BOS

Basic Operating System

Technical Report

Operating Systems Course

Winter 2006/2007

Team Members:

Bernhard Danninger 0422594
Clemens Krainer 9020112

Salzburg, 8 February 2007

Abstract

One part of the Operating Systems Course, Winter 2006/2007 of Prof. C. Kirsch (Department of Computer Sciences, University of Salzburg) was to design and implement an operating system in a student chosen programming language. It was required, that the operating system at least includes some concurrency support, memory management, device abstraction, and file handling.

This paper presents the main concepts of the *Basic Operating System*. First, this document describes the goals of the BOS team, followed by the characteristics of BOS and the used development environment. Second, this document outlines memory management, as well as processes and threads. Third, it describes the implementation of inter process communication, input and output, and file systems. Finally, this work describes the class demonstrations, followed by instructions for unpacking and building BOS.

Content

1. Introduction 4
2. Memory Management 5
2.1 Virtual Memory 5
2.2 Page Replacement 6
2.3 User Heap 6
2.4 User Stack 6
3. Processes and Threads 6
3.1 Process Model 6
3.1.1 Kernel Threads 7
3.1.2 User Processes 7
3.2 Process Creation 7
3.2.1 Executable and Linking Format 7
3.2.2 Starting Programs 8
3.3 Process Termination 8
3.4 Scheduler 8
3.4.1 Round Robin Scheduling 9
3.4.2 Multi Level Feedback Scheduling 9
3.4.3 Wait and Ready Queues 10
3.5 Semaphores 10
4. Inter Process Communication 10
4.1 Message Queues 10
4.2 Pipes 11
5. Input and Output 12
6. File Systems 13
6.1 Virtual File System (VFS) Layer 13
6.2 The GosFS File System 13
7. Demonstrations 14
7.1 File System / Pipes 14
7.2 Message Queues 15
7.3 Scheduling 15
7.4 Paging 17
8. Unpacking and Building BOS 18
9. Conclusion 19
Appendix A Implemented File System Functionality 20
Appendix B Distribution Package 23
Appendix C The C Application Programming Interface for User Programs 24
C.1 File and File System calls 24
C.2 Kernel related Calls 25
C.3 Heap related Calls 25
C.4 Message Queue Calls 26
C.5 Process related Calls 26
C.6 Scheduling related Calls 27
C.7 Semaphore related Calls 27
References 28

1. Introduction

Our goal was to implement a basic operating system that can boot on a real 32 Bit Intel Architecture (IA 32) PC hardware and to meet the course rules. The operating system is required to include at least some form of:

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

We have implemented a *Basic Operating System* (BOS) [1] in C for the IA 32 PC platform. It contains:

- Round Robin and MLF Scheduler
- Virtual Memory Management using Paging and Segmentation
- I/O System for block devices (Floppy and IDE disks)
- I/O System for user interaction (Keyboard and Screen)
- Virtual File System Layer
- Inter Process Communication (Pipes and Message Queues)

The BOS development environment was:

- The educational operating system kernel GeekOS 0.3.0 as a base [2].
- Bochs 2.3 to emulate a complete IA 32 PC platform on Linux [3].
- gcc 4.0.2, nasm 0.98.38, vim 6.3.84 and Anjuta DevStudio 2.0.2
- COMPAQ DESKPRO XL 590 (90 MHz Pentium I)

The BOS software contains the implementation of all six GeekOS [2] projects including a user heap [1].

This document describes the highlights of the implemented BOS functionalities. Chapter 2 starts with memory management, chapter 3 continues with processes and threads, chapter 4 describes inter process communication, chapter 5 covers input and output, and chapter 6 focuses on file systems. Chapter 7 explains the demonstration programs; and chapter 8 affirms unpacking and building BOS. Chapter 9 summarises this paper by depicting our experience with real hardware and proposals for future enhancements.

2. Memory Management

This chapter describes how BOS implements virtual memory layout, segmentation, paging, page replacement, user stack, and user heap.

2.1 Virtual Memory

BOS implements the virtual memory layout recommended in the GeekOS documentation [2], including segmentation and paging. At boot time, the kernel creates a page directory and page tables that directly map all physical memory pages to the virtual addresses of the kernel space. As shown in Figure 1, the kernel virtual memory starts at address 0x00000000 and is 4GB large. From user space view, the user virtual memory starts at address 0x00000000 and is 2GB large.

