Topsy i386
A Teachable Operating System.
The Port to the ia32 Architecture

Semester Thesis of Lukas Ruf

April 1998 – July 1998

Computer Engineering and Networks Laboratory, ETH Zurich
Supervisor: George Fankhauser
Professor: Bernhard Plattner

Abstract

This report provides an overview of the semester project done by Lukas Ruf during the summer semester 1998. Theme of the semester project was the first port of Topsy and conjointly the proof of its portability. Topsy, short for "A Teachable Operating System", is a small multithreaded operating system developed and used for teaching purposes at the Swiss Federal Institute of Technology in Zurich. Topsy was principally developed by George Fankhauser. Topsy is a layered operating system, i.e. all hardware dependencies are encapsulated. So porting Topsy requires the development of the hardware dependent part providing the same functionality on the target system.

The ia32 architecture as found in most commonly used personal computers was the target platform of this port from MIPS architecture. Topsy i386 is a standalone operating system running in full 32bit protected mode. It provides a fully compatible system interface to the source code of MIPS-Topsy v1.0.

For a better understanding of this report a basic knowledge of the ia32 architecture is helpful. This can be found in the books noted in the bibliography in Appendix D, "The i386 Port".

This report was intentionally written on a high level of abstraction as most of this work lays in the coding of the hardware dependent part of the operating system itself. However, a discussion of portability as notes on the applied memory protection scheme and results can be found in this report. Most of the documentation is written in Appendices C, "A Guide to Port Topsy", and D, "The i386 Port", of the manual.

Additionally this report includes Appendix C, "A Guide to Port Topsy", Appendix D, "The i386 Port" and Appendix E, "Given Problem" of the manual for a compact reading.

This semester project has fully reached its goals: A running Topsy i386 together with a proof of portability are available.

Abstract

Dieser Bericht gibt einen Ueberblick zur Semesterarbeit von Lukas Ruf waehrend des Sommersemesters 1998. Thema dieser Arbeit war die erste Portierung von Topsy auf eine andere Hardware-Plattform mit einer gleichzeitigen Verifikation seiner Portierbarkei. Topsy, Kurzform fuer "A Teachable Operating System", ist ein "kleines aber feines" multithreaded Betriebssystem, das an der Eidgenoessischen Technischen Hochschule in Zuerich (ETHZ) hauptsaechlich von George Fankhauser entwickelt wurde und im Praktikum zur viertsemestrigen Vorlesung Technische Informatik II eingesetzt wird. Das Design von Topsy ist schichtenorientiert, d.h. die Hardware-Abhaengigkeiten sind getrennt vom restlichen Betriebssystem. Dieses Schichtenmodell ermoeglicht eine einfache Portierbarkeit. Eines der Ziele bei der Entwicklung war die einfache Portierbarkeit von Topsy.

Um Topsy auf eine neue Hardware-Plattform zu portieren, muss lediglich der Hardware-abhaengige Teil an die Gegebenheiten der Zielplattform angepasst werden. Hauptbedingung ist, dass die Schnittstelle, wie sie beim originalen Topsy vorgegeben wurde, moeglichst eingehalten wird.

Die Zielplattform bei dieser Portierung war die ia32-Architektur, wie sie heute in den meisten eingesetzten Personal Computer verwendet wird. Urspruenglich wurde Topsy fuer eine Praktikumsumgebung mit einer MIPS-Architektur entwickelt. Topsy i386 ist ein eigenstaendiges Betriebssystem, das im 32bit Protected Mode der ia32-Architektur laeuft. Die vorliegende Version als Resultat der Semesterarbeit stellt eine zur Version 1.0 des MIPS-Topsy's voll kompatible System-Schnittstelle zur Verfuegung.

Fuer das Verstaendnis dieser Dokumentation, d.h. dieses Berichtes und der beiden Appendizes C und D des Topsy Benuetzer-Handbuches, wird eine Grundkenntnis der ia32-Architektur vorausgesetzt, die den Buechern des Literaturanhanges entnommen werden kann.

Dieser Bericht wurde absichtlich auf einer hohen Abstraktionsstufe geschrieben, da der Hauptteil dieser Semesterarbeit in der realisierten Portierung des Betriebssystems, d.h. im Quellcode selbst liegt. Dennoch wird in diesem Bericht eine Eroerterung der Portabilitaetsverifikation als auch eine Darstellung des angewendeten Schutzschemas gegeben.

Die Dokumentation der vorliegenden Portierung wurde zur Hauptsache in den beiden Appendizes C, "A Guide to Port Topsy", und D, "The i386 Port", des Topsy Benuetzer-Handbuches geschrieben. Sie wurden fuer die Abgabe der Semesterarbeit in diesen Bericht eingebunden.

Das Ziel dieser Semesterarbeit wurde erreicht: Das Resultat der Arbeit ist ein lauffaehiges Topsy i386 mit einer Verifikation der Portierbarkeit des Systems.

Contents

Chapter 1

Introduction

The 4th term students of the department of electrical engineering at Swiss Federal Institute of Technology in Zurich (ETHZ) are taught modern programming skills like multi threading, multi tasking and simple operating system development. Practical exercises are performed using a small, simple and multi threaded operating system called Topsy, "A Teachable Operating System". This operating system was developed for the Practice IDT-MIPS Board as found at the ETHZ, connected to Sun Workstations, running Solaris. These boards are connected via a serial line to the workstation. Topsy is downloaded to the board and booted there. The user interface is an xterm-shell on the Sun, cooperating with the debugger required to start the operating system.

As normally students do own neither a Sun Workstation nor are they able to use such a MIPS Board at home. Personal computers on an Intel processor basis are wide spread, and most of the students own one. For examination preparation students are to learn the multi threaded programming by practice. To provide the required platform on standard personal computer this semester project was started.

1.1 Semester Project Overview

This semester project was intended to port the Topsy Hardware Abstraction Layer from the MIPS architecture to the commonly used ia32 architecture, i.e. 32bit processors as nowadays found in personal computers (80386, 80486, Pentium, Pentium Pro, Pentium II, and compatible). The operating system source code to port was the first release of Topsy, version 1.0.

The design of Topsy is clearly layered, i.e. the functionality of the operating system is divided in a hardware dependent layer (the Hardware Abstraction Layer: HAL), a hardware independent kernel, and user layer.

Providing Topsy on other platforms than the original MIPS board requires porting the hardware abstraction layer to the new platforms; kernel and user

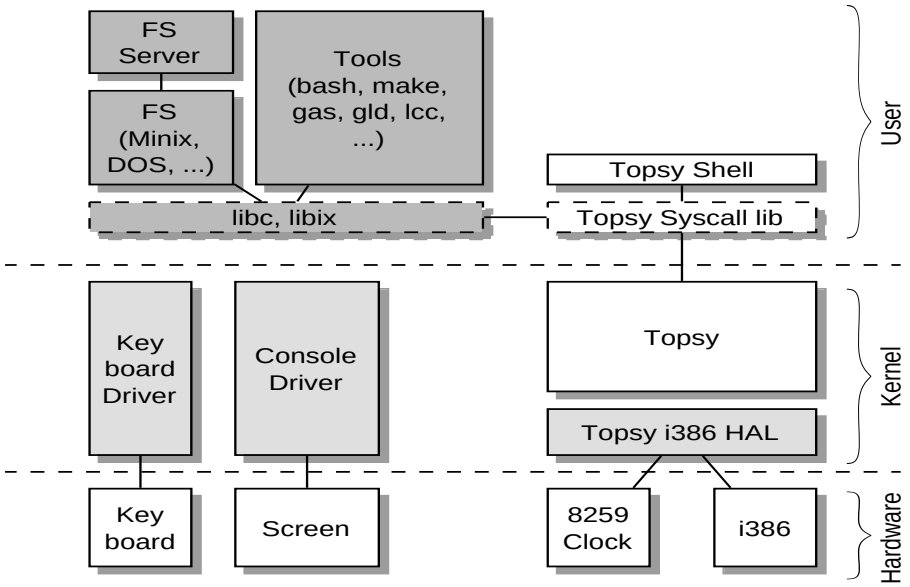

Figure 1.1: Layered Model of Topsy

layer do not require changes. As noted above the xterm-shell running Topsy on Sun/MIPS provides keyboard and console handling at lowest level. Topsy on other platforms require the development of them. A console driver, i.e. keyboard handling and screen output, was developed as part of the standalone operating system hardware dependencies. Provided with this release is a keyboard mapping near to the Swiss German VSM keyboard layout.

As this semester project consists of the first port, the proof of portability had to be done, too (see below Section "Proof of Portability").

Topsy i386 as released with this documentation boots from a standard high density floppy disk. No drive access in protected mode is available at this time, so all data has to be loaded during boot up: in real mode as the BIOS interrupts of the personal computer do their job only in real mode.

For a detailed explanation of ported functions, implementation requirements and implementation decisions see Appendix C, "A Guide to Port Topsy", Section "Formal HAL definition", and Appendix D, "The i386 Port".

1.2 Acknowledgements

Acknowledgements of this semester project go to the project supervisor George Fankhauser for accepting and supporting this theme as for encouraging discussions, too. Further acknowledgements are returned to the Computer Engineering and Networks Laboratory at ETHZ for the enabling of this semester project itself.

Chapter 2

Problem

The original problem can be found in Appendix E. This chapter provides a short summary of problems given in this semester project.

2.1 Problem Summary

Summarizing the given problem the following points have to be considered:

- Port of the hardware abstraction layer from MIPS to ia32 architecture
- Proof of portability of Topsy
- Development of a keyboard driver
- Development of a screen driver
- Development of a general purpose dynamical driver loading mechanism, i.e. a load procedure for drivers not statically linked to the kernel image
- Summarizing report
- Documentation of functional system

Chapter 3

Development Environment

3.1 Development Platform

Development of Topsy i386 had to be done using freely available development tools. Not to waste time with configuring and installing the development operating system stable Linux version 2.0 was chosen. Another point of interest was the availability of the GNU Tools as Topsy for MIPS was developed using them.

Users of Topsy i386 should have installed GNU Tools, i.e. the compiler, linker and object copy. Any additional tools required for the creation of bootable program images are provided together with the source code of Topsy i386.

3.2 Development Hardware

Topsy i386 was developed on a personal computer with a fast harddisk and a fast processor. The personal computer on a ia32 platform was chosen as development platform as on the one hand the process could be performed without using cross development and on the other hand the personal computer (as the target system) was available at home.

When developing or porting an operating system, continuous testing is required after reaching a certain degree of development progress. Debugging an operating system at initial development is not possible using a professional available debugger – this would require a compatible and stable running operating system. Only tests "by hand" i.e. following formerly specified statements, displaying register and stack contents are available.

To provide a fast test equipment a personal computer with only minimal equipment was installed: a floppy disk drive, a VGA (Video Graphic Adapter) controller, a modern processor and a minimal size of RAM were built in the test workstation.

Chapter 4

Results

Referring to chapter 2 all goals except the driver loading were fully reached, . Currently the drivers are still statically linked to the kernel image. The user block is split off from the kernel block, providing an own C-Startup function. No protected mode disk interface is provided yet, i.e. the disks can only be accessed using the BIOS functionality provided by a personal computer (BIOS operates only in real mode). So the complete code must be loaded to RAM during startup, i.e before the switch to protected mode. Further on, the use of BIOS restricts the size of executable code as in real mode standard personal computer provide only 640KB of RAM. For a detailed discussion on starting Topsy i386, please refer to Appendix D, "The i386 Port".

