Porting the Unix v7 kernel to the RISC Processor Eco32

Author: Dennis Kuhn in WS 2002/03

17th February 2003

Abstract

This document is a diploma work which describes the important parts of porting the Unix v7 kernel to the Eco32 RISC processor. The first steps therefore are to introduce the two different processor architectures and describe the ansifcation of the code . Short notice is given to some device drivers and the magic of process switching is disclosed, followed by an abstract section about the new memory management.

Foreword

First of all, I would like to thank all the people whose participation helped towards this work. Special thanks goes primarily to Prof. Dr.-Ing. Hellwig Geisse, who gave me his full assistance throughout. From the very beginning, he lent an open ear to my unusual thoughts, helping me to organise them. Dr. Geisse spent a lot of time in rapidly developing the necessary simulator, compiler, assembler and linker. Furthermore he extracted the original sources and constructed a file-system on his virtual disk. I also wish to thank all my friends who continued visiting me when I didn't call them. A very big thanks to Mr. John Lions for his great book. Last but not least a great thank to my mother, Gordon Krüger and mostly to Jacqueline Parnow, whose involvement led to this document becoming readable.

Contents

Chapter 1

Introduction :

The target of the project was to port an operating system, to an virtual RISC processor. The doors should be opened to an operating system which could relative easily ported to similar processor architectures[1]. Why Unix v7 ?

Most parts are written in C and are free (nowadays),

Excellent documentation of the very similar Unix v6 code[Lions]

Multiuser and multiprocessing capabilities,

Operating System which is concise enough to port it within a small time-frame.

Unix v7 was written in the 70s by Dennis Ritchie and Ken Thompson at the Bell laboratories. It was designed for the PDP-11, a nice room filling computer, build and often used in the 70s. Its home were big dark cellars of universities. Nowadays the Unix version 1,2,3,4,5,6,7 versions distributed under the Caldera license, and are readily available. Before the Caldera license, Unix was only available for some universities. Since the 70s Unix has further developed and is widely distributed. The principles setup in the earlier versions have survived the long period and are still effective. In 1977 John Lions finished his work on the book "Lions' Commentary on UNIX 6th Edition". This book explains nearly every line code in the operating system. With the help of this book it was possible to understand the more and less difficult parts of Unix.

The target system Eco32 is a virtual processor[Geisse]. Dr. Geisse wrote all necessary programs to port Unix to his processor, these being the

Extractor to extract the original sources from tape copies,

Eco32-SIM The processor itself,

LCC-back-end ANSI C-compiler,

[1] The vision is an operating system for the mmix processor of Donald E. Knuth

ASLD Assembler and linker,

disk-driver The block device driver to talk with the virtual disk,

disk/FS A virtual disc containing the original Unix FS with / and /usr ,and a swap partition,

1.1 How to read this document

The last chapter of the document is a .glossary explaining some of the words used in this document.

Such SYMBOLS reference real existing function-names, programs or similar matters.

Aliases serve to shorten any long name like *P1* instead of processes 1. Reference to all discussed files, functions and programs can be found in the index, as well as many important words. For technical reasons, all underscores '_' are replaced by a hyphen '-' within the index !

1.2 The development environment

The development platform is a x86 Linus system, simulating the target system Eco32. To compile, assemble and link, the Unix sources, the LCC in combination with the ASLD were used.

I used Anjuta to work with the sources. Anjuta is a free, convenient and fast integrated-development-environment. The primary features I used are the symbol-tables of all function, structs, macros, etc. A lot of time saved the auto-completion while writing source. Especially convenient is the cooperation with the grep and lcc output. This means if the lcc gave out error lines containing a line number, it is possible to jump directly to the certain line of code. When searching patterns in the source trees using "GREP -N <PATTERN> */*.C */*.H"[2], it is possible to jump to the found lines.

This document is written using LyX[LyX], which enables one to write a LaTeX document without any knowledge about TeX and LaTeX. The graphics are clicked with XFig[XFig], which of course is free software, too.

From the commercial point of view, Dr. Geisse was the custodian, and customer of this diploma work.

1.3 Steps of porting

The porting of Unix v7 is divided into several technical parts, , the first workpiece being getting the code compiled at the target platform. This step is called ansification, as the aim is to convert the K&R C code into ANSI conform code. When the code is able to compile and link, the machine depended files have to be re-implemented.

The new memory management had to be redesigned completely, because the dimensions and way of memory management differed too much to port it. I decided to implement a pseudo "Two-levels page-tables" memory management.

After that the sys-calls, as system entries, are described and ported. Thus it is possible to write some small user[3] level test applications. The user level program is put on the disk at /ETC/INIT, so it plays the role of the init process. This document closes up with the detailed explanation of the sample init-process. The very last but not least section describes in short how to let the PDP-11 simulator run.

[2] this command line searches in all folders below in all source and header files for the specified pattern. The lines are given out with filename and line-number (caused by the parameter -N)

[3] User level means kernel mode, and has nothing to do with user and root ! The user level programs run as root.

Chapter 2

The Hardware

The first big differences between the two machines are the time periods in which the two processors existed and maybe would exist. The PDP-11 is a big power consuming computer from the 70s versus the Eco32, a simulated RISC processor at the beginning of the 21th millennium. Both processors have two different run modes. On the one hand, one with full access to hardware and memory, called kernel mode and on the other hand, one restricted to only access to a virtual address space for the user. Some special assembler instructions are only executable from kernel mode, i.E. instructions to manipulate the processor status word or to modify the virtual address mapping.

2.1 PDP-11

The PDP-11 is a 16 bit processor with little endianes. It comes up with 8 general purpose registers and 400 hardwired CISC instructions. Only R0 up to R5 are really general purpose registers. R6 is used for stack addressing, this register is existing in two varieties, one being accessible in user mode and the other in kernel mode; the switching is automatically made by the processor on exception entrance. R7 is used as program counter, which points to the next instruction to be executed.

2.1.1 Virtual memory mapping

The PDP-11 has a separate virtual mapping for user- and kernel-mode. Every virtual address space is divided into 8 segments on the PDP-11/40 and 16 at the PDP-11/70. The PDP-11/70 has separate mapping for data and instruction space. We shall now just look at the PDP-11/40 architecture to keep it simple. Every of the 8 segments could have up to 127 blocks, one block has the size of 64 bytes. These 64 bytes are the core clicks, the smallest allocalable size of memory. Every segment is described through two registers, an address register, which contains the physical address, and a description register which contains information about the access permitions and the growing direction. Thus we came up with 32 registers on the PDP-11/40 and 64 on the PDP-11/70.

Phys. Address	Description
KDSA0	KDSD0
KDSA1	KDSD1
KDSA2	KDSD2
KDSA3	KDSD3
KDSA4	KDSD4
KDSA5	KDSD5
KDSA6	KDSD6
KDSA7	KDSD7

The table shows the 16 registers for **K**ernel **D**ata **S**pace mapping. The same register block exists for the user space mapping, simply called UDS[AD][1]. And finally user and kernel registers appears in an additional set for instruction space mapping called [UK]IS[AD] on the PDP-11/70.

[1] UDS[AD] stands for the combinations UDSA and UDSD.

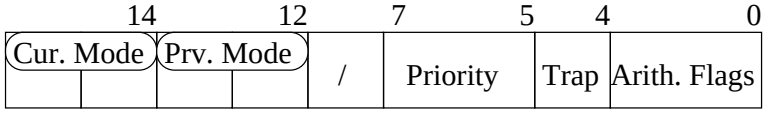
2.1.2 Exceptions

Whenever a trap or device interrupt occurs, R6 is switched to kernel R6. The contents of the current PSW and R7[2], are pushed on kernel stack. The previous mode bits are copied from the actual mode bits, and the actual mode bits are set to kernel mode. The new priority is set with a value from the interrupt vector table. On an exception return the R7- and the PSW-register are restored from stack. When returning in user mode, the hardware switches R6[3] to user R6.

2.1.3 Priorities and the processor status word

The priorities are increasing, which means if currently running on priority level 5, only interrupts with a priority level below can interrupt. The PDP-11 features 8 priority levels, but just priorities starting at 4 till 7 are implemented. Running at priority level 7 prohibits any interruption by a device interrupt. The trap bit indicates if a processor trap occurred.

Both mode bits set to 0 reflects kernel mode, and both set to 1 means user mode.



[2] R7 is the program-counter register on the PDP-11.
[3] R6 is the stack-pointer register.

2.2 Eco 32

The Eco32 is a virtual 32 bits processor with big endianes. It was developed since November 2002 by Mr. Hellwig Geisse. He created a RISC processor with 61 efficient instructions, holding 32+4 register. The register $2 to $29 are for general purpose. $0 always contains 0. $31 is the function return register which is filled by the JAL and JALR instruction. On an exception $30 is filled with the program counter. $1 is used by the assembler to hide the maximum of 16 bit constants from the programmer. Additionally there are four special registers, the first being the processor status word and the other three are used to communicate with TLB.

2.2.1 Virtual addressing hardware : TLB

The Eco32 also comes up with 64 addressing registers, but in a completely different way. It works with a TLB (**T**ranslation **L**ookaside **B**uffer) to map virtual into physical addresses. The TLB consists of 32 32-Bit register pairs, where the first register contains the virtual page start address and the corresponding register the physical frame address[4].

Thus, the programmer has full freedom in implementing a memory management. The programmer just has to feed the TLB with address pairs. If a virtual address isn't found in the TLB, a TLB-miss-exception is thrown by the processor. This exception has to be caught and solved by the operating system, which fills the TLB with the necessary mapping or removes the causing instruction.

Usually when adding a new entry into the TLB, a random old entry is overwritten. This could be fatal for mappings which have to be permanent, i.e the kernel stack mapping. So the TLB comes up with four fixed register pairs, which could only set explicitly and are not overwritten by randomised entries.

It is very important that one virtual address never occurs two times in the TLB, in that case the "TLB double hit exception" is thrown. When using real hardware, this could cause a TLB defect.

For TLB communication 3 special registers exist.

Special1 TLB Index register, used to write or query at a specific TLB index,

Special2 Virtual Page Address, contains the virtual frame start address,

Special3 Physical Frame Address, contains the physical start address of the mapped frame.

The registers could be written with :

MVTS <REGISTER CONTAINING THE VALUE>,<SPECIALREGISTERNUMBER>,

and read out with :

MVFS <REGISTER CONTAINING THE VALUE>,<SPECIALREGISTERNUMBER>.

To transfer values from the special registers into the TLB there are two instructions, TBWR and TBWI. The most common write instruction is TBWR, it writes the contents of the special register 2 and 3 into a random index. To fill the 4 fixed entries at the beginning of the TLB, the TBWI instruction has to be used. It writes the contents of the special registers 2 and 3 into the TLB index specified with special register 1. To read from the TLB, the wanted index has to be moved into special register 1 followed by a TBRI instruction. Now the wanted mapping appears in special register 2 and 3.

[4]Physical frame address = frame number << 12. 12 bits are used for offsets within a page, so a page has the size of 4 kByte.

2.2.2 Virtual addresses

A 32-Bit virtual address is divided into four parts. The least significant bits contain the offset within a virtual page. The following 10 bits contain the page number, in the page table specified by the 8 page directory bits. The 2 topmost bits divide the address space into kernel- and user-space. Furthermore the kernel space is divided into virtual- and physical-space.

