0cm [*UNIX 6th Edition Commentary*] []*UNIX 6th Edition Commentary* [1] []1

A COMMENTARY ON THE SIXTH EDITION UNIX OPERATING SYSTEM

J. Lions.
Department of Computer Science
The University of New South Wales

This booklet has been produced for students at the University of New South Wales taking courses 6.602B and 6.657G. It is intended as a companion to, and commentary on, the booklet *UNIX Operating System Source Code, Level Six*.

The UNIX Software System was written by K. Thompson and D. Ritchie of Bell Laboratories, Murray Hill, NJ. It has been made available under a license from the Western Electric Company.

Contents

Preface

This book is an attempt to explain in detail the nucleus of one of the most interesting computer operating systems to appear in recent years.

It is the UNIX Time-sharing System, which runs on the larger models of Digital Equipment Corporation's PDP11 computer system, and was developed by Ken Thompson and Dennis Ritchie at Bell Laboratories. It was first announced to the world in the July, 1974 issue of the "Communications of the ACM".

Very soon in our experience with UNIX, it suggested itself as an interesting candidate for formal study by students, for the following reasons:

- it runs on a system which is already available to us;
- it is compact and accessible;
- it provides an extensive set of very usable facilities;

  it is intrinsically interesting, and in fact breaks new ground in a number of areas.

Not least amongst the charms and virtues of the UNIX Time-sharing System is the compactness of its source code. The source code for the permanently resident "nucleus" of the system when only a small number of peripheral devices is represented, is comfortably less than 9,000 lines of code.

It has often been suggested that 1,000 lines of code represents the practical limit in size for a program which is to be understood and maintained by a single individual. Most operating systems either exceed this limit by one or even two orders of magnitude, or else offer the user a very limited set of facilities, i.e. either the details of the system are inaccessible to all but the most determined, dedicated and long-suffering student, or else the system is rather specialised and of little intrinsic interest.

There seem to be three main approaches to teaching Operating Systems. First there is the "*general principles*" approach, wherein fundamental principles are expounded, and illustrated by references to various existing systems, (most of which happen to be outside the students' immediate experience). This is the approach advocated by the COSINE Committee, but in our view, many students are not mature or experienced enough to profit from it.

The second approach is the "*building block*" approach, wherein the students are enabled to synthesize a small scale or "toy" operating system for themselves. While undoubtedly this can be a valuable exercise, if properly organised, it cannot but fail to encompass the complexity and sophistication of real operating systems, and is usually biased towards one aspect of operating system design, such as process synchronisation.

The third approach is the "*case study*" approach. This is the one originally recommended for the Systems Programming course in "Curriculum '68", the report of the ACM Curriculum Committee on Computer Science, published in the March, 1968 issue of the "Communications of the ACM".

Ten years ago, this approach, which advocates devoting "most of the course to the study of a single system" was unrealistic because the cost of providing adequate student access to a suitable system was simply too high.

Ten years later, the economic picture has changed significantly, and the costs are no longer a decisive disadvantage if a minicomputer system can be the subject of study. The considerable advantages of the approach which undertakes a detailed analysis of an existing system are now attainable.

In our opinion, it is highly beneficial for students to have the opportunity to study a working operating system in all its aspects.

Moreover it is undoubtedly good for students majoring in Computer Science, to be confronted at least once in their careers, with the task of reading and understanding a program of major dimensions.

In 1976 we adopted UNIX as the subject for case study in our courses in Operating Systems at the University of New South Wales. These notes were prepared originally for the assistance of students in those courses (6.602B and 6.657G).

The courses run for one semester each. Before entering either Course, students are presumed to have studied the PDP11 architecture and assembly language, and to have had an opportunity to use the UNIX operating system during exercises for earlier courses.

In general, students seem to find the new courses more onerous, but much more satisfying than the previous courses based on the "general principles" approach of the COSINE Committee.

Some mention needs to be made regarding the documentation provided by the authors of the UNIX system. As reproduced for use on our campus, this comprises two volumes of A4 size paper, with a total thickness of 3 cm, and a weight of 1250 grams.

A first observation is that the whole documentation is not unreasonably transportable in a student's brief case. However it must not be assumed that this amount of documentation, which is written in a fresh, terse, whimsical style, is necessarily inadequate.

In fact the second observation (which is only made after considerable experience) is that for reference purposes, the documentation is remarkably comprehensive. However there is plenty of scope for additional tutorial material, one part of which, it is hoped, is satisfied by these notes.

The actual UNIX operating system source code is recorded in a separate companion volume entitled "UNIX Operating System Source Code", which was first printed in July, 1976. This is a specially edited selection of code from the Level Six version of UNIX, as received by us in December, 1975.

During 1976, an initial version of the present notes was distributed in roneoed form, and only in the latter part of the year were the facilities of the "nroff" text formatting program exploited. The opportunity has recently been taken to revise and "nroff" the earlier material, to make some revisions and corrections, and to integrate them into their present form.

A decision had to be made quite early regarding the order of presentation of the source code. The intention was to provide a reasonably logical sequence for the student who wanted to learn the whole system. With the benefit of hindsight, a great many improvements in detail are still possible, and it is intended that these changes will be made in some future edition.

It is our hope that this book will be of interest and value to many students of the UNIX Time-sharing System. Although not prepared primarily for use as a reference work, some will wish to use it as such. The indices provided at the end should go some of the way towards satisfying the requirement for reference material at this level.

Since these notes refer to proprietary material administered by the Western Electric Company, they can only be made available to licensees of the UNIX Time-sharing System and hence are unable to be published through more usual channels.

Corrections, criticism and suggestions for improvement of these notes will be very welcome.

Acknowledgments

The preparation of these notes has been encouraged and supported by many of my colleagues and students including David Carrington, Doug Crompton, Ian Hayes, David Horsfall, Peter Ivanov, Ian Johnstone, Chris Maltby, Dave Milway, John O'Brien and Greg Rose.

Pat Mackie and Mary Powter did much of the initial typing, and Adele Green has assisted greatly in the transfer of the notes to "nroff" format.

David Millis and the Publications Section of the University of New South Wales have assisted greatly with the mechanics of publication, and Ian Johnstone and the Australian Graduate School of Management provided facilities for the preparation of the final draft.

Throughout this project, my wife Marianne has given me unfailing moral support and much practical support with proof-reading.

Finally Ken Thompson and Dennis Ritchie started it all.

To all the above, I wish to express my sincere thanks.

The co-operation of the "nroff" program must also be mentioned. Without it, these notes could never have been produced in this form. However it has yielded some of its more enigmatic secrets so reluctantly, that the author's gratitude is indeed mixed. Certainly "nroff" itself must provide a fertile field for future practitioners of the program documenter's art.

John Lions
Kensington, NSW
May, 1977

1 Introduction

"UNIX" is the name of a time-sharing system for PDP11 computers, written by Ken Thompson and Dennis Ritchie at Bell Laboratories. It was described by them in the July, 1974 issue of the "Communications of the ACM".

UNIX has proved to be effective, efficient and reliable in operation and was in use at more than 150 installations by the end of 1976.

The amount of effort to write UNIX, while not inconsiderable in itself (~10 man years up to the release of the Level Six system) is insignificant when compared to other systems. (For instance, by 1968, OS/360 was reputed to have consumed more then five man millennia and TSS/360, another IBM operating system, more than one man millennium.)

Of course there are systems which are easier to understand than UNIX but, it may be asserted, these are invariably much simpler and more modest in what they attempt to achieve. As far as the list of features offered to users is concerned, UNIX is in the "big league". In fact it offers many features which are notable by their absence from some of the well-known major systems.

1.1 The UNIX Operating System

The purpose of this document, and its companion, the "UNIX Operating System Source Code", is to present in detail that part of the UNIX time-sharing system which we choose to call the "UNIX Operating System", namely the code which is permanently resident in the main memory during the operation of UNIX. This code has the following major functions:

- initialisation;
- process management;
- system calls;
- interrupt handling;
- input/output operations;
- file management.

1.2 Utilities

The remaining part of UNIX (which is much larger!) is composed of a set of suitably tailored programs which run as "user programs", and which, for want of a better term, may be termed "utilities".

Under this heading come a number of programs with a very strong symbiotic relationship with the operating system such as

- the "shell" (the command language interpreter)
- "/etc/init" (the terminal configuration controller)

and a number of file system management programs such as:

check	du	rmdir
chmod	mkdir	sync
clri	mkfs	umount
df	mount	update

It should be pointed out that many of the functions carried out by the above-named programs are regarded as operating system functions in other computer systems, and that this certainly does contribute significantly to the bulk of these other systems as compared with the UNIX Operating System (in the way we have defined it).

Descriptions of the function and use of the above programs may be found in the "UNIX Programmer's Manual" (UPM), either in Section I (for the commonly used programs) or in Section VIII (for the programs used only by the System Manager).

1.3 Other Documentation

These notes make frequent reference to the "UNIX Programmer's Manual" (UPM), occasional reference to the "UNIX Documents" booklet, and constant reference to the "UNIX Operating System Source Code".

All these are relevant to a complete understanding of the system. In addition, a full study of the assembly language routines requires reference to the "PDP11 Processor Handbook", published by Digital Equipment Corporation.

1.4 UNIX Programmer's Manual

The UPM is divided into eight major sections, preceded by a table of contents and a KWIC (Key Word In Context) index. The latter is mostly very useful but is occasionally annoying, as some indexed material does not exist, and some existing material is not indexed.

Within each section of the manual, the material is arranged alphabetically by subject name. The section number is conventionally appended to the subject name, since some subjects appear in more than one section, e.g. "CHDIR(I)" and "CHDIR(II)".

Section I contains commands which either are recognised by the "shell" command interpreter, or are the names of standard user utility programs;

Section II contains "system calls" which are operating system routines which may be invoked from a user program to obtain operating system service. A study of the operating system will render most of these quite familiar;

Section III contains "subroutines" which are library routines which may be called from a user program. To the ordinary programmer, the distinctions between Sections II and III often appear somewhat arbitrary. Most of Section III is irrelevant to the operating system;

Section IV describes "special files", which is another name for peripheral devices. Some of these are relevant, and some merely interesting. It depends where you are;

Section V describes "File Formats and Conventions". A lot of highly relevant information is tucked away in this section;

Sections VI and VII describe "User Maintained" programs and subroutines. No UNIXophile will ignore these sections, but they are not particularly relevant to the operating system;

Section VIII describes "system maintenance" (software, not hardware!). There is lots of useful information here, especially if you are interested in how a UNIX installation is managed.

1.5 UNIX Documents

This is a somewhat miscellaneous collection of essays of varying degrees of relevance:

- *Setting up UNIX* really belongs in Section VIII of the UPM (it's relevant);
- *The UNIX Time-sharing System* is an updated version of the original "Communications of the ACM" paper. It should be re-read at least once per month;
- *UNIX for Beginners* is useful if your UNIX experience is limited;
- The tutorials on "C" and the editor, and the reference manuals for "C" and the assembler are highly useful unless you are completely expert;
- *The UNIX I/O System* provides a good overview of many features of the operating system;
- *UNIX Summary* provides a check list which will be useful in answering the question what does an operating system do?

1.6 UNIX Operating System Source Code

This is an edited version of the operating system as supplied by Bell Laboratories.

The code selection presumes a "model" system consisting of:

- PDP11/40 processor;
- RK05 disk drives;
- LP11 line printer;
- PC11 paper tape reader/punch;
- KL11 terminal interface.

The principal editorial changes to the source code are as follows:

- the order of presentation of files has been changed;
- the order of material within several files has been changed;
- to a very limited extent, code has been transferred between files (with hindsight a lot more of this would have been desirable);
- about 5% of the lines have been shortened in various ways to less than 66 characters (by elimination of blanks, rearrangement of comments, splitting into two lines, etc.);
- a number of comments consisting of a line of underscore characters have been introduced, particularly at the end of procedures;
- the size of each file has been adjusted to an exact multiple of 50 lines by padding with blank lines;

The source code has been printed in double column format with fifty lines per column, giving one hundred lines per sheet (or page). Thus there is a convenient relationship between line numbers and sheet numbers.

A number of summaries have been included at the beginning of the Source Code volume:

- A *Table of Contents* showing files in order of appearance, together with the Procedures they contain;
- An *alphabetical list* of procedures with line numbers;
- A *list of Defined Symbols* with their values;
- A *Cross Reference Listing* giving the line numbers where each symbol is used. (Reserved words in "C" and a number of commonly used symbols such as "p" and "u" have been omitted.)

1.7 Source Code Selections

The source code has been divided into five sections, each devoted primarily to a single major aspect of the system.

The intention, which has been largely achieved, has been to make each section sufficiently self-contained so that it may be studied as a unit and before its successors have been mastered:

Section One deals with system initialisation, and process management. It also contains all the assembly language routines;

Section Two deals with interrupts, traps, system calls and signals (software interrupts);

Section Three deals primarily with disk operations for program swapping and basic, block oriented input/output. It also deals with the manipulation of the pool of large buffers;

Section Four deals with files and file systems: their creation, maintenance, manipulation and destruction;

Section Five deals with "character special files", which is the UNIX term for slow speed peripheral devices which operate out of a common, character oriented, buffer pool.

The contents of each section is outlined in more detail in Chapter Four.

1.8 Source Code Files

Each of the five sections just described consists of several source code files. The name of each file includes a suffix which identifies its type:

".s" denotes a file of assembly language statements;

".c" denotes a file of executable "C" language statements;

".h" denotes a file of "C" language statements which is not for separate compilation, but for inclusion in other ".c" files when they are compiled i.e. the ".h" files contain global declarations.

1.9 Use of these notes

These notes, which are intended to supplement the comments already present in the source code, are not essential for understanding the UNIX operating system. It is perfectly possible to proceed without them, and you should attempt to do so as long as you can.

The notes are a crutch, to aid you when the going becomes difficult. If you attempt to read each file or procedure on your own first, your initial progress is likely to be slower, but your ultimate progress much faster. Reading other people's programs is an art which should be learnt and practised because it is useful!

1.10 A Note on Programming Standards

You will find that most of the code in UNIX is of a very high standard. Many sections which initially seem complex and obscure, appear in the light of further investigation and reflection, to be perfectly obvious and "the only way to fly".

For this reason, the occasional comments in the notes on programming style, almost invariably refer to apparent lapses from the usual standard of near perfection.

What caused these? Sometimes it appears that the original code has been patched expediently. More than once apparent lapses have proved not to be such: the "bad" code has been found in fact to incorporate some subtle feature which was not at all apparent initially. And some allowance is certainly needed for occasional human weakness.

But on the whole you will find that the authors of UNIX, Ken Thompson and Dennis Ritchie, have created a program of great strength, integrity and effectiveness, which you should admire and seek to emulate.

2 Fundamentals

UNIX runs on the larger models of the PDP11 series of computers manufactured by Digital Equipment Corporation. This chapter provides a brief summary of certain selected features of these computers with particular reference to the PDP11/40.

If the reader has not previously made the acquaintance of the PDP11 series then he is directed forthwith to the "PDP11 Processor Handbook", published by DEC.

A PDP11 computer consists of a processor (also called a CPU connected to one or more memory storage units and peripheral controllers via a bidirectional parallel communication line called the "Unibus".

2.1 The Processor

The processor, which is designed around a sixteen bit word length for instructions, data and program addresses, incorporates a number of high speed registers.

2.2 Processor Status Word

This sixteen bit register has subfields which are interpreted as follows:

bits	**description**
14,15	current mode (00 = kernel;)
12,13	previous mode (11 = user;)
5,6,7	processor priority (range 0..7)
4	trap bit
3	N, set if the previous result was negative
2	Z, set if the previous result was zero
1	V, set if the previous result gave an overflow
0	C, set if the previous operation gave a carry

The processor can operate in two different modes: kernel and user. Kernel mode is the more privileged of the two and is reserved by the operating system for its own use. The choice of mode determines:

- The set of memory management segmentation registers which is used to translate program virtual addresses to physical addresses;
- The actual register used as r6, the "stack pointer";
- Whether certain instructions such as "halt" will be obeyed.

2.3 General Registers

The processor incorporates a number of sixteen bit registers of which eight are accessible at any time as "general registers". These are known as r0, r1, r2, r3, r4, r5, r6 and r7.

The first six of the general registers are available for use as accumulators, address pointers or index registers. The convention in UNIX for the use of these registers is as follows:

r0, r1 are used as temporary accumulators during expression evaluation, to return results from a procedure, and in some cases to communicate actual parameters during a procedure call;

r2, r3, r4 are used for local variables during procedure execution. Their values are almost always stored upon procedure entry, and restored upon procedure exit;

r5 is used as the head pointer to a "dynamic chain" of procedure activation records stored in the current stack. It is referred to as the "environment pointer".

The last two of the "general registers" do have a special significance and are to all intents, "special purpose":

r6 (also known as "sp") is used as the stack pointer. The PDP11/40 processor incorporates two separate registers which may be used as "sp", depending on whether the processor is in kernel or user mode. No other one of the general registers is duplicated in this way;

r7 (also known as "pc") is used as the program instruction address register.

2.4 Instruction Set

The PDP11 instruction set includes double, single and zero operand instructions. Instruction length is usually one word, with some instructions being extended to two or three words with additional addressing information.

With single operand instructions, the operand is usually called the "destination"; with double operand instructions, the two operands are called the "source" and "destination". The various modes of addressing are described later.

The following instructions have been used in the file "m40.s" i.e. the file of assembly language support routines for use with the 11/40 processor. Note that N, Z, V and C are the condition codes i.e. bits in the processor status word ("ps"), and that these are set as side effects of many instructions besides just “bit”, “cmp” and “tst” (whose stated function is to set the condition codes).

adc Add the contents of the C bit to the destination;

add Add the source to the destination;

ash Shift the contents of the defined register left the number of times specified by the shift count. (A negative value implies a right shift.);

ashc Similar to “ash” except that two registers are involved;

asl Shift all bits one place to the left. Bit 0 becomes 0 and bit 15 is loaded into C;

asr Shift all bits one place to the right. Bit 15 is replicated and bit 0 is loaded into C;

beq Branch if eaual, i.e. if Z = l;

bge Branch if greater than or equal to, i.e. if N = V;

bhi Branch if higher, i.e if C = 0 and Z = 0;

bhis Branch if higher or the same, i.e. if C = 0;

bic Clear each bit to zero in the destination that corresponds to a non-zero bit in the source;

bis Perform an “inclusive or” of source and destination and store the result in the destination;

bit Perform a logical “and” of the source and destination to set the condition codes;

ble Branch if greater than or equal to, i.e if Z = 1 or N = V;

blo Branch if lower (than zero), if C = l;

bne Branch if not equal (to zero), i.e. if Z = 0;

br Branch to a location within the range (. -128, . +127) where “.” is the current location;

clc Clear C;

clr Clear destination to zero;

cmp Compare the source and destination to set the condition codes. N is set if the source value is less than the destination value;

dec Subtract one from the contents of the destination;

div The 32 bit two’s complement integer stored in rn and r(n+l) (where n is even) is divided by the source operand. The quotient is left in rn, and the remainder in r(n+l);

inc Add one to the contents of the destination;

jmp Jump to the destination;

jsr Jump to subroutine. Register values are shuffled as follows:

pc, rn, –(sp) = dest., pc, rn

mfpi Push onto the current stack the value of the designated word in the “previous” address space;

mov Copy the source value to the destination;

mtpi Pop the current stack and store the value in the designated word in the “previous” address space;

mul Multiply the contents of rn and the source. If n is even, the product is left in rn and r(n+l);

reset Set the INIT line on the Unibus for 10 milliseconds. This will have the effect of reinitialising all the device controllers;

ror Rotate all bits of the destination one place to the right. Bit 0 is loaded into C, and the previous value of C is loaded into bit 15;

rts Return from subroutine. Reload pc from rn, and reload rn from the stack;

rtt Return from interrupt or trap. Reload both pc and ps from the stack;

sbc Subtract the carry bit from the destination;

sob Subtract one from the designated register. If the result is not zero, branch back “offset” words;

sub Subtract the source from the destination;

swab Exchange the high and low order bytes in the destination;

tst Set the condition codes, N and Z, according to the contents of the destination;

wait Idle the processor and release the Unibus until a hardware interrupt occurs.

The “byte” version of the following instructions are used in the file “m40.s”, as well as the “word” versions described above:

bis	inc
clr	mov
cmp	tst

2.5 Addressing Modes

Much of the novelty and complexity of the PDP11 instruction set lies in the variety of addressing modes which may be used for defining the source and destination operands.

The addressing modes which are used in "m40.s" are described below.

Register Mode: The operand resides in one of the general registers, e.g.

```
clr r0
mov r1,r0
add r4,r2
```

In the following modes, the designated register contains an address value which is used to locate the operand.

Register Deferred Mode: The register contains the address of the operand, e.g.

```
inc (r1)
asr (sp)
add (r2),r1
```

Autoincrement Mode: The register contains the address of the operand. As a side effect, the register is incremented after the operation, e.g.

```
clr  (r1)+
mfpi (r0)+
mov  (r1)+,r0
mov  r2,(r0)+
cmp (sp)+,(sp)+
```

Autodecrement Mode: The register is decremented and then operand, e.g.

```
inc -(r0)
mov -(r1),r2
mov (r0)+,-(sp)
clr -(sp)
```

Index Mode: The register contains a value which is added to a sixteen bit word following the instruction to form the operand address, e.g.

```
clr  2(r0)
movb 6(sp),(sp)
movb _reloc(r0),r0
mov  -10(r2),(r1)
```

Depending on your viewpoint, in this mode the register is either an index register or a base register. The latter case actually predominates in "m40.s". The third example above is actually one of the few uses of a register as an index register. (Note that "_reloc" is an acceptable variable name.)

There are two addressing modes whose use is limited to the following two examples:

```
jsr pc,(r0)+
jmp *0f(r0)
```

The first example involves the use of the "*autoincrement deferred*" mode. (This occurs in the routine "call" on lines 0785, 0799.) The address of a routine intended for execution is to be found in the word addressed by r0, i.e. two levels of indirection are involved. The fact that r0 is incremented as a side effect is not relevant in this usage.

The second example (which occurs on lines 1055, 1066) is an instance of the "*index deferred*" mode. The destination of the "jump" is the content of the word whose address is labelled by "0f" *plus* the value of r0 (a small positive integer). This is a standard way to implement a multi-way switch.

The following two modes use the program counter as the designated register to achieve certain special effects.

Immediate Mode: This is the pc autoincrement mode. The operand is thus extracted from the program string, i.e. it becomes an immediate operand, e.g.

```
add $2,r0
add $2,(r1)
bic $17,r0
mov $KISA0,r0
mov $77406,(r1)+
```

Relative Mode: This is the pc index mode. The address relative to the current program counter value is extracted from the program string and added to the pc value to form the absolute address of the operand, e.g.

```
bic $340,PS
bit $1,SSR0
inc SSR0
mov (sp),KISA6
```

It may be noted that each of the modes "index", "index deferred", "immediate" and "relative" extends the instruction size by one word.

The existence of the "autoincrement" and "autodecrement" modes, together with the special attributes of r6, make it conveniently possible to store many operands in a stack, or LIFO list, which grows downwards in memory. There are a number of advantages which flow from this: code string lengths are shorter and it is easier to write position independent code.

2.6 Unix Assembler

The UNIX assembler is a two pass assembler without macro facilities. A full description may be found in the "UNIX Assembler Reference Manual" which is contained in the "UNIX Documents"

The following brief notes should be of some assistance:

(a) a string of digits may define a constant number. This is assumed to be an octal number unless the string is terminated by a period ("."), when it is interpreted as a decimal number.

(b) The character "/" is used to signify that the rest of the line is a comment;

(c) If two or more statements occur on the same line, they must be separated by semicolons;

(d) The character "." is used to denote the current location;

(e) UNIX assembler uses the characters $ and "*" where the DEC assemblers use "#" and "@" respectively.

(f) An identifier consists of a set of alphanumeric characters (including the underscore). Only the first eight characters are significant and the first may not be numeric;

(g) Names which occur in "C" programs for variables which are to be known globally, are modified by the addition of a prefix consisting of a single underscore. Thus for example the variable "_regloc" which occurs on line 1025 in the assembly language file, "m40.s", refers to the same variable as "regloc" at line 2677 of the file, "trap.c";

(h) There are two kinds of statement labels: name labels and numeric labels. The latter consist of a single digit followed by a colon, and need not be unique. A reference to "nf" where "n" is a digit, refers to the first occurrence of the label "n:" found by searching forward.

A reference to "nb" is similar except that the search is conducted in the backwards direction;

(i) An assignment statement of the form

identifier = expression

associates a value **and** type with the identifier. In the example

. = 60^.

the operator '^' delivers the value of the first operand and the type of the second operand (in this case, "location");

(j) The string quote symbols are "<" and ">".

(k) Statements of the form

.globl x, y, z

serve to make the names "x", "y" and "z" external;

(l) The names "_edata" and "_end" are loader pseudo variables which the define the size of the data segment, and the data segment plus the bss segment respectively.

2.7 Memory Management

Programs running on the PDP11 may address directly up to 64K bytes (32K words) of storage. This is consistent with an address size of sixteen bits. Since it is economical and not unreasonable to do so the larger PDP11 models may be equipped with larger amounts of memory (up to 256K bytes for the PDP11/40) plus a mechanism for converting sixteen bit *virtual* (program) addresses into *physical* addresses of eighteen bits or more. The mechanism, which is known as the memory management unit, is simpler on the PDP11/40 than on the 11/45 or the 11/70.

On the PDP11/40 the memory management unit consists of two sets of registers for mapping virtual addresses to physical addresses. These are known as "active page registers" or "segmentation registers". One set is used when the processor is in user mode and the other set, in kernel mode. Changing the contents of these registers changes the details of these mappings. The ability to make these changes is a privilege that the operating system keeps firmly to itself.

2.8 Segmentation Registers.

Each set of segmentation registers is composed of eight pairs, each consisting of a "*page address register*" (PAR) and a "*description register*" (PDR).

Each pair of registers controls the mapping of one *page* i.e. one eighth part of the virtual address space which 8K bytes (4K words).

Each page may be regarded as an aggregate of 128 blocks, each of 64 bytes (32 words). This latter size is the "grain size" for the memory mapping function, and as a practical consequence, it is also the "grain size" for memory allocation.

Any virtual address belongs to one page or other. The corresponding physical address is generated by adding the relative address within the page to the contents of the corresponding PAR to form an extended address (18 bits on the PDP11/40 and 11/45; 22 bits on the 11/70).

Thus each page address register acts as a relocation register for one page.

Each page can be divided on a 32 word boundary into two parts, an upper part and lower part. Each such part has a size which is a multiple of 32 words. In particular one part may be null, in which case the other part coincides with the whole page.

One of the two parts is deemed to contain valid virtual addresses. Addresses in the remaining part are declared invalid. Any attempt to reference an invalid address will be trapped by the hardware. The advantage of this scheme is that space in the physical memory need only be allocated for the valid r)art of a page.

2.9 Page Description Register

The page description register defines:

(a) the size of the lower part of the page. (The number stored is actually the number of 32 word blocks less one);

(b) a bit which is set when the upper part is the valid part. (Also known as the "expansion direction" bit);

(c) access mode bits defining "no access" or "read only access" or "read/write access".

Note that if the valid part is null, this fact must be shown by setting the access bits to "no access".

2.10 Memory Allocation

The hardware does not dictate the way areas in physical memory which correspond to the valid parts of pages should be allocated (except to the extent that they must begin and end on a 32 word boundary). These areas may be allocated in any order and may overlap to any extent.

In practice the allocation of areas of physical memory is much more disciplined as we shall see in Chapter Seven. Areas for pages which are related are most often allocated contiguously and in the order of their page numbers, so that all the segment areas associated with a single program are contained within one or at most two large areas of physical memory.

2.11 Memory Management Status Registers

In addition to the segmentation registers, on the PDP11/40 there are two memory management status registers:

SR0 contains abort error flags and other essential information for the operating system. In particular memory management is enabled when bit 0 of SR0 is on;

SR2 is loaded with the 16 bit virtual address at the beginning of each instruction fetch.

2.12 "i" and "d" Spaces

In the PDP11/45 and 11/70 systems, there are additional sets of segmentation registers. Addresses created using the pc register (r7) are said to belong to "i" space, and are translated by a different set of segmentation registers from those used for the remaining addresses which are said to belong to "d" space.

The advantage of this arrangement is that both "i" and "d" spaces may occupy up to 32K words, thus allowing the maximum space which can be allocated to a program to be increased to twice the space available on the PDP11/40.

2.13 Initial Conditions

When the system is first started after all the devices on the Unibus have been reinitialised, the memory management unit is disabled and the processor is in kernel mode.

Under these circumstances, virtual (byte) addresses in the range 0 to 56K are mapped into identically valued physical addresses. However the highest page of the virtual address space is mapped into the highest page of the physical address space, i.e. on the PDP11/40 or 11/45, addresses in the range

0160000 to 0177777

are mapped into the range

0760000 to 0777777

2.14 Special Device Registers

The high page of physical memory is reserved for various special registers associated with the processor and the peripheral devices. By sacrificing one page of memory space in this way, the PDP11 designers have been able to make the various device registers accessible without the need to provide special instruction types.

The method of assignment of addresses to registers in this page is a black art: the values are hallowed by tradition and are not to be questioned.

3 Reading "C" Programs

Learning to read programs written in the "C" language is one of the hurdles that must be overcome before you will be able to study the source code of UNIX effectively.

As with natural languages, reading is an easier skill to acquire than writing. Even so you will need to be careful lest some of the more subtle points pass you by.

There are two of the "UNIX Documents" which relate directly to the "C" language:

"C Reference Manual", by Dennis Ritchie

"Programming in C – A Tutorial", by Brian Kernighan

You should read them now, as far as you can, and return to reread them from time to time with increasing comprehension.

Learning to write "C" programs is not required. However if you have the opportunity, you should attempt to write at least a few small programs. This does represent the accepted way to learn a programming language, and your understanding of the proper use of such items as:

semicolons;
"=" and "=="
"{" and "}"
"++" and "−−"
declarations;
register variables;
"if" and "for" statements

You will find that "C" is a very convenient language for accessing and manipulating data structures and character strings, which is what a large part of operating systems is about. As befits a terminal oriented language, which requires concise, compact expression, "C" uses a large character set and makes many symbols such as "*" and "&" work hard. In this respect it invites comparison with APL.

There many features of "C" which are reminiscent of PL/1, but it goes well beyond the latter in the range of facilities provided for structured programming.

3.1 Some Selected Examples

The examples which follow are taken directly from the source code.

3.2 Example 1

The simplest possible procedure, which does nothing, occurs twice(!) in the source code as "nullsys" (2864) and "nulldev" (6577), sic.

```
6577 nulldev ()
     {
     }
```

While there are no parameters, the parentheses, "(" and ")", are still required. The brackets "{" and "}" delimit the procedure body, which is empty.

3.3 Example 2

The next example is a little less trivial:

```
6566 nodev ()
     {
      u.u_error = ENODEV;
     }
```

The additional statement is an assignment statement. It is terminated by a semicolon which is part of the statement, not a statement separator as in Algol-like languages.

"ENODEV" is a defined symbol, i.e. a symbol which is replaced by an associated character string by the compiler preprocessor before actual compilation. "ENODEV" is defined on line 0484 as 19. The UNIX convention is that defined symbols are written in upper case, and all other symbols in lower case.

"=" is the assignment operator, and "u.u_error" is an element of the structure "u". (See line 0419.) Note the use of "." as the operator which selects an element of a structure. The element name is "u_error" which may be taken as a paradigm for the way names of structure elements are constructed in the UNIX source code: a distinguishing letter is followed by an underscore followed by a name.

3.4 Example 3

```
6585 bcopy (from, to, count)
     int *from, *to;
     {
      register *a, *b, c;
      a = from;
      b = to;
      c = count;
      do
        *b++ = *a++;
      while (--cc);
     }
```

The function of this procedure is very simple: it copies a specified number of words from one set of consecutive locations to another set.

There are three parameters. The second line

```
int *from, *to;
```

specifies that the first two variables are pointers to integers. Since no specification is supplied for the third parameter, it is assumed to be an integer by default.

The three local variables, a, b, and c, have been assigned to registers, because registers are more accessible and the object code to reference them is shorter. "a" and "b" are pointers to integers and "c" is an integer. The register declaration could have been written more pedantically as

```
register int *a, *b, c;
```

to emphasise the connection with integers.

The three lines beginning with "do" should be studied carefully. If "b" is a "pointer to integer" type, then

```
*b
```

denotes the integer pointed to. Thus to copy the value pointed to by "a" to the location designated by "b", we could write

```
*b = *a;
```

If we wrote instead

```
b = a;
```

this would make the value of "b" the same as the value of "a", i.e. "b" and "a" would point to the same place. Here at least, that is not what is required.

Having copied the first word from source to destination, we need to increase the values of "b" and "a" so that the point to the next words of their respective sets. This can be done by writing

```
b = b+1; a = a+1;
```

but "C" provides a shorter notation (which is more useful when the variable names are longer) viz.

```
b++; a++;
```

Now there is no difference between the statements "b++;" and "++b;" here.

However "b++" and "++b" may be used as terms in an expression, in which case they are different. In both cases the effect of incrementing "b" is retained, but the value which enters the expression is the initial value for "b++" and the final value for "++b".

The "−−" operator obeys the same rules as the "++" operator, except that it decrements by one. Thus "−−c" enters an expression as the value after decrementation.

The "++" and "−−" operators are very useful, and are used throughout UNIX. Occasionally you will have to go back to first principles to work out exactly what their use implies. Note also there is a difference between `*b++` and `(*b)++`.

These operators are applicable to pointers to structures as well as to simple data types. When a pointer which has been declared with reference to a particular type of structure is incremented, the actual value of the pointer is incremented by the size of the structure.

We can now see the meaning of the line

```
*b++ = *a++;
```

The word is copied and the pointers are incremented, all in one hit.

The line

```
while (--c);
```

delimits the end of the set of statements which began after the "do". The expression in parentheses "−−c", is evaluated and tested (the value tested is the value after decrementation). If the value is non-zero, the loop is repeated, else it is terminated.

Obviously if the initial value for "count" were negative, the loop would not terminate properly. If this were a serious possibility then the routine would have to be modified.

3.5 Example 4

```
6619 getf (f)
     {
       register *fp, rf;
       rf = f;
       if (rf < 0 || rf >= NOFILE)
         goto bad;
       fp = u.u_ofile[rf];
       if (fp != NULL)
       return (fp);
     bad:
       u.u_error = EBADF;
       return (NULL);
     }
```

The parameter "f" is a presumed integer, and is copied directly into the register variable "rf". (This pattern will become so familiar that we will now cease to remark upon it.)

The three simple relational expressions

```
rf < 0    rf >=NOFILE    fp != NULL
```

are each accorded the value one if true, and the value zero if false. The first tests if the value of "rf" is less than zero, the second, if "rf" is greater than the value defined by "NOFILE" and the third, if the value of "fp" is not equal to "NULL" (which is defined to be zero).

The conditions tested by the "if" statements are the arithmetic expressions contained within parentheses.

If the expression is greater than zero the test is successful and the following statement is executed. Thus if for instance, "fp" had the value 001375, then

```
fp != NULL
```

is true, and as a term in an arithmetic expression, is accorded the value one. This value is greater than zero, and hence the statement

```
return(fp);
```

would be executed, to terminate further execution of "getf", and to return the value of "fp" to the calling procedure as the result of "getf".

The expression

```
rf < 0 || rf >= NOFILE
```

is the logical disjunction ("or") of the two simple relational expressions.

An example of a "goto" statement and associated label will be noted.

"fp" is assigned a value, which is an **address**, from the "rf"-th element of the array of integers "u_ofile", which is embedded in the structure "u".

The procedure "getf" returns a value to its calling procedure. This is either the vale of "fp" (i.e. an address) or "NULL".

3.6 Example 5

```
2113 wakeup (chan)
     {
       register struct proc *p;
       register c, i;
       c= chan;
       p= &proc[0];
       i= NPROC;
       do {
            if (p->p_wchan == c) {
              setrun(p);
            }
            p++;
       } while (--i);
     }
```

There are a number of similarities between this example and the previous one. We have a new concept however, an array of structures. To be just a little confusing, in this example it turns out that both the array and the structure are called "proc" (yes, "C" allows this). They are declared on Sheet 03 in the following form:

```
0358 struct proc
     {
       char p_stat;
       ..........
       int p_wchan;
       ..........
     } proc[NPROC];
```

"p" is a register variable of type pointer to a structure of type "proc".

```
p = &proc[0];
```

assigns to "p" the address of the first element of the array "proc". The operator "&" in this context means "the address of".

Note that if an array has n elements, the elements have subscripts 0, 1, .., (n-1). Also it is permissible to write the above statement more simply as

```
p = proc;
```

There are two statements in between the "do" and the "while". The first of these could be rewritten more simply as

```
if (p->p wchan == c) setrun (p);
```

i.e. the brackets are superfluous in this case, and since "C" is a free form language, the arrangement of text between lines is not significant.
The statement

```
setrun (p);
```

invokes the procedure "setrun" passing the value of "p" as a parameter (All parameters are passed by value.). The relation

```
p->p_wchan == c
```

tests the equality of the value of "c" and the value of the element "p_wchan" of the structure pointed to by "p". Note that it would have been wrong to have written

```
p.p_wchan == c
```

because "p" is not the **name** of a structure.

The second statement, which cannot be combined with the first, increments "p" by the size of the "proc" structure, whatever that is. (The compiler can figure it out.)

In order to do this calculation correctly, the compiler needs to know the kind of structure pointed at. When this is not a consideration, you will notice that often in similar situations, "p" will be declared simply as

```
register *p;
```

because it was easier for the programmer, and the compiler does not insist.

The latter part of this procedure could have been written equivalently but less efficiently as

```
............
i = 0;
do
  if (proc[i].p_wchan == c)
    setrun (&proc[i]);
while (++i < NPROC);
```

3.7 Example 6

```
5336 geterror (abp)
     struct buf *abp;
     {
       register struct buf bp;
       bp = abp;
       if (bp->b flags & B_ERROR)
         if ((u.u_error=bp->b_error)==0)
           u.u_error = EIO;
     }
```

This procedure simply checks if there has been an error, and if the error indicator "u.u_error" has not been set, sets it to a general error indication

"B_ERROR" has the value 04 (see line 4575) so that, with only one bit set, it can be used as mask to isolate bit number 2. The operator "&" as used in

```
bp->b_flags & B_ERROR
```

is the bitwise logical conjunction ("and") applied to arithmetic values.

