The UNIX Fourth Edition Source Code Commentary

A Complete Guide to Understanding the UNIX v4 Operating System

Briam Rodriguez

20260113.127 Edition

Contents

This book a labor,
Of love for my dear mother,
And all of mankind.

The UNIX Fourth Edition Source Code Commentary

A Complete Guide to Understanding the UNIX v4 Operating System

Based on the original Bell Labs source code by Ken Thompson and Dennis Ritchie[1]

About This Book

This book provides a comprehensive, line-by-line commentary on the UNIX Fourth Edition source code. UNIX v4 represents one of the most elegant and influential pieces of software ever written—an entire operating system in roughly 10,000 lines of code that you can actually understand.

Unlike modern operating systems with millions of lines of code, UNIX v4 is small enough for one person to comprehend completely. This book will guide you through every major component, explaining not just *what* the code does, but *why* it was designed that way.

0.0.1. Why UNIX v4?

UNIX Fourth Edition (November 1973)[2] occupies a unique position in computing history:

- **First C-based UNIX** — While earlier versions were written in assembly, v4 was rewritten in C, making it the ancestor of all modern UNIX systems
- **Complete and comprehensible** — The entire kernel fits in about 9,000 lines of C and assembly
- **Mature yet minimal** — It includes multiprocessing, a hierarchical filesystem, device drivers, and a shell, but without the complexity that accumulated in later versions
- **Influential design** — The concepts introduced here (everything is a file, small tools that compose, simple process model) became the foundation of modern operating systems

[1] UNIX Fourth Edition was released in November 1973. The source code in this book comes from a tape sent to the University of Utah in June 1974, containing the V4 distribution with minor updates. See Ken Thompson's letter to Martin Newell dated May 31, 1974.

[2] The fourth edition was released in November 1973. The tape recovered from the University of Utah was sent in June 1974. See the TUHS Wiki for edition timeline.

0.0.2. Prerequisites

To get the most from this book, you should have:

- **Basic C programming knowledge** — You don't need to be an expert, but you should understand pointers, structures, and function calls
- **Understanding of fundamental OS concepts** — Processes, files, memory allocation, and the distinction between user and kernel mode
- **Familiarity with assembly language** — Helpful but not required; we explain the PDP-11 assembly as we encounter it

0.0.3. What You Will Learn

By the end of this book, you will understand:

- How an operating system boots and initializes itself
- How processes are created, scheduled, and terminated
- How the filesystem stores and retrieves data
- How system calls transfer control between user programs and the kernel
- How device drivers interface hardware to the rest of the system
- How the shell parses and executes commands
- How the C compiler transforms source code into executables

0.0.4. Source Files Location

All source code references are relative to the `unix_v4/` directory:

```
unix_v4/
├── usr/sys/            # Kernel source
│   ├── ken/            # Ken Thompson's kernel code
│   │   ├── main.c      # Kernel entry point
│   │   ├── slp.c       # Process scheduling, sleep/wakeup
│   │   ├── trap.c      # Trap and interrupt handling
│   │   ├── sys1.c      # fork, exec, exit, wait
│   │   ├── sys2.c      # open, read, write, close
│   │   ├── sys3.c      # seek, stat, dup
│   │   ├── sys4.c      # chmod, chown, time, etc.
│   │   ├── rdwri.c     # readi(), writei()
│   │   ├── fio.c       # File descriptor operations
│   │   ├── iget.c      # Inode operations
│   │   ├── nami.c      # Path name resolution (namei)
│   │   ├── alloc.c     # Disk block allocation
│   │   ├── clock.c     # Clock interrupt handler
│   │   └── sig.c       # Signal handling
│   ├── dmr/            # Dennis Ritchie's device drivers
│   │   ├── bio.c       # Buffer cache
│   │   ├── tty.c       # Terminal line discipline
```

```
│   │   ├── kl.c          # Console driver
│   │   ├── rk.c          # RK05 disk driver
│   │   ├── mem.c         # Memory devices (/dev/mem, /dev/null)
│   │   ├── malloc.c      # Core memory allocator
│   │   └── pipe.c        # Pipe implementation
│   ├── conf/             # Configuration and machine-dependent code
│   │   ├── low.s         # Interrupt vectors
│   │   └── mch.s         # Machine-dependent assembly
│   └── *.h               # Header files (proc.h, user.h, inode.h, etc.)
├── usr/source/           # User programs
│   ├── s1/               # Section 1 - User commands (cat, ls, echo)
│   ├── s2/               # Section 2 - System utilities (sh, login, init)
│   ├── s3/               # Section 3 - Libraries
│   └── s7/               # Section 7 - Miscellaneous
├── usr/c/                # C compiler source
│   ├── c0*.c             # Compiler pass 0 (lexer, parser)
│   └── c1*.c             # Compiler pass 1 (code generator)
├── bin/                  # Binary executables
├── lib/                  # Libraries and compiler passes
└── etc/                  # System configuration (init, passwd)
```

0.0.5. Reading This Book

Each chapter follows a consistent structure:

1. **Overview** — What the chapter covers and why it matters
2. **Source Files** — Which files we'll examine
3. **Prerequisites** — What you should understand first
4. **Concepts** — Background needed to understand the code
5. **Code Walkthrough** — Line-by-line analysis of key functions
6. **Key Data Structures** — Annotated structure definitions
7. **How It All Fits Together** — Diagrams and explanations
8. **Experiments** — Things to try yourself
9. **Summary** — Key takeaways
10. **Further Reading** — Related chapters and external resources

0.0.6. Notation Conventions

Throughout this book:

- `function()` — Function names appear in monospace with parentheses
- `variable` — Variable and structure names appear in monospace
- `file.c:123` — File references include line numbers where helpful
- `0177776` — Octal numbers (common in PDP-11 code) start with 0
- **Bold** — Key terms on first introduction
- *Italic* — Emphasis or book/paper titles

0.0.7. A Note on the C Dialect

The C in UNIX v4 predates the 1978 K&R standard. You'll notice:

```
/* Assignment operators are reversed */
x =+ 1;    /* Modern: x += 1 */
x =| 4;    /* Modern: x |= 4 */
x =- y;    /* Modern: x -= y (ambiguous with x = -y!) */

/* No void type - functions return int by default */
sleep(chan, pri)    /* No return type declaration */
{
    ...
}

/* Parameter types declared separately */
sleep(chan, pri)
int chan;           /* Parameter type declarations */
int pri;            /* after the parameter list */
{
    ...
}

/* =0 initializes to zero */
int x 0;            /* Modern: int x = 0; */
```

We'll point out these differences as they arise.

0.0.8. Acknowledgments

The Original Authors

- **Ken Thompson and Dennis Ritchie** — For creating UNIX and making computing what it is today
- **Bell Labs** — For fostering an environment where this work could flourish

The UNIX v4 Tape Recovery

The source code studied in this book comes from a magnetic tape sent from Ken Thompson to Martin Newell at the University of Utah in June 1974. Newell was conducting pioneering computer graphics research (including the Utah Teapot). The tape survived because Jay Lepreau held onto it when it would have been discarded; it was rediscovered among his papers in July 2025.

Timeline of recovery (from Angelo Papenhoff's 39C3 presentation):

- **June 1974** — Tape sent from Ken Thompson to Martin Newell
- **Jay Lepreau** — Saved the tape from being discarded (found among his papers)
- **28 July 2025** — Found by Aleks Maricq (University of Utah)
- **Rob Ricci** (University of Utah) — Spread the word about the discovery
- **Thalia Archibald** (University of Utah) — Researched the tape's background and history

- **18 Dec 2025** — Driven to the Computer History Museum by Jon Duerig
- **19 Dec 2025** — Read and uploaded to archive.org by Al Kossow, Len Shustek, and Thalia Archibald
- **20 Dec 2025** — Booted on emulator by Angelo Papenhoff (squoze.net)
- **24 Dec 2025** — Booted on real PDP-11/45 by Jacob Ritorto
- **26 Dec 2025** — Booted on real PDP-11/40 by Ashlin Inwood

Archives and Community

- **The Computer History Museum** — For preserving this important history
- **The Internet Archive** — For hosting the recovered tape image (utah_unix_v4_raw)
- **The UNIX Heritage Society** — For maintaining archives of early UNIX
- **squoze.net** — For the UNIX v4 restoration and emulation documentation (squoze.net/UNIX/v4)

This Book

- **Thalia Archibald** — For historical corrections and feedback
- **Warren Toomey** — For technical corrections and feedback

"UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity." — Dennis Ritchie

---

How to Use This Book

0.0.1. For Sequential Reading

If you're new to operating systems internals, read the chapters in order. Part I provides essential background, Part II covers the kernel core, and each subsequent part builds on what came before.

