- MinOS -
A minimalistic operating system

Stefan Hinterkoerner, Florian Landolt

Term 2007/2008

Contents

1 Introduction **3**
1.1 Abstract 3
1.2 Development-Environment 3
1.3 Pair Programming 4

2 Minos Features **4**
2.1 Overview 4
2.2 Executables 4
2.3 Processes 4
2.4 Memory Management 5
2.5 Scheduling 6
2.6 Semaphores 8
2.7 File System 9
2.7.1 Virtual Filesystem Layer 9
2.7.2 Buffer Cache 10
2.7.3 GOSFS 10
2.8 System Calls 12

3 Programs **13**
3.1 User Programs 13
3.2 Unit Tests 13

4 Conclusion **14**

1 Introduction

1.1 Abstract

This project emerged from the operating systems course held by Prof. Dr. Christoph Kirsch at the University of Salzburg's department of computer science. The goal of this course is to build an operating system in order to understand and appreciate principled engineering of operating systems. The operating system therefore should at least support some form of concurrency support, memory management, device abstraction, and file handling.

MinOS is a student project which aims to create a minimalist operating system on the bare metal. Not only does MinOS stand for ”minimalist operating system”, it also refers to Greek mythology. Minos was a mythical king of Crete, son of Zeus and Europa.
Taking into account that we consider a minimalist approach to creating an OS the most elegant, if not superior way, the name of a mythical king seems appropriate for our project.

MinOS is based on the GeekOS operating system kernel. We opted for using a basic framework instead of really writing it from scratch for two main reasons. Firstly we did not really know how to start such a project and secondly, this project was limited to a single term.

Although the GeekOS projects partially force one to do things in a certain way, we tried to use GeekOS as a vantage point while trying to find alternative approaches to improve certain aspects of the system.

1.2 Development-Environment

As programming environment we chose Eclipse 3.3 with CDT and Subclipse Plugin. For compiling, we have used nasm 0.98.38 and gcc-3.4. Higher versions of gcc proved difficult to use. As mentioned above we have used GeekOS 0.3.0 as our vantage point.

Short overview of the machines we utilized for building our operating system:

Type	CPU(s)	RAM	OS
IBM x335 Sever	2 x Intel Xeon 2.8 GHz, 512 KB Cache	1 GB	Ubuntu Server 7.10, 32 Bit
PC	AMD Athlon64 3400+, 512 KB Cache	1 GB DDR-RAM	Ubuntu 7.10, 32 Bit
PC	AMD Athlon64 3500+, 1024 KB Cache	2 GB DDR-RAM	Ubuntu 7.10, 32 Bit
Apple MacBook	Intel Core Duo 2 x 2.0 GHz, 1st Gen.	1 GB DDR2-RAM	Mac OS X 10.5.1
Apple MacBook	Intel Core 2 Duo 2 x 2.0 GHz, 2nd Gen.	2 GB DDR2-RAM	Mac OS X 10.5.1

The server was used as SVN- and Build-Server, whereas the PCs served as our workstations.

1.3 Pair Programming

Pair Programming is a software development technique where two programmers work on a single machine. While one types in the code, the other one revises it. Thoese two roles are switched frequently. This technique has the advantage that one can concentrate on writing the code while the other is able to contribute ideas or improvements and face possible problems which may arise.

We employed this technique, because we did not want to split up the whole project. This way each of us was forced to deal with every single aspect of the project. Another advantage of this approach is that there was no danger of specialization.

2 Minos Features

2.1 Overview

In this chapter the various features of MinOS will be discussed in detail. The following key-features were implemented:

- **ELF** is used to represent and load executables.
- The memory is managed by **segmentation**.
- **Semaphores** are used for process synchronization.
- MinOS supports two different preemptive **scheduling** policies.
- A **Filesystem** (GOSFS) is used to provide a basic form of file handling.