The above expression is greater than one if bit 2 of the element "b_flags" of the "buf" structure pointed to by "bp", is set.

Thus if there has been an error, the expression

```
(u.u_error) = bp->b_error)
```

is evaluated and compared with zero. Now this expression includes an assignment operator "=". The value of the expression is the value of "u.u_error" **after** the value of "bp->b_flags" has been assigned to it.

This use of an assignment as part of an expression is useful and quite common.

3.8 Example 7

```
3428 stime ()
     {
       if (suser()) {
         time[0] = u.u_ar0[R0];
         time[1] = u.u_ar0[R1];
         wakeup (tout);
       }
     }
```

In this example, you should note that the procedure "suser" returns a value which is used for the "if" test. The three statements whose execution depends on this value are enclosed in the brackets "{" and "}".

Note that a call on a procedure with no parameters must still be written-with a set of empty parentheses, sic.

```
suser ()
```

3.9 Example 8

"C" provides a conditional expression. Thus if "a" and "b" are integer variables,

```
(a > b ? a : b)
```

is an expression whose value is that of the larger of "a" and "b".

However this does not work if "a" and "b" are to be regarded as unsigned integers. Hence there is a use for the procedure

```
6326 max (a, b)
     char *a, *b;
     {
       if (a > b)
         return(a);
       return(b);
     }
```

The trick here is that "a" and "b", having been declared as pointers to characters are treated for comparison purposes as unsigned integers.

The body of the procedure could have been written as

```
max (a, b)
char *a, *b;
{
  if (a > b)
    return(a);
  else
    return(b);
}
```

but the nature of "return" is such that the "else" is not needed here!

3.10 Example 9

Here are two quickies which introduce some different and exotic looking expressions. First:

```
7679 schar()
     {
       return *u.u_dirp++ & 0377);
     }
```

where the declaration

```
    char *u_dirp;
```

is part of the declaration of the structure "u".

"u.u_dirp" is a character pointer. Therefore the value of "*u.u_dirp++" is a character. (Incrementation of the pointer occurs as a side effect.)

When a character is loaded into a sixteen bit register, sign extension may occur. By "and"ing the word with 0377 any extraneous high order bits are eliminated. Thus the result returned is simply a character.

Note that any integer which begins with a zero (e.g. 0377) is interpreted as an octal integer.

The second example is:

```
1771 nseg(n)
     {
       return ((n+127)>>7);
     }
```

The value returned is n divided by 128 and rounded up to the next highest "integer".

Note the use of the right shift operator ">>" in preference to the division operator "/".

3.11 Example 10

Many of the points which have been introduced above are collected in the following procedure:

```
2134 setrun (p)
     {
       register struct proc *rp;
       rp = p;
       rp->p_wchan = 0;
       rp->p_stat = SRUN;
       if (rp->p_pri < curpri)
         runrun++;
       if (runout != 0 &&
           (rp->p_flag & SLOAD) == 0) {
          runout = 0;
          wakeup (&runout);
       }
     }
```

Check your understanding of "C" by figuring out what this one does.

There are two additional features you may need to know about:

"&&" is the logical conjunction ("and") for relational expressions. (Cf. "||" introduced earlier.)

The last statement contains the expression

```
&runout
```

which is syntactically an address variable but semantically just a unique bit pattern.

This is an example of a device which is used throughout UNIX. The programmer needed a unique bit pattern for a particular purpose. The exact value did not matter as long as it was unique. An adequate solution to the problem was to use the address of a suitable global variable.

3.12 Example 11

```
4856 bawrite (bp)
     struct buf *bp;
     {
       register struct buf *rbp;
       rbp = bp;
       rbp->b_flags =| B_ASYNC;
       bwrite (rbp);
     }
```

The second last statement is interesting because it could have been written as

```
rbp->b_flags = rbp->b_flags | B_ASYNC;
```

In this statement the bit mask "B ASYNC" is "or"ed into "rbp->b_flags". The symbol "|" is the logical disjunction for arithmetic values.

This is an example of a very useful construction in UNIX, which can save the programmer much labour. If "O" is any binary operator, then

```
x = x O a;
```

where "a" is an expression, can be rewritten more succinctly as

```
x =O  a;
```

A programmer using this construction has to be careful about the placement of blank characters, since

```
x =+ 1;
```

is different from

```
x = +1;
```

What is to be the meaning of

```
x =+1;          ?
```

3.13 Example 12

```
6824 ufalloc ()
     {
       register i;
       for (i=0; i<NOFILE; i++)
         if (u.u_ofile[i]==NULL) {
           u.u_ar0[R0] = i;
           return (i);
         }
       u.u_error = EMFILE;
       return (-1);
     }
```

This example introduces the "for" statement, which has a very general syntax making it both powerful and compact.

The structure of the "for" statement is adequately described on page 10 of the "C Tutorial", and that description is not repeated here.

The Algol equivalent of the above "for" statement would be

```
for i:=1 step 1 until NOFILE-1 do
```

The power of the "for" statement in "C" derives from the great freedom the programmer has in choosing what to include between the parentheses. Certainly there is nothing which restricts the calculations to integers, as the next example will demonstrate.

3.14 Example 13

```
3949 signal (tp, sig)
     {
       register struct proc *p;
       for (p=proc;p<&proc[NPROC];p++)
         if (p->p ttyp == tp)
           psignal (p,sig);
     }
```

In this example of the "for" statement, the pointer variable "p" is stepped through each element of the array "proc" in turn.

Actually the original code had

```
for (p=&proc[0];p<&proc[NPROC];p++)
```

but it wouldn't fit on the line! As noted earlier, the use of "proc" as an alternative to the expression "&proc[0]" is acceptable in this context.

This kind of "for" statement is almost a cliche in UNIX so you had better learn to recognise it. Read it as

for p = each process in turn

Note that "proc[NPROC]" is the address of the (NPROC+1)-th element of the array (which does not of course exist) i.e. it is the first location beyond the end of the array.

At the risk of overkill we would point out again that whereas in the previous example

```
i++;
```

meant add one to the integer "i", here

```
p++;
```

means "skip p to point to the next structure".

3.15 Example 14

```
8870 lpwrite ()
     {
      register int c;
      while ((c=cpass()) >= 0)
        lpcanon(c);
     }
```

This is an example of the "while" statement, which should be compared with the "do ... while ..." construction encountered earlier. (Cf. the "while" and "repeat" statements of Pascal.)
The meaning of the procedure is

Keep calling "cpass" while the result is positive, and pass the result as a parameter to a call on lpcanon.

Note the redundant "int" in the declaration for "c". It isn't always omitted!

3.16 Example 15

The next example is abbreviated from the original:

```
5861 seek ()
     {
       int n[2];
       register *fp, t;
       fp = getf (u.u_ar0[R0]);
       ...........
       t = u.u_arg[1];
       switch (t) {

       case 1:
       case 4:
         n[0] =+ fp->f_offset[0];
         dpadd (n, fp->f_offset[1]);
         break;

       default:
         n[0] =+ fp->f_inode->i size0 & 0377;
         dpadd(n,fp->f_inode->i_size1);

       case 0:
       case 3:
         ;
```

```
    }
    ...........
}
```

Note the array declaration for the two word array "n", and the use of getf (which appeared in Example 4).

The "switch" statement makes a multiway branch depending on the value of the expression in parentheses. The individual parts have "case labels":

- If "t" is one or four, then one set of actions is in order.
- If "t" is zero or three, nothing is to be done at all.
- If "t" is anything else, then a set of actions labelled "default" is to be executed.

Note the use of "break" as an escape to the next statement after the end of the "switch" statement. Without the "break", the normal execution sequence would be followed within the "switch" statement.

Thus a "break" would normally be required at the end of the "default" actions. It has been omitted safely here because the only remaining cases actually have null actions associated with them.

The two non-trivial pairs of actions represent the addition of one 32 bit integer to another. The later versions of the "C" compiler will support "long" variables and make this sort of code much easier to write (and read).

Note also that in the expression

```
fp->f_inode->i_size0
```

there are two levels of indirection.

3.17 Example 16

```
6672 closei (ip, rw)
     int *ip;
     {
       register *rip;
       register dev, maj;
       rip = ip;
       dev = rip->i_addr[0];
       maj = rip->i_addr[0].d major;
       switch (rip->i_mode&IFMT) {

       case IFCHR:
         (*cdevsw[maj].d_close)(dev,rw);
         break;

       case IFBLK:
         (*bdevsw[maj].d_close)(dev,rw);
       }
       iput(rip);
     }
```

This example has a number of interesting features.

The declaration for "d_major" is

```
struct {
   char d_minor;
   char d_major;
}
```

so that the value assigned to "maj" is the hiqh order byte of the value assigned to "dev".

In this example, the "switch" statement has onlv two non-null cases, and no "default". The actions for the recognised cases, e.g.

```
(*bdevsw[maj].d_close)(dev,rw);
```

look formidable

First it should be noted that this is a procedure call, with parameters "dev" and "rw".

Second "bdevsw" (and "cdevsw") are arrays of structures, whose "d_close" element is a pointer to a function, i.e.

```
bdevsw[maj]
```

is the name of a structure, and

```
bdevsw[maj].d_close
```

is an element of that structure which happens to be a pointer to a function, so that

```
*bdevsw[maj].d_close
```

is the name of a function. The first pair of parentheses is "syntactical sugar" to put the compiler in the right frame of mind!

3.18 Example 17

We offer the following as a final example:

```
4043 psig ()
     {
       register n, p;
       .........
       switch (n) {

       case SIGQIT:
       case SIGINS:
       case SIGTRC:
       case SIGIOT:
       case SIGEMT:
       case SIGEPT:
       case SIGBUS:
       case SIGSEG:
```

```
    case SIGSYS:
        u.u arg[0] = n;
        if (core())
          n =+ 0200;
    }
    u.u_arg[0]=(u.u_ar0[R0]<<8) | n;
    exit ();
  }
```

Here the "switch" selects certain values for "n" for which the one set of actions should be carried out.

An alternative would have been to write a "monster" "if" statement such as

```
if (n==SIGQUIT || n==SIGINS || ...
           ... || n==SIGSYS)
```

but that would not have been either transparent or efficient.

Note the addition of an octal constant to "n" and the method of composing a 16 bit value from two eight bit values.

4 An Overview

The purpose of this chapter is to survey the source code as a whole i.e. to present the "wood" before the "trees"

Examination of the source code will reveal that it consists of some 44 distinct files, of which:

- two are in assembly language, and have names ending in "s";
- 28 are in the "C" language and have names ending in "c";
- 14 are in the "C" language, but are not intended for independent compilation, and have names ending in "h".

The files and their contents were arranged by the programmers presumably to suit their convenience and not for ours. In many ways the divisions between files is irrelevant to the present discussion and might well be abolished entirely.

As mentioned already in Chapter One, the files have been organised into five sections. As far as was possible, the sections were chosen to be of roughly equal size, to cluster files which are strongly associated and to separate files which are only weakly associated.

4.1 Variable Allocation

The PDP11 architecture allows efficient access to variables whose absolute address is known, or whose address relative to the stack pointer can be determined exactly at compile time.

There is no hardware support for multiple lexical levels for variable declarations such as are available in block structured languages such as Algol or Pascal. Thus "C" as implemented on the PDP11 supports only two lexical levels: global and local.

Global variables are allocated statically; local variables are allocated dynamically within the current stack area or in the general registers (r2, r3 and r4 are used in this way).

4.2 Global Variables

In UNIX with very few exceptions, the declarations for global variables have been all gathered into the set of "h" files. The exceptions are:

(a) the static variable "p" (2180) declared in "swtch" which is stored globally, but is accessible only from within the procedure "swtch" (Actually "p" is a very popular name for local variables in UNIX.);

(b) a number of variables such as "swbuf" (4721) which are referenced only by procedures within a single file, and are declared at the beginning of that file.

Global variables may be declared separately within each file in which they are referenced. It is then the job of the loader, which links the compiled versions of the program files together to match up the different declarations for the same variable.

4.3 The 'C' Preprocessor

If global declarations must be repeated in full in each file (as is required by Fortran, for instance) then the bulk of the program is increased, and modifying a declaration is at best a nuisance, and at worst, highly error-prone.

These difficulties are avoided in UNIX by use of the preprocessor facility of the "C" compiler. This allows declarations for most global variables to be recorded once only in one of the few "h" files.

Whenever the declaration for a particular global variable is required the appropriate "h" file can then be "included" in the file being compiled.

UNIX also uses the "h" files as vehicles for lists of standard definitions for many symbolic names which represent constants and adjustable parameters, and for declaration of some structure types.

For example, if the file bottle.c contains a procedure "glug" which global variable called "gin" which is declared in the file "box.h" then a statement:

```
#include "box.h"
```

must be inserted at the beginning of the file "bottle.c" When the file "bottle.c" is compiled, all declarations in "box.h" are compiled, and since they are found before the beginning of any procedure in "bottle.c" they are flagged as external in the relocatable module which is produced.

When all the object modules are linked together, a reference to "gin" will be found in every file for which the source included "box.h" All these references will be consistent and the loader will allocate a single space for "gin" and adjust all the references accordingly.

4.4 Section One

Section One contains many of the "h" files and the assembly language files.

It also contains a number of files concerned with system initialisation and process management.

4.5 The First Group of '.h' Files

param.h [Sheet 01] contains no variable declarations, but many definitions for operating system constants and parameters, and the declarations for three simple structures. The convention will be noted of using "upper case only" for defined constants.

systm.h [Sheet 02; Chapter 19] consists entirely of declarations (with definitions of the structures "callout" and "mount" as side-effects). Note that none of the variables is initialised explicitly, and hence all are initialised to zero.

The dimensions for the first three arrays are parameters defined in param.h. Hence any file which "includes" "systm.h" must have previously included "param.h".

seg.h [Sheet 03] contains a few definitions and one declaration, which are used for referencing the segmentation registers. This file could be absorbed into "param.h" and "systm.h" without any real loss.

proc.h [Sheet 03; Chapter 7] contains the important declaration for "proc" which is both a structure type and an array of such structures. Each element of the "proc" structure has a name which begins with "p_" and no other variable is so named. Similar conventions are used for naming the elements of the other structures.

The sets of values for the first two elements, "p_stat" and "p_flag" have individual names which are define.

user.h [Sheet 04; Chapter 7] contains the declaration for the very important "user" structure, plus a set of defined values for "u_error".

Only one instance of the "user" structure is ever accessible at one time. This is referenced under the name "u" and is in the low address part of a 1024 byte area known as the "per process data area".

In general the complete "h" files are not analysed in detail later in this text. It is expected that the reader will refer to them from time to time (with increasing familiarity and understanding).

4.6 Assembly Language Files

There are two files in assembly language which comprise about 10% of the source code. A reasonable acquaintance with these files is necessary.

low.s [Sheet 05, Chapter 9] contains information, including the trap vector, for initialising the low address part of main memory. This file is generated by a utility program called "mkconf" to suit the set of peripheral devices present at a particular installation.

m40.s [Sheets 06..14; Chapters 6, 8, 9, 10, 22] contains a set of routines appropriate to the PDP11/40, to carry out a variety of specialised functions which cannot be implemented directly in "C".

Sections of this file are introduced into the discussion as and where appropriate. (The largest of the assembler procedures, "backup" has been left to the reader to survey as an exercise.)

There is an alternative to "m40.s" which is not presented here, namely "m45.s" which is used on PDP11/45's and 70's.

4.7 Other Files in Section One

main.c [Sheets 15..17; Chapters 6, 7] contains "main" which performs various initialisation tasks to get UNIX running. It also contains "sureg" and "estabur" which set the user segmentation registers.

slp.c [Sheets 18..22; Chapters 6, 7, 8, 14] contains the major procedures required for process management including "newproc", "sched", "sleep" and "swtch".

prf.c [Sheets 23, 24; Chapter 5] contains "panic" and a number of other procedures which provide a simple mechanism for displaying initialisation messages and error messages to the operator.

malloc.c [Sheet 25; Chapter 5] contains "malloc" and "mfree" which are used to manage memory resources.

4.8 Section Two

Section Two is concerned with traps, hardware interrupts and software interrupts.

Traps and hardware interrupts introduce sudden switches into the CPU's normal instruction execution sequence. This provides a mechanism for handling special conditions which occur outside the CPU's immediate control.

Use is made of this facility as part of another mechanism called the "system call" whereby a user program may execute a "trap" instruction to cause a trap deliberately and so obtain the operating system's attention and assistance.

The software interrupt (or "signal" is a mechanism for communication between processes, particularly when there is "bad news".

reg.h [Sheet 26; Chapter 10] defines a set of constants which are used in referencing the previous user mode register values when they are stored in the kernel stack.

trap.c [Sheets 26..28; Chapter 12] contains the "C" procedure "trap" which recognises and handles traps of various kinds.

sysent.c [Sheet 29; Chapter 12] contains the declaration and initialisation of the array "sysent" which is used by "trap" to associate the appropriate kernel mode routine with each system call type.

sysl.c [Sheets 30..33; Chapters 12, 13] contains various routines associated with system calls, including "exec" "exit" "wait" and "fork".

sys4.c [Sheets 34..36; Chapters 12, 13, 19] contains routines for "unlink", "kill" and various other minor system calls.

clock.c [Sheets 37, 38; Chapter 11] contains "clock" which is the handler for clock interrupts, and which does much of the incidental housekeeping and basic accounting.

sig.c [Sheets 39..42; Chapter 13] contains the procedures which handle "signals" or "software interrupts" These provide facilities for interprocess communication and tracing.

4.9 Section Three

Section Three is concerned with basic input/output operations between the main memory and disk storage.

These operations are fundamental to the activities of program swapping and the creation and referencing of disk files.

This section also introduces procedures for the use and manipulation of the large (512 byte) buffers.

text.h [Sheet 43; Chapter 14] defines the "text" structure and array. One "text" structure is used to define the status of a shared text segment.

text.c [Sheets 43, 44; Chapter 14] contains the procedures which manage the shared text segments.

buf.h [Sheet 45; Chapter 15] defines the "buf" structure and array, the structure "devtab" and names for the values of "b_error" All these are needed for the management of the large (512 byte) buffers.

conf.h [Sheet 46; Chapter 15] defines the arrays of structures "bdevsw" and "cdevsw" which specify the device oriented procedures needed to carry out logical file operations.

conf.c [Sheet 46; Chapter 15] is generated, like "low.s" by the "mkconf" utility to suit the set of peripheral devices present at a particular installation. It contains the initialisation for the arrays "bdevsw" and "cdevsw" which control the basic i/o operations.

bio.c [Sheets 47..53; Chapters 15, 16, 17] is the largest file after "m40.s" It contains the procedures for manipulation of the large buffers, and for basic block oriented i/o.

rk.c [Sheets 53, 54; Chapter 16] is the device driver for the RK11/K05 disk controller.

4.10 Section Four

Section Four is concerned with files and file systems.

A file system is a set of files and associated tables and directories organised onto a single storage device such as a disk pack.

This section covers the means of creating and accessing files; locating files via directories organising and maintaining file systems. It also includes the code for an exotic breed of file called a "pipe".

file.h [Sheet 55; Chapter 18] defines the "file" structure and array.

filsys.h [Sheet 55; Chapter 20] defines the "filsys" structure which is copied to and from the "super block" on "mounted" file systems.

ino.h [Sheet 56] describes the structure of "inodes" as recorded on the "mounted" devices. Since this file is not "included" in any other, it really exists for information only.

inode.h [Sheet 56; Chapter 18] defines the "inode" structure and array. "inodes" are of fundamental importance in managing the accesses of processes to files.

sys2.c [Sheets 57..59; Chapters 18, 19] contains a set of routines associated with system calls including "read", "write", "creat", "open" and "close"

sys3.c [Sheets 60, 61; Chapters 19, 20] contains a set of routines associated with various minor system calls.

rdwri.c [Sheets 62, 63; Chapter 18] contains intermediate level routines involved with reading and writing files.

subr.c [Sheets 64, 65; Chapter 18] contains more intermediate level routines for i/o, especially "bmap" which translates logical file pointers into physical disk addresses.

fio.c [Sheets 66..6; Chapters 18, 19] contains intermediate level routines for file opening, closing and control of access.

alloc.c [Sheets 69..72; Chapter 20] contains procedures which manage the allocation of entries in the "inode" array and of blocks of disk storage.

iget.c [Sheets 72..74; Chapters 18, 19, 20] contains procedures concerned with referencing and updating "inodes".

nami.c [Sheets 75, 76; Chapter 19] contains the procedure "namei" which searches the file directories.

pipe.c [Sheets 77, 78; Chapter 21] is the "device driver" for "pipes" which are a special form of short disk file used to transmit information from one process to another.

4.11 Section Five

Section Five is the final section. It is concerned with input/output for the slower, character oriented peripheral devices.

Such devices share a common buffer pool, which is manipulated by a set of standard procedures.

The set of character peripheral devices are exemplified by the following:

KL/DL11	interactive terminal
PC11	paper tape reader/punch
LP11	line printer

tty.h [Sheet 79; Chapters 23, 24] defines the "clist" structure (used as a list head for character buffer queues), the "tty" structure (stores relevant data for controlling an individual terminal), declares the "partab" table (used to control transmission of individual characters to terminals) and defines names for many associated parameters.

kl.c [Sheet 80; Chapters 24, 25] is the device driver for terminals connected via KL11 or DL11 interfaces.

tty.c [Sheets 81..85; Chapters 23, 24, 25] contains common procedures which are independent of the attaching interfaces, for controlling transmission to or from terminals, and which take into account various terminal idiosyncrasies.

pc.c [Sheets 86,87; Chapter 22] is the device handler for the PC11 paper tape reader/punch controller.

lp.c [Sheets 88, 89; Chapter 22] is the device handler for the LP11 line printer controller.

mem.c [Sheet 90] contains procedures which provide access to main memory as though it were an ordinary file. This code has been left to the reader to survey as an exercise.

Section One

Section One contains many of the global declaration files and the assembly language files.

It also comtains a number of files concerned with system initialisation and process management.

5 Two Files

This chapter is intended to provide a gentle introduction to the source code by looking at two files in Section One which can be isolated reasonably well from the rest.

The discussion of these files supplements the discussion of Chapter Three and includes a number of additional comments regarding the syntax and semantics of the "C" language.

5.1 The File 'malloc.c'

This file is found on Sheet 25 of the Source code, and consists of just two procedures:

```
malloc (2528)   mfree (2556)
```

These are concerned with the allocation and subsequent release of two kinds of memory resources, namely:

main memory in units of 32 words (64 bytes);

disk swap area in units of 256 words (512 bytes).

For each of these two kinds of resource, a list of available areas is maintained within a resource "map" (either "coremap" or "swapmap"). A pointer to the appropriate resource "map" is always passed to "malloc" and "mfree" so that the routines themselves do not have to know the kind of resource with which they are dealing.

Each of "coremap" and "swapmap" is an array of structures of the type "map" as declared at line 2515. This structure consists of two character pointers i.e. two unsigned integers.

The declarations of "coremap" and "swapmap" are on lines 0203, 0204. Here the "map" structure is completely ignored – a regrettable programming short-cut which is possible because it is not detected by the loader. Thus the actual numbers of list elements in "coremap" and "swapmap" are "CMAPSIZ/2" and "SMAPSIZ/2" respectively.

5.2 Rules for List Maintenance

(a) Each available area is defined by its size and relative address (reckoned in the units appropriate to the resource);

(b) The elements of each list are arranged at all times in order of increasing relative address. Care is taken that no two list elements represent contiguous areas – the alternative course, to merge the two areas into a single larger area is always taken;

(c) The whole list can be scanned by looking at successive elements of the array, starting with the first, until an element with a zero size is encountered. This last element is a "sentinel" which is not part of the list proper.

The above rules provide a complete specification for "mfree", and a specification for "malloc" which is complete except in one respect: We need to specify how the resource allocation is actually made when there exists more than one way of performing it.

The method adopted in "malloc" is one known as "First Fit" for reasons which should become obvious.

As an illustration of how the resource "map" is maintained, suppose the following three resource areas were available:

- an area of size 15 beginning at location 47 and ending at location 61;
- an area of size 13 spanning addresses 27 to 39 inclusive;
- an area of size 7 beginning at location 65.

Then the "map" would contain:

Entry	Size	Address
0	13	27
1	15	47
2	7	65
3	0	??
4	??	??

If a request for a space of size 7 were received, the area would be allocated starting at location 27, and the "map" would become:

Entry	Size	Address
0	6	34
1	15	47
2	7	65
3	0	??
4	??	??

If the area spanning addresses 40 to 46 inclusive is returned to the available list, the "map" would become:

Entry	Size	Address
0	28	34
1	7	65
2	0	??
3	??	??

Note how the number of elements has actually decreased by one because of amalgamation though the total available resources have of course increased.

Let us now turn to a consideration of the actual source code.

5.3 malloc (2528)

The body of this procedure consists of a "for" loop to search the "map" array until either:

(a) the end of the list of available resources is encountered; or

(b) an area large enough to honour the current request is found;

2534: The "for" statement initialises "bp" to point to the first element of the resource map. At each succeeding iteration "bp" is incremented to point to the next "map" structure.

Note that the continuation condition "bp->m_size" is an expression, which becomes zero with the sentinel is referenced. This expression could have been written equivalently but more transparently as "bp->m_size>0".

Note also that no explicit test for the end of the array is made. (It can be shown that this latter is not necessary provided CMAPSIZ, SMAPSIZ $\geq$ 2 * NPROC !)

2535: If the list element defines an area at least as large as that requested, then ...

2536: Remember the address of the first unit of the area;

2537: Increment the address stored in the array element;

2538: Decrement the size stored in the element and compare the result with zero (i.e. was it an exact fit?);

2539: In the case of an exact fit, move all the remaining list elements (up to and including the sentinel) down one place.

Note that "(bp-l)" points to the structure before the one referenced by "bp";

2542: The "while" continuation condition does **not** test the equality of "(bp-l)->m_size" and bm->m_size !

The value tested is the value assigned to "(bp->m_size" copied from "bp->m_size".

(You are forgiven for not recognising this at once.);

2543: Return the address of the area. This represents the end of the procedure and hence very definitely the end of the "for" loop.

Note that a value of zero returned means "no luck" This is based on the assumption that no valid area can ever begin at location zero.

5.4 mfree (2556)

This procedure returns the area of size "size" at address "aa" to the "resource map" designated by "mp". The body of the procedure consists of a one line "for" statement, followed by a multiline "if" statement.

2564: The semicolon at the end of this line is extremely significant, terminating as it does the empty statement. (It would aid legibility if this character were moved to a line on its own, as is done on line 2394.)

Depending on your point of view, this statement demonstrates either the power or the obscurity of the "C" language. Try writing equivalent code to this statement in another language such as Pascal or PL/1.

Step "bp" through the list until an element is encountered either with an address greater than the address of the area being returned.

i.e. not "bp->m_addr $\leq$ a"

or which indicates the end of the list

i.e. not "bp->m_size != 0";

2565: We have now located the element in front of which we should insert the new list element. The question is: Will the list grow larger by one element or will amalgamation keep the number of elements the same or even reduce it by one?

If "bp > mp" we are not trying to insert at the beginning of the list. If (bp-l)->m_addr+(bp-l)->m_size==a

then the area being return abuts the previous element in the list;

2566: Increase the size of the previous list element by the size of the area being returned;

2567: Does the area being returned also abut the next element of the list? If so

2568: Add the size of the next element of the list to the size of the previous element;

2569: Move all the remaining list elements (up to the one containing the final zero size) down one place.

Note that if the test on line 2567 fortuitously gives a true result when "bp->m_size" is zero no harm is done;

2576: This statement is reached if the test on line 2565 failed i.e. the area being returned cannot be amalgamated with the previous element on the list.

Can it be amalgamated with the next element? Note the check that the next element is not null;

2579: Provided the area being returned is genuinely non-null (perhaps this test should have been made sooner?) add a new element to the list and push all the remaining elements up one place.

5.5 In conclusion ...

The code for these two procedures has been written very tightly. There is little, if any, "fat" which could be removed to improve run time efficiency. However it would be possible to write these procedures in a more transparent fashion.

If you feel strongly on this point, then as an exercise, you should rewrite "mfree" to make its function more easily discernible.

Note also that the correct functioning of "malloc" and "mfree" depends on correct initialisation of "coremap" and "swapmap". The code to do this occurs in the procedure "main" at lines 1568, 1583.

5.6 The File 'prf.c'

This file is found on Sheets 23 and 24, and contains the following procedures:

```
printf (2340)     panic (2416)
printn (2369)     prdev (2433)
putchar (2386)    deverror (2447)
```

The calling relationship between these procedures is illustrated below:

```
panic    deverror
    |       |
    |     prdev
    |       |
     \     /
     printf
       |
     printn
       |
     putchar
```

5.7 printf (2340)

The procedure "printf" provides a direct, unsophisticated low-level, unbuffered way for the operating system to send messages to the system console terminal. It is used during initialisation and to report hardware errors or the imminent collapse of the system.

(These versions of "printf" and "putchar" run in kernel mode and are similar to, but not the same as, the versions invoked by a "C" program which runs in user mode. The latter versions of "printf" and "putchar" live in the library "/lib/libc.a". You may still find it useful to read the sections "PRINTF(III)" and "PUTCHAR(III)" of the UPM at this point.)

2340: The programmer must have been carried away when he declared all the parameters for this procedure. In fact the procedure body only contains references to "xl" and "fmt".

This serves to reveal one of the facts of "C" programming. The rules for matching parameters in procedure calls and procedure declarations are not enforced, not even with respect to the numbers of parameters.

Parameters are placed on the stack in **reverse** order. Thus when "printf" is called "fmt" will be nearer to the "top of stack" than "xl", etc.

```
|   .   |
---------
|   .   |
---------
|   .   |        stack grows down
---------
|   .   |
---------
|  x2   |
---------
|  xl   |
---------
|  fmt  |
---------
|   .   |        top of stack
---------
```

"xl" has a higher address then "fmt" but a lower address then "x2", because stacks grow downwards on the PDP11.

2341: "fmt" may be interpreted as a constant character pointer. This declaration is (almost) equivalent to

"char *fmt;"

The difference is that here the value of "fmt" cannot be changed;

2346: "adx" is set to point to "xl". The expression "&xl" is the address of "xl". Note that since "xl" is a stack location, this expression cannot be evaluated at compile time.

(Many of the expressions you will find elsewhere involving the addresses of variables or arrays are effective because they **can** be evaluated at compile or load time.);

2348: Extract into the register "c" successive characters from the format string;

2349: If "c" is not a '%' then ...

2350: If "c" is a null character ('\0'), this indicates the end of the format string in the normal way, and "printf" terminates;

2351: Otherwise call "putchar" to send the character to the system console terminal;

2353: A '%' character has been seen. Get the next character (it had better not be the '\0'!);

2354: If this character is a 'd' or 'l' or 'o', call "printn" passing as parameters the value referenced by "adx" and either the value "8" or "10" depending on whether "c" is 'o' or not. (The 'd' and 'l' codes are clearly equivalent.)

"printn" expresses the binary numbers as a set of digit characters according to the radix supplied as the second parameter;

2356: If the editing character is 's', then all but the last character of a null terminated string is to be sent to the terminal. "adx" should point to a character pointer in this case;

2361: Increment "adx" to point to the next word in the stack i.e. to the next parameter passed to "printf";

2362: Go back to line 2347 and continue scanning the format string. Enthusiasts for structured programming will prefer to replace lines 2347 and this by "while (1) {" and "}" respectively .

5.8 printn (2369)

This procedure calls itself recursively in order to generate the required digits in the required order. It might be possible to code this procedure more efficiently but not more completely. (Anyway, in view of the implementation of "putchar", efficiency is hardly a consideration here.)

Suppose n = A*b + B where A = ldiv(n,b) and where B = lrem(n,b) satisfies $0 \leq B < b$. Then in order to display the value for n, we need to display the value for A followed by the value for B.

The latter is easy for b = 8 or 10: it consists of a single character. The former is easy if A = 0. It is also easy if "printn" is called recursively. Since A < n, the chain of recursive calls must terminate.

2375: Arithmetic values corresponding to digits are conveniently converted to their corresponding character representations by the addition of the character '0'.

The procedures "ldiv" and "lrem" treat their first parameter as an unsigned integer (i.e. no sign extension, when a 16 bit value is extended to a 32 bit value before the actual division operation). They may be found beginning on lines 1392 and 1400 respectively.

5.9 putchar (2386)

This procedure transmits to the system console the character which was passed as a parameter.

It illustrates in a small way the basic features of i/o operations on the PDP11 computer.

2391: "SW" is defined on line 0166 as the value "0177570". This is the kernel address of a read only processor register which stores the setting of the console switch register.

The meaning of the statement is clear: get the contents at location 0177570 and see if they are zero. The problem is to express this in "C". The code

if (SW == 0)

would not have conveyed this meaning. Clearly "SW" is a pointer value which should be dereferenced. The compiler might have been changed to accept

if (SW-> == 0)

but as it stands, this is syntactically incorrect. By inventing a dummy structure, with an element "integ" (see line 0175), the programmer has found a satisfactory solution to his problem.

Several other examples of this programming device will be found in this procedure and elsewhere.

In hardware terms, the system console terminal interface consists of four 16 bit control registers which are given consecutive addresses on the Unibus beginning at kernel address 0177560 (see the declaration for "KL" on line 0165.) For a description of the formats and usage of these registers, see Chapter Twenty-Four of the "PDP11 Peripherals Handbook".

In software terms, this interface is the unnamed structure which is defined beginning on line 2313, with four elements which name the four control registers. It does not matter that the structure is unnamed because it is not necessary to allocate any instances of it (the one we are interested in is essentially predefined, at the address given by "KL").

2393: While bit 7 of the transmitter status register ("XST") is off, keep doing nothing, because the interface is not ready to accept another character.

This is a classic case of "busy waiting" where the processor is allowed to cycle uselessly through a set of instructions until some externally defined event occurs. Such waste of processing power cannot normally be tolerated but this procedure is only used in unusual situations.

2395: The need for this statement is tied up with the statement on line 2405;

2397: Save the current contents of the transmitter status register;

2398: Clear the transmitter status register preparatory to sending the next character:

2399: With bit 7 of the control status register reset, move the next character to be transmitted to the transmitter buffer register. This initiates the next output operation;

2400: A "new line" character needs to be accompanied by a "carriage return" character and this is accomplished by a recursive call on "putchar".

A couple of extra "delete" characters are thrown in also to allow for any delays in completing the carriage return operation at the terminal;

2405: This call on "putchar" with an argument of zero effectively results in a re-execution of lines 2391 to 2394.

(It is very hard to see why the programmer chose to use a recursive call here in preference to simply repeating lines 2393 and 2394, since both code efficiency and compactness not to mention clarity seem to have suffered.);

2406: Restore the contents of the transmitter status register. In particular if bit 6 was formerly set to enable interrupts then this resets it.

5.10 panic (2419)

This procedure is called from a number of locations in the operating system. (e.g. line 1605). When circumstances exist under which continued operation of the system seems undesirable.

UNIX does not profess to be a "fault tolerant" or "fail soft" system, and in many cases the call on "panic" can be interpreted as a fairly unsophisticated response to a straightforward problem.

However more complicated responses require additional code, lots of it, and this is contrary to the general UNIX philosophy of "keep it simple".

2419: The reason for this statement is given in the comment beginning at line 2323;

2420: "update" causes all the large block buffers to be written out. See Chapter Twenty;

2421: "printf" is called with a format string and one parameter, which was passed to "panic";

2422: This "for" statement defines an infinite loop in which the only action is a call on the assembly language procedure "idle" (1284).

"idle" drops the processor priority to zero, and performs a "wait". This is a "do nothing" instruction of indefinite duration. It terminates when a hardware interrupt occurs.

An infinite set of calls on "idle" is better than the execution of a "halt" instruction, since any i/o activities which were under way can be allowed to complete and the system clock can keep ticking.

The only way for the operator to recover from a "panic" is to reinitialise the system, (after taking a core dump, if desired).

5.11 prdev (2433), deverror (2447)

These procedures provide warning messages when errors are occurring in i/o operations. At this stage, their only interest is as examples of the use of "printf".

5.12 Included Files

It will be noted that whereas the file "malloc.c" contains no request to include other files, requests to include four separate files are included at the beginning of "prf.c".

(The observant reader will note that these files are presumed to reside one level higher in the file hierarchy than "prf.c" itself.)

The statement on line 2304 is to be understood as if it were replaced by the entire contents of the file "param.h". This then supplies definitions for the identifiers "SW", "KL" and "integ" which occur in "putchar".

We noted earlier that declarations for "KL", "SW" and "integ" occurred on lines 0165, 0166 and 0175 respectively, but this would have been meaningless, if the file "param.h" had not been "included" in "prf.c".

The files "buf.h" and "conf.h" have been included to provide declarations for "d_major", "d_minor", "b_dev" and "b_blkno", which are used in "prdev" and "deverror".

The reason for the inclusion of the fourth file, "seg.h", is a little harder to find. In fact it is not necessary as the code stands, and the author owes his readers an apology. In editing the source code, it seemed like a good idea to move the declaration for "integ" from "seg.h" to "param.h". Q.E.D.

Note that the variable "panicstr" (2328) is also global but since it is not referenced outside "prf.c", its declaration has not been placed in any ".h" file.

6 Getting Started

This chapter considers the sequence of events which occur when UNIX is "rebooted" i.e. it is loaded and initiated in an idle machine.

A study of the initialisation process is of interest in itself, but more importantly, it allows a number of important features of the system to be presented in an orderly manner.

The operating system may have to be restarted in the aftermath of a system crash. It will also have to be restarted frequently for quite ordinary, operational reasons, e.g. after an overnight shutdown. If we assume the latter case, then we can assume that all the disk files are intact and that no special circumstance needs to be recognised or dealt with.

In particular, we can assume there is a file in the root directory called "/unix", which is the object code for the operating system.