0.0.2. For Reference

If you're already familiar with operating systems and want to understand specific subsystems, each chapter is relatively self-contained. Use the cross-references to fill in background as needed.

0.0.3. With the Source Code

This book is meant to be read alongside the actual source code. Keep the `unix_v4/` directory[3] open and follow along. The code is small enough that you can (and should) read all of it.

Let's begin.

[3] The `unix_v4/` directory refers to the UNIX v4 source code available from squoze.net, where Angelo Papenhoff has made the restored source code available for download.

Part I.

Foundation

1. Chapter 1: Introduction

1.1. Overview

Before we dive into the source code, we need to understand the world that created UNIX. The design decisions in UNIX v4 weren't made in a vacuum—they were shaped by the hardware constraints of 1973, the culture of Bell Labs, and the hard lessons learned from Multics. Understanding this context transforms the code from a historical artifact into a masterclass in pragmatic engineering.

This chapter covers the history and philosophy behind UNIX, setting the stage for everything that follows.

1.2. Prerequisites

None—this is where we begin.

1.3. The Birth of UNIX

1.3.1. From Multics to UNIX

In 1964, MIT, General Electric, and Bell Labs began an ambitious project called **Multics** (Multiplexed Information and Computing Service). The goal was to create a computing utility—a system that would provide computing power like a utility company provides electricity, serving hundreds of users simultaneously.

Multics was revolutionary in concept but troubled in execution. It aimed to do everything: security, reliability, hierarchical filesystems, dynamic linking, and more. By 1969, Bell Labs withdrew from the project. It was over budget, behind schedule, and growing increasingly complex.

But two researchers who had worked on Multics—**Ken Thompson** and **Dennis Ritchie**—had tasted what a good operating system could be. They wanted something simpler.

1.3.2. Space Travel and the PDP-7

Ken Thompson had written a game called "Space Travel" that simulated the solar system. Running it on the GE-635 mainframe cost $75 per game in computer time. Thompson found a little-used **PDP-7**

minicomputer and decided to port his game to it.

To make development easier, he needed an operating system. Over a few weeks in 1969, Thompson wrote a simple filesystem, a process model, a command interpreter, and a few utilities. His wife took the kids to visit her parents for a month; Thompson allocated one week each to the kernel, the shell, the editor, and the assembler.

This was UNIX—though it wasn't called that yet. The name came from Brian Kernighan as a pun on Multics: where Multics was "multiplexed," UNIX was "uniplexed," doing one thing well.

1.3.3. The PDP-11 and the Rewrite

In 1970, the Computing Science Research Center at Bell Labs acquired a **PDP-11/20**. The PDP-11 was a revolutionary machine—clean architecture, orthogonal design, and a memory management unit that could support multiple users.

Thompson rewrote UNIX for the PDP-11. But assembly language was tedious, and the system was hard to modify. Thompson wanted a high-level language.

He tried FORTRAN first—it was a disaster. Then he created a language called **B**, based on BCPL. B was typeless, which worked fine on word-addressed machines but poorly on the byte-addressed PDP-11.

Dennis Ritchie extended B into **C**, adding types, structures, and other features. In 1973, Thompson and Ritchie rewrote UNIX in C—the system we're studying in this book.

1.3.4. Why This Version Matters

UNIX v4 (November 1973)[1] is special:

1. **First C version** — This is where UNIX became portable and where C proved itself as a systems programming language
2. **Minimal but complete** — It has multiprocessing, a hierarchical filesystem, device drivers, pipes, and a shell. Yet the kernel is under 9,000 lines of C.
3. **Before the accretion** — Later versions added networking, virtual memory, and hundreds of other features. v4 has the core ideas without the cruft.
4. **The design crystallized** — The fundamental architecture that would influence all future UNIX systems was established by v4.

[1] UNIX v4 was released November 1973. The source code studied in this book comes from a tape sent to Martin Newell at the University of Utah in June 1974. See Thalia Archibald's research at unix-history.

1.4. The Bell Labs Environment

Understanding UNIX requires understanding Bell Labs in the early 1970s.

1.4.1. The Research Culture

Bell Labs was a unique institution. AT&T's telephone monopoly generated enormous profits, a portion of which funded fundamental research with no expectation of immediate commercial return. Researchers had freedom to pursue interesting problems.

The Computing Science Research Center (Department 1127) was particularly unusual. It had about a dozen researchers, no hierarchy to speak of, and no product deadlines. Thompson, Ritchie, and their colleagues could spend years on work that might never ship.

This freedom produced remarkable results: the transistor, information theory, the laser, and UNIX all came from Bell Labs.

1.4.2. Constraints and Creativity

But freedom didn't mean unlimited resources. The PDP-11/20 had:

- **24KB of memory** (later systems had more, but not much)
- **A 2.5MB RK05 disk pack**
- **No memory protection** initially (MMU came with PDP-11/40 and /45)
- **No virtual memory** — What you had was what you got

These constraints forced elegant solutions. When you can't add more code, you make the code you have work harder. Every data structure in UNIX v4 is minimal. Every algorithm is simple. There's no room for bloat.

1.4.3. The Users

UNIX was used for real work at Bell Labs. The first killer app was text processing—Thompson and Ritchie convinced management to buy a PDP-11 by promising to develop a document preparation system for the patents department.

The users were sophisticated programmers who could (and did) read the source code. When something was wrong, they fixed it. This tight feedback loop between developers and users produced a system refined through daily use.

1.5. Design Philosophy

UNIX embodies a coherent design philosophy that emerged from Thompson and Ritchie's Multics experience and the constraints they worked within.

1.5.1. Simplicity

The overriding principle is simplicity. When in doubt, leave it out. When forced to add something, add the simplest thing that could possibly work.

Consider the process model. A process has a process ID, a parent process ID, memory, open files, and not much else. There's no process priority inheritance, no real-time scheduling, no mandatory access control. Just the basics.

Or consider the filesystem. Files are byte streams—the system doesn't know or care about record formats. Directories are files that contain names and inode numbers. That's it.

1.5.2. Everything is a File

In UNIX, almost everything is accessed through the file interface: open, read, write, close. This includes:

- Regular files on disk
- Directories
- Devices (terminals, disks, printers)
- Inter-process communication (pipes)

This unification means programs don't need special cases for different kinds of I/O. `cat` doesn't know if it's reading from a file or a terminal or a pipe—and it doesn't need to.

1.5.3. Small, Sharp Tools

UNIX encourages small programs that do one thing well. Instead of one large program that does everything, you have many small programs that can be combined.

This is made possible by two innovations:

1. **Text streams** — Programs communicate through streams of text, not binary formats
2. **Pipes** — The output of one program can be connected to the input of another

```
who | wc -l      # Count logged-in users
ls | grep foo    # Find files matching "foo"
```

1.5.4. Worse is Better

Richard Gabriel later characterized the UNIX philosophy as "worse is better"—a design that is simpler but less complete will often be more successful than one that is more complex but more correct.

Consider error handling. In UNIX, system calls that fail return -1 and set a global error code. The caller must check every return value. This is inconvenient, error-prone, and not at all elegant.

But it's simple. The kernel doesn't need complex exception handling. User programs can ignore errors if they want. And in practice, it works well enough.

This philosophy runs throughout UNIX:

- The shell is simple (no job control in v4)
- The filesystem is simple (no permissions more complex than read/write/execute)
- The process model is simple (no threads, just processes)

Is this worse? In some sense, yes. Is it better? In practice, often yes—because simple systems are easier to understand, implement, debug, and extend.

1.6. The Cast of Characters

UNIX was created primarily by two people, with significant contributions from others.