The Spc bits have following meaning :

00 Virtual User space

10 Virtual kernel space

11 Physical address space

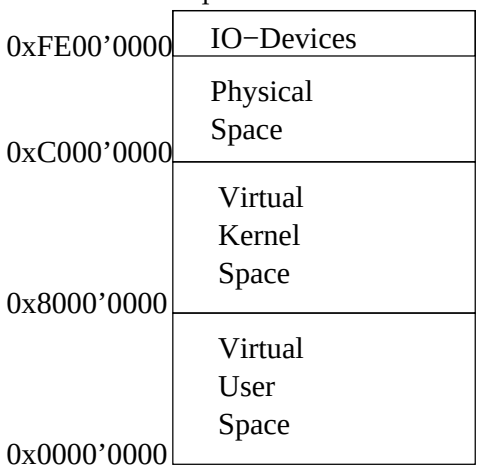
As the least 12 bits are used for the offset within a page, they are left 0 in the virtual address register. In the physical register the least three significant bits are used for permissions.

110 Read write permissions on frame,

100 Read only permission on frame,

2.2.3 Address spaces

The Eco32 address space is divided into four parts :



The physical space "maps" the physical available main memory, only the kernel can access the memory directly. The two virtual spaces are mapped by the TLB. Only the physical space is accessed directly. The most significant bit determines the accessibility in the two run modes. This means that a user program never could access a page in the kernel space. The top part of the address space is used to communicate with the connected hardware.

2.2.4 User / kernel mode

In order to work with the two modes, some bits in the processor status word are reserved. This bits can only be modified in kernel mode. On an interrupt the run mode is automatically set to kernel mode. To find out the run mode before the interrupt, two bits exists, the previous user mode bit, which contains the mode before the actual interrupt, and the old bit, which contains the mode before the last interrupt. The second stage is very important for the TLB refilling method, which should be implemented quicker, rather than only efficient. Corresponding to the user mode bits, there are 3 interrupt enable bits, which indicate the actual, previous and old interrupt enable state.

2.2.5 Exceptions

Exceptions are divided into two parts : device interrupts and traps. Interrupts are caused by the attached hardware like clock, disk and terminal. The processor causes traps on every wrong behaviour of the running code. The priority is increasing, this means that i.e. the disk has the interrupt request number (IRQ#) 8, while the clock has the IRQ# 13. When both interrupts occur simultaneously, the clock comes first. Above all interrupts, the traps are located. The most caused trap is the TLB miss. Interrupts can be enabled and disabled by the interrupt enable bit in the PSW. Traps can never be disabled, what should happen on a disabled TLB miss ? Following interrupts and traps exists :

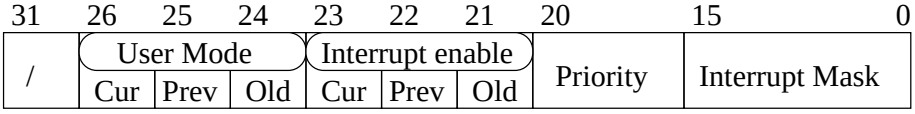
4	Term transmitter Interrupt
5	Term receiver Interrupt
8	Disc Interrupt
13	Timer Interrupt
—-	**Trap Border**
16	Bus address Exception
17	Bus Timeout Exception
18	Illegal instruction Exception
19	Privileged Exception
20	Trap Exception
21	Divide Exception
22	TLB miss Exception
23	TLB double hit Exception
24	Privileged address Exception
25	Write protection Exception

On exception entry the current PC is stored in $30, the user mode and interrupt enable bits are shifted down. The current mode and current interrupt enable bit are set to 0. The PC is set to the second word in physical memory. When returning from exception the PC is restored with the value in $30, the user mode and interrupt enable bits are shifted upwards.

2.2.6 Priorities and the processor status word

The current priority is set by disabling and enabling bits in the INTERRUPT MASK. To disable all interrupts, a 0 has to be put into the CURRENT INTERRUPT ENABLE BIT. I.e. if setting bit 13 to one, clock interrupts can occur if the CURRENT INTERRUPT ENABLE BIT is set to 1. The current IRQ# is reflected in the priority field.

The PSW is the first of the four special registers, so far it can only accessed via the kernel mode instructions MVFS and MVTS. The priority field is the only read only section, all other bits can be modified free by the kernel.



2.3 Common and differences

The only common points are kernel/user mode and the virtual mapping itself. Because of the big freedom on the Eco32 I decided to implement a memory mapping with two levels of page-tables.

Chapter 3

Ansification

The ansification runs through several stages. The biggest technical difference, is the different function declaration, followed by the fact, that most functions were not declared anywhere. Another aspect is, that in K&R C, a variable could be defined at many positions, the linker simply put all same symbols together. In ANSI C this is strictly forbidden, a variable is only allowed to be defined one time, and has to be declared using the EXTERN keyword on every other location of usage. The function declaration transformation and the header file creation is completely automated. First I will guide you through the differences and the algorithms on how to solve the partial problems. Thereafter I shall explain the global ansification algorithm,where all parts play together.

3.1 Function argument declarations

In Kerney and Ritchie C (K&R C), only the variable identifier were written into the function header. The types of the parameter are found between, the closing), of the argument list, and the opening {, of the function body. The variable type only has to occur if it isn't an integer variable, the same applies for return parameter. To make it clear, a sample :

K&R C:

```
MAIN( ARGC, ARGV )
CHAR * ARGV[];
{ IF( ARGC == 0 ) RETURN (-1);
RETURN; }
```

ANSI C:

```
INT MAIN( INT ARGC, CHAR * ARGV[] )
{ IF( ARGC == 0 ) RETURN (-1);
RETURN 0; }
```

The return argument is slightly more complex, in K&R C it is possible to return something or not, in one function. This results in putting RETURN 0 to the locations where nothing returned within a function which sometimes has a return value. The second problem with the return arguments is, that if the return type is specified, it is put in the line above.

To convert K&R source files into ANSI conform source and header files I implemented ANSI_F. It converts K&R function definitions in ANSI definitions. ANSI_F is also capable of creating header files , by simply replacing the function body through a ';'.

ANSI_F always works on a single source file, whether it creates an ANSI conform source file or a header file. Preprocessor directives are put through directly, if not creating a header file. The file is read in line by line, remembering the last line. Two kinds of blocks are from interest, comments and functions. Comments have to be realized, because they could contain everything. Their content is simply put though.

Functions are detected by the two function parentheses '(' and ')'. These parentheses also appear on function calls and functions as arguments. When detecting a function, the last line is saved as possible return argument. The current line, containing the function header, is divided into function identifier and argument identifiers. The next step is to get all argument declarations between the closing ')', of the argument list, and the opening '{' of the function body. Now we "simply" have to match all argument identifiers with the argument declarations.Slight fiddling is necessary in the matching, because fields and function pointers are somewhat unusual.

At this point just the return type is missing, The function body is read in char by char, counting the block depth, to find the end. Finally we take a look on the previous saved possible return value. If it contains something valid[1], it is taken as return type. Elsewhere the body is checked for RETURN statements, if it only contains a "RETURN;", VOID is taken as return type, elsewhere INT. Now all informations about the function are collected. The header is printed out, followed by the body or a semicolon, when generating a header file.

All other lines which are not identified as function, i.E. structs and similar

[1]Not a comment, whitespace or semicolon

things, are suppressed when writing a header file, or put through for source files.

Because application ANSI_F doesn't know much about C, it is not usable for all ansification processes. It strongly depends on the style found in the kernel sources.

3.2 Function declarations

Most functions in K&R C are used undeclared, so the most source file doesn't have a corresponding header file. To simplify this, I automatically created a header file for every source file containing all its declarations. The second step is to find out which source file has to include which header file. It is easy to see that this is a complex problem. Because if we have 2 source files, A.C and B.C. We create 2 header files, A.H and B.H.We then have to search for each of these two files in two header files for wanted declarations. This means we came up with $2^2 = 4$ searches. If we have 4 source files, we create 4 header files, and have to look for every source file in 4 header files. Therefor we have $4^2 = 16$ searches. Our source tree contains more than 60 files, this results in more than $60^2 = 3600$ searches. But no worry, the computer has to search.

A source file is compiled with the GCC to find out the missing declarations. The resulting output is filtered for "IMPLICIT DECLARATION OF FUNCTION", the 7th word in the filtered lines contain the missing function identifier. All missing declarations are sorted and redundant entries removed. For tidying up, all blanks are removed. The described word is done by the script GET-MISSINGDECLARATIONS. The heart of the script is the following command line :

```
GCC -WALL -C $F 2>&1 \
| GREP "IMPLICIT DECLARATION OF FUNCTION" \
| AWK '{ PRINT $7; }' \
| SORT | UNIQ | FILTER.E \' \' >$F.MISS
```

$F is filled with the name of the actual processed file. Every ansified source file is processed in that way. After that, the complex part of finding the missing function identifier in a header file follows.

The resolving part is made by the script RESOLVDEPENDENCIES. It goes through every line of a .MISS file, and searches the missing function identifier in all passed header files. If one or more header files contain the function identifier, the first header file name is enveloped in a #INCLUDE "" and attached to the corresponding .H.DEP file. If no header file contains the missing function identifier, the identifier is attached to the corresponding .H.DEP.MISS file. The main part of the RESOLVDEPENDENCIES resembles the following. $2 contains all header filenames to search in, $f is set to the actual .MISS file name, $h is the generated .H.DEP filename :

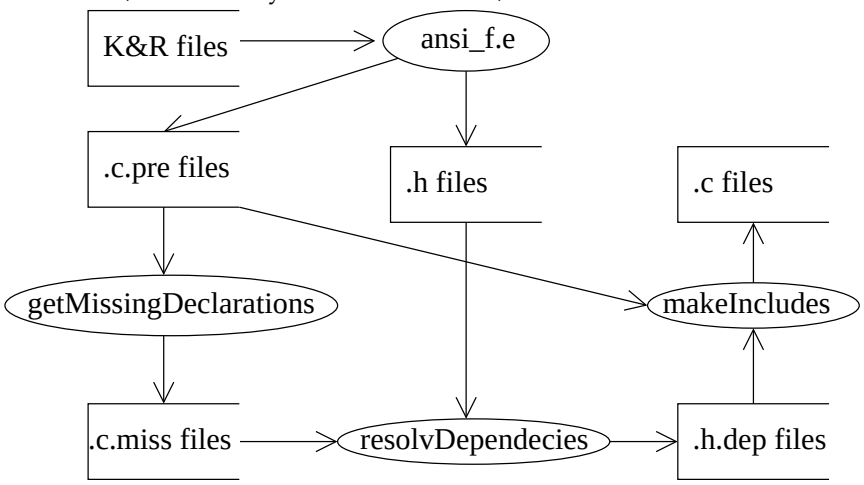
```
1: CAT $F |  WHILE   READ FUNC DO
2:           HF=     $( GREP " $FUNC(" $2 | GREP -V ".H: \*" | HEAD -N 1 );
3:           HF=     $(HF/:*};
4:           IF      [ -Z "$HF" ] ; THEN
5:                   ECHO "$FUNC" >>$F.H.DEP.MISS
6:           ELSE
7:                   ECHO -E "#INCLUDE \"$HF\"";
8:           FI;
9: DONE |    sort |  UNIQ >>$H;
```

In line 2 HF is something like "dev/et.h:void putc(char c)", so line 2 has

the job to remove the ':' and the function identifier, quasi everything behind the ':'. Line 7 writes the enveloped header file name to standard out, which is caught in line 9 an put through SORT and UNIQ, to include every file only once.

3.3 Complete ansification process

The ansification is controlled by the script ANSICOMPL or ANSICOMPLALL, the first script ansifices all files in the current directory, while the second converts all files in both source directories SYS and DEV. One additional script comes into the game, MAKEINCLUDES. It extracts all includes from the .C.PRE file into the .C file and attaches the includes from the .H.DEP file. Finally the content of the .C.PRE file, without any include directives, is attached to the .C file.



The script has two loops. The first puts all K&R files (.C.OLD) through the ANSI_F filter. The filter generates the .C.PRE files, which has ANSI conform function definitions on the one hand, and on the other the .H files containing all declarations. In the second loop, the missing declarations are solved, and a .C file generated which contains all necessary include directives.