A general purpose driver loader would exceed the scope of time available for a one man semester project. The driver handling mechanism would have had to be completely redesigned; this could be part of another semester project. Nevertheless a short discussion on design alternatives for a driver loading mechanism can be found in the last section of this report, Driver Loader 4.3.1 on page 9.

4.1 Proof of Portability

Proofing an operating system on portability aspects means detecting functional definitions that are not portable. Inportabilities must be located in the kernel layer since only the hardware abstraction layer depends on the architecture used.

Topsy as a well designed layered operating system was found really portable. Only some implementation details made Topsy v1.0 inportable. With little effort they were fixed:

- **Threads/TMIPC.c/msgDispatcher()**
  an argument branchDelayBit was specified. On other platforms, like the ia32, this bit may not exist – and so, can not be handled correctly. The branchDelayBit flag makes the msgDispatcher() kill the actual thread if it equals to a MIPS dependent state of a so called "Branch Delay slot".

Functionality of the Topsy kernel can not be provided on other hardware architectures than MIPS if this flag remains in the kernel layer.

- **Threads/TMThread.c/threadBuild()**
  was developed in face of the MIPS architecture and handled only the MIPS dependencies correctly. Other architectures like the ia32 may require additional settings (e.g. segment selectors in ia32) for building an executable thread. This problem could only be solved implementing a function to set the required hardware dependencies for thread execution (so called TMHAl.c/tmSetMachineDependencies()). It might be possible that other ports require an additional setting of hardware dependencies before constructing the thread context; the currently provided function is located after the original thread construction.
  Further on the use of **tmSetReturnAddress()** was not portable: the address should not be written as a literal but with a defined constant. Some other architectures may need another stack layout than the MIPS board.

- **IO/IODevice.c/ioCheckBufferAddress()**
  caused some real trouble when preparing the demonstration of Topsy i386 at ETHZ. In the ia32 world protection is applied by defining different segments which all start at offset zero (refer to Appendix D, section "Basic Initializations").
  Implementing a memory manager that supports memory mapping by using the MMU (Memory Management Unit) capabilities of the i386 would require a function that verifies the buffer addresses.

  Actually Topsy i386 does not support virtually mapped memory management: programming the memory management unit of an ia32 architecture to support virtually mapped memory management would require an effort not possible for this semester project.
  The ioCheckBufferAddress() is not required for Topsy i386 as every message passing from user to kernel space requires an address translation (see Appendix D, "The i386 Port", section "Basic Initializations")

In general the IO Handling is at kernel level and needs a new structure as it is not possible to provide the equal functionality on every platform, e.g. the FPGA driver. Another problem occurred when implementing the drivers itself: In MIPS-Topsy they are hardcoded into IO/IOMain.c. It could be possible to name some standard interface drivers, for example serial, console, keyboard, disk, parallel, network, etc. If a new dynamical driver loading mechanism is implemented, the driver structure must be redesigned.

4.2 Differences

The implementation of Topsy on the PC could be made without really hard differences to the MIPS-Topsy, except for the second timer interrupt and the drivers. The kernel itself does not make use of the second timer channel but provides an initializing of it.

In a PC a timer controller compatible to the Intel 8253 is included as found in the MIPS-board too. Problems raised when trying to provide a second independent timer interrupt to the kernel: the PC makes a hardwired use of all three timer channels:

1. Channel 0 is available for system programmers
2. Channel 1 is used for memory refreshments
3. Channel 2 is hardwired to the PC speaker for generating sounds

The drivers provided by MIPS-Topsy could be adapted in sense of equal functionality. In Topsy i386 only a TTY driver is implemented for handling console input and output, i.e. keyboard handling and graphic adapter controlling in text mode.

4.3 Further Work

Topsy i386 v1.0 is a running standalone operating system providing full functionality compatible to the MIPS-Topsy v1.0. Extending Topsy i386 is always possible, e.g.

- **Memory Management**
  Removing the statically defined maximal available memory size for user programs and implementing the effective available physical RAM size could provide space for more user and/or kernel processes. At initial booting of Topsy i386, the required information is collected, refer to Appendix D, section "Basic Initializations".
- **Driver Loader**
  Loading of drivers provided in a separate module not necessarily statically linked to the kernel at build time. For an initial idea see section 4.3.1.
- **Virtual File System**
  Managing a file system not dependent of the physical medium: providing a RAM disk (no disk access required), accessing a physical disk, see next.
- **Disk Access**
  Accessing a disk at kernel run time without the need of real mode operation, i.e. a driver running at kernel level with direct disk access.

- **Network**
  A driver providing direct network card access on basis of Ethernet or Token Ring could on the one hand show the IP-Stack real world functionality implemented by the semester project of David Schweikert and on the other hand connect Topsy to the outer world.

4.3.1 Driver Loader

This section describes one possible driver loader mechanism.

The dynamic driver loader could be implemented managing the drivers as separate programs like the kernel or user. Every driver runs within his own driver space. The required commands are exchanged via the message passing scheme as used for kernel-user message passing. No extensions regarding program relocation must be made for loading separate drivers as would have to be done if the drivers where moved into the kernel space. The disadvantage with this option can be found as every driver must implement a simple message handler. Further the driver manager linked to the kernel must implement an own memory management system.

A Guide to Port Topsy

Semester Thesis of Lukas Ruf

April 1998 – July 1998
Computer Engineering and Networks Laboratory, ETH Zurich
Supervisor: George Fankhauser
Professor: Bernhard Plattner

Abstract

Appendix C, "A Guide to Port Topsy", provides an abstract overview of the requirements for doing a port of Topsy to another platform. With respect to the i386 port done by Lukas Ruf, every HAL Interface function is listed and shortly commented. A short note how the function is implemented in Topsy i386 completes the formal HAL definition.

A section with difficulties found while the i386 port was programmed should cover possible problems when a new port is started.

Providing hints for a new port strongly depends on the target architecture so only a short description on how the i386 port was done could be implemented there.

Appendix C

A Guide to Port Topsy

C.1 Formal HAL Definition

This chapter provides a summarizing overview of the formal HAL definitions. The interface is specified by noting the function headers. The list of all functions also contains a short description of the appropriate functionality as a short note on implementation applied in Topsy i386 are given. As helpful the notes are listed by catchwords. Detailed implementation information for Topsy i386 can be found in Appendix D, "The i386 Port".

C.1.1 Input/Output

The HAL interface is only used for kernel purposes. Input and output of the user programs are handled by the console driver.

1. **void ioConsoleInit();**
   Initialization of the console. If output is generated via a communication line, this function is used. In Topsy i386 there is no use of it as an attached graphic adapter is assumed.

2. **void ioConsolePutChar(char);**
   Output of a single character. Needs being implemented for low level debug output. Topsy i386 translates this to a function call to kprintc() (IO/ia32/Video.c).

3. **Input, Output for User Programs:**
   Character input and output are handled on a driver basis, i.e. the drivers have to be implemented. In Topsy i386 the driver TTY (IO/ia32/Drivers/TTY.c) handles simultaneously character input and output as needed by ioRead() and ioWrite() (both located in Topsy/syscall.c).

C.1.2 Thread Management

Most of those functions are only called at thread generation or system initialization.

1. **Error setClockValue(ClockId , int , ClockMode);**
   Handles initialization of the clock chip. For preemptive multi threading needs being implemented. Topsy i386 provides only one single timer channel as both others are hardwired to memory refreshment cycles and speaker frequency generating.

2. **void tmResetClockInterrupt(ClockId);**
   Resets the raised clock interrupt. In Topsy i386 this function is not used, i.e. the function body remained empty as this functionality is handled by the Interrupt Handler itself as all raised interrupts need an acknowledgement, otherwise the interrupt controller stops recognizing new interrupts on a less privileged level.

3. **void tmInstallErrorHandlers(void);**
   Installs the default error handlers as the default system call gates too. The default error handlers in fact do only provide an error message and then kill the faulting thread. Topsy i386 provided a slight different default handling, see section D.3.2.

4. **void saveContext(ProcContextPtr);**
   Stores the current processor context in the provided address space. This function call is implied by the general exception handler in MIPS-Topsy at assembler level. Topsy i386 does not make use of it. It saves the status to the stack at transition from assembler to C.

5. **void restoreContext(ProcContextPtr);**
   Restores a new processor context. So, a thread and a process switch are implied by calling this function with a newly scheduled processor context. Topsy i386 must take care of protection level switching as the ia32 architecture includes a multiprocessing and not a multi threading support. A really elegant solution can be found in Threads/ia32/TMHalAsm.S.

6. **void tmSetReturnValue(ProcContextPtr , Register);**
   Sets the thread return value. When a kernel or driver function is called by a tmMsgSend, the return value to the calling thread must be noted somewhere in the calling thread's context. Topsy i386 must store the return value in the thread's stack environment, a function returns the value in a general purpose register (pair) on the ia32 architecture which are restored by a restoreContext() function call.

7. **void tmSetStackPointer(ProcContextPtr , Register);**
   Adjusts the stack pointer of a newly created thread. Topsy i386 must set a location in the thread's stack.

8. **void tmSetReturnAddress(ProcContextPtr , Register);**
   Sets the return address of a thread. When a user thread exits without calling tmExit() the address formerly set by this function is used to call the exit function. Topsy sets this value in the thread's stack.

9. **void tmSetProgramCounter(ProcContextPtr , Register);**
   Sets the initial program counter for a thread start. Topsy i386 sets this value in the thread's stack.

10. **void tmSetStatusRegister(ProcContextPtr , Register);**
    Handles modifications of the processor status flags, e.g. interrupt enabling/disabling on thread start/return. Topsy i386 sets this value in the thread's stack.

11. **void tmSetFramePointer(ProcContextPtr , Register);**
    Sets the initial frame pointer of a thread. Normally this value is initialized to zero. Topsy i386 sets this value in the thread's stack.

12. **void tmSetArgument0(ProcContextPtr , Register);**
    Provides the argument 0 – the first argument – to the started thread. Topsy i386 sets this value in the thread's stack.

13. **void tmSetArgument1(ProcContextPtr , Register);**
    Analogous to tmSetArgument0().

14. **void enableInterruptInContext(InterruptId, ProcContextPtr);**
    Enables specified interrupt lines. Not actually used for controlling single interrupt enabling. Topsy i386 can not enable a single interrupt line per thread context; the interrupt controller acts globally. Topsy i386 only enables or disables all interrupts by setting a processor flag in the thread's context. For proper preemptive operation this flag has to be set always when starting a thread.

15. **void disableInterruptInContext(InterruptId,ProcContextPtr);**
    Analogous to enableInterruptInContext().

16. **void enableAllInterruptsInContext(ProcContextPtr);**
    The functionality of enableAllInterruptsInContect() is restricted to the one noted in enableInterruptInContext(), see above.

17. **void tmInstallExceptionCode();**
    Initializes the default exception handling functionality. Topsy i386 installs

the clock interrupt handler per default in this function. The timer handler at HAL level was found to be a general purpose exception handler more than an error handler.

18. **void syscallExceptionHandler(ThreadId);**
    Default syscall exception handler. Topsy i386 implemented a generalized exception/interrupt handler.