1.6.1. Ken Thompson

Thompson wrote the first UNIX on the PDP-7, then rewrote it for the PDP-11. He created the B programming language, the first UNIX shell, and many core utilities. In the source code, files in `usr/sys/ken/` contain Thompson's kernel code.

Thompson's code is characterized by extreme brevity. Functions are short, variable names are terse, and there's no wasted motion. Looking at `slp.c` (sleep/wakeup and scheduling), you'll see algorithms so tight they border on cryptic—until you understand them, and then they seem inevitable.

1.6.2. Dennis Ritchie

Ritchie created the C programming language and rewrote much of UNIX in it. His code lives in `usr/sys/dmr/` and includes the device drivers and buffer cache. Ritchie also wrote the definitive documentation for C and for UNIX.

Ritchie's code tends to be slightly more expansive than Thompson's, with more comments and clearer structure. The buffer cache (`bio.c`) is a model of clarity.

1.6.3. Others

- **Brian Kernighan** — Named UNIX, contributed utilities, co-authored “The C Programming Language”
- **Doug McIlroy** — Invented pipes, led the research group
- **Joe Ossanna** — Created troff, the text formatter
- **Lorinda Cherry** — Statistical tools and document analysis

1.7. What We’ll Study

The UNIX v4 source code breaks down as follows:

1.7.1. The Kernel (~9,000 lines)

```
usr/sys/ken/     # Thompson's kernel code
    main.c       # Boot and initialization
    slp.c        # Scheduling, context switch, sleep/wakeup
    trap.c       # Trap and interrupt handling
    sysent.c     # System call table
    sys1.c       # Process syscalls: fork, exec, exit, wait
    sys2.c       # File syscalls: open, read, write, close
    sys3.c       # More file: seek, dup, pipe
    sys4.c       # Misc: time, signal, stat
    fio.c        # File descriptor layer
    rdwri.c      # Inode read/write
    iget.c       # Inode cache
    nami.c       # Path resolution (namei)
    alloc.c      # Disk block allocation
    clock.c      # Clock interrupt handler
    sig.c        # Signals
    text.c       # Shared text segments
    subr.c       # bmap and other utilities
    prf.c        # printf for kernel

usr/sys/dmr/     # Ritchie's drivers and buffer cache
    bio.c        # Buffer cache
    tty.c        # Terminal handling
    kl.c         # Console driver
    rk.c         # RK05 disk driver
    mem.c        # /dev/mem, /dev/null
    malloc.c     # Core memory allocator
    pipe.c       # Pipe implementation
    ...          # Other device drivers
```

1.7.2. User Programs

```
usr/source/s2/sh.c       # The shell
usr/source/s1/cat.s      # cat in assembly
usr/source/s1/ls.s       # ls in assembly
```

```
usr/c/                  # The C compiler
```

1.7.3. Header Files

```
usr/sys/param.h    # System parameters
usr/sys/proc.h     # Process structure
usr/sys/user.h     # User structure (u.)
usr/sys/inode.h    # Inode structure
usr/sys/buf.h      # Buffer structure
usr/sys/file.h     # Open file table
usr/sys/filsys.h   # Superblock structure
usr/sys/tty.h      # Terminal structure
```

1.8. Summary

- UNIX emerged from the Multics project, preserving the good ideas while discarding the complexity
- Ken Thompson created UNIX in 1969 on a PDP-7; it was rewritten in C for the PDP-11 in 1973
- UNIX v4 is the first C-based version—complete, minimal, and comprehensible
- The Bell Labs environment provided freedom and constraints that shaped the design
- The UNIX philosophy emphasizes simplicity, the file abstraction, and composable tools
- The source code we'll study is about 9,000 lines of kernel code plus user programs

1.9. Further Reading

- Ritchie, D.M. and Thompson, K., "The UNIX Time-Sharing System," Communications of the ACM, July 1974
- Ritchie, D.M., "The Evolution of the Unix Time-sharing System," AT&T Bell Laboratories Technical Journal, October 1984
- Kernighan, B. and Pike, R., "The Unix Programming Environment," Prentice-Hall, 1984
- Salus, P., "A Quarter Century of UNIX," Addison-Wesley, 1994

Next: Chapter 2 — The PDP-11 Architecture

2. Chapter 2: The PDP-11 Architecture

2.1. Overview

You cannot understand UNIX v4 without understanding the PDP-11. The hardware shapes the software at every level—from the way system calls work to why there are exactly 8 memory segments per process. This chapter covers the PDP-11 architecture as it relates to UNIX, focusing on what you need to know to read the source code.

2.2. Source Files

File	Purpose
`usr/sys/seg.h`	Memory management register definitions
`usr/sys/reg.h`	Register save area offsets
`usr/sys/conf/mch.s`	Machine-dependent assembly routines
`usr/sys/conf/low.s`	Interrupt vector table

2.3. Prerequisites

- Basic understanding of computer architecture (registers, memory, addresses, interrupt handling)
- Familiarity with any assembly language (concepts transfer)

2.4. The PDP-11 Family

The PDP-11 was Digital Equipment Corporation's most successful minicomputer line, introduced in 1970. UNIX v4 ran primarily on the **PDP-11/45**, though it supported the /40 as well.

Key characteristics:

- **16-bit architecture** — Words are 16 bits, addresses are 16 bits

- **64KB address space** — 2^16 = 65,536 bytes maximum
- **Byte-addressable** — Can access individual bytes, not just words
- **Memory-mapped I/O** — Devices accessed through memory addresses
- **Orthogonal instruction set** — Most instructions work with any addressing mode

The 64KB address space was the crucial constraint. An entire UNIX system—kernel, user processes, and I/O devices—had to fit in this space. The Memory Management Unit (MMU) made this possible by providing virtual address translation.

2.5. Registers

The PDP-11 has eight 16-bit general-purpose registers:

```
r0      General purpose, also function return value
r1      General purpose
r2      General purpose
r3      General purpose
r4      General purpose
r5      General purpose, frame pointer by convention
r6 (sp) Stack pointer
r7 (pc) Program counter
```

All registers are equivalent for most operations, but convention and some instructions treat them specially:

- **r0** — Return value from functions; first argument in some calling conventions
- **r5** — Frame pointer by C compiler convention
- **r6/sp** — Stack pointer; push/pop operations use this implicitly
- **r7/pc** — Program counter; can be used as a general register for tricks

The register save area in the kernel (from `reg.h`) shows how registers are stored on the stack during a trap:

```c
/* reg.h - offsets from saved r0 */
#define R0   (0)
#define R1   (-2)
#define R2   (-9)
#define R3   (-8)
#define R4   (-7)
#define R5   (-6)
#define R6   (-3)
#define R7   (1)
#define RPS  (2)    /* Processor Status word */
```

2.6. The Processor Status Word (PS)

The PS register at address `0177776` contains the processor state:

```
 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| CM  | PM  | RS  |     |  IPL   | T| N| Z| V| C|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

CM    - Current Mode (00=kernel, 11=user)
PM    - Previous Mode
RS    - Register Set (PDP-11/45 has two register sets)
IPL   - Interrupt Priority Level (0-7)
T     - Trace bit
N,Z,V,C - Condition codes (negative, zero, overflow, carry)
```

Traps vs. Interrupts: Both traps and interrupts transfer control to the kernel through similar mechanisms, but they differ in origin. An **interrupt** is an asynchronous event from external hardware (disk ready, clock tick, keyboard input). A **trap** is a synchronous event caused by the currently executing instruction—either intentionally (system calls use the `trap` instruction) or due to errors (illegal instruction, memory fault). The PDP-11 handles both through the same vector mechanism, saving the PC and PS on the stack before jumping to a handler address.

The key fields for UNIX are the following:

2.6.1. Current/Previous Mode (bits 14-15, 12-13)

The PDP-11 has two modes:

- **Kernel mode (00)** — Full access to all memory and instructions
- **User mode (11)** — Restricted access, memory mapped through MMU

When a trap occurs, the hardware saves the current PS and sets the new mode to kernel. The **Previous Mode** field records where we came from, so we know whether to access user or kernel space.

