Operating Systems

or how Unix V6 really works

Contents

Chapter 0

What are Operating Systems Good For?

To see why operating systems are in widespread use, imagine a "naked" computer, consisting of a processing unit, which is able to fetch instructions from memory, and to interpret and execute these instructions.

0.0 Program loading

A program has to be entered into memory before it can be executed by the processing unit. Thus a computer without an operating system needs a device, called a "console", which can be used to change the contents of the memory. The memory can be viewed as an array of numbered cells, each containing a nonnegative integer. With switches on the console you enter the binary digits (a.k.a bits)of the number of the cell – which is called address – and then the bits of the contents of the cell. If you are finished entering the whole programm, you enter the address of the first instruction (i. e. the number of the memory cell) and press some start button to execute the program.

This whole cumbersome process is called "loading a program". If the program to be loaded is already stored on a permanent medium like a disk or a tape, the loading can be done by the computer executing another program, the so called loader. The loader has to control the disk in order to copy the program into memory. Loading and controlling the execution of other programs is one of the tasks operating systems do for you.

0.1 Better interface to the hardware

The user written program usually has to do some I/O, which means it has to control I/O devices. For this task, the programmer has to acquire intimate knowledge of how the device is to be programmed. This knowledge as well as the resulting program depends heavily on the particular type and model of the device. So another task of the operating system is to provide a device and model independent application programm interface (API), which hides the hardware specific details in so called "drivers". This makes it easier to adapt the application programs to another disk model.

A typical disk driver presents the contents of a disk to the application program as an array of fixed sized sectors (ranging from 256 to 8192 Bytes). There is a need to partition this lump of sectors in files, which are addressed by names, not by sector numbers. The mapping between filenames and sector numbers as well as creation, changing and deletion of files is done by a filesystem, which is one of the most important parts of an operating system.

Furthermore the operating system provides a user interface, by which a user can specify the program to be loaded and the files to worked on by the loaded program. This can be a so called command interpreter (MS-DOS) or a graphical user interface as implemented by the Finder of the Macintosh Operating System or Windows "Explorer".

0.2 Multiprogramming / Multiuser

The need to run several programs simultaneously is addressed by a multiprogramming system. The original use of this feature was to enhance the throughput of a so called "batch", which is a set of programs. If one program were run after the other, the processor would be idle during the time an I/O request is serviced. In a multiprogramming system the CPU is kept busy executing some other program of the batch. Even though this switching from one program to another adds some overhead, the time needed to complete the whole batch turned out to be shorter.

If the computer is used interactivly, a multiprogramming system allows the user to enter a command while another command is still running. This way the computer can be used editing a file while downloading another file at the same time.

Multiprogramming is pushed further by interactive systems which let more than one person use the system simultaneously. Operating systems with this feature are termed "multiuser systems" or "time sharing systems", because the processor time is shared by several to many users.

The original incentive for a time sharing system was to share the expensive computer. But even today with cheap computers it is necessary to centralize and share resources like printers, files, communication lines, databases, tape libraries and last but not least system administration. This is done by central computers, so called servers, which let thousands of users login simultaneously.

The multiuser operating system has the obligation to allocate the resources (processor, memory, I/O devices) to competing programs. Since a crash of the system might effect thousands of users, it has to protect itself and simultaneously running programs from each other.

Another challange introduced by multiuser systems is security. The operating system has to provide some means to control access to the files. Not every file may be changed or read by every user.

0.3 UNIX V6 and PDP-11

In this course we will study how operating systems are implemented. We will use the UNIX Version 6 (V6) running on the Programmable Data Processor 11/40 (PDP-11/40) as a case study.

Although released 1975, V6 still qualifies for an operating system course, because

- it is small enough to be comprehended by a single person. The kernel source counts at about 9000 lines – compare this to MINIX, which needs 27000 lines to implement only a subset of V6.
- it is powerful enough i. e. it is a multiprogramming/multiuser operating system. Since it was heavily used by its programmers (Ken Thompson, Dennis Ritchie) it exhibits all the features needed and – not more. The core is not hidden by layers of code designed to serve the needs of marketing departments.
- The complete source is available – well documented by the original authors and by others who used V6 in operating system courses - notably by John Lions whose "Commentary on UNIX 6th Edition" was heavily used in preparation of this course.
- Most of the V6 features are employed in modern operating systems. So V6 being outdated does not mean that the knowledge of how V6 works is outdated as well.

The PDP-11 was in widespread use during the seventies and eighties. Its hardware architecture heavily influenced the design of contemporary CISC microprocessors, especially Motorolas 68000 Series and regrettable to a lesser degree Intel's 80x86 family of processors.

UNIX V6 and the PDP-11 heavily influence both contemporary operating systems and hardware architecture. This makes them a model for computing in the 21st century.

0.4 Overview of the rest of this course:

This course takes a bottum up approach:

Chapter 1 starts describing the hardware, i. e. CPU and I/O devices. It introduces the notions of machine and assembler program. This will enable you to understand how the installation and booting programs of V6 are implemented. At the end of this chapter you should be able to install V6 on the SUPNIK Simulator (a.k.a. SIMH), work with the V6 documentation, be able to configure and install a new kernel.

Chapter 2 is about process control, interrupt handling and memory management. At the end of this chapter, you should know about the synchronization problem and its solution in the V6 kernel, the pitfalls of interrupt service routines and the way the hardware supports memory management.

Chapter 3 will cover the I/O subsystem, i. e. the programming of devices and the management of blocked I/O.

Chapter 4 is devoted to the filesystem, i. e. the data structures and algorithm that name a file and allocate disk blocks to a file.

Chapter 5 is about the implementation of the programs needed to build programs, namely assembler, C compiler, linkage editor. It describes implementation of the part of the kernel that loads programs.

0.4.1 Courseware

There are some files at `http://www.ba-stuttgart.de/ helbig/os`:

- `script/` contains (part of) the script to this course.
- `v6/doc/` contains most of the documentation distributed with V6 in postscript format. The Programmers manual is provided as a set of html pages.
- `v6/dist.tap` is a file ready to be used by SIMH. It resembles the original distribution tape for Unix V6.
- `pdp11/doc`
  Instruction Set and H/W Interface to some PDP-11 devices These papers are meant to be studied in parallel to the script.
- `pdp11/progs`
  Some PDP-11 machine programs stored as SIMH command files.

Chapter 1

Programming the PDP-11

I am a very bottom-up thinker

Ken

This tutorial stresses those aspects of machine programming that are needed to study the system programs of Unix. These include parts of the C compiler, the assembler, the link editor, the boot programs, and the kernel. Robert M. Supnik's PDP-11 simulator (SIMH) is employed to provide hands on experience.

SIMH: If you do not have access to a real PDP-11 with Unix V6 running, you need to use a PDP-11 simulator. This lecture assumes that you use Bob Supniks simulator, available for download at http://simh.trailing-edge.com/

1.0.0 Machine Programs and Assembler Programs

A "machine program" is a sequence of bytes that contain numbers in the range $[0, 2^8]$, that is, 8-bit numbers. Each byte is identified by its "address". Instructions are encoded in "words", that is two adjacent bytes. The address of a word is even and equals the address of its "low byte". The odd addressed byte of a word is its "high byte". "Low" and "high" indicate the weight in a base 2^8 number system. The low byte has the weight one and the high byte has the weight 2^8. So, the number stored in a word is

$$word = lowbyte + highbyte * 2^8.$$

Both words and addresses are in the range $[0, 2^{16}]$. The instructions operate on zero, one or two operands, which are byte or word encoded.
Notation: "Numbers" mean nonnegative integers. The terms "unsigned integers" and "signed integers" are widely used to mean nonnegative integers respective integers. The notation "(un)signed" suggests, that there are signs which are (not) attached to integers. But this is misleading.

Exercise: A word has the value 1000. What are the values l, h of its low and high bytes?

The encoding of instructions is defined in the "PDP-11 Instruction Set" (filed in pdp11/doc/cpu), which you should read accompaning this script.

The PDP-11 supports two operand types, namely integers and boolean arrays. The encoding of integers is explained in section 1 below. A boolean value is encoded as a number with "true" ¡-¿ 1 and "false" ¡-¿ 0. The PDP-11 instructions support two sizes of boolean arrays, namely eight and 16. These arrays are encoded in bytes respective words. A word encoded boolean with entries b0, b1, ... b15 is represented by the number

$$b0 * 2^0 + b1 * 2^1 + ... + b15 * 2^{15}.$$

Note the difference between a number and its notation. "Hex" in "hex number" is not an attribute of the number (like e.g. "even") but indicates the notation of a number, namely as a sequence of digits with base 16. Numbers that encode instructions are often notated octal, because the instruction's code is subdivided in fields of three bits each.

Exercise: The odd numbered entries of a byte encoded boolean array are true, the others false. What is its encoding?

People and programming tools use octal or hexadecimal notation for a number just because it is stored in words or bytes. This is a bad habit. Use common sense when choosing the base. In this script, the C language convention is used: A leading 0 indicates base 8, a leading 0x indicates base 16. Otherwise decimal notation is used throughout. Numbers that encode instructions, or numbers in assembler programs however are notated with base 8, even if not starting with a "0".

An assembler program differs from a machine program by naming numbers (symbols) and operations (mnemonics). The assembler builds instructions, addresses and operands from mnemonics, symbols and numbers and writes a sequence of assembled words, thereby translating an assembler program to a machine program. Symbols, mnemonics and numbers are notated using the ASCII character set.

The paper "Unix Assembler Reference Manual" (filed in v6/doc/as.ps) defines syntax and meaning of assembler programs–more comprehensive and less verbose as seems appropiate in this tutorial.

The symbols "r0", "r1", ... "r7", "sp" and "pc" name the general registers.

The assembler keeps track of the current location, that is, the address of the currently assembled word, in the "location counter". Its name is ".", also known as "dot". At the start of an assembly dot is set to the first address of the program, known as the program's "origin", dot is then incremented by 2 with every word assembled.

An "expression" is a number or a symbol, possibly combined with numbers and symbols by operators like "+" or "-". The expression is evaluated during assembly time. This differs from high level languages like "C", where expressions might be translated to a sequence of machine instructions and evaluated at run time.

The assembler implicitly assigns certain "types" to symbols, numbers and expressions. These types include "relative" and "absolute address", "register", and instructions. "Relative" here means the address is relative to the program's beginning, its origin.

The type of an expression can be modified by the '^' operator. E.g. the assembler statement ".=400^." sets the location counter to 0400. The type casting operator "^" casts the type of the left hand expression to the type of the right hand expression. In the example "400^.", the type of ".", which is "relative address", is cast upon the type of "400", which is "absolute address". A line starting with a symbol followed by a colon (":") defines a "label". Its value is set to dot and its type is relative address.

The assembler expects numbers to be notated octal.

An "instruction statement" consists of a mnemonic and–depending on the the number of operands–zero, one or two operand fields. With two operand fields, the source field comes first, followed by a ",", followed by the destination field, e.g., "mov src,dest". An operand field specificies an address mode and a register. Four even numbered modes specify the operand's address (direct addressing) and four odd numbered modes specify the address of the operand's address (indirect addressing). The PDP-11 architecture is "regular", that is, you can use all addressing modes in an

operand field, regardless of the instruction. Furthermore, the registers are "general", that is, you can use all eight registers in the operand specification, regardless of the instruction. Regularity and generality means few simple rules define a powerful system–the hallmark of good design.

The address modes and their names follow.

- Mode 0: register
  Specified by: A register.
  Meaning: The register holds the operand.
  Example: r0

- Mode 1: register indirect
  Specified by: A parenthesized register.
  Meaning: The register holds the operand's address.
  Example: (r0)

- Mode 2, 4: auto-increment; auto-decrement
  Specified by: A parenthesized register suffixed by "+" or prefixed by "-".
  Meaning: The register is incremented after resp. decremented before it is used to address the operand.
  Examples: (r0)+, -(r0)

- Mode 6: indexed
  Specified by: An expression followed by a parenthesized register.
  Meaning: The sum of the expression and register addresses the operand. The value of the expression is assembled in the word following the instruction.
  Example: 10(r0)

- Mode 1, 3, 5, 7: `~ indirect`
  Specified by: A "*" followed by an even mode (i.e. direct) specification.
  Meaning: Replace "operand" by "operand's address" in the meanings of the corresponding direct mode.
  Examples: r0, (r0)+, -(r0), 10(r0)

So address mode 1 can be specified by a parenthesized register or by a "*" followed by a register–syntactic sugar even in assembler! When you drop the expression in mode 7, "0" is assumed.

Exercise: What do you get, if you drop the expression in mode 6?

In modes 6 and 7 the program counter is used implicitly to address the word holding the value of the expression. When the program counter is used implicitly, it is immediately incremented by 2. This is the case whenever an instruction word is fetched or whenever address modes 6 or 7 are encountered.

Exercise: With each of the instructions at address 0, what will be the PC after execution of the instruction?

a) inc r0
b) inc 10(r0) (Note: '10' is octal, this is an assembler statement.)
c) inc *10(r0)
d) inc pc
e) inc (pc)
f) inc 10(pc)
g) inc *10(pc)
h) inc (pc)+

The assembler offers shortcuts to explicitly specify the PC in address modes and furthermore calculate PC-relative offsets from the location counter.

- Mode 2; immediate (register indirect)
  Specified as: The letter "$", followed by an expression.
  Meaning: The value of the expression is the operand. It is stored in the word following the instruction.
  Example: $10, short for "(r7)+; 10"

- Mode 6; relative (indexed)
  Specified as: An expression.
  Meaning: The expression denotes the address of the operand. The offset relative to dot+2 to is assembled in the word following the instruction.

  Example: 10, short for 2(r7)
  In this (and the next example) it is assumed that the word holding the offset is at 4. The offset is relative to .+2, which is 6, and thus turns out to be 2.

- Mode 3, 7; absolute (auto-increment indirect)
  relative indirect (indexed indirect)
  Specified as: A prefixed "*".
  Meaning: Replace "operand" by "operand's address" in the meanings of the direct modes 2 and 6.
  Example: *$10, short for "*(r7)+; 10";
  *10, short for *2(r7)

In both the relative and absolute mode the expression denotes the operand's address, e.g., "10" and "*$10" specifiy the operand at location 010 = 8. The choice between these modes matters when you move the whole machine program from one address range to another. If the operand is moving with the programm, use relative mode; whereas if the operands location is independent of the programs location, use absolute mode.

Note the overloading of the terms "relative" and "absolute": When talking about the types of assembler symbols, "relative" means a value relative to the origin of the program, whereas in connection with address modes, "relative" means an offset relative to the program counter.

Exercise: How many words need to be assembled for each of these instructions:

```
add   r1,r2
sub   *r1,*r2
cmp   (r1),(r2)
mov   (r1)+,-(r2)
mov   *(r1)+,-*(r2)
mov   $10,r2
mov   $10,10
```

Exercise: Assume each of the following instructions start at 0. Each of the other words in memory hold twice their address, that is the word at 2 holds 4. Which words are affected if the instruction at 0 is:

```
mov   (r1),(pc)+
mov   r1,*(pc)+
mov   r1,(pc)
mov   r1,*(pc)
```

Exercise: Assemble this instruction:

```
br    .+20
```

1.0.1 Signed Integers, Unsigned Integers and Addresses

The predominant operand types are signed integers, unsigned integers and addresses. In the PDP-11 instruction set, addresses and unsigned integers don't differ. Furthermore, the additive integer instructions (add, sub, neg, inc and dec) are implement both signed and unsigned arithmetic.

To understand how one instruction works for signed and unsigned operands, note that the result of a word operation is truncated to 16 bits, which means, it is taken "modulo N" with N = 2^{16}. In other words, the additive instructions implement modulo N arithmetic with operands and results in [0, N].

For the upcoming discussion you need a definition of integer division and the modulo operator:

For any integer a and any positive integer b, the modulo and division operators are defined by these two conditions:

$$(DEF0)(a\ div\ b) * b + a\ mod\ b = a0 <= a\ mod\ b < b$$

Exercise: Compute 13 mod 10 and 13 mod 100.

DEF0 means that the div operator rounds towards minus, since the remainder mod is nonnegative. ANSI-C leaves the rounding direction of the "/" operator up to the implementation of the compiler. In PDP11-C, Java and all C-compilers that I enjoyed, integer division rounds towards zero, not towards minus. This matters if the dividend is negative. Furthermore, "div" and "mod" are defined only when the divisor is positive, whereas "/" and "%" are defined for negative divisors as well, again, not by the C-language but by the implementation of the compiler.