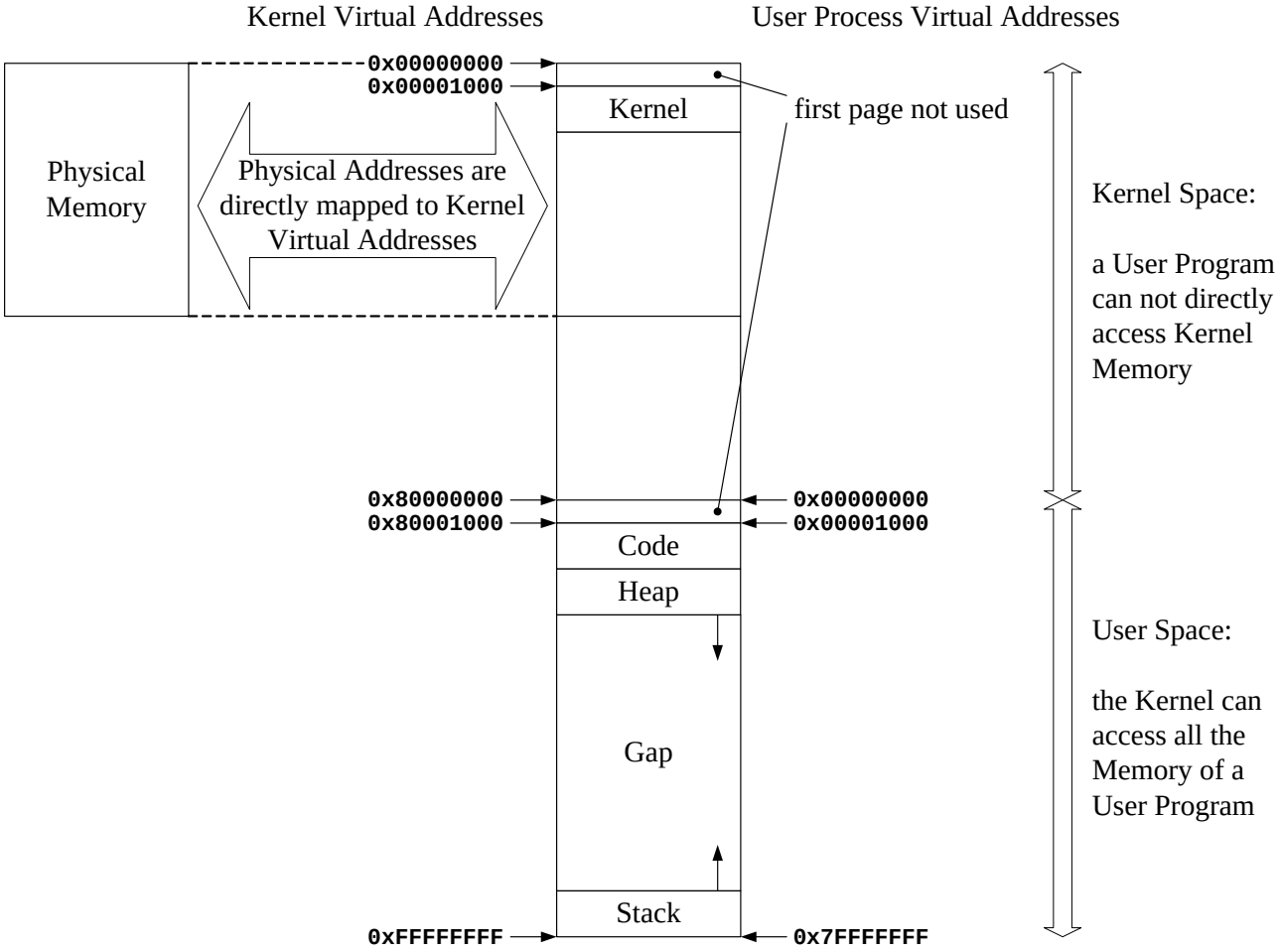

Figure 1 BOS Virtual Memory Layout

While all kernel threads share one page directory and its page tables, each user process owns a separate page directory. At creating a user process, the kernel copies the kernel space page table references to the user page directory and creates disjoined page tables for the user space memory. For a user process the kernel allocates only as much memory as needed to start it. The page fault handler adds extra pages to stack or heap when needed. Figure 1 also depicts that both the first page

of the kernel space and the first page of the user space are not mapped to physical memory. This will help trap null pointer references in the kernel, as well as in a user program.

2.2 Page Replacement

The genuine GeekOS page replacement algorithm always evicts the oldest page-able page in memory. The BOS page replacement algorithm is inspired by the WS-Clock algorithm [4]. Every timer tick, the BOS page cleaner scans the memory and records the last access time of a page in its management information. At each page fault, the page replacement algorithm starts at the last paged out page, searches a page that has not been used longer than the WORKING_SET_TIMEOUT and evicts it. If the algorithm cannot find such an old page, it randomly chooses one for eviction.

For simplicity reasons, BOS keeps pages either in memory or on disk. At loading a page back into memory, the algorithm frees the allocated space in the paging file.

2.3 User Heap

BOS implements the heap of user programs as recommended in the GeekOS documentation [2]. BOS employs the BGET memory allocation package [5] for heap management. BGET is also able to dynamically expand and diminish a heap.

By callback routines in the BOS C-library, BGET announces that it needs more memory for the user heap. The C-library routine simply increases the program size and returns a pointer to the new memory chunk. Actually, it is the page fault handler that finishes the memory allocation by adding pages to the user process as it accesses the new heap memory.

2.4 User Stack

The initial user stack size is one page of memory. This is sufficient for starting a process, but as it runs more stack memory may be required. It is the page fault handler that adds new pages to the stack as needed.

3. Processes and Threads

This chapter explains the BOS process model, creation and termination of user processes, scheduling and semaphores.

3.1 Process Model

BOS distinguishes between two kinds of run-able instances: (1) kernel threads and (2) user processes. The next two subchapters describe the implementation of kernel threads and user processes. Presently, the Global Descriptor Table limits the number of concurrently running user processes to a maximum of fifteen.

3.1.1 Kernel Threads

Kernel threads have full access to all operating and system resources and are used to provide basic functionality needed by the operating itself. In BOS currently the following kernel processes exist:

- Main thread
- Idle thread: process to be scheduled if nothing else has to be done
- Reaper thread: responsible for cleanup after process termination
- IO threads: a means to communicate with IO devices (Floppy and IDE discs)

3.1.2 User Processes

BOS implements user processes as kernel threads with an attached user context. User processes run within an isolated environment and cannot directly access system resources in order to prevent the operating system and hardware from any kind of malicious or just buggy program code. User processes can access system resources via *System Calls* only and so the responsible kernel code can execute the requested operations in a secure way.

3.2 Process Creation

Beside reliability and speed, portability and a broad set of applications are important factors for the diffusion rate and acceptance of an operating system. BOS therefore supports the *Executable and Linking Format* (ELF) [6] as a highly portable object file format. BOS supports spawning processes only. There exists no implementation of well-known UNIX API calls like *fork* and *exec*.

3.2.1 Executable and Linking Format

ELF provides a set of standard binary interface definitions to run on 32-bit Intel Architecture based operating systems as well as other architectures. The ELF specification distinguishes three main ELF object file types:

- *Re-locatable File*: Used for linking with other object files.
- *Executable File*: Executable code and data
- *Shared Object File*: Code and data may be used in conjunction with executables by a dynamic linker. It may as well be used with other shared objects or re-locatable files to create another object file.

BOS is capable of parsing, loading and executing ELF formatted object files of the executable file type. BOS neither supports loading *Re-locatable Files*, nor loading *Shared Object Files*.