At this point, the handmade part follows. The ansified code still contains unresolved calls to assembler functions, and moreover there are the multiple defined variables. First I created the header file CONF/MCH.H, which contains the declarations of the functions implemented in CONF/MCH.S. Somewhat more time-consuming is the collection of all global used variables into the file H/GLOBALS.C. The multiple definitions are converted into declarations by adding the EXTERN keyword. For the first complete linkings, I implemented empty body for the functions declared in CONF/MCH.H.

Chapter 4

System calls

The system-calls, further called *syscalls*, are the entry for user programs into the kernel. The technical depth is explained in the next subsections. *Syscalls* are used whenever an operating system service is needed by a user-program. There are 55 *syscalls* on operating system side, which are mapped into 51 syscalls on user side, as not all are implemented. Some syscalls like EXECE have several implementations on user-side.

In the original implementation, every *syscall* was put into a separate file. As most of the *syscalls* act in the same way, I put all implementations in the file SYS/SYSCALL.S. The system calls could be divided into three classes :

1. returning return-value from *syscall* or error-code on error,
2. returning 0 or error-code on error,
3. individuals like fork, which have their very own implementation with different return locations or similar stuff.

All syscalls matching in the first two groups are automated. Every syscall has its own entry-point where it sets the *syscall*-number. From there it jumps to the common code. All arguments are passed via the register to the kernel causing a trap. If the *syscall* fails, register $4 is set to nonzero[1]. If $4 is not 0, the common error-exit-code CERROR is called, it sets the global variable ERRNO to the returned error code and sets $2[2] to (-1). For those syscalls which return 0 upon success the register $2 is set to zero, if the syscall indicates no failure.

The many individual syscalls are implemented, so far it was possible, in the same manner like the other syscalls.

[1] in the PDP-11 implementation, the error-flag in the PSW was set.
[2] $2 is the first return-value register used by the LCC.

4.1 System calls argument passing

Passing arguments from user-mode to kernel-mode is the main aspect of the *syscall* implementation. For a start, we observe the syscall-number as a usual argument. Unix features two different kinds of arguments passing, directly and indirect. Directly means, the arguments are put behind the TRAP instruction within the code. Indirectly means, that an argument-pointer is put behind the syscall. Lets look at the two examples (TRAP_INSTR is a constant containing the binary trap instruction, <Syscall-Number> has to be replaced with the wanted number) :

Indirect *syscall* :

```
.CODE
LDHI $8,TRAP_INSTR
OR $8,$8,<SYSCALL-NUMBER>
STW $8,$0,SYSCALLARGS
TRAP
.WORD SYSCALLARGS
.DATA
SYSCALLARGS: .WORD 0X0
```

Direct *syscall :*

```
.code
trap <Syscall-Number>
```

The trap handler sets the pointer A to the program-counter of the interrupted user program[3]. The word at A is fetched and compared with the trap-instruction. If they are equal, A is set with the word following the trap instruction. The word referenced through A is fetched and split into *syscall*-number and *syscall*-instruction. The syscall-instruction is compared with the trap-instruction, and if they are not equal an error is caused. Faced to this fact, it is required that the trap-instruction occurs in the data-segment, on indirect *syscalls*. A is moved one word forward.

The two different forms of passing the syscall-number is very important for assembler which does not support arguments within the TRAP-instruction.

To pass the remaining arguments two means are featured : via registers and via the argument pointer. In the file SYS/SYSENT.C the count of arguments and the count of arguments passed in registers is configured, for every *syscall*-handler. The arguments which are not passed in the registers are fetched starting at A. In case of mixed transfers, first the register arguments are fetched followed by the arguments referenced through A.

Finally the user program-counter is moved to the next valid instruction behind the TRAP instruction.

[3]The PC points to the instruction actual executed. In case of a trap, the trap occurs within the TRAP instruction.

4.2 The SYSCALL.S implementation

Because the LCC takes 4 registers-arguments and no *syscall* requires more, all arguments are passed via the register. Therefore no particular attention needs to be given to the count of passed arguments. The only remaining difference is the return value which is solved through two different common code-sections. The following example shows the two different standard-*syscalls*. READ returns the count of read bytes or (-1) on error, while WRITE returns 0 on success and (-1) on error. Both error returning is done through CERROR.

The first two code-pieces show the unique entry which sets the ***s**yscall-**n**umber.*

```
read:
add $8,$0,(TRAP_INSTR | SN_read)
stw $8,$0, SysCallArgs
j SysCallRetA
```

```
write:
add $8,$0,(TRAP_INSTR | SN_write)
stw $8,$0, SysCallArgs
j SysCallRet0
```

The next two pieces are the common code, for returning "syscall-return-value" or "0" on the success. On entry the function-return address $31 is pushed on stack followed by the TRAP instruction. When the trap returns, the function-return address is popped from stack. If an error occurred, the return is made by cerror, Elsewhere returning to the function-caller.

```
SysCallRetA:
sub $29,$29,4
stw $31,$29,0
trap
.word SysCallArgs
ldw $31,$29,0
add $29,$29,4
beq $4,$0, SysCallRetA1
j cerror
SysCallRetA1:
jr $31
```

```
SysCallRet0:
sub $29,$29,4
stw $31,$29,0
trap
.word SysCallArgs
ldw $31,$29,0
add $29,$29,4
beq $4,$0, SysCallRet01
j cerror
SysCallRet01:
add $2,$0,0
jr $31
```

On error the following common code is executed. It stores the *syscall* error-code in the global variable ERRNO and returns (-1).

```
cerror:
stw $2,$0,errno
sub $2,$0,1
jr $31
```

Within the data-segment only one word is needed for all syscall-numbers. Because this *syscall*-library is only a dummy, the errno variable is located here, too.

```
.data
.align 4
SysCallArgs:
.word TRAP_INSTR
errno:
.word 0x0
```

Chapter 5

Device drivers

Nearly half of the sources are device drivers. I only ported the memory device driver, because our simulated hardware only have a disk and a terminal. I only ported the important function VOID PUTCHAR(CHAR C) in the terminal driver DEV/ET.C (**E**co32 **T**erminal). PUTCHAR is used to debug the kernel and called on kernel panic attacks.

The memory device driver is used to write directly into the memory. For physical address no mapping is needed, so the requested address is converted into an absolute address. This address is accessed by SUBYTE or FUBYTE. In the original PDP-11 implementation, the physical address had to be mapped. Therefore, it had to be split into page- and offset-part. The page-part were put into an address register, which contents were saved before. After that the offset could be accessed at the mapped address.

In the case of the disk driver, I was fortunate and obtained the device driver from the manufacturer ;-) The remaining work is to make the correct entries into the block-device-switch in CONF/C.C, and setup the correct settings for swap start (SWPLO) and root device.

Chapter 6

Process switching

This chapter is necessary in understanding the issues in the memory management. Let us assume that process 0 is always existing, and in our environment there are additionally the processes 1 and 2.First we need to find out how a process is represented, and then observe how SWTCH switches through the three processes. Finally the magic is disclosed with a look at SAVE and RESUME which does the switch.

6.1 Process representation

A process is divided into several logical parts :

proc-struct contains informations about a process are not allowed to be swapped out, most important is the field P_ADDR, which points to the process in core or swap.

user-struct contains informations which are only from interested when process is in core.

U-Area contains mapping informations, user struct and kernel stack.

user-frames all frames mapped between virtual 0 and the kernel address space. They contains user- code-, data- and stack-segment.

All proc-structs are located in the array PROC in the kernel data segment. The UArea appears somewhere in the virtual kernel space. At one time, only one U-Area is mapped. So far the U-Area is the most important aspect when switching a process. It should be clear, that the kernel does not have one stack, but indeed the kernel has one stack per process. Thus the kernel always runs in a certain user-context.

6.2 Process switcher SWTCH

The process switching is done by the function SWTCH implemented in SYS/SLP.C. A very important issue is, that process switching is always done to process 0 or from process 0 to another process. Because of that, process 0 is called the *scheduler*. In this respect a switch from process 1 (further called *init*) to process 2 means, switching from *init* to the *scheduler* and finally to process 2.

We assume a running *init*, which has a higher priority then process 2. At the timer interrupt, execution is set to the entrance of CALL. It switches the stack to kernel stack. From this moment on, *init* runs in kernel mode. Now the prime task of CALL begins, it stores all registers on the kernel stack and calls the corresponding interrupt service routine, CLOCK. We assume that CLOCK decides that is time to jab the *scheduler*, so it rises the RUNRUN flag. When returning to CALL, the RUNRUN flag is detected and causes a "Giveup CPU-Trap" by calling TRAP instead of returning to *init*. From TRAP the call goes through QSWTCH to SWTCH.

First workpiece of SWTCH, is to switch into the *scheduler*s environment. This is done by calling SAVE, to save the state of *init*. After that, the RESUME call brings back the *scheduler*. He wakes up at the last SAVE call position, which was within the SWTCH function. The process with the lowest priority[1] is fetched from the run queue, and is resumed. Process 2 wakes up at the previous SAVE call in SWTCH and leaves SWTCH immediately. CALL restores the register contents from kernel stack[2] and jumps back into user mode.

[1] This is the process priority and has nothing to do with the priority levels of exceptions.
[2] including the user mode stack pointer.

6.3 SAVE

A SAVE call looks like that : "IF(SAVE(U.U_RSAV)) RETURN;". This indicates, that SAVE has two ways to return. It is not an error code, which causes the caller of SAVE to return ! It is a RESUME which returns there. If calling SAVE it saves the state of the process, which means all processor registers. At this point the kernel-stack contains the call-stack of getting into SAVE. When the process is later resumed , it will go the same way back. Once back to the call of SAVE, the save returns with a 0, so it reflects that the state of the process is locked and we can safely switch to another process.

The argument passed to SAVE is the location where the register should be saved. Three areas exists there :

u_qsav is taken for quits and interrupts (don't think about it),

u_rsav is used for process **S**witching (stack exchanging),

u_ssav contain the register content while swapping the process,

6.4 RESUME

RESUME is called with the process-pointer P_ADDR and the location of the saved registers within the U-Area. First step is to **map** the new U-Area with help of the P_ADDR. After that the saved registers are **restored** from the new mapped U-Area. The last two steps are the process switch ! In this respect, P_ADDR key to a process. It is the only pointer referencing a process, while it is not running.

Chapter 7

New memory management

This chapter explains the new memory-management I have implemented. This does not describe any non-existing algorithms of demand swapping or similar things. This describes the technical matter, how to handle the virtual mappings. Firstly, the original memory management is briefly described. The underlying concept is discussed in three stages : in general, specialised to our needs and finally an example. The interface to the new memory-management is described in the sections behind the concept. This chapter closes with a further detailed look at the process of swapping.

7.1 The original memory management

The prime assumption is : "**A process image, is a continuous block of physical memory frames**". This fact makes it very easy to copy the image on the one hand, and on the other it makes some effort to change data- or stack-segment size. E.g. if the data-segment is increased by n frames, the operating system checks if enough core is available to copy the image to a bigger core frames block. In the case of increasing data segment, the stack frames have to be moved to the end of the new image.

According to the virtual addressing registers, software prototypes exists for every process. These prototypes contain the offsets within the image of the segments. The content of the prototypes is, increased by the load address, put into the segmentation registers. This means, that a physical address never occurs in the software prototypes. The physical load address is stored in a variable called P_ADDR in the corresponding process struct.

Three functions are used to modify the process image, ESTABUR, EXPAND and SUREG. The image size is modified[1] by EXPAND, the caller has to move the user stack to the correct new location. If EXPAND can't get enough core, it arranges the process to become swapped out. If it allocates enough core memory, the process image is copied to the new location. ESTABURs task is to setup the offsets in the software prototypes, in that way the three pseudo segments are realized. Finally the real mapping settings are done by SUREG, which copies the contents of the software prototypes into the hardware register, increased by the process load address.

