Designing a functional and minimalistic Operating System

August 29, 2012

Abstract

In this paper we will attempt to define an operating system and the core components modern operating systems implement. We will give examples to illustrate the complexity of these systems. We will also describe which components should minimally be implemented to be considered usable, with the focus being process scheduling. After that we will give global implementation considerations and a design focusing on small systems like microprocessors.

Contents

1 Introduction

With the latest trends of processing power and memory becoming cheaper and cheaper, software grows in a similar fashion of requiring more and more of it. Another effect is that microprocessors are becoming more powerful and cheaper. Nowadays, simple cheap microprocessors are available that are more powerful than the Intel 8086 which once ran MS-DOS [1] , at a fraction of the price. Deployment of a microprocessor is however a very technical task: because of their minimal nature a programmer needs to know the specific platform well.

Another reason for this might be the fact that these microprocessors are not popular platforms for operating systems designed to make it easier for programmers to deploy higher-level software. One can argue that most modern operating systems require much more power and memory than a simple microprocessor can deliver, but this raises a question: why? Like stated earlier, weaker processors used to ran operating systems just fine. Operating systems for minimal systems and microprocessors do exist, proving that it is possible. This raises even more questions: What makes these minimal operating systems so special that they run on microprocessors? What components that modern operating systems include are scrapped from these micro counterparts? To combine all the questions raised in one, we present the following main question of this research: *How to design a minimalistic but still functional operating system with as little memory and processor usage as possible.*

This paper attempts to build a general idea of operating system theory in the bare minimum, by designing core components of an operating system in a way that they are even runnable on a simple cheap microprocessor. To do this, we first need to define operating systems, what makes them so complex, and the core tasks they are responsible for. We have to argue which ones we really have to implement, and design a general implementation for these. Do note that we are aiming for a core understanding of these concepts and not a truly usable, working implementation. In fact, there will be no real implementation at all: we will merely focus on the design principles and considerations that come with actual implementations. Due to the overall amount of complexity, we will focus on processor scheduling.

[1] The Atmel ATmega32 is able to run at frequencies up to 10 MHz while most of its instructions require a single clock cycle to perform. Straightforwardly comparing this with the 8086, which had a maximum frequency of 10 MHz and instructions could take up to 6 cycles, it seems the ATmega32 has to be at least as fast, but likely to be much faster of the two. Do note this is only a theoretical estimation.

2 Core tasks

Before we can continue diving into implementations, it is important to define what an Operating System actually needs to implement to be regarded as functional and usable. We will first give a list of basic tasks common operating systems are responsible for according to a well used book about basic operating system theory (Stallings, 2009, Ch.2) [16]. Afterwards, we will list the most important features of several popular modern operating systems. Concluding we will discuss what features we find important enough to implement.

2.1 Objectives and Functions

Stallings calls the OS a program that controls execution of applications and an interface between the applications and hardware, and names the following three objectives [16]:

- Convenience.
- Efficiency.
- Ability to evolve.

He names the following services to provide for convenience:

1. Program development: tools and APIs to aid developers.
2. Program execution: load program and data in memory but also to initialise resources. See the chapter about *Executable Formats* for additional information about this and why it is not a trivial task.
3. Access to I/O devices: a uniform way to access the numerous different kinds of I/O devices.
4. Controlled access to files: a uniform way to access files on the I/O devices that contain file systems. Multiuser systems should also provide mechanisms to control file access.
5. System access: a system implementing protection of resources and data from unauthorised usage that can also resolve conflicts with shared resources.
6. Error detection and response: when any kind of error occurs, the system should detect it and respond in a way that minimizes impact.
7. Accounting: an operating system should also collect usage statistics for analysis of stability and performance.

Regarding efficiency, the operating system needs to manage the resources of the computer in an efficient sense, to maximise the amount of work possible. Regarding evolvability, it should be able to grow with new hardware and services, and will also require the possibility to issue fixes later on.

2.2 Modern implementations

Now that we have a basis, let us take a look at modern OS families, and see how they implement the services: numbers in round brackets indicate a service from the list.

2.2.1 Microsoft Windows®

According to (Russinovich, Solomon) [2], Windows uses a modular structure that is highly API-driven. The OS itself implements an executive (controls basic OS services like memory and thread management) (5), a basic kernel (which controls processor execution), a HAL (Hardware Abstraction Layer, to provide a uniform way to communicate with different platforms and hardware) (3), device drivers (extend the executive for new types of hardware), and a graphical user-interface, with a lot of standard user-applications.

Comparing with the list from the previous section, the operating system has its own set of APIs, runtime environments extending these API capabilities and development tools (Microsoft offers their Microsoft Visual Studio as a different product, however.) to aid developers (1). The Windows API provides a large list of distinct utilities:

- Base Services
- Component Services
- User Interface Services
- Graphics and Multimedia Services
- Messaging and Collaboration
- Networking
- Web Services

It supports its own executable formats (2): in the 16bit days it used relatively simple *DOS EXEs* containing fixed-sized blocks and paragraphs giving relocation possibilities [23]. Since the 32bit releases it uses its own Portable Executable *PE* format, containing an EXE header for backwards-compatibility, but extending the capabilities to dynamic linking at runtime, containing sections which are marked as executable or data (for security purposes), allowing imports and relocations [24]. See the chapter *Executable Formats* for more details. It also provides support for a few disk file systems: *FAT* and *NTFS* (4). The latter allows extensive user privileges and rights (5).

2.2.2 UNIX

Another major OS family is UNIX. There are several popular Unix-like operating systems still in use, of which we will name three: Linux (we will consider only the kernel), BSD (*Berkeley Software Distribution*, there are several closely related kernels like FreeBSD, OpenBSD and NetBSD, we will consider the first) and OS-X (which is also closely related to several BSD families since it shares a lot of code, but it also contains a lot of proprietary additions by Apple, therefore we will consider the complete OS in this case). All UNIX-like systems show large similarities since they are designed off the same system.

Linux One of the popular UNIX implementations is the Open Source kernel Linux. Although the Linux kernel is monolithic (designed as one big whole), it is still structured as modules which are loadable and unload-able from the kernel at runtime. These modules are, however, still executed in the main kernel mode thread [16]. Device drivers are implemented as modules (3). Kernel modules have been developed to support a wide range of file systems, from its own *ext* file system to *FAT*, *NTFS*, *UDF*, *HFS* and several others (4). The OS also implements file ownership and permissions (5).

Communication between the Linux kernel and applications is achieved by *signals* and *system calls*: the kernel can notify processes of certain events (like errors) by signals [16] (6), while processes can request kernel services through system calls [5]. These system calls can be grouped in six categories [16]: file system, process, scheduling, interprocess communication, socket and miscellaneous. The programmer can call them using interrupt *0x80* which transfers control directly to the kernel, or use API wrappers to call them [6] (1).

Linux supports multiple executable formats to load programs in (2). Consider for example the *a.out* format which supports symbol tables and relocations [7], and the *Executable and Linking format* (ELF), which in essence is an object format which can according to the (*System V Application Binary Interface* edition 4.1 section 4, p. 45) [8] either be a *relocatable*, *executable* or *shared object* file.

FreeBSD FreeBSD is based on 4.4BSD, and is in kernel-aspects very similar to Linux, since they are both UNIX-like. The BSD kernel is monolithic and designed to provide the following basic facilities: processes, a file system, communication and system startup. Just like any UNIX system kernel to application communication goes by signals (6) and system calls [9] (1). Kernel modules also implement device drivers (3).

The BSD system kernel services that are usable through system calls are organized as follows [9]:

- Basic kernel facilities (like process management)

- Memory-management support
- Generic system interfaces
- Filesystem
- Terminal-handling
- Interprocess-communication
- Network communication

The kernel provides an executable loader for the same binary formats as Linux (*a.out*, *ELF*) [10] (2), and even provides the possibility to execute binaries written and compiled specifically for Linux [11].