19. **void hwExceptionHandler();**
    Analogous to syscallExceptionHandler() for hardware interrupt handling.

20. **void automaticThreadExit();**
    Address of the automatic thread exit code. This code is copied on to the thread's stack for returning properly when a thread finished operation.

21. **void endAutomaticThreadExit();**
    This function header is only a label, i.e. an address, and notes only the end of the formerly listed function to the C code. In assembler it is coded as a label.

22. **void UTLBMissHandler();**
    User translation lookaside buffer error handler. Called when the processor detects an invalid page flag. Not used in Topsy i386 v1.0 as only direct mapped memory handling is supported.

23. **void endUTLBMissHandler();**
    This function header is only a label, i.e. an address, and notes only the end of the formerly listed function to the C code. In assembler it is coded as a label.

24. **void generalExceptionHandler();**
    Assembler level general exception handler. Called for a generalized exception handling. Topsy i386 implemented this function on C level.

25. **void endGeneralExceptionHandler();**
    This function header is only a label, i.e. an address, and notes only the end of the formerly listed function to the C code. In assembler it is coded as a label.

C.1.3 System Calls

1. **SyscallError tmMsgSend(ThreadId , Message *);**
    Assembler level message sending routine. This is part one of the main functionality, i.e. the message passing. In Topsy i386 this function is implemented with passing the values by registers as done in MIPS-Topsy.

2. **SyscallError tmMsgRecv(ThreadId* , MessageId , Message* , int);**
   Part two of the main functionality. Analogous to tmMsgSend().

C.1.4 Memory Management

1. **void mmInstallErrorHandlers(void);**
   Installation of the memory management fault exception handlers. Not used in Topsy i386 as fault exception handling is part of the generalized exception/interrupt handler, see note on error handling below.

2. **Error mmInitMemoryMapping(Address codeAddr, unsigned long int , Address , unsigned long int , Address);**
   Initialization of the memory mapping functionality. If virtual memory mapping was used, this function should install the translation buffers. In Topsy i386 this function only copies the user program into the user space.

3. **Error mmMapPages(Page , Page , PageStatus);**
   Would map a page. Not used in Topsy i386.

4. **Error mmUnmapPages(Page , Page);**
   Would unmap a page. Not used in Topsy i386.

5. **Error mmMovePage(Page , Page);**
   Movement of a page. Topsy i386 v1.0 does only copy a page; it is a direct memory mapping system.

6. **Error mmProtectPage(Page , ProtectionMode);**
   A protection flag could be set to a single page. Not used in Topsy i386 as this would require the virtual memory mapping management implemented.

7. **Error mmAddressSpaceRange(AddressSpace , Address* , unsigned long int*);**
   Returns the starting address of a requested address space (USER or KERNEL) as the managed size of this range. Topsy i386 provides a compatible functionality.

8. **void setTLBEntry(Register , Register , Register);**
   Would set a translation lookaside buffer entry if this function would be required by the processor. The ia32 architecture handles the TLB management directly in its MMU. So not used in Topsy i386.

C.2 Known Difficulties

When the Topsy i386 port was started a deeper understanding of the original Topsy implementation had to be worked out first as porting a HAL in a formal way is not sufficient: the operating system must be fully understood. The MIPS Topsy includes a lot of message passing functionality in assembler, so, even the MIPS assembler notation must be known.

C.2.1 Points of Problems

A list of found problems while porting Topsy to the ia32 architecture is provided in a catchword style where applicable and helpful. This list is only in respect to the i386 port done – it does neither resemble to be complete for other ports nor for Topsy i386. For detailed information on implementation specialties the source code provides the ultimate instance.

- **tmMsgSend()/tmMsgRecv()**
  The receiver and sender thread IDs are temporarily exchanged. This is reasonable in respect to the automatic thread exit functionality. It was time consuming to detect this problem, most as even the arguments to the assembler functions are noted as expected – but completely different implemented in calls from assembler to C. The exchanging is documented in assembler but no explanation why this has to be done was found.

- **Sequence of Started Threads While Booting**
  The sequence of thread start is very important as the thread IDs are created at runtime for all threads, including the kernel threads. When Topsy i386 was debugged with the clock interrupt enabled, i.e. preemptive, a short delay was to be implemented after every statement printing on screen (else no information could be retrieved as the processing was to fast). The scheduler then was called several times during this time. Always a general protection fault occurred as an invalid thread was picked. Topsy includes the note

  ```
  /* the idle thread(s) guarantee that we always find a
     ready thread */
  ```

  in TMScheduler.c. So a longer period of error seeking was done (Topsy i386 was doubted) until it was noted that the idle thread was not yet started. The sequence of thread creation was changed – afterwards no screen output from user programs could be handled, the thread IDs are defined constants in Topsy/Topsy.h:

```
#define MMTHREADID  -1 /* Memory Manager Thread Id. */
#define TMTHREADID  -2 /* Thread Manager Thread Id. */
#define IOTHREADID  -3 /* IO     Manager Thread Id. */
```

Note that the kernel thread creation sequence is really important.

- **Memory Management on Kernel-User-Transition**
  Depending on the low level memory management applied in a port the messages must be adjusted internally with the right offset as done in Topsy i386 to provide the kernel with the correct local buffer addresses. Messages could be marshaled for easier transition of protection level. Implementing Topsy on a not memory shared multiprocessor environment would require this. For sake of velocity this was not done, so a correct solution must be found when porting Topsy to a new platform. Implementing a virtually mapped memory management in Topsy (note: this must be explicitly programmed for the memory management unit on the target system) could easily solve this problem, but would require more time available.

  The function ioCheckBufferAddress() in IO/IODevice.c must be correctly adjusted to the requirements of the new port's memory management.

- **Starting Topsy**
  Some architectures as for example the ia32 require a more deep initialization during startup before calling the first Topsy kernel function. For example Topsy i386 requires the definition and creation of several control structures for protected mode operation. So a function to provide these initializations must be implemented, refer to Appendix D for a detailed explanation on booting Topsy i386.

- **Segmentation Map**
  Topsy handles the memory layout at load time by a segmenation map. This map is built in MIPS-Topsy by an external tool, the BootLinker. It is statically built into the binary image.
  Topsy i386 builds this segmentation map during system start using the information retrieved from the binary image file header. For a detailed explanation, please refer to Appendix D, Section "Basic Initializations".

- **Processor State Save And Restore**
  Saving the processor state – when a kernel to user mode transition or vice versa is performed – is essential to a multithreaded operating system. Depending on the target hardware platform the save and restore procedures are provided or supported by the processor manufacturer. If not supported, this functionality must be well designed as the complete operation of Topsy depends on it.
  Topsy i386 solved this problem in a elegant way: only a small piece of code

is implemented in assembler, the remaining functionality is coded in C, refer to Appendix D.

- **Automatic Thread Exit**
  Exiting a thread without explicitly calling tmExit() requires the setup of a small help function called automaticThreadExit(). It is copied onto the thread's stack. This function performs the sending of a TMKILL message to the thread handler tmMain(). To execute automatically the thread exit it is important to define a correct return address that makes the thread jumping to this function when the program execution reaches the closing bracelet without a tmExit() function call. Problems occurred when setting the correct return address. The thread is completely built in the kernel memory region and finally copied into the user space. The return address must be correctly adjusted.

- **Driver Handling**
  A driver is opened by calling ioOpen(), closed by calling ioClose(). These functions both are located in Topsy/Syscall.c and perform a transition to IOMain (in IO/IOMain.c). ioOpen() returns the threadid of the started device. When calling the ioRead() or ioWrite() functions, control is passed to IODeviceMain() (in IO/IODevice.c).

C.3 Creating a New Port

When a new port of Topsy must be made, refer to the preceding chapter C.1 and chapter C.2 as to Appendix D, "The i386 Port", too.

The target platform may differ in several aspects, for example on the MIPS board used for the original Topsy a memory mapped IO architecture had to be respected while the personal computer of the Topsy i386 port is a mixed architecture, e.g. the text mode screen output can be generated by accessing a memory mapped region while programming the peripheral support chips as the timer or the interrupt controller requires direct access by IO ports.

Further the initialization of support chips can be completely different from one architecture to another, e.g. the programmable interrupt controller i8259 found in personal computers requires an initialization sequence that is not directly obvious.

Acknowledging peripheral signals is also hardware dependent, e.g. the interrupt controller used in personal computers continues recognization of peripheral interrupts only after the actual and all interrupts with a higher priority are acknowledged while MIPS Topsy needs acknowledgement to break repetitions, see section C.1.2 tmResetClockInterrupt() and in source code Threads/mips/TMClock.c for further details.

Facing problems like those requires detailed information on support chips as processors too. It can be hard to find them; Intel provides a complete reference set on the web since spring 1998: http://www.intel.com/

The i386 Port

Semester Thesis of Lukas Ruf

April 1998 – July 1998
Computer Engineering and Networks Laboratory, ETH Zurich
Supervisor: George Fankhauser
Professor: Bernhard Plattner

Abstract

This part of the i386 documentation provides a detailed explanation of the released port Topsy i386.

Starting with the development environment, this includes all tools used and developed for Topsy i386, a short introduction to "low level" programming is provided.

Passing over to the start of Topsy i386 a detailed explanation of the boot process is given. The initialization of the low level control structures used to make the ia32 architecture run in protected mode is explained including a short notice on the support provided by the ia32 architecture itself.

The Topsy i386 specialties are reviewed in the chapter "Topsy i386 Operation". Solutions found and design decisions taken are listed in this section.

A bibliography of books used for the development of Topsy i386 ends the documentation.

Appendix D

The i386 Port

D.1 Development

This chapter provides a detailed explanation of used and developed tools for the Topsy i386 semester project. The tools developed are all available together with the released Topsy i386 source code.

This subsection does not explain the hardware structure of the personal computer on a ia32 architecture basis more than needed. For further information please refer to the appropriate books listed in the bibliography.

D.1.1 Development Tools

The development was done using the GNU tools as provided with Linux stable version 2.0. Programming the HAL and the Topsy i386 Tools the following GNU development tools were used:

- gcc version 2.7.2.1
- ld version 2.8.1 (with BFD 2.8.1)
- objcopy 2.8.1
- objdump 2.8.1
- Make version 3.76.1
- size 2.8.1

Make managed the whole build run by calling the compiler for every file. On successful completion the linker concatenated and relocated the created object files into an "executable" ELF from which the binary image was drawn by the object copy program. Most on initial development the disassembled listings were of great help; the disassembling was done using objdump out of the created binary

image. The appropriate segment sizes were retrieved by the program size out of the ELF.

Compiler

Gcc was used for compiling C and assembler files. The options for compiling C files were the followings: -fno-builtin -m386 -fno-strength-reduce -fomit-frame-pointer -malign-double -malign-loops=2 -malign-jumps=2 -malign-functions=2

The alignment settings are provided to extend the speed of execution (the ia32 increases processing speed if the referred memory addresses are a multiple of 2, i.e. lay on an even boundary) as for reducing the size of the executable. Size reducing is also done by the "omit-frame-pointer" option. Omitting the frame pointer provides an image that cannot be easily debugged. The target platform is specified by the "386" statement. "no-builtin" makes gcc use only not provided functionality for program creation.

The GNU assembler, gas, makes use of gcc if the suffix of the file is a capital "S", e.g. TMHalAsm.S which includes the lowlevel routines for restoreContext and saveContext.