3.2.2 Starting Programs

In order to run an ELF executable the kernel accomplishes the following tasks:

- Load the program into the memory
- Parse the ELF executable data
- Resolve symbolic references and create a process image containing text, data and stack
- Spawn the created process image

The beginning of every ELF file occupies the ELF header that contains structural information about the file's layout. This header enables validation of the ELF format itself (magic, object file type, architecture). After parsing and validating the ELF header, the kernel loads and parses the file's program header table. This program header table contains all necessary information to load and parse the file's text and data segments. Once the kernel has parsed all segments, it creates a process image and spawns the process.

It is essential, that no user process page table has a reference to the first page of the user address space. This will help to trap null pointer references in user processes. Programs for BOS should therefore start at address 0x00001000. Higher addresses are possible, but BOS cannot use the resulting memory gap for other purposes.

At start time both, heap and stack of a user program have a size of 4KB each. Heap and stack can grow as needed as the process execution proceeds. The new process inherits the file descriptors of the console input and output from its parent process. A parent process can provide arbitrary file descriptors for a process it spawns like files, pipes, console input, and console output. When spawning the initial process, the kernel opens the console input and output explicitly for this process.

3.3 Process Termination

A process may exit either voluntarily by executing the *Exit* library call, or involuntarily because of a protection fault. In either case, the kernel frees the allocated resources of the process. In case of a protection fault, it is the fault handler that calls the *Exit* library call. The *Exit* kernel code moves the deceasing process from the run queue to the graveyard queue. It is the low prioritised *Reaper Thread* that periodically scans the graveyard queue for terminated processes and frees the allocated resources like semaphores, pipes, message queues, files, as well as the allocated memory pages.

3.4 Scheduler

Usually on common operating systems, several processes run concurrently. The scheduler is responsible to decide which processes should be assigned the CPU at which time. A so-called *Dispatcher* does the actual process replacement itself. In BOS dispatching and scheduling is not

separated. A detachment of the *Dispatcher* would be a further improvement but is not needed at the moment.

BOS offers two different pre-emptive scheduling algorithms, which can be changed at runtime using the SETSCHEDULINGPOLICY system call:

- Round Robin Scheduling
- Multilevel Feedback Scheduling

Each scheduling algorithm uses a defined *Quantum*. This is the maximum time ticks a process may use the CPU before the scheduler detaches it. A timer interrupt is used to determine when to schedule a new process. If a running process blocks or voluntarily gives up the CPU a new process is scheduled.

3.4.1 Round Robin Scheduling

Round Robin Scheduling uses one queue for all run able, i.e. not blocked, processes and simply schedules the next process from the queue. The actual implementation uses different priorities within this queue, but different priorities are only assigned to kernel processes. The Idle thread, for example, has the lowest priority. All user processes share the same priority and from a user land perspective scheduling appears to be simply *Round Robin*. The *Round Robin Scheduler* has the following characteristics:

- New processes are put at the end of the run queue.
- User processes are served first in first out.
- The idle process is only scheduled if there is no other process to schedule.
- The scheduling overhead is a function of the number of processes, i.e. O(n)

3.4.2 Multi Level Feedback Scheduling

Unlike *Round Robin Scheduling*, *Multilevel Feedback Scheduling* (MLF) uses a couple of queues, i.e. four in BOS, with different priorities. Processes from queues with higher priority are always scheduled first. The characteristics of MLF in BOS are:

- Four queues with priorities from zero to three.
- Processes from queues with highest priority are always scheduled first (queue 0 has highest priority).
- First Come First Served (FCFS) policy in all queues.
- New threads are always put on end of the queue with the highest priority.
- If a thread does not complete within its quantum, then the thread is moved to the end of the next queue with lower priority.

- Processes will be promoted to the next queue with higher priority if the process was blocked before.
- The idle thread must not leave the queue with the lowest priority.
- Within one queue scheduling is done in constant time, i.e. O(1).

3.4.3 Wait and Ready Queues

Wait and ready queues are used for processes trying to access the same resource concurrently. If a resource is in use by another process, the newly requesting process is moved to a wait queue and put to sleep. As soon as the resource becomes available the first process from the corresponding wait queue (FIFO) is detached from the queue, awakened and the desired resource assigned. In order to provide fast process determination BOS uses separated wait queues for each resource instead of using one long queue for all resources. The most important shared resources using wait queues in BOS are:

- CPU (4 wait queues for Multilevel Feedback Scheduling)
- Mutexes and Semaphores
- Message Queues
- Pipes
- File Systems (GosFS)
- Block Devices like IDE and Floppy discs

3.5 Semaphores

BOS provides semaphores to ensure consistency for shared resources. At the current state of development BOS can handle 20 semaphores whose names may be up to 20 characters long. Each process holds a list of open semaphores. Additionally, each semaphore has a reference counter with the number of registered processes for this semaphore. On process termination, all registered semaphores for this process are released, if not already done by the process itself.

4. Inter Process Communication

BOS provides the mechanisms message queue and pipe for inter process communication (IPC). The next two sub chapters will discuss the current implementation of these mechanisms.

4.1 Message Queues

BOS implements message queues not as recommended in the GeekOS documentation [2], rather UNIX like. In the GeekOS documentation message queues are very similar to pipes. BOS provides message queues for multiple readers and writes. In BOS a message is considered as a single entity, it can be sent or received as one entity. Once a message has been sent, it cannot be modified. If a

process provides a buffer too small for the message it receives, the message is truncated to the provided length and the cut off part of the message is lost.

As displayed in Figure 2, a message queue in BOS is a variable length *First In First Out* (FIFO) queue. When creating a message queue, the calling process can define its length. The maximum size of one message is limited to 4KB.