2.2 Executables

MinOS is able to load and execute user programs. For this purpose the ELF (Executable and Linking Format) standard is employed, which is a format for storing programs on a disk. We opted for using this format not only because GeekOS demanded it, but because it is a common standard file format for relocatable, executable and shared object files. ELF is also the standard binary file format for UNIX-based operating systems. For more detailed information on ELF please check the ELF Specification [3]

In order to load executable files from the disk into the memory, we implemented a parser to map the ELF structures to those MinOS uses to represent executables (see elf.c).

2.3 Processes

A process is a program in execution. It comprises the program code and the current state information (such as the values of the program counter, registers, variables, ...).

Like other operating systems MinOS makes a distinction between the tasks the operating system code is allowed to execute and the tasks user programs are permitted to fulfill. This is necessary in order to protect the system from incorrect or even malicious code.

One of the things that have to be done to achieve this, is to distinguish between (at least) two different types of processes:

- Kernel Processes
  These so called privileged processes have access to the whole system functionality and memory.
- User Processes
  These processes are only allowed to access their own memory and their set of system operations is limited too.

Description of the implementation of this concept in MinOS:
In MinOS the differentiation between the two types of processes was achieved in the following manner:
Both kernel and user processes have the same structure (struct Kernel_Thread, see kthread.h). The structure includes a field ”userContext” of the type struct User_Context. If this field is NULL, the process is a kernel process, if it contains a user context, it is a user process.
Note that user threads have two different stacks! The kernel stack is where the operating system stores the execution context, next to being a regular stack while executing within the kernel. The user stack serves as the program's stack.

2.4 Memory Management

Just like the CPU the memory is an important resource which should be carefully managed by the operating system.Virtual memory is a usefull abstraction. The basic idea behind it is that the size of a program may exceed the size of memory. In order to allow programs to run, even when they are only partially in main memory, the concept of virtual memory seperates the memory addresses used by a program from the actual physical addresses.

Segmentation is one way to implement virtual memory. It provides the system with many address spaces (segments) which are completely independent from each other. A segment specifies a region of memory and the privilege that is required to access it. The address of a segment consits of two parts, a segment number, and an address within the segment (offset). The internal memory starts at 0 and grows to a maximum size. Various segments usually differ in size, and additionally, the size of a segment may change during execution.

Each user program has its own memory segments for code, data, stack and extras. A user program can only access its own segments, which is why a process' attempt to access memory outside its segments, will result in a segementation fault.

Segmentation is supported by Intels IA-32 architecture.
Information describing a segment is stored in a data structure called segment descriptor, which are stored in so-called descriptor tables.

There are two different types of these tables:

- **Local Descriptor Table (LDT)**
  This table stores all the segment descriptors belonging to a user process. There is one LDT per process.

- **Global Descriptor Table (GDT)**
  A GDT stores and manages all the LDTs. There is only one GDT in the whole system.

The two CPU-Registers GDTR and LDTR contain the addresses of the GDT and the current LDT.
In addition, six registers are used to keep track of the process's active segments: CS (Code Segment), SS (Stack Segment), DS (Data Segment) and ES, FS, GS (Extra Data Segments).

Although nowadays segmentation is not the common choice, we decided to employ this technique for various reasons:
First of all, this technique is fairly straightforward and a good way of making oneself familiar with the topic of memory management. However, there are of course other advantages, for example the simplification of linking and, in turn, compiling (symbol table, parse tree, call stack, etc. are represented by different segments). A segment describes a logical entity and therefore segmentation enhances modularity. In comparison to paging, procedures and data can be distinguished and separately protected. The sharing of procedures or data between several processes is also facilitated (which would prove much more difficult with paging).
Naturally there are tradeoffs which have to be recognized - on the one hand, the programmer has to be aware of the fact that this technique is being used, on the other hand, the whole segment has to be resident or not. Fragmentation adds to the list of problems.

2.5 Scheduling

MinOS supports multi-tasking to provide the user with the illusion of parallel program execution. For this reason, we implemented two preemptive scheduling algorithms which will be discussed in detail.