2.6.2. Interrupt Priority Level (bits 5-7)

The IPL controls which interrupts are blocked:

```
/* From mch.s */
spl0()  /* IPL = 0, all interrupts enabled */
spl1()  /* IPL = 1, block level 0 */
spl4()  /* IPL = 4, block disk interrupts */
spl5()  /* IPL = 5, block most device interrupts */
spl6()  /* IPL = 6, block clock interrupts */
spl7()  /* IPL = 7, block all interrupts */
```

The kernel raises the IPL to protect critical sections:

```
/* Typical pattern in the kernel */
spl6();           /* Block interrupts */
/* ... critical section ... */
spl0();           /* Re-enable interrupts */
```

2.7. Memory Layout

With only 64KB of address space, UNIX uses the MMU to multiplex physical memory among:

- The kernel
- User processes (one at a time in memory)
- I/O device registers

2.7.1. Physical Address Space

```
000000 - 157777   RAM (up to 56KB, varies by system)
160000 - 177777   I/O Page (device registers)
```

The top 8KB is always reserved for device registers, limiting usable RAM to 56KB in a basic configuration. Systems with extended memory used the MMU to access more.

2.7.2. Virtual Address Space (per process)

Each process sees:

```
000000 - 017777   Segment 0 (8KB)
020000 - 037777   Segment 1 (8KB)
040000 - 057777   Segment 2 (8KB)
060000 - 077777   Segment 3 (8KB)
100000 - 117777   Segment 4 (8KB)
120000 - 137777   Segment 5 (8KB)
140000 - 157777   Segment 6 (8KB) - User structure in kernel
160000 - 177777   Segment 7 (8KB) - I/O Page
```

2.8. The Memory Management Unit (MMU)

The MMU (KT-11 option on PDP-11/45) translates virtual addresses to physical addresses using **8 segment registers** per mode.

2.8.1. Segmentation Registers

From `seg.h`:

```
/* KT-11 registers */
#define KISA    0172340    /* Kernel I-space Address registers */
#define UISD    0177600    /* User I-space Descriptor registers */
#define UISA    0177640    /* User I-space Address registers */

#define RO      02         /* Read-only */
#define RW      06         /* Read-write */
#define WO      04         /* Write-only (not used) */
#define ED      010        /* Expand downward (for stack) */

struct { int r[8]; };      /* 8 registers per set */
```

Each segment has two registers:

Page Address Register (PAR) — Base physical address (in 64-byte blocks)

Page Descriptor Register (PDR) — Access control and length:

```
 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |        PLF         |W |        |ED| ACF |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

PLF  Page Length Field (bits 14-8, in 64-byte blocks)
W    Written (bit 7, set by hardware when page modified)
ED   Expand Downward (bit 3, 1 for stack segments)
ACF  Access Control Field (bits 2-1: 01=RO, 11=RW)
```

2.8.2. Address Translation

Virtual address → Physical address:

```
Virtual: | Segment (3 bits) | Block (7 bits) | Byte (6 bits) |
         |     0-7          |    0-127       |    0-63       |

Physical = PAR[segment] * 64 + block * 64 + byte
         = (PAR[segment] + block) * 64 + byte
```

Example: Virtual address `0037777` (octal) - Segment: `0037777 >> 13` = 0 (segment 0) - Offset: `0037777 & 017777` = 017777 (8191 decimal) - If PAR[0] = 0100, physical = 0100 * 64 + 8191 = 4096 + 8191 = 12287

2.8.3. How UNIX Uses Segments

In UNIX v4, a typical user process layout:

```
Segment 0-2: Text (code) - Read-only, shared
Segment 3-5: Data + BSS + Heap - Read-write
Segment 6:   (not used by user)
Segment 7:   Stack - Read-write, expands downward

Kernel sees:
Segment 0-5: Kernel code and data
Segment 6:   User structure (u.) for current process
Segment 7:   I/O page
```

The function `estabur()` in `main.c` sets up user segments:

```
estabur(nt, nd, ns)    /* text, data, stack sizes */
{
    /* Set up text segments (read-only) */
    while(nt >= 128) {
        *dp++ = (127<<8) | RO;    /* 8KB read-only */
        ...
    }

    /* Set up data segments (read-write) */
    while(nd >= 128) {
        *dp++ = (127<<8) | RW;    /* 8KB read-write */
        ...
    }

    /* Set up stack segment (expand down) */
    *--dp = ((128-ns)<<8) | RW | ED;
}
```

2.9. Trap and Interrupt Mechanism

2.9.1. Vector Table

The PDP-11 uses a **vector table** in low memory. Each vector is two words: new PC and new PS.

From `low.s`:

```
. = 0^.
    br    1f        / Reset: branch to start
    4

/ trap vectors (addresses 4-36)
    trap; br7+0.    / 4: bus error
    trap; br7+1.    / 10: illegal instruction
    trap; br7+2.    / 14: BPT (breakpoint)
    trap; br7+3.    / 20: IOT trap
    trap; br7+4.    / 24: power fail
    trap; br7+5.    / 30: EMT (emulator trap)
    trap; br7+6.    / 34: TRAP (system call!)
```

When a trap occurs:

1. Hardware pushes PC and PS onto the kernel stack
2. Hardware loads new PC and PS from the vector
3. Execution continues at the new PC (the `trap` routine)

2.9.2. The System Call Trap

UNIX uses the **TRAP** instruction (vector at address 034) for system calls:

```
/ User code to make a system call
sys  write        / This is really: trap #4
```

The trap handler (`trap` in `mch.s`) saves registers and calls the C function `_trap()`:

```
trap:
    mov  PS,-4(sp)
    ...
    jsr  r0,call1; _trap    / Call C trap handler
```

2.9.3. Interrupt Handling

Device interrupts work similarly but have their own vectors:

```
. = 60^.
    klin; br4       / Console keyboard input, priority 4
    klou; br4       / Console keyboard output

. = 100^.
    kwlp; br6       / Clock interrupt, priority 6

. = 220^.
    rkio; br5       / RK disk interrupt, priority 5
```

Each device interrupt calls a C function:

```
klin:  jsr  r0,call; _klrint   / Call klrint() in C
kwlp:  jsr  r0,call; _clock    / Call clock() in C
rkio:  jsr  r0,call; _rkintr   / Call rkintr() in C
```

2.10. Key Machine Instructions

2.10.1. Data Movement

```
mov   src,dst     / Move word
movb  src,dst     / Move byte
clr   dst         / Clear (set to 0)
```

2.10.2. Arithmetic

```
add   src,dst     / dst = dst + src
sub   src,dst     / dst = dst - src
inc   dst         / dst++
dec   dst         / dst--
cmp   src,dst     / Compare (set condition codes)
tst   src         / Test (compare with 0)
```

2.10.3. Logical

```
bic   src,dst     / Bit clear: dst &= ~src
bis   src,dst     / Bit set: dst |= src
bit   src,dst     / Bit test: src & dst (set flags only)
```

2.10.4. Branching

```
br    addr        / Branch always
beq   addr        / Branch if equal (Z=1)
bne   addr        / Branch if not equal (Z=0)
bge   addr        / Branch if >= (signed)
blt   addr        / Branch if < (signed)
bhi   addr        / Branch if > (unsigned)
blos  addr        / Branch if <= (unsigned)
```

2.10.5. Subroutines

```
jsr   r5,addr     / Jump to subroutine, save return in r5
                  / Actually: push r5, r5=pc, pc=addr
rts   r5          / Return: pc=r5, pop r5
```

2.10.6. Stack Operations

```
mov   r0,-(sp)    / Push r0
mov   (sp)+,r0    / Pop into r0
```

2.10.7. Special

```
sys   n           / System call (trap instruction)
rti               / Return from interrupt
wait              / Wait for interrupt
reset             / Reset all devices
```

2.11. Addressing Modes

The PDP-11's power comes from its **orthogonal addressing modes**. Any instruction can use any mode for source and destination:

Mode	Syntax	Name	Meaning
0	Rn	Register	Use register directly
1	(Rn)	Deferred	Memory at address in Rn
2	(Rn)+	Autoincrement	Use (Rn), then Rn += 2
3	@(Rn)+	Autoincr Deferred	Pointer at (Rn), then Rn += 2
4	-(Rn)	Autodecrement	Rn -= 2, then use (Rn)
5	@-(Rn)	Autodecr Deferred	Rn -= 2, use pointer at (Rn)
6	X(Rn)	Index	Memory at Rn + X
7	@X(Rn)	Index Deferred	Pointer at Rn + X

Since PC is r7, modes 2, 3, 6, 7 with PC create additional modes:

Mode	Syntax	Name	Meaning
27	#n	Immediate	Literal value n
37	@#addr	Absolute	Memory at address
67	addr	Relative	Memory at PC + offset
77	@addr	Relative Deferred	Pointer at PC + offset

Examples from UNIX source:

```
mov   r0,r1           / Register to register
mov   (r0),r1         / Memory[r0] to r1
mov   (r0)+,r1        / Memory[r0] to r1, r0 += 2
mov   -(sp),r0        / Push r0 onto stack
mov   (sp)+,r0        / Pop stack into r0
mov   4(sp),r0        / Stack[sp+4] to r0
mov   $100,r0         / Immediate 100 to r0
mov   _variable,r0    / Global variable to r0
```

2.12. The User Structure Address

A critical constant in UNIX:

```
/ From mch.s
_u    = 140000
```

The user structure (`u.`) is always mapped at virtual address `0140000` (octal) in the kernel. This is segment 6 of kernel space. When the kernel switches processes, it changes the segment 6 mapping to point to the new process's user structure.