Modules for several file systems exist for the FreeBSD kernel, including its native *FFS* and *UFS*, but also several Linux file systems for instance *ext*, and other file systems like *ZFS* [12] (4), which also provide user ownership and permissions (5).

Mac OS-X® Apple's Mac OS-X® is structured modularly, consisting of a basic Open Source Kernel environment (called Darwin); on top of that they implement Core Services, FreeBSD user applications and Application Services that implement communication for applications between the kernel [13]. Darwin is based on the FreeBSD and Mach kernels, and also supports dynamic loading of objects into the kernel at runtime. Specifically, the OS-X kernel consists of three parts which provide different services and APIs with different goals [13]:

- Mach manages resources and provides a messaging infrastructure between the rest of the OS layers.
- BSD provides on top of this file system support, networking, UNIX security, syscall support, processes, the FreeBSD kernel API's.
- I/O Kit provides a modular and extensible driver architecture.

Many of the core concepts of OS-X are similar to BSD and Linux, such as the basic kernel facilities, security, signal and syscall communication (however, the Application Services APIs offer a different approach to kernel-application communication on top of that) (1), and similar user permission systems (5).

File systems are also implemented in the BSD layer, but different file systems are supported: Apple's own *HFS*, *HFS+*, and several others like *UFS* and *FAT* (called *MS-DOS* in the documentation) [13] (4). Device drivers can be offered as I/O Kit or kernel extension, of which the first is recommended when possible [13] (3).

OS-X uses its own *MACH-O* binary format, providing flexible static and dynamic linking support. OS-X made a transition from the *PowerPC* to *Intel* platform (but also the transition from 32 to 64 bit) and therefore it implemented support for several architecture targets in one binary. Mach-O binaries can either be targeted for PowerPC (*PPC/PPC64*) or *Universal*, which allows Intel only or both targets [14].

2.3 Discussion

Now that we have an overview of abstract global tasks and lists of concrete examples implemented in modern operating systems, we can discuss which of these we find important to actually implement. Remember that the goal of this research is not to implement a fully functional, modern and usable OS, but rather a minimalistic one containing the core features. This constraint gives us more freedom in both complexity and time.

Considering the global overview we discussed earlier:

1. Program development: we regard this as an extra: the OS will run on its own with no external software and is intended to study operating system internals, thus tools to aid developers to create external software are not required. For programs to do anything at all they do need to be able to communicate with the kernel, thus for a truly functional OS it is important to implement (something similar to) UNIX-signals

and system calls. A reason to choose for this type of communication is that its Linux and FreeBSD implementations and systems are widely documented. The system calls most important to implement are mostly to make the next services in this list usable by programs (thread execution, file access, memory allocation), but because one might also want to be able to observe its effect one could also implement the most simple form of this: basic terminal (screen) text I/O.

2. Program execution: we regard this as an important service: this not only allows the OS to be more modular but also to execute programs which is one of the most important things for a computer to do in general. Since only the concept is required and one does not have to aim for advanced features, a basic choice would be to implement the rather limited but simple binary format *a.out*: it provides basic executable loading and linking but is still relatively simple, which makes it well suited for a proof of concept.
3. Access to I/O devices: a computer without any I/O is very limited, thus this is important to implement for a functional OS. Do note that a screen and keyboard are I/O devices as well, which are required for an operating system to be interactive. The bare minimum is to be able to print a character to the screen, and to be able to request a key to be pressed (to read this character). A consequence of this requirement is that one requires a way of implementing device drivers. Since a newly designed operating system will probably not be designed and built in the same way as existing operating systems, it is not feasible to modify existing device drivers: they will have to be written from scratch.
4. Controlled access to files: since executable programs are files and I/O must be used in a sane way, it is consequence that one would also implement file-system support. However just like with I/O device support one would begin by limiting this to only one simple implementation of *ext* since this is a relatively simple system, while others are either too complex with unneeded features for a proof of concept, or heavily patented and/or licensed. Another possible advantage is that one can use public implementations of *ext*. The most basic system will be aimed at a single user and thus controlled file access is not required.
5. System access: memory, I/O and processor management are core services required by program execution and must be implemented. Multiple processes will be supported (this makes executing the kernel and user processes at the same time more practical), but we will not look at multi-processor or even multi-core support. There are more ways to implement this, and the exact details deserve chapters on their own later in research.
6. Error detection and response: this could be implemented in its most simple form: in case of a process error the process will be killed to minimize damage to other processes or the system in general, and if a serious error already compromised the system we will shut it down completely.
7. Accounting: we view this as something that multi-user operating systems require to be practical: historically this was implemented to bill the different users for their exact usage of the machine. Since our operating system will be for a single user, we will disregard this in whole.

3 Resource Management

As we stated earlier, one of the fundamental tasks for operating systems is to manage shared resources in an efficient way. Also, a minimal processor usage and memory footprint are fundamental components of our main question. Before we can go in depth about this rich subject, we first must clarify what we mean by this, and more specifically state what kind of resources it needs to manage.

3.1 Resources

Remember the previous chapter about tasks we need to achieve. These tasks depend on three kinds of resources that are not unlimited in usage (Tanenbaum, 2006) [15]: the processor [2] , the memory [3] and I/O devices [4] Since we only have a limited amount of these resources to allocate, we have to manage access in an efficient way. Let us start with processor allocation.

3.2 Processor resources

Modern operating systems allow multiple programs to execute at once [5], and even inside programs you might want to be able to run several calculations independent of each other, especially when considering the current trend of multi-core processors. To make all these concepts more organised, operating systems introduce abstractions as processes and threads.

3.2.1 Processes and Threads

Stallings names several definitions for the term process (a program in execution), however he distinguishes two essential kinds of program execution states: processes and threads, which are used depending on the concept they should achieve. Stallings names that processes are used for *resource ownership* (different processes can own or share resources) and *scheduling* (different processes can run at the same time) [16].

To make this distinction more clear, when talking about states of execution that deal with resources are called processes (usually a program that runs is called a process), while states of execution that focus on interleaved execution are called threads (consider this in a hierarchical sense: a process can have several sub-threads, a process can own a piece of memory and share this to his two sub-threads which share this but execute simultaneously [6]) Do note that it is also important that the operating system protects data and/or resources owned by processes: one program should not be able to interfere with resources used by another.

[2] Here, a limited amount of the *processor* might sound a bit strange, what we mean here is processor time: we can only do a limited amount of calculations in a given time.

[3] *CPUs have always been faster than memories* (Tanenbaum, 2006, p. 77) [15]: the processor is typically faster than the time needed to access the memory, and the amount of available memory is obviously limited as well (assuming we are not running on a Turing Machine with an infinitely long tape).

[4] Again, we mean access time to the I/O devices. *As we move down the hierarchy, ..., the access time gets bigger.* (Tanenbaum, 2006, p. 81) [15]: I/O devices are typically much slower than the processor and even the main memory, thus the amount of data we can send to or receive from I/O devices in a given time is also limited.

[5] It would be really inefficient otherwise: imagine having to close your web browser every time you want to copy some information to a text document because your computer cannot run the two programs at the same time.

[6] Consider a program that does calculations, reads input from a file from a slow floppy drive, listens to user input and updates the screen. Also consider the fact that this screen updating might be expensive (3d renderings). We don't want this program to stop responding to user input during the calculating and screen updating, neither do we want to lose any responsiveness when we are waiting for data from the slow floppy drive, thus we employ different threads for this.

3.2.2 Multi-threading

The goal of all these abstractions is of course to offer a structured way of executing multiple programs at the same time, possibly on a true multiprocessor or multi-core machine, or even on a uniprocessor machine. The operating system should divide processor time to threads in a fair and efficient way (executing multiple threads on a single processor by dividing processor time is sometimes called interleaving or time-slicing). This is usually done by a part of the operating system called the *scheduler*, which we will discuss later.