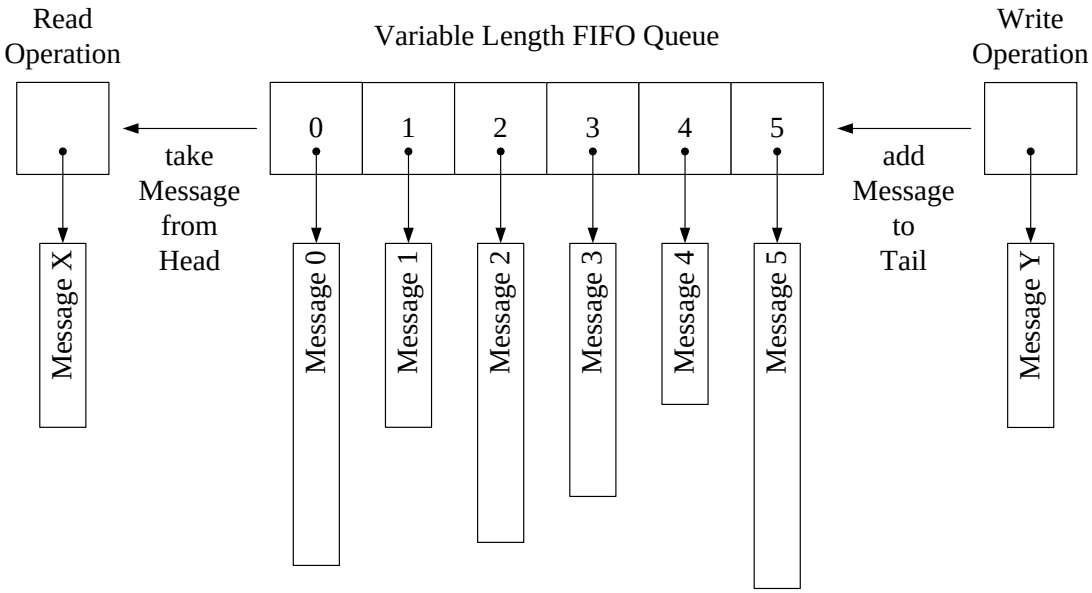

Figure 2 BOS Message Queue Implementation

After successful creation of a message queue, a process may send messages to the queue whether or not a receiving process reads messages from the queue. When the queue reaches its maximum length, the send call will block the emitting process until a reader takes a message from the queue. If a receiving process wants to read a message from an empty queue, the receive call will block the process until a message enters the queue. With multiple processes attached to a message queue, the kernel distributes the messages round robin to receiving processes.

A message queue can live without an attached process, as long as it has messages. In such a case the destruction of a message queue fails.

4.2 Pipes

BOS only provides anonymous and no named pipes. Pipes in BOS look like files, but act like an extension cord between two processes. One of the processes may write only to a pipe, while the peer process may read only from it. As shown in Figure 3, BOS pipes basically are ring buffers together with some management data. Each process owns a *File* structure that references a common *Pipe* structure in the kernel. This *Pipe* structure holds the information necessary to run the attached ring buffer.

The sending process can write data in any granularity it likes to the pipe until the ring buffer is full. Further writes calls will block the sending process. The kernel will transfer as much as possible of

the data to a receiving process as it reads. This behaviour reduces read calls on the pipe and ensures that the data leaves the ring buffer as fast as possible.

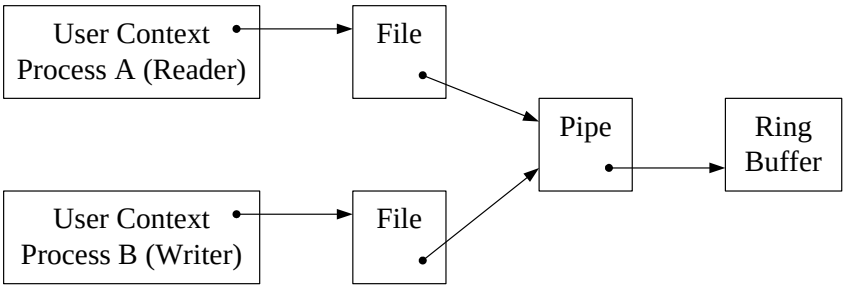

Figure 3 BOS Pipes Implementation

When the sending process closes its side of the pipe, the receiving process still can read the remaining data. If the pipe is empty, the receiving process gets an end of file error code from a reading call.

5. Input and Output

BOS employs a *Virtual File System* (VFS) as a device abstraction layer to handle both, block devices and character devices, as shown in Figure 4.

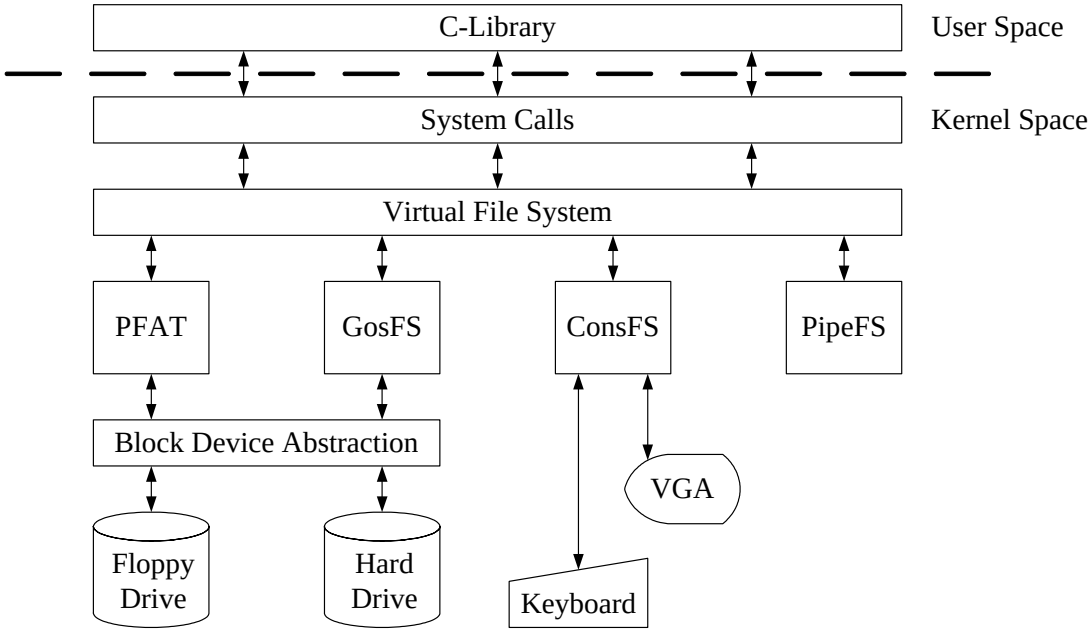

Figure 4 BOS Device Abstraction

The *Console File System* (ConsFS) abstraction offers the keyboard as an input device and the VGA graphics card as a character based output device. Open calls to the ConsFS return a file descriptor for either the keyboard, or the screen. This allows assigning the standard input and output of a process to the console. BOS has no abstraction for character devices yet, therefore the interrupt handler for the keyboard is part of ConsFS.