Example:

$$(-15)\ div\ 12 = -2; (-15)\ mod\ 12 = (-15) - (-2) * 12 = 9$$

but

$$(-15)/12 = -1; (-15)\%12 = (-15) - (-1) * 12 = -3$$

Exercise: Compute (-13) mod 10 and (-13) % 10?

Example: The following ANSI-C function computes i mod k regardless of the implementation of the "%" operator:

```
int
mod(int i, int k) {
        int m;

        m = i \% k;
        if (m < 0)
                return m + k;
        else
                return m;
}
```

From (DEF0) you can derive one simple but fundamental property:
(MOD0) $a\ mod\ b = (a + k * b)\ mod\ b$, for any integer a, k and positive integer b
You certainly are eager to see the above properties proved, aren't you? OK:
With $q0 := a\ div\ b, q1 := (a + k * b)\ div\ b$, DEV0 yields:

$$a\ mod\ b = a - b * q0$$

and

$$(a + k * b)\ mod\ b = a + k * b - q1 * b = a + b * (k - q1)$$

which shows that $(a\ mod\ b)$ and $((a + k * b)\ mod\ b)$ differ by a multiple of b. Since DEF0 guarantees both terms being in [0, b], they must be equal. This proved MOD0.

Exercise: Prove that a mod b differs from a by a multiple of b, that is:

$$(MOD1) a\ mod\ b = a + k * b$$

for a suitable k.

Exercise: Use the MOD1 and MOD0 to prove that $(amodb)\ mod\ b = a\ mod\ b$.

Exercise: Express low and high byte of a word by the div and mod operators.

This ends the treatise of modulo arithmetics.

Back to the PDP-11 instructions which now read:
add(m,n) = (m+n) mod N
sub(m,n) = (n-m) mod N
neg(m) = (-m) mod N

To get rid of the mod operator for neg(m), apply its definition:
The case m = 0 yields:
neg(0) = (-0) mod N = 0 mod N = 0

The case m ¿ 0 yields:

$$\begin{aligned} (-\mathrm{m})\ \mathrm{div}\ \mathrm{N} &= -1, \text{ since } -\mathrm{N} < -\mathrm{m} < 0, \text{ and further} \\ (-\mathrm{m})\ \mathrm{mod}\ \mathrm{N} &= (-\mathrm{m}) - (-1) * \mathrm{N} \\ &= (-\mathrm{m}) + \mathrm{N} \\ &= \mathrm{N} - \mathrm{m} \end{aligned}$$

The number N-m is called the (two's) complement of m. So, unsigned negation computes the complement and unsigned subtraction adds the complement.

In order to reuse these unsigned operations for signed arithmetic you have to choose an encoding that maps nonnegative integers to themselves. The encoding of negative integers needs to mirror the negation of unsigned arithmetic, leading to property

1. i is encoded as i, and -i is encoded as N-i. (for positive i)

   Furthermore, you want the range of encodable integers to be symmetrical around zero, in other words, you want that

2. the negation of an encodable integer leads to an encodable integer.

Properties (1) and (2) leaves you no choice but to define the so called "two's complement encoding" which maps $[-N/2, N/2]$ to $[0, N]$:

$[-N/2, 0) - - > [N/2, N), i - - > i + N$
$[0, N/2) - - > [0, N/2), i - - > i$

MOD0 lets you reformulate this encoding somewhat shorter:

$$[-N/2, N/2) - - > [0, N) i - - > i mod N$$

Exercise: With N = 100 instead of 2^{16}, and which of 1, -1, 30, -30, 50, -50 are encodable? Compute their encodings.

Resume:

- The additive instructions are defined for nonnegative integers.
- They implement $modulo\ 2^{16}$ arithmetic.
- With the two's complement encoding of negative integers, the unsigned operations don't differ from signed operations.
- The range of encodable integers is $[-N/2, N/2]$
- A negative integer i is encoded as $N + i$.
- An integer i and its encoding m are related by $m = i\ mod\ N$

So far we have shown how integers need to be encoded if you want to reuse unsigned operations for integer operations. But we still have to prove that this encoding in fact lets you reuse unsigned operations. Let i and j be encodable integers, and m resp. n their encodings. We need to prove, that

add(m, n)	encodes i+j,
sub(m, n)	encodes i-j
neg(n)	encodes -i

provided that i+j, i-j and -i are encodable integers.

```
With N=10 and i = 2, j = -1 you get m=2, n=9 and
        add(m, n) = add(2, 9) = 11\ mod\ 10 = 1 which encodes (2+(-1)).
        sub(m, n) = sub(2, 9) = -7\ mod\ 10 = 3 which encodes (2-(-1)).
        neg(n) = neg(2) = -2\ mod\ 10 = 8 which encodes (-2).

To prove that add can be reused, we calculate:
        encoding of i+j = (i+j) mod N              (definition of encoding)
                        = (m+n+k*N) mod N          (MOD1 twice)
                        = (m+n) mod N              (MOD0)
                        = add(m, n)                (definition of add)
```

Exercise: Express the integer k above by means of the div operator.

Exercise: Prove, that neg and sub can be reused.
Note that property (2) is broken by the smallest encodable integer $-N/2$, whose encoding $-(N/2) + N = N/2$ is its own complement. Therefore the decoded result of $neg(N/2)$ is $-N/2$ instead of $N/2$, which misses the range of encodable integers by one.

If i is negative, its encoding is $>= N/2$, which means the high bit is set. That's why the high bit is called the "sign" bit. Most instructions set the N-flag to the sign bit of the result.

The "carry flag" (C-flag) is set if the result of add or sub had to be truncated to fit in [0, N). It is useful for comparing unsigned numbers and for coding multiprecision arithmetic. Note that neither the inc nor the dec instructions modify the carry flag.

It follows, that for any m, n in [0, N) you get:
m+n = N*C + add(m, n)
n-m = -N*C + sub(m, n)

Or, reformulated:
C = (m+n) div N,
C = (m-n) div N.

The overflow flag (V-flag) is set if the decoded result is not in $[-N/2, N/2)$. High level programming languages don't let you access these flags, which sometimes makes programming of arithmetic algorithms in C more cumbersome than in assembler. Implicit address calculations as specified by addressing modes do not affect any condition flags. Additive integer arithmetic is provided for byte encoded integers as well–with $N = 2^8$ instead of 2^{16}. Byte instructions need to convert bytes to words if the destination operand is a general register. This is done such that the word encodes the same integer as the byte. Let b be the value of the byte, i the integer encoded by n and w the value of the word.

- Case 0: b encodes nonnegative integer: b in [0, 2^7), i = b, and w = i = b is the word encoding of i.
- Case 1: b encodes a negative integer: b in [2^7, 2^8), $i = b - 2^8$, and $w = 2^{16} + i = 2^{16} + b - 2^8$ is the word encoding of i.

For an alternate view of this conversion, consider the high byte h and low byte l of w:

$$h = w \ div \ 2^8, l = w \ mod \ 2^8.$$

In case 0 you get:

$$h = b \ div \ 2^8 = 0;$$

$$l = b \ mod \ 2^8 = b$$

In case 1 you get:

$$w = 2^{16} + b - 2^8 = (2^8 - 1) * 2^8 + b$$

$$h = w \ div \ 2^8 = 2^8 - 1$$

$$l = w \ mod \ 2^8 = b$$

In both cases l equals b and each bit of h equals b's sign bit. In other words, this conversion "extends the sign" of b to h. "Sign extension" is to be applied whenever you need to convert an integer encoding to an encoding with more bits of the same integer.

Sign extension must not be applied when converting values that don't encode integers. Byte encodings of characters serve as a prominent source of trouble when sign extending 8-bit bytes to 16- or 32-bit words.

Both the PDP-11 machine language and the C language of Unix V6 support unsigned arithmetic only for 16-bit addresses. Integer arithmetic, byte to word conversion and integer comparisons assume signed arithmetic throughout. This makes the semantics easier to comprehend than the mess of "unsigned" and "signed" integers combined with "type propagation", as introduced by later versions of the C language.

Furthermore, the notion of "unsigned" characters is not needed with ASCII, whose codes happen to be encodings of nonnegative integers only and are thus not affected by sign extension. The problem of sign extending characters is solved by the Java language, which simply defines character codes to be unsigned integers and all other integer

types as being signed. In this respect, Java mirrors the simplicity approach of the PDP-11 machine language and C in Unix V6.

Exercise: The DIT is a computer very similar to a PDP-11, but based on decimal digits instead of binary digits. Its bytes hold two dits, its words two bytes. The DIT reuses additive unsigned arithmetic for signed arithmetic the same way as the PDP-11 and provides the same instruction set. Answer the following questions for the DIT:

What is the range of byte resp. word encodable integers?
What is the byte encoding of -45?
Which integer is word encoded by 600?

What are the results of each of the following instructions:

```
movb   $25,r1
movb   $65,r1
```

Which of the condition flags need to hold dits instead of bits to be useful?

Assume fifty is a word containing 50.
Which of the following instructions will set the V flag? What will be the the final value of fifty?

```
negb   fifty
neg    fifty
```

The same with neg first and negb second.

Exercise: Assemble this instruction, for a DIT and for a PDP-11:

```
br   .
```

1.0.2 An example

Its time to illustrate PDP-11 instructions, operand types, machine programs, and assembler programs by coding and running an example program.

Before you run a machine program you need to enter it in memory starting at the program's origin. This is known as "loading". Then set the PC to the address of the first instruction to be executed. This address is called the "entry" of the program. To inspect the contents of registers and memory (also known as the "state") stop the program to freeze the state.

SIMH: Loading, running and stopping programs is explained in chapter 3 of the simulator documentation (filed in simh/simh_doc.txt). See section 3.5 "Examining and Changing State", section 3.6 "Running Programs", and section 3.7.2 "Stopping Programs". Consult the PDP11 part of the documentation (filed in simh/PDP11/pdp11_doc.txt) for the names of registers and flags you want to access.

Program sum1 This program reads $n >= 0$ from the SR, sums up the first n nonnegative integers, writes the sum to the DR and halts. The sum turns out to be $n * (n-1)/2$, which does not fit in a word for $n > sqrt(2N)$. But, since $n < 2^{16}$, the result is $< 2^{32}$ and fits in two words. Registers r1 and r2 hold the high respective low word of the sum.

```
machine program          assembler program
addr    contents
000000                   . = 400^.
                         /* r0:   number of integers added so far */
                         /* r1r2: sum of integers in [0, r0) */
                         /* r3:   n */
                         io      =       177570  / switch and display register
000400  013703                   mov     *$io,r3 / read n from SR in r3
000402  177570
000404  005000                   clr     r0      / set up loop invariant:
000406  005001                   clr     r1
000410  005002                   clr     r2      / r1r2 == sum of [0, r0)
000412  020003           loop:   cmp     r0,r3   / "while (r0 != n) {"
000414  0x0304                   beq     stop
000416  060002                   add     r0,r2
000420  005501                   adc     r1      / r1r2 == sum of [0, r0]
000422  005200                   inc     r0      / r1r2 == sum of [0, r0)
000424  0x01FA                   br      loop    / "}"
000426  010137           stop:   mov     r1,*$io / display high word of sum
000430  177570
000432  000000                   halt
000434  010237                   mov     r2,*$io / display low word of sum
000436  177570
000440  000000                   halt
```

The assembler calulates PC-relative offsets at two locations, namely:

```
at 414: beq instruction
  stop - (dot+2) = 0426 - 0416  = 010 = 8
  halve to get word offset: 4

at 424: br instruction
  loop - (dot+2) = 0412 - 0426 = -014 = -12
  halve to get word offset: -6
  8-bit complement: 2^8 - 6 = 0x100 - 6 = 0xFA
```

Input/Output through the panel: Both the switch register and the display register are at 177570. The hardware uses the direction of data flow (read vs. write) to choose between the two registers. Reading at 177570 selects the switch register whereas writing at 177570 selects the display register. The panel's switches and lights serve as a model for I/O programming. You access the I/O registers the same way you access memory. "Memory mapped I/O" as it is called, employs ordinary instructions to control I/O, saving you the trouble to learn special I/O instructions. The addresses of I/O registers are independent of the program's origin which is reflected by employing absolute address modes.

SIMH: On default, addresses are taken as being bus addresses. This matters when accessing the I/O page; use the switch "-v" to indicate virtual addresses or enter 22-bit bus addresses. Use "e -d dr" to view the display register as a decimal number. The command "e -v 177570" will not work because SIMH-PDP11, like a real PDP-11, chooses the switch register if reading at 0177570.

SIMH: The above machine program is saved in pdp11/progs/sum1.sim as a sequence of SIMH commands. To load it, pass the filename via the command line to SIMH. The file includes two test runs, one to compute sum[0, 100) and one to compute sum[0, 0100000).

SIMH: The switch "-m" of the deposit command makes SIMH look up operation codes and calculate PC-relative offsets. See section 2.11 "Symbolic Display and Input" of its documentation (filed in simh/PDP11/pdp11_doc.txt).

SIMH uses the character "#" instead of "$" and "@" instead of "*". The "-m" switch is close to what a real assembler does, the only thing left are user defined symbols.

1.0.3 an I/O program

In this section you learn how to write programs that access peripheral devices, namely the tape drive (TM11/TU10), and the serial line controler (KL11), which attaches to a teletype. Consult "PDP-11 Devices" for a hardware description. (filed in pdp11/doc/devs).

SIMH: Consult section 2.9 (TM11) in the "PDP-11 Simulator Usage".

Program ltap - load from tape This program copies the first block from tape to memory at 0. It assumes, that the tape is mounted at load point, i.e., positioned before the first block. The program waits until the I/O completes and then jumps to 0, thus starting the program just copied. Of course, this is meaningful only if the block contains a program with origin 0. This is the case with the V6 distribution tape, where the first block contains a program that serves to load other programs needed for installing Unix on a disk. Ltap's origin is 0100000, so it won't be clobbered by the block being copied. The block size is assumed to be 512 byte, which is the default block size of Unix.

```
machine program          assembler program
addr    contents
000000                   . = 100000^.
100000                   mtcs=172522     /MT control and status register
100000                   mtc=mtcs+2      /MT byte count register
100000                   mta=mtc+2       /MT memory address register
100000  005037                  clr *$mta
100002  172526
100004  012737                  mov $177000,*$mtc / complement of 512_10
100006  177000
100010  172524
100012  012737                  mov $060003,*$mtcs
100014  060003
100016  172522
100020  105737                  loop: tstb *$mtcs       / wait for completion
100022  172522
100024  0x80FD                  bpl loop
100026  005007                  clr pc          / jump to zero
```

Run this program with the distribution tape mounted at load point. The tape is filed in v6/tapes/dist and ltap in pdp11/progs/ltap.sim. If everything works well, the installation program will greet you with the "=" prompt. Stop it for now, you'll use it later to install Unix.

Ltap is an example of a "boot" program. Its only purpose is to load another program after power on. The first boot program either has to be keyed in manually or to be loaded from nonvolatile memory (ROM) by the hardware. Since both user time and ROM are expensive, boot programs tend to be short. They load a second boot program from the first block of a tape or disk to memory at 0. The second boot program is large enough, i.e., 512 bytes, to interactively ask the user for the name of program, to locate the specified program on tape and load it.

The second boot program communicates with the user via a teletype (TTY), which is a keyboard and a printer connected to the PDP-11 by a KL11 device as described in "PDP-11 Devices", Section 3.

SIMH: See section 2.2.3 and 2.2.4 for simulator usage of the DL11 device, which is an advanced version of the KL11.

