# hrwOS: Technical Report

Stefan Huber, Christian Rathgeb, Stefan Walkner
Magisterstudium Angewandte Informatik
Universität Salzburg

February 9, 2007

**Abstract**

We have to build an operating system in the course *Ausgewählte Kapitel der Betriebssysteme* of Prof. Kirsch at the University of Salzburg. This operating system has to deal at least with Memory Management, Process Management, Concurrency, File Handling and Inter Process Communication. This report will describe our operating system and its components in detail.

# Contents

# 1 Introduction

Prof. Kirsch held the course *Ausgewählte Kapitel der Betriebssysteme* (Special topics on Operating Systems) in the winter semester 2006/07 at the University of Salzburg. In his opinion someone can only completely understand the concepts of operating systems if such a system is implemented by oneself. Therefore the main exercise in this course is the creation of an operation system in groups of two or three students. This operating system has to deal at least with the following concepts:

- Concurrency support
- Memory Management
- Device Abstraction
- File Handling

At the end our operating system has to run a non-trivial concurrent application. To cope with this exercise in a single term we are allowed to use some start-up frameworks like GeekOS [2], PintOS or similar ones. Those operating systems provide some kind of an operating system skeleton. One part is to pass several subprojects to fill out code step by step and by doing so the functionality of the operating system increases. For example, project 1 of GeekOS deals with the ELF binary loader, Project 2 with segmentation and process creation and so on.

**Project Management** GeekOS and PintOS are very similar operating systems. Our choice was GeekOS because it has more finely granulated subprojects. On the next step we defined project milestones and an estimated schedule:

1. *Getting Started*
   Choose GeekOS, Milestones, Schedule
2. *ELF Binary Loader*
3. *Process Management*
   Segmentation, Process Creation, Syscalls
4. *Scheduling*
   Schedulers, Semaphores
5. *Virtual Memory*
   Paging, Swapping
6. *Filesystem*
7. *IPC*
   Pipes
8. *Running Concurrent Application*

The development progress can be seen on the website of hrwOS [6]. All milestones, except the file system, have been finished in time.

**GeekOS** GeekOS comes with a 46-page documentation `hacking.pdf` which is contained in the GeekOS tarball [1]. On the website of GeekOS [2] you can find the introductory words about GeekOS.

> GeekOS is a tiny operating system kernel for x86 PCs. Its main purpose is to serve as a simple but realistic example of an OS kernel running on real hardware. (Actually, most of the development is done on the Bochs emulator.)
>
> The goal of GeekOS is to be a tool for learning about operating system kernels. As of version 0.2.0, it comes with a set of projects suitable for use in an undergraduate operating systems course, or for self-directed learning. GeekOS has been used in courses at a number of colleges and universities.

In the following report we will describe the main components of hrwOS. These components are vaguely the project milestones described above. To avoid a bloated report we will not discuss source code.

## 1.1 HrwOS Basics

Because hrwOS is inherited from GeekOS they share basic properties. First of all GeekOS is written for the IA-32[1] architecture of the Intel processors. It's almost sure that the programming language is C – only a few hundred lines are nasm assembler code. Namely the kick-off code at the very first boot time and some low level routines to switch (back) to user mode which is used after a schedule.

Originally GeekOS is written to run on the x86 emulator Bochs. Focused on design it's a monolithic kernel. Many design principals are similar to those in Unix/POSIX. For example, the design of the VFS, the treatment of stdin/stdout, pipes and so on. The boot code, for example, uses a few lines from the Linux source code.

In short, hrwOS is a monolithic, multi-threaded, multi-programming, single-user, x86 operating system which uses paging and supports pipes and at least two concrete file systems.

[1] IA-32 is also known as x86

# 2 System calls

After booting hrwOS the init process is started. This process is a shell which gets as stdin and stdout the console. The shell waits for further commands and uses the system calls (Tab.-1) to interact with hrwOS. For example, to launch a new process – lets say 'echo', 'cp' or maybe a second instance of the shell itself – it invokes `Spawn`.

## 2.1 Invocation

To invoke a system call hrwOS makes use of, like most other operating systems, the (software) interrupt system of the IA-32 architecture. Like in Linux the register `eax` contains the number of the system call and the arguments are given via the registers `ebx`, `ecx` and so on. After loading the parameters a `int $0x90`[2] triggers an interrupt with the number `0x90`. Chapter 5 of [3] gives a detailed description of the interrupt system of the IA-32 architecture.

The processor looks up the corresponding entry in the IRQ table and calls the assembly macro `Int_No_Err` in `lowlevel.asm` which afterwards calls the function `Handle_Interrupt`. At this point the code is already executed in kernel mode and no longer in user mode and has access to everything. Here the current state is saved to determine the origin of the interrupt. Now the step to C code follows. The system call handler `Syscall_Handler` has an entry at `0x90` and the system call is further dispatched by `eax` to the specific system call handlers.