This allows code like:

```
u.u_error = EINVAL;    /* Always refers to current process */
```

2.13. Key Machine-Dependent Functions

From `mch.s`, functions the kernel calls:

2.13.1. Save and Restore Context

```
savu(u.u_rsav)    /* Save sp and r5 */
retu(addr)        /* Restore sp and r5, change segment 6 */
aretu(u.u_qsav)   /* Restore for signal/longjmp */
```

2.13.2. Memory Access

```
fubyte(addr)      /* Fetch byte from user space */
fuword(addr)      /* Fetch word from user space */
subyte(addr, v)   /* Store byte to user space */
suword(addr, v)   /* Store word to user space */
```

2.13.3. Memory Operations

```
copyin(src, dst, n)   /* Copy from user to kernel */
copyout(src, dst, n)  /* Copy from kernel to user */
copyseg(src, dst)     /* Copy 64-byte segment */
clearseg(seg)         /* Zero a 64-byte segment */
```

2.13.4. Interrupt Priority

```
spl0()    /* Enable all interrupts */
spl5()    /* Block most device interrupts */
spl6()    /* Block clock */
spl7()    /* Block all interrupts */
```

2.14. Summary

- The PDP-11 is a 16-bit architecture with 64KB address space
- 8 general-purpose registers (r0-r7), with r6=sp and r7=pc
- The PS register controls mode (kernel/user) and interrupt priority
- The MMU provides 8 segments per mode, each up to 8KB
- UNIX uses segments for text, data, stack, user structure, and I/O
- Traps and interrupts use a vector table at low memory
- The orthogonal instruction set allows any addressing mode with any instruction
- Machine-dependent assembly in `mch.s` provides context switch and memory access primitives

2.15. Experiments

1. **Examine vectors**: In the source, trace what happens when a bus error (vector 4) occurs vs. a system call (vector 034).
2. **Segment calculation**: Given a user program with 4KB text, 2KB data, and 1KB stack, calculate what values `estabur()` would put in the segment registers.
3. **Mode tracing**: Follow the PS word through a system call: What mode are we in at each step?

2.16. Further Reading

- PDP-11 Processor Handbook, Digital Equipment Corporation
- Chapter 4: Boot Sequence — See how the MMU is initialized
- Chapter 7: Traps and System Calls — Detailed walkthrough of trap handling

---

Next: Chapter 3 — Building the System

3. Chapter 3: Building the System

3.1. Overview

This chapter explains how UNIX v4 is compiled and linked into a bootable kernel. Understanding the build process reveals the structure of the system—which pieces are written in C, which require assembly, and how device drivers are configured. It also introduces the toolchain that UNIX uses to build itself.

3.2. Source Files

File	Purpose
`usr/sys/conf/mkconf.c`	Configuration generator
`usr/sys/conf/rc`	Build script
`usr/sys/conf/mch.s`	Machine-dependent assembly
`usr/sys/conf/low.s`	Generated interrupt vectors
`usr/sys/lib1`	Ken's compiled kernel objects
`usr/sys/lib2`	DMR's compiled driver objects
`lib/c0`, `lib/c1`	C compiler passes
`lib/crt0.o`	C runtime startup
`lib/libc.a`	C library
`bin/as`	Assembler
`bin/ld`	Linker

3.3. Prerequisites

- Chapter 2: PDP-11 Architecture (understanding of address space and segments)

3.4. The UNIX Toolchain

UNIX v4 includes a complete, self-hosting toolchain:

3.4.1. The C Compiler

The C compiler is a **two-pass** system:

```
source.c -> [c0] -> intermediate -> [c1] -> source.s
```

- **c0** (`lib/c0`) — Lexer and parser; produces intermediate code
- **c1** (`lib/c1`) — Code generator; produces PDP-11 assembly
- **c2** (`lib/c2`, optional) — Peephole optimizer

The `cc` command orchestrates these passes:

```
cc source.c
```

This runs:

1. `c0 source.c /tmp/ctm1` — Parse, produce intermediate
2. `c1 /tmp/ctm1 /tmp/ctm2` — Generate assembly
3. `as /tmp/ctm2` — Assemble to object
4. `ld crt0.o source.o -lc` — Link with runtime and library

3.4.2. The Assembler

The assembler (`as`) translates PDP-11 assembly into object files:

```
as source.s        # Produces a.out
as -o output.o source.s
```

The assembler is itself written in assembly (`usr/source/s1/as*.s`)—a remarkable piece of bootstrapping.

3.4.3. The Linker

The linker (`ld`) combines object files and resolves symbols:

```
ld -x file1.o file2.o -lc    # -x strips local symbols
```

The linker produces **a.out** format executables:

```
a.out header:
    magic number (0407, 0410, 0411)
    text size
    data size
```

```
    bss size
    symbol table size
    entry point
    unused
    relocation suppression flag
```

3.5. Building the Kernel

The kernel build process has three main steps:

1. Configure devices (generate `l.s` and `c.c`)
2. Assemble machine-dependent code
3. Link everything together

3.5.1. Directory Structure

```
usr/sys/
├── ken/          # Thompson's C source
├── dmr/          # Ritchie's C source
├── conf/         # Configuration
│   ├── mkconf.c  # Config generator source
│   ├── mkconf    # Config generator binary
│   ├── mch.s     # Machine code
│   └── rc        # Build script
├── lib1          # Compiled ken/*.c
├── lib2          # Compiled dmr/*.c
└── *.h           # Header files
```

3.5.2. Step 1: Configure Devices

The `mkconf` program generates device configuration. You run it interactively:

```
$ mkconf
rk        # Include RK05 disk driver
tm        # Include TM11 tape driver
console   # Console (required, always present)
mem       # Memory device
clock     # System clock (required)
^D        # End of input
```

`mkconf` produces two files:

l.s — Interrupt vector table:

```
/ low core
br4 = 200
br5 = 240
br6 = 300
```

```
br7 = 340

. = 0^.
    br   1f
    4

/ trap vectors
    trap; br7+0.     / bus error
    trap; br7+1.     / illegal instruction
    trap; br7+2.     / bpt-trace trap
    trap; br7+3.     / iot trap
    trap; br7+4.     / power fail
    trap; br7+5.     / emulator trap
    trap; br7+6.     / system entry (system call!)

. = 40^.
.globl  start
1:  jmp  start

. = 60^.
    klin; br4        / console input
    klou; br4        / console output

. = 100^.
    kwlp; br6        / clock interrupt
    kwlp; br6

. = 220^.
    rkio; br5        / RK disk interrupt

/ interface code to C
.globl  call, trap
.globl  _klrint
klin:   jsr  r0,call; _klrint
.globl  _klxint
klou:   jsr  r0,call; _klxint
.globl  _clock
kwlp:   jsr  r0,call; _clock
.globl  _rkintr
rkio:   jsr  r0,call; _rkintr
```

c.c — Device switch tables:

```
/*
 * Copyright 1974 Bell Telephone Laboratories Inc
 */

int (*bdevsw[])()       /* Block device switch */
{
    &nulldev, &nulldev, &rkstrategy, &rktab,
    0
};

int (*cdevsw[])()       /* Character device switch */
{
```

```
    &klopen,  &klclose,  &klread,  &klwrite,  &klsgtty,
    &nulldev, &nulldev,  &mmread,  &mmwrite,  &nodev,
    &nulldev, &nulldev,  &rkread,  &rkwrite,  &nodev,
    0
};

int rootdev  {(0<<8)|0};   /* Root device: rk0 */
int swapdev  {(0<<8)|0};   /* Swap device: rk0 */
int swplo    4000;         /* Swap starting block */
int nswap    872;          /* Swap size in blocks */
```