An operating system should take note of processes and threads, and offer four basic thread operations [17]:

- Spawn: When a process is spawned it usually also spawns a corresponding process. Threads should also spawn other threads within the same process.
- Block: Block a thread when it needs to wait for an event, allowing the processor to continue with another thread that is ready.
- Unblock: When this event for a blocked thread occurs, a thread should enter a ready state again.
- Finish: When a thread is finished it should release resources and terminate, allowing other threads to be processed.

Implementation Taking the above in account, a thread can be either in a runnable, running, blocked or finished state (more states are of course possible with more complex threading models). Do note that we also need to keep track of processor specific thread data: imagine that when a processor switches from one thread to another, the processor state like registers, program counter and stack pointer needs to be stored because execution needs to resume from the exact same state when the thread is resumed later on). It is also important to know why a blocked process is blocked: for example, is it waiting for a file to become accessible (I/O request)? Or is it waiting for a reply from another process (RPC request)? More reasons for a block are imaginable with more advanced operating systems.

Threading models Let us consider one widely used threading model, *pthread* (POSIX [7] Thread), which are commonly used on UNIX systems like Linux and FreeBSD [16]. We will in particular look at the FreeBSD implementation. Commonly under UNIX systems, a process is created with the system call *fork*, which creates a new process as an exact copy of the calling process [19].

Regarding threads, PThreads adds a lot of thread related routines to call to FreeBSD, for example (note there are many more): *pthread_create* (Spawn), *pthread_mutex_lock* (Block), *pthread_mutex_unlock* (Unblock) and *pthread_cancel* [20]. Note that Block and Unblock are implemented using concepts called *mutexes*, we will get to that in the next section.

3.3 Mutual Exclusion

Concurrency and multi-threading introduces several new problems of its own. A problem is that execution speed is unpredictable. In particular on uniprocessor machines, execution of a thread can stop anywhere. If we have a shared variable, and two threads that change it, we cannot predict the value it has after: this depends on which of the treads finishes writing as last. This problem is usually called a *race condition* [16].

A solution to this problem is referred to as *mutual exclusion*: controlled access to shared resources. The resource that is shared but needs protection is referred to as the *critical resource*, while the part of a program that uses the resource is called a *critical section*. To ensure mutual exclusion, the system must ensure that for each critical resource, only one thread may enter the critical section at a time. There are several ways of implementing this, including *semaphores*, *mutexes*, *monitors* and *spinlocks*, of which we will only explain the first one in a later section (note that mutexes are a special kind of semaphores). There are

[7] Portable Operating Systems based on UNIX

more ways of implementing mutual exclusion, as long as they meet the following requirements listed by Stallings (Stallings, 2009, Ch. 5.1) [16]:

- Mutual exclusion must be enforced.
- A process halting in its noncritical section must not interfere with other processes.
- It must not be possible for a process requiring access to a critical section to be delayed indefinitely (this prevents deadlocks and starvation).
- When no process is in a critical section, any process requesting entry to its critical section must be permitted to enter without delay.
- No assumptions are made about relative process speeds or number of processors.
- A process remains inside its critical section for a finite time only.

The downside to concurrency implementations is that, when used wrong, they can introduce several problems like *deadlocks* and *starvation*. It is important that the operating system prevents these problems.

- Deadlocks occur when two or more processes wait on each other [8].
- Starvation is observed when the scheduler keeps allocating the shared resource in an unfair way: one or more waiting threads is constantly forgotten for some reason, and is therefor starving.
- Busy Waiting is observed, for instance, with *spinlocks*: here we use an endless loop that does nothing to make a thread wait for its turn. However, this is a slightly optimistic description, because the loop condition has to be executed every iteration which makes the process execute whenever it can while waiting, which is usually considered unwanted.

3.3.1 Semaphores

The basic principle is that processes can cooperate by sending simple signals to each other, and we can build complex cooperation upon these basic signals. Semaphores are used for this signaling procedure (Stallings, 2009, Ch. 5.3) [16]. There are two primitive actions required: the possibility to send a signal through a semaphore (*semSignal*)and the possibility to wait until a signal is received through a semaphore (*semWait*). Semaphores are commonly implemented with a counting variable. *semWait* decreases this counter and blocks the calling process if it becomes negative, while *semSignal* increments the counter and unblocks a process that is blocked if this counter becomes zero or higher.

3.3.2 Discussion (CPU)

Operating systems provide a lot of basic usable primitives for managing programs that run with several processes and/or threads. The basic idea of processes is very useful: we could regard a program as a collection of one or more process(es) to implement multiple programs to run at the same time. The operating system itself needs to run next to user programs, so it's natural to provide it in this way. Threads are arguably unneeded: anything that can be done with processes can be done with threads and administration at user level (which requires processes to be able to communicate in a sense). For example, the Linux kernel does not provide distinction between processes and threads: user-level threads are mapped to kernel-level processes (Stallings, 2009, Ch. 4.6) [16], thus only processes are implemented at kernel level, while most other facilities are implemented at user level.

[8] Imagine a four-way crossing where four cars arrive at exactly the same moment. Each car has to wait until the car on his right passes, creating an endless cycle of waiting. This situation will probably never occur because the cars are driven by humans, but computers are prone to this kind of precise behaviour.

3.4 Memory

Memory management is an important and complex task for operating systems as well. Processor speed keeps growing, thus also the rate at which data can be processed and consequentially the amount of memory programs use. Like we stated before, memory is often scarce and slower than the processor. When the system runs out of main memory to use, it has to swap data to even slower I/O devices. It is thus a task for the operating system to efficiently manage what data to keep in memory and what data to swap. Another consideration is that operating systems strive to make things transparent for the user which adds extra complexity [9], usually solved by paging and segmentation. These techniques and their implementation are outside the scope of this work, and are excellent subjects for further additions.

[9] The way memory is allocated to and addressed by the user should always be the same. Imagine two processes, in which both ask for a block of memory. A naive implementation would give both processes two free blocks, and let the processes remember where it is. Now the first process asks for another block. If it was truly naive, the first process now has two different blocks with another block in the middle not belonging to him, and he has to do all the administration. To solve this, processes are given virtual address spaces able to grow and the operating system manages translation from virtual to real memory.

4 Scheduling

We discussed notions for threads and processes as units of execution, which can be executed at the same time (or nearly at the same time, given a uniprocessor). The amount of work a processor can do at the same time is still limited by its amount of cores, and thus the operating system needs to choose and manage what thread or process to execute at what time.

It needs to allocate the processor as a resource to the processes running in a fair and efficient way. This is sometimes also contradictory, thus decisions have to be made. We also have to take states and priorities in account: it would be a waste to give a blocked or really unimportant process the same amount of execution time as a ready or very important task (which would however be fair).

The concept of deciding which process to execute at what time is called *scheduling* since we make a schedule of processor time, and is done by a part of the operating system called the scheduler. Often but not always a distinction is made between scheduling and *dispatching*, of which the first focuses on long-term decisions, while the second is in charge of short-term decisions. In the following chapter we will discuss a uniprocessor scheduler.

4.1 Dispatching

It is useful to take a state model as basis for the dispatcher. We will describe the minimal state model for dispatching defined by J. Nehmer [18] because this is regarded as a good minimal model, often used as basis for expanded models.

Note that his definitions are for processes, but in our definition threads and processes only differ in memory and resource allocation and this model has no notion of either of those. Therefor, in this context it does not matter if one would use processes or threads.

He distinguishes four basic process states, nearly the same as the states named in earlier chapters:

- Suspended/Transient when a process is currently not being handled by the dispatcher it waits as being suspended. This state is used to allow higher order of planning than just the basic dispatcher (remember the planner). The intermediate state Transient can be offered to decouple this long-term planning and the dispatcher: the planner and dispatcher share this state.
- Blocked processes may be forced to enter or leave the blocked state by dispatcher primitives.
- Ready denotes a process that is ready to run.
- Running denotes a process that has a processor allocated to it. Processes remain in this state until they give up their processor by calling a dispatcher primitive.