After performing the system call handler the calling stack is worked off down to the assembler function `Handle_Interrupt`. At this point the decision is made whether a schedule should be performed or not. This is the only point where scheduling is taken into account.

## 2.2 Application Interface

Because programming in assembler is not very convenient, a little libc library supports the application programmer. This library has C wrapper functions for the system calls and higher level functions which are based on these. However, system calls are the only way to communicate with the system on which an application is executed. So the set of system calls can be called the API of the operating system for an application programmer – all higher level libraries are built on it.

Due to the design of the system calls one can recognize the relation to POSIX and Unix. The system calls `PrintString` to `PutCursor` are relics from a time before stdin and stdout were handled over VFS. It would be no problem to get rid of them. There is another difference concerning the system call `Spawn`. In the POSIX world `Spawn` is unknown: There is a system call `fork` which splits up a process into two exact copies where one is parent of the other. To load another excutable a combination of `fork` and `execve` is offered. The latter loads the code section of the process from an executable file.

[2] Linux uses `0x80` here.

| No. | Syscall | Description |
|---|---|---|
| 0 | `Null` | Does nothing. |
| 1 | `Exit` | Terminate current process |
| 2 | `PrintString` | Print string on console |
| 3 | `GetKey` | Get key from keyboard |
| 4 | `SetAttr` | Set printing attributes of console |
| 5 | `GetCursor` | Get position of cursor on console |
| 6 | `PutCursor` | Put the cursor on specific position |
| 7 | `Spawn` | Launch new process by path of executable |
| 8 | `Wait` | Wait for the termination of specific process |
| 9 | `GetPID` | Get PID of calling process |
| 10 | `GetQuantum` | Get time resolution of scheduling |
| 11 | `GetSchedulingPolicy` | Get current scheduling policy |
| 12 | `SetSchedulingPolicy` | Set current schedulint policy |
| 13 | `GetTimeOfDay` | Get the current time |
| 14 | `CreateSemaphore` | Create named semaphore |
| 15 | `P` | Semaphore down |
| 16 | `V` | Semaphore up |
| 17 | `DestroySemaphore` | Destroy the semaphore |
| 18 | `Mount` | Mount filesystem to VFS |
| 19 | `Open` | Open specific file |
| 20 | `OpenDirectory` | Open specific directory |
| 21 | `Close` | Close specific file or directory |
| 22 | `Delete` | Delete file or directory |
| 23 | `Read` | Read from opened file |
| 24 | `Write` | Write to opened file |
| 25 | `Stat` | Get stat of specific file |
| 26 | `FStat` | Get stat of an opened file |
| 27 | `Seek` | Set cursor in opened file |
| 28 | `CreateDir` | Create specific directory |
| 29 | `Sync` | Sync all filesystems |
| 30 | `Format` | Format block device with specific filesystem |
| 31 | `CreatePipe` | Create unnamed pipe |
| 32 | `CreateNamedPipe` | Create named pipe |

Table 1: System calls of hrwOS

# 3 Memory Management

After System Calls Memory Management is the next essential topic to know for implementing applications. Like GeekOS and most other Operating Systems, hrwOS makes use of paging to cope with the concept of Virtual Memory. The IA-32 architecture of Intel lays down the way of using paging straight forward. Chapter 3.3 of [4] gives basic introductions to the memory management of IA-32. Chapter 3 of [3] gives a more detailed insight into paging and related topics.

## 3.1 Memory Model

In the following we will use the terminology of Intel. Intels memory address translation is a two-phase process. First of all the virtual address is translated into a linear address by segmentation. This leads only to offsetting combined with limitation so that a certain continuous piece of memory (in linear address space) is accessed. The second phase uses paging to map used pages in linear address space to a concrete page frame in physical memory. While paging has to be enabled explicitly, segmentation is always present. The only way to "disable" segmentation is to configurate a single segment of the whole 4Gb linear address space. Something similar has to be done if more than 8192 processes want to be supported because the descriptor tables only support $2^{13}$ entries. This limit existed for Linux 2.4 kernels – in Linux 2.6 this limit has been removed. In contrast to that hrwOS uses segmentation.

To simplify memory management significantly hrwOS maps the whole physical memory to the lower address space (see Fig-1). By this the kernel has (virtually) access to the complete physical memory without thinking about paging. Therefore the kernel uses the lower 2Gb as kernel mode memory. The upper 2Gb of the linear address space is user space. Similar to most other operating systems at the lower addresses code and data are located. The stack grows from higher addresses to lower ones and straight above the stack the process arguments are located, which might be passed from the shell. This scheme limits the process to 2Gb.

