognizes this timer interrupt and reacts by asking the scheduler to schedule in a new process. Interaction with the LAPIC is described in section 5.4.

5.2 ACTIVATING MORE PROCESSORS

So far the kernel has run on an emulated single-core machine. However a machine 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 only a single processor to run. It is called the Bootstrap Processor (BP) while the remaining processors, if any, are called Application Processors (APs). It is up the the BP to start up the APs when the system is ready.

We wrote code which starts up the APs. This is accomplished by querying the LAPIC of the BP, having it send an inter-processor interrupt to each AP. Upon receiving such an interrupt, the APs start executing code at an address specified by the AP.

The APs start in 16-bit real mode, just as the BP did. They therefore need to switch to 32-bit protected 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. Multiple processors could modify kernel data simultaneously, leading to race conditions.

We prevented this with a trivial but inefficient approach: we added a lock which must be locked before running any kernel code. Thus only one processor may run kernel code at a time. Still, user-mode processes can run truly concurrently.

The kernel lock is a spinlock. It is a global variable. A processor repeatedly uses the `lock` and `xchg` instructions to atomically exchange the global variable with the value 1. If the old value was zero, the processor now holds the lock and may enter the kernel. Otherwise it retries.

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.

5.4 INTERACTING WITH THE LAPIC

In section 5.1 the scheduler had to ask the LAPIC to generate timer interrupts for it, and in section 5.2 the BP used the LAPIC to send inter-processor interrupts to the APs. We therefore needed code to query the LAPIC.

First, however, we explain the acronym. A Programmable Interrupt Controller (PIC) is a hardware device 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.

The processor communicates with the 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. Thus 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, which we will refer to as the mpconfig method. It involves searching through parts of physical memory to find a structure called the MP floating pointer structure. This structure points to a table called the MP configuration table, which contains the physical address of the LAPIC.

5.5 A SIMPLE FORK MECHANISM

So far, every process was directly spawned by the kernel. However, a process should also be able to spawn processes. We therefore needed 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 keep code 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.[1]
- `sys_page_map` maps a page from the current process into a child process.

[1] For security reasons, the system calls are coded to ensure that a process cannot modify pages in unrelated, non-child processes.

- `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 we set up the page table of a process such that part of the address space contains its page table.

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 which 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, it will eventually be scheduled in.

5.6 COPY-ON-WRITE FORK

The fork mechanism 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 reading and writing of RAM, 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 a page fault will result in process termination. However we have implemented a system call, `sys_env_set_pgfault_upcall`, which lets a process set a handler function. Then 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.

When a process forks, *all* pages are initially 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.

When the child or parent process attempts to write to one of the now-shared pages, a fault will occur since the page is not writable anymore. The kernel sees that the copy-on-write bit is set, and then it delegates to the registered user-mode handler. 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 this project follows the exokernel philosophy, many future features will reside in user land. Two examples are a file system daemon and a daemon implementing a network stack. Other processes need an inter-process communication (IPC) mechanism to make use of these daemons.

We therefore implemented two system calls, `sys_ipc_recv` and `sys_ipc_try_send`. When a process calls `sys_ipc_recv` it will hang, waiting to receive 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 per system call.

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, and 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 user-mode shell program.

6.1 FILE SYSTEM DESIGN

Having a file system lets programs store data persistently on disk. It also provides a storage place for programs, instead of embedding them directly in the kernel as we have done so far. In this section we describe our design of a custom file system.[1]

A file system can be thought of as a way to manage how files, folders, and their metadata are stored on a disk. A raw disk is essentially a portion of memory which can be read from and written to. It is customary to partition the memory of a disk into fixed-size blocks. For our file system, the block size will be 4096 bytes.

One of these blocks is special, since it contains metadata for the file system. This block is called the superblock. This metadata includes such things as the disk size and where to find the root folder.

Our file system is laid out as follows. Block 0, is not used by our FS; it is reserved for the boot loader. Block 1 is the superblock. The next few blocks, starting at block 2, hold a bitmap which determines whether the remaining blocks on disk are in use or free. The remaining blocks are used to store the concrete files and folders.

A file is represented as a `struct file`, which is stored in its own block. Such a `file` struct contains metadata, such as the file name and size. The struct also has 10 pointers to 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.

A folder is 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 which allows distinction 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 user-mode process which we call the file system daemon.

The daemon interacts with the disk using PIO. However a normal user-mode process 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 while spawning the daemon.

The disk knows nothing of the file system which is stored upon it. The daemon can merely read a sector of the disk at a time. It is therefore up to the daemon to implement reading and writing of files and folders according to the file system specification. We wrote the necessary functions for interacting with the file system.