The *Pipe File System* (PipeFS) abstraction provides the pipe functionality by simulating file capabilities, as described above in chapter *4.2 Pipes*.

The *Pseudo File Allocation Table* (PFAT) and the *GeekOS File System* (GosFS) abstraction interact with block devices like floppy or IDE discs via the *Block Device Abstraction Layer*. This additional layer decouples a file system implementation from hardware issues, and allows adding new file systems as well as new hardware easily.

On boot time, the block device driver registers with the *Block Device Abstraction Layer* by providing a *virtual function table* of the device and a name. Further calls to this layer will use the name of the device for identification, e.g. to mount a file system. Once the driver is registered, the kernel spawns a thread for each device type to conduct the I/O requests of the devices. Each thread handles requests via its own queue and notifies the caller when a request completes.

6. File Systems

6.1 Virtual File System (VFS) Layer

The *Virtual File System* (VFS) layer abstracts file systems on a high level. Concrete implementations like PFAT and GosFS register their file system drivers at kernel boot time. Each user process initiated file system operation, through the C library, is trapped into the kernel, where the corresponding system call forwards the request to the VFS. The VFS then redirects the request to the corresponding file system implementation employing a so-called *virtual function tabl*e that references to the real implementation.

BOS provides two types of disk file systems:

- PFAT: read-only file system provided by GeekOS
- GosFS: read-write file system based on inodes

As the genuine GeekOS software provides PFAT, this paper discusses the GosFS implementation only.

6.2 The GosFS File System

The GosFS File System implementation deviates from the GeekOS recommendations and is more inspired by UNIX like file systems that use inodes. The main reason not to follow the GeekOS way is to have more flexibility and an even more universal abstraction to ease further enhancements like soft- and hard-links.

GosFS is a hierarchical file system supporting the following:

- Directories containing up to 240 files or sub-directories
- Files up to 4 GB using direct, indirect and double-indirect referenced blocks
- Hard-links by design but not implemented
- Soft-links by design but not implemented

- Long filename support
- Buffering using the GeekOS buffer-cache
- Concurrency support provided through a single mutex

Each directory allocates one inode, just as files do, except that the *size* indicated in the inode represents the number of directory entries and not the physical size. The content of a directory is stored in separate blocks, referenced by the inode's direct block pointers only. These blocks hold one or more directory entries to represent the directory's content. By utilising direct block pointers only, the number of directory entries is limited to 240. This is sufficient for this project, but should be expanded in future releases. Appendix A shows the structure of a directory entry in detail. This structure enables hard-links, because several different directory entries can reference to one particular inode in a file system.

The VFS provides a common file structure to processes for file handling. Within this file structure there is one pointer to reference to arbitrary data for a specific file system implementation. This arbitrary data is called *File Entry* in GosFS and holds the necessary information for the file system to work with a file. This abstraction makes it possible for different processes to work on the same file without interfering each other’s file position, because each process has its own file object. The arbitrary data structure within this file object then closes the gap to the actual physical file.

7. Demonstrations

This chapter briefly summarizes the demonstration held in class on January 25th. The main goal of this demonstration was to show BOS running on a bare metal COMPAQ DESKPRO XL 590 (90 MHz Pentium I) machine focusing on file system, pipes, message queues, scheduling and paging. For convenience the presented visuals below are Bochs [3] screen shots.

7.1 File System / Pipes

As already mentioned above, BOS implements the GosFS file system not as recommended in the GeekOS documentation, but rather UNIX like maintaining higher flexibility and abstraction. For example file or directory names are not stored within the inode itself, but in the corresponding directory entry. With the usage of pipes it is possible within BOS to redirect the standard output of one program to the standard input of another program; for example "`ls /c | more`" where `ls` is the sender and `more` the receiver program.

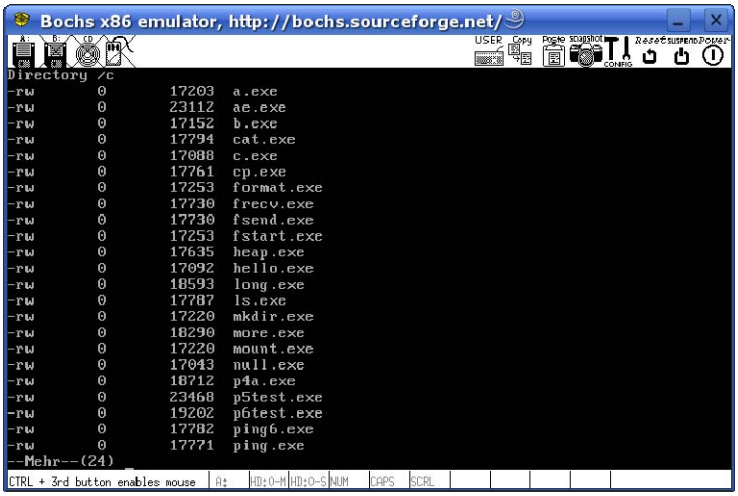

Figure 5 Output of the ls utility piped through the more program

As displayed in Figure 1 `more` shows only the top 24 lines of the `ls` output.