Note that the mapping from linear address space to physical memory is a per-thread-view. This means that another thread has a different mapping to physical memory because user space is located somewhere else in RAM. Despite this fact, the mapping of the physical memory is done for *every* thread and therefore kernel space code can be thought of as accessing real physical memory, no matter which thread is actually executed. So kernel code execution is equal for all threads.

In hrwOS the mapping from virtual address space to linear address space depends on the mode of execution. For kernel space a segment is set up over the whole linear address space in contrast to the user space whose segments are only map to the higher 2Gb of the linear address space. By this user space code cannot even see the kernel which isn't needed anyway.

## 3.2 Organization

GeekOS already provides two kinds of memory allocation functions. First of all an ordinary `Malloc` is provided to allocate data structures for the kernel itself. For allocating whole pages (actually page frames) one can call `Alloc_Page`.

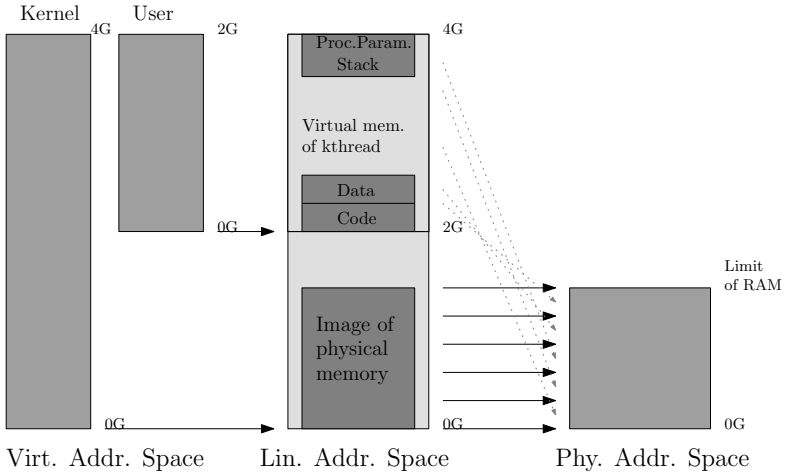


Figure 1: Virtual memory model of a kthread.

There is also a similar function `Alloc_Pageable_Page` which sets the pageable bit of the page. To manage the free and used page frames hrwOS maintains a linked list of `Page` structs. These structs actually represent page frames and carry aditional information. For example, a pointer to the corresponding page table entry and so on.

The page allocation functions have been slightly patched for hrwOS. Originally the `Alloc_Page` function of GeekOS was not able to page out pageable pages if there was no free memory available. This capability only had `Alloc_Pageable_Page`. On the other hand when a new process is spawned, there is heavy use of `Alloc_Page` to avoid page outs of important structures like the page tables itself. By doing so it was not possible to spawn new processes if there was no free page. This way swapping was never performed and was therefore useless. This drawback of `Alloc_Page` has been fixed.

## 3.3 Swapping

When using paging to implement virtual memory, swapping comes almost for free. First of all, there are two main possibilities to maintain swap space: First of all one can use a swap-file on an existing filesystem like Windows or Mac OS X does. Or one can maintain a whole partition/disc as swap space. This would be the ordinary Linux way. GeekOS has a file `pagefile.sys` of fixed size on the first disc which is formated with PFAT – a GeekOS filesystem of its own.

When allocating a new page and no free page exists a page is swapped out to swap space. To determine a "good" page which is not often used, we use a kind of LRU (Least Recently Used). The function `Update_Clock` updates the `clock` member of all `Page` structs periodically[3] for this purpose.

[3]Actually, not periodically in relation to a real clock – but periodically relative to page fault events.

# 4 Processes

## 4.1 User context and kernel threads

In hrwOS every process (actually user context) is represented by a `UserContext` struct. If a process is spawned, a new user context instance is created. If a process terminates the corresponding user context is destroyed. Every instance contains the following properties:

- LDT (Local segment descriptor table)
- Descriptor for code and data segment
- Page directory for the corresponding memory mapping
- Entry, argument block and stack addresses
- Reference counter (if several threads belong to a single user context)
- Semaphore list
- Open files array

At this time multiple threads per process are not supported. However, this could easily be implemented by adding a corresponding system call. This would create a new thread which maps to the current user context. Threads are represented by a `KThread` struct. This struct contains the data of a thread which is necessary to enable context switching, waiting for other threads and so on:

- Stack pointer (which is saved when a context switch is performed)
- No. of ticks (used to indicate a re-schedule)
- Priority
- A page for the stack
- Pointer to user context
- Owner (The thread wich spawned it)
- Reference counter
- Alive and blocked flag
- A join queue (to let other threads wait for that one)
- Exit code
- PID
- The current ready queue (used for the MLF scheduler)

**Process creation** When the system call `Spawn` is called, the following procedure is performed: First of all the executable is read from the filesystem. After that the ELF binary is parsed and all important structures are extracted from it. When this is done the new user process can be set up. This is done by creating a new user context, setting up the paging structures, copying the data into memory and setting up the stack and argument block. At last the segments are set up. When the user process is set up a new kernel thread is created and mapped to the new user context.