This file began life as a set of source files such as we are investigating. These were compiled and linked together in the normal way to form a single object program file, and stored in the root directory.

6.1 Operator Actions

Reinitialisation requires operator action at the processor console. The operator must:

- stop the processor by setting the "enable/halt" switch to "halt";
- set the switch register with the address of the hardware bootstrap loader program;
- depress and release the " load address" switch;
- move the "enable/halt" switch to "enable";
- depress and release the "start" switch.

This activates the bootstrap program which is permanently recorded in a ROM in the processor.

The bootstrap loader program loads a larger loader program (from block #0 of the system disk), which looks for and loads a file called "/unix" into the low part of memory.

It then transfers control to the instruction loaded at address zero.

Address zero is occupied by a branch instruction (line 0508), which branches to location 000040, which contains a jump instruction (line 0522), which jumps to the instruction labelled "start" in the file "m40.s" (line 0612).

6.2 start (0612)

0613: The "enabled" bit of the memory management status register, SR0, is tested. If this set, the processor will dwell forever in a two instruction loop. This register will normally be cleared when the operator activates the "clear" button on the console before starting the system.

A number of reasons have been suggested for the necessity for this loop. The most likely is that in the case of a double bus timeout error, the processor will branch to location zero, and in this situation it should not be allowed to go further.

0615: "reset" clears and initialises all the peripheral device control and status registers

The system will now be running in kernel mode with memory management disabled.

0619: KISA0 and KISD0 are the high core addresses of the first pair of kernel mode segmentation registers. The first six kernel descriptor registers are initialised to 077406, which is the description of a full size, 4K word, read/write segment.

The first six kernel address registers are initialised to 0, 0200, 0400, 0600, 01000 and 01200 respectively.

As a result the first six kernel segments are initialised (without any reference to the actual size of UNIX) to point to the first six 4K word segments of physical memory. Thus the "kernel to physical address" translation is trivial for kernel addresses in the range 0 to 0137777;

0632: "_end" is a loader pseudo variable which defines the extent of the program code and data area. This value is rounded up to the next multiple of 64 bytes and is stored in the address register for the seventh segment (segment #6).

Note that the address of this register is stored in "ka6", so that the content of this register is accessible as "*ka6";

0634: The corresponding descriptor register is loaded with a value which (since "USIZE" is equal to 16) is the description of a read/write segment which is 16 x 32 = 512 words long.

The value 007406 is obtained by shifting the octal value 017 eight places to the left and then "or"ing in the value 6;

0641: The eighth segment is mapped into the highest 4K word segment of the physical address space.

It should be noted that with memory management disabled, the same translation is already in force i.e. addresses in the highest 4K word segment of the 32K program address space are automatically mapped into the highest 4K word segment of the physical address space.

We may note that from this point on, all the kernel mode segmentation registers will remain unchanged with the single exception of the **seventh kernel segmentation address register**.

This register is explicitly manipulated by UNIX to point to a variety of locations in physical memory. Each such location is the beginning of an area 512 words long, known as a "per process data area".

The seventh kernel address register is now set to point to the segment which will become the per process data area for process #0.

0646: The stack pointer is set to point to the highest word of the per process data area;

0647: By incrementing the value of SR0 from zero to one, the "memory management enabled" bit is conveniently set.

From this point, all program addresses are translated to physical addresses the memory management hardware.

0649: "bss" refers to the second part of the program data area, which is not initialised by the loader (see "A.OUT(V)" in the PM). The lower and upper limits of this area are defined by the loader pseudo variables, "_edata" and "_end" respectively;

0668: The processor status word (PS) is changed to indicate that the "previous mode" was "user mode".

This prepares the way for the investigation and initialisation of the areas of physical memory which are not part of the kernel address space. (This involves use of the special instructions "mtpi" and "mfpi" (Move To/From Previous Instruction space) together with some manipulation of the user mode segmentation registers.);

0669: A call is then made to the procedure "main" (1550).

It will be seen later that "main" calls "sched" which never terminates. The need for or use of the last three instructions of "start" (lines 0670, 0671 and 0672) is therefore somewhat enigmatic. The reason will come later. In the meantime you might like to ponder "why?". What do these lines do anyway?

6.3 main (1550)

Upon entry to this procedure:

(a) the processor is running at priority zero, in kernel mode and with the previous mode shown as user mode;

(b) the kernel mode segmentation registers have been set and the memory management unit has been enabled;

(c) all the data areas used by the operating system have been initialised;

(d) the stack pointer (SP or r6) points to a word which contains a return address in "start".

1559: The first action of "main" would appear to be redundant, since "updlock" should have already been set to zero as part of the initialisation performed by "start";

1560: "i" is initialised to the ordinal of the first 32 word block beyond the "per process data area" for process #0;

1562: The first pair of user mode segmentation registers are used to provide a "moving window" into higher areas of the physical memory.

At each position of the window an attempt is made (using "fuibyte") to read the first accessible word in the window. If this is not successful, it is assumed that the end of the physical memory has been reached. Otherwise the next 32 word block is initialised to zero (using "clearseg" (0676)) and added to the list of available memory, and the window is advanced by 32 words.

"fuibyte" and "clearseg" are both to be found in "m40.s", "fuibyte" will normally return a positive value in the range 0 to 255. However, in the exceptional case where the memory location referenced does not respond, the value –1 is returned. The way this is brought about is a little obscure, and will be explained later in Chapter Ten.)

1582: "maxmem" defines the maximum amount of main memory which may be used by a user program. This is the minimum of:

- the physically available memory ("maxmem");
- an installation definable parameter ("MAXMEM") (0135);
- the ultimate limit imposed by the PDP11 architecture;

1583: "swapmap" defines available space on the swapping disk which may be used when user programs are swapped out of main memory. It is initialised to a single area of size "nswap", starting at relative address "swplo". Note that "nswap" and "swplo" are initialised in "conf.c" (lines 4697, 4698);

1589: The significance of this and the next four lines will be discussed shortly;

1599: The design of UNIX assumes the existence of a system clock which interrupts the processor at line frequency (i.e 50 Hz or 60 Hz).

There are two possible clock types available: a line frequency clock (KW11-L) which has a control register on the Unibus at address 777546, or a programmable, real-time clock (KW11-P) located at address 777540 (lines 1509, 1510).

UNIX does not presume which clock will be present. It attempts to read the status word for the line frequency clock first. If successful, that clock is initialised and the other (if present) remains unused. If the first attempt is unsuccessful, then the other clock is tried. If both attempts are unsuccessful, there is a call on "panic" which effectively halts the system with an error message to the operator.

Since the absence of a clock will be indicated by a bus timeout error, it is convenient to make the reference via "fuiword", preceded by the setting of a user mode segmentation register pair (1599, 1600).

1607: Either type of clock is initialised by the statement

*lks = 0115;

As a consequence of this action, the clock will interrupt the processor within the next 20 milliseconds. This interrupt may occur at any time, but it will be convenient for this discussion to assume that no interrupt will occur before initialisation is complete;

1613: "cinit" (8214) initialises the pool of character buffers. See Chapter 23;

1614: "binit" (5055) initialises the pool of large buffers. See Chapter 17;

1615: "iinit" (6922) initialises table entries for the root device. See Chapter Twenty.

6.4 Processes

"process" is a term which has occurred more than once already. A definition which will suit our purposes reasonably well at present is simply "a program in execution".

Details of the representation of processes in UNIX will be discussed in the next chapter. For now we just note that each process involves a "proc" structure from the array called "proc" and a "per process data area" which includes one copy of the structure "u".

6.5 Initialisation of proc[0]

The explicit initialisation of the structure "proc[0]" is performed starting at line 1589. Only four elements are changed from the overall initial value of zero:

(a) "p_stat" is set to "SRUN" which implies that process 0 is "ready to run";

(b) "p_flag" is set to show both "SLOAD" and "SSYS". The former implies that the process is to be found in core (it has not been swapped out onto the disk), and the second, that it should never be swapped out;

(c) "p_size" is set to "USIZE";

(d) "p_addr" is set to the contents of the kernel segmentation address register #6.

It will be seen that process #0 has acquired an area of "USIZE" blocks (exactly the size of a "per process data area") which begins immediately after the official end ("_end") of the operating system data area.

The ordinal number of the first block of this area has been stored for future reference in "p_addr". This area, which was cleared to zero in "start" (0661), contains a single copy of the user structure called "u".

On line 1593, the address of "proc[0]" is stored in "u.u_procp", i.e. the "proc" structure and the "u" structure are mutually linked.

6.6 The story continues ...

1627: "newproc" (1826) will be discussed in detail in the next chapter.

In brief this initialises a second "proc" structure viz. "proc[1]", and allocates a second "per process data area" in core. This is a copy of the "per process data area" for process #0, exact in all but one respect: the value of "u.u_procp" in the second area is "&proc[1]".

We should note here that at line 1889, there is a call on "savu" (0725) which saves the current values of the environment and the stack pointers in "u.u_rsav" before the copy is made.

Also from line 1918 we can see that the value returned by "newproc" will be zero, so that the statements on lines 1628 to 1635 will not be executed;

1637: A call is made to "sched" (1940) which, it may be observed, contains an infinite loop, so that it never returns!

6.7 sched (1940)

At this stage we are only interested in what happens when "sched" is entered for the first time.

1958: "spl6" is an assembler routine (1292) which sets the processor priority level to six. (Cf. also "spl0", "spl4", "spl5" and "spl7" in "m40.s").

When the processor is at level six, only devices with priority seven can interrupt it. The clock whose priority level is six is thus inhibited from interrupting the processor between this point and the subsequent call on "spl0" at line 1976.

1960: A search is made through "proc" whose status is "SRUN" and which is not "loaded".

(Processes #0 and 1 have status "SRUN" and are loaded. All remaining 2193: processes, have a status of zero, which is equivalent to "undefined" or "NULL") .

1966: The search fails ("n" is still –1). The flag "runout" is made non-zero, indicating that there are no processes which are both ready to run and "swapped out" onto disk;

1968: "sleep" is called (to wait for such an event) with a priority "PSWP" (== –100) for when it wakes up, which is in the category of "very urgent".

6.8 sleep (2066)

2070: "PS" is the address of the processor status word. The processor status is stored in the register "s" (0164, 0175);

2071: "rp" is set to the address of the entry in the array "proc" of the current process (still "proc[0]" at this stage!);

2072: "pri" is negative, so the "else" branch is taken, setting the status of the current process (0) to "SSLEEP". The reason for "going to sleep" and the "awakening priority" are noted.

2093: "swtch" is then called.

6.9 swtch (2178)

2184: "p" is a static variable (2180), which means that its value is initialised to zero (1566) and is preserved between calls. For the very first call on "swtch", "p" is set to point to "proc[0]";

2189: "savu" is called to save the stack pointer and the environment pointer for the current process in "u.u_rsav";

2193: "retu" is called to reset the kernel address register for segment #6 to the value passed as an argument (this causes a change in the current process!), and to reset the stack and environment pointers to values appropriate to the revised current process, whose execution is about to be resumed.

The combination of successive calls on "savu" and "retu" at this point constitutes a so-called "coroutine jump" (Cf. "exchange jump" on the Cyber or "Load PSW" on the /360 or "Move Stack" on the B6700).

This time however the coroutine jump is from process 0 to process 0 (not very interesting!).

2201: The set of processes is searched to find the process whose state is "SRUN" and which is loaded and for which "p_pri" is a maximum.

The search is successful and process #1 is found. (N.B. The state of process #0 was just changed from "SRUN" to "SSLEEP" in "sleep" so it no longer satisfies the search criterion);

2218: Since "p" is not "NULL", the idle loop is not entered;

2228: "retu" (0740) causes a coroutine jump to process #1 which becomes the current process.

What is process #1 ? It is a copy of process #0, made at a previous stage of the latter's existence.

This call on "retu" was not preceded by a call on "savu" because the necessary information has in fact been saved already. (Where?)

2229: "sureg" is a routine 1738) which copies into the user mode segmentation registers, the values appropriate for the current process. These have been stored earlier in the arrays "u.u_uisa" and "u.u _uisd".

The very first call on "sureg" copies zeroes and serves no real purpose.

2240: The "SSWAP" flag is not set, so that this enigmatic (2239) section can be ignored for now;

2247: Finally "swtch" returns with a value of "1". But where does the "return" return to? **Not to "sleep" !**

The "return" follows values set by the stack pointer and the environment pointer. These (just before the return) have values equal to those in force when the most recent "savu(u.u_rsav)" was performed.

Now process #1, which is only just starting has never performed a "savu", but values were stored in "u.u_rsav" before the copy of process #0 was made by "newproc", which had been called from "main".

Thus in this case, *the return from "swtch" is made to "main", with a value of one.* (Look over this again, to be sure you understand!)

6.10 main revisited

The story so far: process #0, having created a copy of itself in the form of process #1, has gone to sleep. As a result process #1 has become the current process and has returned to "main "with a value of one. Now read on ...

1627: The statements in "main" which are conditional on "newproc" are now executed:

1628: "expand" (2268) finds a new, larger area (from USIZE*32 to (USIZE+1) *32 words) for process #1, and copies the original data area into it.

In this case, the original user data area consists only of a "per process data area", with zero length data and stack areas. The original area is released;

1629: "estabur" is used to set the "prototype" segmentation registers which are stored in "u.u_uisa" and "u.u_uisd" for later use by "sureg". "estabur" calls "sureg" as its last action.

The parameters for "estabur" are the sizes of the text, data and stack areas plus an indicator to decide whether the text and data areas should be in separate address spaces. (Never true on the PDP11/40.) The sizes are all in units of 32 words;

1630: "copyout" (1252) is an assembler routine which copies an array in kernel space of specified size into a region in user space. Here the array "icode" is copied into an area starting at location zero in user space;

1635: The "return" is not special. From "main" it goes to "start" (0670) where the three last instructions have the effect of causing *execution in user mode of the instruction at user mode address zero.* i.e. the execution of a copy of the first instruction in "icode". The instructions subsequently executed are copies also of instructions in "icode".

AT THIS POINT, THE INITIALISATION OF THE SYSTEM IS COMPLETE.

Process #1 is running and to all intents and purposes, is a normal process. Its initial form is (almost) that which would come from compilation, loading and execution of the simple, but non-trivial "C" program:

```
char *init "/etc/init";
main ( ) {
execl (init, init, 0);
while (1);
}
```

The equivalent assembler program is

```
        sys exec
        init
        initp
        br   .
initp:  init
        0
init:   </etc/init\0>
```

If the system call on "exec" fails (e.g. the file "/etc/init" cannot be found) the process falls into a tight loop, and there the processor will stay, except when the occasional clock interrupt occurs.

A description of the functions performed by "/etc/init" can be found in the section "INIT (VIII)" of the UPM.

7 Processes

The previous chapter traced the developments which occur after "the operating system has been re-booted", and in so doing introduced a number of significant features of the process concept. One of the aims of this chapter is to go back and re-explore some of the same ground more thoroughly.

There are a number of serious difficulties in providing a generally acceptable definition of "process". These are akin to the difficulties faced by the philosopher who would answer "what is life?" We will be in good company if we brush the more subtle points lightly aside.

The definition for "process" already given, "a program in execution", does reasonably well in suggesting what is intended. However it does not fit the case of either process #0 throughout its life or process #1 during its first moments. All other processes in the system however are clearly associated with the execution of some program file or other.

Processes can be introduced into discussions of operating systems at two levels.

At the upper level, "process" is an important organising concept for describing the activity of a computer system as a whole. It is often expedient to view the latter as the combined activity of a number of processes, each associated with a particular program such as the "shell", or the "editor". A discussion of UNIX at this level is given in the second half of Ritchie's and Thompson's paper, "The UNIX Time-sharing System".

At this level the processes themselves are considered to be the active entities in the system, while the identities of the true active elements, the processor and the peripheral devices, are submerged: the processes are born, live and die; they exist in varying numbers; they may acquire and release resources; they may interact, cooperate, conflict, share resources; etc.

At the lower level, "processes" are inactive entities which are acted on by active entities such as the processor. By allowing the processor to switch frequently from the execution of one process image to another, the impression can be created that each of the process images is developing continuously and this leads to the upper level interpretation.

Our present concern is with the low level interpretation: with the structure of the process image, with the details of execution and with the means for switching the processor between processes.

The following observations may be made about processes in the UNIX context:

(a) the existence of a process is implied by the existence of a non-null structure in the "proc" array, i.e. a "proc" structure for which the element "p_stat" is non-null;

(b) for each process there is a "per process data area" containing a copy of the "user" structure;

(c) the processor spends its entire life executing one process or another (except when it is resting between instructions);

(d) it is possible for one process to create or destroy another process;

(e) a process may acquire and possess resources of various kinds.

7.1 The Process

Ritchie and Thompson in their paper define a "process" as the execution of an "image", where the "image" is the current state of a pseudo-computer, i.e. an abstract data structure, which may be represented in either main memory or on disk.

The process image involves two or three physically distinct areas of memory:

(1) the "proc" structure, which is contained within the core resident "proc" array and is accessible at all times;

(2) the data segment, which consists of the "per process data area", combined with a segment containing the user program data, (possibly) program text, and stack;

(3) the text segment, which is not always present, consists of a segment containing only pure program text i.e. re-entrant code and constant data.

Many programs do not have a separate text segment. Where one is defined, a single copy will be shared among all processes which are executions of the same particular program.

7.2 The proc Structure (0358)

This structure, which is permanently resident in main memory, contains fifteen elements, of which eight are characters, six are integers, and one a pointer to an integer. Each element represents information that must be accessible at any time, especially when the main part of the process image has been swapped out to disk:

- "p_stat" may take one of seven values which define seven mutually exclusive states. See lines 0381 to 0387;
- "p_flag" is an amalgam of six one bit flags which may be set independently. See lines 0391 to 0396;

- "p_addr" is the address of the data segment:
  - If the data segment is in main memory this is a block number;
  - otherwise, if the data segment has been swapped out, this is a disk record number;
- "p_size" is the size of the data segment, measured in blocks;
- "p_pri" is the current process priority. This may be recalculated from time to time as a function of "p_nice", "p_cpu" and "p_time";
- "p_pid", "p_ppid" are numbers which uniquely identify a process and its parent;
- "p_sig", "p_uid", "p_ttyp" are involved with external communication i.e. with messages or "signals" from outside the process's normal domain;
- "p_wchan" identifies, for a "sleeping" process ("p_stat" equals either "SSLEEP" or "SWAIT"), the reason for sleeping;
- "p_textp" is either null or a pointer to an entry in the "text" array (4306), which contains vital statistics regarding the text segment.

7.3 The user Structure (0413)

One copy of the "user" structure is an essential ingredient of each "per process data area". At any one time there is exactly one copy of the "user" structure which is accessible. This goes under the name "u" and is always to be found at kernel address 0140000 i.e. at the beginning of the seventh page of the kernel address space.

The "user" structure has more elements than can be conveniently or usefully introduced here. The comment accompanying each declaration on Sheet 04 succinctly suggests the function of each element.

For the moment you should notice:

(a) "u_rsav", "u_qsav" , "u_ssav" which are two word arrays used to store values for r5, r6;

(b) "u_procp" which gives the address of the corresponding "proc" structure in the "proc" array;

(c) "u_uisa[16]", "u_uisd[16]" which store prototypes for the page address and description registers;

(d) "u_tsize", "u_dsize", "u_ssize" which are the size of the text segment and two parameters defining the size of the data segment, measured in 32 word blocks.

The remaining elements are concerned with:

- saving floating point registers (not for the PDP11/40);
- user identification;
- parameters for input/output operations;
- file access control;
- system call parameters;
- accounting information.

7.4 The Per Process Data Area

The "per process data area" corresponds to the valid part (lower part) of the seventh page of the kernel address space. It is 1024 bytes long. The lower 289 bytes are occupied by an instance of the "user" structure, leaving 367 words to be used as a kernel mode stack area. (Obviously there will be as many kernel mode stacks as there are processes.)

While the processor is in kernel mode, the values of r5 and r6, the environment and stack pointers, should remain within the range 0140441 to 01437777. Transition beyond the upper limit would be trapped as a segmentation violation, but the lower limit is protected only by the integrity of the software. (It may be noted that the hardware stack limit option is not used by UNIX.)

7.5 The Segments

The data segment is allocated as one single area of physical memory but consists of three distinct parts:

(a) a "per process data area";

(b) a data area for the user program. This may be further divided into areas for program text, initialised data and uninitialised data;

(c) a stack for the user program.

The size of (a) is always "USIZE" blocks. The sizes of (b) and (c) are given in blocks by "u.u_dsize" and "u.u_ssize". (It may be noted in passing that the latter two may change during the life of a process.)

A separate text segment containing only pure text is allocated as one single area of physical memory. The internal structure of the segment is not important here.

7.6 Execution of an Image

The image currently being executed (and hence the identity of the current process) is determined by the setting of the seventh kernel segmentation address register. If process #i is the current process, then the register has the value "proc[i].p_addr".

It is often desirable to distinguish between a process being executed in kernel mode and the same one being executed in user mode. We will use the terms "kernel process #i" and "user process #i" to denote "process #i executing in kernel mode" and "process #i executing in user mode" respectively.

If we chose to associate processes with particular execution stacks rather than with an entry in the "proc" array, then we would consider kernel process #i and user process #i to be separate processes, rather than different aspects of a single process #i.

7.7 Kernel Mode Execution

The seventh kernel segmentation address register must be set appropriately. None of the other kernel segmentation registers is ever disturbed and so their values are assumed. As was seen earlier, the first six kernel pages are mapped to the first six pages of physical memory, while the eighth is mapped into the highest page of physical memory. The size of the seventh segment is always the same.

In kernel mode the setting of the user mode segmentation registers is in general irrelevant. However they are normally set correctly for the user process.

The environment and stack pointers point into the kernel stack area in the seventh page, above the "user" structure.

7.8 User Mode Execution

Each activation of a user process is preceded and succeeded by an activation of the corresponding kernel process. Accordingly both the user mode and kernel mode registers will be properly set whenever a process image is being executed in user mode.

The environment and stack pointers point into the user stack area. This begins as the upper part of the eighth user page, but may be extended downwards, e.g. to occupy the whole of the eighth page and part or all of the seventh page, etc.

Whereas the setting of the kernel segmentation registers is fairly trivial, setting the user segmentation registers is much less so.

7.9 An Example

Consider a program on the PDP11/40 which uses 1.7 pages of text, 3.3 pages of data, and 0.7 pages of stack area. (Our use of fractions in this example is admittedly a little crude.) The set of virtual addresses would be divided as shown in the following diagram:

888 s1	Stack
888 s1	area
888	
777	
777	
777	
666	
666	
666 d4	
555 d3	
555 d3	
555 d3	
444 d2	Data
444 d2	
444 d2	area
333 d1	
333 d1	
333 d1	
222	
222 t2	
222 t2	Text
111 t1	
111 t1	area
111 t1	

Virtual Address Space

Two whole pages in the virtual address space must be allocated to the text segment, even though the physical area required is only 1.7 pages.

222 t2	
222 t2	Text
111 t1	
111 t1	area
111 t1	

Text Segment

The data and stack areas require the dedication of four and one pages of virtual address space, and 3.3 and 0.7 pages of physical memory respectively.

The whole data segment requires four and one eighth pages of physical memory. The extra eighth is for the "per process data area" which corresponds (from time to time) to the seventh kernel address page.

888 s1	Stack
888 s1	area
666 d4	
555 d3	
555 d3	
555 d3	
444 d2	Data
444 d2	
444 d2	area
333 d1	
333 d1	
333 d1	
ppda	

Data Segment

Note the order of the components of the data segment, and that there is no embedded unused space.

The user mode segmentation need to be set to reflect the values in the following table, where "t", "d" denote the block numbers of beginning of the text and data segments respectively:

Page	Address	Size	Comment
1	t+0	1.0	read only
2	t+128	0.7	read only
3	d+16	1.0	
4	d+144	1.0	
5	d+272	1.0	
6	d+400	0.3	
7	?	0.0	not used
8	d+400	0.7	grows downward

Note the setting of the eighth address register. The address prototypes stored in the array "u.u_uisa" are obtained by setting "t" and "d" to zero.

7.10 Setting the Segmentation Registers

Prototypes for the user segmentation registers are set up by "estabur" which is called when a program is first launched into execution, and again whenever a significant change in memory allocation requires it. The prototypes are stored in the arrays "u.u_uisa", "u.u_uisd".

Whenever process #i is about to be reactivated, the procedure "sureg" is called to copy the the prototypes into the appropriate registers. The description registers are copied directly, but the address registers must be adjusted to reflect the actual location in physical memory of the area used.

7.11 estabur (1650)

1654: Various checks on consistency are performed, to ensure that the requested sizes for the text, data and stack are reasonable.

Note that a non-zero value for "sep" implies separate mappings for the text area ("i" space) and the data area ("d" space). This is never possible on the PDP11/40;

1664: "a" defines the address of a segment relative to an arbitrary base of zero. "ap" and "dp" point to the set of prototype segmentation address and descriptor registers respectively.

The first eight of each of these sets are intended to refer to "i" space, and

1667: "nt" measures the number of 32 word blocks needed for the text segment. If "nt" is non-zero, one or more pages must be allocated for this purpose.

Where more than one page is allocated, all but the last will consist of 128 blocks (4096 words), and will be read only, and will have relative addresses starting at zero and increasing successively by 128.

1672: If some fraction of a page of text is still to be assigned, allocate the appropriate part of the next page;

1677: if "i" and "d" spaces are being used separately, mark the segmentation registers for the remaining "i" pages as null;

1682: "a" is reset because all remaining addresses refer to the data area (not the text area) and are relative to the beginning of this area. The first "USIZE" blocks of this area are reserved for the "per process data area";

1703: The stack area is allocated from the top of the address space towards the lower addresses ("downwards");

1711: If a partial page must be allocated for the stack area, it is the high address art of the page which is valid. (For text and data areas, which grow "upwards", it is the lower part of a partial page which is valid.) This requires an extra bit in the descriptor, hence "ED" ("expansion downwards");

1714: If separate "i" and "d" spaces are not used, only the first eight of the sixteen prototype register pairs will have been initialised by this point. In this case, the second eight are copied from the first eight.

7.12 sureg (1739)

This routine is called by "estabur" (1724), "swtch" (2229) and "expand" (2295), to copy the prototype segmentation registers into the actual hardware segmentation registers.

1743: Get the base address for the data area from the appropriate element of the "proc" array;

1744: The prototype address registers (of which there are only eight for the PDP11/40) are modified by the addition of "a" and stored in the hardware segmentation address registers;

1752: Test if a separate text area has been allocated, and if so, reset "a" to the relative address of the text area to the data area. (Note this value may be negative! Fortunately at this point, addresses are in terms of 32 word blocks.);

1754: The pattern of code now followed is similar to the beginning of the routine, except ...

1762: a rather obscure piece of code adjusts the setting of the address register for segments which are not "writable" i.e. which presumably are text segments.

The code in "estabur" and "sureg" shows evidence of having been developed in several stages and is not as elegant as could be desired.

7.13 newproc (1826)

It is now time to take a good look at the procedure which creates new processes as (almost exact) replicas of their creators.

1841: "mpid" is an integer which is stepped through the values 0 to 32767. As each new process is created, a new value for "mpid" is created to provide a unique distinguishing number for the process. Since the cycle of values may eventually repeat, a check is made that the number is not still in use; if so a new value is tried;

1846: A search is made through the "proc" array for a null "proc" structure (indicated by "p_stat" having a null value);

1860: At this point, the address of the new entry in the "proc" array is stored as both "p" and "rpp", and the address of "proc" entry for the current process is stored both as "up" and "rip";

1861: The attributes of the new process are stored in the new "proc" entry. Many of these are copied from the current process;

1876: The new process inherits the open files of its parent. Increment the reference count for each of these;

1879: If there is a separate text segment increment the associated reference counts. Notice that "rip", "rpp" are used for temporary reference here;

1883: Increment the reference count for the parent's current directory;

1889: Save the current values of the environment and stack pointers in "u.u_rsav". "savu" is an assembler routine defined at line 0725;

1890: Restore the values of "rip" and "rpp". Temporarily change the value of "u.u_procp" from the value appropriate to the current process to the value appropriate to the new process;

1896: Try to find an area in main memory in which to create the new data segment;

1902: If there is no suitable area in main memory, the new copy will have to be made on disk. The next section of code should be analysed carefully because of the inconsistency introduced at line 1891 i.e.
u.u_procp->p_addr != *ka6

1903: Mark the current process as "SIDL" to head off temporarily any further attempt to swap it out (i.e. initiated by "sched" (1940));

1904: Make the new "proc" entry consistent, i.e set rpp->p_addr = *ka6;

1905: Save the current values of the environment and stack pointers in "u.u_ssav";

1906: Call "xswap" (4368) to copy the data segment into the disk swap area. Because the second parameter is zero, the main memory area will not be released;

1907: Mark the new process as "swapped out";

1908: Return the current process to its normal state;

1913: There was room in main memory, so store the address of the new "proc" entry and copy the data segment a block at a time;

1917: Restore the current process' "per process data area" to its previous state;

1918: Return with a value of zero.

Obviously "newproc" on its own is not sufficient to produce an interesting and varied set of processes. The procedure "exec" (3020) which is discussed in Chapter Twelve provides the necessary additional facility: the means for a process to change its character, to be reincarnated.

8 Process Management

Process management is concerned with the sharing of the processor and the main memory amongst the various processes, which can be seen as competitors for these resources.

Decisions to reallocate resources are made from time to time, either on the initiative of the process which holds the resource, of for some other reason.

8.1 Process Switching

An active process may suspend itself i.e relinquish the processor, by calling "swtch" (2178) which calls "retu" (0740). This may be done for example if a process has reached a point beyond which it cannot proceed immediately. The process calls "sleep" (2066) which calls "swtch".

Alternatively a kernel process which is ready to revert to user mode will test the variable "runrun" and if this is non-zero, implying that a process with a higher precedence is ready to run, the kernel process will call "swtch".

"swtch" searches the "proc" table, for entries for which "p_stat" equals "SRUN" and the "SLOAD" bit is set in "p_flag". From these it selects the process for which the value of "p_pri" is a minimum, and transfers control to it.

Values for "p_pri" are recalculated for each process from time to time by use of the procedure "setpri" (2156). Obviously the algorithm used by "setpri" has a significant influence. A process which has called "sleep" and suspended itself may be returned to the "ready to run" state by another process. This often occurs during the handling of interrupts when the process handling the interrupt calls "setrun" (2134) either directly or indirectly via a call on "wakeup" (2113).

8.2 Interrupts

It should be noted that a hardware interrupt (see Chapter Nine) does not directly cause a call on "swtch" or its equivalent. A hardware interrupt will cause a user process to revert to a kernel process, which as just noted, may call "swtch" as an alternative to reverting to user mode after the interrupt handling is complete.

If a kernel process is interrupted, then after the interrupt has been handled, the kernel process resumes where it had left off regardless. This point is important for understanding how UNIX avoids many of the pitfalls associated with "critical sections" of code, which are discussed at the end of this chapter.

8.3 Program Swapping

In general there will be insufficient main memory for all the process images at once, and the data segments for some of these will have to be "swapped out" i.e. written to disk in a special area designated as the swap area.

While on disk the process images are relatively inaccessible and certainly unexecutable. The set of process images in main memory must therefore be changed regularly by swapping images in and out. Most decisions regarding swapping are made by the procedure "sched" (1940) which is considered in detail in Chapter Fourteen.

"sched" is executed by process #0, which after completing its initial tasks, spends its time in a double role: openly as the "scheduler" i.e. a normal kernel process; and surreptitiously as the intermediate process of "swtch" (discussed in Chapter Seven). Since the procedure "sched" never terminates, kernel process #0 never completes its task, and so the question of a user process #0 does not arise.

8.4 Jobs

There is no concept of "job" in UNIX, at least in the sense in which this term is understood in more conventional, batch processing oriented systems.

Any process may "fork" a new copy of itself at any time, essentially without delay, and hence create the equivalent of a new job. Hence job scheduling, job classes, etc. are non-events here.

8.5 Assembler Procedures

The next three procedures are written in assembler and run with the processor priority level set to seven. These procedures do not observe the normal procedure entry conventions so that r5 and r6, the environment and stack pointers, are not disturbed during procedure entry and exit.

As has already been noted, "savu" and "retu" can combine to produce the effect of a coroutine jump. The third procedure, "aretu," when followed by a "return" statement produces the effect of a non-local "goto".

8.6 savu (0725)

This procedure is called by "newproc" (1889, 1905), "swtch" (2189, 2281), "expand" (2284), "trapl" (2846) and "xswap" (4476,4477).

The values of r5 and r6 are stored in the array whose address is passed as a parameter.

8.7 retu (0740)

This procedure is called by "swtch" (2193, 2228) and "expand" (2294).

It resets the seventh kernel segmentation address register, and then resets r6 and r5 from the newly accessible copy of "u.u_rsav" (which it may be noted, is at the beginning of "u").

8.8 aretu (0734)

This procedure is called by "sleep" (2106) and "swtch" (2242).

It reloads r6 and r5 from the address passed as a parameter.

8.9 swtch (2178)

"swtch" is called by "trap" (0770, 0791), "sleep" (2084, 2093), "expand" (2287), "exit" (3256), "stop" (4027) and "xalloc" (4480).

This procedure is unique in that its execution is in three phases which in general involve three separate kernel processes. The first and third of these processes will be called the "retiring" and the "arising" processes respectively. Process #0 is always the intermediate process; it may be the "retiring" or the "arising" process as well.

Note that the only variables used by "swtch" are either registers, or global or static (stored globally).

2184: The static structure pointer, "p", defines a starting point for searching through the "proc" array to locate the next process to activate. Its use reduces the bias shown to processes entered early in the "proc" array. If "p" is null, set its value to the beginning of the "proc" array. This should only occur upon the very first call on "swtch";

2189: A call on "savu" (0725) saves the current values of the environment and stack pointers (r5 and r6);

2193: "retu" (0740) resets r5 and r6, and, most importantly, resets the kernel address register 6 to address the "scheduler's" data segment;

2195: Phase Two begins:

The code from this line to line 2224 is only ever executed by kernel process #0. There are two nested loops, from which there is no exit until a runnable process can be found.

At slack periods, the processor spends most of its time executing line 2220. It is only disturbed thence by an interrupt (e.g. from the clock);

2196: The flag "runrun" is reset. (It is used to indicate that a higher priority process than the current process is ready to run. "swtch" is about to look for the highest priority process.);

2224: The priority of the "arising" process is noted in "curpri" (a global variable) for future reference and comparison;

2228: Another call on "retu" resets r5, r6 and the seventh kernel address register to values appropriate for the "arising" process;

2229: Phase Three begins:

"sureg" (1739) resets the user mode hardware segmentation registers using the stored prototypes for the arising process;

2230: The comment which begins here is not encouraging. We will return to this point again towards the end of this chapter;

2247: If you check, you will find that none of the procedures which call "swtch" directly examines the value returned here.

Only the procedures which call "newproc" which are interested in this value, because of the way the child process is first activated!

8.10 setpri (2156)

2161: Process priorities are calculated according to the formula

$$\text{priority} = \min(127, (\text{time used} + \text{PUSER} + \text{p\_nice}))$$

where

(1) time used = accumulated central processor time (usually since the process was last swapped in), measured in clock ticks divided by 16 i.e. thirds of a second. (More on this later when we discuss the clock interrupt.);

(2) PUSER == 100;

(3) "p_nice" is a parameter used to bias the process priority. It is normally positive and hence reduces the process's effective precedence.

Note the somewhat confusing convention in UNIX that the lower the priority, the higher the precedence. Thus a priority of –10 beats a priority of 100 every time.

2165: Set the rescheduling flag if the process, whose priority has just been recalculated, has less precedence than the current process.

The sense of the test on line 2165 is surprising, especially when it is compared with line 2141. We leave it to the reader to satisfy himself that this is not an error. (Hint: look at the parameters for the calls on "setpri".)

8.11 sleep (2066)

This procedure is called (from nearly 30 different places in the code) when a kernel process chooses to suspend itself. There are two parameters:

- the reason for sleeping;
- a priority with which the process will run after being awakened.

If this priority is negative the process cannot be aroused from its sleep by the arrival of a "signal". "signals" are discussed in Chapter Thirteen.

2070: The current processor status is saved to preserve the incoming processor priority and previous mode information;

2072: If the priority is non-negative, a test is made for "waiting signals";

2075: A small critical section begins here, wherein the process status is changed and the parameters are stored in generally accessible locations (viz. within the array "proc").

This code is critical because the same information fields may be interrogated and changed by "wakeup" (2113) which is frequently called by interrupt handlers;

2080: When "runin" is non-zero, the scheduler (process #0) is waiting to swap another process into main memory;

2084: The call on "swtch" represents a delay of unknown extent during which a relevant external event may have occurred. Hence the second test on "issig" (2085) is not irrelevant;

2087: For negative priority "sleeps", where the process typically waits for freeing of system table space, the occurrence of a "signal" is not allowed to deflect the course of the activity.

8.12 wakeup (2113)

This procedure complements "sleep". It simply searches the set of all processes, looking for any processes which are "sleeping" for a specified reason (given as the parameter "chan"), and reactivating these individually by a call on "setrun".

8.13 setrun (2134)

2140: The process status is set to "SRUN". The process will now be considered by "swtch" and "sched" as a candidate for execution again;

2141: If the aroused process is more important (lower priority!) than the current process, the rescheduling flag, "runrun" is set for later reference;

2143: If "sched" is sleeping, waiting for a process to "swap in", and if the newly aroused process is on disk, wake up "sched".

Since it turns out that "sched" is the only procedure which calls sleep with "chan" equal to "&runout", line 2145 could be replaced by the recursive call

```
setrun(&proc[0]);
```

or better still, by just

```
rp = &proc[0];
goto sr;
```

where "sr" is a label to be inserted at the beginning of line 2139.

8.14 expand (2268)

The comment at the beginning of this procedure (2251) says most of what needs to be said about the procedure, except for the question of "swapping out" when not enough core is available.

Note that "expand" takes no particular notice of the contents of the user data area or stack area.

2277: If the expansion is actually a contraction, then trim off the excess from the high address end;

2281: "savu" stores the values of r5 and r6 in "u.u_rsav";

2283: If sufficient main memory is not available ...