[1] allocating or freeing of core memory

7.2 The concept : pseudo “Two-levels page-tables”

7.2.1 General assumption and concept

The new prime assumption is : **“core memory is allocated only frame by frame, the mapping does the continuous”**. This results in easily finding out the count of free memory. If you want to allocate 2 continuous memory frames, and only the half core memory is used, it should be allocalable. It is, however, possible, that only every second frame is allocated. What now ?

- Moving two frames together - Hard to find who maps them.
- Questions for available memory are only answered with a given frame count - strange way.

To this first complete different assumption, a complete new memory management is added. In general it is the concept of “Two-levels page-tables”. The meaning is that a page-table contains n_{te} physical frame addresses, and a page-directory references n_{de} page-tables. The third player in the game is the offset within a page frame.

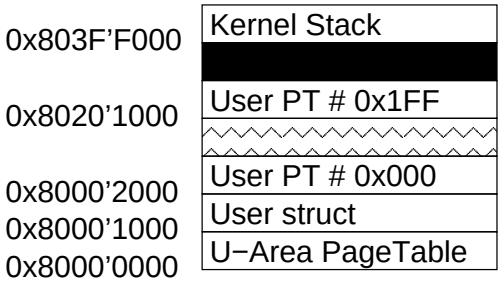
Let us work through an example. The page-size is 4096 Bytes, which results in 12 bits offset. In one frame, 1024 32-bit-words can be placed, so let’s take 10 bits for page table entry. In a 32-Bit address, 10 more bits a left, these are the page directory entry bits. Our 32 bit virtual address has then the form :

The virtual address space starts at 0. Let’s find the physical address for 0x80001’00a. First extracting the page-directory entry by shifting out the 22 least significant bits = 0x200. The 513rd[2] entry in the page-directory brings us to the page-table. Extracting the page-table-entry by removing page-directory entry and shifting out offset, getting entry number 1. In this respect, the second entry in the page-table contains the physical location of the virtual page address 0x8000’1000, in which we want the 10th byte.

[2]First entry is entry number 0.

7.2.2 The specialised concept

The concrete implementation varies according to the "general concept". The first difference is the divided virtual address-space. The lower half is mapped for the user, and the lower upper half is mapped for kernel. Because the user mapping is changed to the current running process, it is inconvient to change the lower half of the page-directory to get the user mapping. Moreover it is a good idea to place the user page-tables in the virtual kernel space. When we continuously arrange all user page tables, there's no need for a user page directory. No memory for unused page-tables is wasted, because it is mapped, so only used page-tables are mapped to a physical frame. More specifically, the user page-tables are located in the U-Area, which is described through the U-Area page-table. Remember that kernel stack and user struct are located in the U-Area. The U-Area has the following composition :



The zig-zag lined region, could be filled with page-tables whenever needed. From bottom to top for data page-tables and reverse for user-stack page-tables. The black hole region is wasted. The count of kernel-stack frames is configurable with the K_STK_SIZE macro. Actually it is set to 4. But one to illustrate should be sufficient.

Owing to he fact, that the whole U-Area is mapped by the U-Area-page-table. Processing switching becomes easier. Only the first entry in the kernel page-directory has to be adjusted to get a new mapping, which is activated by a TLB flush. The U-Area page-tables plays three roles: the first having the prime role as page-table; the second as page-directory for the user page-tables, which is only used indirect through the virtual mapping; the third role that it maps swapped out page-tables, when a process becomes swapped out. We can now, that the U-Area page-table is the new key for a process. According to this, the U-Area page-table is all the time referenced via the P_ADDR field.

Because the kernel page-directory only consists of the pointer to the UArea-page-table, it is only one pointer. If a mapping for the kernel should later be implemented, the pointer has to be replaced with a page-directory.

Pointing out that the page-directories are gone for the usual kernel and user mapping, they still exist in text-segment mapping. In general the text-segments are allocated separate from the process, to share them amongst the processes. When a process is load into memory, the mapping at the beginning is replaced by the mapping of the text segment. This means the whole mapping of the text-segment is copied and the user data-mapping is attached to it.

The whole in four statements :

User page-directory does not exist, because the page-tables are continuous. The page-directory is replaced by the formula :

$$PageTableAddress(n) = n * PageSize + FirstPageTableAddress.$$

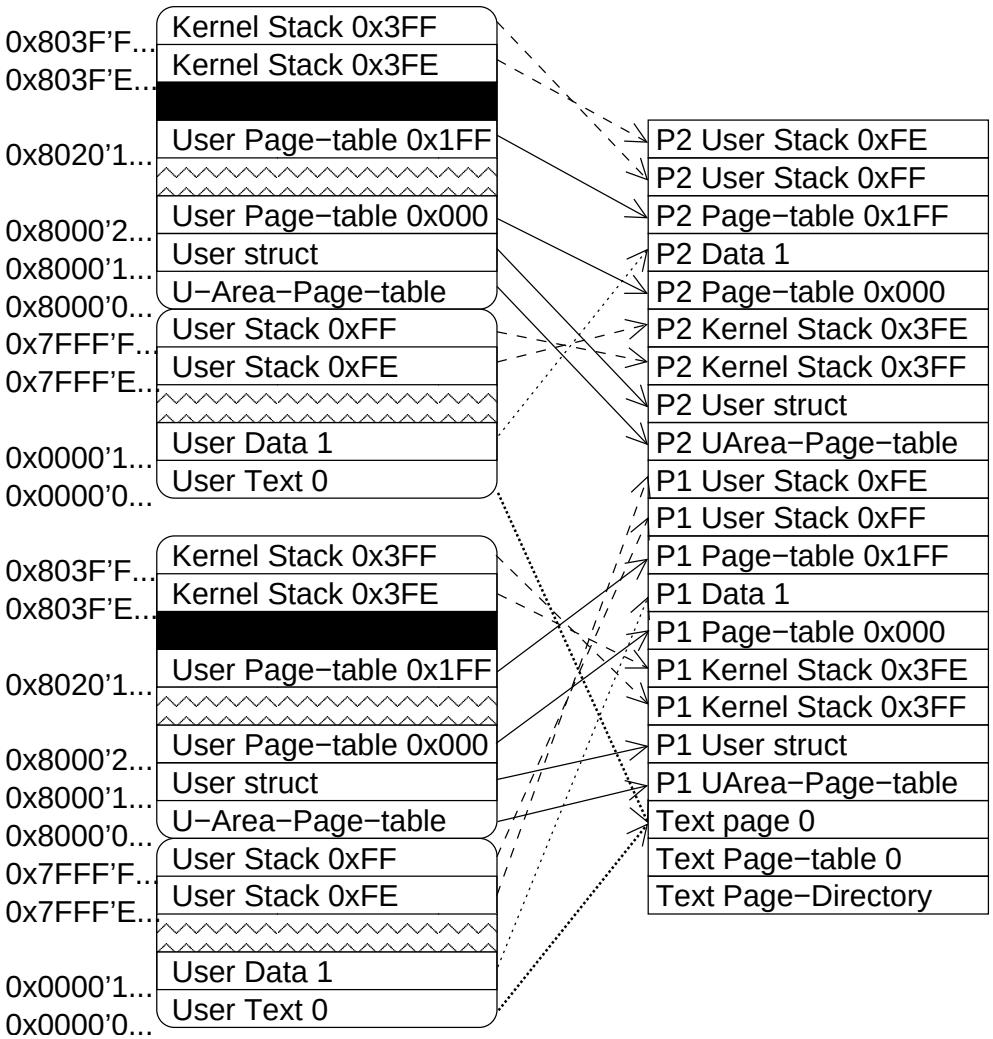
Kernel page-directory is not used, so it is replaced by a pointer to the U-Area-Page-table.

Process switching is primarily changing the U-Area-Page-table-pointer.

Text Page-directory still exists, but never used directly for mapping.

7.2.3 The big Example

First an explanation on the elements which can be seen on the diagram. The two divided boxes on the left represent two virtual address spaces. Every virtual address-space is divided into kernel-space (upper part) and user-space (lower part). The big box on the right reflects a continuous part within the physical core memory. The upper virtual address space, shows the mapping while process 2 (*P2*) is running,. Corresponding to that the lower box shows the virtual address-space while *P1* is running. Every virtual-address-space is labelled with page start addresses. The last free digits ,which contain the offset, are left empty.



First we imagine a running *P1*, *P2* isn't alive yet. *P1* does an EXEC call to a program with shared text. First the operating system, allocates the text-

segment. thus the first frame is filled with the "Text Page Directory" followed by the "Text page-table 0", and finally the text itself. It seems a little bit lavish, to allocate 2 frames, to manage one text frame. On the one hand, in reality, text is most time bigger than one frame, and on the other, the memory-management functions are implemented, in the way, that one frame could be page-directory and -table in one. But don't care about, it's an optimisation. Back to the example, the first three frames of our core are used by the text. Next part is to allocate frames for the UArea, therefore the U-Area-page-table is allocated and filled with the allocated frames for user-struct and kernel-stack. "Page-table 0" is allocated and filled with the entrie's' from the text-page-table. One data frame is allocated and mapped as second page. At this point, *P1* is ready to run with the new text-segment. The first time *P1* tries to push something on the stack, the operating system allocates stack. First allocating the page-table to map the stack, after that one frame, to include the pushed element, is allocated and one additional. Now everything is allocated and mapped for *P1*.

It could be interesting to see how the stack-page pointer (dashed arrows) are crossing, this results from the fact, that the stack is growing backwards, therefor the top-page is allocated first. Important is, that the U-Area page-table has to be allocated first, to map the user page-tables. After allocating user page-tables, user frames can be allocated. Therefore, when a process starts, many nearly empty tables are allocated, which are filled while the process is running. Another important issue is, that the Text-page page-directory and -table only used to copy the right values into the first user page-table*s*.

It's time for *P2*, *P1* does a FORK call, and everything, except the text-segment, is copied from *P1* to the new process *P2*. To share the text-segment, the text-mapping is copied again.

7.3 New Memory-management interface

The memory management is implemented in the file SYS/MEMORYMANAGMENT.C. Most of the functions work on a passed page-directory, in which they allocate or free frames. The conceptual swapping is the task of the memory-management, too. The physical swapping is done by SWAP implemented in DEV/BIO.C.

A physical address has the type PHYS_T. This is simply an INT, variables of this type are mostly written into the TLB. Therefore a page-table is an array of PHYS_T, and a page-directory is an array of pointers to PHYS_T (the strict double pointering is not used !). There are two functions to convert parts of the UArea-Page-table, into a page-directory and back. SEG_T is used whenever a frame number, or a count of frame is needed. This is actually an INT, too.

This chapter first describes the functions to get an overview. After that they are explained in more detail, with some important side-effects. Finally the swapping process is explained in further detail.

7.3.1 All functions in short

MALLOC allocates one frame from the specified map, which could be either COREMAP or SWAPMAP.

MFREE release one specified frame from the passed map, which could be either COREMAP or SWAPMAP, too.

MAVAIL returns the count of free frames on the specified map.

FREETABLES frees a specific count of page tables attached to the passed page-directory.

FREEFRAMES frees a specific count of frames mapped through the passed page-table.

COPYMAPPING copies the mapping from a source page-directory to a destination-directory. This is used to copy the text-segment mapping or when forking a process.

SWAPTABLES swaps a specific count of tables to or from disk. This function is called after swapping out the frames, or before swapping in frames. The entries in the passed page-directory are replaced by the the new frame on disk or in core memory.

SWAPFRAMES swaps a specific count of frames in or out. The entries in the page-tables are replaced by the new frame-number on swap or in core.