Whichever scheduling algorithm is employed, basically MinOS schedules processes as follows:
All processes which are ready to run are stored in `s_runQueue[]`. If the process which is currently running has used up its quantum, a timer intterupt is fired and MinOS invokes `Schedule()`, which assigns the CPU to the process provided by `Get_Next_Runnable()`. Depending on which scheduling policy is set, `Get_Next_Runnable()` calls the appropriate function to get the next process.

- **Priority-based Round Robin (RR)**

  Round robin is a very simple and staightforward scheduling algorithm. There is a single queue for processes ready to run. When a new process is scheduled to run, the process on the head of the queue is taken. If the process has used up its

quantum without blocking it is put at the back of the queue. There is no differentiation between I/O-bound and CPU-bound processes, thus favoring CPU-bound processes, because these tend to use the full quantum, while I/O-bound processes often block before using up their quantum. This produces several problems:
I/O-bound processes hava a poor performance and an increased response time. In order to solve the problem mentioned above, priority-based round robin can be employed. Priority-based round robin chooses the process with the highest priority (I/O-bound processes can have a higher priority than CPU-bound processes). Of course this implicates another problem - while round robin prevents the starvation of processes, this can still occur when using priority-based round robin.

Priority-based round robin was implemented as follows:
`Get_Next_Runnable()` invokes `Find_Best()` which returns the process with the highest priority.

- **Multilevel Feedback (MLF)**

  Instead of supporting only one ready-queue, MLF supports several. Each queue is assigned a priority level. A newly created process is put on the queue with the highest priority level. Each time the process completes a full quantum, it will be demoted to the next lower-priority queue. Whenever a process is blocked, it is promoted to a higher-priority queue. Within each queue a simple FCFS mechanism is used.
  This design has the one big advantage that I/O-bound processes are favoured. Since I/O-bound processes usually block more often than CPU-bound ones, they frequently get promoted and never descend to the lowest-priority queue. Thus, MLF provides good performance for I/O-bound processes, effective use of I/O devices and good response times.
  Nonetheless here too, a tradeoff has to be made: MLF holds the danger of starvation. CPU-bound processes may starve, since they drift down to lower-priority queues, whereas processes in the upper queues are favoured.

  Windows NT 4.0 and early versions of Unix too, use a multilevel feedback algorithm. [1]

  In MinOS MLF was implemented as follows:
  There are four different priorities, therefore `s_runQueue` is now an array of `Thread_Queues` (`s_runQueue[0]` to `s_runQueue[3]`). Each time a process completes a full quantum it is enqueued in the next lower-priority queue. `Wait()` promotes the current process. In `Get_Next_Runnable()` each queue (beginning with the highest-priority queue) is searched for a runnable process by employing an FCFS mechanism. As soon as a process is found, the function `Get_Next_Runnable()` refrains from searching the remaining queues.
  As noted above, a big disadvantage of MLF is the possibility of starvation. The implementation of MLF in MinOS tackles this problem. MinOS-MLF promotes a process to a higher-priority queue after it has spent a certain amount of time in its current queue without being assigned to the CPU. This is done by checking

all processes in the system at regular intervals; if they are in danger of starvation. To be able to determine if a process is starving, a timestamp-flag was added to the `Kernel_Thread` structure. When a process enters a new queue, the value of `numTicks` (also a field in the `Kernel_Thread` structure which counts the numbers of ticks the process has run) is stored in the flag mentioned above. If the value of `numTicks` is equal to the value stored in the flag, the process never ran and therefore starves.The length of an interval can be set with `STARVATION_CHECK_DELAY` which can be found in kthread.c. Of course checking every single process in the system is costly ($O(n)$, where n is the total number of processes).