2284: The environment pointer and stack pointer are recorded again in "u.u_ssav". But note that since no new procedures have been entered, and since there has been no cumulative stack growth, the values recorded are the same as at line 2281;

2285: "xswap" (4368) copies the core image for the process designated by its first parameter to disk.

Since the second parameter is non-zero the main memory area occupied by the data segment is returned to the list of available space.

However the computation continues using the same area in main memory until the next call on "retu" (2193) in "swtch".

Note also that the call on "savu" at line 2189 in "swtch" stores new values in "u.u_rsav" after the disk image has been made (and therefore serves no useful purpose since the core image has already been officially "abandoned");

2286: The "SSWAP" flag is set in the process's proc array element. (This is not swapped out, so the effect is not lost);

2287: "swtch" is called, and the process, still running in its old area suspends itself. Since the call on "xswap" will have resulted in the "SLOAD" flag being switched off, there is no way that "swtch" will choose the process for immediate reactivation.

Only after the disk image has been copied back into core again can the process be activated again. The "return" executed by "swtch" is a return to the procedure which called "expand".

8.15 swtch revisited

What happens to the process when it is reactivated i.e. it becomes the "arising" process in "swtch"?

2228: The stack and environment pointers are restored from "u.u_rsav" (Note that a pointer to "u" is also a pointer to "u.u_rsav" (0415) but ...

2240: If the core image was swapped out e.g. by "expand" ...

2242: No reliance is placed on the values of the stack and environment pointers, and they are reset

The question is if the values stored in "u.u_ssav" at line 2284 are the same as values stored in "u.u_rsav" at line 2281, how did they get to be different?

Presumably this is what "you are not expected to understand" (line 2238) ... clearly "xswap" should be investigated ... the trail finally ends at Chapter Fifteen ... in the meantime you may wish to investigate for yourself so that you may join the "2238" club that much sooner.

8.16 Critical Sections

If two or more processes operate on the same set of data, then the combined output of the set of processes may depend on the relative synchronisation of the various processes.

This is usually considered to be highly undesirable and to be avoided at all costs. The solution is usually to define "critical sections" (it is the programmer's responsibility to recognise these) in the code which is executed by each process. The programmer must then ensure that at any time no more than process is executing a section of code which is critical with respect to a part1cular set of data.

In UNIX user processes do not share data and so do not conflict in this way. Kernel processes however have shared access to various system data and can conflict.

In UNIX an interrupt does not cause a change in process as a direct side effect. Only where kernel processes may suspend themselves in the middle of a critical section by an explicit call on "sleep", does an explicit lock variable which may be observed by a group of processes) need to be introduced. Even then the actions of testing and setting the locks do not usually have to be made inseparable.

Some critical sections of code are executed by interrupt handlers. To protect other sections of code whose outcome may be affected by the handling of certain interrupts, the processor priority is raised temporarily high enough before the critical section is entered to delay such interrupts until it is safe, when the processor priority is reduced again. There are of course a number of conventions which interrupt handling code should observe, as will be discussed later in Chapter Nine.

In passing it may be noted that the strategy adopted by UNIX works only for a single processor system and would be totally inappropriate in a multiprocessor system.

Section Two

Section Two is concerned with traps, hardware interrupts and software interrupts.

Traps and hardware interrupts introduce sudden switches into the CPU's normal instruction execution sequence. This provides a mechanism for handling special conditions which occur outside the CPU's immediate control.

Use is made of this facility as part of another mechanism called the "system call", whereby a user program may execute a "trap" instruction to cause a trap deliberately and so obtain the operating system's attention and assistance.

The software interrupt (or "signal") is a mechanism for communication between processes, particularly when there is "bad news".

9 Hardware Interrupts and Traps

In the PDP11 computer, as in many other computers, there is an "interrupt" mechanism, which allows the controllers of peripheral devices (which are devices external to the CPU) to interrupt the CPU at appropriate times, with requests for operating system service.

The same mechanism has been usefully and conveniently applied to "traps" which are events internal to the CPU, which relate to hardware and software errors, and to requests for service from user programs.

9.1 Hardware Interrupts

The effect of an interrupt is to divert the CPU from whatever it was doing and to redirect it to execute another program.

During a hardware interrupt:

- The CPU saves the current processor status word (PS) and the current program count (PC) in its internal registers;
- the PC and PS are then reloaded from two consecutive words located in the low area of main memory. The address of the first of these two words is known as the "**vector location**" of the interrupt;
- finally the original PC and PS values are stored into the newly current stack. (Whether this is the kernel or user stack depends on the new value of the PS.)

Different peripheral devices may have different vector locations. The actual vector location for a particular device is determined by hard wiring, and can only be changed with difficulty. Moreover there are well entrenched conventions for choosing vector locations for the various devices.

Thus after the interrupt has occurred, because the PC has been reloaded, the source of instructions executed by the CPU has been changed. The new source should be a procedure associated with the peripheral device controller which caused the interrupt.

Also since the PS has also been changed, the processor mode may have changed. In UNIX, the initial mode may be either "user" or "kernel", but after the interrupt, the mode is always "kernel". Recall also that a change in mode implies:

(a) a change in memory mappings. (Note that to avoid any confusion, vector locations are always interpreted as kernel mode addresses.);

(b) a change in stack pointers. (Recall that the stack pointer, SP or r6, is the only special register which is replicated for each mode. This implies that after a mode change, the stack pointer value will have changed even though it has not been reloaded!)

9.2 The Interrupt Vector

For our sample system, the representative peripheral devices chosen are listed in Table 9.1, along with their conventional hardware defined vector locations and priorities.

vector location	peripheral device	interrupt priority	process priority
060	teletype input	4	4
064	teletype output	4	4
070	paper tape input	4	4
074	paper tape output	4	4
100	line clock	6	6
104	programmable clock	6	6
200	line printer	4	4
220	RK disk drive	5	5

Table 9.1 Interrupt Vector Locations and Priorities

9.3 Interrupt Handlers

Within this selection of UNIX source code, there are seven procedures known as "interrupt handlers", i.e. which are executed as the result of, and only as the result of, interrupts:

```
clock  (3725)  pcrint (8719)
rkintr (5451)  pcpint (8739)
klxint (8070)  lpint  (8976)
klrint (8078)
```

"clock" will be examined in detail in Chapter 11. The others are discussed with the code for their associated devices.

9.4 Priorities

An interrupt does not necessarily occur immediately the peripheral device controller requests it, but only when the CPU is ready to accept it. It is usually desirable that a request for a low priority service should not be allowed to interrupt an activity with a higher priority.

Bits 7 to 5 of the PS determine the processor priority at one of eight levels (labelled zero to seven). Each interrupt also has an associated priority level determined by hardware wiring. An interrupt will be inhibited as long as the processor priority is greater than or equal to the interrupt priority.

After the interrupt the processor priority will be determined from the PS stored in the vector location and this does not have to be the same as the interrupt priority. Whereas the interrupt priority is determined by hardware, it is possible for the operating system to change the contents of the vector location at any time.

As a matter of curiosity, it may be noted that the PDP11 hardware restricts the possible interrupt priorities to 4, 5, 6 and 7 i.e. levels 1, 2 and 3 are not supported by the Unibus.

9.5 Interrupt Priorities

In UNIX, interrupt handling routines are initiated at the same priority as the interrupt priority.

This means that during the handling of the interrupt, a second interrupt from a device of the same priority class will be delayed until the processor priority is reduced, either by the execution of one of the "spl" procedures, which are intended for just this purpose (see lines 1293 to 1315), or by reloading the processor status word e.g. upon returning from the interrupt.

During interrupt handling, the processor priority may be raised temporarily to protect the integrity of certain operations. For instance, character oriented devices such as the paper tape reader/punch or the line printer interrupt at level four. Their interrupt handlers call "getc" (0930) or "putc" (0967), which raise the processor priority temporarily to level five, while the character buffer queues are manipulated.

The interrupt handler for the console teletype makes use of a "timeout" facility. This involves a queue which is also manipulated by the clock interrupt handler, which runs at level six. To prevent possible interference, the "timeout" procedure (3835) runs at level seven (the highest possible level).

Usually it does not make sense to run an interrupt handler at a processor priority lower than the interrupt priority, for this would then risk a second interrupt of the same type, even from the same device, before completion of the processing of the first interrupt. This likely to be at best inconvenient and at worst disastrous. However the clock interrupt handler, which once per second has a lot of extra work to do, does exactly this.

9.6 Rules for Interrupt Handlers

As discussed above, interrupt handlers need to be careful about the manipulation of the processor priority to avoid allowing other interrupts to happen "too soon". Likewise care needs to be taken that the other interrupts are not delayed excessively, lest the performance of the whole system be degraded. It is important to note that when an interrupt occurs, the process which is currently active will very likely not be the process which is interested in the occurrence. Consider the following scenario:

User process #m is active and initiates an i/o operation. It executes a trap instruction and transfers to kernel mode. Kernel process #m initiates the required operation and then calls "sleep" to suspend itself to await completion of the operation ...

Some time later, when some other process, user process #n say, is active, the operation is completed and an interrupt occurs. Process #n reverts to kernel mode, and kernel process #n deals with the interrupt, even though it may have no interest in or prior knowledge of the operation.

Usually kernel process #n will include waking process #m as part of its activity. This will not always be the case though, e.g. where an error has occurred and the operation is retried.

Clearly, the interrupt handler for a peripheral device should not made references to the current "u" structure for this is not likely to be the appropriate "u" structure. (The appropriate "u" structure could quite possibly be inaccessible, if it has been temporarily swapped out to the disk.)

Likewise the interrupt handler should not call "sleep" because the process thus suspended will most likely be some innocent process.

9.7 Traps

"Traps" are like "interrupts" in that they are events which are handled by the same hardware mechanism, and hence by similar software mechanisms.

"Traps" are unlike "interrupts" in that they occur as the result of events internal to the CPU, rather than externally. (In other systems the terminology "internal interrupt" and "external interrupt" is used to draw this distinction more forcefully.) Traps may occur unexpectedly as the result of hardware or power failures, or predictably and reproducibly, e.g. as the result of executing an illegal instruction or a "trap" instruction.

"Traps" are always recognised by the CPU immediately. They cannot be delayed in the way low priority interrupts may be. If you like, "traps" have an "interrupt priority" of eight.

"Trap" instructions may be deliberately inserted in user mode programs to catch the attention of the operating system with a request to perform a specified service. This mechanism is used as part of the facility known as "system calls".

Like interrupts, traps result in the reloading of the PC and PS from a vector location, and the saving of the old values of the PC and PS in the current stack. Table 9.2 lists the vector locations for the various "trap" types.

vector location	**trap type**	**process priority**
004	bus timeout	7
010	illegal instruction	7
014	bpt-trace	7
020	iot	7
024	power failure	7
030	emulator trap instruction	7
034	trap instruction	7
114	11/10 parity	7
240	programmed interrupt	7
244	floating point error	7
250	segmentation violation	7

Table 9.2 Trap Vector Locations and Priorities

The contents of Tables 9.1 and 9.2 should be compared with the file "low.s" on Sheet 05. As noted earlier, this file is generated at each installation (along with the file "conf.c" (sheet 46)), as the product of the utility program "mkconf", so as to reflect the actual set of peripherals installed.

9.8 Assembly Language 'trap'

From "low.s" it appears that traps and interrupts are handled separately by the software. However closer examination reveals that "call" and "trap" are different entry points to a single code sequence in the file "m40.s" (see lines 0755, 0776). This sequence is examined in detail in the next chapter.

During the execution of this sequence, a call is made on a "C" language procedure to carry out further specific processing. In the case of an interrupt, the "C" procedure is the interrupt handler specific to the particular device controller.

In the case of a trap, the "C" procedure is another procedure called "trap" (yes, the word "trap" is definitely overworked!), which in the case process of a system error will most likely call priority "panic" and in the case of a "system call", will invoke (indirectly via "trapl"(2841)) the appropriate system call procedure.

9.9 Return

Upon completion of the handling of an interrupt or trap the code follows a common path ending in an "rtt" instruction (0805). This reloads both the PC and PS from the current stack, i.e. the kernel stack, in order to restore the processor environment that existed before the interrupt or trap.

10 The Assembler "Trap" Routine

The principal purpose of this chapter is to examine the assembly language code in "m40.s" which is involved in the handling of interrupts and traps.

This code is found between lines 0750 and 0805, and has two entry points, "trap" (0755) and "call" (0766). There are several different and relevant paths through this code and we shall trace some examples of these.

10.1 Sources of Traps and Interrupts

The discussion in Section One introduced three places where the occurrence of a trap or interrupt was expected:

(a) "main" (1564) calls "fuibyte" repeatedly until a negative value is returned. This will occur after a "bus timeout error" has been encountered with a subsequent trap to vector location 4 (line 0512);

(b) The clock has been set running and will generate an interrupt every clock tick i.e. 16.7 or 20 milliseconds;

(c) Process #1 is about to execute a "trap" instruction as part of the system call on "exec".

10.2 fuibyte (0814), fuiword (0844)

"main" uses both "fuibyte" and "fuiword". Since the former is more complicated in a non-essential way, we leave it to the reader, and concentrate on the latter.

"fuiword" is called (1602) when the system is running in kernel mode with one argument which is an address in user address space. The function of the routine is to fetch the value of the corresponding word and to return it as a result (left in r0). However if an error occurs, the value –1 is to be returned.

Note that with "fuiword", there is an ambiguity which does not occur with "fuibyte", namely a returned value of –1 may not necessarily be an error indication but the actual value in the user space. Convince yourself that for the way it is used in "main", this does not matter.

Also the code does not distinguish between a "bus timeout error" and a "segmentation error".

The routine proceeds as follows:

0846: The argument is moved to r1;

0848: "gword" is called;

0852: The current PS is stored on the stack;

0853: The priority level is raised to 7 (to disable interrupts);

0854: The contents of the location nofault (1466) are saved in the stack;

0855: "nofault" is loaded with address of the routine "err";

0856: An "mfpi" instruction is used to fetch the word from user space.

If nothing goes wrong this value will left on the kernel stack.

0857: The value is transferred from the stack to r0;

0876: The previous values of "nofault" and PS are restored;

Now suppose something does go wrong with the "mfpi" instruction, and a bus time-out does occur.

0856: The "mfpi" instruction will be aborted. PC will point to the next instruction (0857) and a trap via vector location 4 will occur;

0512: The new PC will have the value of "trap". The new PS will indicate:

present mode = kernel mode

previous mode = kernel mode

priority = 7;

0756: The next instruction executed is the first instruction of "trap". This saves the processor status word two words beyond the current "top of stack". (This is not relevant here.);

0757: "nofault" contains the address of "err" and is non-zero;

0765: Moving 1 to SR0 reinitialises the memory management unit;

0766: The contents of "nofault" are moved on top of the stack, **overwriting** the previous contents, which was the return address in "gword";

0767: The "rtt" returns, not to "gword" but to the first word of "err";

0880: "err" restores "nofault" and PS, skips the return to "fuiword", places –1 in r0, and returns directly to the calling routine.

10.3 Interrupts

Suppose the clock has interrupted the processor.

Both clock vector locations, 100 and 104, have the same information. PC is set to the address of the location labelled "kwlp" (0568) and PS is set to show:

present mode = kernel mode
previous mode = kernel or user mode
priority = 6

Note. The PS will contain the true previous mode, regardless of the value picked up from the vector location.

0570: The vector location contains a new PC value which is the address of the statement labelled "kwlp". This instruction is a subroutine call on "call" via r0.

After the execution of this instruction, r0 is left with the address of the code word after the instruction which contains "_clock", i.e. r0 contains **the address of the address** of the "clock" routine in the file "clock.c" (3725).

10.4 call (0776)

0777: Copy PS onto the stack;

0779: Copy r1 onto the stack;

0780: Copy the stack pointer for the previous address space onto the stack. (This is only significant if the previous mode was user mode).

This represents a **special case** of the "mfpi" instruction. See the "PDP11 Processor Handbook", page 6-20;

781: Copy the copy of PS onto the stack and mask out all but the lower five bits. The resulting value designates the cause of the interrupt (or trap). The original value of the PS had to be captured quickly;

0783: Test if the previous mode is kernel or user.

If the previous mode is kernel mode the branch is taken (0784). PS is changed to show the previous mode as user mode (0798);

0799: The specialised interrupt handling routine pointed to by r0 is entered. (In this case it is the routine "clock", which is discussed in detail in the next chapter.)

0800: When the "clock" routine (or some other interrupt handler) returns, the top two words of the stack are deleted. These are the masked copy of the PS and the copy of the stack pointer;

0802: r1 is restored from the stack;

0803: Delete the copy of PS from the stack;

0804: Restore the value of r0 from the stack;

0805: Finally the "rtt" instruction returns to the "kernel" mode routine that was interrupted;

If the previous mode was user mode it is not certain that the interrupted routine will be resumed immediately;

0788: After the specialised interrupt routine (in this case "clock") returns, a check ("runrun > 0") is made to see if any process of higher priority than the current process is ready to run. If the decision is to allow the current process to continue, then it is important that it be not interrupted as it restores its registers prior to the "return from interrupt" instruction. Hence before the test, the processor priority is raised to seven (line 0787), thus ensuring that no more interrupts occur until user mode is resumed. (Another interrupt may occur immediately thereafter, however.)

If "runrun > 0", then another, higher priority, process is waiting. The processor priority is reset to 0, allowing any pending interrupt to be taken. A call is then made to "swtch" (2178), to allow the higher priority process to proceed. When the process returns from "swtch", the program loops back to repeat the test.

The above discussion obviously extends to all interrupts. The only part which relates specifically to the clock interrupt is the call on the specialised routine "clock".

10.5 User Program Traps

The "system call" mechanism which enables user mode programs to call on the operating system for assistance, involves the execution by the user mode program of one of 256 versions of the "trap" instruction. (The "version" is the value of the low order byte of the instruction word.)

0518: Execution of the trap instruction in a user mode program causes a trap to occur to vector location 34 which causes the PC to be loaded with the value of the label "trap" (lines 0512, 0755). A new PS is set which indicates

present mode = kernel mode

previous mode = user mode

priority = 7

0756: The next instruction executed is the first instruction of "trap". This saves the processor status word in the stack two words beyond the current "top of stack".

It is important to save the PS as soon as possible, before it can be changed, since it contains information defining the type of trap that occurred. The somewhat unconventional destination of the "move" is to provide compatibility with the handling of interrupts, so that the same code can be used further on;

0757: "nofault" will be zero so the branch is not taken;

0759: The memory management status registers are stored just in case they will be needed, and the memory management unit is reinitialised;

0762: A subroutine entry is made to "call" using r0. (This neatly stores the old value of r0 in the stack, but not a return address. The new value is the address of the address of the routine to be entered next (in this case the "trap" routine in the file "trap.c" (2693));

0772: The stack pointer is adjusted to point to the location which already contains the copy of PS;

0773: The CPU priority is set to zero;

From here the same path as for an interrupt is followed.

10.6 The Kernel Stack

The state of the kernel stack at the time that the "trap" procedure ("C" version) or one of the specialised interrupt handling routines is entered, is shown in Figure 10.1.

			....	**Previous top of stack**
(rps	2)	7	ps	old PS
(r7	1)	6	pc	old PC (r7)
(r0	0)	5 ->	r0	old r0
		4	nps	new PS after trap
(r1	−2)	3	r1	old r1
(r6	−3)	2	sp	old SP for previous mode
		1	dev	masked new PS
		0 ->	tpc	return address in "call"
(r5	−6	−1	(r5)	old r5
(r4	−7	−2	(r4)	old r4
(r3	−8	−3	(r3)	old r3
(r2	−9	−4	(r2)	old r2

Figure 10.1

Columns (2) and (3) give the positions of stack words relative to the positions in the stack of the words labelled "r0" and "tpc" respectively.

Columns (1) and (2) define (or explain) the contents of the file "reg.h" (Sheet 26).

"dev", "sp", "r1", "nps" "r0", "pc" and "ps" in that order are the names of the parameters used in the declaration of the procedures "trap" (2693) and "clock" (3725).

Note that just before entry to "trap" ("C" version) or the other interrupt handling routines, the values for the registers r2, r3, r4 and r5 have not yet been saved in the stack. This is performed by a call on "csv" (lg20) which is automatically included by the "C" compiler at the beginning of every compiled procedure. The form of the call on "csv" is equivalent to the assembler instruction

```
jsr r5,csv
```

This saves the current value of r5 on the stack and replaces it by the address of the next instruction in the "C" procedure.

1421: This value of r5 is copied into r0;

1422: the current value of the stack pointer is copied into r5.

Note that at this point, r5 points to a stack location containing the previous value of r5 i.e. it points to the beginning of a chain of pointers, one per procedure, which "thread" the stack. When a "C" procedure exits, it actually returns to "cret" (1430) where the value of r5 is used to restore the stack and r2, r3 and r4 to their earlier condition (i.e. as they were immediately prior to entering the procedure). For this reason r5 is often called the **environment pointer**.

11 Clock Interrupts

The procedure "clock" (3725) handles interrupts from either the line frequency time clock (type KW11-L, interrupt vector address 100) or the programmable real-time clock (type KW11-P, interrupt vector address 104).

UNIX requires that at least one of these should be available. (If both are present, only the line time clock is used.)

Whichever clock is used, interrupts are generated at line frequency (i.e. with a 50 Hz power supply, every 20 milliseconds). The clock interrupt priority level is six, higher than for any other peripheral device on our typical system, so that there will usually be very little delay in the initiation of "clock" once the interrupt has been requested by the clock controller.

11.1 clock (3725)

The function of "clock" is one of general housekeeping:

- the display register is updated (PDP11/45 and 11/70 only);
- various accounting values such as the time of day, accumulated processing times and execution profiles are maintained;
- processes sleeping for a fixed time interval are awakened as per schedule;
- core swapping activity is initiated once per second.

"clock" breaks most of the rules for peripheral device handlers: it does reference the current "u" structure, and it also runs at a low priority for some of the time. It abbreviates its activity if a previous execution has not yet completed.

3740: "display" is a no-op on the PDP11/40;

3743: The array "callout" (0265) is an array of "NCALL" (0143) structures of type "callo" (0260). The "callo" structure contains three elements: an incremental time, an argument and the address of a function. When the function element is not null, the function is to be executed with the supplied argument after a specified time.

(For the systems under study, the only function ever executed in this way is "ttrstrt" (8486), handler. (See Chapter 25.));

3748: If the first element of the list is null, the whole list is null;

3750: The "callout" list is arranged in the desired order of execution. The time recorded is the number of clock ticks between events. Unless the first time (the time before the next event) is already zero, (meaning that the execution is already due) this time should be decremented by one.

If this time has already been counted to zero, decrement the next time unless it is already zero also, etc. i.e. decrement the first non-zero time in the list. All the leading entries with zero times represent operations which are already due. (The operations are actually carried out a little later.);

3759: Examine the previous processor status word, and if the priority was non-zero, bypass the next section, which executes those operations which are due;

3766: Reduce the processor priority to five (other level six interrupts may now occur);

3767: Search the "callout" array looking for operations which are due and execute them;

3773: Move the entries for operations which are still not yet due, to the beginning of the array;

3787: The code from here until line 3797 is executed, whatever the previous processor priority, at either priority level five or six;

3788: If the previous mode was "user mode", then increment the user time counter, and if an execution profile is being accumulated, call "incupc" (a895) to make an entry in a histogram for the user mode program counter (PC).

"incupc" is written in assembler, presumably for efficiency and convenience. A description of what it does may be found in the section "PROFIL(II)" of the UPM. See also the procedure "profil" (3667);

3792: If the previous mode was not user mode, increment the system (kernel) time counter for the process.

The code just described performs the basic time accounting for the system. Every clock tick results in the incrementing of either "u.u_utime" or "u.u_stime" for some process. Both "u.u_utime" and "u.u_stime" are initialised to zero in "fork" (3322). Their values are interrogated in "wait" (3270). The values will go negative after 32K ticks (about 10 hours)!

3795: "p_cpu" is used in determining process priorities. It is a character value which is always interpreted as a positive integer (0 to 255). When it is moved to a special register, sign extension occurs so that 255, for instance, becomes like –1. Adding one then leaves a zero result. In this case the value is reduced to –1 again, and stored as 255 unsigned. Note that in the other places where "p_cpu" is referenced (2161, 3814), the top eight bits are masked off after the value has been transferred to a special register;

3797: Increment "lbolt" and if it exceeds "HZ", i.e. a second or more has elapsed ...

3798: Then provided the processor was not previously running at a nonzero priority, do a whole lot of housekeeping;

3800: Decrement "lbolt" by "HZ";

3801: Increment the time of day accumulator;

3803: The events which follow may take some time, but they may reasonably be interrupted to service other peripherals. So the processor priority is dropped below all the device priority levels i.e. below **four**.

However there is now a possibility of another clock interrupt before this activation of the "clock" procedure is completed. By setting the processor priority to **one** rather than to **zero**, a second activation of "clock" will not attempt to execute the code from line 3804 on also. Note however that to the hardware, priority one is functionally the same as priority zero;

3804: If the current time (measured in seconds) is equal to the value stored in "tout", wake all processes which have elected to suspend themselves for a period of time via the "sleep" system call i.e. via the procedure "sslep" (5979).

"tout" stores the time at which the next process is to be awakened. If there is more than one such process, then the remainder, which will have been disturbed, must reset "tout" between them. This mechanism, while quite effective, will not be efficient if the number of such processes ever becomes large.

In this situation, a mechanism similar to the "callout" array (see 3767) would need to be provided. (In fact, how difficult would it be to merge the two mechanisms? What would be the disadvantages ??);

3806: When the last two bits of "time[1]" are zero i.e. every four seconds, reset the scheduling flag "runrun" and wake up everything waiting for a "lightning bolt". ("lbolt" represents a general event which is caused every four seconds, to initiate miscellaneous housekeeping. It is used by "pcopen" (8648).);

3810: For all currently defined processes:

increment "p_time" up to a maximum of 127 (it is only a character variable);

decrement "p_cpu" by "SCHMAG" (3707) but do not allow it to go negative. Note that as discussed earlier (line 3795) "p_cpu" is treated as a positive integer in the range 0 to 255;

if the processor priority is currently set at a depressed value, recalculate it.

Note that "p_cpu" enters into the calculation of process priorities, "p_pri", by "setpri" (2156). "p_pri" is used by "swtch" (2209) in choosing which process, from among those which are in core ("SLOAD") and ready to run ("SRUN"), should next receive the CPU's attention.

"p_time" is used to measure how long (in seconds) a process has been either in core or swapped out to disk. "p_time" is set to zero by "newproc" (1869), by "sched" (2047) and by "xswap" (4386). It is used by "sched" (1962, 2009) to determine which processes to swap in or out.

3820: If the scheduler is waiting to rearrange things, wake it up. Thus the normal rate for scheduling decisions is once per second;

3824: If the previous mode before the interrupt was "user mode", store the address of "r0" in a standard place, and if a "signal" has been received for the process, call "psig" (4043) for the appropriate action.

11.2 timeout (3845)

This procedure makes new entries in the "callout" array. In this system it is only called from the routine "ttstart" (8505), passing the procedure "ttrstrt" (3486). Note that "ttrstrt" calls "ttstart", which may call "timeout", for a thoroughly incestuous relationship!

Note also that most of "timeout" runs at priority level seven, to avoid clock interrupts.

12 Traps and System Calls

This chapter is concerned with the way the system handles traps in general and system calls in particular.

There are quite a number of conditions which can cause the processor to "trap". Many of these are quite clearly error conditions, such as hardware or power failures, and UNIX does not attempt any sophisticated recovery procedures for these.

The initial focus for our attention is the principal procedure in the file "trap.c".

12.1 trap (2693)

The way that this procedure is invoked was explored in Chapter Ten. The assembler "trap" routine carries out certain fundamental housekeeping tasks to set up the kernel stack, so that when this procedure is called, everything appears to be kosher.

The "trap" procedure can operate as though it had been called by another "C" procedure in the normal way with seven parameters

dev, sp, rl, nps, r0, pc, ps.

(There is a special consideration which should be mentioned here in passing. Normally all parameters passed to "C" procedures are passed by value. If the procedure subsequently changes the values of the parameters, this will not affect the calling procedure directly.

However if "trap" or the interrupt handlers change the values of their parameters, the new values will be picked up and reflected back when the "previous mode" registers are restored.)

The value of "dev" was obtained by capturing the value of the processor status word immediately after the trap and masking out all but the lower five bits. Immediately before this, the processor status word had been set using the prototype contained in the appropriate vector location.

Thus if the second word of the vector location was "br7+n;" (e.g. line 0516) then the value of "dev" will be n.

2698: "savfp" saves the floating point registers (for the PDP11/40, this is a no-op!);

2700: If the previous mode is "user mode', the value of "dev" is modified by the addition of the octal value 020 (2662);

2701: The stack address where r0 is stored is noted in "u.u_ar0" for future reference. (Subsequently the various register values can be referenced as "u.u_ar0[Rn]".);

2702: There is now a multi-way "switch" depending on the value of "dev".

At this point we can observe that UNIX divides traps into three classes, depending on the prior processor mode and the source of the trap:

(a) kernel mode;

(b) user mode, not due to a "trap" instruction;

(c) user mode, due to a "trap" instruction.

12.2 Kernel Mode Traps

The trap is unexpected and with one exception, the reaction is to "panic". The code executed is the "default" of the "switch" statement:

2716: Print:

- the current value of the seventh kernel segment address register (i.e. the address of the current per process data area);
- the address of "ps" (which is in the kernel stack); and
- the trap type number;

2719: "panic", with no return.

Floating point operations are only used by programs, and not by the operating system. Since such operations on the PDP11/45 and 11/70 are handled asynchronously, it is possible that when a floating point exception occurs, the processor may have already switched to kernel mode to handle an interrupt.

Thus a kernel mode floating point exception trap can be expected occasionally and is the concern of the current user program.

2793: Call "psignal" (3963) to set a flag to show that a floating point exception has occurred;

2794: Return.

This raises an interesting question: "Why are the kernel mode and user mode floating point exceptions handled slightly diferently?"

12.3 User Mode Traps

Consider first of all a trap which is not generated as the result of the execution of a "trap instruction". This is regarded as a probable error for which the operating system makes no provision apart from the possibility of a "core dump". However the use program itself may have anticipated it and provided for it.

The way this provision is made and implemented is the subject of the next chapter. At this stage, the principal requirement is to "signal" that the trap has occurred.

2721: A bus error has occurred while the system is in user mode. Set "i" to the value "SIGBUS" (0123);

2723: The "break" causes a branch out of the "switch" statement to line 2818;

2733: Apart from the one special case noted, the treatment of illegal instructions is the same at this level as for bus errors;

2739:

2743:

2747:

2796: Cf. the comment for line 2721.

Note that cases "4+USER" (power fail) and "7+USER" (programmed interrupt) are handled by the "default" case (line 2715).

2810: This represents a case where operating system assistance is required to extend the user mode stack area.

The assembler routine "backup" (1012) is used to reconstruct the situation that existed before execution of the instruction that caused the trap.

"grow" (4136) is used to do the actual extension.

The procedure "backup" is non-trivial and its comprehension involves a careful consideration of various aspects of the PDP11 architecture. It has been left for the interested reader to pursue privately.

As noted for the PDP11/40, "backup" may not always succeed because the processor does not save enough information to resolve all possibilities.

218: Call "psignal" (3963) to set the appropriate "signal". (Note that this statement is only reached from those cases of the "switch" which included a "break" statement.);

2821: "issig" checks if a "signal" has been sent to the user program, either just now or at some earlier time and has not yet been attended to;

2822: "psig" performs the appropriate actions. (Both "issig" and "psig" are discussed in detail in the next chapter.);

2823: Recalculate the priority for the current process.

12.4 System Calls

User mode programs use "trap" instructions as part of the "system call" mechanism to call upon the operating system for assistance.

Since there are many possible "versions" of the "trap" instruction, the type of assistance requested can be and is encoded as part of the "trap" instruction.

Parameters which are part of a system call may be passed from the user program in different ways:

(a) via the special register r0;

(b) as a set of words embedded in the program string following the "trap" instruction;

(c) as a set of words in the program's data area. (This is the "indirect" call.)

Indirect calls have a higher overhead than direct system calls. Indirect calls are needed when the parameters are data dependent and cannot be determined at compile time.

Indirect calls may sometimes be avoided if there is only one data dependent parameter, which is passed via r0. In choosing which parameters should be passed via r0, the system designers have presumably been guided by their own experience, since the pattern doesn't satisfy the law of least astonishment.

The "C" compiler does not give special recognition to system calls, but treats them in the same way as other procedures. When the loader comes to resolve undetermined references, it satisfies these with library routines which contain the actual "trap" instructions.

2752: The error indicators are reset;

2754: The user mode instruction which caused the trap is retrieved and all but the least significant six bits are masked off. The result is used to select an entry from the array of structures, "sysent". The address of the selected entry is stored in "callp";

2755: The "zeroeth" system call is the "indirect" system call, in which the parameter passed is actually the address in the user program data space of a system call parameter sequence.

Note the separate uses of "fuword" and "fuiword". The distinction between these is unimportant on the PDP11/40, but is most important on machines with separate "i" and "d" address spaces;

2760: "i=077" simulates a call on the very last system call (2975), which results in a call on "nosys" (2855), which results in an error condition which will usually be fatal for the user mode program;

2762:

2765: The number of arguments specified in "sysent" is the actual number provided by the user programmer, or that number less one if one argument is transferred via r0. The arguments are copied from the user data or instruction area into the five element array "u.u_arg". (From "sysent" (Sheet 29) it would seem that four elements would have been sufficient for "u_area[ ]" – is this an allowance for future inflation?);

2770: The value of the first argument is copied into "u.u_dirp", which seems to function mainly as a convenient temporary storage location;

2771: "trapl" is called with the address of the desired system routine. Note the comment beginning on line 2828;

2776: When an error occurs, the "c-bit" in the old processor status word is set (see line 2658) and the error number is returned via r0.

12.5 System Call Handlers

The full set of system calls may be reviewed in the file "sysent.c" on Sheet 29, but more relevantly, these are discussed in full detail in Section II of the UPM.

The procedures which handle the system calls are found mostly in the files "sysl.c", sys2.c", sys3.c" and "sys4.c".

Two important "trivial" procedures are "nullsys" (2855) and "nosys" (2864) which are found in the file "trap.c".

12.6 The File 'sysl.c'

This file contains the procedures for five system calls, of which three will be considered now, and two ("rexit" and "wait") will be deferred to the next chapter.

The first procedure in this file, and also the first system call we have encountered, is "exec".

12.7 exec (3020)

This system call, #11, changes a process executing one program into a process executing a different program. See Section "EXEC(II)" of the UPM. This is the longest and one of the most important system calls.

3034: "namei" (6618) (which is discussed in detail in Chapter 19) converts the first argument (which is a pointer to a character string defining the name of the new program) into an "inode" reference. ("inodes" are essential parts of the file referencing mechanism.);

3037: Wait if the number of "exec"s currently under way is too large (See the comment on line 3011.);

3040: "getblk(NODEV)" results in the allocation of a 512 byte buffer from the pool of buffers. This buffer is used temporarily to store in core, that information which is currently in the user data area, and which is needed to start the new program. Note that the second argument in "u.u_arg" is a pointer to this information;

3041: "access" returns a non-zero result if the file is not executable. The second condition examines whether the file is a directory or a special character file. (It would seem that by making this test earlier, e.g. just after line 3036, the efficiency of the code could be improved.);

3052: Copy the set of arguments from the user space into the temporary buffer;

3064: If the argument string is too large to fit in the buffer, take an error exit;

3071: If the number of characters in the argument string is odd, add an extra, null character;

3090: The first four words (8 bytes) of the named file are read into "u.u_arg". The interpretation of these words is indicated in the comment beginning on line 3076 and, more fully, in the section "A.OUT(V)" of the UPM.

Note the setting of "u.u_base", "u.u_count", "u.u_offset" and "u.u_segflg" preparatory to the read operation;

3095: If the text segment is not to be protected, add the text area size to the data area size, and set the former to zero;

3105: Check whether the program has a "pure" text area, but the program file has already been opened by some other program as a data file. If so, take the error exit;

3127: When this point is reached, the decision to execute the new program is irrevocable i.e. there is no longer the opportunity to return to the original program with an error flag set;

3129: "expand" here actually implies a major contraction, to the "per process data" area only;

3130: "xalloc" takes care of allocating (if necessary) and linking to the text area;

3158: The information stored in the buffer area is copied into the stack in the user data area of the new program;

3186: The locations in the kernel stack which contain copies of the "previous" values of the registers in user mode are set to zero, except for r6, the stack pointer, which was set at line 3155;

3194: Decrement the reference count for the "inode" structure;

3195: Release the temporary buffer;

3196: Wake up any other process waiting at line 3037.

12.8 fork (3322)

A call on "exec" is frequently preceded by a call on "fork". Most of the work for "fork" is done by "newproc" (1826), but before the latter is called, "fork" makes an independent search for a slot in the "proc" array, and remembers the place as "p2" (3327).

"newproc" also searches "proc" but independently. Presumably it always locates the same empty slot as "fork", since it does not report the value back. (Why is there no confusion on this point?)

3335: For the new process, "fork" returns the value of the parent's process identification, and initialises various accounting parameters;

3344: For the parent process, "fork" returns the value of the child's process identification, and **skips** the user mode program counter by one word.

Note that the values finally returned to a "C" program are slightly different from the above. Refer to the section FORK(II) of the UPM.

12.9 sbreak (3354)

This procedure implements system call #17 which is described in the Section "BREAK (II)" of the UPM. The comment at the head of the procedure has confused more than one reader: clearly the identifier "break" is used in "C" programs (leave an enclosing program loop) in an entirely different way from that intended here (change the size of the program data area).

"sbreak" has clear similarities with the procedure "grow" (4136) but unlike the latter, it is only invoked explicitly and may in fact cause a contraction of the data area as well as an expansion (depending on the new desired size).

3364: Calculate the new size for the data area (in 32 word blocks);

3371: Check that the new size is consistent with the memory segmentation constraints;

3376: The area is shrinking. Copy the stack area down into the former data area. Call "expand" to trim off the excess;

3386: Call "expand" to increase the total area. Copy the stack area up into the new part, and clear the areas which were formerly occupied by the stack.