3.5.3. mkconf Internals

Looking at `mkconf.c`, we see how it works:

```
struct tab {
    char *name;        /* Device name */
    int count;         /* Number configured */
    int address;       /* Interrupt vector address */
    int key;           /* CHAR, BLOCK, INTR flags */
    char *codea;       /* Vector table code */
    char *codeb;       /* Interrupt glue (part 1) */
    char *codec;       /* Interrupt glue (part 2) */
    char *coded;       /* Block switch entry */
    char *codee;       /* Char switch entry */
} table[] {
    "console",
    -1, 60, CHAR+INTR,
    "\tklin; br4\n\tklou; br4\n",
    ".globl\t_klrint\nklin:\tjsr\tr0,call; _klrint\n",
    ".globl\t_klxint\nklou:\tjsr\tr0,call; _klxint\n",
    "",
    "\t&klopen, &klclose, &klread, &klwrite, &klsgtty,",

    "rk",
    0, 220, BLOCK+CHAR+INTR,
    "\trkio; br5\n",
    ".globl\t_rkintr\n",
    "rkio:\tjsr\tr0,call; _rkintr\n",
    "\t&nulldev,\t&nulldev,\t&rkstrategy,\t&rktab,",
    "\t&nulldev, &nulldev, &rkread, &rkwrite, &nodev,",
    ...
};
```

The device table encodes everything needed to generate both assembly and C code for each device.

3.5.4. Step 2: Build Script

The `rc` build script ties everything together:

```
if ! -r l.s -o ! -r c.c goto bad
as l.s                      # Assemble interrupt vectors
mv a.out ../low.o
as mch.s                    # Assemble machine code
mv a.out ../mch.o
cc -c c.c                   # Compile configuration
mv c.o ../conf.o
mv l.s low.s
mv c.c conf.c
ld -x ../low.o ../mch.o ../conf.o ../lib1 ../lib2
mv a.out ../../../unix      # Final kernel
chmod 644 low.s conf.c ../low.o ../mch.o ../conf.o ../../../unix
echo rm mkconf.c and rc when done
exit
: bad
echo l.s or c.c not found
```

3.5.5. Step 3: Understanding lib1 and lib2

The kernel C code is pre-compiled into two libraries:

lib1 (~47KB) — Ken Thompson's kernel code:

- `main.c` — Kernel initialization
- `slp.c` — Scheduler, context switch
- `trap.c` — Trap handler
- `sys1.c` - `sys4.c` — System calls
- `fio.c`, `rdwri.c` — File I/O
- `iget.c`, `nami.c` — Inode operations
- `alloc.c` — Block allocation
- `clock.c` — Clock handler
- `sig.c` — Signals
- And more…

lib2 (~40KB) — Dennis Ritchie's driver code:

- `bio.c` — Buffer cache
- `tty.c` — Terminal handling
- `kl.c` — Console driver
- `rk.c` — RK05 disk driver
- `malloc.c` — Memory allocation
- `mem.c` — Memory device
- And device drivers…

3.5.6. The Link Order Matters

```
ld -x ../low.o ../mch.o ../conf.o ../lib1 ../lib2
```

- `low.o` — Must be first (contains vectors at address 0)
- `mch.o` — Machine code, includes `start:` entry point
- `conf.o` — Device configuration
- `lib1` — Core kernel
- `lib2` — Drivers (depend on kernel functions)

3.6. The Boot Process Overview

When the kernel is loaded:

1. **Bootstrap loader** reads kernel from disk into memory
2. Execution starts at `start:` in `mch.s`
3. `start:` initializes the MMU segments
4. `start:` calls `main()` in C
5. `main()` initializes memory, creates process 0 and 1
6. Process 1 execs `/etc/init`

From `mch.s`:

```
.globl  start, _end, _edata, _main
start:
    bit  $1,SSR0
    bne  start           / loop if restart
    reset

/ initialize system segments
    mov  $KISA0,r0
    mov  $KISD0,r1
    mov  $200,r4
    clr  r2
    mov  $6,r3
1:
    mov  r2,(r0)+
    mov  $77406,(r1)+    / 4k rw
    add  r4,r2
    sob  r3,1b

/ initialize user segment (segment 6)
    mov  $_end+63.,r2
    ash  $-6,r2
    bic  $!1777,r2
    mov  r2,(r0)+        / ksr6 = sysu
    mov  $usize-1\<8|6,(r1)+

/ initialize io segment (segment 7)
```

```
    mov  $7600,(r0)+     / ksr7 = IO
    mov  $77406,(r1)+    / rw 4k

/ get a sp and start segmentation
    mov  $_u+[usize*64.],sp
    inc  SSR0            / Enable MMU!

/ clear bss
    mov  $_edata,r0
1:
    clr  (r0)+
    cmp  r0,$_end
    blo  1b

/ clear user block
    mov  $_u,r0
1:
    clr  (r0)+
    cmp  r0,$_u+[usize*64.]
    blo  1b

/ set up previous mode and call main
    mov  $30000,PS
    jsr  pc,_main

/ on return, enter user mode at 0
    mov  $170000,-(sp)
    clr  -(sp)
    rti
```

3.7. Compiling User Programs

User programs use the standard toolchain:

```
cc program.c           # Compile and link
cc -c module.c         # Compile only
cc -o prog a.o b.o -lc # Link with C library
```

The C library (`lib/libc.a`) provides:

- System call wrappers (`open`, `read`, `write`, etc.)
- String functions (`strlen`, `strcmp`, etc.)
- I/O functions (`printf`, `getchar`, etc.)
- Memory functions (`alloc`, etc.)

The C runtime (`lib/crt0.o`) provides the entry point that calls `main()` and exits properly.

3.8. Rebuilding the Kernel

To modify and rebuild the kernel:

1. **Edit source files** in ken/ or dmr/
2. **Recompile changed files**:

```
cc -c -O slp.c    # Compile with optimization
```

3. **Update the library**:

```
ar r ../lib1 slp.o
```

4. **Reconfigure if devices changed**:

```
chdir ../conf
mkconf < config   # config file has device list
```

5. **Relink**:

```
sh rc
```

6. **Install new kernel**:

```
cp /unix /ounix       # Save old kernel
cp unix /unix         # Install new
sync
```

7. **Reboot** to test the new kernel

3.9. The Complete Picture

```
Source Files:
ken/*.c, dmr/*.c --> [cc] --> lib1, lib2 (pre-compiled)

Configuration:
mkconf --> l.s (vectors)
       --> c.c (device switches)

Assembly:
mch.s --> [as] --> mch.o
l.s   --> [as] --> low.o

Compilation:
c.c --> [cc] --> conf.o

Linking:
low.o + mch.o + conf.o + lib1 + lib2 --> [ld] --> unix

Boot:
bootstrap --> load unix --> start: --> main() --> init
```

3.10. Summary

- The UNIX kernel is built from pre-compiled libraries plus generated configuration
- `mkconf` generates interrupt vectors (`l.s`) and device switch tables (`c.c`)
- The toolchain (cc, as, ld) is self-hosting—UNIX builds itself
- Machine-dependent code in `mch.s` initializes the MMU and calls `main()`
- The link order places interrupt vectors at address 0
- Modifying the kernel requires recompiling, updating libraries, and relinking

3.11. Experiments

1. **Trace mkconf**: Read through `mkconf.c` and trace what happens when you add an "rk" device. What code gets generated?
2. **Examine a.out**: Use `nm` and `size` on the unix binary to see symbol layout and section sizes.
3. **Startup sequence**: Follow the code path from `start:` in `mch.s` to `main()` in `main.c`. What must happen before C code can run?