He also defines three states for *pseudo-processors*. In his model, the concept of a process to an actual processor is abstract. It does not matter if multiple processes are executed by one physical processor via time-sharing or by multiple physical cores: a process is bound to an abstract pseudo-processor when it is being executed in some way.

The states he defines for these pseudo-processors are as follows:

- Idle a processor is idle when no work is available. This state should be implemented with a corresponding machine instruction. The processor can only be awakened from this state through external means, like an interrupt signal instruction.
- Allocated a processor in this state has processes allocated to it and is currently running.
- Free the processor is no longer allocated to a process but did not yet enter the idle state. A processor in this state can easily continue without having to be awakened externally again.

It is worth noting that he also defines state machine transitions and pseudo-code dispatcher primitive functions that these transitions correspond with. Another important point is the notion of preemptive operation in his model. In preemptive mode, whenever a process

is added, unblocked or deallocated, a process with the lowest priority lower than the process in question is also interrupted.

It is open for discussion if this preemptive mode is suitable for modern operating systems since processes run until completion or preemption by a more important process. It is a common use-case that more than one important process runs continuously in the background: to allow them to run in this environment one has to recreate the same process doing only a portion of work every time. To cope in a more usable way we will base our design on this model but choose a different method of preemption: round robin.

4.1.1 Round Robin

Round Robin is a fairly simple dispatching policy that is often used in one way or another. Processor utilization is divided in time slices, at the end of each slice interval an interrupt occurs. One keeps track of all runnable processes in a queue, and a process is chosen at *first-come-first-served* (FCFS) basis. After the clock interval interrupt, the currently running process is interrupted and placed at the back of the queue.

Deciding the length of this time slice *quantum* is essential: too long would result in processes having to wait a long time to get a turn (which might result in low responsiveness of the overall system), while there is also a slight overhead when switching from a process to another [10] so choosing a low quantum might make the system suffer in performance because it switches too often. The next section discusses a practical implementation combining the scheduler and dispatcher that uses time-slicing preemption as well but uses a variable quantum.

4.2 Schedulers

The scheduler makes more long-term decisions. Sometimes processes have to wait for external factors before they can be dispatched, like disk access or other I/O events. A system can also choose to suspend a process by itself for other reasons. When all these prerequisites have been cleared the scheduler has to pass the process on to the dispatcher. A straightforward implementation is placing processes in corresponding waiting queues, and moving it to correct queues when events occur. It is also common in modern systems that the scheduler and dispatcher are combined (and usually referred to as scheduler) in a single system. The next section gives a practical albeit complex example to illustrate the complexity of modern scheduling systems.

4.3 Example from FreeBSD: ULE

FreeBSD currently uses a scheduler called *ULE*, based on the 4.3BSD scheduler but expanded to take advantage of multiprocessor systems [21], and to reduce complexity from linear to constant (important for systems under high load). Do note that they do not make a distinction between the dispatcher and scheduler: the ULE scheduler also serves as a dispatcher. This scheduler consisted of several queues, two CPU-load balancing algorithms, an interactivity scorer, a CPU usage estimator, a slice calculator and a priority calculator [21].

Core structures The core structure per CPU, called the *kseq*, contains three arrays, one for real-time processes, one for interactive processes, and one for non-interactive. Fairness is implemented by keeping two more queues: current and next. Threads are allocated to either the current or next queue. Interactive, interrupt or real-time threads are allocated to the current queue, non-interactive to the next. This results in a very low latency response [21].

[10] The system has to perform a context-switch: saving the complete process state like registers, program counter and stack pointer, decide which process to run next and restore its process state.

Queueing Threads are chosen on priority from the current queue until it is empty. Afterwards the two queues are switched. This design guarantees that every thread is given a turn every two queue switches, regardless of priority [21]. A thread is assigned to a queue until it sleeps, or for the duration of a slice. The base priority, slice size and interactivity score are recalculated each time a slice expires. Threads can also have their priority raised to real-time or interrupt, which effectively guarantees that they will be placed in the current queue as well. This is to prevent important threads from running when an unimportant process holds a required resource [21].

Locks Every queue has an own lock, which is shared for every process in that queue. Locking a thread will lock all threads in the same queue. Giving every thread its own lock makes code less structured and is argued to perform worse, because average case more locks have to be performed. Since ULE 3.0 the cores also share load in two different ways:

- Push when one CPU queue is too full, it will try to push a thread to a CPU with a small queue.
- Pull when one CPU is idle (its queue is empty), it steals a thread from a non-idle CPU.

In these processes the following order will be preserved: physical cores acting as several *hyperthreaded* cores go first, then other cores on the same CPU, and finally other CPU's. This order is chosen to minimize cache misses and the cost of migrating this cache.

Interactivity score ULE determines an interactivity score for threads based on its voluntary sleep time and run time. Interactive processes typically have a high voluntary sleep time (waiting for user input). The formula used by the algorithm is as follows:

$$m = \frac{\text{maximum interactive score}}{2}$$

$$score = \begin{cases} \frac{m}{(\frac{sleep}{run})} & sleep > run \\ \frac{m}{(\frac{sleep}{run})} + m & sleep \leq run \end{cases}$$

This effectively divides the range in two sections: the lower half is assigned to threads who has a higher sleep time than run time, the higher to the others. There is also a limit at which scores are reduced to a fraction, to remember a part of a threads scoring history, but not allow it to grow unbounded. This is argued to be important to notice quickly that a thread switched from interactive to non-interactive (a common use-case since a lot of non-interactive processes are forked from interactive ones) [21].

A threshold on the interactivity score decides whether a thread is marked interactive or non-interactive. Together with the factor of history remembered this is the key factor determining responsiveness under heavy load. A too low threshold will result in false negatives, while a too high threshold would cause false positives.

Priority When the interactivity is calculated, the *niceness*, a relative priority factor, is added to the result to give the true priority. The kseq keeps track of every threads niceness, but also the lowest niceness score. Only threads with a niceness difference with the lowest of 20 will be given a slice inversely proportional to the difference, others are assigned a slice of 0 and placed in the run queue. When one of these threads are run their slice is re-evaluated and they are placed in the next queue. Nicer processes are thus given smaller slices.

Real-time and interrupt threads are allowed to run as long as they are not preempted by more important real-time or interrupt threads, while idle threads are allowed to run as long as there are no other runnable threads on the system,thus slice values are meaningless for them. Interactive threads are given the smallest slice value, to quickly discover when it is no longer interactive [21]. are meaningless for them.

4.4 Implementation notes

After discussing a modern example, we will now discuss a simple implementation of our uniprocessor Round Robin scheduler, and the problems to solve. In the previous chapters we distinguished several required key components that need implementation: the process control block, a preemption system, and a scheduler- and dispatcher system.