**Context switch** A context switch in hrwOS is done via the interrupt system. In chapter 5 of [3] one can read that when an interrupt is performed the stack remains in a specific format. Values which are necessary to restore the control flow in the user space have been pushed. Namely the instruction pointer, stack pointer, code and data segment descriptors and so on. This fact can be used to perform a context switch by just replacing the values on the stack by the corresponding values of the new thread.

In Sec.-2.1 the mechanism of a system call has been explained. Just before exiting the system call a decision is made whether a schedule should be performed. If a new schedule has been performed the pointer `g_currentThread` is set to the winner thread and `esp` is set to its stack. Before returning from the interrupt the corresponding values of this thread are pushed on the stack as it's expected from `reti` and mentioned in the previous paragraph. The context switch is done after calling `reti`.

## 4.2 Scheduling

In hrwOS multiple schedulers can be used and changed at runtime. For this purpose two system calls, namely `SetSchedulingPolicy` and `GetSchedulingPolicy`, have been implemented. Currently we have implemented two scheduling algorithms which will be described in the following.

### 4.2.1 RoundRobin Scheduler

The RoundRobin scheduler is implemented to just work and not to do any unnecessary fancy stuff. It was designed to enable testing of user space program at an early stage of progress in this project. Because this implementation does not take process priorities into account, starvation is not possible and the scheduler is able to find the next process in O(1).

### 4.2.2 MultiLevelFeedback Scheduler

This implementation of MultiLevelFeedback scheduler determines the next process to run in O(1). To increase the responsiveness of the system when running a lot of CPU intensive processes it uses multiple ready queues. A low queue index means higher priority and a higher index means lower priority.

Every time a process uses up its quantum it gets moved into a lower prioritized queue. The idle thread is pushed to the lowest prioritized queue immediately when it's scheduled. Whenever a process gets blocked (i.e I/O) it is moved to a higher prioritized queue for being re-scheduled faster when being unblocked. This scheme allows on the one hand a high response rate of the

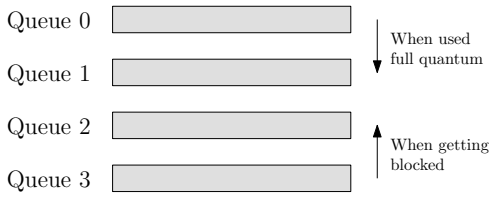


Figure 2: Transitions of threads at an example of four queues.

system when having cpu intensive processes and a preferred handling of I/O intensive processes that have to wait a lot on the other hand. The original implementation idea of MLF is, that processes from a queue are scheduled, if all higher prioritized queues are empty. Our approach is the following.

At every schedule a specific queue is handled by taking the first thread in this queue. We have several constraints considering the sequence of choosing queues:

- Queues with lower indices should be taken more often.
- Starvation has to be avoided – every queue must be taken in a specific amount of time.
- Clustering should be avoided – taking the same queue several times in a row is bad for response time and unfair.

**Mechanism** What we want is a perfect mixing sequence like the following: 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, .... To reach this behaviour we use a counter and consider the least significant bit which is one. The number of this bit indicates the queue which is selected. This mechanism is illustrated in Tab.-2. A possible pseudo code to implement this might look like the following.

```
for ( level=0; level<QUEUE_LEVELS; level++ )
  if ( (counter & (1<<level)) == (1<<level) )
    break;
counter ++;

if ( counter == (1<< QUEUE_LEVELS) )
  counter = 1;
```

**Analysis** When doing so we get a distribution of queues which is illustrated in Fig.-3. Furthermore, the time $T_i$ when the queue $i$ gets selected again, is constant and $T_{i+1} = 2 \cdot T_i$. So the algorithm is totally deterministic and if a process is in queue $n$ at position $k$ then the number of schedules to get scheduled is $k2^n$. This connection makes the scheduler highly predictable.