7.2 Message Queues

Message queues provide another way for inter-process communication in BOS. Here again the implementation is more UNIX like than recommended by the GeekOS documentation. A message queue in BOS can exist even without any program attached, as long as it holds messages.

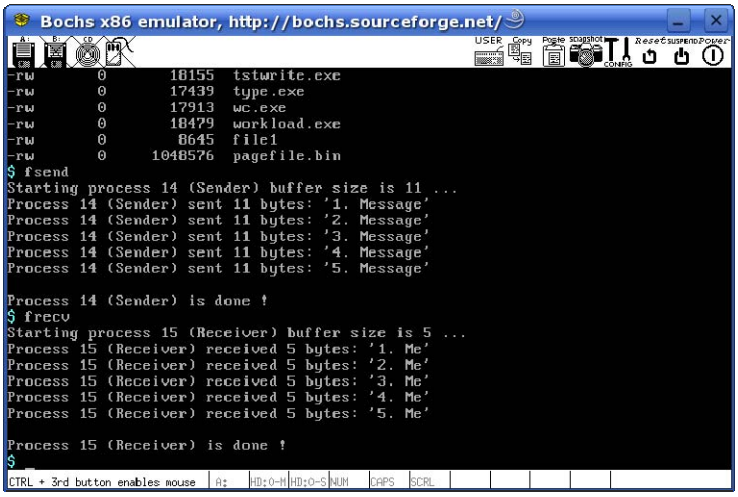

Figure 6 Message Queues Demonstration

The demonstration program "`fsend`" writes five messages to the message queue and terminates, as depicted in Figure 6. The "`frecv`" program receives these messages. For this demonstration the BOS team has chosen to endow the receiving program with a buffer too small to hold a whole message. What happens is that the receiving program fetches a message from the queue, but can only store a part of the message. The rest of the message is simply truncated.

7.3 Scheduling

BOS provides two different scheduling algorithms, which can be changed at runtime: Round Robin and Multilevel Feedback Scheduling. The demonstration program shows the different scheduling

attitudes by spawning seven processes. Three of these processes are CPU hogs printing the green letter C periodically and four of these processes are I/O intensive printing the red letter H. The I/O processes all try to write-access the same file; so these processes will block each other.

The `schedset` command allows switching between the two scheduling algorithms. The demonstration starts with Round Robin Scheduling. For the RR demonstration the commands are:

```
schedset rr 1
workload 1
workload
```

The first call of the `workload` utility loads the program to the main memory. This is necessary, because running this test without pre-loading the program would cause false results.

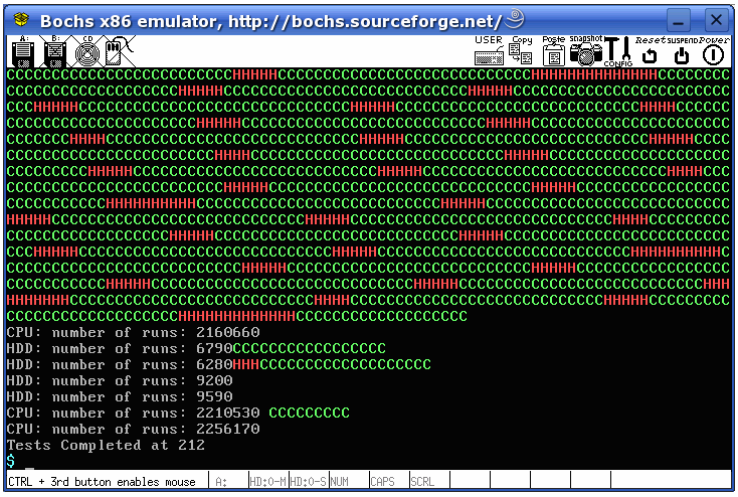

Figure 7 Round Robin Scheduling Demonstration

Figure 7 shows the output after the Round Robin Scheduling Demonstration. The average results are:

- CPU: ((2160660+2210530+2256170)/3)/212 = 10420 runs per cycle
- HDD: ((6790+6280+9200+9590)/4)/212 = 38 runs per cycle

After switching to Multilevel Feedback Scheduling, the I/O throughput increases significantly as visualized in Figure 8. Multilevel Feedback Scheduling promotes blocking processes to a queue with a higher priority, so the scheduler prefers the I/O processes. For the MLF demonstration the commands are:

```
schedset mlf 1
workload
```

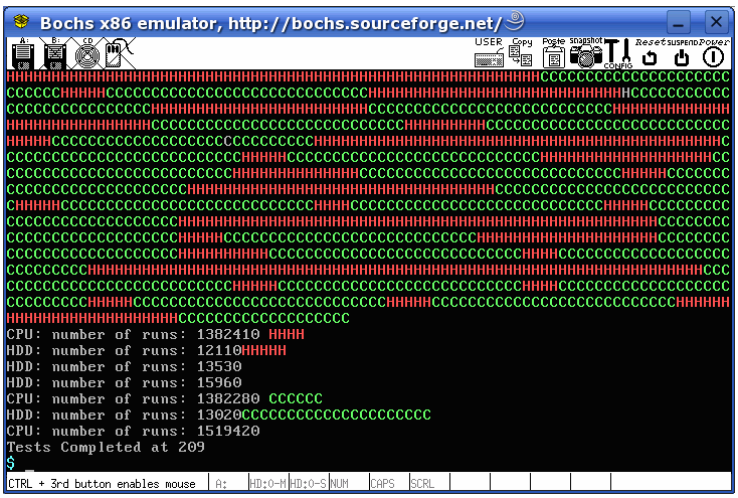

Figure 8 Multilevel Feedback Scheduling Demonstration

The average results using Multilevel Feedback Scheduling are:

- CPU: ((1382410+1382280+1519420)/3)/209 = 6832 runs per cycle
- HDD: ((12110+13530+15960+13020)/4)/209 = 65 runs per cycle

With Multilevel Feedback Scheduling I/O throughput can be increased by 71% whereas CPU throughput decreases by about 34% in this configuration.