Process control block Not only do we need to store some key information about the processes (containing for example a unique process ID, its dispatcher state) but we also have to store the complete program state: registers, program counter, stack pointer and flag (status) register. A process also requires its own stack and heap, of which pointers could be stored in this block as well: we cannot have processes alter other processes' stacks or heaps. Storing these extra pointers and checking for misuse is recommended but optional. These data structures are easily translated to code. We give a C and NASM examples here.

```
/* Rough implementation example in C. */
typedef enum
{
    SUSPENDED,
    BLOCKED,
    READY,
    RUNNING
} pstate_e;
/* Example process control block for the AVR M32.
    This microprocessor has 32 8bit general purpose registers. */
typedef struct
{
    uint8_t pid;
    pstate_e pstate;
    uint8_t registers[32];
    uint16_t pc, sp;
    uint8_t sr;
    uint8_t * code;
    uint8_t code_size;
    uint8_t * stack;
    uint8_t stack_size;
    uint8_t * heap;
    uint8_t heap_size;
} pcontrol_t;
```

Parts of operating systems are often written in assembly because of the extra control and low-level capabilities [11] The NASM assembly dialect offers macro's to implement data structures.

```
; Implementation example in NASM, without storing the extra
; code, stack and heap pointers.
PSTATE_SPD: equ 0
PSTATE_BLK: equ 1
PSTATE_RDY: equ 2
PSTATE_RUN: equ 3

; Example process control block for the Intel x86-32 platform,
; without extensions.
struc   pcontrol_t
        .pid:           resb 1
        .pstate:        resb 1  ; PSTATE enumeration
        .regs:          resd 8  ; eax,ebx,ecx,edx,esp,ebp,esi,edi
        .ip:            resd 1
        .sregs:         resw 7  ; cs,ds,ss,es,fs,gs
        .flags:         resw 1
endstruc
```

The dispatcher requires to keep track of some information too. It has to know where to find the process control blocks (it cannot store or restore states without this information), how many processes are currently running (to prevent it from going out of bounds and grabbing arbitrary data as a process control block, a linked list structure could also solve this problem) and a counter that indicates the current running process (to know what process to store the state from, and what process is next). It is also useful to know the size of a process control block, to calculate where to store the state.

```
/*  Example dispatcher control block on the AVR M32. */
typedef struct
{
    uint8_t td_counter;
    uint8_t td_current;
    uint8_t * pcontrol_begin;
    uint8_t pcontrol_size;
} dcontrol_t;
/*  Example macro to get a pointer to the current
    process control block. */
#define current_pcontrol(x)  \
    ((x).pcontrol_begin      \
    + (x).td_current * (x).pcontrol_size)
```

Preemption system A processor is designed to execute a continuous given block of code, however we want to switch between different blocks of code (processes). We thus need a method to interrupt a running piece of code to jump to the code of the next process. One way of implementing this is using hardware *interrupts*.

Processors usually have a method of allowing running code to be interrupted by an external condition: in which usually the processor disables all other interrupts, pushes the current program counter to the stack (so that it can return here when the interrupt has been *handled*) and jumping to a set address for that given interrupt: the *interrupt handler* or *interrupt vector*.

[11] Not everything is possible in higher level languages: standard C does not offer control over interrupts or (control) registers. Some compilers offer extensions or the possibility to embed assembly, but interfacing between

Free stack space ... Free stack space	Top Bottom
Interrupted address	Pushed on stack by the interrupt event
Process stack ... Process stack	Top Bottom
Dispatcher address	Returned to if the process terminates correctly

Table 1: Example stack-layout after the interrupt has been fired. The process stack must remain intact, and the address points to where in the execution the interrupt fired, so we have to return there when resuming this process. If the process terminates correctly, it fires a return when its stack has been cleaned up. The dispatcher should have put an address there.

One could use a timer interrupt in the following way to implement Round Robin preemption by registering a function that does the following to the timer interrupt vector:

1. Disable interrupts;
2. Save the current program state;
3. Decide what process to execute next;
4. Set the correct timer interrupt;
5. Restore the program state of the next process;
6. Enable interrupts;
7. Begin executing the code of the next process.

The process of switching from one running process to another is often called *context switching*. It is important to enable the interrupts at the last moment before resuming, and disabling them immediately after the interrupt fires so that the scheduler can never be interrupted. If the scheduler would be interrupted at any point, the state of a process might get lost.

Also, the address of another function must be known as return address of the current running code [12] so that it will be called when a process terminates inside its quota:

1. Disable interrupts;
2. Remove current process from ready queue;
3. Decide what process to execute next;
4. Set the correct timer interrupt;
5. Restore the program state of the next process;
6. Enable interrupts;
7. Begin executing the code of the next process.

C and assembly can sometimes introduce more problems than it is worth.

[12] In binary programs this usually means this return address must be at the bottom of the stack since the *return* instruction often takes an address from the stack and jumps there (and assuming the stack was kept intact, the stack pointer points to the bottom of the stack at termination time), but this does not need to be the case: in FreeBSD executables call the exit system call with their return value when they terminate, which on the *x86* platform corresponds to calling the software interrupt (0x80) with the value 1 in the *eax* register and the return value on top of the stack [22].

Saving the program state is not as trivial as it would sound: almost all instructions alter the program state in a way. This means that we have to store the unaltered program state using operations that alter the program state. This process is heavily platform dependent (since it matters what operations are available and how they alter the state), so we will describe a global method.

One large assumption that we make in the following paragraphs is that once a stack variable is 'popped', it is regarded as removed from the stack. Usually, popping a variable from the stack does not actually delete it: it is stored in the target register and the stack pointer is altered in a way that it now points to the value below. This is normally followed by another push operation, which overwrites the previously popped value. Therefor we regard this assumption as realistic. With this assumption, the scheduler is free to increase the running process' stack, overwriting old values that are popped anyway, as long as it restores the stack pointer to the original value afterwards and it never goes below this stack pointer. Only in the case where the programmer abuses the stack in a way that it reuses values previously popped, this assumption can and will cause problems.

The problems we encounter here:

- We have to store the registers (including the flag or state register), program counter and stack pointer. The program counter is typically pushed on top of the stack when an interrupt occurs: we have to remove it before we can store the stack pointer.
- Using the stack in any way may not alter the running program's stack. It is typically safe to increase the stack a bit, as long as we keep it intact.
- Almost all operations require that we alter one or more registers as well, so they have to be stored early.
- Arithmetic operations (and other operations, depending on the platform) alter the status register. We have to store this register before performing one of these instructions.

Because it is platform dependent which registers can be used for what purpose, it is impossible to give a concrete solution. A global solution can be defined assuming there is a pointer register (in the case of a more general purpose instruction set, one can simply reserve a general purpose register) and enough general purpose registers to serve as temporary registers.

Assuming the dispatcher's main thread immediately calls this function with the state of the last process still intact,

- Disable interrupts: an interrupt will cause the state to be lost.
- Push as many registers as we will need as temporaries to the stack.
- Push the status register to the stack so we can perform arithmetic operations safely. One can choose to push all registers, making the storing process an easy loop in certain cases, or only a small set to minimize stack usage.
- Calculate the current process control block pointer using the temporary registers, this step defines the amount of temporaries required.
- Store all other registers using the pointer, now these can be used as temporaries.
- Pop the status register to a new temporary and store it using the pointer.
- Pop the old temporary registers from the stack and store them using the pointer, this should leave the pointer in a register, all registers safely stored and the stack back to its original value.
- Pop the return address of the running process and store it using the pointer.
- Store the stack pointer using the pointer.

Free stack space ... Free stack space	Top Bottom
Status register	Stored to allow arithmetics in the dispatcher
Temporary registers ... Temporary registers	First to be stored Last to be stored
Interrupted address	Pushed on stack by the interrupt event
Process stack ... Process stack	Top Bottom
Dispatcher address	Returned to if the process terminates correctly

Table 2: Example stack-layout during the process of storing the state. The status register has been stored on top of the list of temporary registers we have to store.

Restoring the program state of the next process is the next step in context switching. The problem we encounter here is similar, but reverse: we need to restore registers (restricting them for usage), we can only perform arithmetic operations before we restore the status register, we can only manipulate the stack in a growing fashion.

The sequence in which we need to perform our operations again depends on what operations manipulate the status register, and depending on this one might prefer to first push the complete state onto the stack before restoring it, making the restoring only depending on the stack. Because this procedure is called either after storing the previous process, or a terminated process, the current program state including the stack is freely manipulatable as we stated before.

One last consideration is how to restore the program counter: often architectures allow jumping to constant addresses or to the address contained by a (special) register. However the first is not optimal because we then have to modify our running code which is often not allowed by architectures (that protect code segments from writing). The second is also not suited: we have to restore the complete state before we jump, including this register itself. A better solution is by abusing the return instruction. This grabs a value from the stack and jumps there. Remember our assumption that we can safely increase the stack as long as we make sure it is intact again when we actually enter the next process. If we ensure the stack is intact and then push the address of the next process to the stack, a simple return will do the job.