Linker

The linker was invoked with a slightly modified linker script: on the one hand the starting address was modified to relocate the single blocks to offset zero of the applicable segment and on the other hand the kernel and user are completely separated. Further the .data segment is concatenated into the .rodata segment together with the origin .rodata. Only two segments are created as Topsy internally also distinguish only two different segment types: data and text.

The segment start address can be specified providing the offset right after the starting segment, e.g. ".text 0x00000000".

Binary Image

objcopy created the binary image file to be executed by the loader directly without address relocation. This was done using the following options: -R .note -R .comment -O binary

The "-R" statements remove the not used segments (reduces size). By defining the "-O binary" flag the target file format is specified.

Disassembling

On initial development of Topsy i386 disassembled binary image files were of great help. Understanding the compiler and linker output is quite essential. A human readable formatted file with address information was created using objdump with the following arguments: -b binary -D -s -EL -m i386

"-b binary" specifies the input file format. The binary image file was disassembled. "-D" makes objdump disassemble all. "-s" makes objdump providing the full content of the input. "-EL" specifies the input file of format little endian, by the way, the standard format of a personal computer with a ia32 architecture. "-m i386" specifies the machine from where the input file comes.

Segment Size

Using the GNU tool "size" the appropriate space occupied by one of the two included segments could be determined in a easy way. The program size is called with argument "-A" to make size resemble output from System V size:

```
kernel.elf  :
subsection          size         addr
.text              31008            0
.rodata            10552        32768
.bss                 844        43320
.note                900        44164
.comment             900        45064
Total              44204
```

This information is gathered into a file to provide the required information to the kernel patch program (see below).

D.1.2 Topsy i386 Tools

KernPatch

KernPatch is another important tool for generating a loadable and bootable kernel and user image file. It inserts information of load size (in tracks of a 1. 44MB floppy disk, i.e. rounded multiple of 18*512B = 9216 Bytes) and code as data segment sizes into the binary image file header. The load size is used to read the image from floppy disk during boot of Topsy i386. Code and data segment sizes are used to initialize the Segmentation Map of Topsy (refer to the MIPS-Topsy Documentation therefore).

Binary Image File Header

The Binary Image File Header is located at the very beginning of an image file, i. e. either kernel or user image make use of the same functionality. The first ten bytes of this header must be

```
"TOPSYi386"\0
```

to mark this file as a Topsy i386 image. At location 0x10 the load size is inserted, 0x14 contains the code and 0x18 the data segment size. Every size note consists

```
typedef struct BinaryFileDesc_t {
    char marker[10]; /* "TOPSYi386"'\0' */
    char filler1[6]; /* Unused space to reach 4Byte alignment */
    long tracksize;  /* Number of Tracks to read from disk */
    long codesize;   /* Size of Code Segment in file */
    long datasize;   /* Size of Data Segment in file */
    char filler2[36];/* Undefined in BinaryFileDesc  */
} BinaryFileDesc;
typedef BinaryFileDesc *BinFileDescPtr;
```

Figure D.1: Binary Image File Header

of four Bytes. So, at location 0x1C additional information could be located. Execution of binary image starts at 0x40. 64 bytes of header information were thought to be far enough for future extensions.

It is important to note that the information of code segment size implies also the starting address of the data segment size in the image file: the data start equals to the code size rounded up to 4KB page boundary. This information is used while dynamically generating the segmentation map, see below.

Information required to patch the kernel is retrieved by parsing the output file of the size command (output redirected into a file *.size, e.g. kernel.elf.size).

The Binary Image File Header can be found in Startup/ia32/BinaryFile.h.

D.1.3 Keyboard

Keyboard is in fact another Topsy i386 Tool. It is separetely noted as the understanding of Keyboard is important for defining new keyboard layouts as the processing of keystrokes too. This section explains the keyboard handling of Topsy i386 while regarding to the tool Keyboard.

Topsy i386 is a standalone operating system. No other tools for running or booting Topsy i386 are required. So the low level keyboard handling must be implemented. Supported in this version are all MFII Keyboards (101 keys), tests with older ones could not be performed in lack of such a thing.

Note: In this subsection a port is short for an IO port.

When IBM designed the Personal Computer Keyboard mapping scheme they appeared to encode the key stroke signaling not straitforward. A difficult scheme of keystroke signal encoding was provided with key codes, extended key codes and second extended key codes. The keyboard controller provides these information on press and release time of keystrokes via a port in IBM's port landscape: Ports 0x60 and 0x64. The port 0x60 is used for input (R:Reading) and output (W:Writing) of bytes to the controller, 0x64 handles the keyboard control regis-

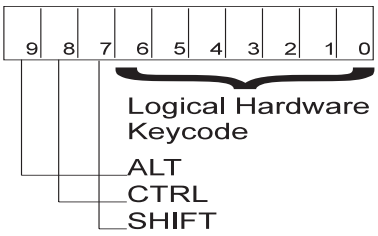

Figure D.2: 10bit Topsy Keycode

ter, readable and writable. Whenever the keyboard controller wishes attention an interrupt 1 is signalled. Depending on whether a key is pressed or released bit 7 of the received key code is unset or set, thus providing 128 easy to distinguish key codes in one-byte. This range would be sufficient for all key codes now available (incl. the "MS-keys"). But IBM decided to provide a coding scheme not directly obvious to everybody. Depending on the kind of key code (normal, extended, 2nd extended) one, two or even three bytes are generated and sent to port 0x60. This could be easily handled if the same key would not generate different key codes depending on the additional keys pressed. Further, some key codes change depending on first and single or repetitive key code generation. Solving this problem required a lot of time as no really character key code mapping was found that provided accurate information on all keystrokes; the single key code to character mapping was done by hand.

Another point of interest is the fact, that pressing several keys simultaneously generates always only the last key code except the last key pressed is a key modifier (alt, ctrl, shift).

Further, where needed, a raw key code translation is performed mapping the not uniquely defined keystroke to a uniquely defined one. This unique key code points into the first table of keystroke translation, the so called Hardware Key Translation Table. At the pointed location a so called logical hardware key code is noted. This is retrieved and combined with the key modifiers pressed. Depending on the key modifiers pressed the leading three bits are set or unset. This leads to a 10bit Topsy Keycode, see figure D.2.

This Topsy Keycode provides eight ranges depending of which key modifiers are pressed simultaneously together with the "normal" keys. Normal keys are all others than the key modifiers. The key modifiers pressed solely do not generate a keycode to be further processed.

The Topsy Keycode (D.2) points into a table of 1024 different one-byte ASCII characters, i.e. the keyboard layout. The "normal" range (no modifier pressed) of the Topsy Keycode is as close as possible to the ASCII character code to provide an easy way to create different keyboard layouts.

Special Keycode Processing

Three exceptions were made in keycode processing:

1. **ALT-CTRL-DELETE**
   Pressing this combination immediately reboots the system. This is handled internally in the keyboard handler. Rebooting the system in version 1.0 switches back to real mode an performs a computer restart by jumping to a predefined location in the ROM area. This can be extended in later versions, for example by sending terminate messages to the running threads.

2. **Special Code 0x00**
   If at the location pointed into the 2nd Key Translation Table the value of 0x00 is found, processing is aborted, i. e. inserting character 0x00 into the keyboard buffer is impossible.

3. **Special Code 0xFF**
   If at the location pointed into the 2nd Key Translation Table the value of 0xFF is found it was intended to send a special message to the process/thread the focus is actually set. Actually the processing of 0xFF ends as with 0x00 (lack of time).

Take a closer look to the key translation tables as provided with this documentation, resp. the source code of Topsy i386 for a deeper understanding.

Generating the Key Translation Tables

Provided with the source of Topsy i386 is a small tool called Keyboard that generates a table (an assembler array of byte) out of the as argument to this tool specified ASCII text file. As noted above, a key message sending was intended to implement. So, a second table will be generated if not inhibited by an additional argument to the table creator. This second table is included into the build process of a Topsy i386 kernel image. The generated key translation tables have to be copied into the directory IO/ia32 by hand or specifying "install" as argument to make. Take a look at the included Makefile in directory ../Tools/Keyboard.

Initial Keyboard Layout

Provided with Topsy i386 is – for a first release – a close mapping to Swiss German VSM keyboards. This keyboard layout is defined in ../Tools/Keyboard/KeybSG.TBL.

New Keyboard Layouts

A new keyboard layout can be generated by modifying KeybSG.TBL. Modify it according to the rules noted in Keyboard.dok. The hardware translation table

(../Tools/Keyboard/MakeKey.TBL) should not be altered. If modified, the mappings in a keyboard layout table must be adjusted accordingly.

Keyboard Control when Topsy i386 is Running

The complete key code handling is managed in IO/ia32/Keyboard.c. Interrupt 0x01 is handled by _KeyboardISR() in this source. This handler first calls the function KeyModifier with the raw key code as argument. The modifier-handler does the real hard work, handling the different kind of keystrokes, deciding whether the keystroke has to be processed further on or not (when a "normal" key is released, no code is generated to be inserted into the keyboard buffer). The first translation is performed in KeyMappedCode(), the second in KeyASCIICode(). Into the statically allocated keyboard buffer the character is put by the function KeyIntoBuffer(). An interface to the TTY driver is provided by the _GetChar() and _IsKeyPressed() functions.

D.2 Booting Topsy i386

Topsy i386 in the released version boots by default from a high density 1.44MB floppy disk, i.e. 512 bytes per sector, 18 sectors per track, 80 tracks per side, and two sides. The boot procedure is split off into three stages:

1. Master Boot Record
2. Boot Loader
3. Kernel and User

The splitting of the boot procedure into three independent programs provides an easier adaption to the requirements of loading Topsy i386 by another boot loader like loadlin for example. A further advantage was found while developing Topsy i386: the real mode programs could be separated completely from the running Topsy, so nearly the same startup module could be used for kernel and user.

D.2.1 Master Boot Record: CoreBoot

When power is turned on, the PC jumps into the ROM BIOS code to perform the basic computer tests. The code there loads the very first physical sector from a floppy or hard disk H/T/S 0/0/1 (Head/Track/Sector) into RAM at location 0x0000:0x7C00 in real mode segment-offset notification, i.e. linear 0x7C000, and executes this small program.

BIOS completes work and passes control to this master boot record program. The code there normally loads the operating system, so does the Topsy boot record program called CoreBoot.

CoreBoot must set the boot stack; the boot stack is set to 0x1E000 (linear).

CoreBoot initializes the VGA controller to enable direct video access on the first text page located at IO address 0xB8000 in the IO space. A short message to show functionality is displayed and then the next stage is loaded. CoreBoot assumes the third track to be the second stage boot loader named CoreLoad.

D.2.2 Boot Loader: CoreLoad

CoreLoad is stored on disk at location H/T/S 0/1/1, i.e. the third track on disk. The whole track is read by CoreBoot into RAM at location 0x10000. When CoreLoad receives control it verifies the following points:

- Processor type: at least a i386 or compatible is required.
- Processor mode: real mode, the processor's boot mode, is required.

If all tests were successfully passed, the kernel and user blocks are loaded from disk.

The binary image file header (refer to section D.1.2) is analyzed. If a valid Topsy i386 image file is found, it is loaded using the tracksize, i.e. the number of tracks to be read by CoreLoad. A multiple of whole tracks is loaded as reading track wise is supported by the called BIOS function (INT 0x13) and is much faster than reading sector wise.