7.4 Paging

Program `rec` recursively calls a subroutine that allocates a memory page on the call stack. Every time the subroutine allocates a page it writes a dot (´.´) to the screen and every 50 times it writes the decreasing counter to the screen. With 16MB main memory we start with a counter of 3750, i.e. we issue this command: `"rec 3750"`. While the program executes fast at high counter values, at about 251 the program gets significantly slow and disk I/O starts. At this time the page fault handler writes main memory to disk to free pages for the growing program.

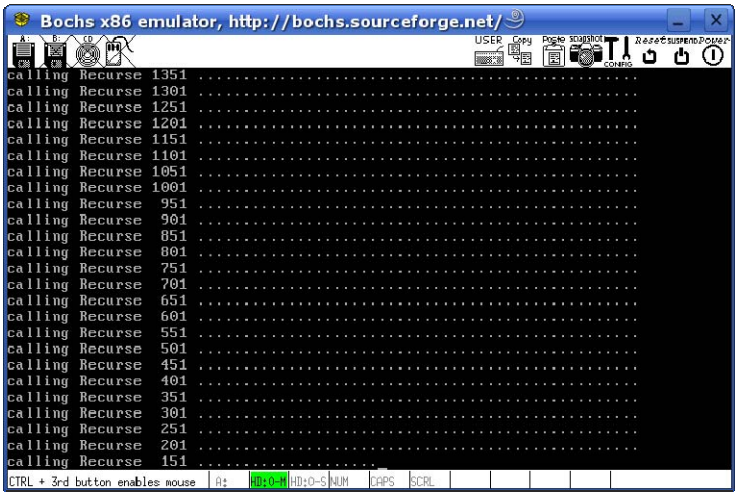

Figure 9 Paging Demonstration

Figure 9 shows the running program `rec` and an active hard drive `HD:0` (green), because the page fault handler writes to the paging file.

8. Unpacking and Building BOS

The BOS development environment consists mainly of following components:

- A Intel Pentium PC with SuSE Linux 10.0
- Bochs 2.3
- gcc 4.0.2
- nasm 0.98.38
- findutils 4.2.23
- coreutils 5.3.0
- binutils 2.16.91.0.2

After downloading the distribution package `bos-1.0-src.tar.gz` building the project works as follows.

1. change in a newly created folder and place the distribution package there

2. unpack the distribution package as follows
```
tar -xzvf bos-1.0-src.tar.gz
```

3. change to the build directory
```
cd bos-1.0/build
```

4. build BOS with the make utility
```
make depend
make all
```

5. start bochs after a successful build.
```
bochs –q
```

   There might be some adjustments necessary in file `.bochsrc`, namely the path names of the `vgaromimage` and the `romimage` parameters.

9. Conclusion

This work has presented an implementation of an operating system based on the GeekOS framework. This chapter concludes the paper by summarising the current situation, and providing an outlook to future enhancements.

Our goal was to implement a basic operating system that can boot on a real IA 32 PC hardware. Finally, the operating system runs on Intel 486 and Pentium I machines, but refuses to run on newer hardware. Real hardware as a development platform is very time consuming. Therefore the BOS development was done with the Bochs emulator, and only stable releases have been tested on bare metal. Real hardware showed that the initialisation of the keyboard is incomplete, because only a part of the keyboard sends useful key codes yet.

BOS includes concurrency support including two different scheduling algorithms, memory management with segmentation and paging, device abstraction via a virtual file system, as well as a block device abstraction layer, the BOS variant of the GosFS file system and inter process communication with pipes and message queues.

Future works on BOS could cover the following topics:

- Today BOS allows only one thread per user program. Future implementation should allow multiple kernel-scheduled threads.
- BOS should provide signals, at least the signals interrupt (SIGINT), hang-up (SIGHUP), terminate (SIGTERM) and kill (SIGKILL).
- Symbolic and hard links are not yet implemented, but should be provided by BOS.
- Serial I/O, TTY and pseudo TTY interfaces should be provided, as well as a device file system. The device file system could act like a real file system, and abstract the available hardware to virtual files. By opening such a virtual file, a user process could access devices like e.g. serial lines.
- Login and password authentication should be added also.
- The paging file has always to be `/c/pagefile.bin`, and must be present at boot time. Other systems allow booting the kernel without a paging file and assign the paging file via a command line utility. For BOS this would be beneficial in future releases.
- BOS should initialise the keyboard to enable all available keys.

Appendix A Implemented File System Functionality

This chapter summarises the implemented functionality of file systems in BOS.

Block Device Functions

Supported Functions	**Description**
Open	open a device
Close	close a device
GetNumBlocks	get number of blocks in a given device

Virtual File System Functions

VFS Object	**Supported Functions**	**Description**
File System	Format	create super block and root-directory entry
	Mount	check validity of the file system; prepare environment for file system usage
Mount Point	Open	open a file for usage
	Create Directory	create a new directory
	Open Directory	prepare directory for usage
	Stat	fetch metadata for a file or directory
	Sync	write dirty date to disc
	Delete	delete a file or directory
File	FStat	fetch metadata for given file
	Read	read a file's content
	Write	write to a file
	Seek	change position within file
	Close	close a file handle
	Clone	clone a file handle

Directory	FStat	fetches metadata for given directory
	Seek	change position within a directory
	Close	close a directory file handle
	ReadEntry	read a directory entry

Superblock

The super block maintains all necessary metadata to operate on the file system.

Field	**Length [bytes]**	**Description**
magic	4	magic number to identify the GOSFS file system
supersize	4	size of the super block in bytes
size	4	size of whole file system in blocks
inodes	112	array of inodes
bitSet[]	depends on device size	Set of bits to mark free and used blocks of the file system.

Instance

On mount of the file system an instance object is created and stored within the mount point to be able to handle all further file system requests.

Field	**Length [bytes]**	**Description**
lock	16	mutex to lock the file system in order to provide concurrency
buffercache	4	pointer to a buffer to provide buffered IO requests
superblock	depends on device size	the file system’s super-block

Inode

Inodes are used to maintain the essential metadata for files and directories; except the name of the file or directory.