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,

[1] To keep our file system simple, we have left out many potential features, such as file and folder permissions, symbolic and hard links, and timestamps.

and stat files. The IPC details are hidden inside the user-mode C standard library, giving processes the usual interface with functions such as `open`, `read` and `write`.

For sake of illustration, the following happens when a process wants to open a file:

- The process calls the `open` library function.
- The library uses IPC to contact the file system daemon.
- The daemon reads the superblock to find the block of the root folder.
- The daemon follows pointers from folder to folder according to the given path.
- When the path has been traversed, the daemon has found the block that contains the `file` struct.
- The offset into the disk that holds the `file` struct is stored in a file descriptor.
- File descriptors are numbered; the daemon returns this number as the result of the IPC call.

When the process subsequently reads from the file descriptor, the daemon uses the pointers in the `file` struct to find the raw data and return it via IPC.

We have also implemented a block caching system to improve the efficiency of the file system daemon. 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.

6.3 SHELL

To test the interaction between programs and the disk, we implemented a simplistic shell which allows loading and execution of programs from disk. The shell allows reads from and writes to the disk through input redirection (<) and output redirection (>). The shell additionally supports piping of the output of one program into another program. A small program, ”cat”, lets the user read files from the disk. A sample interaction is shown in figure 6.1.

```
$ ls
cat
echo
ls
sh
$ echo "Hello, OS." > motd
$ cat motd
"Hello, OS."
```

Figure 6.1: A sample interaction with the shell

7 NETWORKING

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

7.1 THE NETWORK CARD

The operating system connects to a network using a network card, whose task it is to send and receive packets by interacting with the physical layer. QEMU emulates the Intel e1000 network card, which we therefore targeted. We added the card to our emulated machine with the "-net nic,model=e1000" option to QEMU, which creates a virtual router at IP 10.0.2.2 and assigns the guest an IP of 10.0.2.15. This let us hardcode the the IP address rather than having to implement a DHCP client or similar.

The e1000 manual describes in detail the internals of the card, as well as the steps a driver must take to initialize the card, read incoming packets, and transmit outgoing packets.

The e1000 maintains two circular packet queues: a transmit queue and a receive queue. To send a packet, the kernel adds it to the transmit queue, which the network card periodically drains. When a packet arrives on the wire, the network card adds it to the receive queue, which the kernel periodically drains.

The items in the queues are not raw packets, but rather small descriptor structs, each of which describes an area of physical memory which can hold a packet. The queues are implemented as arrays, with each queue having a head and tail register to keep track of the head and tail of the queue.

7.2 THE NETWORK CARD DRIVER

We wrote an e1000 network card driver using the information from the manual. Our kernel uses PIO to scan the PCI bus, iterating over each connected device until it finds one whose vendor and device ID indicate it being an e1000 network card. The PCI interface supplies a physical address where the driver can use MMIO to communicate with the e1000.

With communication possible, the first task of the driver was to initialize the network card. The driver allocates the arrays that make up the packet queues and fills these with valid descriptors. It zeroes out the head and tail registers.

Once our driver has initialized the card, it writes into a register to enable it. It is then 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_receive` and `sys_transmit`, respectively.

If the transmit queue ever becomes full due to the card draining it too slowly, it is up to the driver what should happen. For simplicity we have chosen to simply drop the packet if the queue is full. This is not a problem because the higher-level protocols are resistant to packet loss. Alternatively the driver could have blocked, waiting for space in the queue. If, on the other hand, the receive queue is full, the e1000 is designed to simply drop further packets.

7.3 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 requests to send data via IPC, produces packets, and hands these over to the operating system via the `sys_transmit` system call. Likewise, it takes in requests to receive data and uses `sys_receive`, parses the resulting packets, and hands over the data to the requesting process.

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

7.4 WEB SERVER

To test the new functionality, we wrote a simple web server which can serve files from the file system over HTTP. The web server uses the network daemon to accept incoming connections and receive HTTP requests. It parses each request, queries the file system daemon to retrieve the requested file contents, and uses the network daemon to send back the HTTP response.

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

In this lab we implemented a graphical user interface to our operating system, with each graphical application havings its own window, and the user being able to interact using a mouse.

The first step to construct a GUI was the ability to draw a pixel to the screen. We then used this primitive as a building block to draw more complex shapes, such as rectangles, windows, etc.

With the ability to draw pixels, we designed a graphics stack. This involved making various decisions, e.g., whose responsibility it is to draw pixels, and how input events should flow through the system from the kernel to user space. We decided upon a design with a central display server which is responsible for drawing all other applications and for forwarding input events from the kernel to the active application.

Finally we added a few sample graphical applications to demonstrate the new features.

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

In section 5.4 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 timer 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 timer interrupts 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.

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

10

FUTURE WORK

11 CONCLUSION

- conclusion

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

`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