The Kernel image is read first from disk into RAM at location 0x20000 (=128KB). The User image is read second from disk into RAM at location 0x40000 (=256KB).

In real mode the RAM is logically divided into segments of 64KB. For operating systems only 640KB are usable in real mode. Actually the kernel block is about 45KB in size, the user block 20KB. To provide room for initial extensions to the user block, the image can grow to 2*(64KB - 512B). This size results from the number of whole tracks fitting into one real mode segment, i.e. seven tracks fit into one real mode segment.

While the images are loaded from disk they are split as obvious. CoreLoad must not concatenate these potential four parts, this functionality is implemented in BuildSegMap() for the user block, see Startup/ia32/init.c.

An analogous function for kernel concatenation must be implemented in Startup/ia32/start.S when the kernel should reach the size of 65025B. Slight extensions make the kernel image fast grow to bigger sizes, so 127KB block size should provide some room for first extensions, then a protected mode file handling with disk interface should have been included to load the remaining parts into RAM while Topsy i386 is already full functional in protected mode.

After both parts were successfully loaded into RAM CoreLoad prepares the transition to protected mode.

D.2.3 Basic Initializations

Address Line 20

In real mode only 1MB is directly addressable as noted before. This is done using 20 address lines of the bus. Intel decided when the 80286 was developed in 1982 to provide an extended addressing scheme up to 16MB. When this new address range should be addressed directly the famous A20 gate – the 21st address line – was enabled. By default it is disabled, compatibility to programs should be provided. IBM decided to enable the A20 gate by programming the keyboard controller (i8255). When in 1983 Intel developed the 80386 the address room directly addressable was extended once more using the full bandwidth of the 32b bus: 4GB. Intel preserved the same functionality to turn on the A20 gate.

Programmable Interrupt Controller Initialization

In this section the word "interrupt" names an exception generated by a peripheral support chip by raising the corresponding interrupt request line. The word "exception" is used when control is passed to the processor.

In 1981 the IBM PC was designed using the i8088 processor which provided only 8 fault exceptions. IBM decided to implement 8 hardware interrupt lines wired to the i8259 PIC (Programmable Interrupt Controller). These lines were by default programmed to map to the exception vectors 0x08 to 0x0F. With the IBM XT the number of hardware interrupt lines was extended to 16, a 2nd, cascaded PIC was used. For sake of compatibility the formerly specification remained, the newly created interrupt lines were programmed to exception vectors 0x70 to 0x78. The 80386 included new fault exceptions. The processor decides were the default exception vectors are located. So Intel reserved the 32 first vectors for internal use. Faults normally halt a processor in real mode, fault exception handling is provided only in a very rudimentary way. So in real mode the default programming could be kept alive.

In protected mode the fault exception handling is supported by the processor with features as for example instruction restart. It could be possible to keep the standard mapping from hardware interrupt to exception vectors but this would require an exception-source verification for every exception occurring. To avoid this annoying and time consuming procedure the PIC can be reprogrammed to map the interrupt requesting lines to another vector. The 32 first exceptions are reserved for internal use of the ia32 processors. So, CoreLoad re-programs the hardware interrupt lines to be mapped to exceptions 0x20 to 0x2F. A direct mapping was provided, so subtracting the offset 0x20 results in the number of the original interrupt request line.

Further when the reprogramming of the PIC was done, Topsy i386 implemented a constant priority interrupt strategy, i.e. the hardware interrupt priority always remain equal and does not alter on every found interrupt. An explicit end-of-interrupt command must be sent to the PIC to acknowledge the handled interrupt and re-enable recognition of further interrupts at equal or lower priority.

GDT and IDT Initialization

After the support chips were initialized, i.e. the A20 gate (see section D.2.3) and the PIC (see section D.2.3) CoreLoad has to initialize the control structures used for a secure switch from real mode to protected mode: this is done by the creation of the Global Descriptor Table. The Interrupt Descriptor Table Base Register is loaded with default values. The Global Descriptor Table is the controlling structure for memory access and program execution in protected mode.

Separating the available memory into segments is needed as otherwise no protection could be applied.

Seg. No.	Seg. Name		Seg. Size
1.	NULL	Blocking of selectors set invalid	0
2.	Kernel Code	Execution of kernel code	512KB
3.	Kernel Data	Kernel data segment	128MB
4.	Global Data	Whole virtual address space	4GB
5.	VGA Text Page 0	Avoid overwritten memory mapped IO ranges	4000B
6.	GDT	Allow modifications of the GDT itself	60KB

Table D.1: GDT as initialized by CoreLoad

Table D.1 provides an overview of the basically installed Global Descriptor Table. Further explanation is needed on the one hand of the structure itself and on the other hand of the sizes applied.

Structuring the memory is a decision with effects to the running operating system, for example the address adjustment is required as the user space is not overlapping the kernel space, see below.

Protection mode operation of the ia32 processors requires some control structures, see section D.2.3. They are of no use to the running Topsy kernel in version 1.0, so they are completely separated from the kernel address space. They are located in the first 128KB of RAM providing enough space for extensions as for example virtually mapped memory management with hardware support. Continuous address space ends at 0xA0000 (=640KB). There the memory mapped IO area is located. That's why the kernel code segment as a continuous direct mapped memory block includes a range of 512KB (640KB-128KB). Initial memory management of Topsy i386 is restricted to the first 128MB of RAM. So, the kernel data segment includes only a range of about 128MB of RAM.

The global data segment is provided to allow access to all potential available RAM in ia32 environment, 4GB.

To securely avoid overwrite operations of essentially required memory mapped IO areas the VGA text page one was separated. Extensions as displaying other text pages or switching to graphic mode would require a redesign of this selector.

Finally the Global Descriptor Table segment selector is required to allow modifications and extensions to the GDT when Topsy i386 is running. It could be possible to implement the modifications with the Global Data Segment, but this design was more appreciated; no offset addition is required when modifying the GDT, overruns of the GDT segment are blocked too. The GDT in fact was restricted to 60KB. A complete real mode page could be used for the GDT (64KB), but the lowest 4KB include on the one hand the original real mode interrupt vector table and on the other hand the BIOS information area that could be of use when implementing for example a parallel port driver, a serial line interface or even a harddisk driver. Further the BIOS information area is already used as the real mode interrupt vector table is located there: simultaneously pressing ALT-CTRL-DELETE performs a so called "warm reset" of the PC (no hardware

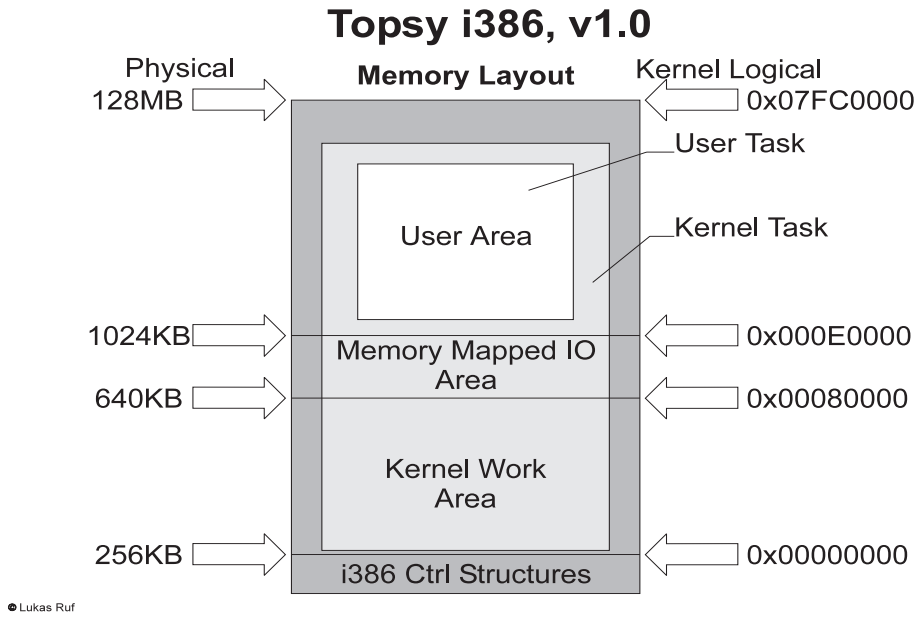

Figure D.3: Memory layout of Topsy i386 v1.0

checks are performed); this requires at location 0x472 the value 0x1234 as the real mode interrupt vector table too – if no real mode interrupt vector table is available, the computer hangs as a consequence to a double fault exception.

This segmentation results in a physical memory layout presented in figure D.3.

It is important to note, that the values provided on the left side of e figure D.3 denote the physical location in RAM while the ones on the right side the displacement relative to the kernel data segment start.

Switching to Protected Mode

The code fragment provided in figure D.4 performs the complete switch to protected mode. The ia32 archtitecture starts in real mode for compatibility reasons.

In fact the "real" switch to protected mode is done when bit zero of the control register zero (CR0) is set to one: Protected Mode Enable. The byte sequence that coded "protected mode far jump" is not available as a mnemonic instruction neither in GNU Tools nor in TASM: the actual code segment still is a 16b segment.

```
;Topsy i386                                              Lukas Ruf
;-------------------------------------------------------------
; Switch to Protected Mode
PM_Switch       PROC    NEAR
        ;////////////////////////////////
        ;/ Kernel is at right place    //
        ;/ - Switch to PM              //
        ;/ - Jump into Kernel          //
        ;////////////////////////////////
        LIDT    FWORD PTR IDTptr        ; Interrupt Descriptor Table
        LGDT    FWORD PTR GDTptr        ; Global Descriptor Table
        MOV     EAX,CR0                 ;
        OR      EAX,1                   ; Switch to Protected Mode
        MOV     CR0,EAX                 ;
        JMP     NowInPM                 ; Clear Prefetch Queue
        NOP
        NOP
NowInPM:
        MOV     BX,gOSData_Sel              ; Setup Data Selectors
        MOV     DS,BX                       ; D Selector
        MOV     ES,BX                       ; E Selector
        MOV     FS,BX                       ; F Selector
        MOV     GS,BX                       ; G Selector
        MOV     SS,BX                       ; S Selector
        MOV     EAX,OSS_TOP                 ; Set Top of Stack
        MOV     ESP,EAX
        ;/////////
        ;// PROTECTED MODE FAR JUMP TO (OSCodeSelector:OSKernBegin) =
;// C-Kernel-Start
        ;/////////
        DB 066h                 ; 32 bit Data prefix
        DB 067h                 ; 32 bit Address instruction prefix
        DB 0EAh                 ; far jump opcode
        DD cOSKern_Begin        ; 32 bit Offset   ( Code Offset )
        DW gOSCode_Sel          ; 16 bit Selector ( Code Selector )
        RET                     ; This should never be executed,
PM_Switch  ENDP
```

Figure D.4: Switch from Real to Protected Mode

```
void architectureInit() {
   TextSize = PAGEBOUNDARY(__getTextSize());
   DataSize = PAGEBOUNDARY(__getDataSize());
   VideoInit();
   ClearScreen();
   /* GDTInit() must come first */
   GDTInit();        /* Global Descriptor Table Initialization */
   /* IDTInit() must come second */
   IDTInit();        /* Interrupt Descriptor Table Initialization *
   TSSInit();        /* Kernel Task State Selector */
   ExceptionInit();  /* Processor Exception Handling */
   MemoryInit();     /* Low Level Memory Management */
   InterruptInit();  /* Peripheral Interrupt Handling */
   TimerInit();      /* start Timer 0 */
   BuildSegMap();    /* Create this "SegMap Table" */
   return;
}
```

Figure D.5: Architecture Init as Performed by Topsy i386

Starting the Kernel

Kernel startup is responsible for setting the basic protected mode wiring to allow the Topsy Kernel being run without modifications. This wiring is managed by the architecture init function found in Startup/ia32/init.c.