3.12. Further Reading

- Chapter 4: Boot Sequence — Detailed walkthrough of `main()`
- Chapter 18: The C Compiler — How cc, c0, c1 work
- Chapter 19: The Assembler — The as implementation

Next: Part II — The Kernel

Chapter 4: Boot Sequence

Part II.

The Kernel

4. Chapter 4: Boot Sequence

4.1. Overview

This chapter traces the path from power-on to a running UNIX system. We follow the code from the hardware reset through `main()`, watching as the kernel discovers memory, creates the first processes, and hands control to `/etc/init`. By the end, you'll understand how UNIX bootstraps itself from nothing.

4.2. Source Files

File	Purpose
`usr/sys/conf/mch.s`	`start:` entry point, MMU setup
`usr/sys/ken/main.c`	`main()`, memory init, process 0 & 1
`usr/sys/ken/slp.c`	`newproc()`, `sched()`
`usr/sys/systm.h`	Global variables
`usr/sys/param.h`	System constants

4.3. Prerequisites

- Chapter 2: PDP-11 Architecture (MMU, segments, traps)
- Chapter 3: Building the System (kernel structure)

4.4. The Bootstrap Process

Before `main()` can run, several things must happen:

```
Power On
    ↓
Bootstrap loader (in ROM or toggled in)
    ↓
```

```
Load kernel from disk to memory
    ↓
Jump to start: (in mch.s)
    ↓
Initialize MMU segments
    ↓
Enable memory management
    ↓
Clear BSS and user area
    ↓
Call main()
```

4.4.1. The start: Entry Point

From `mch.s`, the first kernel code to execute:

```
.globl  start, _end, _edata, _main
start:
    bit  $1,SSR0
    bne  start           / Loop if MMU already on (restart)
    reset                / Reset all devices
```

The first instruction checks if the MMU is already enabled—if so, this is a restart after a crash, and we loop forever (a deliberate hang to allow debugging).

4.4.2. Initialize Kernel Segments

```
/ initialize system segments
    mov  $KISA0,r0       / Kernel segment address registers
    mov  $KISD0,r1       / Kernel segment descriptor registers
    mov  $200,r4         / 8KB in 64-byte blocks
    clr  r2              / Start at physical 0
    mov  $6,r3           / 6 segments
1:
    mov  r2,(r0)+        / Set segment base address
    mov  $77406,(r1)+    / 4KB read-write
    add  r4,r2           / Next 8KB block
    sob  r3,1b           / Loop 6 times
```

This creates identity mapping for the first 48KB: virtual address X maps to physical address X. Segment descriptors `077406` means:

- Length = 127 blocks (8KB)
- Access = read-write

4.4.3. Initialize User Segment (Segment 6)

```
/ initialize user segment
    mov  $_end+63.,r2    / End of kernel + round up
    ash  $-6,r2          / Convert to 64-byte blocks
    bic  $!1777,r2       / Mask to valid range
    mov  r2,(r0)+        / ksr6 = address of u.
    mov  $usize-1\<8|6,(r1)+   / 16 blocks, read-write
```

Segment 6 holds the **user structure** (`u.`)—the per-process kernel data. It's placed just after the kernel's BSS.

4.4.4. Initialize I/O Segment (Segment 7)

```
/ initialize io segment
    mov  $7600,(r0)+     / ksr7 = 0760000 (I/O page)
    mov  $77406,(r1)+    / 4KB read-write
```

Segment 7 maps the I/O page where device registers live.

4.4.5. Enable Memory Management

```
/ get a sp and start segmentation
    mov  $_u+[usize*64.],sp    / Stack at top of u.
    inc  SSR0                   / Enable MMU!
```

The stack pointer is set to the top of the user structure, then `inc SSR0` turns on the MMU. From this point, all memory accesses go through address translation.

4.4.6. Clear BSS and User Area

```
/ clear bss
    mov  $_edata,r0
1:
    clr  (r0)+
    cmp  r0,$_end
    blo  1b

/ clear user block
    mov  $_u,r0
```

```
1:
    clr  (r0)+
    cmp  r0,$_u+[usize*64.]
    blo  1b
```

The BSS (uninitialized data) and user structure are zeroed. This is essential—C assumes uninitialized globals are zero.

4.4.7. Enter C Code

```
/ set up previous mode and call main
    mov  $30000,PS       / Previous mode = user
    jsr  pc,_main        / Call main()

/ on return, enter user mode at 0
    mov  $170000,-(sp)   / PS: user mode, IPL 0
    clr  -(sp)           / PC: address 0
    rti                  / "Return" to user mode
```

The previous mode is set to user (for later `mfpi`/`mtpi` instructions), then `main()` is called. When `main()` returns (in the child process), `rti` "returns" to user mode at address 0, executing the init code.

4.5. The main() Function

Now we enter C code. Let's walk through `main()` section by section.

4.5.1. Header and Data

```
#include "../param.h"
#include "../user.h"
#include "../systm.h"
#include "../proc.h"
#include "../text.h"
#include "../inode.h"
#include "../seg.h"

int lksp[]
{
    0177546,    /* KW11-L clock */
    0172540,    /* KW11-P clock */
    0           /* End marker */
};
```

`lksp` is a list of possible clock device addresses. UNIX probes each to find which clock is present.

4.5.2. The icode Array

```
int icode[]
{
    0104413,    /* sys exec */
    0000014,    /* address of "/etc/init" */
    0000010,    /* address of argv */
    0000777,    /* (unused) */
    0000014,    /* argv[0] = "/etc/init" */
    0000000,    /* argv[1] = NULL */
    0062457,    /* "/et" */
    0061564,    /* "c/" */
    0064457,    /* "in" */
    0064556,    /* "it" */
    0000164,    /* "\0" (null terminator) */
};
```

This is machine code! It's the first program that process 1 executes:

```
    sys exec            / System call: exec
    "/etc/init"         / Path argument
    argv                / Argument vector
```

Disassembled:

- `0104413` = `sys` instruction (exec is syscall 11, octal 013)
- The rest are the arguments: path string and argv pointers

This is how UNIX bootstraps user space—by hardcoding the first `exec()` call in machine language.

4.5.3. Memory Discovery

```
main()
{
    extern schar;
    register i1, *p;

    /*
     * zero and free all of core
     */
    updlock = 0;
    UISA->r[0] = KISA->r[6] + USIZE;
    UISD->r[0] = 077406;
    for(; fubyte(0) >= 0; UISA->r[0]++) {
        clearseg(UISA->r[0]);
        maxmem++;
        mfree(coremap, 1, UISA->r[0]);
    }
```

```
    printf("mem = %l\n", maxmem*10/32);
    maxmem = min(maxmem, MAXMEM);
    mfree(swapmap, nswap, swplo);
```

This discovers how much RAM the system has:

1. **Set up a probe segment** — UISA[0] points past the kernel, UISD[0] allows access
2. **Loop probing memory** — `fubyte(0)` tries to read address 0 of the current segment
3. **If successful** — Memory exists; clear it and add to free list
4. **If fails** — We've hit non-existent memory; stop

The `mfree()` calls add each 64-byte block to `coremap` (free memory list).

After the loop, `maxmem` contains the total memory in 64-byte blocks. The conversion `maxmem*10/32` prints kilobytes (64 bytes × 10/32 = 20 bytes… actually this prints in some odd unit).

Finally, swap space is added to `swapmap`.

4.5.4. Clock Detection

```
    /*
     * determine clock
     */
    UISA->r[7] = KISA->r[7];
    UISD->r[7] = 077406;
    for(p=lksp;; p++) {
        if(*p == 0)
            panic("no clock");
        if(fuword(*p) != -1) {
            lks = *p;
            break;
        }
    }
```

UNIX needs a clock for timekeeping and scheduling. This code:

1. Maps segment 7 to the I/O page
2. Probes each possible clock address
3. If `fuword()` succeeds (returns != -1), the clock exists
4. Saves the clock address in `lks`
5. If no clock found, `panic("no clock")` halts the system

4.5.5. Process 0 Setup

```
    /*
     * set up system process
     */
    proc[0].p_addr = KISA->r[6];
    proc[0].p_size = USIZE;
    proc[0].p_stat = SRUN;
    proc[0].p_flag =| SLOAD|SSYS;
    u.u_procp = &proc[0];
```

Process 0 is the **swapper** (scheduler). It's special:

- `p_addr` — Points to the user structure (segment 6)
- `p_size` — USIZE (16) blocks = 1024 bytes
- `p_stat` — SRUN (runnable)
- `p_flag` — SLOAD (in memory) | SSYS (system process)

The user structure pointer `u.u_procp` is set to point back to proc[0].