Note, that this pattern is by definition completely self-similar. The table Tab.-2 is symmetric relative to line 8. Furthermore the interval $[1, 7]$ of lines is symmetric relative to line 4 and so on. More generally speaking,

$$\forall n \in \mathbb{N}\ \forall k \in \mathbb{N}_0 : \qquad [k2^{n+1} + 1, (k+1)2^{n+1} - 1] \text{ symm. to } (2k+1)2^n$$

| | | | | **counter** | **queue** |
|---|---|---|---|---|---|
| 0 | 0 | 0 | **1** | 1 | 0 |
| 0 | 0 | **1** | 0 | 2 | 1 |
| 0 | 0 | 1 | **1** | 3 | 0 |
| 0 | **1** | 0 | 0 | 4 | 2 |
| 0 | 1 | 0 | **1** | 5 | 0 |
| 0 | 1 | **1** | 0 | 6 | 1 |
| 0 | 1 | 1 | **1** | 7 | 0 |
| **1** | 0 | 0 | 0 | 8 | 3 |
| 1 | 0 | 0 | **1** | 9 | 0 |
| 1 | 0 | **1** | 0 | 10 | 1 |
| 1 | 0 | 1 | **1** | 11 | 0 |
| 1 | **1** | 0 | 0 | 12 | 2 |
| 1 | 1 | 0 | **1** | 13 | 0 |
| 1 | 1 | **1** | 0 | 14 | 1 |
| 1 | 1 | 1 | **1** | 15 | 0 |

Table 2: Resulting queue-sequence when using four queues.

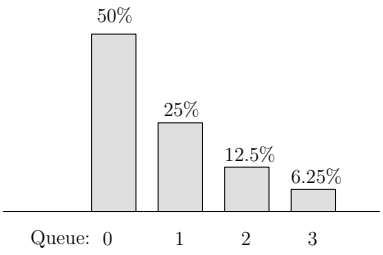


Figure 3: Reoccurence of queues when using MLF.

This yields in a fractal pattern which emphasis the argument concerning perfect mixing of queues. Furthermore you can easily see that the scheduling decision is done in $O(1)$ complexity due to a constant number of queues.

## 4.3 Semaphores

Semaphores are the only synchronization method available in user space[4]. For the implementation of semaphores a global semaphores list exists which contains all semaphores that were created by processes using the `Create_Semaphore` system call. The semaphores in the list are uniquely identified by a semaphore name (specified by `Create_Semaphore`). Every user context holds a list of semaphores for which the process did the `Create_Semaphore` system call. By using those lists we make sure that on the one hand semaphores can be shared among processes and on the other hand that a process can only use the semaphore system calls on semaphores it references to. After referencing to a semaphore a process can use the `P` and `V` system calls which work in the same semantic as suggested by Dijkstra.

Whenever a process calls `Destroy_Semaphore` the semaphore will be removed from the process semaphores list and the reference counter of the semaphore will be decreased. If the reference counter reaches zero the semaphore is removed from the global semaphores list and the memory allocated for the semaphore is freed. If a process doesn't call `Destroy_Semaphore` the reference will be removed upon process termination anyway...

To realize the semaphores the `Wait()` and `Wake_Up_One()` functions already offered by geekOS are used for `P()` and `V()` appropriately. It is necessary to use `Wake_Up_One()` to wake up exactly one waiting process instead of using `Wake_Up()` which will wake up all waiting processes, choose one of them and put the rest to sleep again. This phenomenon is called a "Thundering Herd" problem and has to be avoided in order to improve performance.

[4]In kernel mode mutexes are supported too

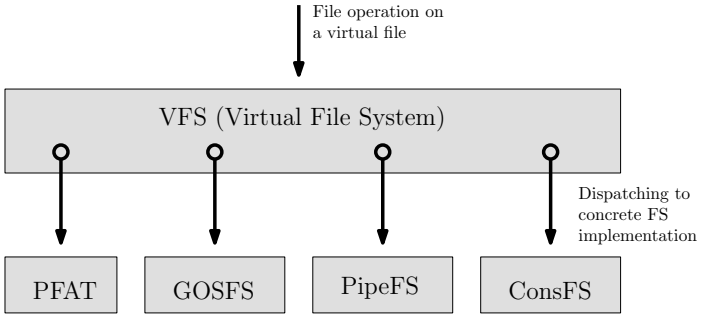


Figure 4: VFS architecture with all file systems.

# 5 Filesystem

Like any other modern Operating System, hrwOS uses an abstract file system architecture. Similar to the paradigms of OOP[5] a VFS (Virtual File System) layer handles all file operations and dispatches them to concrete file system implementations. For this purpose at boot time a virtual root `/` is available. Every concrete file system is mounted into this tree. The corresponding file system of a file can be determined by the file name. This process is similar to the one of Linux.

A possible file system implementation models not necessary a traditional file system on a block device. Reading from the keyboard and writing to a terminal can be considered as file system operations on a "virtual" console-file-system. The same holds for pipes which can be thought of being files on a "pipe-file-system".