When Topsy i386 is started it first retrieves the binary image file header information. This information is preserved in the kernel data segment for further use.

The assembler startup function sets segment selectors for security and calls the main() function in the hardware independent Topsy startup code. The first function called by main() must be the architectureInit(); the code as used in Topsy i386 is provided in figure D.5.

Protected Mode Control Structures

The protected mode control structures are set in the function architectureInit(). The sequence of initialization is really important. Refer to figure D.5.

Global Descriptor Tale Initialization

The initialization of the Global Descriptor Table in fact does nothing than setting the number of fixed implemented GDT entries. Even if this seems to be silly, it is important as all of the function accessing and modifying the GDT depend on this value. GDT functions are located in Memory/ia32/MMHal.c.

Interrupt Descriptor Table Initialization

The Interrupt Descriptor Table Initialization sets up a description entry of the IDT in the GDT. In this version of Topsy i386 the IDT is statically set to 0x10000, right after the top of the GDT. This selector is used to access and modify the IDT. 256 IDT entries can be set. As explained before, the first 32 entries are set to handle the processor fault exceptions, the following 16 to handle the hardware interrupts and the 49th entry for syscall handling, i.e. tmMsgSend() and tmMsgRecv(). IDT functions are located in Memory/ia32/MMHal.c.

Task State Segment Table Initialization

The Task State Segment is the location where the ia32 architecture would save the processors state when switching between processes. In Topsy i386 this segment contains only the required settings of code, data and stack segment for protection level transition from user to kernel level. TSS functions are located in Memory/ia32/MMHal.c.

Exception Initialization

The Exception Initialization is used to set the processor fault exception vectors to point to the lowest level exception handling functions. For further information on exception handling please refer to section D.3.1 on page 17. Low Level Exception Init is implemented in Threads/ia32/TMHal.c.

Low Level Memory Initialization

The available RAM is determined by the low level memory initializer by writing a predefined value to the last four bytes of a megabyte. The next opertions read the value back and compares it to the predefined one. This procedure is repeated until the read value is not equal to the written one.

This information could be used to extend the memory handling abilities of Topsy itself, i.e. the user space could be extended from 1MB. The low level memory init already initializes the required structures for virtually mapped memory management supported by the ia32-MMU. The space provided for this future virtual mapped memory management is actually limited to 128MB but could easily be extended to the full range of 4GB.

Peripheral Interrupt Initialization

The Peripheral Interrupt Initialization sets the hardware exception vectors in the IDT to point to the lowest level hardware exception handling functions. For further information on exception handling please refer to section D.3.1 on page 17. Low Level Interrupt Init is implemented in Threads/ia32/TMHal.c.

Timer Start

Timer Start is needed there as the Timer Interrupt is the first and vital hardware exception for Topsy i386. The PIC i8259 needs an explictly initialization of the hardware interrupts. To enable the settings of Timer mode and frequency from the portable kernel level by the function setClockValue() the low level settings are handled here. Low Level Timer Init is implemented in Threads/ia32/TMClock.c.

Dynamically Created Segmentation Map

In Topsy the Segmentation Map Descriptor Table is build during link time. The address of this table is statically defined in Makefile.mips as `SEGMAP=800FFF00`. This SEGMAP is then passed to the compiler to set this value in Startup/Startup.c.

Topsy i386 generates this table using the values patched in the Binary Image File Header in function BuildSegMap() located in Startup/ia32/Init.c. To make the address of the so created table globally available and usable where needed SEGMAP was defined as the address of this table in IOHal.c. This is not a really beautiful solution – but the portable level should not be modified when possible.

Starting the User

As already mentioned above the user program makes use of nearly the same startup functionality, so, on the one hand as the user is linked separately into a standalone image and on the other hand as this image is constructed the same way: 64 Byte of binary image file information (refer to section D.1.2), instruction start at location 0x40.

Remarks

CoreBoot and CoreLoad were developed using TASM (Borland's Turbo Assembler) Version 5. This was done as on the one hand they do only fundamental functions not needed in protected mode while running Topsy i386 and on the other hand when this project was started, the author already knew how to handle the real mode assembler programming using TASM while the real mode programming with the 32b GNU Tools looked not very useful to him. So the binaries are provided together with the source code. In the near future CoreBoot and CoreLoad are ported to as86 if this assembler supports the same functionality. It is not appreciated to make use of hacks as splitting an image file to receive CoreBoot and CoreLoad separately.

D.3 Topsy i386 Operation

This chapter discusses the solution found to solve the difficulties of porting Topsy to ia32 architecture. It is intended to focus only to the solutions not obvious to everybody. Only the following points include solutions not implemented in every other freely available operating system:

Exception Handling discussed in section D.3.1.

Context Saving explained in section D.3.3.

Restore Context explained in section D.3.4.

Address Adjustment explained in section D.3.5.

D.3.1 Exception Handling

As noted before the exception handling of Topsy i386 is split into a part coded in assembler (the so called low level exception handling) and into a "portable" part (the exception handling itself). This was done as the ia32 provides on the one hand a table of 256 possible exception handler addresses and on the other hand no possibility to retrieve the cause of the exception itself if the exception does not result from a peripheral device. A slight difference is noted between the processor exceptions, the hardware interrupts and the software exception. So they are discussed separately. The code can be found in Threads/ia32/TMHalAsm.S.

When Topsy i386 was developed it was intended to provide a most elegant and generalized solution for exception handling that avoids redundancy wherever possible. So some extra instruction were inserted when needed to build a consistent frame. A detailed explanation of stack layout when entering an exception handler can be found in all the books noted in the bibliography.

D.3.2 Low Level Exception Handling

The ia32 architecture provides three kinds of fault exceptions: faults, traps and aborts. Further some of them provide an error code, that for example includes the faulting memory address, and some do not. So, to provide a consistent stack layout a dummy code (zero) is pushed onto the stack when no error code is provided by the processor. Further the exception error code is saved for later use. For a more detailed imagination see the listing in subsection D.3.2.

Differences to MIPS-Topsy

Topsy i386 implemented a default fault exception handling in a slightly different way to the one found in the original Topsy. The default error handlers there display only a message and execute a call to the error handler, it distinguishes

further the kind of exception and decides which kind of handler should be called by implementing the following functions in Threads/mips/TMHalAsm.S:
void syscallExceptionHandler(ThreadId);
void hwExceptionHandler()

Topsy i386 generalized this default handling in providing the exception number which serves as index into a table of fault exception messages. The standardized fault exception handler then only displays the corresponding message and executes the same call to the general error handler which kills the faulting thread.

Specialized exception handlers can be set as normal via tmSetExceptionHandler() or tmSetInterruptHandler(). This does not affect the low level exception handlers.

Topsy i386 tried to leave assembler level as soon as possible and to implement the required functionality better readable in C.

Implementation

The following code is only an extract to show the principals of the implementation coded in Threads/ia32/TMHalAsm.S. Refer to figure D.6 on page 20.

Code explanation (figure D.6):

- Fault Exception Handlers
  The low level fault exception handlers are named `__NEx_nn__` where "nn" is the hexadecimal number of the fault exception starting at 00 and running up to 1F, e.g. `__NEx_00__` is the low level fault exception handler for divisions by zero.
  The exception number saved on the stack is coded in `T_Ex_nn` where "nn" is analogous to the the note above. As some exception provide an error code and others do not, on the one hand a macro is used to make the name `__NEx_nn__` globally available to the installer ExceptionInit() coded in Threads/ia32/TMHal.c. and another macro is used to call the generalized low level exception handler.

- Interrupt Handlers
  A hardware interrupt calls the corresponding low level interrupt handler always in the same way. So only one macro entry handles the complete function definition.
  The low level interrupt handlers are named `__NIR_nn__` and provide the interrupt number by `T_IR_nn` where "nn" is corresponds to the IDT slot number and simultaneously to the hardware interrupt number if the number is calculated modulo 0x10.

- Software Exception Handlers
  Message passing is realized using the function tmMsgSend() and tmMsgRecv(). They in fact do nothing other than packing the information required and provided into processor registers and launching a software exception by `INT 0x30`. The low level software exception handler is defined analogous to the interrupt and fault exception handlers. Two software exception handlers are defined: the first one, named `__INT_30` and providing the exception number `T_IS_30`, is used as the SYSCALL (known from MIPS); the second one, named `__INT_30` and providing the exception number `T_IS_30`, is the handler of "unhandled" interrupts as erroneous user or kernel programs may launch a software interrupt easily by noting INT 0x30 anywhere in the program code. Uninitialized IDT entries may lead to unpredictable functions, normally a general protection fault. The second software exception handles all not used IDT entries. The addresses are all set to this second software exception handler.

The secondly called macro TRAP() first saves all general purpose registers on the stack ("pushal"), saves the provided exception number and then jumps to the generalized low level exception handler, called `__general_ExceptionHandler`.

The low level generalized exception handler saves all segment selectors on stack not implicitly saved by the processor and sets the data and extra segment selectors to provide a standard C environment. The current top of stack pointer is saved for stack adjustments made in the general exception handler at C level, see figure D.6.

Stack Frame on Entering C Code

When `_INTHandler()` is called the following stack frame is provided as listed in figure D.7 on page 21.