- Increase the current process counter (rolling back to zero if it was the last process).
- Calculate the current process control block pointer.
- Restore the status register.
- Restore the stack pointer.
- Push the program counter to the stack.
- Restore all registers.
- Enable interrupts again.
- Return, causing the program counter to be removed from the stack and performing the actual jump.

Free stack space ... Free stack space	Top Bottom
Process address	Pushed on stack by the dispatcher
Process stack ... Process stack	Top Bottom
Dispatcher address	Returned to if the process terminates correctly

Table 3: Example stack-layout during the process of restoring the state. The free stack space is usable, as long as at the time of jumping to the process code the stack points to the process address.

5 Conclusion

Even building a very simple operating system can be a overwhelming and complex task, because operating systems are very complex systems. By systematically defining the borders can make it a more doable but still very challenging task. After that it becomes a technical task because it requires extensive knowledge of the architecture one works on. An operating system should be convenient, efficient and able to evolve.

5.1 Core concepts

Exactly the core concepts an operating system offers differs from operating system to operating system. Core services to consider are program development and execution, I/O devices-, controlled file and system access, error detection and response and accounting. Of these, we regard program execution, basic access to I/O devices, simple file access (not necessarily controlled) and system access the most important: without these an operating system cannot operate. Program development services, advanced error detection, controlled file access and accounting are regarded as optional. They make the operating system better suited for day-to-day usage but are not really required.

The components of the operating system require a way to communicate with each other. Also, running programs can request services of the operating system. To account for this, a manner of communication with the operating system is required. A suitable architecture is to provide an API through system calls.

Program execution brings in complexities regarding linking different sources of code together to minimise code duplication. Access to I/O devices and file systems require a layer of abstraction in the form of a driver system. Lastly, system access requires resource management in the form of memory allocation, protection and ideally also caching but also processor scheduling and dispatching.

System access A uniform way of offering access to the hardware of the system to running programs is important for an operating system. Programs should be able to dynamically request memory and obtain processor time in a fair manner. For reasons like abstraction and to be able to run multiple programs simultaneously, it is also important to offer the concept of processes as units of execution. Processes should also be able to communicate, and methods to fairly share resources between them are important.

Processor scheduling and dispatching Both the scheduler and dispatcher should be fair and efficient. Good state machines for dispatcher primitives exist in literature, providing a good basis for implementations. A fairly simple, efficient and fair dispatching policy is round robin, which is implementable on architectures which provide preemption or interruption methods. More advanced examples exist in practice which use queues, time slicing

and variable priorities (often based on interactivity) and offer better performance. Implementation difficulties arise depending on the target platform. It is a good idea to keep track of how the stack is used. A general solution is difficult to give, but important considerations can be formulated.

5.2 Problems to solve

Several technical issues surfaced when attempting to create an actual working implementation based on the dispatcher design. It is difficult to program all components in a higher level programming language. For instance, C abstracts away from concepts like the stack, the program counter, interrupts and control registers. Programming everything in pure assembly makes this already complex task even more challenging.

Interfacing between C and assembly seems like a good solution, however this creates problems of its own. It is sometimes hard to predict how the C compiler translates its input, which can have problematic consequences. Because of the fact that operating systems run on bare hardware with almost no help, a lot of wheels had to be re-invented (imagine things like booting up the system and printing text to a screen), which consumed precious time.

On both the AVR and Intel platforms, there where problems regarding the registration of interrupt vectors. The C compiler generated a lot of unnecessary prologue and epilogue code and made it very hard to acquire the program counter, while it seemed an extremely complex task in assembly. On the Intel platform, progress was frozen because the BIOS only loads the *MBR* [13] of its boot device automatically, which we almost exceeded just with our boot- and printing code. Wanting to execute more requires manually loading this from the disk to the memory, which also introduced too much complexities.

5.3 Pointers for future work

While we give a broad overview of the complexity of some of the core aspects, this paper is far from complete. Further work should be done in all of the subjects we described. We only described the basic preemption system implementation, but did not actually offer one.

More work should be done describing the difficulties and workings of the other important core services, and if possible the remaining core services as well. Taking this all together should describe a meaningful document on designing operating systems and provide a basis for an actual complete implementation to use next to the written material as teaching tool.

[13] Master Boot Record, this denotes the first 512 bytes of a MBR formatted disk, executed when a disk is booted. Because of the small size of this block, this MBR usually contains code to chainload the real bootloader.

A Executable formats

This appendix will give a very brief overview in the form of a case study of a core concept of operating systems that could seem simple but is actually rather complex: executable loading.

Programs have become increasingly complex, but there was a time when storage was so limited that even binaries were considered big. To work around this and other problems people designed executable formats, providing features like dynamic runtime linking and relocation. Consider a piece of code that is reused several times (usually regarded as a library). To avoid having to include this piece of code in every binary that uses it (also consider that it could be part of a bigger library with code that is not used), they designed techniques to link binaries at runtime. For simplicity, we will only consider one format.

A.1 EXE

The old MS-DOS EXE file format contains a fixed header and zero or more executable sections, existing in blocks and segments of fixed size. Each block is 512 bytes, and each paragraph is 16 bytes. Every multi-byte value is stored least significant byte first. [23]

A.1.1 EXE-Header

Offset	Value
`0x00 - 0x01`	Magic number (*0x4D 0x5A*).
`0x02 - 0x03`	The byte-size of the last block, or zero if 512.
`0x04 - 0x05`	The number of blocks in the file.
`0x06 - 0x07`	Number of relocations stored in the header (may be 0).
`0x08 - 0x09`	Number of paragraphs stored in the header.
`0x0A - 0x0B`	Number of paragraphs minimally required in memory for the program to run.
`0x0C - 0x0D`	Maximum number of paragraphs the program uses in memory.
`0x0E - 0x0F`	Relative value of the stack segment.
`0x10 - 0x11`	Initial value of the *sp* register.
`0x12 - 0x13`	Word checksum value: the 16-bit sum of the complete header should be 0, but this checksum is not always used.
`0x14 - 0x15`	Initial value of the *ip* register.
`0x16 - 0x17`	Initial value of the *cs* register, relative to the segment the program was loaded at.
`0x18 - 0x19`	Offset of the first relocation item in the file.
`0x1A - 0x1B`	Overlay number.

Example For example, consider a simple *Hello World* program written in C, compiled using *mingw-gcc* 4.5.3 under Windows 7. This is actually a PE executable, but the PE header requires a MS-DOS executable header for backwards compatibility, with the actual PE offset at 0x3C [24]. The header is from *0x00* to *0x1B*, thus *0x1C*, or 28 bytes long. We generate a hex-dump of the header on a Unix environment as follows:

```
hexdump -n 28 hello.exe
```

A hex-dump of the file header shows the following data:

```
0000000 | 4d 5a | 90 00 | 03 00 | 00 00 | 04 00 | 00 00 | ff ff | 00 00
0000010 | b8 00 | 00 00 | 00 00 | 00 00 | 40 00 | 00 00
```

Remember that multi-byte characters are stored least significant byte first.

`0x4D 0x5A`	The magic number.
`0x90 0x00`	The last block is *0x90* bytes long.
`0x03 0x00`	The file contains *0x03* blocks.
`0x00 0x00`	The header contains zero relocations.
`0x04 0x00`	The header contains *0x04* paragraphs.
`0x00 0x00`	The program requires no extra paragraphs minimally.
`0xFF 0xFF`	Apparently the program maximally uses *0xFFFF* additional paragraphs.
`0x00 0x00`	Relative stack segment is zero.
`0xB8 0x00`	*sp* should initially be set to *0xB8*.
`0x00 0x00`	Checksum is unused in this binary.
`0x00 0x00`	*ip* is initialised at zero.
`0x00 0x00`	*cs* is initialised at the start segment of the program.
`0x40 0x00`	The offset of the first relocation is *0x40*.
`0x00 0x00`	Overlay number is unused.