MinOS is capable of switching between the two scheduling policies at runtime. By using the system call `SetSchedulingPolicy`, a user program can choose the policy and quantum which should be applied. Switching at runtime implies some aspects which have to be considered. The transition from round robin to multilevel feedback does not pose a big challenge, because the processes automatically go through the various priority queues. When switching from MLF to RR, the individual priority queues have to be reduced to a single queue.

There is one more point worth mentioning:
ensuring that the idle thread is the last thread of the lowest priority queue is tricky and the computational effort is high. We have found another solution for this problem: the function `Get_Next_Runnable()` searches (all) the queue(s) for runnable processes. If none are found, the function returns the idle thread, which means that the idle thread never has to be inserted in a queue.

In order to test our implementations, we enhanced the abilities of shell.exe, so it would be able to run background processes, besides writing our own unit tests (see section"Unit Tests").

2.6 Semaphores

MinOS applies sempahores to synchronize processes. It supports 32 semaphores with a maximum name length of 25 character, storing them in an array.
To mark the semaphores as occupied or free, we make use of a bitset. Using a bitset offers a crucial advantage: rather than iterating over an array to find a free slot, a bitset uses bit-shifting which is much more cost-effective. The value supplied by the bitset function `Find_First_Free_Bit` is used as the index for the array. Each process has to store the semaphore IDs it has acquired, for which purpose once again a bitset was employed.

The following list is an extract of functions implemented to realize the concept of semaphores:

- **int CreateSemaphore(char*, int)**
  This function accepts two parameters: a string representing the semaphore's name and an initial count value. If the corresponding semaphore is found or newly created, its ID is returned. The value -1 is returned, if the maximum amount of semaphores has already been reached.

- **int wait(int) (= P())**
  Takes the semaphore id, acquires the semaphore and blocks.

- **int signal(int) (= V())**
  Takes the semaphore id, releases the corresponding semaphore and wakes up the next process waiting for this semaphore.

- **int DestroySemaphore(int)**
  This function accepts a semaphore ID as parameter. It returns the value -1, either if there was no reference to the invoking process or no semaphore with the given ID was found. Otherwise it removes the references and returns 0. If the reference counter is 0, the semaphore is being destroyed.

For testing purposes we have written some unit tests (`sem1a.exe, sem1b.exe, sem2.exe`). For detailed information on these tests please check on section ”Unit Tests”.

There is one last point in our implementation which is worth to be mentioned here: We have provided the kernel with an additional functionality. If a process quits without having its semaphores properly destroyed, the kernel recognizes this and destroys them for the process. This functionality is supplied in `Destroy_Thread()` which can be found in `kthread.c`.

2.7 File System

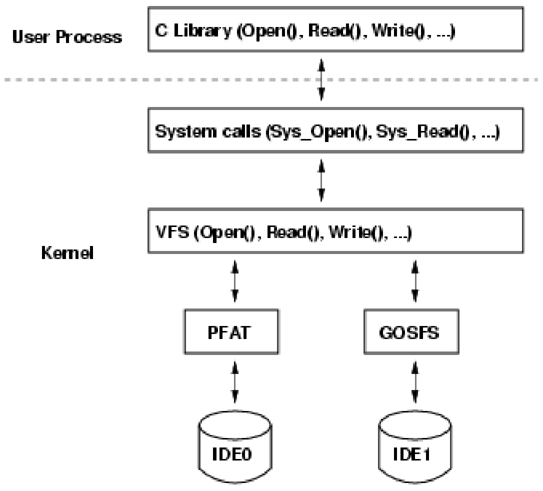
MinOS supports two different file systems:

- PFAT (read-only)
  The Pseudo FAT filesystem was already provided by GeekOS. The hard disk where the kernel is stored (`diskc.img`) is formatted with this file system.

- GOSFS
  We implemented a basic file system called GOSFS to familiarize ourselves with this topic. The code of our file system can be found in the files `gosfs.h` and `gosfs.c`.