The following procedures which are also contained in "sysl.c" are described in Chapter 13:

```
rexit (3205)     wait (3270)
exit  (3219)
```

12.10 The Files 'sys2.c' and 'sys3.c'

"sys2.c" and "sys3.c" are mainly concerned with the file system and input/output, and they have been relegated to Section Four of the operating system source code.

12.11 The File 'sys4.c'

All the procedures in this file implement system calls. The following procedures are described in Chapter 13:

```
ssig (3614)    kill (3630)
```

The following procedures are straightforward and have been left for the amusement and edification of the reader:

```
getswit (3413) sync (3486)
gtime (3420)   getgid (3472)
stime (3428)   getpid (3480)
setuid (3439)  nice (3493)
getuid (3452)  times (3656)
setgid (3460)  profil (3667)
```

The following procedures which are concerned with file systems, are described later:

```
unlink (3510)   chown (3575)
chdir (3538)    smdate (3595)
chmod (3560)
```

13 Software Interrupts

The principal concern of this chapter is the content of the file "sig.c", which appears on Sheets 39 to 42. This file introduces a facility for communication between processes. In particular it provides for the course of one "user mode" process to be interrupted, diverted or terminated by the action of another process or as the result of an error or operator action.

In this discussion the term "software interrupt" has been deliberately used in place of the term "signal". This latter has been eschewed because it has obtained connotations in the UNIX milieu which are rather different from the usage of ordinary English.

UNIX recognises 20 ("NSIG", line 0113) different types of software interrupts, of which (as the reader may discover for himself by perusal of the the Section "SIGNAL (II)" of the UPM) thirteen have standard names and associations. Interrupt type #0 is interpreted as "no interrupt".

Within the "per process data area" of each process is an array, "u.u_signal", of "NSIG" words. Each word corresponds to a different software interrupt type and defines the action which should be taken if the process encounters that kind of software interrupt:

u_signal[n]	when interrupt #n occurs
zero	the process will terminate itself;
odd non-zero	the software interrupt is ignored;
even non-zero	the value is taken as the address in user space of a procedure which which should be executed forthwith.

Interrupt type #9 ("SIGKILL") is especially distinguished because UNIX ensures that "u.u_signal[9]" remains zero until the very end of a process's existence, so that if a process is ever interrupted for that reason, it will always terminate itself.

13.1 Anticipation

Each process can set the contents of the array "u.u_signal[ ]" (with the exception of "u.u_signal[9]" as just noted) in anticipation of future interrupts so that the appropriate action is taken. The user programmer does this via the "signal" system call (see "SIGNAL (II)" of the UPM).

Thus if for example the programmer wishes to ignore software interrupts of type #2 (which result if the user hits the "delete" key on his terminal), he should set "u.u_signal[2]" to one by executing the system call

"signal (2,1);"

from his "C" program.

13.2 Causation

An interrupt is "caused" for a process quite simply by setting the value of "p_sig" (0363) in the process's "proc" entry, to the type number appropriate to the interrupt (i.e. a value in the range 1 to "NSIG"–1).

"p_sig" is always directly accessible, even when the affected process and its "per process data area" have been swapped out to disk. Obviously this mechanism only allows one interrupt per process to be outstanding at any one time. The outstanding interrupt will always be the most recent one, unless one of the interrupts was of type #9, which always prevails.

13.3 Effect

The effect of a software interrupt never takes place immediately. It may occur after only some slight delay if the affected process is currently running, or possibly after a considerable delay if the affected process is suspended and has been swapped out.

The action dictated by the interrupt is always inflicted on the affected process by **itself**, and hence can only occur when the affected process is active.

Where the effect is to execute a user defined procedure, the kernel mode process adjusts the user mode stack to make it appear that the procedure had been entered and immediately interrupted (hardware style) before executing the first instruction. The system then returns from kernel mode to user mode in the usual manner. The result of all this is that the next user mode instruction which is executed is the first instruction of the designated procedure.

13.4 Tracing

The software interrupt facility has been extended to provide a powerful but somewhat inefficient mechanism whereby a parent process may monitor the progress of one or more child processes.

13.5 Procedures

Since the interrelationships of the procedures associated with software interrupts are somewhat confusing at first sight, it is worthwhile introducing the procedures briefly before plunging in with both feet ....

13.6 A. Anticipation

"ssig" (3614) implements system call #48 ("signal") to set the value in one element of the array "u.u_signal".

13.7 B. Causation

"kill" (3630) implements system call #37 (kill) to cause a specified interrupt to a process defined by its process identifying number.

"signal" (3949) causes a specified interrupt to be caused for all processes controlled and/or initiated from a specified terminal.

"psignal" (3963) is called by "kill" (3649) and "signal" (3955) (also "trap" (2793, 2818) and "pipe" (7828)) to do the actual setting of "p_sig".

13.8 C. Effect

"issig" (3991) is called by "sleep" (2085), "trap" (2821) and "clock" (3826) to enquire whether there is an outstanding non-ignorable software interrupt for the active process just waiting to happen.

"psig" (4043) is called whenever "issig" returns a non-zero result (except in "sleep" where things are a little more complex) to implement the action triggered by the interrupt.

"core" (4094) is called by "psig" if a core dump is indicated for a terminating process.

"grow" (4136) is called by "psig" to enlarge the user mode stack area if necessary.

"exit" (3219) terminates the currently active process.

13.9 D. Tracing

"ptrace" (4164) implements the "ptrace" system call #26.

"stop" (4016) is called by "issig" (3999) for a process which is being traced to allow the supervising parent to have a "look-see".

"procxmt" (4204) is a procedure called from "stop" (4028) whereby the child carries out certain operations related to tracing, at the behest of the parent.

13.10 ssig (3614)

This procedure implements the "signal" system call.

3619: If the interrupt reason is out of range or is equal to "SIGKILL" (9), take an error exit;

3623: Capture the initial value in "u.u_signal[a]" for return as the result of the system call;

3624: Set the element of "u.u_signal" to the desired value ...

3625: If an interrupt for the current reason is pending, cancel it. (It is not clear why this step should be necessary or even desirable. Any suggestions??)

13.11 kill (3630)

This procedure implements the "kill" system call to cause a specified type of software interrupt to another designated process.

3637: If "a" is non-zero, it is the process identifying number of a process to be interrupted. If "a" is zero, then all processes originating from the same terminal as the current process are to be interrupted;

3639: Consider each entry in the "proc" table in turn and reject it if: it is the current process (3640); it is not the designated process (3642); no particular process was designated ("a" == 0) but it does not have the same controlling terminal or it is one of the two initial processes (3644); the user is not the "super user" and the user identities do not match (3646);

3649: For any process that survives the above tests, call "psignal" to change "p_sig".

13.12 signal (3949)

For every process, if it is controlled by the specified terminal (denoted by "tp"), hit it with "psignal".

13.13 psignal (3963)

3966: Reject the call if "sig" is too large (but why not if negative?? "kill" does not check this parameter before passing it to "psignal". Admittedly the "kill" command could only result in a positive value for "sig" ...);

3971: If the current value of "p_sig" is NOT set to "SIGKILL", then overwrite it (i.e. once a process has been "killed outright" there is no way to revive it.);

3973: Seems to be an error here ... for "p_stat" read "p_pri" ... improve the priority of the process if it is not too good;

3975: If the process is waiting for a non-kernel event i.e. it called "sleep" (2066) with a positive priority, then set it running again.

13.14 issig (3991)

3997: If "p_sig" is non-zero, then ...

3998: If the "tracing" flag is on, call "stop" (this topic will be resumed later);

4000: Return a zero value if "p_sig" is zero. (This apparently redundant test is necessary because "stop" may reset "p_sig" as a side effect.);

4003. If the value in the corresponding element of "u.u_signal" is even (may be zero) return a non-zero value;

4006: Otherwise return a zero value.

The comment regarding the frequency of calls on "issig" which occurs on lines 3983 to 3985 needs some clarification. At least one call on "issig" is a part of every execution of "trap" but only of one interrupt routine ("clock", which calls "issig" only once per second). In cases where "pri" is positive, "sleep" (2073, 2085) calls "issig" before and after calling "swtch".

13.15 psig (4043)

This procedure is only called if "u.u_signal[n]" was found by "issig" to have an even value. If this value is found (4051) to be non-zero, it is taken as the address of a user mode function which has to be executed.

4054: Reset "u.u_signal[n]" except the case where the interrupt for an illegal instruction or trace trap;

4055: Calculate the user space addresses of the lower of two words which are to be inserted into the user mode stack ...

4056: Call "grow" to check the current user mode stack size, and to extend it (downwards!) if necessary;

4057: Put the values of the processor status register and the program counter which were captured at the time of the "trap" or hardware interrupt (in the case of a "clock" interrupt) into the user stack, and update the "remembered" values of r6, r7 and the processor status word. Upon returning to user mode, execution will resume at the beginning of the designated procedure. When this procedure returns, the procedure which was originally interrupted will be resumed;

4066: If "u.u_signal[n]" is zero, then for the interrupt types listed, generate a core image via the procedure "core";

4079: Store a value in "u.u_arg[0]" composed of the low order byte of the remembered value of r0, and of "n", which records the interrupt type and whether a core image was successfully created;

4080: Call "exit" for the process to terminate itself.

13.16 core (4094)

This procedure copies the swappable program image into a file called "core" in the user's current directory. A detailed explanation of this procedure must wait until the material of Sections Three and Four, which deal with input/output and file systems, have been covered.

13.17 grow (4136)

The parameter, "sp", of this procedure defines the address of a word which should be included in the user mode stack.

4141: If the stack already extends far enough, simply return with a zero value.

Note that this test relies on the idiosyncrasies of 2's complement arithmetic, and if both

$$|\mathrm{sp}| > 2^{15}$$

and

$$|\mathrm{u.u\_size} * 64| > 2^{15}$$

the decision to extend the stack may be taken wrongly at this juncture;

4143: Calculate the stack size increment needed to include the new stack point plus a 20*32 word margin;

4144: Check that this value is in fact positive (i.e. we are not dealing with a failure of the test on line 4141.);

4146: Check that the new stack size does not conflict with the memory segmentation constraints ("estabur" sets "u.u_error" if they do) and reset the segmentation register prototypes;

4148: Get a new, enlarged data area, copy the stack segments (32 words at a time) into the high end of the new data area, and clear the segments which now become the stack expansion;

4156: Update the stack size, "u.u_ssize" and return a "successful" result.

13.18 exit (3219)

This procedure is called when a process is to terminate itself.

3224: Reset the "tracing" flag;

3225: Set all of the values in the array "u.u_signal" (including "u.u_signal[SIGKILL]") to one so that no future execution of "issig" will ever be followed by execution of "psig";

3227: Call "close" (6643) to close all the files which the process has open. (For the most part, "closing" simply involves decrementing a reference count.);

3232: Reduce the reference count for the current directory;

3233: Sever the process's connection with any text segment;

3234: A place is needed to store "per process" information until the parent process can look at it. A block (256 words) in the swap area of the disk is a convenient place;

3237: Find a suitable buffer (256 words) and ...

3238: Copy the **lower half** of the "u" structure into the buffer area;

3239: Write the buffer into the swap area;

3241: Enter the core space occupied by the process into the free list. (This space is of course still in use, but the use will terminate before any other process gets to dip into the free list again. This could not be done any sooner, because, as will be seen later, both "getblk" and "bwrite" can call "sleep", during which all sorts of things might happen. In view of all this, it might be reasonable if the statement

expand (USIZE);

were inserted after line 3226.);

3243: Set the process state to "zombie" (i.e. "a corpse said to be revived by witchcraft" (O.E.D.));

3245: The remaining code searches the "proc" array to find the parent process and to wake it up, to make any children "wards of the state", and, if they have "stopped" for tracing, to release them. Finally the code includes (for this process) a last call on "swtch".

Before going on to consider tracing, there are two routines which are closely associated with "exit", which can be conveniently disposed of now.

13.19 rexit (3205)

This procedure implements the "exit" system call, #1. It simply salvages the low order byte of the user supplied parameter and saves it in "u.u_arg[0]". which is in the lower half of the "u" structure i.e. the part that is written to the "swap area" as a "zombie".

13.20 wait (3270)

For every call on "exit", there should be a matching call on "wait" by an anxious parent or ancestor. The principal function of the latter procedure, which implements the "wait" system call, is for the parent or ancestor to find and dispose of a "zombie" child.

"wait" also has a secondary function, to look for children which have "stopped" for tracing (which is the next major topic).

3277: Search the whole "proc" array looking for child processes. (If none exist, take an error exit (line 3317));

3280: If the child is a "zombie":

- save the child's process identifying number, to report back to the parent;
- read the 256 word record back from the disk swap area, and release the swap space;
- reinitialise the "proc" array entry;
- accumulate the various accounting entries;

  save the "u_arg[0]" value also to report back to the parent;

3300: Is the child in a "stopped" state? (If so, wait for the discussion on tracing);

3313: If one or more children were found but none were "zombies" or "stopped", "sleep" and then look again.

13.21 Tracing

The tracing facilities are provided through a modification and extension of the software interrupt facilities. Briefly, if a parent process is tracing the progress of child process, every time the child process encounters a software interrupt, the parent process is given the opportunity to intervene as part of the total response to the interrupt.

The parent's intervention may involve interrogation of values within the child process's data areas, including the "per process data area". Subject to certain constraints, the parent process may also change values within these data areas.

The source of the software interrupts may be the parent process, the user himself (e.g. by entering "kill" commands or "delete"s through his terminal) or the child process itself (e.g. instructions or other maladies).

The communication between child and parent processes is a kind of ritual dance:

(1) the child experiences a software interrupt and "stops";

(2) the waiting parent discovers the "stopped" child (line 3301), and revives. Subsequently ...

(3) the parent may execute the "ptrace" system call which has the effect of leaving a request message in the system defined structure "ipc" (3939) for the child process;

(4) the parent then goes to "sleep" while the child "wakes up";

(5) the child reads the message in "ipc" and acts upon it (e.g copying one of its own values into "ipc.ip_data");

(6) the child then goes to "sleep" while the parent "wakes up";

(7) the parent inspects the result, as recorded in "ipc", of the operation;

(8) steps (3) to (7) may be repeated several times in succession.

Finally the parent may allow the child to continue its normal execution, possibly without ever knowing that a software interrupt had occurred.

A discussion of the tracing facility is contained in the Section "PTRACE (II)" of the UPM. To the list of functional limitations noted in the "Bugs" paragraph, we can add the following comments on efficiency:

- There should be a mechanism for transferring large blocks (e.g. up to 256 words at a time) of information from the child to the parent (though not necessarily in the reverse direction);

- There should be a proper coroutine procedure (analogous to "swtch") to allow rapid transfer of control between child and parent.

13.22 stop (4016)

This procedure is called by "issig" (3999) if the tracing flag ("STRC", 0395) is set.

4022: If your parent is process 1 (i.e. "/etc/init"), then call "exit" (line 4032);

4023: Otherwise look through "proc" for your parent ... wake him up ... declare yourself "stopped" and ... call "swtch" (Note do NOT call "sleep". Why?);

4028: If the tracing flag has been reset, or the result of the procedure "procxmt" is true, return to "issig";

4029: Otherwise start again.

13.23 wait (3270) (continued)

3301: If the child process has "stopped" and ...

3302: If the "SWTED" flag is not set (i.e. the parent hasn't noticed this child lately) ...

3303: As an "aide-memoire" set the "SWTED" flag. Set "u.u_ar0[R0]", "u.u_ar0[R1]" so that the child process status word is returned to the parent;

3309: The "SWTED" flag was set. This means that the parent, by performing at least two "waits" in succession without any intervening call on "ptrace", is not very interested in the child. So reset both the "STRC" and the "SWTED" flags and release the child. (Note the use of "setrun" (not "wakeup") to complement the call on "swtch" (4027)).

13.24 ptrace (4164)

This procedure implements the "ptrace" system call, #26.

4168: "u.u_arg[2]" corresponds to the first parameter in the "C" program calling sequence. If this is zero, a child process is asking to be traced by its parent, so set the "STRC" flag and return.

Note that this code handles the only explicit action the child process is asked to take with respect to tracing. There is no real reason why even this action should be taken by the child process and not by the parent process. From a security point of view it is most probably desirable that a child process should only be traceable if it gives its permission. On the other hand, if the child asks to be traced and is then ignored by the parent, the child process may be blocked indefinitely. Perhaps the best solution would be for the "STRC" flag to be set only after explicit action by **both** the parent **and** the child.

4172: Search the "proc" table looking for a process which: is stopped; matches the given process identifying number; is a child of the current process;

4181: Wait for the "ipc" structure to become available if it is currently in use;

4183: Copy the parameters into "ipc" ...

4187: reset the "SWTED" flag, and ...

4188: return the child to a "ready to run" state;

4189: Sleep until "ipc.ip_req" is nonpositive (4212);

4191: Extract a value that is to be returned to the parent process, check for errors, unlock "ipc" and "wake up" any processes waiting for "ipc".

Note that the "sleeps" on lines 4182, 4190 are for essentially different reasons, and could be differentiated to good effect by replacing "&ipc" by "&ipc.ip_req" on lines 4190 and 4213.

13.25 procxmt (4204)

This procedure is executed by the child process under the influence of data left by the parent in the ipc structure.

4209: If "ipc.ip_lock" is set wrongly for the current process, then certainly the rest of "ipc" should be ignored.

After "stop" (4027) calls "swtch", the chide process is restarted by one of three calls on "setrun" which leave the "STRC" and "SWTED" flags in the state indicated:

		STRC	SWTED	ipc.ipc_lock
exit	(3254)	set	set	arbitrary
wait	(3310)	reset	reset	arbitrary
ptrace	(4188)	set	reset	properly set

In the third case "ptrace" will always set "ipc.ip_lock" properly, before the child is restarted, so that there is then no chance of the test on 4209 into "ipc" failing.

In the second case, where the parent has ignored the child, "procxmt" will never in fact be called.

By executing the statement "return(0)"; on line 4210, "procxmt" forces "stop" to loop back to line 4020. In the case where the parent has already died, the test on line 4022 will then fail, and a call on "exit" (4032) will result.

4211: Store the value of "ipc.ip_req" before resetting the latter, "wake up" the parent, and select the next action as indicated.

The various actions are adequately explained in Section "PTRACE (II)" of the UPM, with the one qualification that cases 1, 2 and 4, 5 are documented the wrong way around (i.e. "I" and "D" spaces respectively, not "D" and "I"!).

Section Three

Section Three is concerned with basic input/output operations between the main memory and disk storage.

These operations are fundamental to the activities of program swapping and the creation and referencing of disk files.

This section also introduces procedures for the use and manipulation of the large (512 byte) buffers.

14 Program Swapping

UNIX, like all time-sharing systems, and some multiprogramming systems uses "program swapping" (also called "rollin/roll-out") to share the limited resource of the main physical memory among several processes.

Processes which are suspended may be selectively "swapped out" by writing their data segments (including the "per process data") into a "swap area" on disk

The main memory area which was occupied can then be reassigned to other processes, which quite probably will be "swapped in" from the "swap area".

Most of the decisions regarding "swapping out", and all the decisions regarding "swapping in", are made by the procedure "sched". "Swapping in" is handled by a direct call (2034) on the procedure "swap" (5196), whereas "swapping out" is handled by a call (2024) on "xswap" (4368).

For those archaeologists who like to ponder the "bones" of earlier versions of operating systems, it seems that originally "sched" called "swap" directly to "swap out" processes, rather than via "xswap". The extra procedure (one of several to be found in the file "text.c") has been necessitated by the implementation of the sharable "text segments".

It is instructive to estimate how much extra code has been necessitated by the text segment feature: in "text.c" are four procedures "xswap", "xalloc", "xfree" and "xccdec", which manipulate an array of structures called "text", which is declared in the file "text.h". Additional code has also been added to "sysl.c" and "slp.c".

14.1 Text Segments

Text segments are segments which contain only "pure" code and data i.e. code and data which remain unaltered throughout the program execution, so that they may be shared amongst several processes executing the same program.

The resulting economies in space can be quite substantial when many users of the system are executing the same program simultaneously e.g. the editor or the "shell".

Information about text segments must be stored in a central location, and hence the existence of the "text" array. Each program which shares a text segment keeps a pointer to the corresponding text array element in "u.u_textp".

The text segment is stored at the beginning of the code file. The first program to begin execution causes a copy of the text segment to be made in the "swap" area.

When subsequently no programs are left which reference the text segment, the resources absorbed by the text segment are released. The main memory resource is released whenever there are no programs which reference the text segment currently in main memory; the "swap" area is released in general whenever there are no programs left running which reference the text segment.

The numbers in each of these states are denoted by "x_ccount" and "x_count" respectively. Decrementing these numbers is handled by the routines "xccdec" and "xfree" which also take care of releasing resources when the counts reach zero. ("xccdec" is called whenever a program is swapped out or terminates. "xfree" is called by "exit" whenever a program terminates.)

14.2 sched (1940)

Process #0 executes "sched". When it is not waiting for the completion of an input/output operation that it has initiated, it spends most of its time waiting in one of the following situations:

A. (runout) None of the processes which are swapped out is ready to run, so that there is nothing to do. The situation may be changed by a call to "wakeup", or to "xswap" called by either "newproc" or "expand".

B. (runin) There is at least one process swapped out and ready to run, but it hasn't been out more than 3 seconds and/or none of the processes presently in main memory is inactive or has been there more than 2 seconds. The situation may be changed by the effluxion of time as measured by "clock" or by a call to "sleep".

When either of these situations terminate:

1958: With the processor running at priority six, so that the clock can't interrupt and change values of "p_time", a search is made for the process which is ready to run and has been swapped out for the longest time;

1966: If there is no such process then situation A holds;

1976: Search for a main memory area of adequate size to hold the data segment. If an associated text segment must be present also but is not currently in main memory, the area is increased by the size of the text segment;

1982: If an area of adequate size is available the program branches to "found2" (2031). (Note that the program does not handle the case where there is sufficient space for both text and data segments but in distinct areas of main memory. Would it be worth while to extend the code to cover this possibility?);

1990: Search for a process which is in main memory, but which is not the scheduler or locked (i.e. already being swapped out), and whose state is "SWAIT" or "SSTOP" (but **not** "SSLEEP") (i.e. the process is waiting for an event of low precedence, or has stopped during tracing (see Chapter Thirteen)). If such a process is found, go to line 2021, to swap the image out.

Note that there seems to be a bias here against processes whose "proc" entries are early in the "proc" array;

2003: If the image to be swapped in has been out less than 3 seconds, then situation B holds;

2005: Search for the process which is loaded, but is not the scheduler or locked, whose state is "SRUN" or "SSLEEP" (i.e. ready to run, or waiting for an event of high precedence) and which has been in main memory for the longest time;

2013: If the process image to be swapped out has been in main memory for less than 2 seconds, then situation B holds.

The constant "2" here (also the "3" on line 2003) is somewhat arbitrary. For some reason the programmer has departed from his usual practice of naming such constants to emphasise their origins;

2022: The process image is flagged as not loaded and is swapped out using "xswap" (4368).

Note that the "SSWAP" flag is not set here because the process swapped out is not the current process. (Cf. lines 1907, 2286);

2032: Read the text segment into main memory if necessary. Note that the arguments for the "swap" procedure are:

- an address within the swap area of the disk;
- a main memory address (ordinal number of a 32 word block);
- a size (number of 32 word blocks to be transferred);
- a direction indicator ("B_READ==1" denotes "disk to main memory");

2042: Swap in the data segment and ...

2044: Release the disk swap area to the available list, record the main memory address, set the "SLOAD" flag and reset the accumulated time indicator.

14.3 xswap (4368)

4373: If "oldsize" data was not supplied, use the current size of the data segment stored in "u";

4375: Find a space in the disk swap area for the process's data segment. (Note that the disk swap area is allocated in terms of 512 character blocks);

4378: "xccdec" (4490) is called (unconditionally!) to decrease the count, associated with the text segment, of the number of "in main memory" processes which reference that text segment. If the count becomes zero, the main memory area occupied by the text segment is simply returned to the available space. (There is no need to copy it out, since, as we shall see, there will be a copy already in the disk swap area);

4379: The "SLOCK" flag is set while the process is being swapped out. This is to prevent "sched" from attempting to "swap out" a process which is already in the process of being "swapped out". (This can only happen if "swapping out" was started initially by some routine other than "sched" e.g. by "expand");

4382: The main memory image is released except when "xswap" is called by "newproc";

4388: If "runout" is set, "sched" is waiting for something to "swap in", so wake it up.

14.4 xalloc (4433)

"xalloc" is called by "exec" (3130), when a new program is being initiated, to handle the allocation of, or linking to, the text segment. The argument, "ip", is a pointer to the "mode" of the code file. At the time of this call, "u.u_arg[1]" contains the text segment size in bytes.

4439: If there is no text segment, return immediately;

4441: Look through the "text" array for both an unused entry and an entry for the text segment. If the latter can be found, do the bookkeeping and go to "out" (4474);

4452: Arrange to copy the text segment into the disk swap area. Initialise the unused text entry, and get space in the disk swap area;

4459: Change the space occupied by the process to one large enough to contain the "per process data" area and the text segment;

4460: The call on "estabur" is necessary to set the user mode segmentation registers before reading the code file;

4461: A UNIX process can only initiate one input/output operation at a time. Hence it is possible to store i/o parameters at standard locations in the "u" structure, viz. "u.u_count", "u.u_offset[ ]" and "u.u_base";

4462: The octal value 020 (decimal 16) is an offset into the code file;

4463: Information is to be read into the area beginning at location zero in the user address space;

4464: Read the text segment part of the code file into the current data segment;

4467: "Swap out" the data segment (minus the "per process data") into the disk swap area reserved for the text segment;

4473: "Shrink" the data segment – it is about to be swapped out;

4475: "sched" always "swaps in" the text segment before the data segment i.e. there is no mechanism for bringing the text segment into main memory once the data segment is present. If the text segment is not in main memory, get back into step by "swapping out" the data segment to disk.

It will be noted that the code to handle text segments is very conservative whenever the situation starts to get complicated. For example, the "panic" (4451) when no more text entries are available would seem to be a rather extreme reaction. However the strategy of being generous with "text" array space is quite likely to be less expensive than the code needed to do "better". What do you think?

14.5 xfree (4398)

"xfree" is called by "exit" (3233); when a process is being terminated, and by "exec" (3128), when a process is being transmogrified.

4402: Set the text pointer in the "proc" entry to "NULL";

4403: Decrement the main memory count and if it is now zero ...

4406: and if the text segment has not been flagged to be saved, ...

4408: Abandon the image of the text segment in the disk swap area;

4411: Call "iput" (7344) to decrement the "inode" reference count and if necessary delete it.

"ISVTX" (5695) is a mask which defines the "sticky bit" mentioned in section "CHMOD(I)" of the UPM. If this bit is set, the disk copy of the text segment is allowed to remain in the disk swap area even when no programs are running which reference it, in the expectation that it will be required again shortly. This is an efficient device for commonly used programs such as the "shell" or the editor.

15 Introduction to Basic I/O

There are three files whose contents need to be thoroughly absorbed before the subject of UNIX input/output is broached in detail.

15.1 The File 'buf.h'

This file declares two structures called "buf" (4520) and "devtab" (4551). Instances of the structure "buf" are declared as 'bfreelist (4567) and as the array "buf" (!) (4535) with "NBUF" elements.

The structure "buf" is possibly misnamed because it is in fact a **buffer header** (or buffer control block). The buffer areas proper are allocated separately and declared (4720) as

```
''char buffers [NBUF] [514];''
```

Pointers from the "buf" array to the "buffers" array are set up by the procedure "binit".

Other instances of the structure "buf" are declared as "swbuf" (4721) and "rrkbuf" (5387). No 514 character buffer areas are associated with "bfreelist" or "swbuf" or "rrkbuf".

The "buf" structure may be divided into three parts:

(a) flags These convey status information and are contained within a single word. Masks for setting these flags are defined as "B_WRITE", "B_READ" etc. in lines 4572 to 4586.

(b) list pointer Forward and backward pointers for two doubly linked lists, which we shall refer to as the "b"-list and the "av"-list.

(c) i/o parameters A set of values associated with the actual data transfer.

15.2 devtab (4551)

The "devtab" structure has five words, the last four of which are forward and backward pointers.

One instance of "devtab" is declared within the device handler for each block type of peripheral device. For our model system the only block device is the RK05 disk, and "rktab" is declared as a "devtab" structure at line 5386.

The "devtab" structure contains some status information for the the device and serves as a list head for:

(a) the list of buffers associated with the device, and simultaneously on the "av"-list;

(b) the list of outstanding i/o requests for the device.

15.3 The File 'conf.h'

The file "conf.h" declares:

- yet another way to dissect an integer into two parts ("d_minor" and "d_major"). Note that "d_major" corresponds to "hibyte" (0180);
- two arrays of structures;
- two integer variables, "nlkdev" and "nchrdev".

The two arrays of structures, "bdevsw" and "cdevsw", are declared but not dimensioned or initialised in "conf.h". The initialisation of these arrays is performed in the file "conf.c".

15.4 The File 'conf.c'

This file, along with "low.s", is generated individually at each installation (to reflect the set of peripherals actually installed) by the program "mkconf". (In our case, "conf.c" reflects the representative devices for our model system.)

This file initialises the following:

```
bdevsw  (4656)   swapdev (4696)
cdevsw  (4663)   swplo   (4637)
rootdev (4635)   nswap   (4698)
```

15.5 System Generation

System generation at a UNIX installation consists mainly of:

- running "mkconf" with appropriate input;
- recompiling the output files (created as "c.c" and "l.s");
- reloading the system with the revised object files.

This process only takes a few minutes (not the several hours of some other operating systems). Note that "bdevsw" and "cdevsw" are defined differently in "conf.c" from elsewhere, namely as a one dimensional array of pointers to functions which return integer values. This quietly ignores the fact that, for example, "rktab" is not a function, and relies on the linking program not to enquire too closely into the nature of the work which it is performing.

15.6 swap (5196)

Before plunging into all the detail of the file "bio.c", it will be instructive as well as convenient to examine one routine which was introduced earlier, namely "swap".

The buffer head "swbuf" was declared to control swapping input/output, which must share access to the disk with other activity. No element of "buffers" is associated with "swbuf". Instead the core area occupied (or to be occupied) by the program serves as the data buffer.

5200: The address of the flags in "swbuf" is transferred to the register variable "fp" for convenience and economy;

5202: The "B_BUSY" flag is tested, and if it is on, a swap operation is already under way, so that the "B_WANTED" flag is set and the process must wait via a call on "sleep".

Note that the code loop on lines 5202 to 5205 runs at priority level six, i.e. one higher than the disk interrupt priority.

Can you see why this is necessary? Under what conditions will the "B_BUSY" flag be set?

5206: The flags are set to reflect:

- "swbuf" is in use ("B_BUSY");
- physical i/o implying a large transfer direct to/from the user data segment ("B_PHYS");
- whether the operation is read or write. ("rdflg" is a parameter to "swap");

5207: The "b_dev" field is initialised. (Presumably this could have been performed once during initialisation rather than every time "swbuf" is used, i.e. in "binit".);

5208: "b_wcount" is initialised. Note the negative value and the effective multiplication by 32;

5210: The hardware device controller requires a full physical address (18 bits on the PDP 11/40). The block number of a 32 word block must be converted into two parts: the low order ten bits are shifted left six places and stored as "b_addr", and the remaining six high order bits as "b_xmem". (On the PDP 11/40 and 11/45 only two of these bits are significant.);

5212: A mouthful at first glance! Shift "swapdev" eight places to the right to obtain the major device number. Use the result to index "bdevsw". From the structure thus selected, extract the strategy routine and execute it with the address of "swbuf" passed as a parameter;

5213: Explain why this call on "spl6" is necessary;

5214: Wait until the i/o operation is complete. Note that the first parameter to "sleep" is in effect the address of "swbuf";

5216: Wakeup those processes (if any) which are waiting for "swbuf";

5218: Reset the process or priority to zero, thus allowing any pending interrupts to "happen";

5219: Reset both the "B_BUSY" and "B_WANTED" flags.

15.7 Race Conditions

The code for "swap" has a number of interesting features. In particular it displays in microcosm the problems of race conditions when several processes are running together.

Consider the following scenario:

No swapping is taking place when process A initiates a swapping operation. Denoting "swbuf.b_flags" by simply "flags", we have initially

```
flags == null
```

Process A is not delayed at line 5204, initiates its i/o operation and goes to sleep at line 5215. We now have

```
flags == B_BUSY | B_PHYS | rdflg
```

which was set at line 5206.

Suppose now while the i/o operation is proceeding, process B also initiates a swapping operation. It too begins to execute "swap", but finds the "B_BUSY" flag set, so it sets the "B_WANTED" flag (5203) and goes to sleep also (5204). We now have

```
flags == B_BUSY | B_PHYS | rdflg | B_WANTED
```

At last the i/o operation completes. Process C takes the interrupt and executes "rkintr", which calls (5471) "iodone" which calls (5301) "wakeup" to awaken process A and process B. "iodone" also sets the "B_DONE" flag and resets the "B_WANTED" flag so that

```
flags == B_BUSY | B_PHYS | rdflg | B_DONE
```

What happens next depends on the order in which process A and process B are reactivated. (Since they both have the same priority, "PSWP", it is a toss-up which goes first.)

Case (a): Process A goes first. "B_DONE" is set so no more sleeping is needed. "B_WANTED" is reset so there is no one to "wakeup". Process A tidies up (5219), and leaves "swap" with

```
flags == B_PHYS | rdflg | B_DONE
```

Process B now runs and is able to initiate its i/o operation without further delay.

Case (b): Process B goes first. It finds "B_BUSY" on, so it turns the "B_WANTED" flag back on, and goes to sleep again, leaving

```
flags == B_BUSY | B_PHYS | rdflg |
         B_DONE | B_WANTED
```

Process A starts again as in Case (a), but this time finds "B_WANTED" on so it must call "wakeup" (5217) in addition to its other chores. Process B finally wakes again and the whole chain completes.

Case (b) is obviously much less efficient than case (a). It would seem that a simple change to line 5215 to read

```
sleep (fp, PSWP-1);
```

would cost virtually nothing and ensure that Case (b) never occurred!

The necessity for the raising of processor priority at various points should be studied: for example if line 5201 was omitted and if process B had just completed line 5203 when the "i/o complete" interrupt occurred for Process A's operation, then "iodone" would turn off "B_WANTED" and perform "wakeup" before process B went to sleep ... forever! A bad scene.

15.8 Reentrancy

Note also the assumption made above, that both process A and process B could execute "swap" simultaneously. All UNIX procedures are in general "re-entrant" (which means multiple simultaneous executions are possible). How would UNIX have to change if re-entrancy were not allowed?

15.9 For the Uninitiated

We can now return to complete an investigation started in Chapter Eight concerning "aretu" and "u.u_ssav":

After setting "u.u_ssav" (2284), "expand" calls (2285) "xswap", which calls (4380) "swap", which calls (5215) "sleep", which calls (2084) "swtch", which **resets** "u.u_rsav" (2189).

Thus in fact "u.u_rsav" finally gets reset to a value appropriate to four procedure calls deeper than that for "u.u_ssav".

15.10 Additional Reading

The article "The UNIX I/O System" by Dennis Ritchie is highly pertinent.

16 The RK Disk Driver

The RK disk storage system employs a removable disk cartridge containing a single disk, which is mounted inside a drive with moving read/write heads.

The device designated RK11-D consists of a disk controller together with a single drive. Additional drives, designated RK05, up to a total of seven, may be added to a single RK11-D.

A requirement for more than eight drives would require an additional controller with a different set of UNIBUS addresses. Also the code in the file "rk.c" would have to be modified to handle the case of two or more controllers. This case is most unlikely because requirements for large amounts of online disk storage will be more economically provided otherwise e.g. by the RP04 disk system.

Cartridge capacity:	1,228,800 words
	(4800 512 byte records)
Surfaces/cartridge:	2
Tracks/surface:	200 (plus 3 spare)
Sectors/Track:	12
Words/Sector:	256
Recording density:	2040 bpi maximum
Rotation speed:	1500 rpm
Half revolution:	20 msecs
Track positioning:	
	10 msecs (one track)
	50 msecs (average)
	85 msecs (worst case)
Interrupt Vector Address:	220
Priority Level:	5

Unibus Register Addresses

Drive Status	RKDS	777400
Error	RKER	777402
Control Status	RKCS	777404
Word Count	RKWC	777406
Current bus address	RKBA	777410
Disk address	RKDA	777412
Data Buffer	RKDB	777416

Table 16.1 RK Vital Statistics

The average total access time is 70 milllseconds. With multi-drive subsystems, seeking by one drive may be overlapped with reading or writing by another drive. However this feature is not used by UNIX because of bugs which existed at one time in the hardware controller.

In initiating a data transfer, RKDA, RRBA and RKC are set, and then RKCS is set. Upon completion, status information is available in RKCS, RRER and RKDS. When an error occurs, UNIX simply calls "deverror" (2447) to display RKER and RKDS on the system console, without any attempt at analysis. An operation is repeated up to ten times before an error is reported by the device driver.

The register formats which are described fully in the "PDP11 Peripherals Handbook" are reflected in the program code at several points. The following summaries suffice to describe the features used by UNIX:

Control Status Register (RKCS)

bit	description
15	Set when any bit of RKER (the Error Register) is set;
7	Set when the control is no longer engaged in actively executing a function and is ready to accept a command;
6	When set, the control will issue an interrupt to vector address 220 upon operation completion or error;
5–4	Memory Extension. The two most significant bits of the 13 bit physical bus address. (The other 16 bits are recorded in RKBA.);
3–1	Function to be performed: CONTROL RESET: 000 WRITE: 001 READ: 010 etc.,
0	Initiate the function designated by bits 1 to 3 when set. (write only);

Word Count Register (RKWC)

Contains the twos complement of the number of words to be transferred.

Disk Address Register (RKDA)

bit	description
15–13	Drive number (0 to 7)
12–5	Cylinder number (0 to 199)
4	Surface number (0,1)
3-0	Sector address (0 to 11)

16.1 The file 'rk.c'

This file contains the code which is specific to the RK disk system, i.e. which is the RK "device driver".