1.0.4 Programming the TTY

This program reads two decimal digits from the console keyboard and prints the product on the console printer. It uses the stack for subroutine linkage, so it has to be located out of the yellow area. To provide feedback to the user, every character read from the keyboard will be echoed onto the printer, the program prompts with a "*". The user is then expected to type the digits and the carriage return key, without intervening white space.

```
machine program          assembler program
addr    contents
                         icsr=177560     / input control and status
                         ibuf=177562     / input buffer
                         ocsr=177564     / output control and status
                         obuf=177566     / output buffer
000000                   .=400^.
000400  012706                  mov     $160000,sp      / initialize stack
000402  160000
000404  112700           loop:   movb    $'*,r0 / out the prompt
000406  000052
000410  004767                   jsr     pc,out
000412  000114
000414  004767                   jsr     pc,in   / read first digit
000416  000124
000420  012702                   mov     $0-'0,r2 / convert from ascii to int
000422  177720
000424  060002                   add     r0,r2   / r2=first digit
000426  004767                   jsr     pc,in   / read second digit
000430  000112
000432  012703                   mov     $0-'0,r3 / convert from ascii to int
000434  177720
000436  060003                   add     r0,r3   / r3=second digit
000440  004767                   jsr     pc,in   / read carriage return
000442  000100
000444  012700                   mov     $12,r0 / out line feed
000446  000012
000450  004767                   jsr     pc,out
000452  000054
000454  070203                   mul     r3,r2   / r2=0, $r3=r2*r3$
000456  071227                   div     $12,r2 / r2=high digit, r3=low digit
000460  000012
000462  010200                   mov     r2,r0   / convert high dig. to ascii
000464  062700                   add     $'0,r0 / and out
000466  000060
000470  004767                   jsr     pc,out
000472  000034
000474  010300                   mov     r3,r0   / convert low digit to ascii
000476  062700                   add     $'0,r0 / and out
000500  000060
000502  004767                   jsr     pc,out
000504  000022
000506  012700                   mov     $12,r0 / out line feed
000510  000012
000512  004767                   jsr     pc,out
000514  000012
000516  012700                   mov     $15,r0 / out carriage return
000520  000015
000522  004767                   jsr     pc,out
000524  000002
000526  0x01D6                   br      loop
```

```
000530  105737          out:    tstb    *$ocsr / wait until last character
000532  177564
000534  0x80FD                  bpl     out     /   is sent to the printer
000536  010037                  mov     r0,*$obuf / send this character
000540  177566
000542  000207                  rts     pc

000544  105737          in:     tstb    *$icsr  / wait until next character
000546  177560
000550  0x80FD                  bpl     in      /   is sent from the keyboard
000552  113700                  movb    *$ibuf,r0 / in this character
000554  177562
000556  004767                  jsr     pc,out  / and print it
000560  177746
000562  000207                  rts     pc
```

This program and ltap use a busy loop to wait for the completion of I/O, which is signaled by bit 7 in the command and status register. This bit happens to be the sign bit of the register's low byte and is thus copied to the N flag by the tstb instruction. Sensing the I/O status in a loop is also known as "polling".

Polling is good enough as long as only one program is running, as is the case in the booting phase. But this method is not sufficient to support a multitasking system like Unix, where interrupts signal the completion of I/O commands.

Note that the program continues with other instructions after writing a character to the output buffer, rather than waiting for the character to be sent to the printer. The program only waits before writing the next character. This technique saves some time, since CPU and device do their jobs concurrently.

The program assumes that the printer can print the characters as fast as they are sent. This is true with a transmission rate of 110 bits per second (BPS) and a printing speed at 10 CPS, since 11 bits are transferred per character (1 start, 7 code, 1 parity, 2 stop). Carriage return (CR) from the right margin takes more time, and the character following CR might be printed while the carriage is still returning. This is avoided by sending a line feed (LF) character after CR. Line feed does not interfere with carriage movement. If you allways send CR followed by LF, you don't need to worry about breaking the printer. This is quite different with higher transmission rates.

The Unix "Command Line Interface" (CLI) is designed for this kind of terminals. You have to keep this in mind when talking to Unix. For example, the printer cannot erase a character or move a cursor. Instead, the "DEL" control character just moves the carriage backwards by one column. This can be exploited to print two characters at the same position. Overstriking, as it is called, is used by Unix's NROFF typesetting system to produce bold typeface or to underscrore characters. On video terminals, overstriking means replacing, and manual pages rendered by nroff will show the underlines only instead of underlined characters. Fancy pagers like "less" try to emulate overstriking in that they switch to bold type when they see a "character-backspace-same character" sequence. On a printer, lines won't scroll off, so the CLI does not need pagers to pause printing. But it is badly needed on video terminals and running ancient Unix on those terminals is a pain. Use a GUI terminal emulator, configured with a big scroll buffer, to simulate a printer.

Unix will print a character as soon as it is typed by the user, i.e., it "echoes" the input. The kernel saves the input characters until it reads a CR. Only then the characters are transferred to a user program, like the shell or the editor. This lets the kernel support command line editing: The last character is deleted from the line buffer by typing the '#' character, it is called the "erase" character. The "kill"-character, which is '@', deletes all characters in the buffer. But neither '@' nor '#' will erase any characters on the display–after all it's a printer. The backspace character is treated as a normal character by Unix to allow effects like overstriking. So, when you type the backspace key, or one of the '#' or '@'-characters, the line as seen by Unix differs from what you see on the terminal. If you

want to turn off the special meaning of characters like '#' and '@', prefix them with " ", the "escape" character.

Exercise: What will the kernel send to a user program after typing
helbig@ba-stuttgart
#include

What do you need to type to send the intended lines to the user program?

The TTY keyboard has a control key. It toggles the high bit of the character code when pressed together with another key. If you want to send the backspace character (code 010) to the computer, you press the control key together with the H key (code 0110). The control key is provided with the same meaning by contemporary terminal emulators. It is needed to send the DEL (code 0177) character, which is CNTL-? ('?' = code 077). The Unix kernel assigns this character a special meaning as the the interrupt character. It asks Unix to stop the currently running program. Nowadays, the interrupt character is CNTL-C. Finally, the CNTL-D character (code 04) will send the current line to a user program reading from the terminal. When the line is empty, that is, CNTL-D is typed at the beginning of the line, the user program gets zero bytes, which it usually treats as an "End Of File" condition.

By the way, the shift key of the ASR keyboard toggels bit 4 of the character code when pressed together with a digit-key. So "shift 1" ('1' = code 061) will send "!" (code 041). This influenced the layout of PC keyboards. The german one differs only at three keys (shift 3, shift 7 and shift 0) from this so called bit-paired layout of the ASR-33.

It takes some patience to master command line editing. But it's worth it; up to today, the Unix CLI is considered to be one of the most effective ways to interact with a computer.

1.0.5 Reusability of MUL, DIV for unsigned arithmetic

The multiplicative instructions interpret their operands as encoded integers, that is these instructions provide signed arithmetic. This section investigates the reusability of these instructions for unsigned arithmetic. Throughout this section, N=2^{16} if to be applied for the PDP-11.

MUL computes a longword p from to words m and n. Let ps, i and j be the integers encoded by p, m and n. Then, because MUL is defined to yield the integer product, you get:
(0) ps = i*j
If MUL worked for unsigned integer, like the additive instructions do, then
(1) p = m*n

must hold. Does it?
Depends on the sign of i and j:

a)	both nonnegative:	i ¿= 0, j ¿= 0
b)	one negative, one nonnegative:	i ¡ 0, j ¿= 0
c)	both negative:	i ¡ 0, j ¡ 0

Because of symmetry, the case i ¿= 0, j ¡ 0 is covered by b).

Case a) Then m = i, n = j, p = ps and (1) follows from (0). MUL passed Case a).

Case b) If j = 0, p = i*j = 0 = m*n, that is, MUL passes. If j ¿ 0, you get:

p	$= ps + N^2$	(p is encoding of ps ¡ 0)
	$= i * j + N^2$	(from (0))
	$= (m - N) * n + N^2$	(i = m-N, j=n)
	$= m * n + N^2 - n * N$	

So MUL flunks when one factor is negative and the other positive. And we are done. But curiosity lets us analyse the next case:

	p	$= ps$	(ps ¿ 0)
		$= i * j$	(from (0))
Case c)		$= (m - N) * (n - N)$	(i=m-N, j=m-N)
		$= m * n + N^2 - m * N - n * N$	

Again, MUL flunks, as you might have expected by now.

But not all is lost! In Case b) and Case c) you might fix p. Case b) gives you:

$$p + n * N - N^2 = m * n$$

How do you add $n * N - N^2$? Use ADD to add n to the highword of p. This will overflow, because $p + n * N = m * n + N^2 > N^2$. This overflow effects the subtraction of N^2 from p.

Case c) is analogously: Add n and m to the highword of p to get $m * n$.

Fixing is not needed when you are content with the low word of the result, for this is $(p \ mod \ N) = (m * n) \ mod \ N$ as can be seen from the following formulas by applying (MOD0):

Case a) $p = m * n + N * 0$
Case b) $p = m * n + N * (N - n)$
Case c) $p = m * n + N * (N - m - n)$

Resume: The MUL instruction does not provide unsigned arithmetic. But it can be used to program an unsigned multiplication. If the unsigned result fits in 16 bit, the encoding of the signed product equals the unsigned product.

The DIV instruction divides a word encoded integer, the divisor, into a longword encoded integer, the dividend. Both the quotient and the remainder are word encoded integers. The DIV instruction, as opposed to the div operator of DEF0, rounds towards zero.

```
CAST:
i: dividend, in [-N^2/2, N^2/2)     encoded by m in [0, N^2)
j: divisor,  in [-N/2, N/2)         encoded by n in (0, N)

qs: signed quotient i/j             encoded by q in [0, N)
rs: signed remainder i%j            encoded by r in [0, N)

qu: unsigned quotient m/n
ru: unsigned remainder m%n

DIV computes qs and a rs such that:
      qs * j + rs  = i        and      |rs| <= j and sign(rs) = sign(j)

DIVU, the unsigned division, computes qu and ru such that:
      qu * n + ru = m         and      0 <= ru < n
```

Note that qs, rs, qu and ru need not necessarily fit in words, that is overflow might occur.

Signed division overflows if qs is not in [-N/2, N/2). In that case DIV sets the V flag, and the result of DIV is not specified. DIV sets the C flag if the divisor equals zero.

Unsigned division overflows if qu ¿= N. This is to be indicated by the V flag. In that case, the result of DIVU is not specified, i. e., need not to be computed at all.

With these specifications under our belt, we are ready to investigate the reusability of DIV in an implementation of DIVU. Again, we need to separate cases according to the sign of the encoded integers.

Case a: m in $[0, N^{2/2}), nin[0, N/2)$
Then i = m, j = n. And q, r are equal the unsigned results, provided DIV does not overflow! If m, n are such that DIV might overflow but DIVU might not overflow, we cannot use DIV for DIVU. Let sof indicate signed overflow and usof unsigned overflow. So we have to check if there are m, n such that sof && not usof might hold.

```
sof && not usof <=>     qs >= N/2     && qu < N
                        <=> qs*j >= N/2*j  && qu*n < N*n
                        <=> i-rs >= N/2*j  && m-ru < N*n
                        <=> m-ru >= N/2*n  && m-ru < N*n
                        <=> m >= N/2*n+ru  && m < N*n+ru

sof && not usof         => m >= N/2*n      && m < N*n+n
sof && not usof        <=   m >= N/2*n+n   && m < N*n
```

The last two lines mean in English:
If $N/2 * n + n <= m$ and $m < N * n$, we can be sure that DIV will overflow and DIVU will not, i.e. we can be sure that q and r are wrong.

If $m < N/2 * n$ or $m >= N * n + n$ we can be sure that DIV will not overflow or DIVU will overflow, i.e. we can be sure that q and r are the unsigned results or need not to be computed.

Case b: $min[0, N^2/2), nin[N/2, N)$. Then: $i = m, j = n - N, qs <= 0, rs >= 0$, and q,r are defined by $(N - q)(N - n) + r = m$ and $0 <= r < N - n$.

So $q = N - m/(N - n)$ and $r = m\%(N - n)$ which is not what we want! And which doesn't look like it helps us to get what we want either.

How about sof, even though we don't have any use of DIV.

```
sof <=> qs    < -N/2
    <=> qs*j  > -N/2*j
    <=> i     > -N/2*j+rs
    <=>m      > N/2*(N-n) + rs ,          0 <= rs < N-n

sof  =>m      > N/2*(N-n)
sof <= m      > N/2*(N-n) + N-n
```

In English:
If $m <= N/2 * (N - n)$ we can be sure, that we don't have signed overflow.
If $m > N/2 * (N - n) + N - n$ we can be sure, that we have signed overflow.

Case c: $min[N^2/2, N^2)$, n in $(0, N/2)$
All values of m and n here cause an unsigned overflow, because $m >= N^2/2$ and $N*n+ru <= N*(N/2-1)+N/2-2 = N^2/2-N/2-2 < N^2/2$, which yields $m >= N*n+ru$, that is usof.

So in this case, we not only don't want DIV, but we even don't need it!

Case d: $min[N^2/2, N^2)$, n in [N/2, N).
Then $i = m - N^2$, $j = n - N, qs >= 0$, $n - N < rs <= 0$. And q, r solve $q*(n-N)+r-N = m-N^2$, $n-N < r-N <= 0$.

So $q = m/(n-N)$ and $r = N + m\%(n-N)$, again not what we want.

Resume: The results of DIV differ from the results of DIVU. The differences are too big to be fixed, as opposed to the situation of MUL and MULU.

Now how about confining to 16 bit dividends, that is clearing the upper word of the dividend before applying DIV?

Then $i = min[0, N)$. Since $m < N <= N*n$, we have no usof.
Since $m < N < N^2/2$, only two cases are left depending on n:

Case a, 16 bit: $n < N/2$.
Then q and r as delivered by DIV are correct, provided there is no sof.
$sof <=> i - rs >= N/2 * j$
$<=> i >= N/2*j + rs$ For $j = 1$ rs is zero and the right side equals $N/2$. For $j > 1$ the right side is $>= N$, but since $i < N$, there will be no signed overflow.

Result: There is signed overflow if and only if n=1 and m in [N/2, N). This can easily tested, and for the other values of m and n, DIVU does not differ from DIV.

Case b, 16 bit: $n >= N/2$. Like in the 32 bit case b), DIV cannot be used. But that's not too bad: Since $m < N$, and $n >= N/2$, qu is easily computed without DIV by:
$qu = 0 if m < n$
$qu = 1 otherwise.$

Resume: Unsigned division of a 16 bit dividend can be implemented by DIV with two exceptions:
a) divisor = 1
b) $divisor >= 2^{15}$

In Unix V7 the C language is extended to support unsigned integers. The C compiler implemented +, -, *, / and % with the corresponding signed instructions.

This is an error for / and %, since the implementation needs to take the exceptions a) and b) into account. It looks like this error of the V7 C compiler went unnoticed to date, showing that unsigned division was not used much in V7 C-programs.

Exercise: What will be 60000/40000 and 60000%40000 if V7's C is used?

1.0.6 Unsigned double precision division

The DIV instruction only supports word sized divisors, yielding word sized quotients and remainders. But sometimes you need to compute a quotient and a remainder of a longword dividend and a longword divisor. So let m in $[0, N^2)$ be the dividend, n in $[N, N^2)$ be the divisor. We want to compute p and r such that

$$m = q * n + rand 0 <= r < n$$

This is a longword equation. We aim at separately solving the high word part of the equation. To this end depart dividend and divisor into their high and low words:

$$m = mh * N + ml, n = nh * N + nl$$

Since $q <= m/n < N^2/N = N$, q fits in a word. With these names, the equation reads:

$$mh * N + ml = q * nh * N + q * nl + r$$

Now, only $q * nl + r$ contributes to both the low and the high word of the equation. We separate them by setting

$$s = (q * nl + r) divN; t = (q * nl + r) modN$$

Now, the equation reads:

$$mh * N + ml = (q * nh + s) * N + t$$

And we can solve the high word part:

$$mh = q * nh + s$$

Unsigned single precision division of nh in mh yields q and s. We use these values to compute r from

$$N * s + ml = nl * q + r$$

Note, that nl*q is unsigned multiplication of words yielding a longword.

Exercise: The above derivation is wrong. Why? Hint: Try the division with N=10, m=22 and n=12 to see that it is wrong.

1.1 Installing Unix V6

The program product was the last one assembled manually. Unix was used to develop Unix so we'll use Unix to study Unix. For this, you first need to install a bootable binary disk from the distribution tape. The paper "SETTING UP UNIX-Sixth Edition" (filed in v6/doc/start.ps) explains usage of the secondary boot program. Use it to load "tmrk". "tmrk" copies blocks from tape to disk. You can use ltap as the primary boot program, which loads the secondary boot from the distribution tape. Before booting Unix from disk, set the SR to 173030. This will boot into single user mode, which is enough right now. After all, we only have a single terminal.

SIMH: A version of the distribution tape suitable for SIMH is filed at v6/dist.tap.

SIMH: Use "boot rk0" to load the first block from disk. You don't need to key in a primary boot program to boot from disk.

SIMH: Use "set tto 7b" to set the output to seven bits per character. The SIMH setting defaults to eight bits per character and doesn't work, because the TTY driver sets bit 7 to a parity bit.

The shell prompts you with "#" to enter commands. To use unix, you need to read the "Unix Programmer's Manual", see v6/doc/index.html.