The beginning executable data is calculated with the number of paragraphs in the header: each paragraph is 16 bytes long, thus we multiply this by 16. $0x04 * 0x10 = 0x40$. The end of the EXE data is computed using the amount of blocks in the file and the size of the last block. Every but the last block is 512 bytes, thus $0x02 * 0x200 + 0x90 = 0x690$.

Inspecting the executable data from *0x40* (64) which is also *0x40* long:

```
hexdump -s 64 -n 64 hello.exe
```

This shows the following data:

```
0000040 | 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68
0000050 | 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f
0000060 | 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20
0000070 | 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00
```

From *0x4E* the data is an ascii string: *This program cannot be run in DOS mode.* The code segment starts at *0x40*, thus the string is located at offset *0x0E*.

Disassembling the code gives the following code [14]:

```
0x0E    PUSH CS
0x1F    POP DS
0xBA    MOV WORD DX, 0x000E
0x0E    –
0x00    –
0xB4    MOV BYTE AH, 0x09
0x09    –
0xCD    INT 0x21
0x21    –
```

This code first loads the *cs* to *ds*, setting *ds* at the current code segment. Calling interrupt *0x21* in DOS performs a different action depending on the value in ah, with value *0x09* it prints a string pointed by *[ds:dx]* [15]. Since *ds* is set on the current code segment , and *dx* is set to *0x0E*, the string pointed is located at code segment with offset *0x0E*, which is exactly where the ascii string is located.

```
0xB8    MOV WORD AX, 0x4C01
0x01    –
0x4C    –
0xCD    INT 0x21
0x21    –
```

This moves *0x01* to *al* and *0x4C* to *ah*, and calls interrupt *0x21*. With *0x4C* in *ah*, this interrupt exits the current program with the value in *al* as return value, in effect terminating with exit code *0x01* [16].

[14] http://ref.x86asm.net/coder32.html
[15] http://www.ctyme.com/intr/rb-2562.htm
[16] http://www.ctyme.com/intr/rb-2974.htm

B Bootloaders

The next appendix gives another case study in a rather complex side of operating systems that can get rather technical and hardware specific: bootloaders. The basic idea behind them is that booting up a system is very hardware specific. A bootloader is designed to offer a layer of abstraction for this: a bootloader is designed for a specific platform and will leave the system in a consistent, well defined state before it attempts to run the operating system. This way the operating system (if compatible with this specific bootloader) knows what state it can expect when it starts up.

In this case study we will regard the Intel x86-64 platform. Here we will give a technical case study of some of the core aspects: programming machines at the bare minimum, setting them up for execution and in this specific case, communicating with the BIOS[17], which is a basic system available on IBM compatible computers to offer rudimentary I/O services.

B.1 Fist steps

The very first attempt was (in Intel ASM syntax for NASM) to actually assemble and run code on a (virtual) machine. We will go into more depth later.

```
        BITS 16
main:
        mov ax, 0x07C0
        add ax, 288
        mov ss, ax
        mov sp, 4096
        mov ax, 0x07C0
        mov ds, ax

        call hello_world
main_loop:
        hlt
        jmp main_loop

hello_string db "Hello World!", 0
hello_world:
        mov bx, hello_string
        mov ah, 0x0E
_hello_world_loop:
        mov al, [bx]
        cmp al, 0x00
        jz _hello_world_end
        int 0x10
        inc bx
        jmp _hello_world_loop
_hello_world_end:
        ret

        times 510-($-$$) db 0
        dw 0xAA55
```

Time to describe this big piece of assembly code in more depth.

```
        BITS 16
        main:
        mov ax, 0x07C0
        add ax, 288
```

[17] Basic In- and Output System

```
        mov ss, ax
        mov sp, 4096
        mov ax, 0x07C0
        mov ds, ax
        call hello_world
```

Boot code must be in 16 bits. This is how we tell this to the compiler. After this we reserve some space on the stack, which is not very important now. The BIOS automatically loads the MBR code in the RAM at address *0x7C0*, which explains that number. The last line makes us call the *hello_world* procedure (we'll get there later).

```
main_loop:
        hlt
        jmp main_loop
```

This is simply an easy way to make the code loop without end. We *halt* the CPU until the next interrupt, and loop back again after.

```
hello_string db "Hello World!", 0
hello_world:
        mov bx, hello_string
        mov ah, 0x0E
_hello_world_loop:
        mov al, [bx]
        cmp al, 0x00
        jz _hello_world_end
        int 0x10
        inc bx
        jmp _hello_world_loop
_hello_world_end:
        ret
```

Before we can explain this piece of code, we first note that calling the CPU-interrupt *0x10* with the value *0x0E* in the *ah* register will call a *BIOS* function to print an ascii-character of the value in the register *al* to the screen and advance the cursor.

We first define a constant null-terminated text string in the binary to print, and assign a label to it. The procedure *hello_world* begins with storing the address of the string in the *bx* register. Then we move the constant *0x0E* in the *ah* register (remember the interrupt). Then we enter a loop, which starts by moving the byte where *bx* points to into *al* (the character to print). We then compare it with zero and jump out of the loop (and simply return out of the call) if they are equal (to terminate printing at the null-byte). If this is not the case, we call the interrupt *0x10*, increase *bx* by one and loop. This effectively prints the string pointed to by *bx* to the screen.

```
times 510-($-$$) db 0
dw 0xAA55
```

When an x86 machine boots up from a disk drive, it checks if this disk has a master boot record or *MBR*. This is the first 512 bytes of the disk, and contains binary code that is executed to boot the system. However, the last two bytes of the MBR must be equal to *0xAA55* before the BIOS accepts this as a valid MBR. This piece of code fills up the binary with zero's until its size is 510 and then inserts *0xAA55*.

How to actually assemble and execute this piece of code to test? We assembled this on a *GNU/Linux* system using the *NASM* assembler, and ran it in *qemu* as follows:

```
nasm -f bin -o hello_world.bin hello_world.asm
qemu -fda hello_world.bin
```

We tell nasm to build a pure binary and tell qemu to run it as a boot floppy.

B.2 Protected Mode

The Intel x86-64 CPU boots up in 16 bits mode for backwards compatibility, emulating an older processor, called *Real Mode*. To use the modern 32- or 64 bit features of this CPU, it must be manually set to *Protected Mode*. Our next bootloader will illustrate how to do this. We will use a slightly exotic example to show a way of separating different stages of the bootloader: we use two stages, stage1.asm and stage2.asm, and build both to different binary files (stage1.bin, stage2.bin), and directly copy stage2.bin into stage1.bin at a known offset. To automate this we use the following Makefile:

```
all:    stage1 stage2
        dd if=stage1.bin of=bootloader.bin
        dd conv=notrunc bs=1 seek=256 if=stage2.bin of=bootloader.bin

stage1:
        nasm -f bin -o stage1.bin stage1.asm
stage2:
        nasm -f bin -o stage2.bin stage2.asm
```

Here we show the first stage, stage1.asm:

```
[ORG 0x7C00]

start:
        xor ax, ax
        mov ds, ax
        mov ss, ax
        mov sp, 0x9C00

        mov si, msg_bprot
        call _sprint16

        cli
        push ds

        lgdt [gdtinfo]

        mov eax, cr0
        or al,1
        mov cr0, eax

        call 0x7D00

        and al,0xFE
        mov  cr0, eax

        pop ds
        sti

        mov si, msg_shutd
        call _sprint16

hang:
        hlt
        jmp hang

msg_bprot       db '[Boot:1] Entering protected mode...', 13, 10, 0
msg_shutd       db '[Boot:1] Shutdown!', 13, 10, 0

_sprint16:
        lodsb
        cmp al, 0
        jz _sprint16_end
        mov ah, 0x0E
        int 0x10
        jmp _sprint16
_sprint16_end:
```

```
        ret

gdtinfo:
        dw gdt_end - gdt - 1
        dd gdt
gdt                         dd 0,0
flatdesc    db 0xFF, 0xFF, 0, 0, 0, 10010010b, 11001111b, 0
gdt_end:

times 510-($-$$) db 0
db 0x55, 0xAA
```