The exception handler in TMHal.c is defined as following:

```
void _INTHandler(unsigned int esp, struct ProcContext_t frame);
```

It is important that the complete ProcContext is directly referenced by defining the struct and not a pointer to it. So, the processor state is retrieved by the argument "frame".

```
/* Definitions for Interrupt Handling.  */
#define TRAP(_irqno_) \
        pushal ; nop ; \
        pushl $(_irqno_) ; \
        jmp __general_ExceptionHandler
#define IRQ(_name_,_irqno_) \
        .align 4 ; \
        .globl _name_ ; _name_: ; \
        cli ; pushl $0 ; \
        TRAP(_irqno_)
/* Definitions for Exception Handling.  */
/* Define the entry for IDT Vectorization:-)
   Please remark: as some exception serve with an error code,
   there is no generalization possible :-( */
#define IDTVEC(_name_) .align 4 ; .globl _name_ ; _name_:

/* _INTHandler is the generalized C exception handler coded in
   TMHal.c */
.extern _INTHandler
/* Define the Exception Handlers */
IDTVEC(__NEx_00)         /* Divide Error */
        pushl   $0       /* dummy error Code */
        TRAP(T_Ex_00)    /* call trap handler */
IDTVEC(__NEx_0B)         /* Segment not present (Error Code) */
        TRAP(T_Ex_0B)    /* call trap handler */
/* Define the Interrupt Handlers.  */
IRQ(__NIR_00,T_IR_00)
...
IRQ(__INT_30,T_IS_30)   /* Software Interrupt equals to "SYSCALL" :-) */
IRQ(__INT_31,T_IS_31)   /* Software Interrupt: Block all other */
...
__general_ExceptionHandler:
        pushl   %ds
        pushl   %es
        pushl   %fs
        pushl   %gs
        movl    $gcKDSEL,%eax
        movw    %ax,%ds
        movw    %ax,%es
        pushl   %esp     /* provide even current stack pointer for
                            stack address adjustment purposes in
                            _INTHandler() */
        call _INTHandler /* I suppose _INTHandler() always calls */
                         /* restoreContext() :-) */
        cli              /* If this _INTHandler() ever should */
        hlt              /* return, hang the machine: I'm human.  */
```

Figure D.6: Implementation details for low level exception handling

```
#ifndef __TMHAL_H
#define __TMHAL_H
...
/*
 * Exception Stack Frame
 */
typedef struct ProcContext_t {
  int tf_gs;
  int tf_fs;
  int tf_es;
  int tf_ds;
  int tf_trapno;
  int tf_edi;
  int tf_esi;
  int tf_ebp;
  int tf_temp_esp;
  int tf_ebx;
  int tf_edx;
  int tf_ecx;
  int tf_eax;
  /* NOTE: To this location tf_temp_esp is pointing to :-) */
  int tf_err;
  int tf_eip;
  int tf_cs;
  int tf_eflags;
  /* below only when crossing rings (e.g. user to kernel) */
  int tf_esp;
  int tf_ss;
} ProcContext;
...
#endif __TMHAL_H
```

Figure D.7: Stack Frame

D.3.3 Context Saving

Topsy i386 saves the processor context when an exception occurred on the stack and passes this information to the C level code. The C code now can easily copy the data to the context frame provided in the thread list. So, the context is saved by C code and not as done in MIPS-Topsy by assembler code with some address retrievals.

It is most important to note that the ia32 architecture was developed to support multiprocessing environments. In such operating systems a process is specified by its own TSS. The ia32 architecture automatically saves and restores the complete register set in a TSS on process transitions. No support is provided to multi threaded operating systems. In ia32 notification style Topsy i386 provides only a single tasking operating system, as only one TSS is initialized and started – no other is required. A protection level transition can be performed in the current task context as an exception handling without protection level transition too. If a protection level switch happens, the ia32 processor retrieves the target code and stack segment selector as the target top of stack pointer from the current TSS: in Topsy i386 the so called Exception Stack located physically at 0x9E000, virtually at 0x7E000 (relative to the kernel data segment start). If no protection level transition is performed, the current stack is used for information saving. So, the actual stack segment selector and top of stack pointer are NOT saved to stack. To provide a unique stack layout without too complicated assembler coding the C code always saves the processor context in the Topsy thread environment first and then adjusts the stack settings for restore. The error code is removed from stack by adding a defined constant to the Topsy thread environment top of stack pointer. For further details please refer to the source code provided in Threads/ia32/TMHal.c.

D.3.4 Restore Context

Processor context restoration is even more crucial in a multi threaded operating system on ia32 architecture than the context saving. The most important code fragment is provided in the next subsection (see section D.3.4) and explained afterwards.

Implementation

The code restoration needs some hard core coding with stack adjustments directly. So, not equal to the context save, it really needs coding in assembler. The complete restore function is provided in figure D.8 on page 23.

The reader noted that the processor context was directly restored from the thread environment processor context space. On the one hand this provides a really elegant solution on the other hand there is in fact no byte copy required

```
/* restoreContext(ProcContextPtr Context); ****************************/
FRAME(restoreContext)
ENTER   /* Execute an ENTER to function entry as to provide a stack
             * frame as common in C/Topsy-Assembler */
    cli                     /* disable interrupt execution -- we modify
                             * the stack directly. */
    movl    ARG1,%eax       /* store the address of ProcContext in eax */
    movl    %eax,%esp       /* let esp directly point to this context. */
    popl    %gs             /* restore these two segment selectors */
    popl    %fs
    popl    %es             /* restore Extra Segment Selector */
    popl    %ds             /* restore Data  Segment Selector */
    addl    $4,%esp         /* remove INT no. from stack  */
    popal                   /* restore all general purpose registers */
    addl    $4,%esp         /* remove err no. too */
            /* No modifications were made to the stack structure provided
             * by the Process Context. */
    cmpl    $gcKCSEL, 4(%esp)       /* compare the provided Code Selector */
    jne     _Nothing_Has_To_Be_Done /* jump if not equal */
        /* A restoration of a kernel origin has to be performed */
    pushl   $gcKDSEL        /* make sure the stack segment equals to the */
    popl    %ss             /* kernel data segment */
            /* save %edi to xtemp: no mutex or semaphore is required as all
             * ints are disabled --> directly access all */
    movl    %edi,%ss:xtemp  /* note: here we are in kernel data segment ! */
    movl    12(%esp),%edi   /* %edi = tf_esp */
            /* make room for the temporarily setting of EIP, CS and EFLAGS
             * on the stack of the thread to be started */
    subl    $12,%edi
    movl    %eax,%ss:xtemp2 /* we can do this: SS is Kernel Data Segment */
    /* %edi points to space where EIP,CS and EFLAGS have to be moved to */
    movl    0(%esp),%eax
    movl    %eax,%ss:0(%edi)        /* transfer EIP */
    movl    4(%esp),%eax
    movl    %eax,%ss:4(%edi)        /* transfer CS */
    movl    8(%esp),%eax
    movl    %eax,%ss:8(%edi)        /* transfer EFLAGS */
    /* restore the previous saved %eax from the kernel data segment */
    movl    %ss:xtemp2,%eax
    movl    %edi,%esp       /* now forget the old stack, move to the new */
    movl    %ss:xtemp,%edi  /* restore the original %edi */
_Nothing_Has_To_Be_Done:
iret                    /* and go either to heaven or hell !! */
/**End of restoreContext()************************************************/
```

Figure D.8: Assembler Code Fragment for Context Restoration

as the context is not modified while restored. The `pushl` and `popl` instructions used in this function do only overwrite the exception number of the context to be restored. This is no problem at this point as the processor context will be saved by a next exception providing a new "exception number".

For further details refer to the source of the low level exception handling provided in TMHalAsm.S and TMHal.c both located in Threads/ia32.

D.3.5 Address Adjustment

Depending on the memory layout defined by setting the entries in the Global Descriptor Table a segmentation of the RAM was realized in Topsy i386. This segmentation is explicitly required to provide kernel and control structure memory protection of the user programs. The user space starts at location 0x100000 (1MB boundary), the kernel space at location 0x020000 (128KB boundary). Segmentation in ia32 architecture provides an address space starting locally to segment start at offset zero.

The kernel data segment includes the user space too. This is required to easily access that space, e.g. for program installations. User program start does not require any address adjustment as the kernel simply "jumps" to the user startup routine (see subsection D.2.3). But already when a thread is started and therefore the kernel required to install a new stack segment or the user executes an ioWrite() or ioRead() function calls kernel and user both refer to different addresses locally even the space located in RAM dereferenced is equal to both.

So, on transitions from kernel to user space and vice versa the message base address and all buffer addresses as included in some messages need an adjustment, i.e. either an addition or a subtraction of 0xE0000 to or from the address. This displacement is obviously retrieved from the subtraction of 0x100000 minus 0x020000, the start addresses of the user and kernel space; refer to figure D.3 on page 12.

These address adjustments are performed in the function msgAdjust() found in
Threads/ia32/TMHal.c. Refer to this source for implementation details and documentation.

This solution was implemented for simplicity reason and lack of time. The advantages of this solution are laying in its evidence for even non professionals as in the development requirements too – no further hardware dependencies' knowledge had to be achieved. The disadvantages are obvious too: Processing all possible messages requires a huge amount of cycles. Further implementing a new type of message with buffer addresses included requires an extension of the address adjustment function (by the way: anyone implementing a new message type should be capable to extend the required function).

Implementation Alternatives

Several alternatives to the implemented address adjustment can be found; some of them are provided in table D.2.

- **Virtually Mapped Memory Management**
  This solution would be the perfect one. The advantages are clear: Fast, flexible, extendable. But that solution would require a big investment of time finding the correct solution.

- **No Segmentation, One Data Segment**
  Not applicable as there could no memory and instruction execution protection be provided. Else equal addresses would dereference the same space in RAM.

- **Segmentation, Overlapping Data Segments**
  Instruction execution protection could be applied but without any protection of kernel memory regions.

- **Mirroring Current Solution, Starting at 1MB Boundary**
  This means that both segments start at location 0x100000. First the user space, restricted to "small" size, is overlapped by the kernel space. The kernel is by linking relocated to the top of RAM area and must be installed by the kernel loader to this address. The kernel so could dereference the memory address directly. The disadvantage is obvious: On the one hand user programs normally grow faster than kernel extensions and on the other hand installing a version of Topsy i386 linked for one RAM equipment is either not runable (too less RAM installed) or wastes a huge amount of RAM as the space between the top end of the kernel area is not equal to the size of RAM available in the system built the image.

- **Creating a Kernel Alias to the User Segment**
  Obviously the virtually defined boundaries (refer to section D.2.3 on page 10) can be used several times. The kernel could install a descriptor in the GDT that points to the user space separately but with kernel data privileges. So both, kernel and user offsets would be equal. This would be the fastest solution at all as only the setting of the segment selectors would be required. The problem for this solution is found in the Topsy portable memory management system which is not prepared for this. Further all modern protected mode C compilers known to the developer of Topsy i386 do not support a multi segmented memory layout.
  This interesting solution possibility should point of further researches but could not be handled as part of the semester project.

Table D.2: Alternatives to the Implemented Address Adjustment

D.4 Remarks on ia32 Protecte Mode

The most accurate and detailed explanation of protected mode operation can be found in the book by Tom Shanley [1]. Nevertheless a short remark will be provided here:

The ia32 architecture starts in real mode, i.e. 20 address line are usable and no protection mechanism can be applied. The current mode is defined by bit zero of the control register zero (CR0). If this bit is set to one, the processor executes instructions in protected mode. The protected mode itself offers an instruction control depending on the four provided protection levels, a restriction of direct access to IO ports can be installed (which is applied per process), a restriction of memory access per process (depends on the segments installed in the GDT or LDT), and the default operand and opertor size, i.e. a segment in protected mode can be any combination of 16b or 32b code, resp. 16b and 32b data size. A virtually mapped memory management is supported by the MMU.
Summary of ia32 Protected Mode:

- Protection Levels
- Instruction Control
- IO Port Control
- Memory Access Control
- Operator, Operand Size Definition
- Virtually Mapped Memory Management

Instruction Control depends on the current protection level of the executing process. IO Port Control is defined by a optionally set bitmap of accessible IO Ports in the Task State Segment. Memory Access Control and Operator/Operand Size Definition depend on the segment declaration in the GDT or LDT. Virtually Mapped Memory Management is enabled by setting bit 31 in CR0.

D.4.1 Topsy i386 Implementation

Topsy i386 runs in a segmented memory model with two protection levels: kernel at protection level 0 and user at protection level 3. Both levels are set to operate in fully 32b mode, i.e. the default operand and operator size is set to 32b. Memory Access Control is granted by the two not overlapping memory spaces, resp. by the two protection levels applied to the memory spaces. IO Port Access is restricted to protection level 0. In version 1.0 of Topsy i386 no Virtually Mapped Memory Management supported by the ia32 MMU is implemented.