The TTY driver assumes an upper case only console. Use "stty -lcase" to teach it better. Another source of confusion might be the "change directory" command, which reads "chdir" in Unix V6 and not "cd" as in its successors.

Notation: foo(number) references the manual page "foo" in man.number.ps For example stty(I), tty(IV) and chdir(I). The manual pages are provided as html pages as well, e. g., you find the stty page at v6/doc/I/stty.html

You want to verify the installation by running two file system checks, namely icheck(VIII) and dcheck(VIII). Icheck checks that every block in a filesystem is either allocated to exactly one file or on the list of free blocks. Dcheck verifies that every file has at least one name, i.e., an entry in a directory, and that the number of names of a file equals the link count stored with the file.

Icheck and dcheck require a device name argument. Device names are directory entries much like names of regular files. They bind a name to a device, i.e., memory, terminal, tape or disk. Unlike regular file names, device names are created with mknod(VIII). It takes four arguments: the name of the device, conventionally an entry in the /dev directory; the type of the device (c for character, b for block); a major number, which identifies the device driver, and a minor number, which identifies one of the eight possibly attached disk drives.

Device files are called "special files" as opposed to "regular files".

The major number is an index to the driver's entry in either the "character device switch" (cdevsw) or the "block device switch" (bdevsw). Both tables are written into the file /usr/sys/conf/c.c by a program called "mkconf". "Mkconf" itself is distributed as a source file /usr/sys/conf/mkconf.c. For the commands needed to build mkconf and c.c, consult the shell script "/usr/sys/run". It configures and builds the distribution kernels for a variety of hardware configurations.

Besides c.c, mkconf creates the assembler program l.s, which defines the low memory part of the kernel, namely the trap and interrupt vectors, which is the yellow area. In other words, mkconf selects the drivers and creates the corresponding interrupt vectors as needed by a particular hardware configuration. When the kernel is linked, only drivers that are referred to in c.c, will be included.

You only need to specify rk and tm to mkconf. Other devices, like the console driver or the memory, are included automatically by mkconf.

Here is a typescript that creates mkconf, c.c and l.s:

```
# chdir /usr/sys/conf
# cc mkconf.c
# mv a.out mkconf
# mkconf
rk
tm
done
#
```

From the c.c file we learn the assignement of major numbers to drivers:
block devices rk=0, tm=3,

character devices: kl=0, mem=8, rk=9, tm=12

You are now ready to name the rk0 character device. Since character device in this case means the kernel does not do any "blocking", the name traditionally starts with an extra "r" for raw device.

The typescript continued:

/etc/mknod /dev/rrk0 c 9 0

And finally the filesystem checks:

```
# icheck /dev/rrk0
/dev/rrk0:
spcl            5
files         292
large          95
direc          24
indir          95
used         2902
free         1011
# dcheck /dev/rrk0
/dev/rrk0:
```

If your output looks similar to the above, you are ready for the completion of the installation:

Create block and character device names for rk1 and tm0. The names of the tape device files are "/dev/mt0" and "/dev/rmt0", not "/dev/tm0".

Copy the source disk (blocks [4100, 8100) on tape) to /dev/rrk1.

To copy blocks, use dd(I):
dd if=/dev/rmt0 of=/dev/rrk1 skip=4100 count=4000

SIMH: Remember to attach rk1 before running the dd command.

Now you are ready to mount(VII) the file system:
/etc/mount /dev/rk1 /usr/source

Note that /etc/mount wants you to specifiy a block device.

1.2 Standalone Programs

The secondary boot programs are used for two purposes:

- load programs that are needed to install Unix on a disk (e.g. tmrk)
- load the kernel

But they are specified independendly of their original purpose, namely:

read the name of a program from the terminal

locate the blocks of the program by means of a directory

copy the blocks to memory starting at address 0

jsr pc,0

Since the boot programs do just that, they can be used to load and start any program with origin = entry = 0. This blends nicely with the programs created by the assembler or the linkage editor. The boot programs don't touch the MMU and don't change the IPL. So, when loaded after power on, this means the MMU is turned of, the interrupts are blocked, and the CPU runs in kernel mode.

The boot programs use an existing directory structure: On tape, it is the one be created by tp(I), on disk, it is the root directory of the file system. This design decision saves work: First, you don't have to think out another directory structure. Second, you don't have to provide programs that maintain that directory.

There are no special purpose tools needed to build and install the boot programs themselves. Their sources are stored in /usr/source/mdec together with a shell script containing commands to build and install them in /usr/mdec. The tp(I) and mkfs(I) commands install a boot program from /usr/mdec on the first block of the tape resp. the disk.

"Install" means to put a machine program on disk or tape from which it can be loaded, i. e. copied to memory.

To make the boot programs compatible to the rest of the system, two problems need to be solved:

- The boot program and the program to be loaded both occupy block zero of the memory, that is while loading, the boot program is writing over its own code.
- The file produced by the assembler has "a.out"-format. That is, the file does not start with the machine program but with a 16 byte header, which is then followed by the machine program. If this file including the header would be copied to memory at zero, origin and entry would be 20 (octal) instead of zero.

The boot program solves the first problem by moving its own code from [0, 512) to [48K-512, 48K) before it starts loading another program. To make this work, the (short) part of the program that is used to move itself needs to be position independent, that is the code works regardless of its origin. The whole program is assembled with the origin set to 48K-512. How this is done, see section 9 of the assembler manual, where the usage of the "relocation counter" is described. Why is the boot program moving itself just below 48K? This is advertised as the minimum storage requirement for running Unix. So, standalone programs including the kernel must not exceed 48K-512. This sounds bad, but kernels in those days tend to be small. For example, size(I) shows that code and data of a kernel including the tm and rk drivers occupies 24842 byte.

The second problem is solved by the boot program: It first copies the standalone program to memory and checks if it starts with an a.out(V) header. If so, it moves the program down 16 bytes before jumping to 0. In other words, it skips the a.out header. The first word of the header is 407 (octal), which is used for the check. 407 happens to be the "br .+20" instruction, i. e. it jumps over the header. So the primary boot program does not need to worry about the header of the secondary boot program. But when the secondary boot program moves itself to upper memory, it skips its own program header, if it finds it. This is another reason for initial part of the boot program to be position independend: It might be running with origin 0 or origin 16, depending on the presence of the a.out header.

The boot system shows the traits of a typical Unix design:

There is little information transferred between boot - and booted program, i.e. the boot program "does not know" much about the kernel, and thus works for any standalone program.

The kernel "does not know" the media and boot program by which it was loaded and thus can be loaded from any tape or disk model.

This "thin" interface helps to build flexible systems, saves you lines of code and hours of work as opposed to special purpose tools, i.e. a boot program with a fat interface to the kernel that can only be used to boot the kernel.

Furthermore it is typical of well designed systems in general and Unix in particular to avoid special purpose tools. The same tools can be employed to build and install three quite different types of programs, namely

- Programs to be run under the control of the kernel
- standalone programs like the kernel or the install programs
- (secondary) boot programs

Exercise: Create an assembler source of the sum1 program in Unix. Assemble and install it on disk and on (a new) tape. Load and start it from both media. What needs to be changed in sum1 to make it work as a standalone program?

Exercise: Assume the symbol l is defined as a label and the symbol n as an absolute number. Labels are values relative to the origin whereas absolute numbers are –well– absolute. Which of the following statements are affected by setting the relocation counter to 1000?

```
mov   $l,r0
mov   $n,r0
mov   l,r0
mov   n,r0
```

1.3 Maintaining userland software.

> Es hat nicht soviel Tag im Jahr, wie der Fuchs am Schwanz hat Haar.
>
> Dieter Krebs, Sketchup

This chapter shows how to maintain C written commands and the C library by fixing four time/date related bugs that cause trouble in modern times.

Time and date in Unix.

The date(I) command is used to display and set the current time, which is kept by the kernel in a longword as the number of seconds since 00:00, Jan 1, 1970, Greenwich Mean Time (GMT). This number is incremented by the clock interrupt service routine ones every second. The time(II) and stime(II) system calls read and set the time using kernel representation. File modification and access times are stored in the kernel representation as well. Since this representation is independend of the local time, you don't have to adjust the time at two o' clock on sunday morning twice a year to account for daylight saving time. And you don't have to adjust timestamps when exchanging files between Unix sytems in different timezones.

The kernel representation encodes both date and time, that is you get both with one system call. This contrasts with other operating systems that provide separate calls for date and time, confusing programs that read the date just before midnight and the time just after midnight.

Local time comes into play only when the date(I) command sets the time provided on the commandline or when programs like ls() display a timestamp. Displaying time and thus converting from internal representation to local time is encapsulated by the function ctime(III) in the C library. The date(I) command is the only place that converts

from local time to kernel representation.

The date command can't handle dates in the 21st century, that is it suffers from an ordinary year 2000 problem. This is easily fixed. Ctime() assumes the 20th century when printing a date, another simple Y2K problem. Furthermore, Ctime() uses a division that overflows when applied to dates younger than some day in 1999. This is somewhat harder to fix. The third bug we'll fix occurs in the find(I) command, when comparing timestamps in files with the current date.

1.3.0 Making date() Y2K ready

You can find the source of the date command by typing
chdir /usr/source
find . -name "date.[cs]" -a -print
It turns out that there is a file s1/date.c. We are lucky since we don't have to fix an assembler program, whose source files end in ".s".

While setting the year, date(I) reads two digits from the command line assuming the 20th century.

Exercise: Fix this, i.e., assume the 21st century if the year entered is less than fifty and the 20th century otherwise. Because date uses the buggy ctime(), it doesn't make sense yet to build and install the fix.

1.3.1 Fixing ctime

The functions in ctime.c handle three different encodings of time:

kernel-rep:	seconds in a longword, as used by the kernel
array-rep:	seconds, minutes, hours, day of month, month, year in an array of integer. The entries of the array are specified in the manual page ctime(III).
ASCII-rep:	ASCII string like "Sun Feb 2 15:57:42 2003"

Ctime() converts from kernel-rep to the ASCII-rep in two steps: It calls localtime() to convert from kernel-rep to array-rep and asctime() to convert from array-rep to ASCII-rep.

Exercise: Like date(), asctime() is not Y2K ready. It prints "19" instead of "20" even so the year is in the 21st century. This is the second bug. Fix it.

Localtime() subtracts the timezone offset from the kernel-rep to get a kernel-rep of local standard time. Then it calls gmtime() to convert from kernel-rep to array-rep. Localtime() finally uses the array-rep to decide whether daylight saving time is in effect. If so, it adds one hour to the kernel-rep and calls gmtime() again to get the array-rep of local time.

Exercise: The time zone is initialized to represent EST (Eastern Standard Time). Change it and its name (in tzname) to represent your local time zone. Use "CET" and "CES" for the Central European Time and Central European Summer Time.

Exercise: Eastern time switches to daylight saving time at last sunday in April and back to standard time at last sunday in October. Adapt this to the rules of your time zone.

Gmtime() is the only place in Unix, that converts kernel-rep to array-rep. And this is where the third bug lurks. The conversion is started by splitting the seconds in number of days before the current days and number of seconds in the current day, which is easily done by dividing the number of seconds by the number of seconds per day (spd). The

quotient is the number of days before the current day and the remainder is the number of seconds in the current day.

But spd is not an encodable integer, so gmtime() uses a trick: It divides by seconds per eight hours instead. This works fine, as long as the quotient is an encodable integer, that is, less than 2^{15}.

Exercise: Write a C-program that prints the last date before this quotient overflows. Use hmul(III) to compute the kernel-rep of that date.

To fix the bug, let us code the function ulldiv(), that divides a long unsigned dividend by a long unsigned divisor, computing a long unsigned remainder and an unsigned quotient. Since in our case the divisor is greater than 2^{16}, the qotient is less than 2^{16}, so a word suffices to hold the quotient.

The engineering of ulldiv() is separated in three steps: Step one: Derive an algorithm for division of nonnegative integers that only uses additive and compare operations assuming unlimited size of integers.

Step two: Choose a representation of long integers in terms of the types provided by C.

Step three: Code the additive and compare operations needed in ulldiv for the new data type.

Step one: With m ¿= 0, n ¿ 0 we are to compute q and r such that the post-conditions DIV0 and DIV1 hold:

DIV0 $m = q * n + r$
DIV1 $0 <= r < n$

One technique of deriving a program is to weeken the post-condition such that they can easily be initialized and then to strengthen them in a loop. In this case, DIV1 is a candidate to be weekened to DIV1a: $0 <= r <= m$. In the loop we then need to decrement r until DIV1 holds, while maintaining DIV0 and DIV1a.

DIV0 and DIV1a hold with this initialization:

$$q = 0; r = m,$$

We arrive at the algorithm:

```
ullidv(m, n)
int m, n;
{
        int q;
        int r;

        q = 0;
        r = m; /* DIV0 and DIV1a hold and is maintained by the loop */
        while (r >= n) {
                q++;
                r =- n;
        }
        /* DIV0 and DIV1a and r < n hold, which implies the post conditions */
}
```

Now, the above algorithm is correct but takes a long time if the quotient is large. Thesis can be accelarated by doubling the amount to be subtracted each time. We introduce two variables nn and qq and replace the above loop by:

```
        while (r >= n) {
                qq = 1;
                nn = n;
                while (r >= nn) {
                        q =+ qq;
                        r =- nn;
                        qq =+ qq;
                        nn =+ nn;
                }
        }
```

In addition to DIV0 and DIV1a, the inner loop maintaines

DIV2: $nn = n * qq$ and $qq > 0$

Exercise: Prove, that
a) DIV0, DIV1a and DIV2 hold before the inner loop.
b) DIV0, DIV1a and DIV2 is maintained by the inner loop, that is if these conditions hold before an iteration, they hold after an interation.
c) Both the inner and the outer loop terminate.

This finishes step 1, which was highly influenced by a similar discussion from Edsgar W. Dijkstra in "A Discipline of Programming", 1976, Prentice Hall, pp. 57-58.

Step 2: The primitive types in C are the ones the PDP-11 offers, namely integers, pointers and bit arrays. The compare operations provided by the integer type are useless for unsigned integers, so we build our long integer from pointers. In C, when you add an integer to a pointer, the integer is multiplied by the size of the object pointed to. This is not what we want, so we use "pointer to character" as our primitive type. The MUL and DIV instructions use long operands by storing the most significant word in the lower numbered register. We adopt this convention and store a long integer in an array of two words, starting with the high word. Time's kernel-rep also starts with the most significant word.

So, this is how variables m, n that represent long unsigned integers are to be defined in C:

$$char * m[2], *n[2];$$

The definition of ulldiv() reads:

```
/* Unsigned division of long dividend m and long divisor n.
 * Returns the low word of the quotient m/n.
 * On return, the divisor will be set to the remainder m\%n of the division.
 */

char *
ulldiv(m, n)
char *m[], *n[];
{
        ...
}
```

The return type is unsigned integer, indicated by "char *".

Step 3: The operations we need to implement for the new type unsigned long integer are:

- lgeq(a, b), which returns 1 if $a >= b$ and 0 otherwise.

- ldec(a, b), which subtracts b from a, assuming $b <= a$.

- ldouble(a), which doubles a.

In C, when an argument is an array, the address of its first entry is passed to the function. This address is then used to change the argument. On the opposite, when an integer or character is passed, a copy of the variable is accessed in the function's body, and changes to the parameter won't affect its value outside the function. This "feature" is used by the definition of ulldiv, which changes the parameter n and by ldec(a, b) and ldouble(a) which change a.

The implementation of ldec() and lgeq() is straight forward, but ldouble(a) will overflow if $a >= 2^{31}$. In this case the inner loop of the above program is to be terminated, which is still correct since $nn >= 2^{31} implies 2*nn >= 2^{32} > r$.

Here is an implementation of ldec(a, b):

```
while (r >= n) {
        qq = 1;
        nn = n;
        while (r >= nn) {
                q =+ qq;
                r =- nn;
                qq =+ qq;
                nn =+ nn;
        }
}
```

Exercise: Code ldouble(), lgeq() and ulldiv().

Answer: In the file s4/ulldiv.c

Exercise: Fix gmtime() by using ulldiv().

1.3.2 Installing the fixes

We first compile the fixed library sources, that is s4/ctime.c and the new s4/ulldiv.c. Inspect the run shell script in s4 to see how to compile ctime.c. The compilation leaves object file ctime.o and ulldiv.o in s4. Objects are machine programs in a.out format. These objects are to be installed in the archive /lib/libc.a by means of the ar(I) command. Note, that the run script does not put the newly created objects into the archive. Use instead

ar rv /lib/libc.a ctime.o ulldiv.o

The run script only recreates a new archive from the files in the old one.
Now, we have to relink any command sources, that use ctime, localtime and gmtime.