LOADPROC loads a process from swap, using the functions above.

STOREPROC stores a process on swap using the functions above.

ESTABUR modifies the segment size of the current running process. It is the **only** function which allocates memory to a process, only estabur has the control over the segment size variables.

7.3.2 MALLOC

INT MALLOC(STRUCT MAP *MP);

MALLOC is called either with COREMAP or SWAPMAP as argument. Both are global pointers, pointing to a data-structure which is used to manage the available memory. The primary assumption is, that the frame 0 could never be allocated. Returning 0 should indicate that no more memory is available. It is not possible to allocate more than one frame at a time.

7.3.3 MFREE

VOID MFREE(STRUCT MAP *MP,INT A);

MFREE is used to release an allocated frame. On startup it is used to free all core- and swap-memory. The frame 0 must not be freed. This would cause malloc to return 0 on the next call, and no more memory allocated.

7.3.4 MAVAIL

int MAVAIL(struct map *mp);

MAVAIL is used to determine early if enough memory is available on the passed map.

7.3.5 FREETABLES

VOID FREETABLES(PHYS_T * PAGEDIR[], INT FIRSTTABLE, INT COUNT, BOOL UPWARDS);

FREETABLES is called to release the page-tables attached to a page-directory. The COUNT parameter specifies how many tables should be freed. E.g. if text- and data-segment are using 1028 memory-frames, 2 tables have to freed starting at zero. 2 pages results from the fact, that one page-table contains 1024 entries. With help of the UPWARDS flag, freeing in both directions is possible. Upwards to free text- and data-page-tables, downwards to free stack-page-tables.

A very important fact is, that frames are only freed by a call to MFREE, all data within the page-tables and the directory are still valid.

7.3.6 FREEFRAMES

VOID FREEFRAMES(PHYS_T * PAGEDIR[], INT FIRSTFRAME, INT COUNT, BOOL UPWARDS, BOOL COREMAP);

FREEFRAMES is very similar to FREETABLES, and is called before freeing the page-tables, e.g. when a process is exiting. This function has one additional argument to specify the map, where the frames should be freed from. This parameter is missing in the FREETABLES function, because a page-table has to be in core to free the frames mapped through it.

7.3.7 COPYMAPPING

VOID COPYMAPPING(PHYS_T * SPAGEDIR[], PHYS_T * DPAGEDIR[], INT FIRSTFRAME, INT COUNT, BOOL UPWARDS, BOOL ALLOCTABLES);

COPYMAPPING has quite a lot of arguments, which has to be set carefully ! This function is used to copy an existing mapping to an new already allocated page-directory. According to the other functions the start and count of frames, and the direction could be specified. ALLOCTABLES is used to specify, if the page-tables were allocated previously. E.g. is this the case when swapping in a process and ESTABUR already has setup the mapping skeleton.

7.3.8 SWAPTABLES

VOID SWAPTABLES(PHYS_T * PAGEDIR[], INT FIRSTTABLE, INT COUNT, BOOL UPWARDS, BOOL SWAPIN, BOOL FREE);

SWAPTABLE swaps the specified count of page-tables in or out. The SWAPIN flag determines the swap direction. Swapping in means to load frames from disk into core. If the FREE argument is TRUE, the source of the frames are freed. This is not wanted when coping a process over the swap. When swapping in, the entries in the page-directory contain swap-page numbers. According to these numbers, the page-tables are load from the swap-pages into a new core frames. The entries in the page-directory are replaced by the new address. The same just reverted is done when swapping out.

7.3.9 SWAPFRAMES

VOID SWAPFRAMES(PHYS_T * PAGEDIR[], INT FIRSTFRAME, INT COUNT, BOOL UPWARDS, BOOL SWAPIN, BOOL FREE);

SWAPFRAMES is very similar to SWAPTABLES. The main difference is the swapping of frames instead of page-tables. This makes a big difference how the new addresses are stored, because the permission flags shouldn't get lost while swapping. Therefore the following macros are used to handle core-frame- or swap-disc-frame-numbers within page-table-entries :

DBLK_TO_PTE(D,TE) puts the specified swap-disk-frame-number D into the page-table-entry TE,

FRAM_TO_PTE(F,TE) puts the specified core-frame-number F into the page-table-entry TE,

DBLK_FR_PTE(TE) extracts swap-disk-frame-number from the page-table-entry TE,

FRAM_FR_PTE(TE) extracts core-frame-number from the page-table-entry TE.

7.3.10 LOADPROC

VOID LOADPROC(STRUCT PROC * P);

LOADPROC swaps in the specified process, expect the text-segment. If the shared text pointer P->P_TEXTP is valid, the text-mapping is copied. At the time of the call, P->P_ADDR points to swap-page which stores the U-Area-Page-table. The U-Area-Page-table is swapped into a new core-frame and the frame-number is stored in P_ADDR. Now the rest is swapped in using the SWAPTABLES and SWAPFRAMES functions.

7.3.11 STOREPROC

VOID STOREPROC(STRUCT PROC * P, BOOL FREE);

STOREPROC swaps out the specified process in the inverse way of LOADPROC. After swapping out, P->P_ADDR is replaced with the swap-disk-frame-number of the stored U-Area-Page-table.

7.3.12 ESTABUR

INT ESTABUR(SEG_T NT,SEG_T ND,SEG_T NS,INT SEP,INT XRW);

ESTABUR manages the process memory allocating and releasing. To end its works it flushes the TLB. The last argument determines whether the text-segment is read-write or read-only. The arguments NT, ND and NS specify the size in core frames of the text-, data- and text-segment. After allocating or freeing core-frames, the new sizes are stored in U_TSIZE, U_DSIZE, U_TSIZE, which are located in the user-struct. Finally the process size in P_SIZE is newly calculated. Whenever a page-table is allocated or freed, U_USIZE is adjusted. U_USIZE reflects the size of the U-Area in clicks. The flag called SEP, is always cleared in our environment, because we have no separate instruction and data address space.

The beautiful name ESTABUR has the origin "ESTABLISH USER REGISTERS".

7.4 Swapping

This section describes the process in detail. First the swap in process done by LOADPROC is described. In general the swap out process acts in the same way, only reversing. When swapping out, particular care should be taken, therefore the specialities of swapping out are described in the STOREPROC subsection.

7.4.1 Swapping in with LOADPROC

This subsection describes the action of LOADPROC in further detail. To keep it simple, it is assumed, that the core is not fragmented, and every new allocated frame is located on top of the previous allocated. In reality many frames could appear between two allocated frames. First an explanation of the symbols, which are used in the diagrams :

Boxes, every box reflects one physical core-frame. There are two different kinds. The small rectangle-boxes are used, when maximally the name of the frame is from interest. In most rectangle boxes the name does not appear and should be clear through the referencing arrow. The bigger boxes with round edges are used when many entries within this one frame are of interest.

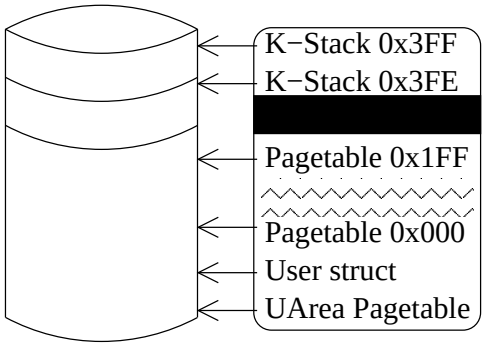
Disk, symbolises the swap disk.

arrows, the different kinds of arrows are only taken to keep the overview. Arrows point from a page-table-entry to disk or to an allocated core-frame.

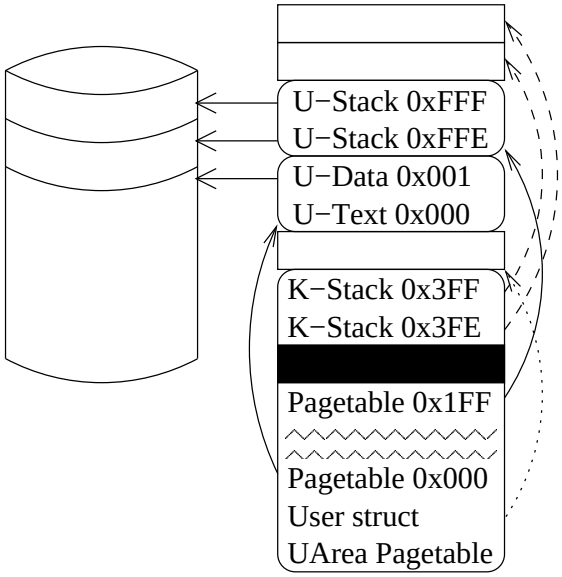
Black-Whole shows wasted space, it does NOT indicate one physical frame !

Zig-Zag-Lines indicates space where entries or frames could appear.

While a process is swapped out, P_ADDR contains the swap-page-number of the stored U-Area-Page-table, which is swapped in first of all. All entries in the swapped in U-Area-Page-table reference swap-page-number :



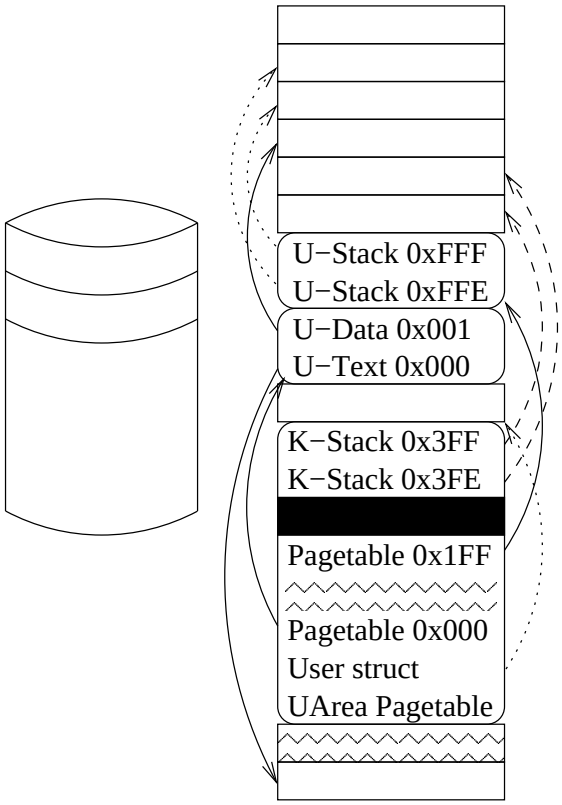
First the user struct is swapped in separately. Next step is to allocate and swap in frame wise using SWAPFRAMES. Because SWAPFRAMES expects a page-directory as argument, the first entry in the U-Area-Page-table[3], is converted into a physical pointer. Therefore the U-Area-Page-table is now a page-directory, which first, and only, entry references itself in function of a page-table. Now all referenced entries got swapped in :



At this point, the swapped in page-tables-entries point to the swap. Only the text entries are invalid and are replaced later by the text-mapping. To swap in the referenced pages, the pointer within the U-Area-Page-table, which reference page-tables, has to be converted into absolute-pointers. The middle of the U-Area-Page-table now looks like a page-directory for the user-mapping.

[3]it references always itself, because it is the page-table of beginning of the virtual-kernel-address-space, where it is located.

Therefore all user frames could be swapped in calling SWAPFRAMES two times[4] :



To finish the work, the converted entries within the U-Area-Page-table are converted back and the mapping of the text-segment is copied. Facing this, the text-entry points to core-memory, which was allocated before this process was swapped in.

[4]Two times, because it is only swapped into one direction at one call.

7.4.2 Swapping out with STOREPROC

We will now look at the specialities of swapping out. In general swapping out is the same like swapping in inverted. First of all the referenced core-frames are swapped to disk, and the new location is stored in the page-tables. After that the page-tables are swapped out and referenced by the U-Area-Page-table. Last but not least, the U-Area-Page-table is swapped out.