Bibliography

[1] Protected Mode Software Architecture by Tom Shanley, Addison Wesley 1996

[2] Microsoft's 80386/80486 Programming Guide by Ross P. Nelson, 2nd Edition, Microsoft Press 1991

[3] PC Hardware Buch by Hans-Peter Messmer, 5th Edition, Addison Wesley 1998

[4] PC intern 3.0, Systemprogrammierung by Michael Tischer, 1st Edition, Data Becker 1992

[5] Developing Your Own 32-Bit Operating System by Richard A. Burgess, 1st Edition, Sams Publishing 1995

[6] Dissecting DOS by Michael Podanoffsky, Addison Wesley 1995

[7] The Design of the UNIX Operating System by Maurice J. Bach, Prentice Hall 1986

[8] The Magic Garden Explained, The Internals of UNIX System V Release 4 by Berny Goodheart and James Cox, Prentice Hall 1994

[9] UNIX Internals, A Practical Approach by Steve D Pate, Addison Wesley 1996

[10] An Introduction to Berkeley UNIX and ANSI C by Jack Hodges, Prentice Hall 1995

[11] Operating Systems, Design And Implementation by Andrew S. Tannenbaum, Prentice Hall 1987

[12] The Basic Kernel Source Code Secrets by William Frederick Jolitz and Lynne Greer Jolitz, Peer-to-Peer Communications Inc. 1996

[13] Operating Systems, A Design Oriented Approach by Charles Crowley, Irwin 1997

[14] Linux-Kernel-Programmierung by Michael Beck et al., 3rd Edition, Addison Wesley 1995

[15] IBM Personal System/2 and Personal Computer BIOS Interface Technical Reference IBM 1987

[16] IBM Personal System/2 Hardware Interface Technical Reference IBM 1988

D.5 Acronyms Used During the Documentation

The acronyms used during the documentation of Topsy i386 are listed in the followin table:

IBM	International Business Machine Coporation
VGA	Video Graphics Adapter
GDT	Global Descriptor Table
LDT	Local Descriptor Table
IDT	Interrupt Descriptor Table
TSS	Task State Segment
GNU	GNU is Not UNIX
PC	Personal Computer
b	bit
B	Byte (=8b)
KB	Kilo Byte (=1024B)
MB	Mega Byte (=1024KB)
GB	Giga Byte (=1024MB)
ASCII	American Standard Code for Information Interchange
MMU	Memory Management Unit
PIC	Programmable Interrupt Controller
PIT	Programmablr Intervall Timer i8253/i8254
VSM	Verband Schweizerischer Maschinenindustrien
TASM	Turbo Assembler by Borland International Co.

Given Problem

Semester Thesis of Lukas Ruf

April 1998 – July 1998

Computer Engineering and Networks Laboratory, ETH Zurich
Supervisor: George Fankhauser
Professor: Bernhard Plattner

Appendix E

Given Problem

E.1 Original Problem

Semesterarbeit fuer	Herrn Lukas Ruf
Aufgabenstellung:	Prof. Dr. B. Plattner, George Fankhauser
Thema:	Topsy i386
Beginn der Arbeit:	27.3.1998
Abgabetermin:	3.7. 1998
Betreuung:	George Fankhauser, Marcus Brunner
Arbeitsplatz:	@home
Umgebung:	PC, Gnu-Tools, Topsy Source

E.1.1 Einleitung

Topsy ist ein portables micro-kernel Betriebssystem, das am TIK fuer den Unterricht entworfen wurde. In der ersten Version wurde es fuer die Familie der 32-bit MIPS Prozessoren gebaut. Es zeichnet sich durch eine saubere Struktur, eine hohe Portabilitaet (Trennung des Systems in hardware-abhaengige und -unabhaengige Module) und eine gute Dokumentation [1] aus.

Weitere Dokumentation ueber Topsy ist unter http://www.tik.ee.ethz.ch/ topsy verfugbar.

E.1.2 Aufgabenstellung

Obwohl Topsy auf einem MIPS-Simulator auf Java Virtual Machines und somit auf fast allen Rechnern laeuft, ist eine wichtige Motivation dieser Aufgabe die Verbreitung des Systems auf guenstiger und handelsueblicher Hardware (off-the-shelf). Dies sollte wiederum andere Studenten dazu anregen, mit dem System zu experimentieren. Im weiteren eroeffnet das grosse Angebot an Peripherie-Karten fuer den PC neue Moeglichkeiten (z.B. im Netzwerk-Bereich).

Die hellgrauen Module sind fuer Topsy i386 zu implementieren

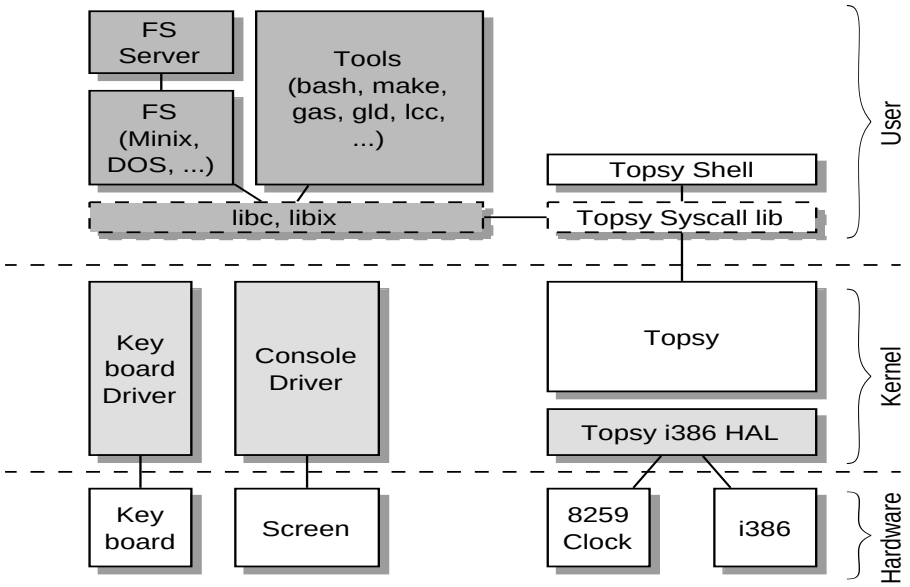

Figure E.1: Layered Model of Topsy

In dieser Arbeit sollen die Kernel-Teile implementiert werden, die neu fuer die PC Architektur geschrieben werden muessen. Dies sind im speziellen der intel-HAL und die wichtigsten Treiber fuer den PC (der hellgraue Teil des Diagramms).

Die Arbeit beschraenkt sich auf diese Kernel-relevanten Teile. Es ist dabei mit aeusserster Sorgfalt vorzugehen um eine moglichst fehlerfreie und stabile Basis zu erarbeiten.

E.1.3 Ziele

Die Portabilitaet von Topsy wird durch Implementation gezeigt. Ein spezieller Aspekt ist dabei die Einfuehrung eines dynamischen Laders.

Topsy laeuft auch auf handelsueblichen PCs ab intel 80386 Prozessoren. Dazu gehoeren folgende Treiber, die ins System integriert werden sollen:

- Clock (preemption)
- Console (out)
- Keyboard (in)

Die Treiber sollen den Mechanismus des dynamischen Laders verwenden.

E.1.4 Vorgehen

Machen Sie sich sowohl mit dem Betriebssystem Topsy wie auch mit der Zielplattform (Prozessor, Peripherie) vertraut.

Stellen Sie sich die notwendigen Tools auf einer Hostumgebung (z.B. Linux) zusammen. Dazu gehoeren gcc, gas, gld, objcopy (alle i386 oder cross-i386) und bash, gmake, java (plattformunabhaengig).

Es ist eine Boot-Strategie zu entwickeln. Es darf mit bestehenden Loadern und Boot-Strap-Tools gearbeitet werden, diese sollten aber frei und in Source verfuegbar sein.

Studieren Sie die HAL-Schnittstelle von Topsy und implementieren Sie den i386-HAL. Aehnliche Routinen koennen als Hilfstellung auch in anderen (protected mode) Betriebssystemen wie Linux oder Mach fuer i386 gefunden werden.

Fuer den PC muessen die 3 wichtigsten Treiber neu geschrieben werden: Clock, Keyboard, Console.

Compilieren und testen Sie das neue Topsy i386 erst mit ausgeschalteten Interrupts. Bootet das System wie gewohnt, koennen auch die Treiber getestet werden. Adaptieren Sie das Test-Tool Crashme um das neue System einem intensiveren Test zu unterziehen.

E.1.5 Bemerkungen

Mit dem Betreuer sind woechentliche Sitzungen zu vereinbaren. In diesen Sitzungen soll der Student muendlich ueber den Fortgang der Arbeit berichten und anstehende Probleme diskutieren.

Am Ende der zweiten Woche ist ein Zeitplan fuer den Ablauf der Arbeit vorzulegen und mit dem Betreuer abzustimmen.

Am Ende des zweiten Monats der Arbeit soll ein kurzer schriftlicher Zwischenbericht abgegeben werden, der ueber den Stand der Arbeit Auskunft gibt.

Am Ende der zweiten Woche ist ein Zeitplan fuer den Ablauf der Arbeit sowie eine schriftliche Spezifikation der Arbeit vorzulegen und mit dem Betreuer abzustimmen.

Bereits vorhandene Software kann uebernommen und gegebenenfalls angepasst werden.

Die Dokumentation ist mit dem Textverarbeitungsprogramm "FrameMaker" zu erstellen.

E.1.6 Ergebnisse der Arbeit

Neben einem muendlichen Vortrag von 20 Minuten Dauer im Rahmen des Fachseminars Kommunikationssysteme sind die folgenden schriftlichen Unterlagen abzugeben:

- Ein kurzer Bericht. Dieser enthaelt eine Darstellung der Problematik, eine Beschreibung der untersuchten Entwurfsalternativen, eine Begruendung fuer die getroffenen Entwurfsentscheidungen, sowie eine Auflistung der geloesten und ungeloesten Probleme. Eine kritische Wuerdigung der gestellten Aufgabe und des vereinbarten Zeitplanes rundet den Bericht ab (in vierfacher Ausfuehrung). Der Bericht soll zudem zwei neue Kapitel des Topsy Manuals [1] (Appendix C und D) enthalten, die eine Anleitung fuer weitere Portierungen geben ("A Guide to Porting Topsy"), und die vorliegende Portierung beschreiben ("The i386 Port"). Beide Appendices sind auf Englisch zu schreiben.

- Ein Handbuch zum fertigen System bestehend aus Systemuebersicht, Implementationsbeschreibung, Beschreibung der Programm- und Datenstrukturen sowie Hinweise zur Portierung der Programme.

- Eine Sammlung aller zum System gehoerenden Programme.

- Die vorhandenen Testunterlagen und -programme.

- Eine englischsprachige Zusammenfassung von 1 bis 2 Seiten, die einem Aussenstehenden einen schnellen Uberblick ueber die Arbeit gestattet. Die Zusammenfassung ist wie folgt zu gliedern: (1) Introduction, (2) Aims & Goals, (3) Results, (4) Further Work.

E.1.7 Literatur

- G. Fankhauser, C. Conrad, E. Zitzler and B. Plattner., Topsy - A Teachable Operating System, TIK, 1997

- Topsy home page: http://www.tik.ee.ethz.ch/ topsy