With find(I) and grep(I) you get a list of the affected programs.
Sources that contain "localtime":
s1/cron.c
s1/date.c
s2/sa.c
s2/tp3.s

Sources that contain "ctime":
s1/date.c
s1/dump.c

s1/ln.c (does not call ctime, contains the string ctime only)
s1/ls.c
s2/mail.c
s2/pr.c
s2/prof.c
s2/ps.c
s2/restor.c
s2/sa.c
s2/who.c
s7/nroff1.s
fort/rt2/ctime.s (does not call ctime)

Neither "gmtime" nor "asctime" occure in any source file besides ctime.c.

Again, inspect run in the source subdirectories to find the command line for building and installing the program: For example you rebuild date by:

```
cc -s -O date.c
cp a.out /bin/date
```

The cc command controls the building of C programs. First it invokes the two phases and the optimizer (caused by the -O flag) of the C compiler which translate from C to assembler language. cc then calls the assembler to produce a machine program. Finally, cc invokes the link editor ld, passing among others the "-s" flag, which means that ld should discard the symbol table and relocation table. ld searches the archive /lib/libc.a for objects that define unresolved symbols and includes them in the resulting a.out.

Rebuilding and reinstalling all of the above commands finishes these fixes.

1.3.3 Fixing find's trouble with old timestamps in files

Unix V6 records two timestamps for each file. One timestamp tells when the file was modified the last time and the other tells when the file's content was accessed the last time. The kernel representations of these timestamps are stored in the inode, which a user program like ls reads with the stat(II) system call. The "-l" flag tells ls(I) to print the modification time while the -lu flag prints the usage time. The modification time is updated when the content or the inode is changed. The usage time is updated only when the content is read, not when the inode is read. In the V7 filesystem, a third timestamp reflects modification of the inode and the modification time is set only when the content is modified.

Exercise: What would ls -lu show, if the file's access time would be updated each time the files content or its inode is read?

The -mtime and -atime flags of find(I) select files on account of their timestamps. Let t be the timestamp, n the current time, spd the number of seconds per day and d a number. Then the condition checked is specified by the following table.

commandline flag	file qualifies, if
-mtime d	(n-t)/spd = d
-mtime -d	(n-t)/spd ¡ d
-mtime +d	(n-t)/spd ¿ d

Exercise: Checking the condition would be easier when you'll have to code a multiplication instead of a division: -mtime d (n-f) = d*spd What's wrong with that?

Very similar to gmtime(), find() has to implement somehow a division of a longword by a longword.

But find() uses a different technique: With the longword s = n-t it approximates the quotient by s[0]*3/4. That is, instead of dividing by spd = 86400, find computes $(s/2^{16}) * 3/4 = s/(87381 + 1/3)$. Note, that the last equation assumes division of real numbers instead of integers.

This trick suffers from two problems with really old timestamps, as they occure nowadays with files that were not touched since 1975:

- the error done by the approximation gets larger
- worse yet, s[0]*3 will overflow

Exercise: What is the smallest s such that s[0]*3/4 != s/spd?

Exercise: Compute s such that $s/spd - s/(87381 + 1/3) = 1$, assuming real numbers. 7693356.5217393888

Exercise: Write a program that prints the smallest s such that the error introduced by the approximation grows to two.

Exercise: The oldest timestamps in the V6 distribution are from May 14, 1975. Give an approximation of the current year when find's technique starts overflowing when being applied to these old files.

The fix is made easy by ulldiv(): ndays() returns the number of days since the timestamp t. The external array now holds the current time.

```
int
ndays(t)
char *t[2];
{
        char *spd[2]; /* seconds per day */
        char *dt[2];  /* delta t */

        spd[0] = 1; spd[1] = 20864;
        dt[0] = now[0]; dt[1] = now[1];

        ldec(dt, t);
        return ulldiv(dt, spd);
}
```

Exercise: Repair find() so it checks correctly old timestamps and install your fix.

Both the gmtime() and find() problems were fixed when C supplied the type "long int". Anyway, no matter how large the primitive types offered by a language are, there are always situations you need larger ones.

Chapter 2

Interrupts and Multiprocessing

2.0.0 Why interrupts?

The I/O code introduced so far repeatedly check the done-bit of the CS-register until it indicates "last I/O command completed". "Polling", as this technique is called, is simple and easily mastered by the programmer. You can achive concurrent execution of the I/O command (done by the device) and some useful computation (done by the CPU), but often the CPU spends a lot of time waiting for I/O completion. This is OK for simple operating systems that support only one CPU-process at a time, like the PC-BIOS, PC-DOS, early versions of Apple's Macintosh Operating System or Microsoft's Windows. But the programmers and designers of bigger computers felt a need to better exploit the CPU by loading multiple programs at once, so the CPU can switch to another program, when one program needs to wait for I/O completion.

This is what hardware interrupts were designed for.

Interrupting hardware introduces a new challenge to the programmer: The CPU jumps out of the program to the interrupt service routine (ISR).

Even worse, this may happen any time during the execution of the program.

2.0.1 Rules observed by the interrupting hardware

If interrupts were to occure any time any place, the design of programs that employ interrupts would be impossible. So there are some rules bounding the undeterministic nature of interrupts.

When the interrupt causes the CPU to jump out of the current process to the ISR, the PC and the Processor Status Word (PSW) are changed to the values from the interrupt vector entry. To continue the interrupted process, both of these values need to be restored, this is done by the RTI instruction.

Exercise: When the ISR consists of an RTI instruction only, the state of the interrupted process is not changed by the interrupt – with one exception– namely?

The machine instructions are "atomic" with respect to interrupts, that is, an instruction is completed by the CPU before transfering to the ISR.

Exercise: Which values are affected by the instruction "mov (r0)+,(r1)+"?

A device will issue an interrupt request, only if the "interrupt enable" bit in its CSR is set. After reset, this bit is unset.

An interrupt request is honored by the CPU only if the IPL is less than the BR. Since the greatest possible BR is 7, no interrupts will be served if the IPL equals 7. After reset, the IPL is set to 7.

Exercise: The entry of sum1 is carefully chosen at 0400 so its code would be out of the interrupt vector. For two reasons this is unnecessary. Namely?

2.0.2 Process control in Unix.

A process is a program in the middle of execution. The same process can run in one of two operation modes, the kernel mode and the user mode. The memory addressed in kernel mode is disjunct from the memory addressed in user mode. This protects kernel code, data and device registers from processes running in user mode. A process in user mode switches to kernel mode when executing a "SYS" instruction. It switches back from kernel mode to user mode when issuing the corresponding "RTI"-instruction.

Exercise: After reset, the PDP-11 runs in kernel mode. So, when the first process switches to user mode, the RTI instruction was not preceeded by a SYS instruction. What needs to be done to make the RTI switch to user mode?

Exercise: The PDP11 makes it impossible to switch from user mode to kernel mode by means of an RTI instruction. Why?

When in kernel mode a process may initiate an I/O instruction. Instead of waiting for completion, it marks itself in a global process table as waiting for this event and then transfers control to some other process which is runnable. This is called a "context switch".

The ISR called on completion of the I/O searches the process table for processes waiting for this event and marks them as runnable. When the interrupted process is in kernel mode, the ISR continues this process by means of an RTI instruction. When the interrupted process was in user mode, the ISR consults the process table for another process that is runnable and performs a switch to that process. Note that the context might be switched by an ISR only if the current process is in user mode, never if it is in kernel mode. This rule solves the synchronization problem with respect to data structures shared by kernel mode processes since an asynchronous process switch won't occur.

The process table is an example of a shared data structure that is modified from processes (in kernel mode) and ISRs. To make this modification atomic, a process protects itself from being interrupted by setting the IPL accordingly. The ISR itself usually runs at an IPL set to the BR of the device, thus protecting itself from being interrupted by the device it is serving. By prohibiting interrupts it is guaranteed that processes and ISR see the protected data structures in a consistent state, that is a state satisfying structural invariants.

It is very hard to decide if a change of a data structure needs to be atomic, i. e. protected by raising the IPL. To decide, the programmer needs to know every detail of every code possible to be executed by an ISR interrupting a process in kernel mode. If you protect too much, chances increase, that you will lose interrupts. If you protect too little, you might break data structures.

This synchronization problem is lessened by a rule governing the kernel design: Minimize access to shared state if the ISR interrupted a process in kernel mode.

Compare this to the rule imposed on the Java Swing library: After initialization, access to any Swing class is confined exclusively to one thread, the Event Dispatcher Thread. This rule holds with the exception of four methods that may be called from other threads to enable interthread communication.

When the ISR returns to a kernel process (i. e. a process running in kernel mode), it is its duty to restore the state of the process. From the stack of the interrupted process, the ISR restores the R1, R0, PSW and PC, the latter two by the RTI instruction. The general registers R2 to R5 are not touched by the part of the ISR written in assembler. Most of the ISR is written in C which makes heavy use of these registers. But the subroutine calling and returning sequence emitted by the C compiler guarantees, that the caller's registers R2 to R5 are restored on return from a C routine. Since the ISR obeys the C calling sequence, it does not explicitly restore these registers. The SP is restored implicitly since each push is balanced by a pull.

Things are slightly more complicated when the ISR interrupts a user process and needs to do a context switch before returning. This is necessary to protect the whole system from looping user processes holding the CPU for ever. In Unix, a process has two stacks, one active in kernel mode and one active in user mode. This is dictated by the fact that the addressable memory, which includes the stack area, is disjoint in user and kernel mode. The PDP 11 supports two stacks by suppling two stack pointers, the kernel stack pointer (KSP) and the user stack pointer (USP). To prepare for a context switch, the ISR pushes the USP onto the kernel stack. Part of the context switched at the end of the ISR is the kernel stack, where the state of the user process including the USP is saved. When the ISR restores the state from this other stack, it effectively returns to a process different from the interrupted one.

Exercise: During a context switch, the ISR accesses a lot of global data which are modified by kernel processes without raising the IPL. Why does this not crash the system?

2.1 Memory Management Unit and Multiprocessing

The MMU lets the software define the mapping of virtual addresses to physical addresses, a.k.a. bus addresses.

2.1.0 What is the MMU good for?

The reasons to use an MMU include:

- Access more memory:
  Without an MMU, the PDP-11 CPU can access 2^{16} bytes, which is the number of virtual addresses. With an MMU, the PDP-11 can access 2^{18} bytes, which is the number of bus addresses. Because of two extra address bits, DEC gave an MMU to the PDP-11 – three years after its birth. The alternative, which is to widen the virtual addresses, turns out to be much more expensive: it affects the width of the general registers and nearly every machine instruction. It took DEC another three years to introduce a PDP-11 with 32 bit virtual addresses. The changes were drastic enough to justify a new family name for 32-bit PDP-11, namely VAX (Virtual Address eXtension). On all modern architecures, the number of virtual addresses exceeds the size of installed memory, so the original incentive using an MMU disappeared.

- Relocation: If you want to load more than one program they have to be assigned to non overlapping memory ranges, leading to different origins. Addresses relative to the origin thus need to be adjusted. The MMU is exploited to avoid this by mapping identical virtual address ranges to different bus addresses. This is also called hardware relocation. Compare this to secondary boot programs, where the relocation is done by the assembler.

- Protection: A program can only access memory that it can address. To protect memory (or device registers) from a process, don't let the MMU map to it.

2.1.1 The MMU of the PDP-11

The virtual address range is devided into eight 8K pages. The mapping of a page is controlled by its page address register and its page description register (PAR0 - PAR7 and PDR0 - PDR7). The page address register contains a click number, a click being a range of 64 bus addresses. The 64 byte click can be viewed as the unit of memory mapping and allocation.

The PDR contains the following subfields:
size number of clicks in lower part less one, range: [0, 128). the lower part is [0, size], the upper part is [size, 128). upart A boolean, meaning upper (if true) or lower part of page is mapped. racc A boolean, meaning read access allowed. wacc A boolean, meaning write access allowed.

With MMU turned off a hardwired mapping is in effect, which maps the first seven pages to the identical bus addresses, and the last, the I/O page, is mapped to the last page of bus addresses. All addresses grant read/write permissions.

Exercise: Describe the contents of the paging registers such that the MMU emulates the hardwired address mapping.

Just as there are two stack pointers, there are two sets of paging register, one active in kernel mode, one active in user mode.

There are two instructions that let you access words as if in previous mode(PM): move to previous space (mtpi) move from previous space (mfpi). These instructions take one operand specification. They pull respective push a word using the current mode stack, and write respective read the operand. If the operand is a memory word, its address is translated using the paging registers from the PM, if the operand is the stack pointer, the one active in previous mode is accessed.

Exercise: When executed in user mode, the RTI instruction won't set the PM field to kernel mode when restoring the PSW. Why is this important for Unix?

After power on, all 32 paging registers are set to zero.

The MMU executes a "segmentation fault trap" if memory is accessed through virtual addresses that are not mapped or don't have the appropiate read/write permissions.

2.1.2 The storage segments of a machine program.

A machine program as stored in an a.out format consists of three segments:

- A text segment, containing the program code.

- A data segment, containing explicitly initialized data.

- A bss segment, representing implicitly initialized data.

The data represented in the bss segment will all be initialized to zero when the program is loaded. Since its dull to store zeroes, the bss segment is not written to the a.out file.

The C language puts string constants, global and local static variables into these segments. The location of these data is fixed for the lifetime of the program. On the contrary, storage for local nonstatic variables is allocated dynamically on the stack. Storage to these variables is allocated when the subroutine is called and freed when the subroutine returns.

The data and bss segment are seperated only in the a.out structure. When the program is loaded, there is only one combined data segment.

The a.out format specifies two types of programs, the type "executable" which is the format of the boot and standalone programs and the type "pure executable". Text and data segment of an ordinary executable is layed out contiguously in terms of its virtual addresses, whereas the data segment of a pure executable continues at the next page boundary after the text segment. The pure layout lets you control the mapping of the data segment independently from the mapping of the text segment.

Exercise: Three pure executables have text segments with the sizes
a) 16K-2
b) 16K
c) 16K+2
Where do the data segments start?

The machine programs of both types of executables are built relative to origin zero.

The a.out header contains the sizes of each segment. They can be printed by the size(I) command. For a small V6 kernel it prints

23460+1382+15438=40280 (116530)

These decimal numbers are the sizes in bytes of the text, data and bss segments and its sum. For the octal addict the sum is given in octal notation as well.

The file(I) of V6 command prints "executable" and "pure executable" to indicate the type of the file.

2.1.3 Kernel mode address mapping

The kernel, like all standalone programs, is an ordinary executable.

Before the kernel turns on the MMU, it needs to set up the MMU kernel mode paging registers.

Exercise: Guess what happens if the MMU is turned on but the pageing registers were left as they are after power on?

The mapping of all but page six is set up to emulate the hardwired mapping. From page six, only the lower 1K addresses are mapped to memory. This is allocated to the "user block", which contains per kernel process data, namely the "user" structure (290 Byte) and the kernel stack. Despite its name, the user block is addressable only in kernel mode.

Exercise: What do you think is the initial value of the kernel stack pointer?

A user block is allocated to each process. During a context switch, PAR6 is updated so it points to the user block of the next process. The other kernel mode paging registers are not modified after initialization. Text and data segment are shared among the kernel processes.

The part of the user state that needs to be restored on return to user mode is saved on the kernel stack, that is, in the user block. Furthermore, copies of the user mode paging registers are kept in the user block. In course of a context switch, these registers are reloaded. This way, the contents of the user block control which of the user processes will be continued by the RTI instruction.

2.1.4 User mode address mapping

As opposed to kernel mode, storage is not shared in user mode. This keeps the hard stuff related to shared memory confined to kernel code.

Starting with page 0, the addresses are mapped to include just enough clicks for text and data. For ordinary executables, the PDRs are set to map the lower part with read/write permission. For pure executables, the text segment is mapped with read only permission, the data segment with read/write permission. Since a text segment is not modifyable, it can be shared among processes executing the same program, without introducing shared memory complications to user land.
In both types the upper part of page 7 is mapped to memory allocated to the user mode stack. Initially, 20 memory clicks are allocated to the user stack. Naturally, read/write permission is turned on for the stack.

Exercise: Determine the initial value of the user stack pointer.