This makes it possible to abstract the concept of `stdin` and `stdout` files. If we use the power of a virtual file system, the shell can process commands like `echo Hello > /d/output.txt` or `cat /d/output.txt | wc` because `stdin` and `stdout` can be turned round to files or pipes. With this concept the hrwOS-shell gets potentially the power of an ordinary Unix shell.

The implementation of this abstraction is similar to the one of Linux resp. to the virtual table of C++ class-instances to implement polymorphic methods. Every concrete file system implements a specific "protocol" in terms of a well-defined structure of function pointers to the concrete file system operations. Every `File` structure instance gets a pointer to this structure to map to the corresponding file system operations.

## 5.1 ConsFS

The simplest file system is ConsFS whose read operation reads keys from the keyboard and write operation prints to the terminal. Other operations like open, seek or stat are not supported. The shell is the only process for which stdin and stdout for which files from ConsFS are created. All further processes get their stdin and stdout as copies from them.

[5]Object Oriented Programming

## 5.2 PipeFS

PipeFS is similar to ConsFS. Every pipe maintains a circular buffer of a certain size to write and read into. The shell performs a call like `cat /d/output.txt | wc` as follows. The file `/d/output.txt` is opened and a pipe is created. After that the process `cat` is spawned and gets – like stdin – the file `/d/output.txt` and – like stdout – the write file of the pipe. Similar to that the wc process gets as stdin the read part of the pipe and stdout of the shell.

## 5.3 PFAT

The PFAT file system is a simple file system which comes already with GeekOS and uses a file allocation table. It does not support directory hierarchies and is used as read-only. Its only purpose is to provide the programmer with a simple start-up file system to launch processes in early development state.

## 5.4 GOSFS

Every operating system (at least every general purpose OS) must have support for saving data on persistent storage. Like most other operating systems too, hrwOS implements this as a file system on a blocking device. PFAT is not useful for this purpose since it is read-only. GOSFS should at least support the following features:

- Basic file operations: open (includes create), read, write, close, seek, stat
- Basic directory operations: create, open, enumerate files/directories, close
- Basic file system operations: format, mount

**Inode structure** Directories are used in the ordinary semantic of directory trees where each inner node is a directory which contains files and directories for itself. This abstraction is reached by an inode structure similar to that of ext2. The layout of a directory entry and the addressing scheme is illustrated in Fig.-5. Despite GOSFS does not support double indirect addressing of blocks for very large files, the code is prepared for this extension. However, by indirect blocks GOSFS can handle files up to 4.227.072 Bytes[6].

**Disc layout** The disc layout of a GOSFS partition is illustrated in Fig.-6. Like most other file systems too, one of the first entries is a magic number. By this number we want to make sure that we really mount a GOSFS partition. To support further versions we need to distinct them and have a version slot for this purpose. After that a information of the number of blocks follows. By this we know how many bitmap blocks we need. The last information in the superblock is the root directory of the file system.

The following blocks contain a bitmap where each bit indicates if a block is free or not. This is used when a new file is created, deleted or the size changes. GOSFS supports arbitrary many bitmap blocks. Therefore the size of a partition is only limited by $2^{32}$ blocks. If GOSFS is thought of being used in a real environment, GOSFS would bring much more security aspects

[6] When assuming 4k blocks resp. 8 sectors per block

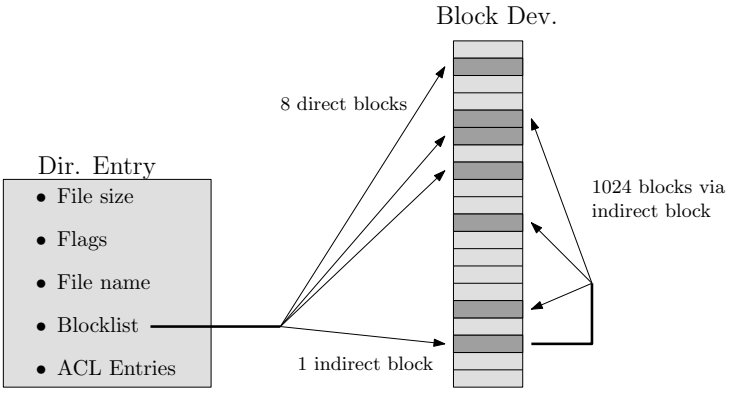


Figure 5: Inode structure and block addressing.