Swapping out is not only done to get a process out of core. Imagine we have one existing process, which allocated more than the half of the available core-memory. This fat process now FORKs. It is impossible do FORK within the core. Therefore the swapping out is simulated, by swapping out and copying the mapping beforehand. This results in two identically images, one time in core and one time on swap. The image on swap is referenced by the new process and the old image in core is still referenced by the origin process.

While swapping out, the entries within the page-table and -directory are modified. Therefor the core-mapping becomes invalid. With the result that, the mapping has to be copied, before the swapping out process can begin. Because the mapping has to be copied, temporary memory has to be allocated. In case of less memory, this could cause a dead-lock, because the kernel wants to free memory by swapping out, but can not swap out, because no frames are left. To solve this, one frame always has to exist, to copy the U-Area-Page-table. While swapping out a process, it is not possible to run in user mode for this process. Therefore the page-tables mapping virtual user space, are not used and can be swapped easily. The frames still used are the original U-Area-Page-table and kernel-stack. While swapping out the frames, they are only released and can still be used. In short :

U-Area-Page-table is copied, the original references possible released pages while swapping out.

Swapped-out pages are referenced through their corresponding page-table, which is not used through user.

Copied-UArea-Page-table is swapped out last, with swap-references to user-page-tables and kernel-stack.

Because only one static frame is available for copying the mapping, a fork could only be done if at less enough core for a complete second U-Area is available.

Chapter 8

The running system

This chapter explains some code details and how to let the kernel run on the Eco32 simulator. The sample init-process is explained in detail, so the dynamic of the system can be seen when allocating, forking, swapping, killing and dying is done. The sample does not include user-access to the disk, which could be enabled within the sample program. A short section, about how to write an own init-processes closes my part of porting. The very lasts but not least section of this document describes how to setup a PDP-11 simulator, with putting the original binaries on it's disk and let the Unix v7 run. When the simulator runs on a Pentium around 100 MHz the real speed is simulated. The installation of Unix on the simulator is done using the original installation instructions !

8.1 Getting, extracting and compiling the sources

The ported Unix v7 sources are found under the simulator sources. Therefore, only the simulator sources have to be downloaded from [Geisse]. The sources have to be extracted with "TAR XVZF ECO32-*x.y*.TAR.GZ", *x.y* reflects the actual release. After extracting and changing into the new directory "ECO32-X.Y", the simulator has to be compiled by a "MAKE". The compiler is compiled using "MAKE COMPILER", and finally the operating system with "MAKE UNIX". The following directories and files are most relevant :

BUILD/BIN contains LCC, ASLD, various disk- and FS-tools. More important the Eco32-Simulator itself, called SIM.

BUILD/RUN contains all additional stuff needed to run the simulator, these are the disk and the operating system.

DOC/ holds some documents to the simulator.

V7/DOCUMENTS the original v7 postscript documentations V7VOL1.PS, V7VOL2A.PS and V7VOL2B.PS. The volume 2b includes the original man-pages and the introductions to setup up Unix v7 on a PDP-11.

V7/EXTRACTED contains the extracted original sources. These are the kernel source and the standard user programs including the libraries.

V7/ANSIFICED holds the ansificed and linkable sources. No more porting is done with these files.

V7/PORTED source tree of ported pieces,

V7/PORTED/USR/SYS the ported kernel-sources, also referred as *kernel-dir.*

Depending on the distribution, the directories V7/EXTRACTED and V7/ANSIFICED, like the postscript are packed into TAR.BZ2 files.

8.1.1 The kernel directory

The original kernel contains of the following three directories[1]:

DEV/ containing many device-drivers,

H/ system-wide important include files. The file GLOBALS.C now locates all global used variables.

CONF/ machine depend implementations, primary low level copy and system entrance.

SYS/ the kernel itself.

Additionally the MAKEFILE came up with the TOOLS directory. The TOOLS directory contains the "binary to include file"-converter CONV which converts executables into an includable format. This is used to replace the hexadecimal init code. To work with Unix map the tool MKMAP exists. It sorts the symbols and adds the segment-address to the symbols offset.

[1] all relative to the *kernel-dir*

8.1.2 Some porting code guidelines

Because on of the primary target is to get an portable operating system, I added " _ECO32" to every filename which is machine dependent. These are the following files :

DEV/BIO_ECO32.C only the function PHYSIO is machine dependent, and is left empty. It is used for low level device test.

DEV/MEM_ECO32.C is the device driver to access the physical memory,

H/MCH_ECO32.H contains machine depended function definitions, which only appear on the Eco32 platform,

H/REG_ECO32.H contains the location of the stored register on stack while exception,

H/SEG_ECO32.H the home of all converting memory management definition and converting macros,

SYS/MAIN_ECO32.C little machine depend startup code, primary the initialisation of process 0,

SYS/MACHDEP_ECO32.C machine depended startup and memory allocation. It is the home of the signal handler caller too.

SYS/UREG_ECO32.C just contains the main function ESTABUR to modify the process segment sizes.

8.1.2.1 Debug output

For every code-file with debug-output, three debug-level are supported, which are set in the CDEFS in the MAKEFILE :

0 quiet,

1 some important functions print out there arguments when called,

2 verbose debugging is enabled.

Not really a tidbit, but exists. For some code files similar definitions appear on the beginning :

```
VOID PRINTNULLTRAP( CHAR * HANS, ... ) {;}
#IFDEF DEBUG_TRAP
 #DEFINE DBG_PRINT1 PRINTF
 #IF DEBUG_TRAP > 1
  #DEFINE DBG_PRINT2 PRINTF
 #ELSE
  #DEFINE DBG_PRINT2 PRINTNULLTRAP
 #ENDIF
```

```
#ELSE
 #DEFINE DBG_PRINT1 PRINTNULLTRAP
 #DEFINE DBG_PRINT2 PRINTNULLTRAP
#ENDIF
```

This block cause DBG_PRINT1 and DBG_PRINT2 to be defined to PRINTF or the empty function. The definition depends on the definition of DEBUG_TRAP which is set within MAKEFILE. DBG_PRINT[12] are used within the code-files to produce debug-output. The following definitions are evaluated for the corresponding file :

DEBUG_BIO dev/bio_eco32.c, the low-level SWAP-function produces some debug-output.

DEBUG_CLOCK sys/clock.c, level 1 gives out the calculated CPU-usage of every process. Level 2 prints out a lot.

DEBUG_ET dev/et.c, gives out nearly nothing.

DEBUG_IDLE defined to 1 every 64th idle is given out. When defined to 2, every idle call is given out.

DEBUG_MEMMAN sys/MemoryManagment.c, level 1 gives out the important calls with arguments. Verbose debugging is possible with level 2.

DEBUG_NAMEI sys/namei.c, level 1 only gives out only one error-message. While level 2 debugs the whole process of finding an inode to a corresponding filename.

DEBUG_SIG sys/sig.c,the important system-call-handler grow produces 2 levels of debug output.

DEBUG_SLP sys/slp.c SWAPIN, SWTCH and NEWPROC produces two levels of debugging output.

DEBUG_SYS1 sys/sys1.c, EXECE, GETXFILE, WAIT and FORK gives out debugging information in two levels.

DEBUG_SYS2 sys/sys2.cOPEN produces some debugging output.

DEBUG_TEXT sys/text.cXSWAP, XFREE, XALLOC and XEXPAND support debugging in two levels.

DEBUG_TRAP sys/trap.c the TRAP-handler gives out only trap-number, PC at trap, and syscall-function-name on debug level 1. Verbose output is created on debug-level 2.

DEBUG_TTY dev/tty.c

8.2 Tidbits

In such a delicious operating system, some new tidbits have to be done. The first is to get rid of permanent inconsistency, when redefining constants and macros in the C or assembler part, by letting the C preprocessor, process the assembler file. The second tidbit includes the linker generated map-file into the file SYS/TRAP.C. Therefore the trap handler is capable of getting the kernel-function name on traps and system calls.

8.2.1 MCH_ECO32.PRES

The machine depending assembler implementation varies a bit from the common way, because it includes C-header-files. To translate the file MCH_ECO32.PRES into the common file MCH_ECO32.S, two steps are done by the MAKEFILE. In common the C preprocessor is used to resolve any macro definitions. The first step is to find out all macros included by MCH_ECO.PRES. This step is needed, because all the declarations in the header files are worthless for the assembler. The macros are extracted by :

CAT CONF/MCH_ECO32.PRES | CPP -I CONF/ -DM -P >CONF/MCH_ECO32.DEFS

Now the extracted macros are used on the content of MCH_ECO32.PRES without the include-directives. The resulting output is stored in the assembler file MCH_ECO32.S :

CAT CONF/MCH_ECO32.PRES | GREP -V "#INCLUDE" | CPP -P -IMACROS CONF/MCH_ECO32.DEFS >CONF/MCH_ECO32.S

With this little workaround in the MAKEFILE it is possible, to use any constants and macros which are defined in any included header file. The advantage is the single point of definitions.

8.2.2 The included Unix-map

To become debugging a little more comfortable, the file UNIX.MAP is very important. It is created by the linker and contains a relation between all exported symbols and their offset within a segment. The segment could be either BSS, CODE or DATA. The segment-offset is listed at the end of the map.

To convert the UNIX.MAP into an includable file, I've written the tool MKUNIXMAPH. The prime task of MKUNIXMAPH is to sort the symbols by the offset. Finally the segment-offset is added to every symbol offset.

Because linking is done after compiling, the MAKEFILE checks if the new map-file differs from the previous map file and recompiles eventually.

Whenever a critical trap occurs, or the debug-level is high enough, the name of the causing kernel function is written. When debugging system-calls, the name of the called system-call-handler is printed out by TRAP.

8.2.3 Mosix featuring

The MAKEFILE is designed to use a Mosix-cluster when running the simulator. Every mosix system contains the program /BIN/CPUJOB, which executes the passed program with all passed parameters. Therefore if this program doesn't exist, a mini shell-script is generated which calls the first argument with all other passed arguments. The use of CPUJOB causes Mosix to start the passed program on the node with the most powerful CPU.

8.3 Running the example

At this point, the simulator, compiler and operating should be compiled. Two steps have to be done before running the system. First the INIT program has to be copied to the hard-disk together with a valid file-system. Next step is to copy the unix-binary, which acts the function of the ROM-program. Therefore type the following two commands :

MAKE UNIX-DISK

MAKE UNIX-RUN

The UNIX-DISK target creates the file-system containing the init-process. The UNIX-RUN target copies the binary and starts the simulator. Within the simulator, the command "C" has to be given, to continue the execution.

The next subsection explains the code of the running init-process, followed by a subsection explaining the resulting output.

8.3.1 The sample overview

Within the sample program some tests are implemented. These are tests for handling files and some different exec calls. Within this example the tests SWAPTEST and MEMTEST are used.