Field	**Length [bytes]**	**Description**
inode	4	inode number
size	4	size of file in bytes for files or number of directory-entries for directories
link_count	4	references to this inode (hard-links)
blocks_used	4	number of blocks allocated for file or directory
flags	4	flag used to indicate directories, files, SUID
time_access		last time file was accessed (not used)
time_modified	4	last time file was modified (not used
time_inode	4	last time inode was modified (not used)
time_inode	4	last time inode was modified (not used)
blocklist	4	pointers to direct, indirect and double-indirect blocks
acl	40	access control list (not used)

Directory Entry

Field	**Length [bytes]**	**Description**
type	4	type of the directory entry • THIS ".": special type referencing the directory itself • PARENT "..": special type referencing the parent directory • REGULAR: file or directory entry • FREE: directory entry not used
inode	4	referenced inode number
filename	128	name of file or directory (inode) in this directory

File Entry

Field	Length [bytes]	Description
inode	4	pointer to the referenced inode
instance	4	pointer to the file system entry instance we are working on
references	4	number of file descriptors referencing this entry

Appendix B Distribution Package

This chapter briefly describes the directory structure of the distribution package

File / Folder	Content
COPYING	contains the copyright holder and rules about copying this software
LICENSE-klibc	additional license information
README	contains a description how to build and run BOS
build	contains the Makefile to build BOS and all build results
doc	contains the BOS technical report and the BOS presentation slides
include	contains the include files of the kernel an the C library
scripts	contains the scripts GeekOS provides
src	contains the source code of the kernel, the C library, the tools and the user programs

Appendix C The C Application Programming Interface for User Programs

This chapter presents an overview of the major BOS system calls available to user processes.

C.1 File and File System calls

To access the following system calls, a program must include:

#include <fileio.h>

Format a given device with a specified file system. Devices can be *ide0* and *ide1* and file system types can be *pfat* of *gosfs*.

int Format(const char *dev, const char *fstype);

Mount a device *dev* with file system *fstype* to a given mount point named in *prefix*.

int Mount(const char *dev, const char *prefix, const char *fstype);

Synchronise all file systems to disk.

int Sync(void);

Retrieve the status data of a specified or already open file.

int Stat(const char *path, struct VFS_File_Stat *stat);

int FStat(int fd, struct VFS_File_Stat *stat);

Delete a file or directory. A directory must be empty when calling this subroutine to succeed.

int Delete(const char *path);

Open a file

int Open(const char *path, int mode);

Close a file or directory

int Close(int fd);

Read from a previously opened file into a buffer with specified length.

int Read(int fd, void *buf, unsigned long len);

Write to a previously opened file from a buffer with specified length.

int Write(int fd, const void *buf, unsigned long len);

Set read and write pointer of a file.

int Seek(int fd, int pos);

Create a directory.

int Create_Directory(const char *path);

Open a directory.

int Open_Directory(const char *path);

Read a directory entry. Subsequent calls will return all entries of a directory, one at a call.

int Read_Entry(int fd, struct VFS_Dir_Entry *dirEntry);

Create a pipe and return the reader and writer file descriptors.

int Create_Pipe(int *readfd, int *writefd);

C.2 Kernel related Calls

To access the following system calls, a program must include:

#include <kernel.h>

Select a particular paging algorithm.

int Select_Paging_Algorithm (int alg);

C.3 Heap related Calls

To access the following system calls, a program must include:

#include <malloc.h>

Allocate dynamic memory

void *Malloc(size_t size);

Free previously allocated dynamic memory

void Free(void* buf);

C.4 Message Queue Calls

To access the following system calls, a program must include:

#include <mq.h>

Create a message queue

int Message_Queue_Create(const char *name, ulong_t queue_size);

Destroy a message queue

int Message_Queue_Destroy(int mqid);

Send a message to a previously opened message queue

int Message_Queue_Send(int mqid, void * buffer, ulong_t message_size);

Read a message from a previously opened message queue

int Message_Queue_Receive(int mqid, void * buffer, ulong_t message_size);

C.5 Process related Calls

To access the following system calls, a program must include:

#include <process.h>

Terminate the current process

int Exit(int exitCode);

Create a new process

int Spawn_Program(const char *program, const char* command, int stdinFd, int stdoutFd);

Create a new process using a given executable search path

int Spawn_With_Path(const char *program, const char *command, int stdinFd, int stdoutFd, const char *path);

Wait for termination of a child process

int Wait(int pid);

Get the identification of the current process

int Get_PID(void);

C.6 Scheduling related Calls

To access the following system calls, a program must include:

#include <sched.h>

Set the scheduling policy

int Set_Scheduling_Policy(int policy, int quantum);

Get the time of day in timer ticks

int Get_Time_Of_Day(void);

C.7 Semaphore related Calls

To access the following system calls, a program must include:

#include <sema.h>

Create a semaphore

int Create_Semaphore(const char *name, int ival);

Enter a critical section

int P(int sem);

Leave a critical section

int V(int sem);

Destroy a semaphore

int Destroy_Semaphore(int sem);

References

[1] B. Danninger, C. Krainer (2007) *Basic Operating System*. http://www.cs.uni-salzburg.at/~ck/wiki/index.php?n=OS-Winter-2006.BOS (8 February 2007)

[2] D. Hovemeyer (2006) *GeekOS: a tiny operating system kernel for x86 PCs*. http://geekos.sourceforge.net/ (8 February 2007)

[3] T. R. Butler, et al. (2006) *Bochs, the cross platform IA-32 emulator*. http://bochs.sourceforge.net (8 February 2007)

[4] R.W. Carr, J.L. Hennessy (1981) WSClock – A Simple and Effective Algorithm for Virtual Memory Management. *Proceedings Eight Symposium on Operating Systems Principles*, ACM, pp. 87-95.

[5] J. Walker (1996) *The BGET Memory Allocator*. http://www.fourmilab.ch/bget (8 February 2007)

[6] Tools Interface Standards Committee (1995) *Executable and Linkable Format (ELF) Specification*.