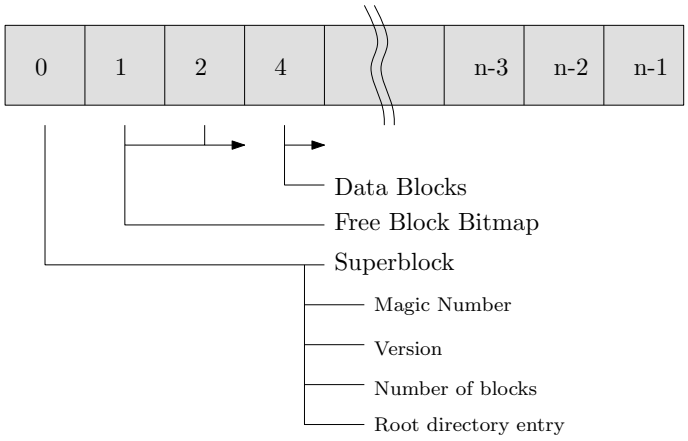


Figure 6: Disc layout of a GOSFS partition.

into design. For example checksums for the superblock and bitmap blocks, superblock backups, integrity checks of management data and things like that.

**Directory content** In general, GOSFS does not distinguish between files and directories[7]. That is, the content of a directory is saved as the content of a directory. The content of a directory on the other hand is an array of directory entries. By this scheme the basic file access can be reused for directory access. For example, finding the $n$-th block of a file/directory which has to be resolved by the GOSFS block addressing scheme in Fig.-5.

**Concurrency and Caching** GOSFS supports concurrent access to a file. This is, two distinct processes can handle the same file at the same time. This is made possible by the usage of `FS_Buffer` and `FS_Buffer_Cache`. If a file is re-opened by another process, the file is not really accessed on the hard disc.

[7] At inode level. At a higher instance there is a difference, of course.

In reality this process gets its own `File` object which points to the already opened `GOSFS_File` object. Only the process which opens a file at first creates the `GOSFS_File` object too. The whole communication is handled by `FS_Buffer` objects which are owned by `GOSFS_File`. By doing so, one has two great advantages:

- Support for caching often accessed data from hard disc
- Support for concurrent access via lockings in `FS_Buffer`

**Preventing fragmentation** File systems of this type have the problem of increasing fragmentation. There are several ways to reduce, avoid or compensate fragmentation. One could compact the blocks after deleting a file which is expensive. This job could also be done in a certain amount of time, for example by a defragmentation job like its done in Windows. Another way is the one of ext2 file system. Ext2 divides the partition into several equal-sized groups of blocks. A file on ext2 is distributed on this groups as uniformly as possible (see [5]). By this the tendency of a smooth head positioning increases. This principle could easily be implemented by GOSFS.

# 6 Final remarks

## 6.1 Getting Started with hrwOS

Unpacking the source of hrwOS, do the following steps to get hrwOS running:

1. Go to subdirectory `build`
2. Type `make clean && make depend` to clean the project and renew the dependencies
3. Type `make` to compile the project
4. Type `bochs` to start the simulation
5. After booting a shell appears with a prompt `$`

**NOTE:** During development - which we did on various machines and operating systems - we found out that the combination of which "gcc" and "bochs" versions are used is crucial. Unreproducible crashes and similar might be the affects. We developed in bochs-2.1.1 and gcc-4.1.1.

A good way of starting is to call `ls /c/` to list all available executables. By this `ls` itself is shown which was just called. Most of these should be quite self-explanatory. Some special user space programs are described in the following:

**cp, ls, mkdir** This tools should be known by any POSIX-OS user. Calling those with no arguments, shows the way to call them.

**p5test** This is a automatic testing tool which tests the primitive file system operations.

**Dining philosophers** To meet the requirements of the Operating System course we had to build a concurrent application. We decided to implement the dining philosophers. To start the program just type for example `phitable 4 2`. The first parameter is the number of philosophers to create and the second the number of meals each philosopher has to take.

We chose a server-client like architecture where each philosopher spawned communicates with the "server" (which is called "phitable") using pipes. The name of the pipe to send messages to the server and the name of the pipe to read messages from the server are passed to each philosopher via program arguments. Some other parameters like the number of iterations to do and the name of the semaphore for guaranteeing mutual exclusion while accessing the pipes are passed as well. So there are two pipes and one semaphore created by the phitable for each philosopher. For having a standardized way for communication there exists a little protocol: A philosopher that wishes to acquire its left and right forks has to send "Acquire <LeftForkID> <RightForkID>" which is then answered by the server with either `200 OK` or `400 NOT OK` to indicate whether the acquire was successful or not. In order to release the forks "Release <LeftForkID> <RightForkID>" is expected which is again responded by the server with `200 OK` or `400 NOT OK`. So the synchronization problem in this implementation does not arise because of having to acquire the left and right forks