2.7.1 Virtual Filesystem Layer

GeekOS uses a Virtual Filesystem Layer to hide the various implementations of file systems from the user programs. Moreover, by employing a VFS layer it is also possible to use multiple filesystems simultaneously. When a user programm invokes a file system syscall, it will pass the request to the VFS, causing it to invoke the corresponding function of the underlaying file system.


User Process
C Library (Open(), Read(), Write(), ...)
System calls (Sys_Open(), Sys_Read(), ...)
Kernel
VFS (Open(), Read(), Write(), ...)
PFAT
GOSFS
IDE0
IDE1

2.7.2 Buffer Cache

For direct access to the hard disk a buffer cache is applied. A buffer cache provides an abstraction of the hard disk and simplifies access to it. A single buffer represents a whole disk block. The buffer is kept in memory (and therefore it can be easily accessed), after it has been modified, it is written back to disk.

2.7.3 GOSFS

GOSFS supports a maximum hard disk size of 32MB. For read and write operations addressed to the hard disk it uses a buffer cache. Filenames can be up to 127 characters long.

Design aspects of GOSFS:
In GOSFS 8 IDE sector swith a size of 512 bytes are a block. So the size of a block is 4kb. GOSFS supports up to 8192 blocks. The superblock is located in the first block of the hard disk. It comprises several fields: the magic number (unique filesystem identifier) of GOSFS is 'GOSF'. The next field of the superblock points to the block containing the root directory. The field named "size" stores the file system's (disk) size. The free blocks bitmap contains all the information necessary to manage the unused blocks of the disk.
A block contains directory entries or file data. A directory entry supplies the information, whether it is a directory or a file. If it is a directory the `blockList` of this entry contains pointers to other blocks where you can find the directories and files stored in this directory. If it is a file, the `blockList` points to the blocks with the file data.

The following list is an extract of functions implemented to realize the GOSFS file system:

- **Format**
  This function prepares a hard disk for GOSFS. First the hard disk size will be calculated, then the superblock and the root directory will be written.

- **Mount**
  With mount you can make a GOSFS formatted hard disk available for read and write operations. The method will check the superblock and root directory integrity.

- **Open**
  Opens a file. The method will check the path and then goes through the directories until it finds the file. If the file can not be found or created or if it is a directory, it will return an error code.

- **Open_Directory**
  This function opens a directory. It is almost the same as Open.

- **Create_Directory**
  Creates a directory at the end of the specified path.

- **Stat**
  Stat searches in the given path for the requested file or directory and returns meta-informations about it. The informations will be passed to the Virtual File System.

- **FStat**
  Extracts the file meta informations of an given file and returns it to the virtual filesystem layer.

- **Read**
  Read gets a given number of bytes from the harddisk and writes it into a buffer in the memory..

- **Read_Entry**
  Copies the information from a directory entry to the virtual file system layer.

- **Write**
  Writes a given number of bytes from a buffer in memory to the harddisk and updates the meta-information of the file. Updates the superblock.

- **Seek**
  Seek changes the current read/write position in a file.

- **Close**
  Closes an open file. All references in the kernel will be deleted.

- **Delete**
  Deletes a given file or directory. After deleting, the method will update the superblock.

- **Sync**
  Sync flushes all filesystem buffers to the disk.

2.8 System Calls

The technique of system calls was incorporated to provide an interface to the system's data and services.
For details, please check `syscall.c` which can be found in the source code of our project.

List of implemented system calls:

- General system calls:

Null:	Does nothing but immediately returning to the user program.
Exit:	Calls all function needed to terminate the user program.
PrintString:	Print a string to the console.
GetKey:	Get a single key press from the console.
SetAttr:	Calls the corresponding function to set the current text attributes.
GetCursor:	Get the current cursor position.
PutCursor:	Set the current cursor position.
Spawn:	Create a new user process.
Wait:	Wait for a process to exit.
GetPID:	Get process id of the current thread.