It is forked three times within this sample :

1. At the beginning of the SWAPTEST, the forked child gets killed, and
2. a new child is forked and both leave swapTest,
3. after the child terminated its parent,

the new parent causes a segmentation fault, while the child exits with exit-code 0XDEAD0000.

```
#INCLUDE "../H/SYSCALL.H"        PID is 0 for children and..
INT PID = 0;                     .."PID of child" for parents.
INT MAIN( INT ARGC,
  CHAR *ARGV[],
  CHAR * ENVP[])
{                                Set the signal handler SIGTERM..
 SIGNAL( SIGTRM, SIGTERM );      ..for signal SIGTRM.
 SWAPTEST();                     forks via swap.
 IF( PID )                       Parent has to loop until child..
  WHILE( 1 );                    ..sends SIGTRM.
 ELSE {
  KILL( 1, SIGTRM );             Child kills parent with SIGTRM
  PID = FORK();                  Lonely child forks
  IF( PID )                      The new parent allocates and
   MEMTEST();                    touches memory until segm.-fault
 }
 _EXIT( 0XDEAD0000 | PID );      The new child exits 0xDead0000
 WHILE( 2 );                     This code is never reached !
 RETURN 0;
}
```

The SWAPTEST makes a fork via swap, because the forking process is larger than the half core.

```
VOID SWAPTEST(VOID) {              Maximal core is something
 INT MAXMEM = 0X70000;             below 0x90 core-frames.
 INT I, * PI;
 BRK( 0X70000 );                   Allocate 0x70 core-frames
 PID = FORK();                     fork via swap
 FOR( PI = 0;                      This loop reads one word
  PI < (INT *)MAXMEM;              from all allocated frames
  PI = (INT *)((INT)PI + 0X1000))
 {
  I = *PI;
 }                                 The parent kills the child with
 IF( PID ) {                       SIGKIL which is uncatchable.
  KILL( PID, SIGKIL );             Wait until child is dead,
  WAIT(0);                         and get a new child
  PID = FORK();
 } ELSE                            Here the first child is catched,
  WHILE( 1 );                      to get killed.
}
```

The memTest allocates memory and touches the last CHAR in the pre-last frame in an infinite loop. The loop is interrupted when the fetching causes a segmentation fault.

```
VOID MEMTEST(VOID) {               Start with nearly the..
 INT MAXMEM = 0X70000;             ..whole core-memory
 CHAR C, * PCHAR;
 WHILE( 1 ) {
  BRK( MAXMEM );                   Allocate memory,
  PCHAR = (CHAR *)MAXMEM - 1;set pointer to last character..
  C = *PCHAR;                      ..within page, and fetch it
  MAXMEM += 0X8000;                Increment by 8 core-frames.
 }
}
void sigTerm() {                   Signal-catcher for SIGTERM,
 _exit( 0xDeadBeef );              exit with exit-code = 0xDeadBeef
}
```

8.3.2 The sample output

All processes are called by their aliases like *init* instead of process 1, or *scheduler* instead of process 0 and *P2* as alias for process 2.

The first three lines[2] contain information about the available memory. Therefore nearly the half of the 1 MB main memory is used by the operating system. The swap-partition is completely unused.

ECO32 > C
MEMFRAMES = 138 (0X8A)
MEM = 552 KB
SWAP = 2000 KB (0XFA0 DBLCKS, 0X1F4 SWAPFRAMES)

Now only debug output follows. In this sample, debug-level 1 is enabled for SYS/TRAP.C, SYS/MEMORYMANAGMENT.C and SYS/CLOCK.C.

8.3.2.1 Startup-init-process

The first ESTABUR call, establishes the data-segment for the startup-init-process. This process is a smart hexadecimal-code assembler-program, which causes an EXEC("/ETC/INIT")-*syscall*. The *syscall* is initiated through the trap-instruction, which causes a trap 0x14. The arguments to the exec-call, are the pointer the file-name and the pointer to the argument-vector. The equal sign reflects that the return value will come. It is possible that any other output will come before the return value !

-> ESTABUR(NT = 0(0), ND = 1(0), NS = 0(0), SEP = 0(0), XRW = 4)
|-> USER-TRAP=0X14 PID=0X1 PS=0X274FFFF @=0X0
SYSCALL @0XC0008508 EXEC(0X18, 0X10,) =

The next three estabur-calls are caused through the exec-call. The first estabur call, checks if it is possible to create the process. Now only the data-segment is allocated, to put the code into it. After that the stack is allocated. This shows the simplest variant of the binaries, no text-sharing is possible. The last lines are the return-arguments of the *syscall*. Within the EXEC-*syscall* it is only important that fail is zero. If the *syscall* would fail, the startup-init-process would run into an infinite loop. Elsewhere, the control is at *init*.

-> ESTABUR(NT = 0(0), ND = 2(1), NS = 2(0), SEP = 0(0), XRW = 4)
-> ESTABUR(NT = 0(0), ND = 2(2), NS = 0(2), SEP = 0(0), XRW = 4)
-> ESTABUR(NT = 0(0), ND = 2(2), NS = 2(0), SEP = 0(0), XRW = 4)
RET0= 0X0, RET1= 0XC0100000, FAIL= 0

[2] behind the continue command

8.3.2.2 Init becomes alive

First the the user-program sets the signal-handler for signal 0xF[3]. The passed address 0x1058 points into the "syscall-library". The library catches all registered signals, stores the environment, calls the user signal handler, restores the environment. And finally causes an exception-return to the location passed by the operating system.

The second syscall allocates 0x70 frames for the users data-segment and returns successful.

```
|-> USER-TRAP=0X14 PID=0X1 PS=0X374FFFF @=0XFFC
SYSCALL @0XC000B6B8 SSIG( 0XF, 0X1058, ) =
RET0= 0X0, RET1= 0XC0100000, FAIL= 0
0 cpu= C -> 9 | 1 cpu= 9D -> 7D |                    ignore this line !
|-> User-Trap=0xC PID=0x1 PS=0x36DF.. @=0x1004      ignore this line !
|-> USER-TRAP=0X14 PID=0X1 PS=0X374FFFF @=0XA54
SYSCALL @0XC0009C7C SBREAK( 0X70000, ) =
-> ESTABUR( NT = 0(0), ND = 70(2), NS = 2(2), SEP = 0(0), XRW = 4)
RET0= 0X0, RET1= 0XC0100000, FAIL= 0
```

8.3.2.3 The *scheduler* and cpu-usage

The following lines shows the frequently cpu-usage-calculation which is caused through HZ[4] clock interrupts. First recognise, that the calculation output comes before the trap, because the clock-interrupt-handler does the calculation and then causes a "Giveup CPU-Trap[5]". The first line shows that the *scheduler* cpu-usage is set from 0x9 down-to 0x7, *init*s cpu-usage is decreased from 0x9C to 0x7C.

The cpu-usage of the running process is increased every clock-interrupt. The decreasing is done by multiplying the factor $SCHMAQ = 8/10$. The magic scheduling factor causes the cpu-usage of not running processes to drop down, while the usage of the running process still increments (or only decrements very less.). The cpu-usage of the *scheduler* is initial at 9 and drops to 7 till next scheduling. In this time the initial cpu-usage of *init* was at 0x9C and moved to 0x9A. Now it seems as if they both decrement by 2, but the 2 decrements have nearly no influence at a usage of 0x9A (154 decimal). When looking on the first *ignored-code-line* above, we see that the *scheduler* started at 0xC which equals 12, and the 7 below is nearly the half.

```
0 CPU= 9 -> 7 | 1 CPU= 9C -> 7C |
|-> USER-TRAP=0XC PID=0X1 PS=0X36DFFFF @=0XA5C
0 CPU= 7 -> 5 | 1 CPU= 9A -> 7B |
|-> USER-TRAP=0XC PID=0X1 PS=0X36DFFFF @=0XA5C
```

[3] SIGTRM = SIGnal-TeRMinate, causes a process to finish and maybe tidy up before.
[4] HZ is set to 60, which causes every second a new cpu-usage calculation.
[5] is used to schedule, 0xC is the trap number.

8.3.2.4 *swapTest*

Init forks the first time within the SWAPTEST. The *syscall* has to STOREPROC the new process without freeing the core-frames. To fork on less core, the P_ADDR is copied from the parent and then the new process is swapped out.

The next two *syscalls* are the KILL and the WAIT until the child is dead. The wait syscall causes *init* to become swapped out and *P2* swapped in.

|-> USER-TRAP=0X14 PID=0X1 PS=0X374FFFF @=0X758
SYSCALL @0XC0009968 FORK( ) =
STOREPROC 0X2(P=0XC006C6C8, FREE=0)
RET0= 0X2, RET1= 0XC0100000, FAIL= 0
0 CPU= FF -> CC | 1 CPU= AD -> 8A | 2 CPU= 53 -> 42 |
|-> USER-TRAP=0XC PID=0X1 PS=0X36DFFFF @=0X328
|-> USER-TRAP=0X14 PID=0X1 PS=0X374FFFF @=0X1274
SYSCALL @0XC000B780 KILL(0X2, 0X9,) =
RET0= 0X2, RET1= 0XC0100000, FAIL= 0
0 CPU= CD -> A4 | 1 CPU= B2 -> 8E | 2 CPU= 42 -> 34 |
|-> USER-TRAP=0XC PID=0X1 PS=0X36DFFFF @=0X127C
|-> USER-TRAP=0X14 PID=0X1 PS=0X374FFFF @=0X83C
SYSCALL @0XC0009750 WAIT( ) =
STOREPROC 0X1(P=0XC006C6A0, FREE=1)
LOADPROC 0X2 FROM 0X7A -> 0X76, 1
RET0= 0X1, RET1= 0XC0100000, FAIL= 0

8.3.2.5 looping P2 dies

all allocated frames are freed. The first two call shows the use of the U-Area-Page-table as page-directory starting at the second word. Within the last three calls, the UArea-Page-table first entry references itself as page-table.

FREEFRAMES(PD= 0X80..8, 1.FRAME= 0X0, N= 0X70, UP= 1, COREMAP
FREEFRAMES(PD= 0X80..8, 1.FRAME= 0X7FFFF, N= 0X2, UP= 0, COREMAP
FREEFRAMES(PD= 0X80..0, 1.FRAME= 0X3FF, N= 0X4, UP= 0, COREMAP
FREEFRAMES(PD= 0X80..0, 1.FRAME= 0X1, N= 0X2, UP= 1, COREMAP
FREEFRAMES(PD= 0X80..0, 1.FRAME= 0X201, N= 0X1, UP= 0, COREMAP

8.3.2.6 Init swaps in and forks again

LOADPROC 0X1 FROM 0XF4 -> 0X76, 1
RET0= 0X2, RET1= 0X9, FAIL= 0
0 CPU= FF -> CC | 1 CPU= B7 -> 92 |
|-> USER-TRAP=0XC PID=0X1 PS=0X36DFFFF @=0X844
|-> USER-TRAP=0X14 PID=0X1 PS=0X374FFFF @=0X758
SYSCALL @0XC0009968 FORK( ) = STOREPROC 0X3(P=0XC006C6C8, FREE=0)
RET0= 0X3, RET1= 0X9, FAIL= 0

8.3.2.7 Scheduling to P3

Inits cpu-usage increases from 0xC0 up to 0xC2, while *P3s* cpu-usage is falling from 0x50 to 0x33. Every 0x10 cpu-usage, the priority of a process is increased. Therefore the priority of *P3* fall by 2. *Init* becomes swapped out and *P3* swapped in, because *P3* was out for three HZ.