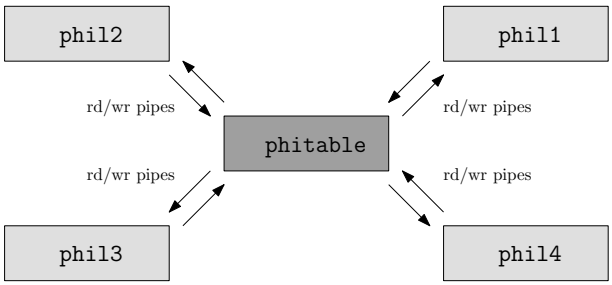


Figure 7: Client/Server architecture of the Dining Philosophers implementation.

but having to synchronize the communication. Therefore a mutex is used whenever the pipes are accessed. Because the lack of some event based I/O mechanisms or similar reading messages is done using busy waiting.

## 6.2 Future work

There are several points which could be implemented to improve functionality and performance. Some of this extensions are easy to write. Others require just lots of code. Some of this extensions are described in the following.

**Multiple threads per process** To implement multiple threads per process hrwOS has to be slightly modified. This requires a system call `Fork` or similar which starts a new thread which points to the same user context. Generally speaking this should be the trick.

**User-/kernel-thread mapping** Currently there is no difference between a thread in user space and a thread in kernel space. One could think about different concepts to, for example, improve performance and the possibility to generalize hrwOS to a multi-processor operating system. For example a $m : n$ mapping could be implemented where a pool of $n$ kernel threads exists. And every user thread, when executed, is assigned to a kernel thread.

**Heap** The heap is one of the work intensive future-work ideas. We didn't implemented a heap because this would have been a very time-intensive task and a heap was not immediately necessary for our purpose. Furthermore a heap is a user-space concept and the kernel only offers growable space which is managed by the user space `Malloc`.

**Implementing a procfs** Currently there is no way to get information about the current state of the operating system like the number of processes, free space and things like that in user space. To fill this gap one could implement another virtual file system, a procfs like it's done in Linux. By this arbitrary many information can be transported to user space. This would also be very interesting for debugging.

**Implementing events** A process of hrwOS has no possibility to wait for a specific event. For example, a philosopher at the dining philosophers problem

waits until a fork gets free. Currently the process has to do busy waiting. Implementing a events resp. conditional variables could solve this drawback.

## 6.3 Annotation

At the beginning of this report we cited Prof. Kirsch (cf. Sec.-1) about the motivation of implementing a Operating System. In the end we can agree with the following reason.

One can learn a lot about building a house by reading books which describe facts about statics, properties of materials and things like that. Reading dozens of books, one gets to know a lot of problems and solutions building a house.

The same holds for building Operating Systems. But does this person know what it's like sitting hours and hours in front of disassembled code and trying to find a bug of accessing invalid memory? Furthermore recognizing after 3 or 4 hours that a linked list is not initialized or a index in the LDT is invalid. How important it is...

- ...to have symbol tables of the kernel code available for the debugger!
- ...to think very accurately about every line of code because thinking 30 minutes more can save 2 hours of debugging!
- ...to intensively use asserts in code!

In short, we can say, that we learned a lot about building a Operating System. From now on, configuring the Linux kernel is seen from a completely different point of view. But we didn't just learned a lot about coding and the internals of the IA-32 architecture. Because we have been a group of three students we had to use a version control system, in our case subversion. This is a completely another way to code because the own thoughts have to be written down so that the others can read it. Furthermore we defined a whole schedule for this project where deadlines had to be met (cf. Sec.-1). In conclusion, this course was a very instructive experience in many respects.

# References


[1] daveho@users.sourceforge.net. Geekos tarball. `http://sourceforge.net/project/showfiles.php?group_id=35950`, 2006.

[2] daveho@users.sourceforge.net. Geekos website. `http://geekos.sourceforge.net/`, 2006.

[3] Intel ®. *IA-32 Architecture Software Developer's Manual: Volume 3A, System Programming Guide, Part 1*, 2006. Order Number: 253668-020US.

[4] Intel ®. *Intel ®64 and IA-32 Architectures Software Developer's Manual: Volume 1, Basic Architecture*, 2006. Order Number: 253665-021.

[5] Rémy Card, Theodore Ts'o, Stephen Tweedie. Design and implementation of the second extended filesystem. `http://e2fsprogs.sourceforge.net/ext2intro.html`, 2006.

[6] Stefan Huber, Christian Rathgeb, Stefan Walkner. hrwos website. `http://www.cs.uni-salzburg.at/~ck/wiki/index.php?n=OS-Winter-2006.HrwOS`, 2006.

[7] A. S. Tannenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, 2001. ISBN-13 978-0130313584.

[8] tbutler@uninetsolutions.com. Bochs website. `http://bochs.sourceforge.net/`, 2006.

# Index