16.2 rkstrategy (5389)

The strategy routine is called, e.g. from "swap" (5212), to handle both read and write requests.

5397: The test and call on "mapalloc" here is a "no-op" except on the PDP11/70 system;

5399: The code from here to line 5402 appears to be unnecessarily devious! See the discussion of "rkaddr" below. If the block number is too large, set the "B_ERROR" flag and report "completion";

5407: Link the buffer into a FIFO list for the controller. The list is singly linked, uses the "av_forw" pointer of the "buf" structures, and has head and tail pointers in "rktab". Interrupts from disk devices may not be allowed after the first step;

5414: If the RK controller is not currently active, wake it up via a call on "rkstart" (5440), which checks that there is something to do (5444), flags the controller as busy (5446) and calls "devstart" (5447), passing as parameters:

- a pointer to the first enqueued buffer header;
- the address of the RKDA disk address register. (The value passed is in effect 0177412. See lines 5363, 5382.);
- a "disk address" computed by "rkaddr";
- zero (not really important in our discussion, and may be ignored).

16.3 rkaddr (5420)

The code in this procedure incorporates a special feature for files which extend over more than one disk drive. This feature is described in the UPM Section "RK(IV)". Its usefulness seems to be restricted.

The value returned by "rkaddr" is formatted for direct transmission to the control register, RKDA.

16.4 devstart (5096)

This procedure when called for the RK disk loads appropriate values into the registers RKDA, RKBA, RKWC and RKCS in succession. Only the last value needs to be computed at this stage.

The calculation, though messy in appearance, is straight forward. Note that "hbcom" is zero and "rbp->b_xmem" contains the two high order bits of the physical core address. The loading of RKCS initialises the disk controller i.e. the operation is now entirely under the control of the hardware.

"devstart" returns to "rkstart" (5448), which returns to "rkstrategy" (5416), which resets the processor priority and returns to "swap" (5213), which ...

16.5 rkintr (5451)

This procedure is invoked to handle the interrupts which occur when RK disk operations are completed.

5455: Check for a false alarm!

5459: Inspect the error bit; if set ...

5460: Call "deverror" (2447) to display a message on the system console terminal;

5461: Clear the internal registers of the disk controller and ...

5462: Wait till this is completed (usually a few microseconds);

5463: If the operation has been retried less than ten times, call "rkstart" to try again. Otherwise give up and report an error;

5469: Set the "retry" (!) count back to zero, remove the current operation from the "actf" list, and complete the operation by calling "iodone";

5472: "rkstart" is called unconditionally here. If the call is not necessary (because the "actf" list is empty) "rkstart" will return immediately (5444).

16.6 iodone (5018)

This routine is primarily concerned with the return of resources when a block i/o operation has completed. It:

- frees up the Unibus map (for 11/70's, if appropriate);
- sets the "B_DONE" flag;
- releases the buffer if the i/o was asynchronous, or else resets the "B_WANTED" flag and wakes up any process waiting for the i/o operation to complete.

17 Buffer Manipulation

In this chapter we look at the file "bio.c" in detail. It contains most of the basic routines used to manipulate buffer headers and buffers (4535, 4720).

Individual buffer headers are tagged by a device number "b_dev", (4527) and a block number "b_blkno", (4531). (Note the way in which the latter is declared as an unsigned integer.)

Buffer headers may be linked simultaneously into two lists:

the "b"-lists are lists, one per device controller, which link together buffers associated with that device type;

the "av"-list is a list of buffers which may be detached from their current use and converted to an alternate use.

Both the "av"-list and the various "b"-lists are doubly linked to facilitate insertion and deletion at any point.

17.1 Flags

If a buffer is withdrawn temporarily from the "av"-list, then its "B_BUSY" flag is raised.

If the contents of a buffer correctly reflect the information that is or should be stored on disk, then the "B_DONE" flag is raised.

If the "B_DELWRI" flag is raised, the contents of the buffer are more up to date than the contents of the corresponding disk block, and hence the buffer must be written out before it can be reassigned.

17.2 A Cache-like Memory

It will be seen that the large buffers in UNIX are manipulated in a way which is analogous to the operation of hardware cache attached to the main memory of a computer e.g. the PDP11/70.

Buffers are not assigned to any particular program or file, except for very short intervals at a time. In this way a relatively small number of buffers can be shared effectively amongst a large number of programs and files.

Information is left in the buffers until the buffer is needed i.e. immediate "write through" is avoided if only part of the buffer has recently been changed. Programs which read or write records which are small compared with the buffer size are then not penalised unduly.

Finally when programs are terminated and files are closed, the problems of ensuring that the program's buffers are flushed properly (problems which have plagued other operating systems) have largely disappeared.

There is one area of practical concern: if the decision "when to write" is left to the operating system alone, then some buffers may not be written out for a very long time. Accordingly there is a utility program which runs twice per minute and forces all such buffers to be written out unconditionally. This limits the likely amount of damage that a sudden system crash may cause.

17.3 clrbuf (5038)

This routine zeros out the first 256 words (512 bytes) of the buffer. Note that the parameter passed to "clrbuf" is the address of the buffer header. "clrbuf" is called by "alloc" (6982).

17.4 incore (4899)

This routine searches for a buffer that is already assigned to a particular (device, block number) pair. It searches the circular "b"-list whose head is the "devtab" structure for the device type. If a buffer is found, the address of the buffer header is returned. "incore" is called by "breada" (4780, 4788).

17.5 getblk (4921)

This routine performs the same search as "incore" but goes further in that if the initial search is unsuccessful, a buffer is allocated from the "av"-list (available list).

By a call on "notavail" (4999), the buffer is removed from the "av"-list and flagged as "B_BUSY".

"getblk" is more suspicious of its parameters than "incore". It is called by

```
exec   (3040)        writei (6304)
exit   (3237)        iinit  (6928)
bread  (4758)        alloc  (6981)
breada (4781,4789)   free   (7016)
smount (6123)        update (7216)
```

4940: At this point the required buffer has been located by searching the "b"-list. Either it is "B_BUSY" in which case a "sleep" must be taken (4943), or else it is appropriated (4948);

4953: If the required buffer has not been located, and if the "av"-list is empty, set the "B_WANTED" flag for the "av"-list and go to "sleep" (4955);

4960: If the "av"-list is not empty, select the first member, and if it represents a "delayed write" arrange to have it written out asynchronously (4962);

4966: "B_RELOC" is a relic! (See 4583);

4967: The code from here until 4973 unconditionally removes the buffer from the "b"-list for its current device type and reinserts it into the bn-list for the new device type. Since this will frequently be a "no-op" i.e. the new and old device type will be the same, it would seem desirable to insert a test

```
if (bp->b_dev == dev)
```

before executing lines 4967 to 4974.

Note the special handling for calls where "dev == NODEV" (–1). (Such calls incidentally are made without a second parameter - tut! tut! See e.g. 3040.)

"bfreelist" serves as the "devtab" structure for the "b"-list for "NODEV".

17.6 brelse (4869)

This procedure takes the buffer passed as a parameter and links it back into the "av"-list.

Any process which is either waiting for the particular buffer or any available buffer is woken up.

Note however that since both "sleeps" (4943, 4955) are at the same priority, if two processes are waiting – one for the particular buffer and one for any buffer – it will be a toss-up which will get it.

By giving the first priority over the second (e.g. by biasing by one) the race should be resolved more satisfactorily. The disadvantage of such a change might be that it could lead to a deadlock situation in certain rather peculiar circumstances.

If an error has occurred e.g. upon reading information into the buffer the information in the buffer may be incorrect. The assignment on line 4883 ensures that the information in the buffer will not be mistakenly retrieved subsequently. The "B_ERROR" flag is set e.g. by "rkstrategy" (5403) and "rkintr" (5467).

To see how this could occur, consider what happens to a buffer when a disk i/o operation is completed:

5471 "rkintr" calls "iodone";

5026 "iodone" sets the "B_DONE" flag;

5028 "iodone" calls "brelse";

4387 "brelse" resets the "B_WANTED", "B_BUSY" and "B_ASYNC" flags but not the "B_DONE" flag;

. . . .

4948 "getblk" finds the buffer and calls "notavail";

5010 "notavail" sets the "B_BUSY" flag;

4759 "bread" (which called "getblk") finds the "B_DONE" flag set and exits.

Note that buffer headers are removed from the "av"-list by "notavail" and are returned by "brelse". Buffer headers are moved from one "b"-list to another by "getblk".

17.7 binit (5055)

This procedure is called by "main" (1614) to initialise the buffer pool. Empty, doubly linked circular lists are set up:

- for the "av"-list ("bfreelist" is head);
- the "b"-list for null devices ("dev == NODEV") ("bfreelist" is again head);
- a "b"-list for each major device type.

For each buffer:

- the buffer header is linked into the "b"-list for the device "NODEV" (–1);
- the address of the buffer is set in the header (5067);
- the buffer flags are set as "B_BUSY" (this doesn't seem to be really necessary) (5072);
- the buffer header is linked into the "av"-list by a call on "brelse" (5073);

The number of block devices is recorded as "nblkdev". This is used for checking values for "dev" in "getblk" (4927), "getmdev" (6192) and "openi" (6720). Inspection of "bdevsw" (4656) shows that "nblkdev" will be set to eight whereas the value one is what is really required.

This result could be obtained by "editing" as follows:

```
/5084/m/5081/ "nblkdev=i;
/5083/m/5077/ "i++
```

17.8 bread (4754)

This is the standard procedure for reading from block devices. It is called by:

```
wait   (3282)      iinit  (6927)
breada (4799)      alloc  (6973)
statl  (6051)      ialloc (7097)
smount (6116)      iget   (7319)
readi  (6258)      iupdat (7386)
writei (6305)      itrunc (7426, 7431)
bmap   (6472,6488) namei  (7625)
```

"getblk" finds a buffer. If the "B_DONE" flag is set no i/o is needed.

17.9 breada (4773)

This procedure has an additional parameter, as compared with "bread". It is called only by "readi" (6256).

4780: Check if the desired block has already been assigned to a buffer. (It may not yet be available, but at least is it there?);

4781: If not initiate the necessary read operation but don't wait for it to finish;

4788: Look around for the "read ahead" block. If it is not there, allocate a buffer (4789) but release it (4791) if the buffer is already ready;

4793: The "read ahead" block is not ready, so initiate an asynchronous read operation;

4798: If a buffer was assigned to the current block call "bread" to wrap it up, else...

4800: Wait for the completion of the operation which was started at line 4785.

17.10 bwrite (4809)

This is the standard procedure for writing to block devices. It is called by "exit" (3239), "bawrite" (4863), "getblk" (4963), "bflush" (5241), "free" (7021), "update" (7221) and "iupdat" (7400). N.B. "writei" calls "bawrite" (6310)!

4820: If the "B_ASYNC" flag is not set, the procedure does not return until the i/o operation is completed;

4823: If the "B_ASYNC" is set, but "B DELWRI" **was** not set (note "flag" is set at line 4816) call "geterror" (5336) to check on the error flag. (If "B_DELWRI" was set, and there is an error, sending the error indication to the right process is "too hard."). The call (4824) on "geterror" will only report errors related to the initiation of the write operation.

17.11 bawrite (4856)

This procedure is called by "writei" (6310) and "bdwrite" (4845). "writei" calls either "bawrite" or "bdwrite" depending on whether the block to be written has been wholly or partially filled.

17.12 bdwrite (4836)

This procedure is called by "writei" (6311) and "bmap" (6443, 6449, 6485, 6500 and 6501 !).

4844: Don't delay the write if the device is a magnetic tape drive ... keep everything in order;

4847: Set the "B_DONE", "B_DELWRI" flags and call "brelse" to link the buffer into the "av"-list.

17.13 bflush (5229)

This procedure is called by "update" (7201), which is called by "panic" (2420), "sync" (3489) and "sumount" (6150).

"bflush" searches the "av"-list for "delayed write" blocks and forces them to be written out asynchronously.

Note that as "notavail" adjusts the links of the "av"-list, the search (which runs at processor priority six) is reinitiated after each "delayed write" block is encountered.

Note also that since it happens that "bflush" is only called by "update" with "dev" equal to "NODEV", line 5238, in particular, could be simplified.

17.14 physio (5259)

This routine is called to handle "raw" input/output i.e. operations which ignore the normal 512 character block size.

"physio" is called by "rkread" (5476) and "rkwrite" (5483) which appear as entries in the array "cdevsw" (4684)

"Raw i/o" is not an essential feature of UNIX. For disk devices it is used mainly for copying whole disks and checking the integrity of the file system as a whole (see e.g. ICHECK (VIII) in the UPM), where it is convenient to read whole tracks, rather than single blocks, at a time.

Note the declaration of "strat" (5261). Since the actual parameter used e.g. "rkstrategy" (5389) does not return any value, is this form of declaration really necessary?

Section Four

Section Four is concerned with files and file systems.

A file system is a set of files and associated tables and directories organised onto a single storage device such as a disk pack.

This section covers the means of creating and accessing files, locating files via directories, and organising and maintaining file systems.

It also includes the code for an exotic breed of file called a "pipe".

18 File Access and Control

A large part of every operating system seems to be concerned with data management and file management, and UNIX turns out to be no exception.

Section Four of the source code contains thirteen files.

The first four contain common declarations needed by various of the other routines:

"file.h" describes the structure of the "file" array;

"filsys.h" describes the structure of the "super block" for "mounted" file systems;

"ino.h" describes the structure of "inodes" recorded on "mounted" devices;

"inode.h" describes the structure of the "inode" array;

The next two files, "sys2.c" and "sys3.c" contain code for system calls. ("sys1.c" and "sys4.c" were presented in Section Two).

Tne next five files, "rdwri.c", "subr.c", "fio.c", "alloc.c" and "iget.c", together present the principal routines for file management, and provide a link between the i/o oriented system calls and the basic i/o routines.

The file "nami.c" is concerned with searching directories to convert file pathnames into "inode" references.

Finally, "pipe.c" is the "device driver" for pipes.

18.1 File Characteristics

A UNIX file is conceptually a named character string, stored on one of a variety of peripheral devices (or in the main memory), and accessible via mechanisms appropriate to the usual peripheral devices.

It will be noted that there is no record structure associated with UNIX files. However "newline" characters may be inserted into the file to define substrings analogous to records.

UNIX carries the ideas of device independence to their logical extreme by allowing the file name in effect to determine uniquely all relevant attributes of the file.

18.2 System Calls

The following system calls are provided expressly for file manipulation:

#	Name	Line	#	Name	Line
3	read	5711	14	mknod	5952
4	write	5720	15	chmod	3560
5	open	5765	16	chown	3575
6	close	5846	19	seek	5861
8	creat	5781	21	mount	6086
9	link	5909	22	umount	6144
10	unlink	3510	41	dup	6069
12	chdir	3538	42	pipe	7723

18.3 Control Tables

The arrays "file" and "inode" are essential components of the file access mechanism.

18.4 file (5507)

The array "file" is defined as an array of structures (also named "file").

An element of the "file" array is considered to be unallocated if "f_count" is zero.

Each "open" or "creat" system call results in the allocation of an element of the "file" array. The address of this element is stored in an element of the calling process's array "u.u_ofile". It is the index of the newly allocated element of the latter array which is passed back to the user process. Descendants of a process created by "newproc" inherit the contents of the parent's "u.u_ofile" array.

Each element of "file" includes a counter, "f_count", to determine the number of current processes which reference it.

"f_count" is incremented by "newproc" (1878), "dup" (6079) and "falloc" (6857); it is decremented by "closef" (6657) and (if the file can't be opened) by "openl" (5836).

The "f_flag" (5509) of the "file" element notes whether the file is open for reading and/or writing or whether it is a "pipe" or not. (Further discussion of "pipes" will be deferred till Chapter Twenty-One.)

The "file" structure also contains a pointer, "f_inode" (5511) to an entry in the "inode" table, and a 32 bit integer, "f_offset" (5512), which is a logical pointer to a character within the file.

18.5 inode (5659)

"inode" is defined as an array of structures (also named "inode").

An element of the "inode" is considered to be unallocated if the reference count, "i_count", is zero.

At each point in time, "inode" contains a single entry for each file which may be referenced for normal i/o operations, or which is being executed or which has been executed and has the "sticky" bit set, or which is the working directory for some process.

Several "file" table entries may point to a single "inode" entry. The inode entry describes the general disporition of the file.

18.6 Resources Required

Each file requires the dedication of certain system resources. When a file exists, but is not being referenced in any way, it requires:

(a) a directory entry (16 characters in a directory file);

(b) a disk "inode" entry (32 characters in a table stored on the disk);

(c) zero, one or more blocks of disk storage (512 characters each).

In addition if the file is being referenced for some purpose, it requires

(d) a core "inode" entry (32 characters in the "inode" array);

Finally if a user program has "opened" the file for reading or writing, a number of resources are required:

(e) a "file" array entry (8 characters);

(f) an entry in the user program's "u.u_ofile" array (one word per file, pointing to a "file" array entry);

Mechanisms have to be set up for allocating and deallocating each of these resources in an orderly manner. The following table gives the names of the principal procedures involved:

resource	obtain	free
directory entry	namei	namei
disk "inode" entry	ialloc	ifree
disk storage block	alloc	free
core "inode" entry	iget	iput
"file" table entry	falloc	closef
"u_ofile" entry	ufalloc	close

18.7 Opening a File

When a program wishes to reference a file which already exists, it must "open" the file to create a "bridge" to the file. (Note that in UNIX, processes usually inherit the open files of their parents or predecessors, so that often all needed files are already implicitly open.) If the file does not already exist, it must be "created".

This second case will be investigated first:

18.8 creat (5781)

5786: "namei" (7518) converts a pathname into an "inode" pointer. "uchar" is the name of a procedure which recovers the pathname, character by character, from the user program data area;

5787: A null "inode" pointer indicates either an error or that no file of that name already exists;

5788: For error conditions, see "CREAT (II)" in the UPM;

5790: "maknode" (7455) creates a core "inode" via a call on "ialloc" and then initialises it and enters it into the appropriate directory. Note the explicit resetting of the "sticky" bit

18.9 openl (5804)

This procedure is called by "open" (5774) and "creat" (5793, 5795), passing values of the third parameter, "trf", of 0, 2 and 1 respectively. The value 2 represents the case where no file of the desired name already exists.

5812: The second parameter, "mode", can take the values 01 ("FREAD"), 02 ("FWRITE") or 03 ("FREAD | FWRITE") when "trf" is 0, but only 02 otherwise;

Whete a file of the desired name already exists, check the access permissions for the desired mode(s) of activity via calls on "access" (6746), which may set "u.u_error" as a side-effect;

5824: If the file is being "created", eliminate its previous contents via a call on "itrunc" (7414). The code here could be improved by changing the test to "(trf == 1)". Verify that this would be so.

5826: "prele" (7882) is used to "unlock" "inodes". Where, you may ask, did the "inode" get "locked", and why?

5827: Note that "falloc" (6847) calls "ufalloc" (6824) as the first thing it does;

5831: "ufalloc" leaves the user file identifying number "u.u_ar0[R0]". Why does this statement occur where it does, instead of after line 5834?

5832: "openi" (6702) is called to call handlers for special files, in cae any device specific actions are required (for disk files there is no action);

5839: In the case of an error while making the "file" array entry, the "inode" entry is released by a call on "iput".

It will be seen that responsibility is quite widely distributed. The "file" table entry is initialised by "falloc" and "openl"; the "inode" table entry, by "iget", "ialloc" and "maknode".

Note that "ialloc" clears out the "i_addr" array of a newly allocated "inode" and "itrunc" does the same for a pre-existing "inode", so that after the "creat" system call, there are no disk blocks associated with the file, now classed as "small".

18.10 open (5763)

We now turn to consider the case where a program wishes to reference a file which already exists.

"namei" is called (5770) with a second parameter of zero to locate the named file. ("u.u_arg[0]" contains the address in the user space of a character string which defines a file path name.)

"u.u_arg[1]" has to be incremented by one, because there is a mismatch between the user programming conventions and the internal data representations.)

18.11 openl revisited

"trf" is now zero, so access permissions are checked (5813) but the existing file (if any) is not deallocated (5824).

What is a little disconcerting here is that, apart from the call on "falloc" (5827), there is no direct call on any of the "resource allocation" routines. Of course, for an existing file, neither directory entry nor disk "inode" entry nor disk blocks need be allocated. The core "inode" entry is allocated (if necessary) as a side-effect of the call on "namei", but ... where is it initialised?

18.12 close (5846)

The "close" system call is used to sever explicitly the connection between a user program and a file and thus can be regarded as the inverse of "open".

The user program's file identification is passed via r0. The value is validated by "getf" (6619), the "u.u_ofile" entry is erased, and a call is made on "closef".

18.13 closef (6643)

"closef" is called by "close" (5854) and by "exit" (3230). (The latter is more common since most files do not get closed explicitly but only implicitly when the user program terminates.)

6649: If the file is a pipe, reset the mode of the pipe and "wakeup" any process which is waiting for the pipe, either for information or for space;

6655: If this is the last process to reference the file, call "closei" (6672) to handle any special end of file processing for special files and then call "iput";

6657: Decrement the "file" entry reference count. If this now zero, the entry is no longer allocated.

18.14 iput (7344)

"closei", as its last action calls "iput". This routine is in fact called from many places, whenever a connection to a core "inode" is to be severed and the reference count decremented.

7350: If the reference count is one at this point, the "inode" is to be released. While this is happening, it should be locked.

7352: If the number of "links" to the file is zero (or less) the file is to be deallocated (see below);

7357: "iupdat" (7374) updates the accessed and update times as recorded on the disk "inode";

7358: "prele" unlocks the "inode". Why should it be called here as well as at line 7363?

18.15 Deletion of Files

New files are automatically entered into the file directory as permanent files as soon as they are "opened". Subsequent "closing" of a file does not automatically cause its deletion. As was seen at line 7352, deletion will occur when the field "i_nlink" of the core "inode" entry is zero. This field is set to one initially by "maknode" (7464) when the file is first created. It may be incremented by the system call "link" (5941) and decremented by the system call "unlink" (3529).

Programs which create temporary "work files" should remove these files before terminating, by executing an "unlink" system call. Note that the "unlink" call does not of itself remove the file. This can only happen when the reference count ("i_count") is about to be decremented to zero (7350, 7362).

To minimise the problems associated with "temporary" files which survive program or system crashes, programmers should observe the conventions that:

(a) temporary files should be "unlinked" immediately after they are opened;

(b) temporary files should always be placed in the "tmp" directory. Unique file names can be generated by incorporating the process's identifying number into the file name (See "getpid" (3480)).

18.16 Reading and Writing

It is of interest to work through an abbreviated summary of the code which is invoked when a user process performs a "read" system call before examining the code in detail.

```
.... read (f, b, n); /*user program/

                {trap occurs}
2693 trap

                {system call #3}
5711 read ();
5713   rdwr (PREAD);
```

Execution of the system call by the user process results in the activation of "trap" running in kernel mode. "trap" recognises system call #3, and calls (via "trapl") the routine "read", which calls "rdwr".

```
5731 rdwr

5736 fp = getf (u.u_ar0[R0];
5743 u.u_base = u.u_arg[0];
5744 u.u_count = u.u_arg[1];
5745 u.u_segflg = 0;
5751 u.u_offset[1] = f2->f offset[1];
5752 u.u_offset[0] = fp->f offset[0];
5754 readi(fp->f inode);
5756 dpadd(fp->f offset,
              u.u_arg[1]-u.u_count);
```

"rdwr" includes much code which is common to both "read" and "write" operations. It converts, via "getf" (6619), the file identification supplied by the user process into the address of an entry in the "file" array.

Note that the first parameter of the system call is passed in a different way from the remaining two parameters.

"u.u_segflg" is set to zero to indicate that the operation destination is in the user address space. After "readi" is called with a parameter which is an "inode" pointer, the final accounting is performed by adding the number of characters requested for transfer less the residual number not transferred (left in "u.u_count") to the file offset.

```
6221 readi

6239 lbn = lshift (u.u_offset, -9);
6240 on = u.u_offset[1] & 0777;
6241 n = min (512 - on, u.u_count);
6248 bn = bmap(ip, lbn);
6250 dn = ip->i_dev;
6258 bp = bread (dn, bn);
6260 iomove (bp, on, n, B READ);
6261 brelse (bp);
```

"readi" converts the file offset into two parts: a logical block number, "lbn", and an index into the block, "on". The number of characters to be transferred is the minimum of "u.u_count" and the number of characters left in the block (in which case additional block(s) must be read (not shown)) (and the number of characters remaining in the file (this case is not shown)).

"dn" is the device number which is stored within the "inode". "bn" is the actual block number on the device (disk), which is computed by "bmap" (6415) using "lbn".

The call on "bread" finds the required block, copying it into core from disk if necessary. "iomove" (6364) transfers the appropriate characters to their destination, and performs accounting chores.

18.17 rdwr (5731)

"read" and "write" perform similar operations and share much code. The two system calls, "read" (5711) and "write" (5720), call "rdwr" immediately to:

5736: Convert the user program file identification to a pointer in the file table;

5739: Check that the operation (read or write) is in accordance with the mode with which the file was opened;

5743: Set up various standard locations in "u" with the appropriate parameters;

5746: "pipes" get special treatment right from the start!

5755: Call "readi" or "writei" as appropriate;

5756: Update the file offset by, and set the value returned to the user program to, the number of characters actually transferred.

18.18 readi (6221)

6230: If no characters are to be transferred, do nothing;

6232: Set the "inode" flag to indicate that the "inode" has been accessed;

6233: If the file is a character special file, call the appropriate device "read" procedure, passing the device identification as parameter;

6238: Begin a loop to transfer data in amounts up to 512 characters at a time until (6262) either an irrecoverable error condition has been encountered or the requested number of characters has been transferred;

6239: "lshift" (1410) concatenates the two words of the array "u.u_offset", shifts right by nine places, and truncates to 16 bits. This defines the "logical block number" of the file which is to be referenced;

6240: "on" is a character offset within the block;

6241: "n" is determined initially as the minimum of the number of characters beyond "on" in the block, and the number requested for transfer. (Note that "min" (6339) treats its arguments as unsigned integers.)

6242: If the file is not a special block file then ...

6243: Compare the file offset with the current file size;

6246: Reset "n" as the minimum of the characters requested and the remaining characters in the file;

6248: Call "bmap" to convert the logical block number for the file to a physical block number for its host device. There will be more on "bmap" shortly. For now, note that "bmap" sets "rablock" as a side effect;

6250: Set "dn" as the device identification from the "inode";

6251: If the file is a special block file then ...

6252: Set "dn" from the "i_addr" field of the "inode" entry. (Presumably this will nearly always be the same as the "i_dev" field, so why the distinction?)

6253: Set the "read ahead block" to the next physical block;

6255: If the blocks of the file are apparently being read sequentially then ...

6256: Call "breada" to read the desired block and to initiate reading of the "read ahead block";

6258: else just read the desired block;

6260: Call "iomove" to transfer information from the buffer to the user area;

6261: Return the buffer to the "av"-list.

18.19 writei

6303: If less than a full block is being written the previous contents of the buffer must be read so that the appropriate part can be preserved, otherwise just get any available buffer;

6311: There is no "write ahead" facility, but there is a "delayed write" for buffers whose final characters have not been changed;

6312: If the file offset now points beyond the recorded end of file character, the file has obviously grown bigger!

6318: Why is it necessary/desirable to set the "IUPD" flag again? (See line 6285.)

18.20 iomove (6364)

The comment at the beginning of this procedure says most of what needs to be said. "copyin", "copyout", "cpass" and "passc" may be found at lines 1244, 1252, 6542 and 6517 respectively.

18.21 bmap (6415)

A general description of the function of "bmap" may be found on Page 2 of "FILE SYSTEM (V)" of the UPM.

6423: Files of more than 2**15 blocks (2**24 characters) are not supported;

6427: Start with the "small" file algorithm (file is not greater than eight blocks i.e. 4096 characters);

6431: If the block number is 8 or more, the "small" file must converted into a large file. Note this is a side effect of "bmap", and should occur only when "bmap" has been called by "writei" (and never by "readi" – see line 6245). Thus all files start life as "small" files and are never explicitly changed to "large" files. Note also that the change is irreversible!

6435: “alloc” (6956) allocates a block on device “d” from the device’s free list. It then assigns a buffer to this block and returns a pointer to the buffer header;

6438: The eight buffer addresses in the “i_addr” array for the “inode” are copied into the buffer area and then erased;

6442: “i_addr[0]” is set to point to the buffer which is set up for a “delayed” write;

6448: The file is still small. Get the next block if necessary;

6456: Note the setting of “rablock”;

18.22 Leftovers

You should investigate the following procedures for yourself:

```
seek  (5861)   statl (6045)
sslep (5979)   dup   (6069)
fstat (6014)   owner (6791)
stat  (6028)   suser (6811)
```

19 File Directories and Directory Files

As we have seen, much important information about individual files is contained in the "inode" tables. If the file is currently accessible, or being accessed, the relevant information is held in the core "inode" table. If a file is on disk (more generally, on some file system volume) and is not currently accessible, then the relevant "inode" table is the one recorded on the disk (file system volume).

Notably absent from the "inode" table is any information regarding the "name" of the file. This is stored in the directory files.

19.1 The Directory Data Structure

Each file must have at least one name. A file may have more than one distinct name, but the same name may not be shared by tw distinct files, i.e. each name must define a unique file.

A name may be multipart. When written, the parts or components of the name are separated by slashes ("/"). The order of components within a name is significant i.e. "a/b/c" is different from "a/c/b".

If file names are divided into two parts: an initial part or "stem" and a final part or "ending", then two files whose names have identical stems are usually relate in some way. They may reside on the same disk, they may belong to the same user, etc. Users make initial reference to files by quoting the file name, e.g. in the "open" system call. An important operating sysem function is to decode the name into the corresponding "inode" entry. To do this, UNIX creates and maintains a directory data structure. This structure is equivalent to a directed graph with named edges.

In its purest form, the graph is a tree i.e. it has a single root node, with exactly one path between the root and any node. More commonly in UNIX (but not so commonly in other operating systems) the graph is a lattice which may be obtained from a tree by coalescing one or more groups of leaves.

In this case, while there is still only one path between the root and any interior node, thee may be more than one path between the root and a leaf. Leaves are nodes without successors and correspond to data files. Interior nodes are nodes with successors and correspond to directory files.

The name for a file is obtained from the names of the edges of the path between the root and the node corresponding to the file. (For this reason, the name is often referred to as a "pathname".) If there are several paths, then the file has several names.

19.2 Directory Files

A directory file is in many respects indistinguishable from a non-directory file. However it contains information which is used in locating other files and hence its contents are carefully protected, and are manipulated by the operating system alone.

In every file, the information is stored as one or more 512 character blocks. Each block of a directory file is divided into 32 * 16 character structures. Each structure consists of a 16 bit "inode" table pointer and a 14 character name. The "inode" pointer is to the "inode" table on the same disk or file system volume as the files which the directory references. (More on this later.) An "inode" value of zero defines a null entry in the directory.

The procedures which reference directories are:

```
namei  (7518) search directory
link   (5909) create alternate name
wdir   (7477) write directory entry
unlink (3510) delete name
```

19.3 namei (7518)

7531: "u.u_cdir" defines the "inode" of a process's current directory. A process inherits its parent's current directory at birth ("newproc", 1883). The current directory may be changed using the "chdir" (3538) system call;

7532: Note that "func" is a parameter to "namei" and is always either "uchar" (7689) or "schar" (7679);

7534: "iget" (7276) is called to:

- wait until such time as the "inode" corresponding to "dp" is no longer locked;
- check that the associated file system is still mounted;
- increment the reference count;
- lock the "inode";

7535: Multiple slashes are acceptable! (i.e. "////a///b/" is the same as "/a/b");

7537: Any attempt to replace or delete the current working directory or the root directory is bounced immediately!

7542: The label "cloop" marks the beginning of a program loop that extends to line 7667. Each cycle analyses a component of the pathname (i.e. a string terminated by a null character or one or more slashes). Note that a name may be constructed from many different characters (7571);

7550: The end of the pathname has been reached (successfully). Return the current value of "dp";

7563: "search" permission for directories is coded in the same way as "execute" permission for other files;

7570: Copy the name into a more accessible location before attempting to match it with a directory entry. Note that a name of greater than "DIRSIZ" characters is truncated;

7589: "u.u_count" is set to the number of entries in the directory;

7592: The label "eloop" marks the beginning of a program loop which extends to line 7647. Each cycle of the loop handles a single directory entry;

7600: If the directory has been searched (linearly!) without matching the supplied pathname component, then there must be an error unless:

(a) this is the last component of the pathname, i.e. "c=='\0' ";

(b) the file is to be created, i.e. "flag == 1"; ard

(c) the user program has "write" permission for the directory;

7606: Record the "inode" address for the directory for the new file in "u.u_pdir";

7607: If a suitable slot for a new directory entry has previously been encountered (7642), store the value in "u.u_offset[1]"; else set the "IUPD" flag for the "dp" designated "inode" (but why?);

7622: When appropriate, read a new block from the directory file (note the use of "bread") (why not "breada"?), after carefully releasing any previously held buffer;

7636: Copy the eight words of the directory entry into the array "u.u_dent". The reason for copying before comparing is obscure! Can this actually be more efficient? (The reason for copying the whole directory at all is rather perplexing to the author of these notes.);

7645: This comparison makes efficient use of a single character pointer register variable, "cp". The loop would be even more efficient if word by word comparison were used;

7647: The "eloop" cycle is terminated by one of:

```
return(NULL);     (7610)
goto out;         (7605, 7613)
```

a successful match so that the branch to "eloop" (7647) is not taken;

7657: If the name is to be deleted ("flag==2"), if the pathname has been completed, and if the user program has "write" access to the directory, then return a pointer to the directory "inode";

Save the device identity temporarily (why not in the register "c"?) and call "iput" (7344) to unlock "dp", to decrement the reference count on "dp" and to perform any consequent processlng;

7664: Revalidate "dp" to point to the "inode" for the next level file;

7665: "dp==NULL" shouldn't happen, since the directory says the file exists! However "inode" table overflows and i/o errors can occur, and sometimes the file system may be left in an inconsistent state after a system crash.

19.4 Some Comments

"namei" is a key procedure which would seem to have been written very early, to have been thoroughly debugged and then to have been left essentially unchanged. The interface between "namei" and the rest of the system is rather complex, and for that reason alone, it would not win the prize for "Procedure of the Year".

"namei" is called thirteen times by twelve different procedures:

line	routine	parameters	
3034	exec	uchar	0
3543	chdir	uchar	0
5770	open	uchar	0
5914	link	uchar	0
6033	stat	uchar	0
6097	smount	uchar	0
6186	getmdev	uchar	0
6976	owner	uchar	0
5786	creat	uchar	1
5928	link	uchar	1
5958	mknod	uchar	1
3515	unlink	uchar	2
4101	core	schar	1

It will be seen that:

(a) there are two calls from "link";

(b) the calls can be divided into four categories, of which the first is by far the largest;

(c) the last two categories have only one representative each;

(d) in particular, there is only one call involving the routine "schar", which is always for a file called "core". (If this case were handled as a special case e.g. where the second parameter had the value "3", then the "uchar"s and "schar" could be eliminated.)

"namei" may terminate in a variety of ways:

(a) if there has been an error, then a "NULL" value is returned and the variable "u.u_error" is set.

(Most errors result in a branch to the label "out" (7669) so that reference counts for the inodes are properly maintained (7670). This is not necessary if the failure occurs in "iget" (7664).);

(b) if "flag==2" (i.e. the call is from "unlink"), the value returned (in normal circumstances) is an "inode" pointer for the parent directory of the named file (7660);

(c) if "flag==1" (i.e. the call is from "creat" or "link" or "mknod", and a file is to be created if it does not already exist) and if the named file does not exist, then a "NULL" value is returned (7610). In this case a pointer to the "inode" for the directory which will point to the new file, is left in "u.u_pdir" (7606). (Note also that in this case, "u.u_offset" is left pointing either at an empty directory entry or at the end of the directory file.);

(d) if in the remaining cases, the file exists, an "inode" pointer for the file is returned (7551). The "inode" is locked and the reference count has been incremented. A call to "iput" is needed subsequently to undo both these side effects.

19.5 link (5909)

This procedure implements a system call which enters a new name for an existing file into the directory structure. Arguments to the procedure are the existing and the new names of the file;

5914: Look up the existing file name;

5917: If the file already has 127 different names, quit in disgust;

5921: If the existing file turns out to be a directory, then only the super-user may rename it;

5926: Unlock the existing file "inode" This is locked when the first call on "namei" does an "iget" (7534, 7664).

Under what conditions would the failure to unlock the "inode" here be disastrous? The chances that the existing file would be a directory encountered in the search for the new name would seem slight, if not impossible. Most probably the relevant circumstance is where the system is attempting to recreate an alternative file name or alias, which **already** exists;

5927: Search the directory for the second name, with the intention of creating a new entry;

5930: There is an existing file with the second name;

5935: "u.u_pdir" is set as a side effect of the call on "namei" (5928). Check that the directory resides on the same device as the file;

5940: Write a new directory entry (see below);

5941: Increase the "link" count for the file.

19.6 wdir (7477)

This procedure enters a new name into a directory. It is called by "link" (5940) and "maknode" (7467) with a pointer to a (core) "inode" as parameter.

The sixteen characters of the directory entry are copied into the structure "u.u_dent", and written from there into the directory file. (Note that the previous content of "u.u_dent" will have been the name of the last entry in the directory file.)

The procedure assumes that the directory file has already been searched, that the "inode" for the dlrectory file has already been allocated and that the values of "u.u offset" have been set appropriately.

19.7 maknode (7455)

This procedure is called from "core" (4105), "creat" (5790) and "mknod" (5966), after a previous call on "namei" with a second parameter of one, has revealed that no file of the specified name existed.

19.8 unlink (3510)

This procedure implements a system call which deletes a file name from the directory structure. (When all references to a file are deleted, the file itself will be deleted.)