Since addresses just below the stack are not mapped, a stack overflow will effect a segmentation fault. The trap routine then tries to allocate more memory to the user stack; reprograms the user paging registers to account for the larger stack, undoes any modifications to the registers that were side effects of the trapped instruction and returns to the user process, with the PC pointing to the offending instruction, thus reexecuting it with a greater stack. The MMU supports this task by leaving the PC of the trap causing instruction in a special MMU register.

Exercise: Determine the initial value of the size field in PDR7.

To allow for dynamic storage allocation, Unix provides the brk(II) system call. It moves the break between mapped and unmapped addresses effectively changing the size of the data segment.

2.2 Dynamic Memory Allocation

2.2.0 Memory allocation

The low clicks of memory are allocated to the text and data segment of the kernel – as dictated by the hardwired address mapping. The remaining memory is allocated to processes as one contiguous range of clicks per process, starting with the user block and followed by memory allocated to user text, data, and stack. A block of contiguous clicks is called a "lump" (in this script only). The number of its first click is its "address".

Address and size of the lump allocated to a process is kept in its entry in the process table. (See /usr/sys/proc.h; p_addr and p_size, both as clicks)

Exercise: The memory allocated to a process starts at click 02000, the size of its user text is 010030 byte, the size of its user data 0500 bytes. The initial user stack size is 20 clicks and the size of the user block is 1K byte. The type of the user program is executable. How many clicks will be allocated for this process initially? Describe the contents of the user page registers and kernel PAR6.

Answer: The number of clicks allocated to the process are:

user-block:	1K byte = 16 clicks =	020 clicks
text/data:	010030 + 0500 = 010530 byte =	0106 clicks (rounded up!)
stack:	20 clicks =	024 clicks
sum:		0152 clicks

Contents of paging register:

space	page	PAR	size	PDR.size	PDR.upart	PDR.wacc	PDR.racc
kernel	6	02000	020	017	false	true	true
user	0	02020	0106	0105	false	true	true
user	7	02126	024	0154	true	true	true

All other user paging register in user paging register are invalid.

Lumps that are currently not used are represented by an array, the "map". Each two-word-entry of a map holds address and size of a free lump. (See struct map in /usr/sys/ken/malloc.c). The list of used entries in a map is terminated by an entry with size = 0. The map is implicitly initialized, in particular, its first entry is zero, thus marking the end of an empty list. This reminds of character strings in C, where the empty string is represented by 0.

The map is accessed exclusively by
mfree(map, address, size)
and
address malloc(map, size).

Mfree() enters a lump of free memory into the map. If the preceeding or following memory areas turn out to be free, mfree() combines the corresponding neighbouring lumps with the new lump.

Malloc() searches the map for a free lump with at least the size requested and returns its address after adjusting size and address of the remaining free lump. If it turns out to be empty, malloc() removes its entry from the map. Malloc() returns zero, if there is no free lump that is big enough.

The file /usr/sys/ken/malloc.c hides the implementation of dynamic memory allocation from the rest of the kernel. It keeps the definition of a map entry (struct map) private. Including comments the source is only 87 lines. It is the clients responsibility to provide empty maps that are large enough. Mfree() and malloc() work independently of the units of memory. The only condition is that the unit of addresses must match the unit of sizes.

2.2.1 Swapping

If the size of all lumps exceeds available memory, the image of a process is written onto disk and the memory is mfree'd to be allocated to other processes. This is called "swapping". To continue a swapped out process, its image

needs to be reloaded.

Shared text segments do not need to be swapped out with the image of every process using it, instead it suffices to be swapped ones for all processes using it. This saves considerable swapping I/O and was the incentive to introduce the pure executable format. Candidates for this format are programs that tend to be executed concurrently by more than one process. In V6 a

chdir /bin
$\#file * |greppure$

yields

```
as:   pure executable
bas:  pure executable
cc:   pure executable
ed:   pure executable
ld:   pure executable
ls:   pure executable
sh:   pure executable
#
```

Swapping is done by one dedicated process, the "swapper", which is the first process to be created during initialization of the kernel. The swapper is a special process in that it will never run in user mode; it follows that its lump consists of the user block only. The swapper will never be swapped out, if so, this would be about the last thing, the swapper would do.

When a process switches to another process, it does not do so directly but instead switches to the swapper, which will then look in the process table for another process to be continued. If there is none, the swapper will loop waiting for an ISR to wake up a process. If the image of the awakened process is loaded, the sapper will switch to it right away. Otherwise it will try to swap in the process, possibly after having swapped out other processes to free the memory needed for the new process.

Later versions of Unix, notably BSD Unix, introduced a more sophisticated memory management scheme called "paging". Then a process can run even with only part of its image loaded, whereas with swapping, a process can run only if all of its image is loaded.

Exercise: In V6, two resources limit the size of a program, i. e. a program cannot be executed if its size exceeds one of these limits. What are these resources?

The amount of installed memory is not to be configured by the user; instead the kernel determines it automatically. In a loop it probes clicks with increasing click numbers until a trap at 4 is executed, which means in this case that the memory does not exist. A click is probed by putting its number in the user PAR0 and executing

```
mfpi  0
```

Every successfully probed click is entered into the map of free lumps by
mfree(map, click number, 1).

The kernel prints the amount of memory that is left after allocating its own segments and the user block of the swapper.

Exercise: My kernel prints "mem = 1040" in units of 1/10th K words. The size of this kernel is 39934 as printed by size(I). How much memory does the kernel believe is installed? How much memory does SIMH provide? Explain

the difference.

Answer: Taking into account that memory is allocated at click boundaries, you calculate:
Clicks installed by SIMH: 256 * 1024 / 64 = 4096.
Clicks used by the kernel: (39934 + 1024 + 63) / 64 = 640.
Free clicks: (104 * 2048) / 64 = 3328.
Difference: 4096 - 640 - 3328 = 128.

128 clicks, which are 8K bytes, are not accessable, since the last page of bus addresses is mapped to I/O devices, leaving only 248K of bus addresses to be usable to access memory.

The disk area to be used for swapping needs to be configured in /usr/sys/conf/c.c. Mkconf takes the first block device in its parts list to be the swap device and allocates the blocks in the range [4000, 4872) to swap space.

The free blocks in the swap space are represented by an array, the swapmap, that is accessed by the same functions that are used for maintaining free memory, namely mfree() and malloc(). The addresses in the swapmap mean block numbers instead of click numbers and the unit of the size is 512 instead of 64. The map representing lumps of free memory is called the coremap.

Exercise: In c.c a comment warns that the swap area must not begin with disk block number zero. Why?

Exercise: Code a call of mfree(), such that the swapmap will represent one free lump with size nswap and address swplo.

These two lines from c.c show how variables are explicitly initialized in C, i. e. there must be no "=" sign.
```
int   swplo   4000;
int   nswap   872;
```

A device is specified by a major and a minor number. They are word encoded as the high respective low byte. Here is the line in c.c as created by mkconf:
```
int   swapdev   (0¡¡8)—0;
```

In C the initializer must be surrounded by curly braces if it is a constant expression. Only with simple constants the braces may be omitted.

Exercise: Code an initializer for swapdev that specifies the second RK disk to hold the swap area.

2.2.2 The sizes of swapmap and coremap.

The coremap and the swapmap are defined in /usr/sys/systm.h. Since the struct map, which defines an entry of a map, is private to malloc.c, it cannot by used in systm.h. The maps are therefore defined as arrays of integer. The sizes of the process table and the map arrays are configured by C symbols, which are #define'd in /usr/sys/param.h, with NPROC=50, CMAPSIZ=100 and SMAPSIZ=100; unfortunately without a hint regarding the relation of the map sizes to the size of the process table. So a map can hold up to NPROC-1 lumps – remember, one entry is needed to terminate the list of occupied entries.

To determine the size of the maps, we need to determine an upper bound of the number of free lumps ever managed by one map. It seems easier to determine an upper bound of allocated lumps. Property(0) relates both numbers:
(0) If at least one lump is allocated, the number of free lumps
is bounded by the number of allocated lumps plus one.

First, we prove property(0), whose wording is somewhat clumpsy. To arrive at a simpler statement, just define the

(nonexisting) lump adjacent to the rightmost free lump to count as allocated. Furthermore, we define free lumps to be 'white' and allocated lumps to be 'black'.

With this convention, (0) reads:
(1) The number of white lumps is bounded by the number of black lumps.

We are now going to prove (1) by showing that property (2) is an "invariant", that is it holds initially and is maintained by malloc() and mfree().
(2) The right neighbour of every white lump is black.

Property (2) holds initially:
After initialization, there are no white lumps, so (2) is valid.

If (2) holds before malloc() is called, it holds after malloc() returns: Malloc() allocates a lump. In particular, the neighbours of all white lumps remain black.

If (2) holds before mfree() is called, it holds after mfree() returns: Since mfree() combines the new white lump with its white neighbours, the right neighbour of the resulting white lump is black.

This completes the prove of (1) and (0).

Exercise: From property (2) one is tempted to infere that the number of white lumps equals the number of black lumps. But this does not hold. Why?

Exercise: Would malloc() still maintain property(1) if it would cut the lump to be allocated from the right end of a white lump or from the middle? Assume, that the size of the white lump exceeds the requested size.

To determine A, the upper bound of concurrently allocated lumps, we start with grep(I) to find the six places in the kernel that call malloc():

function	allocates a lump for ...
newproc()	... a newly created process.
sched()	... a process that is swapped in or out.
expand()	... itself, when it needs more memory.
exit()	... swapping out the user block.
xswap()	... swapping out a process.
xalloc()	... the text segment of pure executables.

Newproc() allocates a lump for every newly created process. Since we have at most NPROC processes, including the swapper process, whose lump is never allocated from a map, we arrive at $A <= NPROC - 1$.

Expand() is called when a process needs to grow its memory. It allocates a lump with the new size, copies the old lump to the new lump and then frees the old lump. While copying there are two lumps allocated simultaneously. Since at most one process is copying its lump at a time, we conclude that $A <= NPROC$. Here we exploit the fact that processes in kernel mode are not preempted.

Xalloc() allocates a lump for each shared text segment stemming from pure executables. At most NTEXT shared text segments can be active concurrently, so $A <= NPROC + NTEXT$.

The remaining three functions never allocate more than one lump for the same process in one map, so $A <= NPROC + NTEXT$.

The maximal number of free lumps, F, satisfies
F ¡= NPROC+NTEXT+1

because of (0).

It might be possible to prove that F is slightly less than that. But this would assume more details of when and how the kernel allocates respective frees lumps. It seems better to avoid complicating stuff and accept some waste of memory.

Exercise: Show that F = NPROC+NTEXT+1, by constructing a pathological example. Answer: Set NPROC=2, NTEXT=0. Assume, that the one lump allocated for the one and only user process is shrinked at both ends. This gives you two free lumps. Then, the process calls expand(), allocating the new lump in the middle of one of the free lumps. During copying, three free lumps need to be managed by the map.

Taking into account the end marker in the array, we arrive at S, the size of a map to be:
S = F+1 = NPROC+NTEXT+2

This is quite different from the sizes as configured in V6, with S = NPROC! But even so V6 was in widespread use, the maps didn't seem to overflow.

But I feel more comfortable to run a system from which I know that the maps won't overflow. While fixing that, we can configure better values for NPROC and NTEXT as well.

NTEXT is defined in param.h as 40. That seems rather generous! In fact, it exceeds the total number of pure executables in V6: This command prints one line for each pure executable:
$find/ - execfile; | greppure$

Piping the above command to wc(I) yields 19 lines. Using the fact, that there is at most one shared segment per pure executable, NTEXT=19 would be enough. These settings should suffice and avoid overflow of the maps:

```
#define   NTEXT     19
#define   NPROG     40
#define   CMAPSIZ   122
#define   SMAPSIZ   122
```

It would be far better to configure CMAPSIZ and SMAPSIZ in terms of NTEXT and NPROC, like:

```
#define CMAPSIZ ((NTEXT + NPROG + 2)*2)
#define SMAPSIZ CMAPSIZ
```

But the V6 C-preprocessor does not rescan the replacement string for defined identifiers. This is fixed in V7 C.

While tuning the kernel parameters, one wonders what happens if any of those parameters are too small. Well, it depends:

NPROG: The kernel will check if the process table is full before creating a new process. If so, it sets the error number accordingly (EAGAIN, see INTRO(II) for the meaning of error numbers), and continues without creating the process.

NTEXT: This is the size of the array "text", defined in /usr/sys/text.h. It has one entry per active text segment. If this table runs out of free entries while loading a pure executable, the kernel will "panic", that is, print a message on the console, in this case "out of text", and stop. (see /usr/sys/ken/text.c)

CMAPSIZ, SMAPSIZ: The mfree() function does not even check for overflow when inserting another free lump. So anything might happen if CMAPSIZ or SMAPSIZ are too small. It will be very hard to pinpoint the cause of

erratic behaviour in that case.

On first sight, these different attitudes towards robustness seem randomly. But they can be viewed as the result of carefully considerating the tradeoffs of robust code vs. simple code. The design of Unix is governed by a strong appreciation of simple, clear code. If it's easy to recover from an error, the kernel will recover. Otherwise, if it's easy to detect an error condition, it will panic. If even detecting is hard, as is the case with the overflow of maps, it won't even do that. This is justified by the fact that errors in Unix are very rare – a direct consequence of the code being simple.

2.2.3 Makeing changes in param.h effective

Every source file that #include's param.h needs to be recompiled – that is done by /usr/sys/run – and the resulting *.o files need to be archived in lib1 respective lib2 – that is not done by /usr/sys/run!

To update the libraries, use
    ar r ../lib1 *.o

in /usr/sys/ken respective
    ar r ../lib2 *.o

in /usr/sys/dmr.

With new objects in the libraries, build a kernel in /usr/sys/conf. You may want to use /usr/sys/run for guidance. Install the kernel as /unix so you won't overwrite /rkunix and reboot. (remember sync!)

Exercise: The ps(I) command will show spurious processes. Why?

Answer: The file ps.c #includes param.h using NPROC to determine the size of the process table. This is the only userland program that needs to be recompiled when param.h is changed.

By the way, the only other userland program that includes param.h is the C debugger, cdb(I).

Compiling kernel sources is automated by three shell scripts that I added to the system. The scripts dmr/run and ken/run compile C files supplied as arguments on the command line and archive the objects in the libraries.

The script conf/run assembles and compiles sources from the conf directory, links them with the libraries and installs the linked kernel.

Here is ken/run:

```
: loop
if \$1x = x goto done
cc -O -c \$1
shift
goto loop
: done
ar r ../lib1 *.o
rm *.o

and here conf/run:

as m40.s
cp a.out m40.o
```

```
cc -c c.c
as l.s
ld -x a.out m40.o c.o ../lib1 ../lib2
cmp a.out /unix
cp a.out /unix
rm c.o
rm m40.o
rm a.out
```

Exercise: Give the commands to recompile all kernel sources. **Answer:**
chdir /usr/sys/ken
run *.c
chdir ../dmr
run *.c

2.2.4 Implementation of swap():

If the swapper decides to swap out a process, it takes address and size of its lump from the process table (p_addr and p_size), malloc's() swapspace and mfree's() the memory. Finally, it sets p_addr to the block number of the swapped out lump. It uses the function swap(blkno, coreaddr, count, rdflg) from /usr/sys/dmr/bio.c. The unit of coreaddr and count is click. The rdflg controls the direction of the transfer, "on" indicates from disk to memory.

Exercise: Write a C program swapout(p) that swaps out a process, with p pointing to its entry in the process table.

```
swapout(p)
struct proc *p;
{
        int daddr;

        daddr = malloc(swapmap, (p->p_size + 7)/8);
        swap(daddr, p->p_addr, p->p_size, 0);
        p->p_addr = daddr;
}
```

Exercise: Write a C program that sets the I/O register (RKCS, RKWC, RKBA, RKDA) of the RK drive to start swap I/O. Use the parameters as given to swap() and the external variable swapdev defined in c.c. Refer to pdp11/doc/devs for the specification of the RK11 device.

```
        ...
\#define RK 0177404      /* address of RKCS, control and status register */\\
        ...\\
_______
struct {
        int rkcs;
        int rkwc;
        int rkba;
        int rkda;
};

swap(blkno, coreaddr, count, rdflg)
{

        RK->rkba = coreaddr << 6;          /* lower 16 bits of bus address */
```

```
        RK->rkcs = (coreaddr >> 11) << 4; /* upper 2 bits  of bus address */
        RK->rkwc = -count * 32;           /* complement of word count */
        RK->rkda = blkno % 24;            /* head and block */
        RK->rkda =| (blkno / 24) << 5;    /* track */
        RK->rkda =| (swapdev & 7) << 13;  /* disk */
        RK->rkcs =| 1<<6;                 /* interrupt enable */
        RK->rkcs =| (rdflg ? 5 : 3);      /* read/write and go */
}
```

2.3 Dynamic Memory Allocation with the Stack

The stack is used to allocate memory. In contrast to a general allocation algorithm, e. g. malloc() and mfree(), a stack can be employed whenever the

STACK CONSTRAINT: last allocated – first freed

is acceptable. This contstraint is exploited to arrive at a very fast algorithm.