- Scheduling and semaphore system calls:

SetSchedulingPolicy:	This system call allows switching the scheduling policy at runtime.
GetTimeOfDay:	Get the time of day.
CreateSemaphore:	Create a semaphore.
P:	Acquire a semaphore.
V:	Release a semaphore.
DestroySemaphore:	Destroy a semaphore.

- File I/O system calls:

Mount:	Mount a filesystem.
Open:	Open a file.
OpenDirectory:	Open a directory.
Close:	Close an open file or directory
Delete:	Delete a file.
Read:	Read from an open file.
ReadEntry:	Read a directory entry from an open directory handle.
Write:	Write to an open file.
Stat:	Get file metadata.
FStat:	Get metadata of an open file.
Seek:	Change the access position in a file.
CreateDir:	Create a directory.
Sync:	Flush filesystem buffers.
Format:	Format a device with a given filesystem.

3 Programs

3.1 User Programs

The following list is an extract of user programs provided by MinOS:

- **format.exe**
  Formats a block device by writing filesystem metadata to the device so that it can be mounted.
  Usage: format.exe device fstype
- **mkdir.exe**
  Creates a directory
- **mount.exe**
  Mounts a block device on a given path in the overall filesystem.
  Usage: mount.exe device path-prefix fstype
- **touch.exe**
  Changes file access and modification times. If the file does not exist, it is created.
  Usage: touch.exe filename
- **shell.exe**
  Basic shell for executing commands

3.2 Unit Tests

For testing our various implementations we have used some user programs provided by GeekOS as well as some self-written user programs, which served as unit tests.

- **sem1a.exe**
  Tries to create more semaphores than the system is able to support.
- **sem1b.exe**
  Tests if the kernel destroys those semaphores which were not freed by the thread itself.
- **sem2.exe**
  This unit test tries to create multiple semaphores which have the same name.
- **sched.exe**
  Uses scheda.exe and schedb.exe to visualize the chosen scheduling algorithm and quantum.
- **starve.exe, block.exe, quick.exe**
  These programs test the MLF-Scheduler's ability to prevent the starvation of processes.
- **workload.exe**
  This program is provided by GeekOS for testing semaphores and scheduling algorithms. It can also be used to switch scheduling policies at runtime.
  Usage: workload.exe policy quantum
  policy: rr (round robin), mlf (multilevel feedback)
  quantum: digit between 2 and 100

4 Conclusion

The main objective of this project was to build a basic and minimalistic operating system intended to run on the bare metal. By working ourselves through the various parts constituting an operating system, we gained an insight into the overall layout of the latter and its principles. Aiming to incorporate some form of concurrency support, we implemented two different scheduling algorithms and used semaphores to synchronize concurrent processes. Concerning memory management, we opted for segmentation out of reasons discussed in the section on ”Memory Management”. In terms of file handling, we created our own, rather basic, file system based on the guidelines of GeekOS.

GeekOS is a good way to get started, especially if one is facing the challenge of building an operating system for the first time, since it limits the expenditure of time needed for lowlevel hardware programming.
However, there are also some aspects we did not like about it. The system is monolithically structured which may be a disadvantage. The internal structure of the kernel provides a further reason for criticism, because it seperates components from each other which are logically linked. For example the code of the scheduler is spread over various parts of the system, instead of containing it in one single file. As GeekOS contains several bugs (which are time-consuming to find), the time saved at the outset is lost.

Since operating systems had aroused our interest long before this course, we are currently toying with the idea of rewriting our operating system based on the knowledge acquired during the past semester and without GeekOS as our vantage point.

References

[1] www.wikipedia.org.

[2] Ruediger Brause. *Betriebssysteme - Grundlagen und Konzepte*. Springer, 2004.

[3] TIS Committee. *Unix in a Nutshell*. O'Reilly, 2005.

[4] William Stallings. *Operating Systems - Internals and Design Principles*. Prentice-Hall, 2005.

[5] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 2001.

[6] Andrew S. Tanenbaum. *Operating Systems Design and Implementation*. Prentice-Hall, 2006.