3515: Search for a file with the specified name, and if it exists, return a pointer to the "inode" of the immediate parent directory;

3518: Unlock the parent directory;

3519: Get an "inode" pointer to the file itself;

3522: Unlinking directories is forbidden, except for super-users;

3528: Rewrite the directory entry with the "inode" value set to zero;

3529: Decrement the "link" count.

Note that there is no attempt to reduce the size of a directory below its "high water" mark.

19.9 mknod (5952)

This procedure, which implements a system call of the same name, is only executable by the super-user. As explained in the Section "MKNOD(II)" of the UPM, this system call is used to create "inodes" for special files.

"mknod" also solves the problem of "where do directories come from"? The second parameter passed to "mknod" is used, without modification or restriction to set "i_mode". (Compare "creat" (5790) and "chmod" (3569)). This is the only way an "inode" can get flagged as a directory, for instance.

In such cases, the third parameter passed to "mknod" **must** be zero. This value is copied into "i_addr[0]" (as is appropriate for special files), and, if non-zero, will be accepted uncritically by "bmap" (6447). It might be prudent to insert a test

```
if (ip->i_mode & (IFCHR & IFBLK) != 0)
```

before line 5969, rather than rely indefinitely on the infallibility of the super-user.

19.10 access (6746)

This procedure is called by "exec" (3041), "chdir" (3552), "core" (4109), "openl" (5815, 5817), "namei" (7563, 7664, 7658) to check access permission to a file. The second parameter, "mode", is equal to one of "IEXEC", "IWRITE" and "IREAD", with octal values of 0100, 0200 and 0400 respectively.

6753: "write" permission is denied if the file is on a file system volume which has been mounted as "read only" or if the file is functioning as the text segment for an executing program;

6763: the suer-user may not execute a file unless it is "executable" in at least one of the three "permission" groups. In any other situation he is always allowed access;

6769: If the user is not the owner of the file, shift "m" three places to the right so that group permissions will be operative ... If the groups don't match, shift "m" again;

6774: Compare "m" and the access permissions.

Note that there is an anomaly here in that if a file has a "mode" of 0077, the owner cannot reference it at all, but everyone else can. This situation could be changed satisfactorily by inserting a statement

```
m =|  (m |  (m >> 3)) >> 3;
```

after line 6752, and replacing lines 6764, 6765 by

```
if (m & IEXEC && (m & ip->i_mode) == 0)
```

20 File Systems

In most computer systems more than one peripheral storage device is used for the storage of files. It is now necessary to discuss a number of matters pertaining to the management by UNIX of the whole set of files and file storage devices. First, some definitions:

file system: an integrated collection of files with a hierarchical system of directories recorded on a single block oriented storage device;

storage device: a device which can store information (especially disk pack or DECtape, etc.);

access device: a mechanism for transferring information to or from a storage device;

a storage device is only **accessible** if it is inserted in an access device. In this situation reference to the storage device is made via a reference to the access devce;

a storage device is acceptable as a **file system volume** if:

(a) information is recorded as addressable blocks of 512 characters each, which can be independently read or written.

(Note IBM compatible magnetic tape does not satisfy this condition.);

(b) the information recorded on the device satisfies certain consistency criteria:

block #1 is formatted as a "super block" (see below);

blocks #2 to #(n+1) (where n is recorded in the "super block") contain an "inode" table which references all files recorded on the storage device, and does not reference any other files;

directory files recorded on the storage device reference all, and only, files on the same storage device, i.e. a file system volume constitutes a self-contained set of files, directories and "inode" table;

a file system volume is mounted if the presence of the storage device in an access device has been formally recognised by the operating system.

20.1 The 'Super Block' (5561)

The "super block" is always recorded as block #l on the storage device. (Block #0 is always ignored and is available for miscellaneous uses not necessarily concerned with UNIX.)

The "super block" contains information used in allocating resources, viz. the storage blocks and the entries in the "inode" table recorded on the file system. While the file system volume is mounted a copy of the "super block" is maintained in core and updated there. To prevent the storage device copy becoming too far out of date, its contents are written out at regular intervals.

20.2 The 'mount' table (0272)

The "mount" table contained an entry for each mounted file system volume. Each entry defines the device on which the file system volume is mounted, a pointer to the buffer which stores the "super block" for the device, and an "inode" pointer. The table is referenced as follows:

iinit (6922) which is called by "main" (1615), makes an entry for the root device;

smount (6086) is a system call which makes entries for additional devices;

iget (7276) searches the "mount" table if it encounters an "inode" with the 'IMOUNT' flag set;

getfs (7167) searches the "mount" table to find and return a pointer to the "super block" for a particular device;

update (7201) is called periodically and searches the "mount" table to locate information which should be written from core tables into the tables maintained on the file system volumes;

sumount (6144) is a system call which deletes entries from the table.

20.3 iinit (6922)

This routine is called by "main" (1615) to initialise the "mount" table entry for the root device.

6926: Call the "open" routine for the root device. Note that "rootdev" is defined in "conf.c" (4695);

6931: Copy the contents of the root device "super block" into a buffer area not associated with any particular device;

6933: The zeroeth entry in the "mount" table is assigned to the root device. Only two of the three elements are explicitly initialised. The third, the "inode" pointer, will never be referenced;

6936: The "locks" stored in the "super block" are explicitly reset. (These locks may have been set when the "super block" was last written onto the file system volume);

6938: The root device is mounted in a "writable" state;

6939: The system sets its idea of the current time and date from the time recorded in the "super block". (If the system has been stopped for an appreciable period, the computer operator will need to reset the contents of "time".)

20.4 Mounting

From an operational view point, "mounting" a file system volume involves placing it in a suitable access device, readying the device, and then entering a command such as parameters.

```
''/etc/mount /dev/rk2 /rk2''
```

to the "shell", which forks a program to perform a "mount" system call, passing pointers to the two file names as parameters.

20.5 smount (6086)

6093: "getmdev" decodes the first argument to locate a block oriented access device;

6096: "u.u_dirp" is reset preparatory to calling "namei" to decode the second file name. (Note that "u.u_dirp" is set by "trap" to "u.u_arg[0]" (2770);

6100: Check that the file named by the second parameter is in a satisfactory condition, i.e. no one else is currently accessing the file, and that the file is not a special file (block or character);

6103: Search the "mount" table looking for an empty entry ("mp->m bufp==NULL") or an entry already made for the device. (The "mount" data structure is defined at line 0272);

6111: "smp" should point to a suitable entry in the "mount" table;

6113: Perform the appropriate "open" routine, with the device name and a read/write flag as arguments. (As was seen earlier, for the RK05 disk the "open" routine is a "no-op");

6116: Read block #1 from the device. This block is the "super block";

6124: Copy the "super block" into a buffer associated with "NODEV", from the buffer associated with "d". The second buffer will not be released again until the device is unmounted;

6130: "ip" points to the "inode" for the second named file. This "inode" is now flagged as "IMOUNT". The effect of this is to force "iget" (7292) to ignore the normal contents of the file, while the file system volume is mounted. (In practice, the second file is an empty file created especially for this purpose.)

20.6 Notes

1. The "read/write" status of a mounted device depends only on the parameters provided to "smount". No attempt is made to sense the hardware "read/write" status. Thus if a disk is readied with "write protect" on, but is not mounted "read only", then the system will complain vigorously.

2. The "mount" procedure does not carry out any kind of label checking on the "mounted" file system volume. This is reasonable in a situation where file system volumes are rarely rearranged. However in situations where volumes are mounted and remounted frequently, some means of verifying that the correct volume has been mounted would seem desirable. (Further, if a file system volume contains sensitive information, it may be desirable to include some form of password protection as well. There is room in the "super block" (5575) for the storage of a name and an encrypted password.)

20.7 iget (7276)

This procedure is called by "main" (1616,1618), "unlink" (3519), "ialloc" (7078) and "namei" (7534, 7664) with two parameters which together uniquely identify a file: a device, and the "inode" number of a file on the device. "iget" returns a reference to an entry in the core "inode" table.

When "iget" is called, the core "inode" table is searched first to see if an entry already exists for the file in the core "inode" table. If not, then "iget" creates one.

7285: Search the core "inode" table ...

7286: If an entry for the designated file already exists ...

7287: Then if it is locked go to sleep;

7290: Try again. (Note the whole table needs to be searched again from the beginning, because the entry may have vanished!);

7292: If the IMOUNT flag is on ... this is an important possibility for which we will delay the discussion;

7302: If the "IMOUNT" flag is not set, increase the "inode" reference count, set the "ILOCK" flag and return a pointer to the "inode";

7306: Make a note of the first empty slot in the "inode" table;

7309: If the "inode" table is full, send a message to the operator, and take an error exit;

7314: At this point, a new entry is to be made in the "inode" table;

7319: Read the block which contains the file system volume "inode". Note the use of "bread" instead of "readi", the assumption that "inode" information begins in block #2 and the convention that valid "inode" numbers begin at one (not zero);

7326: A read error at this point isn't very well reported to the rest of the system;

7328: Copy the relevant "inode" information. This code makes implicit use of the contents of the file "ino.h" (Sheet 56), which isn't referenced explicitly anywhere.

Let us now return to unfinished business:

7292: The "IMOUNT" flag is found to be set. This flag was set by "smount", when a file system volume was mounted;

7293: Search the "mount" table to find the entry which points to the curent "inode". (Although searching this table is not a horrendous overhead, it does seem possible that a "back pointer" could be conveniently stored in in the "inode" e.g. in the "i_lastr" field. This would save both time and code space.;

7396: Reset "dev" and "ino" to the mounted device number and the "inode" number of the root directory on the mounted file system volume. Start again.

Clearly, since "iget" is called by "namei" (7534, 7664), this technique allows the whole directory structure on the mounted file system volume to be integrated into the pre-existing directory structure. If we momentarily ignore the possible deviations of directory structures away from tree structures, we have the situation where a leaf of the existing tree is being replaced by an entire subtree.

20.8 getfs (7167)

There is little that needs to be said about this procedure in addition to the author's comment. This procedure is called by

`access`	`(6754)`	`ialloc`	`(7072)`
`alloc`	`(6961)`	`ifree`	`(7138)`
`free`	`(7004)`	`iupdat`	`(7383)`

Note the cunning use of "n1", "n2" which are declared as character pointers i.e. as unsigned integers. This allows only one sided tests on the two variables at line 7177.

20.9 update (7201)

The function of this procedure, in its broadest terms, is to ensure that information on the file system volumes is kept up to date. The comment for this procedure (beginning on line 7190) describes the three main sub-functions, (in the reverse order!).

"update" is the whole business of the "sync" system call (3486). This may be invoked via the "sync" shell command. Alternatively there is a standard system program which runs continuously and whose only function is to call "sync" every 30 seconds. (See "UPDATE(VIII)" in the UPM.)

"update" is called by "sumount" (6150) before a file system volume is unmounted, and by "panic" (2420) as the last action of the system before activity ceases.

7207: If another execution of "update" is under way, then just return;

7210: Search the "mount" table;

7211: For each mounted volume, ...

7213: Unless the file system has not been recently modified or the "super block" is locked or the volume has been mounted "read only" ...

7217: Update the "super block", copy it into a buffer and write the buffer out onto the volume;

7223: Search the "inode" table, and for each non-null entry, lock the entry and call "iupdat" to update the "inode" entry on the volume if appropriate;

7229: Allow additional executions of "update" to commence;

7230: "bflush" (5229) forces out any "delayed write" blocks.

20.10 sumount (6144)

This system call deletes an entry for a mounted device from the "mount" table. The purpose of this call is to ensure that traffic to and from the device is terminated properly, before the storage device is physically removed from the access device.

6154: Search the "mount" table for the appropriate entry;

6161: Search the "inode" table for any outstanding entries for files on the device. If any such exist, take an error exit, and do not change the "mount" table entry;

6168: Clear the "IMOUNT" flag.

20.11 Resource Allocation

Our attention now turns to the management of the resources of an individual FSV (file system volume).

Storage blocks are allocated from the free list by "alloc" at the request of "bmap". Storage blocks are returned to the free list by "free" at the behest of "itrunc" (which is called by "core", "openl" and "iput").

Entries in the FSV "inode" tables are made by "ialloc", which is called by "maknode" and "pipe". Entries in this table are cancelled by "ifree", which is called by "iput".

The "super block" for the FSV is central to the resource management procedures. The "super block" (5561) contains:

- size information (total resources available);
- list of up to 100 available storage blocks;
- list of up to 100 available "inode" entries;
- locks to control manipulation of the above lists;
- flags;
- current date of last update.

If the list in core of available "inode" entries for the file system volume ever becomes exhausted, then the entire table on the FSV is read and searched to rebuild the list. Conversely if the available "inode" table overflows, additional entries are simply forgotten to be rediscovered later.

A different strategy is used for the list of available storage blocks. These blocks are arranged in groups of up to one hundred blocks. The first block in each group (except the very first) is used to store the addresses of the blocks belonging to the previous group. Addresses of blocks in the last incomplete group are stored in the "super block".

The first entry in the first list of block numbers is zero, which acts as a sentinel. Since the whole list is subject to a LIFO discipline, discovery of a block number of zero in the list signifies that the list is in fact empty.

20.12 alloc (6965)

This is called by "bmap" (6435, 6448, 6468, 6480, 6497) whenever a new storage block is needed to store part of a file.

6961: Convert knowledge of the device name into a pointer to the "super block";

6962: If "s_flock" is set, the list of available blocks is currently being updated by another process;

6967: Obtain the block number of the next available storage block;

6968: If the last block number on the list is zero, the entire list is now empty;

6970: "badblock" (7040) is used to check that the block number obtained from the list seems reasonable;

6971: If the list of available blocks in the "super block" is now empty, then the block just located will contain the addresses of the next group of

6972: Set "s_flock" to delay any other process from getting a "no space" indication before the list of available blocks in the "super block" can be replenished;

6975: Determine the number of valid entries in the list to be copied;

6978: Reset "s_flock", and "wakeup" anyone waiting;

6982: Clear the buffer so that any information recorded in the file by default will be all zeros;

6983: Set the "modified" flag to ensure that the "super block" will be written out by "update" (7213).

20.13 itrunc (7414)

This procedure is called by "core" (4112), "openl" (5825) and "iput" (7353). In the first two cases, the contents of the "file" are about to be replaced. In the third case, the file is about to be abandoned.

7421: If the file is a character or block special file then there is nothing to do;

7423: Search backwards the list of block numbers stored in the "inode";

7425: If the file is large, then an indirect fetch is needed. (A double indirect fetch is needed for blocks numbered seven and higher.);

7427: Reference all **257** elements of the buffer in reverse order. (Note this seems to be the only place where characters #512, #513 of the buffer area are referenced. Since they will presumably contain zero, they will contribute nothing to the calculation. Hence if "510" were substituted for "512" here, and again on line 7432, a general improvement all round would result (?));

7438: "free" returns an individual block to the available list;

7439: This is the end of the "for" statement commencing on line 7427. (Likewise the statement which begins at 7432 ends at 7435.);

7443: Clear the entry in "i_addr[ ]";

7445: Reset size information, and flag the "inode" as "updated".

20.14 free (7000)

This procedure is called by "itrunc" (7435, 7438, 7442) to reinsert a simple storage block into the available list for a device.

7005: It is not clear why the "s_fmod" flag is set here as well as at the end of the procedure (line 7026). Any suggestions?

7006: Observe the locking protocol;

7010: If no free blocks previously existed for the device, restore the situation by setting up a one element list containing an entry for block #0. This value will subsequently be interpreted as an "end of list" sentinel;

7014: If the available list in the "super block" is already full, it is time to write it out onto the FSV. Set "s_flock";

7016: Get a buffer, associated with the block now being entered in the free list;

7019: Copy the contents of the super block list, preceded by a count of the number of valid blocks, into the buffer; write the buffer; unset the lock and "wakeup" anybody waiting,

7025: Add the returned block to the available list.

20.15 iput (7344)

This procedure is one of the most popular in UNIX (called from nearly thirty different places) and its use will have already been frequently observed.

In essence it simply decrements the reference count for the "inode" passed as a parameter, and then calls "prele" (7882) to reset the "inode" lock and to perform any necessary "wakeup"s.

"iput" has an important side effect. If the reference count is going to be reduced to zero, then a release of resources is indicated. This may be simply the core "inode", or both that and the file itself, if the number of links is also zero.

20.16 ifree (7134)

This procedure is called by "iput" (7355) to return a FSV "inode" to the available list maintained in the "super block". If this list is already full (as noted above) or if the list is locked (using "s_ilock") the information is simply discarded.

20.17 iupdat (7374)

This procedure is called by "statl" (6050), "update" (7226) and "iput" (7357) to revise a particular "inode" entry on a FSV. It does nothing if the corresponding core "inode" is not flagged ("IUPD" or "IACC");

The "IUPD" flag may be set by one of

```
unlink (3530)         bmap (6452,6467)
chmod  (3570)         itrunc  (7448)
chown  (3583)         maknode (7462)
link   (5942)         namei   (7609)
writei (6285,6318)    pipe    (7751)
```

The "IACC" flag may be set by one of

```
readi  (6232)       maknode (7462)
writei (6285)       pipe    (7751)
```

The flags are reset by "iput" (7359).

7383: Forget it, if the FSV has been mounted as "read only";

7386: Read the appropriate block containing the FSV "inode" entry. As observed earlier with respect to "iget", note the the use of "bread" instead of "readi", the assumption that the "inode" table begins at block #2 and the convention that valid "inode" numbers begin at one;

7389: Copy the relevant information from the core "inode";

7391: If appropriate, update the time of last access;

7396: If appropriate, update the time of last modification;

7400: Write the updated block back to the FSV.

21 Pipes

A "pipe" is a FIFO character list, which is managed by UNIX as yet another variety of file.

One group of processes may "write" into a "pipe" and another group may "read" from the same "pipe". Hence "pipe"s may be, and are used, primarily for interprocess communication.

By exploiting the concept of a "filter", which is a program which reads an input file and transforms it into an output file, and by using "pipes" to link two or more programs of this type together, UNIX offers its users a surprisingly comprehensive and sophisticated set of facilities.

21.1 pipe (7723)

A "pipe" is created as a result of a system call on the "pipe" procedure.

7728: Allocate an "inode" for the root device;

7731: Allocate a "file" table entry;

7736: Remember the "file" table entry as "r" and allocate a second "file" table entry;

7744: Return user file identifications in R0 and R1;

7746: Complete the entries in the "file" array and the "inode" entry.

21.2 readp (7758)

"pipes" are different from other files in that two separate offsets into the file are kept – one for "read" operations and one for "write" operations. The "write" offset is actually the same as the file size.

7763: the parameter passed to "readp" is a pointer to a "file" array entry, from which an "inode" pointer can be extracted;

7768: "plock" (7862) ensures that only one operation takes place at a time: either "read" or "write";

7776: If a process wishing to write to a "pipe" has been blocked because the pipe was "full" (or rather because the valid part of the file had reached the file limit), it will have signified its predicament by setting the "IWRITE" flag in "ip->i_mode";

7786: Release the lock before going to sleep;

7787: "i_count" is the number of file table entries pointing at the "inode". If this is less than two, then the group of "writers" must be extinct;

7789: A process waiting for input will raise the "IREAD" flag. Since a pipe cannot be full and empty simultaneously, no more than one of the flaqs "IWRITE" or "IREAD" should be set at any one time;

7799: "prele" unlocks the file and "wakes up" any process waiting for the pipe.

21.3 writep (7805)

The structure of this procedure echoes that of "readp" in many respects.

7828: Note that a "writer", which finds that there are no more "readers" left, receives a "signal" just in case he is not monitoring the result of his "write" operation.

(A "reader" in the analogous situation receives a zero character count as the result of the read, and this is the standard end-of-file indication.)

7835: The "pipe" size is not allowed to grow beyond "PIPSIZ" characters. As long as "PIPSIZ" (7715) is no greater than 4096, the file will not be converted to a "large" file. This is highly desirable from the viewpoint of access efficiency.

(Note that "PIPSIZ" limits the "write" offset pointer value. If the "read" offset pointer is not far behind, the true content of the "pipe" may be quite small).

21.4 plock (7862)

Lock the "inode" after waiting if necessary. This procedure is called by "readp" (7768) and "writep" (7815).

21.5 prele (7882)

Unlock the "inode" and "wake" any waiting processes. This procedure is called by several others (especially "iput"), in addition to "readp" and "writep".

Section Five is the final section: last but not least. It is concerned with input/output for the slower, character oriented peripheral deviees.

Such devices share a common buffer pool, which is manipulated by a set of standard procedures.

The set of character oriented peripheral devices are exemplified by the following:

- KL/DL11 interactive terminal
- PC11 paper tape reader/punch
- LP11 line printer.

22 Character Oriented Special Files

Character oriented peripheral devices are relatively slow (< 1000 charaeters per second) and involve character by character transmission of variable length, usually short, records.

A device handler (as its name suggests) is the software part of the interface between a device and the general system. In general, the device handler is the only part of the software which recognises the idiosyncrasies of a particular device.

As far as possible or reasonable, a single device driver is written to serve many separate devices of similar types, and, where appropriate, several such devices simultaneously. The group of "interactive terminals" (with keyboard input and a serial printer or visual display output) can just be coerced with difficulty into a single device driver, as the reader may judge during his perusal of the file "tty.c".

The standard UNIX device handlers for character devices make use of the procedures "putc" and "getc" which store and retrieve characters into and from a standard buffer pool. This will be described in more detail in Chapter Twenty-Three.

The "PDP11 Peripherals Handbook" should be consulted for more complete information on the device controller hardware and the devices themselves.

22.1 LP11 Line Printer Driver

This driver is to be found in the file "lp.c" (Sheets 88, 89). Much of the complexity of this driver is contained in the proeedure "lpcanon" (8879). This procedure is involved in the proper handling of special characters and this is a separate issue from the one we wish to study first.

Initially one may ignore "lpcanon" by assuming that all calls upon it (lines 8859, 8865, 8875) are simply replaced by similar calls upon "lpoutput" (8986). "lpcanon" acts as a "final filter" for characters going to the line printer: handling code conversions, special format characters, etc.

22.2 lpopen (8850)

When a line printer file is opened, the normal calling sequence is followed:

"open" (5774) calls "open1", which (5832) calls "openi", which (6716) calls, in the case of a character special file, "cdevsw[..].d_open". In the case of the line printer, this latter translates (4675) to "lpopen".

8853: Take the error exit if either another line printer file is already open, or if the line printer is not ready (e.g the power is off, or there is no paper, or the printer drum gate is open, or the temperature is too high, or the operator has switched the printer off-line.)

8857: Set the "lp11.flag" to indicate that the file is open, the printer has a "form feed" capability and lines are to be indented by eight characters.

22.3 Notes

(a) "lp11" is a seven word structure defined beginning at line 8829. The first three words of the structure in fact constitute a structure of type "clist" (7908). Only the first element is explicitly manipulated in "lp.c". The next two are used implicitly by "putc" and "getc".

(b) "flag" is the fourth element of the structure. The remaining three elements are

"mcc"	maximum character count
"ccc"	current character count
"mlc"	maximum line count

(c) The line printer controller has two registers on the UNIBUS.

Line Printer Status Register ("lpsr")

bit 15 Set when an error condition exists (see above);

bit 7 "DONE" Set when the printer controller is ready to receive the next character;

bit 6 "IENABLE" Set to allow "DONE" or "Error" to cause an interrupt;

Line Printer Data Buffer Register ("lpbuf")

Bits 6 through 0 hold the seven bit ASCII code for the character to be printed. This register is "write only".

8858: Set the "enable interrupts" bit in the line printer status register;

8859: Send a "form feed" (or "new page") character to the printer, to ensure that characters which follow will start on a new page. (As already noted above, at this stage we are ignoring "lpcanon" and assuming line 8859 to be simply "lpoutput (FORM)". "lpcanon" does things like suppressing all but the first "form feed" in a string of "form feed"s and "new line"s, to avoid wasting paper.);

22.4 lpoutput (8986)

This procedure is called with a character to be printed, as a parameter.

8988: "lp11.cc" is a count of the number of characters waiting to be sent to the line printer. If this is already large enough ("LPHWAT", 8819), "sleep" for a while (so as not to flood the character buffer pool);

8990: Call "putc" (0967) to store the character in a safe place. (The function of "putc" and its companion "getc" is a major topic to be discussed in Chapter Twenty Three.) It should be noted that no check is made that "putc" was successful in storing the character. (There may have been no space in the character buffers.) In practice there seems to be no real problem here, but one can wonder.

8991: Raise the processor priority sufficiently to inhibit the interrupts from the line printer, call "lpstart" and then drop the priority again.

22.5 lpstart (8967)

While the line printer is ready, and while there are still characters stored away in the "safe place", keep sending characters to the printer controller.

The presumption is that while the controller is building up a set of characters for a complete line, the "DONE" bit will reset faster than the CP can feed characters to the controller.

However once a print cycle has been initiated, the "DONE" bit will not be reset again for a period of the order of 100 milliseconds (depending on the speed of the printer).

Note that during this series of data transfers, interrupts will be inhiblted and so "lpint" will not be getting into the act whenever the "DONE" bit is set, except possibly once at the very end when the processor priority is reduced again.

22.6 lpinit(8976)

This procedure is called to handle interrupts from the line printer. As mentioned above, most potential interrupts are ignored by the processor. Those interrupts which are accepted by the CP will be associated with either

(a) completion of a print cycle; or

(b) the printer going ready after a period during which the "Error" bit was set; or

(c) the last transfer in a series of character transfers;

8980: Start transferring characters into the printer buffer again;

8981: Wakeup the process waiting to feed characters to the printer if the number of characters waiting to be sent is either zero or exactly "LPLWAT" (8818).

This latter condition is somewhat puzzling in that it will only **occasionally** be satisfied. The intention surely is "if the number of characters in the list is getting low, start refilling". However if "lpstart" carries out a series of transfers without interruption (at least by "lpint") the number of characters could go from a value greater than "LPLWAT" to one less than this without this test ever being made. Accordingly the waiting process will not be awakened until the list is completely empty. The result could be frequently to delay the initiation of the next print cycle, and hence to allow the printer to run below its rated capacity.

One solution to this problem is to change entirely the buffering strategy for line printers. A less drastic change would involve inventing a new flag, "lp11.wflag" say, replacing lines 8981, 8982 by something like

```
if (lp11.cc <= LPLWAT && lp11.wflag)
{       wakeup (&lp11);
        lp11.wflag = 0
}
```

and replacing line 8989 by

```
{       lp11.wflag++;
        sleep(&lp11, LPRI);
}
```

22.7 lpwrite (8870)

This is the procedure which is invoked as a result of the write system call:

"write" (5722) calls "rdwr", which (5755) calls "writei", which (6287) calls "cdevsw[..].d write", which translates (4675) to "lpwrite".

"lpwrite" takes the non-null characters of a null terminated string recorded in the user area, and passes them to "lpoutput" (via "lpcanon") one at a time.

The list of procedure calls which leads to the invocation of this procedure is similar to that for "lpopen". A "form feed" character is output to clear the current page, and the "open" flag is reset.

22.8 Discussion

"lpwrite" is called one or more times to send a string of characters to the printer. In turn it calls "lpcanon" which calls "lpoutput". If at any point too many characters are stored away, the procss will "sleep" in "lpoutput". Sooner or later "lpoutput" will continue, will store the character in a buffer area, and will then call "lpstart" to send, if possible, a string of characters to the printer controller.

"lpstart" is called both when more characters are available to be sent, and when an interrupt from the printer is taken.

The majority of calls on "lpstart" will in fact achieve nothing. Occasionally (usually when the printer has just completed a print cycle) "lpstart" will be able to send a whole string of characters to the printer controller.

22.9 lpcanon (8879)

This procedure interprets characters being sent to the line printer and make various modifications, insertions and deletions. It thus functions as a filter.

8884: The section of code from here to line 8913 is concerned with character translation when the full 96 character set is not available, and a 64 character set is in use.

Since the capabilities of a printer do not usually change with time, the defined variable "CAP" (8840) must be set once and for all (at a particular installation).

The run-time test on (lp11.flag & CAP) could be replaced by a compile-time test on (CAP) and if the compiler has its "druthers", if CAP turns out to be zero, the whole section of code to line 8913 could be compiled down to nothing.

The present code could be said to plan ahead for a situation where an installation may have two or more printers of different types. Even so there is a basic inconsistency here in the use of "CAP", "IND" and "EJECT" on the one hand, and "EJLINE" and "MAXCOL" on the other. In fact since forms of different sizes are not uncommonly used on a single printer, the last two should not be constants at all, but should be dynamically settable.

8885: Lower case alphabetics are translated by the addition of a constant, which is conveniently defined as "'A' – 'a'";

8887: Certain of the remaining characters are special characters which are printed as a similar character with an overprinted minus sign, e.g. "{" (8889) is printed as "{-";

8909: The "similiar" character is output via a recursive call on "lpcanon", which will increment "lp11.ccc" by one as a side effect;

8910: Decrement the current character count (for the same effect as a "back space" character) and ...

8911: Prepare to output a minus sign;

8915: The "switch" statement beginning here extends to line 8963. Certain characters involved in vertical and horizontal spacing are given special interpretations with delayed actions;

8917: For a horizontal tab character, round the current character count up to the next multiple of eight. Do not output any blank characters immediately;

8921: For a "form feed" or "new line" character, if:

(a) the printer does not have a "page restore" capability; or

(b) the current line is not empty; or

(c) some lines have been completed since the last "form feed" character, then ...

8925: reset "lp11.mcc" to zero;

8926: Increment the completed line count;

8927: Convert a "new line" character to a "form feed" if sufficient lines have been completed on the current page, and the printer has a "form feed" capability;

8929: Output the character, and if was a "form feed", reset number of completed lines zero;

Examination of this code will show that:

(a) Any string of "form feed"s or "new line"s which begins with a "form feed", will, if sent to a printer with "form feed" capability, be reduced to a single "form feed";

(b) A "form feed" character sent to a printer without the "form feed" capability, will cause a new line to be started but will be passed on otherwise without comment.

8934: For "carriage return"s, **and**, note, "form feed"s and "new line"s, reset the current character count to zero or eight, depending on "IND", and return;

8949: For all other characters ...

8950: If a string of "backspace"s (real or contrived) and/or "carriage returns" has been received, output a single "carriage return" and reset the maximum character count to zero;

8954: Provided the count does not exceed the maximum line length, output blank characters to bring the maximum character count to the current character count. (Perhaps these two variables would be more accurately called the "actual character count" and the "logical character count".);

8959: Output the actual character.

22.10 For idle readers: A suggestion

It will be observed that backspaces for overprinting or underscoring characters introduce separate print cycles, and where these features are in heavy use, the effective output rate of the printer may be drastically reduced. If this is considered a serious problem, "lpcanon" could be rewritten to ensure that no more than two print cycles are used per line in such cases.

22.11 PC-11 Paper Tape Reader/Punch Driver

This driver is to be found in the file "pc.c" on Sheets 86, 87. It is simpler than the line printer driver in that there is ro routine analogous to "lpcanon". However it is more complicated in that there is both an input and an output device which can be simultaneously and independently active.

A description of the operation of this device is inciuded in the document "The UNIX I/O System" by D. Ritchie. Certain special features may be noted:

(1) Only one process may open the file for reading at a time, but there is no limit on the nmber of writers;

(2) This routine pays a little more attention to error conditions than the line printer driver, but the treatment is still not exhaustive;

(3) "passc" (8695) knows how many characters are required and returns a negative valie when "enough" is reached;

(4) "pcclose" is careful to flush out any remaining characters in the input queue if and only if it believes the device was opened for input.

23 Character Handling

Buffering for character special devices is provided via a set of four word blocks, each of which provides storage for six characters. The prototype storage block is "cblock" (8140) which incorporates a word pointer (to a similar structure) along with the six characters.

Structures of type "clist" (7908) which contain a character counter plus a head and tail pointer are used as "headers" for lists of blocks of type "cblock".

"cblock"s which are not in current use are linked via their head pointers into a list whose head is the pointer "cfreelist" (3149). The head pointer for the last element of the list has the value "NULL".

A list of "cblock"s provides storage for a list of characters. The procedure "putc" may be used to add a character to the tail of such a list, and "getc", to remove a character from the head of such a list.

Figures 23.1 through 23.4 illustrate the development of a list as characters are deleted and added.

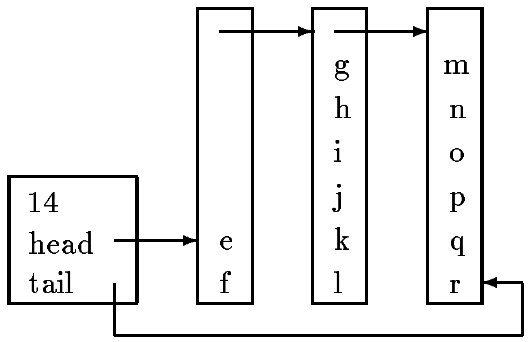

Figure 23.1

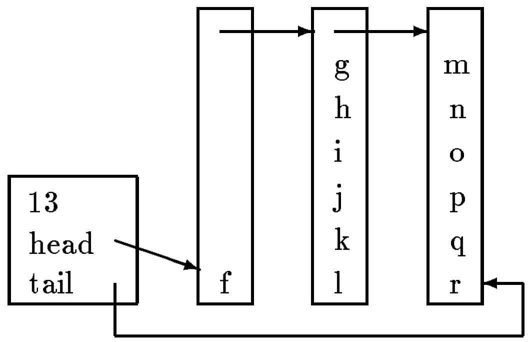

Figure 23.2

Initially the list is assumed to contain the fourteen characters "efghijklmnopqr". Note that the head and tail pointers point to **characters**. If the first character, "e", is removed by "getc", the situation portrayed in Figure 23.1 changes to that of Figure 23.2. The character count has been decremented and the head pointer has been advanced by one character position.

If a further character, "f", is removed from the head of the list, the situation becomes as in Figure 23.3. The character count has been decremented; the first "cblock" no longer contains any useful information and has been returned to "cfreelist"; and the head pointer now points to the first character in the second "cblock".

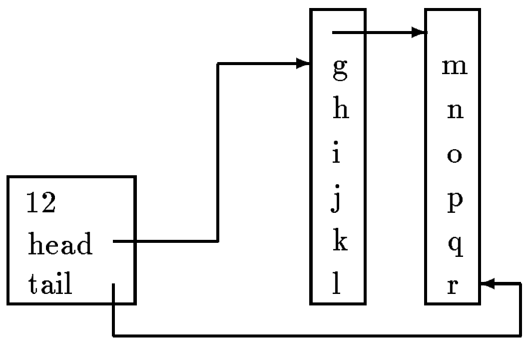

Figure 23.3

The question now poses itself: "how is the difference between the first and second situations detected so that the action taken is always appropriate?":

The answer (if you have not already guessed) involves looklng at the value of the pointer address modulo 8. Since division by eight is easily performed on a binary computer, the reason for the choice of six characters per "cblock" should now also be apparent.

The addition of a character to the list is illustrated in the change between Figure 23.3 and Figure 23.4.

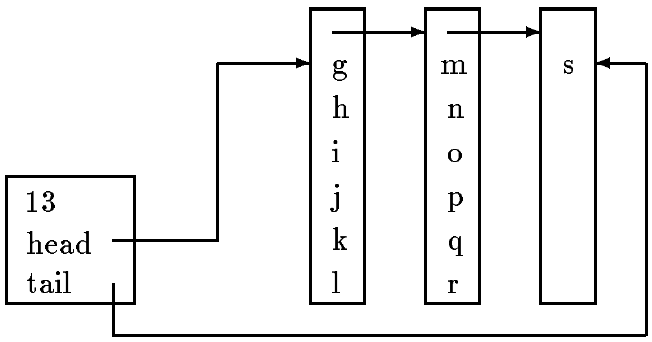

Figure 23.4

Since the last "cblock" ln Figure 23.3 was full, a new one has been obtained from "cfreelist" and linked into the list of "cblock"s. The character count and tail pointer have been adjusted appropriately.

23.1 cinit (8234)

This procedure, which is called once by "main" (1613), links the set of character buffers into the free list, "cfreelist", and counts the number of character device types.

8239: "ccp" is the address of the first word in the array "cfree" (8146);

8240: Round "ccp" up to the next highest multiple of eight, and mark out "cblock" sized pieces, taking care not to exceed the boundary of "cfree".

Note. In general there will be "NCLIST – 1" (rather than "NCLIST") blocks so defined;

8241: Set the first word of the "cblock" to point to the current head of the free list.

Note that "c_next" is defined on line 8141, and that the initial value of "cfreelist" is "NULL".

8242: Update "cfreelist" to point to the new head of the list;

8244: Count the number of character device types. Upon reference to "cdevsw" on Sheet 46, it will be seen that "nchrdev" will be set to 16, whereas a more appropriate value would be 10.

23.2 getc (0930)

This procedure is called by

```
flushtty (8258, 8259, 8264)
canon    (8292)    pcread  (8688)
ttstart  (8520)    pcstart (8714)
ttread   (8544)    lpstart (8971)
pcclose  (8673)
```

with a single argument which is the address of a "clist" structure.

0931: Copy the parameter to r1 and save the initial processor status word and value of r2 on the stack;

0934: Set the processor priority to five (higher than the interrupt priority of a character device);

0936: r1 points to the first word of a "clist" structure (i.e. a character count). Move the second word of this structure (i.e. a pointer to the head character) to r2;

0937: If the list is empty (head pointer is "NULL") go to line 0961;

0938: Move the head character to r0 and increment r2 as a side effect;

0939: Mask r0 to get rid of any extended negative sign;

0940: Store the updated head pointer back in the "clist" structure. (This may have to be altered later.);

0941: Decrement the character count and if this is still positive, go to line 0947;

0942: The list is now empty, so reset the head and tail character pointers to "NULL". Go to line 0952;

0947: Look at the three least significant bits of r2. If these are non-zero, branch to line 0957 (and return to the calling routine forthwith);

0949: At this point, r2 is pointing at the next character position beyond the "cblock". Move the value stored in the first word of the "cblock" (i.e. at r2 – 8), which is the address of the next "cblock" in the list to the head pointer in the "clist". (Note that r1 was incremented as a side effect at line 0941):

0950: The last value stored needs to incremented by two (Consult Figures 23.2 and 23.3);

0952: At this point, a "cblock" determined by r2 is to be returned to "cfreelist". Either r2 points into the "cblock" or just beyond it. Decrement r2 so that r2 will point into the "cblock";

0953: Reset the three least significant bits of r2, leaving a pointer to the "cblock";

0954: Link the "cblock" into "cfreelist";

0957: Restore the values of r2 and PS from the stack and return;

0961: At this point the list is known to be empty because a "NULL" head pointer was encountered. Make sure that the tail pointer is "NULL" also;

0962: Move –1 to r0 as the result to be returned when the list is empty.