The stack consists of a memory area and a stack pointer. The stack pointer divides the area in a free and an allocated part. Let the memory at [m, n) be reserved for the stack. If the stack "grows downward", as is dictated by the PDP-11, the area at

[m, sp) is free, and at [sp, n) is allocated.

Memory is freed by incrementing the stack pointer ("pop") and allocated by decrementing the stack pointer ("push").

Exercise: Characterize

a) an empty stack

b) stack overflow

c) stack underflow

by equations involving m, n and sp.

Exercise: Give an expression that evaluates to the address of the last allocated memory word.

2.3.0 Subroutine Calls

When a subroutine is called, the return address needs to be stored somewhere. In ancient times, a fixed location per subroutine is allocated to the return address. This place is usually in the program text, e.g. in a jmp instruction, which is then executed on return. This technique has two disadvantages:

- The code is modified, ruling out read only code as in pure executables or ROMs.
- A subroutine with self modifying code is not reentrant. In particular, this means, that only one process at a time can call the routine. Furthermore, those subroutines cannot be called recursivly, even with only one process active.

Both problems are solved, when memory is allocated to the return address only when the subroutine is called. Since a subroutine only returns after all nested subroutines have finished, the stack constraint holds, and thus a stack suffices to allocate memory. This is exactly what the PDP-11 instruction set supports, as opposed to older architectures like IBM mainframes.

Unix C programs not only allocate the return address from the stack, but all items whose lifetime is limited by the activation time of a subroutine. These are

- arguments passed to functions
- return address
- the caller's register values, that need to be restored on return.
- automatic variables
- intermediate results while evaluating expressions.

In the following the conditions to be established by the caller and callee are given. The top of the stack is notated as a comma separated list of the allocated items. The size of each item is one word. This is true when ignoring double arguments, which are four words long. "ra" denotes the "return address".

When passing n arguments to a function, as in f(arg1, arg2, ..., argn), the CALL condition reads:

CALL: Top of stack: ra, arg1, arg2, ..., argn
Note that the above condition implies sp = &ra.

Exercise: Write an assembler program that establishes CALL for this invokation of printf().

```
...
printf("a: %d, b: %s: n", a, b);
...
```

Assume that the symbols "fstring", "_a", "_b" and "_printf" are of type "data address" respective "text address" with the following values:

```
name            value
fstring         address of the format string
_a              address of a
_b              address of a word containing the address of b
_printf         entry of printf
```

The underlines are prefixed by the C compiler to names defined in C sources.

Answer:

```
mov   _b,-(sp)
mov   _a,-(sp)
mov   $fstring,-(sp)
jsr   pc,_printf
```

The RETurn condition is to be established by the callee whenever it returns.

RET:	
Top of stack:	arg1, arg2, ..., argn
reg values:	r0=return value; r2, r3, r4, r5 = r2c, r3c, r4c, r5c; pc=ra

Here, r?c denotes the value of r? at the time the subroutine was called.

2.3.1 The stack frame in C programs.

The above conditions need to be met by the caller or the callee whenever at least one of them are C functions. The other one might be written in assembler.

In this section we learn how C functions manage the stack.

C functions use r0 and r1 to hold temporary values which need not to be restored by the callee, whereas registers r2, r3, and r4 are allocated to variables of storage class register.

C programs save and restore registers by calling the assembler routines csv resp. cret. A C function always starts with the instruction

$$"jsr r5, csv"$$

which establishes the entry condition of csv:

CSV-ENTRY:
top of stack: r5c, ra, arg1, arg2, ..., argn
reg value: r5 = return address from csv

CSV pushes registers and an extra word onto the stack, so on return from csv the CSV-EXIT condition holds:
CSV-EXIT:
top of stack: temp1, r2c, r3c, r4c, r5c, ra, arg1, arg2, ..., argn
reg value: r5 = r5c.

The function body stores automatic variables and temporary values on the stack. Temporary values include arguments and return addresses for nested calls. Note, that csv pushes one extra word onto the stack ready to hold one temporary word. Further temporaries are to be pushed and popped. Thus, the extra word saves code to pop one word, which leads to more compact code. This trick saves about five percent.

Exercise: Complete the calling sequence of the above printf call.

```
mov     \_b,(sp)                / use extra temporay word
mov     \_a,-(sp)
mov     \$fstring,-(sp)
jsr     pc,\_printf
cmp     (sp)+,(sp)+     / only need to pop two instead of three words.
```

C functions use r5 as a base register when addressing items on the stack.

Register r5 is not modified while the function is executed, whereas the stackpointer varies while pushing and popping temporary values. The fixed part of the stack is called a "frame", and r5 a "frame" pointer. A C-function with k word of automatic store establishes a frame on the stack with the following layout.
FRAME: temp1, autok, ..., auto1, r2c, ..., r5c, ra, arg1,..., argn
r5 = &r5c

Exercise: Code the printf call assuming a and b are the only auto variables.

```
mov     -10.(r5),(sp)   / b
mov     -8.(r5),-(sp)   / a
        ...
```

Exercise: Code the printf call assuming a and b are the only arguments of the caller:

```
mov     6.(r5),(sp)     / b
mov     4.(r5),-(sp)    / a
        ...
```

To establish the return condition, a C functions jumps to cret.
CRET-ENTRY:
top of stack: ..., r2c, r3c, r4c, r5c, ra, arg1, arg2, ..., argn
reg values: r5 = &r5c; r0 = return value

Note that the stack pointer does not occur in CRET-ENTRY, as is indicated by '...' at the top of stack. This means, that the stack pointer does not need to be adjusted to point to temp before jumping to cret.

The exit condition of cret equals the return condition: CRET-EXIT = RET.

The frame pointer is the head of a linked list of active frames. Assume C functions f0 calls f1 which calls f2 ... which calls fn. Because r5 addresses the frame pointer of the caller, the FRAME condition implies the FRAME-LINK equations. (* means –as in C– the dereference operator.)
tinputlisting2.4.21.txt

2.3.2 Long Jumps

In C you can only "goto" a local label, that is you cannot jump into another function. Older languages, e.g. Pascal, support long jumps, so there seems to be a need for this feature. With C, long jumps are provided by a pair of library routines, "setexit(III)" and "reset(III)". A precondition for calling reset() is that setexit() was called by a function that is still active. Then reset() jumps to the statement following setexit().

In the following example the precondition is broken, since a() is not active when b() calls reset().

```
a()
{
        b();
        c();
}

b()
{
        setexit();
}

c()
{
        reset();
}
```

This example is correct but rather silly, it constitutes a nonterminating loop:

```
a()
{
       setexit();
       b();
}

b()
{
       reset();
}
```

The program loops, because b() is called after setexit() returns and after reset() jumps back. If this is not what you want, you need to descriminate both cases. According to the man page, after reset() jumps, all variables have the values they had when reset() was called (not when setexit() is called). Exploiting this lets you fix the program:

```
a()
{
        int first;

        first = 1;
        setexit();
        if (first) {
                first = 0;
                b();
        }
}

b()
{
        reset();
}
```

Implementation of long jumps:

For the following discussion assume
f0 calls setexit
f1 calls f2
...
fn calls reset

Then reset has to establish f1's RET condition modified by ra being setexit's return address instead of f1's return address. It does so by patching ra in f1's frame, establish CRET-ENTRY with r5 pointing to f1's frame and then jump to cret to establish the modified RET condition. To this end, setexit saves its frame pointer and its return address in global storage sr5 respective spc. The file s5/reset.s contains both reset and setexit:

```
_setexit:
        jsr     r5,csv
        mov     r5,sr5
        mov     2(r5),spc
        jmp     cret

_reset:
        mov     sr5,r5
        mov     spc,2(r5)
        jmp     cret
```

Exercise: Suppose reset() would restore r5 and then jump to cret without modifying 2(r5). What would happen then in the example program?

Answer: Reset() would jump after b() instead of after setexit(), that is it would be a "nonlocal return".

Exercise: Code an example such that reset breaks CRET-ENTRY.

Answer: Setexit saves its FP in sr5. But to establish CRET-ENTRY, reset needs r5 to be set to f1's frame pointer! So reset only works, if setexit's frame pointer = f1's frame pointer, that is if both frames happen to be at the same location. This is certainly not the case if f1 is called with at least two arguments.

Want to fix reset()? Here it goes:
reset() sets r5 to f1's frame pointer by exploiting the FRAME-LINK equations. Starting with the frame pointer of

fn(), (which is r5), it dereferences until it gets at the frame pointer of f1(). Again from the FRAME-LINK equations you derive that r5 is the frame pointer of f1() if and only if:

*r5 = frame pointer of f0.

Reset needs to check this condition, so setexit saves the frame pointer of f0 instead of its own frame pointer.

Then, in C, the code would look like:

```
while   (*r5 != sr5)
        r5 = *r5;
```

Translated to assembler, we arrive at a better version of reset:

```
_reset:
1:
        cmp     *r5,sr5
        beq     1f
        mov     *r5,r5
        br      1b
1:
        mov     spc,2(r5)
        jmp     cret
```

There is still another flaw with this version of reset. What happens, if f0() calls both setexit() and reset()? Then, the loop will miss f0's frame and run havoc going through the frames of f0's callers. When called from f(0), reset() must not restore any registers but return right away to spc. Here is reset.s with both fixes applied.

```
.globl _setexit
.globl _reset
.globl csv, cret

_setexit:
        jsr     r5,csv
        mov     (r5),sr5
        mov     2(r5),spc
        jmp     cret

_reset:
        cmp     r5,sr5
        bne     1f      /"1f" is the first label "1:" in forward direction.
        mov     spc,(sp)
        rts     pc
1:
        cmp     *r5,sr5
        beq     1f
        mov     *r5,r5
        br      1b      /"1b" is the first label "1:" in backward direction.
1:
        mov     spc,2(r5)
        jmp     cret
.bss
sr5:    .=.+2
spc:    .=.+2
```

This version of reset() is still broken – arguments passed to f1 are never popped.

This program exhibits the error:

```
main()
{
```

```
        setexit();
        b(1, 2);
}

b(arg1, arg2)
{
        printf("address of arg1: 0%o\n", &arg1);
        reset();
}
```

It prints:

address of arg1: 0177744

address of arg1: 0177742

...

address of arg1: 022256

address of arg1: 022254

address of arg1: Memory fault – Core dumped

In Unix V7 long jumps are supported by the routines setjmp respective longjmp. They suffer from the same error.

To fix this bug, reset needs to

- make r2,r3,r4,r5 comply with f1's RET condition (as done before by cret)
- make sp and pc comply with setexit's RET condition.

To support this, setexit() additionally saves a complying stack pointer at ssp.

A correct version of setexit/reset is:

```
.globl  _setexit\\
.globl  _reset\\

_setexit:
        mov     r5,sr5          / f0's frame pointer
        mov     (sp)+,spc       / spc complies with RET condtion
        mov     sp,ssp          / sp and ssp comply with RET condtion
        mov     spc,pc

_reset:
        cmp     r5,sr5  / r5 = fn's frame pointer, sr5 = f0's frame pointer
        bne     1f
        br      2f      / fn's FP == f0's FP => reset() called directly by f0.
1:
        cmp     (r5),sr5 / r5 = f1's FP ?
        beq     1f
        mov     (r5),r5
        br      1b
1:                      / r5 = f1's  FP.
        sub     $6,r5   / restore from f1's frame to make r2, r3, r4, r5
        mov     (r5)+,r2 / comply with f1's RET condition.
        mov     (r5)+,r3
        mov     (r5)+,r4
        mov     (r5),r5
2:
        mov     ssp,sp
        mov     spc,pc  / sp and pc comply with setexit's RET condition.
```

```
.bss
sr5:    .=.+2
spc:    .=.+2
ssp:    .=.+2
```

2.3.3 Signals

Unix kernels provides "signals", which are similar to hardware interrupts or traps in that they can occur any time. A process is signalled because of

- The process triggers a hardware trap, e.g. a MMU fault.
- Another process executes the kill(II) system call.
- The tty's ISR triggers a signal when it detects that the interrupt key (^?) or the quit key

  (^\)

  is pressed.
- The tty's ISR triggers a signal when it detects, that the telefone line is hung up.

There are fifteen different signals, numbered one (the hangup signal) to 15. For the details see signal(II). On default, a process that recieves a signal will terminate. The signal(II) system call lets a program specify to ignore a signal or that a user supplied function, a "signal service routine" (SSR), will be executed when a signal occurs. On return from the SSR the process will resume at the point it was interrupted.

The following program prints something when the user presses ^?.

```
main()
{
        int f();

        signal(2, f);
        for (;;);
}

f()
{
        printf("hi!\n");
}
```

After processing the signal, it is reset to the default reaction. In the above example, a second ^? will terminate the process.

Below is a transcript showing how to send a signal to the process with the kill(I) command. The process is started in the background with its process id printed by the shell. This id is then used in the kill command to address the process.

```
% a.out&
527
% kill -2 527

% hi!
kill -2 527
%
```

The SSR is called asynchronously. To continue the process, all registers, including r0 and r1, and the condition flags need to be restored. Servicing a signal starts in kernel mode, where the process pushes the PSW and the the

return address on the user mode stack. It reads both of them from the kernel stack, where they were saved by the last ISR. Before returning to user mode, the process patches the saved PC to point to a SSR wrapper, which is a library routine to be executed in user mode. The SSR wrapper pushes r0 and r1 on the stack, calls the SSR, pops r0 and r1 and continues the process by issuing an RTT instruction,thereby restoring PSW and PC as saved in kernel mode.

In the following pci and pswi mean the values of those registers when the process was interrupted in user mode.

WRAPPER-ENTRY:
top of stack: pci,pswi

SSR-ENTRY:
top of stack: ra,r0c,r1c,pci,pswi

SSR-RET:
top of stack: r0c,r1c,pci,pswi
register: r2,r3,r4,r5=r2c,r3c,r4c,r5c; pc=ra

WRAPPER-RET:
top of stack: empty
register: r0,r1,r2,r3,r4,r5=r0c,r1c,r2c,r3c,r4c,r5c; pc=pci; psw=pswi

Exercise: The wrapper routine additionally saves and restores r2,r3, and r4. Why?

Answer: Don't know.

Since signals, like interrupts, may occur at any time, the storage below the stack pointer might be overwritten at any time. So only the storage at [sp, n) is save.

Exercise: What's wrong with this version of csv?

```
csv:
        mov r5,r0
        mov sp,r5
        mov r4,-2(sp)
        mov r3,-4(sp)
        mov r2,-6(sp)
        sub $8.,sp
        jmp (r0)
```

Two programs in Unix V6, namely cron and init, use the reset function as an SSR to catch hangup signals. The init process is the first user process. It waits for connections on terminal lines and starts login sequences. The terminals to be serviced are specified in /etc/ttys. A hangup signal causes init to reread the configuration file. This feature lets the administrator add or remove terminal lines without the need to reboot. The hangup signal became a popular technique to make server processes – a.k.a "daemons" – reconfigure themselves. Examples include httpd and inetd.

Below is a typical scheme of a daemon controllable by hangup:

```
main()
{
        setexit();
```

```
        signal(1, reset);
        init();
        for (;;) {
                job = wait_for_a_job();
                do(job);
        }
}
```

This scheme is broken with our version of reset! Reset depends on the FRAME-LINK equations which might not be valid when the signal is caught.

Exercise: Which condition from this chapter contradicts the FRAME-LINK equations?

Answer: The CSV-ENTRY condition requires "r5=return address" whereas FRAME-LINK requires "r5=caller's frame pointer".

Exercise: Point out the places in reset() that break the frame link condition.

How is this to be fixed? The original reset does not depend on FRAME-LINK. But it depends on f1's frame pointer = setexit's frame pointer, which is not guaranteed to even hold without signals. Unix V7's longjmp depends like our reset on FRAME-LINK, but it checks the frame pointer while dereferencing. If it's zero it stops searching for f1's frame and does the long jump without restoring r2, r3 and r4. Since I am not at all comfortable with this solution, the only way out seems to change the specification to something that can be implemented. The problem lies in the fact that there is no way to compute f1's frame pointer which in turn is needed to restore r2, r3 and r4, which might be allocated to register variables.