Time to describe this code in more depth.

```
[ORG 0x7C00]

start:
        xor ax, ax
        mov ds, ax
        mov ss, ax
        mov sp, 0x9C00

        mov si, msg_bprot
        call _sprint16
```

The first line tells the assembler to add *0x7C00* to every offset, because this is the address at which this code will be loaded. We set *ds* and *ss* to 0, and set the stack pointer to be *0x2000* higher than the load address (reserving some stack space). We then load the address of a string to the *si* register and call a 16bit string printing procedure which uses the BIOS (slightly modified from the hello world bootloader described earlier with the difference that we use the 16bits string index register and the *lodsb* instruction which loads a byte from *[ds:is]* to *al*).

```
        cli
        push ds

        lgdt [gdtinfo]

        mov eax, cr0
        or al,1
        mov cr0, eax

        call 0x7D00
```

Here we actually perform the needed steps to enter protected mode (Intel, Ch. 9.9.1) [26]. We first disable interrupts since we do not want to be interrupted during this procedure. We push the old ds register because we will also demonstrate how to return to real mode later. After that we use a special instruction that loads a *Global Descriptor Table*, which is the core of protected mode. This mode is named protected since it adds data protection mechanisms: we can set memory segments as data or code and adding read and write rights (Intel, Ch. 3.4.5.1) [26]. To set this we use the global descriptor table. The *lgdt* instruction expects a specific format: first the last byte of the table as a 16 bit word, and then the start of the table as 32 bit value (Intel, Ch. 3.5.1) [26] and it then reads and stores it, giving each segment an 8 byte descriptor: segment limit field, base address fields, type field, S, DPL and P flags (we won't go in more depth here) (Intel, Ch. 3.4.5) [26]. After that, we have to set the last bit of the *cr0* register: also called the *PE* register, this enables protected mode (Intel, Ch. 2.5) [26]. After setting *PE* we must perform a jump or a call (Intel, Ch. 9.9.1) [26], thus we call the known address of the second stage (recall that we load stage1 at address *0x7C00*, and we copy stage2 at offset 256, which is *0x100* in hexadecimal notation).

```
        and al,0xFE
        mov  cr0, eax

        pop ds
        sti

        mov si, msg_shutd
        call _sprint16
```

Now we disable the *PE* bit to return to real mode, pop the old *ds* and enable interrupts again, print another message through the now again available BIOS, and halt.

Time to describe stage2, which is rather short and only contains a method to print a string in 32 bit mode without using the BIOS.

```
[ORG 0x7D00]

start:
        mov  bx, 0x08
        mov  ds, bx

        mov ebx, 0xB8000
        mov ecx, msg_bprot
        mov ah, 0x02
        loop:
                mov al, [ecx]
                inc ecx
                cmp al, 0
                jz end
                mov word [ebx], ax
                inc ebx
                inc ebx
                jmp loop
end:    ret
msg_bprot       db '[Boot:2] Entered protected mode!', 0
```

Note the new offset value. First we set the ds register to 0x08, which is the first descriptor. Then we copy the message to *0xB800*, which is the start of the video memory. Do note that every letter has to be followed by a byte describing the colour from the palette, this is why we copy words instead of bytes (the letter is stored in *al*, the colour in *ah*, thus we choose the second value from the palette).

References

[1] Stallings, W. (2009). *Operating Systems: Internals and Design Principles.* Upper Saddle River (USA): PEARSON, Prentice-Hall.

[2] Russinovich, M. E., Solomon, D.A. (2009). *Windows® Internals.* O'Reilly Media, Inc.

[3] Delorie, DJ. (2010). *EXE Format.* Retrieved from http://www.delorie.com/djgpp/doc/exe/

[4] *Microsoft Portable Executable and Common Object File Format Specification* (2010). Retrieved from http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx

[5] *syscals(2) - Linux manual page.* (2010). Retrieved from http://www.kernel.org/doc/man-pages/online/pages/man2/syscalls.2.html

[6] M. K. Johnson, S. Scalsky. (1996). The Linux Documentation Project: *How System Calls Work on Linux/i86.* Retrieved from http://tldp.org/LDP/khg/HyperNews/get/syscall/syscall86.html

[7] D. Ritchie (Bell Labs) (1917). *a.out - Assembler and Link Editor Output* Retrieved from http://cm.bell-labs.com/cm/cs/who/dmr/man51.pdf

[8] The Santa Cruz Operation, Inc & AT&T. (1997). *System V Application Binary Interface Edition 4.1* Retrieved from http://www.sco.com/developers/devspecs/gabi41.pdf

[9] M. K. McKusick, K. Bostic, M. J. Karels, J. S. Quarterman. (1996). The Design and Implementation of the 4.4BSD Operating System (chapter 2). Addison-Wesley Longman, Inc. Retrieved from http://www.freebsd.org/doc/en_US.ISO8859-1/books/design-44bsd/book.html#OVERVIEW

[10] *execve(2) - FreeBSD 8.2 System Calls Manual.* (2010). Retrieved from http://www.freebsd.org/cgi/man.cgi?query=execve&sektion=2

[11] B. N. Handy, R. Murphey, J. Mock. *FreeBSD Handbook: Linux Binary Compatibility* Retrieved from http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/linuxemu.html

[12] T. Rhodes. *FreeBSD Handbook: File Systems Support* Retrieved from http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/filesystems.html

[13] *Mac OS X Developer Library: Kernel Programming Guide.* (2012). Retrieved from https://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/KernelProgramming/Architecture/Architecture.html

[14] *Mac OS X Developer Library: Mac OS X ABI Mach-O File Format Reference.* (2009). Retrieved from https://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html

[15] Tanenbaum, A. (2006). *Structured Computer Organization.* Upper Saddle River (USA): PEARSON, Prentice-Hall.

[16] Stallings, W. (2009). *Operating Systems: Internals and Design Principles.* Upper Saddle River (USA): PEARSON, Prentice-Hall.

[17] Anderson, T.; Bershad, B.; Lazowsska, E.; and Levy, H. *Thread Management for Shared-Memory Multiprocessors.* In Tucker, A. ed. *The Computer Science Handbook.* Boca Raton, FL: CRC Press, 2004.

[18] Nehmer, J. (1975). *Dispatcher primitives for the construction of operating system kernels. Acta Informatica, 5(4),* 237-255. doi:10.1007/BF00264560

[19] *fork(2) - FreeBSD 8.2 System Calls Manual.* (1993). Retrieved from http://www.freebsd.org/cgi/man.cgi?query=fork&sektion=2&manpath=FreeBSD+9.0-RELEASE

[20] *pthread(3) - FreeBSD 8.2 Library Functions Manual.* (2010). Retrieved from http://www.freebsd.org/cgi/man.cgi?query=pthread&sektion=3&manpath=FreeBSD+9.0-RELEASE

[21] J. Roberson (The FreeBSD Project). (2003). *ULE: A Modern Scheduler For FreeBSD.* Retrieved from http://www.usenix.org/event/bsdcon03/tech/full_papers/roberson/roberson.pdf

[22] G. A. Stanislav. (2001). *FreeBSD Developers' Handbook - Chapter 11 x86 Assembly Language Programming.* Retrieved from http://www.freebsd.org/doc/en/books/developers-handbook/x86-system-calls.html

[23] Delorie, DJ. (2010). *EXE Format.* Retrieved from http://www.delorie.com/djgpp/doc/exe/

[24] *Microsoft Portable Executable and Common Object File Format Specification* (2010). Retrieved from http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx

[25] http://ref.x86asm.net/coder32.html

[26] Intel®64 and IA-32 Architectures Software Developers Manual, Volume 3A: System Programming Guide, Part 1 Retrieved from http://www.intel.com/design/processor/manuals/253668.pdf