23.3 putc (0967)

This procedure is called by

```
canon     (8323)
ttyinput  (8355, 8358)
ttyoutput (8414, 8478)
pcrint    (8730)
pcoutput  (8756)
lpoutput  (8990)
```

with two arguments: a character and the address of a "clist" structure.

"getc" and "putc" have related functions and the codes for the two procedures are similar in many respects. For this reason the code for "putc" will not be examined in detail, but is left for the reader.

It should be noted that "putc" can fail if a new "cblock" is needed and "cfreelist" is empty. In this case a non-zero value (line 1002) is returned rather than a zero value (line 0996).

Note. The procedures "getc" and "putc" discussed here are **NOT** directly related to the procedures d1scussed in the Sections "GETC(III)" and "PUTC(III)" of the UPM.

23.4 Character Sets

UNIX makes use of the full ASCII character set, which is displayed in Section "ASCII(V)" of the UPM. Since knowledge of this character set is often assumed without comment, not always justifiably, some comment here would seem to be in order.

"ASCII" is an acronym for "American Standard Code for Information Interchange".

23.5 Control Characters

The first 32 of the 128 ASCII characters are non-graphic and are intended for the control of some aspect of transmission or display. The control characters explicitly used or recognised by UNIX are

Numeric Value		Description	UNIX Name
004	eot	end of transmission or (control-D)	004
010	bs	back space	010
011	ht	(horizontal) tab	'\t'
012	nl	new line or line feed	FORM
014	np	new page or form feed	'\r'
015	cr	carriage return	'\n'
034	fs	file separator or quit	CQUIT
040	sp	forward space or blank	' '
0177	del	delete	CINTR

It will be noted that the last two of these belong to the last 96 characters or the graphic portion, of the code.

23.6 Graphic Characters

There are 96 graphic characters. Two of these, the space and the delete, are not "visible", and may be ciassified with the control characters.

The graphic characters may be divided into three groups of 32 characters, which may be roughly characterised as

- numeric and special characters
- upper case alphabetic characters
- lower case alphabetic characters.

Of course, since there are only 26 alphabetic characters, the latter two groups include some special characters as well. In particular, the last group includes the following six nonalphabetic characters:

140	`	reverse apostrophe
173	{	left brace
174	|	vertical bar
175	}	right brace
176	~	tilde
177		delete

23.7 Graphic Character Sets

Devices such as line printers or terminals which support **all** the ASCII graphic symbols are often said to support the 96 ASCII character set (though there are only 94 graphics actually involved).

Devices which support all the ASCII graphic symbols except those in the last group of 32, are said to support the 64 ASCII character set. Such devices lack the lower case alphabetics and the symbols listed above, namely "~", "{", "|" and "\}". Note that "delete", since it is not a visible character, can still be supported.

Devices in this latter group may be referred to as "upper case only".

Sometimes some of the graphic symbols may be non-standard, e.g. ← instead of _ and this can be inconvenient, though not usually fatal.

UNIX prefers, as the reader is no doubt well aware, to view the world through "lower case" spectacles. Alphabetic characters received from an "upper case only" terminal are translated immediately upon receipt from upper case to lower case. A lower case alphabetic may subsequently be translated back to upper case if it is preceded by a single backslash. For output to such a terminal, both upper and lower case alphabetic characters are mapped to uppercase.

The conventions for line printers and terminals are different because:

(a) for line printers, horizontal alignment is usually important, and it is possible (without too much difficulty) to print composite, overstruck characters (using the minus sign in this case); and

(b) for terminals, horizontal alignment is not considered to be so important; backspacing to provide overstruck characters does not work on most VDUs; and, since the same graphic conventions are used for both input and output, the symbols should be as convenient to type as possible.

23.8 maptab (8117)

This array is used in the translation of character input from a terminal preceded by a single backslash, "\".

There are three characters, 004 (eot), '#' and '@', which always have special meanings and need to be asserted by a backslash whenever they are to be interpreted literally. These three characters occur in "maptab" in their "natural" locations (i.e. their locations in the ASCII table). Thus for example '#' has code 043 and

```
maptab[043] == 043.
```

The other non-null characters in "maptab" are involved in the translation of input characters from "upper case only" devices and do not occur in their "natural" locations but in the location of their equivalent character, e.g. "\{" occurs in the natural location for "{", since "\{" will be interpreted as "{", etc.

Note the situation regarding alphabetic characters. This is only explicable when it is remembered that the alphabetic characters are all translated to lower case before any backslash is recognised.

23.9 partab (7947)

This array consists of 256 characters, like "maptab". Unfortunately the initialisation of "partab" was omitted from the UNIX Operating System Source Code booklet. It is certainly needed, and so is given now:

```
  char partab [] {

0001,0201,0201,0001,0201,0001,0001,0201,
0202,0004,0003,0205,0005,0206,0201,0001,
0201,0001,0001,0201,0001,0201,0201,0001,
0001,0201,0201,0001,0201,0001,0001,0201,
0200 0000,0000,0200,0000,0200,0200,0000,
0000 0200,0200 0000,0200,0000,0000,0200,
0000,0200,0200 0000,0200,0000,0000,0200,
0200,0000,0000,0200,0000,0200,0200,0000,
0200,0000,0000,0200,0000,0200,0200,0000,
0000,0200,0200,0000,0200,0000,0000,0200,
0000,0200,0200,0000,0200,0000,0000,0200,
0200 0000,0000 0200,0000,0200,0200,0000,
0000 0200,0200 0000,0200,0000,0000,0200,
0200,0000,0000,0200,0000,0200,0200,0000,
0200,0000,0000,0200,0000,0200,0200,0000,
0000,0200,0200,0000,0200,0000,0000,0201
   };
```

Each element of "partab" is an eight bit character, which, with the use of appropriate bitmasks (0200 and 0177), can be interpreted as a two part structure:

bit 7	parity bit;
bits 3-5	not used. Always zero;
bits 0-2	code number.

The parity bit is appended to the seven bit ASCII code when a character is transmitted by the computer, to form an eight bit code with even parity.

The code number is used by "ttyoutput" (8426) to classify the character into one of seven categories for determining the delay which should ensue before the transmission of the next character. (This is particularly important for mechanical printers which require time for the carriage to return from the end of a line, etc.)

24 Interactive Terminals

Our remaining task, to be completed in this and the following chapter, is to consider the code which controls interactive terminals (or "terminals", for short).

A wide variety of terminals is available and several different types may be simultaneously attached to a single computer. Distinguishing characteristics for different classes of terminal include (besides such non-essential features as shape, size and colour):

(a) transmission speed, e.g. 110 baud for an ASR33 teletype, 300 baud for a DECwriter, 2400 baud or 9600 baud for a Visual Display unit ("VD");

(b) graphic character set, notably the full ASCII graphic set and the 64 graphic subset;

(c) transmission parity: odd, even, none or inoperatlve;

(d) output technique: serial printer or visual display;

(e) miscellaneous: combined carriage return/line feed character, half duplex terminal (input characters do not need echoing); recognition of tab characters;

(f) characteristic delays for certain control functions, e.g. carriage returns may not be completed within a single character transmission time, etc.

As well as the wide variety of terminals which are available and in use, there is also a variety of hardware devices which may be used to interface a terminal to a PDP 11 computer. For example:

DL11/KL11	single line, asynchronous interface; 13 standard transmission rates between 40 and 9600 baud;
DJ11	16 line, asynchronous, buffered serial line multiplexer; 11 speeds between 75 and 9600 baud, selectable in four line groups;
DH11	16 line, asynchronous, buffered, serial line multiplexer; 14 speeds, individually selectable; DMA transmission

Each of the above interfaces will work in full or half duplex mode; handle 5, 6, 7 or 8 level codes; generate odd, even or no parity; and generate a stop code of 1, 1.5 or 2 bits.

In addition to the above asynchronous interfaces, there are a number of synchronous interfaces, e.g. DQ11.

Each interface has its own control characteristics and it requires a separate operating system device driver. The common code which can be shared between these is gathered into a single file "tty.c", to be found on Sheets 81 to 85. A set of common definitions is gathered in the file "tty.h" on Sheet 79.

By way of example, Sheet 80 contains the file "kl.c", which constitutes the device driver for a set of DL11/KL11 interfaces. This device driver always needs to be present, since one KL11 interface is invariably included in a system for the the operator's console terminal.

24.1 The 'tty' Structure (7926)

An instance of "tty" is associated with every terminal port to the system (no matter what type of hardware interface is used). A "port" in this context is a place to attach a terminal line. Hence a DL11 supplies only one port, whereas a DJ11 supplies up to sixteen ports.

The "tty" structure consists of sixteen words and includes:

A.	t_dev t_addr	fixed for a particular terminal port;
B.	t_speeds t_erase t_kill t_flags	fixed for a particular terminal. These values may be set by "stty" and interrogated by "gtty";
C.	t_rawq t_canq t_outq	list heads for **three** character queues: the so-called "raw" input, "cooked" input and the output queues;
D.	t_state t_delct t_col t_char	status information which changes frequently during normal processing;

Table 24.1

24.2 Interactive Terminals

The reader should study the information on Sheet 79 carefully. Certain items listed below are not referenced in any essential way in the selection of code examined here.

```
t_char    (7940)    NLDELAY   (7974)
t_speeds  (7941)    TBDELAY   (7975)
```

HUPCL	(7966)	CRDELAY	(7976)
ODDP	(7972)	WOPEN	(7985)
EVENP	(7973)	ASLEEP	(7993)

24.3 Initialisation

Initialisation of the "tty" structures is the responsibility of the various "open" routines in the device drivers, for example, "klopen" (8023).

The items in Group B of Table 24.1 may be changed by a "stty" system call. The current values may be interrogated by a "gtty" system call.

A description of these is contained in the sections, "STTY(II)" and "GTTY(II)" of the UPM. These calls are invoked by the "stty" shell command which is described in the section "STTY(I)".

Since the "stty" and "gtty" system calls require a file descriptor as a parameter, they can only be applied to an "open" character special file.

The two system calls share a good deal of common code. We will trace the progress of an execution of "stty" below and leave the tracing of a similar execution of "gtty" to the reader.

24.4 stty (8183)

This procedure implements the "stty" system call. It copies three words of user parameter information into "u.u_arg[..]" using the parameter supplied as a pointer, and then calls "sgtty".

24.5 sgtty (8201)

8206: Get a validated pointer to a "file" array entry;

8209: Check that the file is a "character special";

8213: Call the appropriate "d_sgtty" routine for the device type. (See Sheet 46.)

Note that the "d_sgtty" routine is "nodev" for the line printer and paper tape reader/punch.

24.6 klsgtty (8090)

This is an example of a "d_sgtty" routine. It calls "ttystty" passing a pointer to the appropriate "tty" structure as a parameter.

24.7 tysty (8577)

A call originating from "stty" will have a second parameter of zero.

8589: Empty all the queues associated with the terminal forthwith. They quite likely contain nonsense;

8591: Reset the speed information (useful in the case of a DH11 interface, but of little interest for the present selection of code);

Reset the "erase" character and the "kill" character. ("kill" here denotes "throw away the current input line".) Note that if these characters are changed away from their normal values of "#" and "@" respectively, no corresponding changes are made to "maptab". Nor should they!),

8593: Reset the "flags" defining some relevant terminal characteristics (see Sheet 79):

flag	**bit**	**if set ...**
XTABS	1	the terminal will not interpret horizontal tab characters correctly;
LCASE	2	the terminal supports only the 64 character ASCII subset;
ECHO	3	the terminal is operating in full duplex mode, and input characters must be echoed back;
CRMOD	4	upon input, a "carriage return" is replaced by a "line feed"; upon output, a "line feed" is replaced by a "carriage return" and a "line feed";
RAW	5	input characters are to be sent to the program exactly as received, without "erase" or "kill" processing, or adjustment for backslash characters.

In addition, the following bits are interrogated by "ttyoutput" (8373) in choosing the delay which should ensue after the character indicated is sent, before sending the next character:

8,9	line feed;
10,11	horizontal tab;
12,13	carriage return;
14	vertical tab or form feed.

24.8 The DL11/KL11 Terminal Device Handler

The file "kl.c" constitutes the device handler for terminals connected to the system via DL11/KL11 interfaces. This group always has at least one member – the operator's console terminal. Hence this device handler will always be present.

Each DL11/KL11 hardware controller provldes an asynchronous, serial interface to connect a single terminal to a PDP 11 system. For more complete details regarding this interface the reader should consult the "PDP11 Peripherals Handbook".

24.9 Device Registers

Each DL11/RL11 unit has a group of four registers occupying four consecutive words on the UNIBS. UNIX maps a structure of type "klregs" (8016) onto each register group.

Receiver Status Register (klrcsr)

bit 7 **Receiver Done.** (A character has been transferred into the Receiver Data Buffer Register.);

bit 6 **Receiver Interrupt Enable.** (When set, an interrupt is caused every time bit 7 is set.);

bit 1 **Data terminal ready;**

bit 0 **Reader Enable. Write only.** (When set, bit 7 is cleared.).

Receiver Data Buffer Register (klrbuf)

bit 15 Error indication, when set.

bits 7-0 Received character, Read only.

Transmitter Status Register (kltcsr)

bit 7 **Transmitter ready.** This is cleared when data is loaded into the Transmitter Data Buffer, and is set when the latter is ready to receive another chatacter;

bit 6 **Transmitter Interrupt Enable.** (when set, causes an interrupt to be generated whenever bit 7 is set.)

Transmitter Data Buffer Register (kltbuf)

bits 7-0 Transmitted data. Write only.

24.10 UNIBUS Addresses

The Receiver Status Register always has its lowest address starting on a four word boundary. (The addresses which follow are all 18 bit octal addresses.)

	Receiver Status	Transmitter Status
Operator's console	777560 →	777566
Group Two	776500 →	776506
	776510 →	77656
	776670 →	776676
Group Three	775610 →	775616
	775620 →	775626
	776170 →	776176

Apart from the operator's console interface which has its own standard UNIBUS location, the interfaces are gathered into two groups (for reasons which are irrelevant here). Within each group, by convention, registers are allocated in consecutive locations starting at the lowest address.

24.11 Software Considerations

"NKL11" (8011) must be set to define, for a particular installation, the number of interfaces in the first two groups, and "NDL11" (8012), the nmber in the third group. Any hardware alterations which changed the actual number of interfaces would have to be reflected in the software by changing and recomiling "kl.c", and relinking the operaing system.

It will be seen that "klopen" calculates the correct kernel mode address (16 bits) for the Receiver Status Register for each interface, and this is stored (8044) into the the "t_addr" element of the appropriate "tty" structure.

24.12 Interrupt Vector Addresses

The vector addresses for the first interface are 060 and 064 (for receiver and transmitter interrupts, respectively). Additional DL11/KL11 interfaces have vector addresses which are always at least 0300, and which are assigned according to rules which take into consideration other interfaces which may be present.

The second word of an interrupt doubiet is the "new processor status" word. The five low order bits of this word may be chosen arbitrarily, and are in fact used to define the minor device number (cf. a similar use to distinguish the various kinds of "traps" – see Sheet 05). A masked version of the new processor status word is provided to the interrupt handling routines as the parameter "dev" (see e.g. line 8070).

24.13 Source Code

We can now turn to a detailed study of the code in the files "kl.c" (Sheet 80) and "tty.c" (Sheets 81 to

85). We shall look first at "opening" and "closing" terminals as character special files and the handling of interrupts. Then in the next chapter we shall look at the receipt of data from the terminal, and finally transmission of data to the terminal.

"klread" (8062), "klwrite" (8066) and "klsgtty" (8090) have already been discussed above.

24.14 klopen (8023)

This procedure is called to "open" a terminal as a character special file. This call is usually made by the program "/etc/init" for each terminal which is to be active in the system. Since child processes inherit the open files of their parents, it is not usually necessary for other processes to "open" the device again. It will be noted that the there is no attempt to stop two unrelated processes having the terminal as an open file simultaneously.

8026: Check the minor device number;

8030: Locate the appropriate "tty" structure;

8031: If the process opening the file has no associated controlling terminal designate the current terminal for this role. (Note that the reference stored is the address of a "tty" structure.):

8033: Store the terminal device number in the "tty" structure

8039: Calculate the address of the appropriate set of device registers for the terminal and store

8045: If the terminal is not already "open", do some initialisation of the "tty" structure ..

8046: "t_state" is set to show the file is "open", so that the next three lines will not be executed if the file is opened a second time, possibly undoing the effect of a "stty" system call:

"t_state" is also set to show "CARR_ON" ("carrier on"). This is a software flag which shows that the terminal is logically enabled, regardless of the true hardware status of the terminal. If "CARR_ON" is reset for a terminal, the system **should** ignore all input from the terminal.

(This does not seem to be entirely true, and this point will be taken up again later.);

8047: The standard terminal is assumed to be unable to interpret horizontal tabs, to support only the 64 character ASCII subset, to run in full duplex mode and to require both "carriage return" and "line feed" characters to provide normal "new line" processing. (Could this be a Model 33 teletype?);

8048: The "erase" and "kill" characters are set according to the UNIX convention;

8051: The Receiver Control Status register is initialised with the pattern "0103" so that the terminal is made ready, reading is enabled and receiver interrupts are enabled;

8052: The Transmitter Control Status register is initialised so that an interrupt will be generated whenever the interface is ready to receive another character.

Note that the "open" routine does not distinguish between the cases where the file is opened for reading only, or writing only, or for both reading and writing.

24.15 klclose (8055)

8057: Find the address of the appropriate "tty" structure in the array of such structures, "kl11" (8015). (This operation may be observed in all the procedures in the second column of Sheet 80, and its relevance should be noted.)

8058: "wflushtty" (8217) allows the output queue for the terminal to "drain" and then flushes the input queue

8059: "t_state" is reset so that "ISOPEN" and "CARR_ON" are no longer true.

24.16 klxint (8070)

This procedure is executed in response to a transmitter interrupt. It should be compared with "pcpint" (8739) and "lpint" (8976). Note that the parameter "dev" is a masked version (low order five bits preserved) of the "new processor status" word in the interrupt vector. Provided the vector was properly initialised, the minor device number will be properly identified.

The second part of the test on line 8074 will be discussed at the end of the next chapter.

24.17 klrint (8078)

This procedure is executed in response to a receiver interrupt. It is not so readily compared with "pcrint" (8719) although similarities certainly exist.

8083: Read the input character from the Receiver Data Buffer register;

8084: Enable the receiver for the next character;

8085: The comment says “hardware botch”. Better believe it;

8086: Pass the character to “ttyinput” to insert it into the appropriate “raw” input queue.

25 The File "tty.c"

In this, the last chapter, the intricacies of interactive terminal handlers are finally unveiled, including:

(a) the handling of the "erase" and kill characters;

(b) the conversion of characters during input and output for upper case only terminals;

(c) the insertion of delays after various special characters such as "carriage return".

The routines "gtty" (8165), "stty" (8183), "sgtty" 82a1) and "ttystty" (8577) were dealt within the previous chapter.

25.1 flushtty (8252)

The purpose of this procedure is to "normalise" the queries associated with a particular terminal. Its effect is to terminate transmission to the terminal forthwith and to throw away any accumulated input characters.

8258: Throw away everything in the "cooked" input queue;

8259: ditto for the output queue;

8260: Wakeup any process waiting to extract a character from the "raw" input queue;

8261: ditto for the output queue;

8263: Raise the processor priority to prevent an interrupt from the terminal while ...

8264: the "raw" input queue is flushed,

8265: the "delimiter count" is properly set to zero.

"flushtty" is called by "wflushtty" (see below) and "ttyinput" (8346, 8350) when either:

(a) the terminal is not operating in "raw" mode and a "quit" or "delete" character is received from the terminal; or

(b) the "raw" input queue has grown unreasonably large (presumably because no process is reading input from the terminal);

25.2 wflushtty (8217)

This procedure waits until the queue of characters for a terminal is empty (because they've all been sent!) and then calls "flushtty" to clean up the input queues.

"wflushtty" is called (3053) by "klclose". This does not happen very often – in fact only when all files referencing the terminal are closed i.e. usually only when the user logs off.

It is also called by "ttystty" (8589) just before the terminal environment parameters are adjusted.

25.3 Character Input

For a program requesting input from a terminal, there is a chain of procedure calls which extends to "ttread" ...

25.4 ttread (8535)

8541: Check that the terminal is **logically** active;

8543: If there are characters in the "cooked" input queue **or** a call on "canon" (8274) is successful ...

8544: transfer characters from the "cooked" input queue until either it is empty or enough characters have been transferred to suit the user's requirements.

25.5 canon (8274)

This procedure is called by "ttread" (8543) to transfer characters from the "raw" input queue to the "cooked" input queue (after processing "erase" and "kill" characters and, in the case of upper case only terminals, processing "escaped" characters, i.e. characters preceded by the character '\'). "canon" returns a non-zero value if the "cooked" input queue is no longer empty.

8284: If the number of delimiters in the "raw" input queue is zero then ...

8285: if the terminal is logically inactive, then just return;

8286: otherwise go to "sleep".

Note that delimiters in this context are characters of all ones (octal value is 377) and are inserted by "ttyinput" (8358).

8291: Set "bp" to point to the third character of the work array, "canonb";

8292: Begin a loop (extending to line 8318) which removes one character from the "raw" queue per cycle;

8293: If the character is a delimiter, reduce the delimiter count by one and exit the loop i.e. go to line 8319;

8297: If the terminal is not operating in "raw" mode ...

8298: If the previous character (note the "bp[−1]" notation!) was not a backslash, '\', execute the code from line 8299 to 8307, otherwise execute the code beginning at line 8309.

Previous character was not a backslash

8299: If the characters is an "erase" and ...

8300: if there is at least one charater to erase, backup the pointer "bp";

8302: Start on the next cycle of the loop beginning at line 8292;

8304: If the character is a "kill", throw away all the characters accumulated for the current line, by going back to line 8290;

8306: If the character is an "eot" (004) (usually generated at the terminal as "control-D"), ignore it (and do not put it inot "canonb") and start on the next cycle;

(If this character occurs at the beginning of a line, then subsequently "ttread" (8544) will find no characters in the "cooked" input queue i.e. it will read a zero length record, which then leads to the program receiving the normal "end of file" indication.)

Previous character was a backslash

8309: If "maptab[c]" is non-zero, and either "maptab[c] == c" or the terminal is upper case only, then ...

8310: if the last character but one was **not** a backslash ('\'), then replace "c" by "maptab[c]" and back up "bp" (so that the backslash will be erased).

Character ready

8315: Move "c" into the next character in "canonb", and if this array is now full, leave the loop.

Line completed

8319: At this point, an input line has been assembled in the array "canonb";

8322: Shift the contents of "canonb" into the "cooked" input queue, and return a "successful" result.

25.6 Notes

(A) The reason why "bp" starts (8291) at the third character of "canonb" can be found on line 8310.

(B) A number of subtleties in the handling of backslashes (which the reader will no doubt have encountered in his practical use of UNIX) are still not immediately apparent. Since "maptab[c]" is zero for "c == '\' " (octal value of 134), all backslashes get copied into "canonb". A single backslash will be subsequently overwritten if the following character is to be asserted (as in the case of '#' or '@' or eot (004), or if the case of an alphabetic character is to be changed for an upper case only terminal.

25.7 ttyinput (8333)

"canon" removes characters from the "raw" input queue. They are put there in the first place by "ttyinput" which is called by "klrint" (8087) whenever an input character is received from the hardware controller.

The parameters passed to "ttyinput" are a character and a reference to a "tty" structure.

8342: If the character is a "carriage return" and the terminal operates with a "carriage return" only (instead of a "carriage return" "line feed" pair) change the character to a "new line";

8344: If the terminal is not operating in "raw" mode and the character is a "quit" or "delete" (7958) then call "signal" (3949) to send a software interrupt to every process which has the terminal as its controlling terminal, flush all the queues associated with the terminal, and return;

8349: If the "raw" input queue has grown excessively large, flush **all** the queues for the terminal and return. (This may seem a trifle harsh at first sight but it will usually be what is required.);

8353: If the terminal has a limited character set, and the character is an upper case alphabetic, translate it into lower case;

8355: Insert the character into the "raw" input queue;

8356: If the terminal is operating in "raw" mode, or the character was a "new line" or "eot" then ...

8357: "wakeup" any process waiting for input from the terminal, place a delimiter character (all ones) also in the "raw" queue and increment the delimiter count Note this is one point where possible failure of "putc" (when there is no buffer space) is explicitly recognised. A failure occurring here would explain why the test on line 8316 may sometimes succeed.

8361: Finally, if the input character is to be echoed i.e. the terminal is running in full duplex mode, insert a copy of the character into the output queue, and and arrange to have it transmitted ("ttstart") back to the terminal.

25.8 Character Output – ttwrite (8550)

This procedure is called via "klwrite" (8067) when output is to be sent to the terminal.

8556: If the terminal is inactive, do nothing;

8558: Loop for each character to be transmitted ...

8560: While there are still an adequate number of characters queued for transmission to the terminal ...

8561: call "ttstart" just in case it is time to send another character to the terminal;

8562: Setting the "ASLEEP" flag here (also in "wflushtty" (8224)) is rather pointless since is is never interrogated and never reset until the file is closed;

8563: Go to sleep. In the meanwhile the interrupt handler will be draining characters from the output queue and sending them down the line to the terminal;

8566: Call "ttyoutput" to insert the character in the output queue and arrange to have it transmitted;

8568: Call "ttstart" again, for luck.

25.9 ttstart

This procedure is called whenever it seems reasonable to try and send the next character to the terminal. It often achieves nothing useful.

8514: See the comment on line 8499. This code is not relevant here;

8518: If the controller is not ready (i.e. bit 7 of the transmitter stalus register is not set) or the necessary delay following the preious character has not yet elapsed, do nothing;

8520: Remove a character from the output queue. If "c" is positive, the queue was not empty (as expected) ...

8521: If "c" is less than "0177" it is a character to be transmitted ...

8522: After setting the parity bit from the corresponding element of the array "partab", write "c" to the transmitter data buffer register to initiate the hardware operation;

8524: Otherwise ("c" > 0177) the character was inserted in the output queue to signal a delay. Call "timeout" (3845) to make an entry in the "callout" list. The result of this will be to initiate an execution of "ttrstrt" (8486) after "c & 0177" clock ticks . It will be seen that "ttrstrt" calls "ttstart" again, and that the manipulation of the "TIMEOUT" flag (8524, 8491) will ensure that if another execution of "ttstart" is initiated in the interim, on behalf of the same terminal, it will (8518) return without doing anything.

25.10 ttrstrt (8486)

See the comment above for line 8524.

25.11 ttyoutput (8373)

This procedure has more comments in the source code and hence requires less explanation than some others. Note the use of recursion (8392) to generate a string of blanks in place of a tab character. Other recursive calls are on lines 8403 and 8413.

25.12 Terminals with a restricted character set

8400: "colp" points to a string of pairs of characters. If the character to be output matches the second character of any of these pairs, the charactcr is replaced by a backslash followed by the first character of the pair;

8407: Lower case alphabetics are converted to upper case alphabetics by the addition of a constant.

Note. The conversion here should be compared wth the handling of the reverse problem on input. Here we have an algorithm which clearly trades space (no table analogue to "maptab") for time (a serial search through the string on line 8400). A space conserving approach could be adopted in "canon" but the problem is rather more complicated there.

8414: Insert the character into the output queue. If perchance, "putc" fails for lack of buffer space, don't worry about inserting any subsequent delay, or updating the system's idea of the current printing column;

8423: Set "colp" to point to the "t_col" character of the "tty" structure, i.e. "*colp" has a value which is the ordinal number of the column which has just been printed;

8424: Set "ctype" to the element of "partab" corresponding to the output character "c";

8426: Mask out the significant bits of "ctype" and use the result as the "switch" index;

8428: (Case 0) The common situation! Increment "t_col";

8431: (Case 1) Non-printing characters. This group consists of the first, third and fourth octet of the ASCII character set, plus "so" (016), "si" (017) and "del" (0177). Don't increment "t_col";

8434: (Case 2) Backspace. Decrement "t_col" unless it is already zero;

8439: (Case 3) Newline. Obviously "t_col" should be set to zero. The main problem is to calculate the delay which should ensue before another character is sent.

For a Model 37 teletype, this depends on how far the print mechanism has progressed across the page. The value chosen is at least a tenth of a second (six clock ticks) and may be as much as $((132/16) + 3)/60 = 0.19$ seconds.

For a VT05, the delay is 0.1 second. For a DECwriter it is zero because the terminal incorporates buffer storage and has a double speed "catch up" print mode;

8451: (Case 4) Horizontal tab. Assign the value of bits 10, 11 of "t_flags" to "ctype";

8453: For the only non-trivial case recognised ("c" == 1 or Model 37 teletype), calculate the the number of positions to the next tab stop (via the obscure calculation of line 8454). If this turns out to be four columns or less, take it as zero;

8458: Round "*colp" (i.e. the value pointed to by "colp"!) to the next multiple of 8 less one;

8459: Increment "*colp" to be an exact multiple of eight;

8462: (Case 5) Vertical Motion. If bit 14 is set in "t_flags", make the delay as long as possible, i.e. 0177 or 127 clock ticks, i.e. just over two seconds;

8467: (Case 6) Carriage Return. Assign the value of bits 12, 13 of "t_flags" to "ctype";

8469: For the first class, allow a delay of five clock ticks;

8472: For the second class, allow a delay of ten clock ticks;

8475: Set the "*colp" (the last column printed) to zero.

Before leaving the file "tty.c", there are two matters which deserve further examination.

25.13 A. The test for 'TTLOWAT' (Line 8074)

On line 8074 in "klxint", a test is made whether to restart any processes waiting to send output to the terminal. The test is successful if the number of characters is zero or if it is equal to "TTLOWAT".

If the number of characters is between these values, no "wakeup" is performed until the queue is completely empty, with the strong likelihood that there will then be a hiatus in the flow of output to the terminal. Since temporary interruptions to the flow of output are quite frequently observed in practice and represent a source of occasional irritation if nothing more, one may reasonably enquire "is there any way the character count can get from being greater than "TTLOWAT" to below it, without this being detected at line 8074?"

Quite clearly there is, since each call on "ttstart" can decrement the queue size, and only one such call is followed by the test. Thus if the call on "ttstart" from one of "ttrstrt" (8492) or "ttwrite" (8568) happens to cross the boundary, a delay will result. The probability that this will happen is small, but finite and hence the event is likely to be observed in any reasonably long output sequence.

There are two other situations in which "ttstart" is called which seem to be satisfactory. At "ttwrite" (8561) the queue is at its maximum extent; and at "ttyinput" (8363) there is a preceding call on "ttyoutput" which usually (but not invariably!) will have added a character to the output queue.

25.14 B. Inactive Terminals

When the last special file for a terminal is closed, "klclose" (8055) is called and resets (8059) the "ISOPEN" and "CARR_ON" flags. However the "read enable" bit of the receiver control status register is not reset, so that incoming characters may still be received and will be stored away (8087) in the terminal's "raw" input queue by "klrint" (8078), and "ttyinput" (8333), which do not test the "CARR_ON" flag, to see if the terminal is logically connected.

These characters may accumulate for a long time and clog up the character buffer storage. Only when the "raw" input queue reaches 256 characters ("TTYHOG", 8349) will the contents of this queue be thrown away. It does seem therefore, that a statement to disable reader interrupts should be included in "klclose" before line 8058.

25.15 Well, that's all, folks ...

Now that you, oh long-suffering, exhausted reader have reached this point, you will have no trouble in disposing of the last remaining file, "mem.c" (Sheet 90). And on this note, we end this discussion of the UNIX Operating System Source Code.

Of course there are lots more device drivers for your patient examination, and in truth the whole UNIX Timesharing System Source Code has hardly been scratched. So this is not really

THE END

26 Suggested Exercises

Any operating system design involves many subjective and ad hoc judgements on the part of system's designers. At many places in the UNIX source code you will find yourself wondering "Why did they do it that way?", "What would happen if I changed this?"

The following exercises express some of these questions. Some can be answered from an examination of the source code alone after a study in more depth; others require some experimental probing and measurement, for which read-only access to the file "/dev/kmem" via terminal will prove invaluable; and still others really require the construction and testing of experimental versions of the operating system.

26.1 Section One

1.1 Devise changes to "malloc" (2528) to implement the Best Fit algorithm.

1.2 Rewrite the procedure "mfree" (2556) to render its function more easily discernible by the reader.

1.3 Investigate the adequacy of the sizes of the arrays "coremap" and "swapmap" (0203, 0204). How should "CMAPSIZ" and "SWAPSIZ" change when "NPROC" is increased?

1.4 Prove that "malloc" and "mfree" jointly solve the memory aliocation problem correctly.

1.5 By monitoring the contents of "coremap", estimate the efficiency with which main memory is utilised. Estimate also the cost of compacting "in use areas" of main memory from time to time to reduce memory fragmentation.

Hence decide whether it would be worthwhile to extend the present memory allocation scheme to include memory compaction.

1.6 In setting the first six kernel page description registers, UNIX does not make use of all the hardware protection features that are available e.g. some pages which contain only pure text could be made read-only. Devise changes to the code to maximise the use of the available hardware protection.

1.7 Compile the program

```
char *init ''/etc/init'';
main ( ) {
execl (init, init, 0);
while (1);
}
```

and compare the result with the contents of the array "icode" (1516).

1.8 Investigate the size required for kernel mode stack areas. Hence show that the 367 word area which is provided is adequate.

1.9 If main memory consists of several independent memory modules and one of these, not the last, is down, "main" will not include memory modules beyond the one which is down, in the list of available space in "coremap". Devise some simple changes to "main" to handle this situation. what other parts of the system would also need revision?

1.10 Rewrite the routines "estabur" (1650) and "sureg" (1739) so that they will work as efficiently as possible on the PDP11/40. How often are these routines used in practice? Would it really be worthwhile trying to implement your improved versions?

1.11 Investigate the overheads involved in initiating a new process. Perform a series of measurements for a set of different sized programs under different conditions.

1.12 Evaluate the following scheme which is intended by Ken Thompson as the basis for a revised scheduling algorithm:

A number "p" is kept for each process, stored as "p_cpu". "p" is incremented by one every clock tick that the process is found to be executing. "p" therefore accumulates the CPU usage. Every second, each value of "p" is replaced by four fifths of its value rounded to the nearest integer. This means that "o" has values which are bounded by zero and the solution of the equation $k = 0.8 * (k + HZ)$ i.e. 4*HZ. Hence if HZ is 50 or 60, and "p" is integerised, "p" can be stored in one byte.

1.13 The "proc" table is always searched via a direct linear search. As the table size is increased, the search overheads also increase. Survey the alternatives for improving the search mechanism, when "NPROC" is say 300.

26.2 Section Two

2.1 Explain in detail how the system reacts to a floating point trap which occurs when the processor is in kernel mode.

2.2 When a process dies, a "zombie" record is written to disk, and is subsequently read back by the parent. Devise a scheme for passing back the necessary information to the parent which will avoid the overhead of the two i/o operations.

2.3 Document "backup" (1012).

2.4 It is relatively easy using the "shell" to set up a set of asynchronous processes which will flood your terminal with useless output. Trying to stop these processes individually can be a problem, since their identifying numbers may not be known. Use of the command "kill 0" is usually an act of sheer desperation. Devise an alternative scheme, e.g. based on the use of messages such as "kill –99", which will be effective, but more selective.

2.5 Design a form of coroutine jump whlch will cause control to pass more efficiently between a program which is being traced, and its parent.

26.3 Section Three

3.1 Rewrite the procedure "sched" to avoid the use of "goto" statements.

3.2 Modify "sched" so that the text segment and data segment for a program will possibly be allocated in separate main memory areas if a single large area is not immediately available.

3.3 If the system crashes and must be "rebooted" the contents of the buffers which were not written out at the time of the crash are lost.

However if a core dump is taken, the contents of the buffers can be obtained and hence the contents of the disk can be brought completely up to date. Outline a detalled plan for carrying out this scheme. How effective do you think it would be?

3.4 Explain why the buffer areas declared on line 4720 are 514, and not 512, characters long.

3.5 Explain how deadlock situations may arise if there are too few "large" buffers available. What measures can you suggest to alleviate the problem, assuming that increasing the number of buffers is not possible.

26.4 Section Four

4.1 Devise a scheme for labelling file system volumes and checking these labels when the volumes are mounted.

4.2 Discuss the problems of supporting ANSI standard labelled tapes under UNIX, and propose a solution.

4.3 Design a scheme for providing index sequential access to files.

4.4 The emergence of the "sticky bit" (see "CHMOD(I)" in the PM) confirms that there are some residual advantages in allocating all the space for a file contiguously. Discuss the merits of making "contiguous files" more generally available.

4.5 Devise a technique to measure the efficiency of pipes. Apply the technique and report your results.

4.6 Devise modifications to "pipe.c" which will make pipes more efficient according to the following scheme: whenever the "read" pointer is greater than 512, rotate the non-null block numbers in the "inode" ana decrease both the "read" and "write" pointers by 512.

5.1 By monitoring the number of free buffers or otherwise, determine whether the number of character buffers provided at your installation is adequate.

5.2 Perform measurements and/or experiments to determine whether the character buffer blocks would be more efficiently utilised if they consisted of four or eight characters, rather than six, per block.

5.3 Redesign the line printer driver to handle overprinting and backspacing more efficiently in the sense of minimising the number of print cycles.

5.4 Document "mmread" (0916) and "mmwrite" (9042).

26.5 General

6.1 The easiest way to vary the main memory space used by the operating system is to vary "NBUF". If this is forbidden, propose the best way to:

(a) reduce the space required by 500 words;

(b) utilise an additional 500 words.

6.2 Discuss the merits of "C" as a systems programming language. What features are missing? or superfluous?