The man page reads:
all accessable data have values as of the time reset was called.
Weaken it by excluding register variables:
all accessable data except register variables have values as of the time reset was called.

This leads to code that is simpler and correct – even with signals.

But we have to check all functions that call setexit to make sure they don't depend on register being restored. Furthermore, all programs have to be rebuilt with the fixed setexit.

The table lists sources that contain setexit.

file	reset() in SSR	register	was broken
s1/cdb1.c	yes	no	yes
s1/cron.c	yes	yes	yes
s1/ed.c	yes	no	yes
s1/init.c	yes	yes	yes
s2/sh.c	no	yes	yes

The column "register" indicates whether f0 accesses register variables after calling setexit() and thus are broken by the change of the specification. The column "was broken" indicates whether the original version of reset leads to an error – either because setexit's frame pointer might differ from f1's frame pointer or because reset() might have been called by an SSR while in f0, which would overwrite the setexit's frame.

Installing init and sh is not straight forward. Even as root you cannot overwrite /etc/init or /bin/sh, because both are active pure executables. And you probably don't want to overwrite them either – if the new version wouldn't

work, you cannot reboot the system. A somewhat safer strategy is to first rename the old versions and then install the new files.

2.4 Fork() and Exec()

The system calls fork(II) and exec(II) create a new process and load a new program. Separating loading and starting a program is unique to Unix. It leads to simpler functions and offers more flexibility than the traditional approach which lumps both functions in one system call.

2.4.0 Newproc() and swtch()

You are supposed to understand what is going on in both functions. But a famous comment in the swtch() source claims
"You are not expected to understand this"

So I decided to modify both functions to make them somewhat easier to comprehend. These modified sources are discussed in this chapter.

The newproc() function is called by a process running in kernel mode to create a new process. Newproc() determines a fresh process ID, locates an empty slot in the process table, sets up an entry in the slot, and copies its image, that is, its user block, text, data and user stack segments into a newly allocated memory area.

As a result, the new process, the "child", differs from its creating process, its "parent" only in

- - p_pid (its process id)
- - p_ppid (the process id of its parent)
- - p_addr (address of its image)
- - u.u_procp (address of its entry in the process table)

The newly created process, a.k.a. the "child process", will start running when it is selected by the swtch() function. A process calls swtch() when it needs to wait for some event, e. g. I/O completion, and just before it returns to user mode. Furthermore, swtch() is called by an ISR when the interrupted process was running in user mode.

A running process is characterized by its PROC-RUN condition. Let p point to the process' entry in the process table. Then PROC-RUN reads:
PROC-RUN:

$$\begin{array}{ll} \text{KPAR6} & = p->p\_paddr \\ \text{u.u\_procp} & = \text{p} \end{array}$$

Note that KPAR6, the kernel mode paging register 6, selects the user block which in turn contains the kernel mode stack area. So, changing KPAR6 implies exchanging the stack area. Therefore swtch needs to adjust its stack- and frame pointer to the new stack area.

While running under control of the old process, swtch() saves r5 in its process table entry. Then swtch() selects the next process to be run and establishes PROC-RUN and restores r5 for that process. That is, swtch() continues with another frame of the new process.

To save the frame pointer, swtch() calls
savfp($&u.u\_procp->p\_rsav$)

savfp() is as simple as it gets:

```
_savfp:
        mov   r5,*2(sp)
        rts   pc
```

To establish PROC_RUN and CRET-ENTRY for the next to run process, swtch calls

$$retfp(p->p\_rsav, p->p\_addr)$$

which is implemented as:

```
_retfp:
        bis   $340,PS       / set interrupt priority level to seven
        mov   (sp)+,r0
        mov   (sp)+,r5
        mov   (sp),KISA6
        mov   r5,sp
        sub   $8.,sp        / TOS in FRAME: tmp,r2c,r3c,r4c,r5c,...
        bic   $340,PS       / set interrupt priority level to zero
        jmp   (r0)
```

Exercise: Why is it absolutly necessary that retfp protects itself from being interrupted?

Exercise: Why does retfp save its return address in r0 instead of keeping it on the stack and returning with an rts instruction?

Note that swtch() neither saves nor restores the stack pointer. Instead retfp computes it from r5 according to the FRAME condition that holds with no auto variables.

When the child process starts running the first time, u.uprocp points to its parent's entry instead of to its own entry, thereby breaking PROC_RUN. To fix, swtch sets u.uprocp to its own entry.

After establishing PROC_RUN, swtch calls sureg to restore the user mode paging registers from values saved in the user block.

When setting up the process table entry for the new process, newproc() saves its framepointer in p_rsav of the new process. This effects that swtch() returns to the caller of newproc() when the new process starts running. Up to now, parent and child have no way to determine which is which since the child is a duplicate of parent. To help out, newproc() returns 0 and swtch returns 1, leading to the pattern

```
statements executed by parent
if (newproc()) {
statements executed by child only
} else {
statements executed by parent only
}
statements executed by both
```

A similar pattern applies when calling fork() in user mode: fork() returns the child's process ID when executed by the parent and zero when executed by the child.

When the child process starts running the first time, u.uprocp points to its parent's entry instead of to its own entry, thereby breaking PROC_RUN. To fix, swtch sets u.uprocp to its own entry.

Exercise: What's wrong with this version of swtch()?
swtch()

```
{
struct proc *p; /* proc entry of next process */
... /* look for next process to be run */
retfp(p->p_rsav, p->p_addr);
u.uprocp = p;
sureg();
return 1;
}
```

2.4.1 Expand and Swtch

When a process wants to expand its own image but is short of memory, it swaps out itself and then calls swtch() to transfer to some other process. The process will then be swapped in by the swapper as soon as there is enough memory and be continued when swtch() decides to. This is not as simple as it sounds! The kernel stack is part of the process' image and is constantly changing as long as the process runs. After the process initiated the swap I/O the image will be copied to disk while the contents are changing. This is an example of a "race condition", that is the outcome depends on who is first.

```
expand()
{
...
initiate to swap out the own image
...
swtch()
continue after being swapped in with greater image.
...
}
```

The disk driver might copy before swtch() is called or after swtch() is called. The first case means trouble, since the frame of swtch will not be saved to disk. In the first case, the frame pointer as saved by swtch will not be valid when the process continues after being swapped in again. To solve this problem, swtch must not save its frame pointer, but the frame pointer of expand(), that is the previous frame pointer. This is save, since the previous frame is valid all the time. The argument pfp controls which frame pointer to save:

swtch(pfp)

```
{
...
        if (pfp)
                savpfp(&u.u_procp->p_rsav);
        else
                savfp(&u.u_procp->p_rsav);
...
}
```

Of course, as you might guess, swtch(1) will not return to its caller, but to the caller of its caller.

2.4.2 Exec()

Exec() replaces its text- and data segments by the ones loaded from the program file, the name of which is passed as the first argument to the system call. The second argument of exec() is a zero terminated array of character pointers, which point to zero terminated arrays of characters, the "arguments", that are to be passed to the program.

Exec() sets the user mode paging registers to point to the newly loaded segments.

Whenever an interrupt or a system call happens, an ISR wrapper routine pushes all registers on the kernel stack before calling the ISR respective system call. On return the wrapper routine restores the registers from the stack. Exec() clears all saved values except the user mode stack pointer, which is initialized to point to the arguments to be passed to the program.

Exercise: Which program entry does exec() assume?

Answer: Zero, since the PC is set to zero.

Exec() does not create a new process, per process data remain unchanged. In particular these include:

- p_pttyp (device ID of controlling terminal)

- u_ofile (array of open files, indexed by file descriptors)

- u_cdir (current directory)

- u_uid, u_gid (user and group ID)

A tty driver sends hangup, interrupt and quit signals to all processes it controls. A process gets controlled by a terminal the first time, it opens it. A process never gets rid of its controlling terminal. Successive opening of other terminals do not change the controlling terminal.

An 'open file' is a file, i. e. an inode, together with its current file position.

The current directory comes into play, whenever a file name is to be resolved to an inode. If the name starts with an '/', the root directory is searched for, otherwise the current directory. A '/' allone means the root directory itself and an empty filename means the current directory itself.

A file gets the user and group ID of the process that created the file. The permissions to read, write and execute a file are defined in three sets. One sets applies, when the user ID of file and process match, another set applies, when the group IDs match and a third set applies, when neither group ID nor user ID match.

There are two exec() wrappers in the C-library, execl and execv. They are called like:

```
execl(filename, arg0, arg1, ..., argn, 0);
```

and

```
execv(filename, argv);
```

Here filename is a character pointer to the program's name, argv is a zero terminated array of character pointers, and arg0, arg1, ... are character pointers.

Both execl and execv pass the filename as the first parameter to exec(). Execv passes argv as the second parameter, whereas execl passes the address of arg0 as the second parameter. Execl can only be used when the number of arguments is known at coding time. It saves you building the argv array. Otherwise, execv() must be employed.

In either case, the user stack of the new program will be initialized to:

INIT-STACK:

n+1, arg0, arg1, ..., argn, -1, chars0, chars1, ..., charsn
*arg0 = first character of chars0
*arg1 = first character of chars1
...
*argn = first character of charsn

chars0, ... charsn stand for the null terminated arrays of characters whose addresses were passed to exec() in the argv array. If the total length of the strings is odd, an extra null character is appended to charsn.

All C programs start with crt0, the C runtime starter, at address zero. It inserts an argv parameter into the stack before calling the C function main(). argv is the address of arg0.

MAIN-STACK:

n+1, argv, arg0, ..., argn, -1, chars0, ... charsn
*argv = arg0

This way, main() only needs a fixed number of parameters, namely the argument count (n+1) and the argument vector (argv) to access all arguments.

Exercise: Consider the program a which executes program b:

```
main()
{
        setexit();
        signal(1, reset);
        init();
        for (;;) {
                job = wait_for_a_job();
                do(job);
        }
}
```

What are the addresses as printed by b?

Answer: The stack just before main is called by crt0 looks like this:
2, argv, arg0, arg1, -1, 'hello', ' world'

Taking into account that an extra null character is appended to chars1, you compute:

item	length	address	printed value
chars1	7	$2^{16}-8$	-8
chars0	6	$2^{16}-14$	-14
-1	2	$2^{16}-16$	-16
arg1	2	$2^{16}-18$	-18
arg0	2	$2^{16}-20$	-20
argv	2	$2^{16}-22$	-22

The "-1" on the stack is used by ps(I) to find the beginning of chars0, when it prints the command of a process.

Exercise: Assume, zero instead of '-1' is the value that exec inserts as the start marker. ps searches the start marker beginning at the right end of the stack. What would be the "command" as printed by ps for a process executing b?

Answer: Nothing, if an extra null character was appended for even length padding, since then the null character terminating charsn and the extra null character combine as a zero valued integer, which will then be interpreted as the start marker by ps. All this trouble can be solved, if exec would append a '1' character for even length padding.

When the shell executes a command, it sets each entry in the argument vector to the "words" of the command. E. g. if the command reads

% a hello world

the three words "a", "hello", and "world" are passed. Its file name, in our case "/usr/helbig/a", is not passed to the program.

Exercise: The linked list of frames of a C user program is terminated by a certain value of the frame pointer. Which value and why.

Answer: The value is zero. Exec clears r5, which is the frame pointer as saved in main's frame.

2.4.3 Exit() and Wait()

If a process is done or terminated by a signal, it calls exit() which swaps out its user block, frees all memory and notifies its parent. Exit() does not yet clear its entry in the process table, which is still needed to pass information to the parent process, when it executes wait(). Exit() has one argument, the 'status', that is passed by wait() to the parent process. This way, the child passes information to its parent.

The wait() call waits for the termination of one of its children. It returns three values:

- the process ID of the terminated child,
- the 'status' as passed to exit,
- the signal number, if the child was terminated by a signal, or zero otherwise.

Finally, wait() releases the process table entry and frees the user block of the terminated child. So, if a process terminates without waiting for each of its children, the process table will be filled with entries of terminated process, a.k.a "zombies". To avoid this, exit() sets the parent of each of its children to '1', the process ID of the init process. To take care of orphaned terminated children, init may never terminate itself and must repeatedly call wait.

2.4.4 Initializing and running the multiuser service.

Multiuser service means that several users may authenticate themselves, start programs and leave the system when done.

In Unix v6, the course of action is:

- swapper process:

This process is the first process and the ancestor of all other processes. It is the only process, which is not created by newproc(). It is its own parent. It is never swapped out and it never runs in user mode. It creates the init process and then waits in a loop for processes to be swapped in or out. The swapper initializes itself to:

controlling terminal:	none
open files:	none
current directory:	"/"
user and group ID:	zero, that is "root".

- init process:

The init process creates a text segment and copies a small machine program into it:

```
execl("/etc/init", "/etc/init", 0);
for (;;);
```

Then init enters user mode with the PC set to zero, thus executing the above program.

If "/etc/init" does not exist, Unix will loop for ever with two processes – not very exciting.

"init" starts with creating a new process, called "init2", and waits for for completion of "init2".

- init2 process:

init2 wants to execute the shell which assumes the first two file descriptors ("standard in" = 0, "standard out" = 1, and "standard error" = 2) to be valid, that is point to open files.

Therefore init2 opens "/", which sets standard in, and duplicates it by calling dup(), to let the standard out file descriptor point to the same file. With these settings, init2 calls

execl("/bin/sh", "/bin/sh", "/etc/rc", 0);

The shell then executes its argument, the file "/etc/rc", which reads on my system:

```
rm -f /etc/mtab
/etc/update
/etc/mount /dev/rk1 /usr/source
rm -f /tmp/*
```

These commands are run with the settings:

controlling terminal:	none
open files:	"/etc/rc", "/", "/"
current directory:	"/"
user and group ID:	zero, that is "root".

The shell sets the standard input to the command file and calls dup(1) to set the standard error descriptor. The other settings are still inherited from the swapper process.

Daemons like /etc/update should be run without a controlling terminal, so they won't be killed by signals sent from a tty. Update loops, calling sync() every 30 seconds to flush buffer contents to disks. Before looping, update forks. The parent process exits without waiting. The child process loops. This idiom is typically for daemons: It lets the shell continue after starting the daemon. Daemons should not use the inherited file descriptors if started by /etc/rc. Writing to "/" is not possible.

When the shell executed the last command, it exits from the init2 process and its parent, the init process, continues.

- init process:
It reads the file /etc/ttys into a tty-table. Thise file specifies for each terminal device file whether or not it is to be served. For each such device file, init creates a tty process, remembers its process id in the tty table and then loops waiting for termination of child processes. When one does, it uses the child's process id to locate its entry in the tty-table, forks a new tty process for this entry and continues waiting.

- tty process:
This process modifies the process settings to:

controlling terminal:	/dev/ttyx	(as specified by tty table)
open files:	/dev/ttyx, /dev/ttyx	
current directory:	"/"	
user and group ID:	zero, that is "root".	

It then changes permission and owner of the terminal, such that only root can read from it befor executing /etc/getty with:

execl("/etc/getty", "-", com, 0);

Com is a one character string that is taken from /etc/ttys and used by getty to identify the communication parameters of the terminal line, like baud rate, number of stop bits, etc. Gettys task is to configure the terminal. It does so by trying different settings writing "login:" and reading the user name. When getty thinks, the terminal settings are ok, the tty process executes the login program:

execl("/bin/login", "login", name, 0);

passing the user name as an argument.

The login program reads the file /etc/passwd, locates the user's entry and asks the user to type in a password. It encodes the password and compares with the one from /etc/passwd. If valid, it sets the process uid, gid and cdir from /etc/passwd before it finally executes the program as specified in /etc/passwd, which is usually "/bin/sh".

execl(program, "-", 0)

The shell, still executed by the tty-process, calls dup(1) to set file descriptor 2 to the terminal and starts reading and executing command lines. The shell is executed with these process settings, which are inherited by its children.

controlling terminal:	/dev/ttyx	(as specified by tty table)
open files:	/dev/ttyx, /dev/ttyx, /dev/ttyx	
current directory:	user's home	(as specified by /etc/passwd)
user and group ID:	as specified by /etc/passwd	

The following table relates processes to processes they create and to programs they execute in user mode.

process	creates process	executes program
swapper	init	-
init	init2, tty processes	icode, /etc/init
init2	-	/bin/sh
tty processes	-	/etc/getty, /bin/login, /bin/sh

This table again stresses the difference of 'process' and 'program'.