0 CPU= FF -> CC | 1 CPU= C0 -> 99 | 3 CPU= 50 -> 40 |
|-> USER-TRAP=0XC PID=0X1 PS=0X36DFFFF @=0X150
0 CPU= CD -> A4 | 1 CPU= C1 -> 9A | 3 CPU= 40 -> 33 |
|-> USER-TRAP=0XC PID=0X1 PS=0X36DFFFF @=0X150
0 CPU= A5 -> 84 | 1 CPU= C2 -> 9B | 3 CPU= 33 -> 28 |
|-> USER-TRAP=0XC PID=0X1 PS=0X36DFFFF @=0X150
STOREPROC 0X1(P=0XC006C6A0, FREE=1)
LOADPROC 0X3 FROM 0X7A -> 0X76, 1
RET0= 0X1, RET1= 0X9, FAIL= 0

8.3.2.8 *P3* kills *init* with SIGTRM

and forks a new child *P4.*

|-> USER-TRAP=0X14 PID=0X3 PS=0X374FFFF @=0X1274
SYSCALL @0XC000B780 KILL(0X1, 0XF,) =
RET0= 0X0, RET1= 0X9, FAIL= 0
0 CPU= FF -> CC | 1 CPU= BA -> 94 | 3 CPU= 48 -> 39 |
|-> USER-TRAP=0XC PID=0X3 PS=0X36DFFFF @=0X127C
|-> USER-TRAP=0X14 PID=0X3 PS=0X374FFFF @=0X758
SYSCALL @0XC0009968 FORK( ) =STOREPROC 0X4(P=0XC006C6F0, FREE=0
)
RET0= 0X4, RET1= 0X9, FAIL= 0

8.3.2.9 *P3* starts memory allocation.

and becomes swapped out. The last break-*syscall* allocates the last available core, the fetching of char succeeds, because only one frame could not be allocated.

```
|-> USER-TRAP=0X14 PID=0X3 PS=0X374FFFF @=0XA54
SYSCALL @0XC0009C7C SBREAK( 0X70000, ) =
-> ESTABUR( NT = 0(0), ND = 70(70), NS = 2(2), SEP = 0(0), XRW = 4)
RET0= 0X4, RET1= 0X9, FAIL= 0
0 CPU= FF -> CC | 1 CPU= 94 -> 76 | 3 CPU= 71 -> 5A | 4 CPU= 54 -> 43 |
|-> USER-TRAP=0XC PID=0X3 PS=0X36DFFFF @=0XA5C
|-> USER-TRAP=0X14 PID=0X3 PS=0X374FFFF @=0XA54
SYSCALL @0XC0009C7C SBREAK( 0X78000, ) =
-> ESTABUR( NT = 0(0), ND = 78(70), NS = 2(2), SEP = 0(0), XRW = 4)
<- ESTABUR E_NO_MEM !!
RET0= 0XC, RET1= 0X9, FAIL= 57344
0 CPU= CD -> A4 | 1 CPU= 76 -> 5E | 3 CPU= 8B -> 6F | 4 CPU= 43 -> 35
| |-> USER-TRAP=0XC PID=0X3 PS=0X36DFFFF @=0XA5C
STOREPROC 0X3( P=0XC006C6C8, FREE=1 )
```

8.3.2.10 *Init* dies at SIGTRM

it receives the signal while looping in main. It is put into the signal-handler where it exits with 0XDEADBeef.

```
LOADPROC 0X1 FROM 0XF4 -> 0X76, 1
0 CPU= FF -> CC | 1 CPU= 67 -> 52 | 3 CPU= 8E -> 71 | 4 CPU= 35 -> 2A |
|-> USER-TRAP=0XC PID=0X1 PS=0X36DFFFF @=0X150
|-> USER-TRAP=0X14 PID=0X1 PS=0X374FFFF @=0X1274
SYSCALL @0XC000931C REXIT( 0XDEADBEEF, ) =
FREEFRAMES(PD= 0X80..8, 1.FRAME= 0X0, N= 0X70, UP= 1, COREMAP
FREEFRAMES(PD= 0X80..8, 1.FRAME= 0X7FFFF, N= 0X2, UP= 0, COREMAP
FREEFRAMES(PD= 0X80..0, 1.FRAME= 0X3FF, N= 0X4, UP= 0, COREMAP
FREEFRAMES(PD= 0X80..0, 1.FRAME= 0X1, N= 0X2, UP= 1, COREMAP
FREEFRAMES(PD= 0X80..0, 1.FRAME= 0X201, N= 0X1, UP= 0, COREMAP
```

8.3.2.11 *P4* leaves main with 0xDEAD0000.

```
LOADPROC 0X4 FROM 0X7A -> 0X76, 1
RET0= 0X3, RET1= 0X9, FAIL= 0
|-> USER-TRAP=0X14 PID=0X4 PS=0X374FFFF @=0X1274
SYSCALL @0XC000931C REXIT( 0XDEAD0000, ) =
FREEFRAMES(PD= 0X80..8, 1.FRAME= 0X0, N= 0X70, UP= 1, COREMAP
FREEFRAMES(PD= 0X80..8, 1.FRAME= 0X7FFFF, N= 0X2, UP= 0, COREMAP
FREEFRAMES(PD= 0X80..0, 1.FRAME= 0X3FF, N= 0X4, UP= 0, COREMAP
FREEFRAMES(PD= 0X80..0, 1.FRAME= 0X1, N= 0X2, UP= 1, COREMAP
FREEFRAMES(PD= 0X80..0, 1.FRAME= 0X201, N= 0X1, UP= 0, COREMAP
```

8.3.2.12 *P3s* segmentation-fault

comes back, and allocates memory again. Now no frame is left and a segmentation-fault is caused while fetching the char.

```
loadProc 0x3 from 0x175 -> 0x76, 1
|-> User-Trap=0x14 PID=0x3 PS=0x374FFFF @=0xA54
SysCall @0xC0009C7C sbreak( 0x80000, ) =
-> Estabur( nt = 0(0), nd = 80(77), ns = 2(2), sep = 0(0), xrw = 4)
size= 0x81 0x81
<- Estabur E_NO_MEM !!
ret0= 0xC, ret1= 0x9, Fail= 57344
0 cpu= FF -> CC | 3 cpu= 7A -> 61 |
|-> User-Trap=0xC PID=0x3 PS=0x36DFFFF @=0xA5C
|-> User-Trap=0x16 PID=0x3 PS=0x376FFFF @=0x2B8
USER SEG FAULT !!
Sending sig = 0xB -> 0x3
freeFrames(PD= 0x80..8, 1.Frame= 0x0, n= 0x77, Up= 1, CoreMap
freeFrames(PD= 0x80..8, 1.Frame= 0x7FFFF, n= 0x2, Up= 0, CoreMap
freeFrames(PD= 0x80..0, 1.Frame= 0x3FF, n= 0x4, Up= 0, CoreMap
freeFrames(PD= 0x80..0, 1.Frame= 0x1, n= 0x2, Up= 1, CoreMap
freeFrames(PD= 0x80..0, 1.Frame= 0x201, n= 0x1, Up= 0, CoreMap
```

8.3.2.13 Idle forever

is done by the operating system without processes.

8.4 Writing own programs

The test-programs are written using the "syscall-library". At the moment the program is always a replacement for the init process. Whenever a valid init-program exist, the location and filename should be modified. Within the SYS/INIT.C sources, some examples are shown to OPEN/CREATE files and write into them.

8.4.1 Compiling and linking

Look at the sample SYS/INIT.C. This file is compiled and linked separate from the operating system. Within the Makefile in the kernel-dir the target init.e exist, which does the following :

```
INIT.E: SYS/INIT.C SYS/SYSCALL.S
  $(CC) -O $@ SYS/INIT.C SYS/SYSCALL.S
```

Therefore it is compiled in the usual way. Now the executable has to be put on the disk.

8.4.2 The file-system

The target $(DISK), which is defined to $(BUILD)/RUN/SYSTEM.DISK, uses the MKFS tool to generate the file-system :

```
$(DISK): init.e proto.root
  $(BUILD)/bin/mkfs $(BUILD)/run/system.disk proto.root
```

The file PROTO.ROOT contains the description of the file-system and the files on it. In our case PROTO.ROOT would have the left content and produce the right file-system :

```
../../../../BUILD/RUN/MBR.OUT
2000 640
D–777 0 0
DEV D–777 0 0
  CONSOLE C–644 0 1 0 0
  $
ETC D–755 0 0
  INIT —755 0 0 ./INIT.E
  $
$
```

```
/
/DEV/
/DEV/CONSOLE
/ETC/
/ETC/INIT
```

The first line specifies the master-boot-record to use, it is generate with one of the simulator tools. The third line creates the root-directory with read/write/execute permissions enabled for anyone. The owner and group is reflected by the two last 0s. In the DEV-directory a character device console is created with major-number 0 and minor-number 1. Within the ETC-directory the file INIT is created using INIT.E.

8.5 Unix v7 on a PDP-11 Simulator

This section describes[6] how to get the PDP-11 simulator run with Unix v7. It is much better described in the distribution at Dr. Geisses Unix v7 distribution `http://TeleXX.mni.FH-Giessen.de/PDP11-UNIX`. The original 7 tape binaries are converted into a tape with a tool from Dr. Geisse into a tape which is understand by the simulator. Then starting the simulator and setting up Unix v7 in the same way it ever was ;-)

First download the distribution, extract it, and change into them with :
TAR XVZF UNIX-V7-1.TAR.GZ
CD UNIX-V7/
Compile the simulator :
CD SIM/
MAKE PDP11
Compile the tape tool and make the tape :
CD ../MKTAPE
MAKE
CD ../V7
CP F0 F1 F2 F3 F4 F5 F6 ../MKTAPE
CD ../MKTAPE
./MKTAPE
CD ../RUN/
CP ../SIM/PDP11 ../MKTAPE/UNIX_V7.TM .
./PDP11 SETUP.CONF
RUN 100000
^E
RUN 0
create file-system (':' indicates the shell prompt and '?' indicates a question you should answer, don't type them !)
: TM(0,3)
? 5000
restore data
: TM(0,4)
? TM(0,5)
? HP(0,0)
<RETURN>
Boot Unix to make first setup
: HP(0,0)HPTMUNIX
MV HPTMUNIX UNIX
make device entries
CD DEV
MAKE RP04
MAKE TM
/ETC/MKFS /DEV/RP3 153406

[6] "tells which keys has to be pressed"

DD IF=/DEV/NRMT0 OF=/DEV/NULL BS=20B FILES=6
RESTOR RF /DEV/RMT0 /DEV/RP3
<RETURN>
Now wait until the file-system is extracted, and create boot-block
/ETC/MOUNT /DEV/RP3 /USR
DD IF=/USR/MDEC/HPUBOOT OF=/DEV/RP0 COUNT=1
umount and stop the machine
/ETC/UMOUNT /DEV/RP3
SYNC
SYNC
^E
QUIT
Now standard procedure to start the simulator is :
./PDP11 RUN.CONF
BOOT
exit single-user-mode and login as ROOT with password ROOT
^D
ROOT
ROOT

Chapter 9

Glossary

$2,$3	are the both return value registers, used by the LCC on the Eco32 processor.
array	is equivalent, to a field in C. The word field is used in this document as struct field !
clicks	is the smallest potion in which memory could be allocated, one whole frame, 4096 bytes.
core	Is the equivalent to "core memory", it reflects the physical main memory. Core is always allocated in whole frames.
field	is always one field within a struct, no an array !!
FORK	is a system-call to duplicate the calling process.
P_ADDR	Is located in the proc struct. Within the original memory management, it pointed to the physical location of the process image. In the new memory management it points to the U-Area-Page-Table. Pointing in this context is reflected through a physical frame number !
PROC-struct	Contains all information about a process which are need while it's swapped out. These are information like, swap/core load address or process size.
swap-page-number	$= SwapDiskBlockNumber * (CoreFrameSize/DiskBlockSize)$, in this case $CoreFrameSize = 4096$, $DiskBlockSize = 512$
text	is equivalent to code or instructions within this document. It references the code-segment.
U-Area	is allocated once per process. It contains all necessary informations needed while a process is in core. The U-Area contains : U-Area-Page-table, user-struct, user page-tables and kernel-stack.

Index

Bibliography

[Anjuta] Anjuta Dev Studio, fast and comfortable IDE, http://Anjuta.org

[Geisse] Prof. Dr. H. Geisse, ECO32 Project, http://telexx.mni.fh-Giessen.de/ECO32

[Lions] "Lions Commentary on Unix v6", John Lion, Annabooks, ISBN 1-57398-013-7

[LyX] LyX - The Document Processor, http://www.LyX.org

[XFig] XFIG Drawing Program for the X Window System, http://www.XFig.org