4.5.6. Subsystem Initialization

```
    /*
     * set up 'known' i-nodes
     */
    sureg();
    *lks = 0115;
    cinit();
    binit();
    iinit();
    rootdir = iget(rootdev, ROOTINO);
    rootdir->i_flag =& ~ILOCK;
    u.u_cdir = iget(rootdev, ROOTINO);
    u.u_cdir->i_flag =& ~ILOCK;
```

(continued on next page)

Now the kernel initializes its subsystems:

Call	Purpose
`sureg()`	Set up segment registers from u.u_uisa/u.u_uisd
`*lks = 0115`	Start the clock (magic value enables interrupts)
`cinit()`	Initialize character buffer freelists
`binit()`	Initialize buffer cache
`iinit()`	Read superblock, initialize inode table
`iget(rootdev, ROOTINO)`	Get root directory inode

The root directory inode is retrieved twice:

- `rootdir` — Global pointer used by `namei()`
- `u.u_cdir` — Process 0's current directory

Both have `ILOCK` cleared so they can be used immediately.

4.5.7. Creating Process 1 (init)

```
    /*
     * make init process
     * enter scheduling loop
     * with system process
     */
    if(newproc()) {
        expand(USIZE+1);
        u.u_uisa[0] = USIZE;
        u.u_uisd[0] = 6;
        sureg();
        copyout(icode, 0, 30);
        return;
    }
    sched();
}
```

This is the magic moment—creating the first user process:

In the parent (process 0): - `newproc()` creates process 1 and returns 0 - Falls through to `sched()`, entering the scheduler loop forever

In the child (process 1): - `newproc()` returns 1 (non-zero) - `expand(USIZE+1)` — Grow to 17 blocks (1 block for user code) - Set up segment 0 to map user memory - `copyout(icode, 0, 30)` — Copy the init code to user address 0 - `return` — Returns from main(), hitting the `rti` in mch.s

The `rti` at the end of `start:` pops a user-mode PS and PC=0, causing process 1 to start executing `icode` at address 0. This code does `exec("/etc/init", ...)`, replacing itself with the init program.

4.6. The sureg() Function

```
sureg()
{
    register *up, *rp, a;

    a = u.u_procp->p_addr;
    up = &u.u_uisa[0];
    rp = &UISA->r[0];
    while(rp < &UISA->r[8])
        *rp++ = *up++ + a;
```

`sureg()` copies the segment register values from the user structure to the actual hardware registers, adjusting by the process's physical base address.

The user structure stores *relative* segment addresses (relative to the process's memory). `sureg()` converts these to *absolute* physical addresses by adding `p_addr`.

4.7. The estabur() Function

```
estabur(nt, nd, ns)    /* text, data, stack sizes (in 64-byte blocks) */
{
    register a, *ap, *dp;

    /* Check if it fits */
    if(nseg(nt)+nseg(nd)+nseg(ns) > 8 || nt+nd+ns+USIZE > maxmem) {
        u.u_error = ENOMEM;
        return(-1);
    }
```

`estabur()` (establish user registers) sets up the memory map for a process. It takes three sizes in 64-byte blocks:

- `nt` — Text (code) size
- nd — Data size
- `ns` — Stack size

First it checks:

1. Total segments needed ≤ 8
2. Total memory needed ≤ available

```
    /* Set up text segments (read-only) */
    a = 0;
    ap = &u.u_uisa[0];
    dp = &u.u_uisd[0];
    while(nt >= 128) {          /* Full 8KB segments */
        *dp++ = (127<<8) | RO;  /* Max length, read-only */
        *ap++ = a;
        a =+ 128;
        nt =- 128;
    }
    if(nt) {                    /* Partial segment */
        *dp++ = ((nt-1)<<8) | RO;
        *ap++ = a;
        a =+ nt;
    }
```

Text segments are read-only (RO). Each full segment is 128 blocks (8KB).

```
    /* Set up data segments (read-write) */
    a = USIZE;                  /* Data starts after user struct */
    while(nd >= 128) {
        *dp++ = (127<<8) | RW;
        *ap++ = a;
        a =+ 128;
        nd =- 128;
    }
    if(nd) {
        *dp++ = ((nd-1)<<8) | RW;
        *ap++ = a;
        a =+ nd;
    }
```

Data segments are read-write (RW), starting at offset USIZE (after the user structure).

```
    /* Clear unused middle segments */
    while(ap < &u.u_uisa[8]) {
        *dp++ = 0;
        *ap++ = 0;
    }

    /* Set up stack (grows downward from top) */
    a =+ ns;
    while(ns >= 128) {
        a =- 128;
        ns =- 128;
        *--dp = (127<<8) | RW;
        *--ap = a;
    }
    if(ns) {
        *--dp = ((128-ns)<<8) | RW | ED;  /* ED = expand down */
```

```
        *--ap = a-128;
    }
    sureg();
    return(0);
}
```

The stack is set up from the top of the address space, growing downward. The ED (expand down) bit tells the MMU that valid addresses are at the *top* of the segment.

4.8. Key Data Structures

4.8.1. Global Variables (systm.h)

```
int  coremap[CMAPSIZ];   /* Free memory map */
int  swapmap[SMAPSIZ];   /* Free swap map */
int  *rootdir;           /* Root directory inode */
int  time[2];            /* System time */
int  maxmem;             /* Max memory available */
int  *lks;               /* Clock address */
int  rootdev;            /* Root device number */
int  swapdev;            /* Swap device number */
int  swplo;              /* Swap starting block */
int  nswap;              /* Swap size */
char runrun;             /* Reschedule flag */
```

4.8.2. Process Table Entry (proc.h)

```
struct proc {
    char p_stat;     /* Process state */
    char p_flag;     /* Flags */
    char p_pri;      /* Priority */
    char p_sig;      /* Pending signal */
    char p_time;     /* Time in memory/swap */
    int  p_ttyp;     /* Controlling terminal */
    int  p_pid;      /* Process ID */
    int  p_ppid;     /* Parent process ID */
    int  p_addr;     /* Address of user struct */
    int  p_size;     /* Size in blocks */
    int  p_wchan;    /* Wait channel */
    int  *p_textp;   /* Text segment pointer */
};
```

4.9. Boot Timeline

```
t=0 [Power On]
        Bootstrap loads kernel from disk

t=1 [Kernel Entry]
        start: executes
        - MMU initialized
        - Segments set up
        - BSS cleared
        - main() called

t=2 [Kernel Init]
        main() runs
        - Memory discovered
        - Clock started
        - Process 0 created
        - cinit(), binit(), iinit()
        - Root filesystem mounted
        - Process 1 forked

t=3 [Fork]
        Process 0: enters sched()
        Process 1: returns from main()
                   rti to user mode
                   executes icode
                   exec("/etc/init")

t=4 [User Space]
        /etc/init runs
        - Opens console
        - Spawns getty on terminals
        - System ready for login
```

4.10. Summary

- The bootstrap loads the kernel and jumps to `start:`
- `start:` in mch.s initializes the MMU and calls `main()`
- `main()` discovers memory by probing with `fubyte()`
- Process 0 (swapper) is created by filling in `proc[0]`
- Subsystems are initialized: buffers, inodes, root filesystem
- Process 1 is forked and runs `icode`, which execs `/etc/init`
- Process 0 enters `sched()` and never returns
- Process 1 becomes `/etc/init`, the ancestor of all user processes

4.11. Experiments

1. **Trace memory discovery**: Add a printf inside the memory probe loop to see each block being found.
2. **Decode icode**: Disassemble the `icode` array by hand. Verify it does `exec("/etc/init", argv)`.
3. **Boot without clock**: What happens if you remove clock detection? (Hint: `panic`)

4.12. Further Reading

- Chapter 5: Process Management — How `newproc()` works
- Chapter 8: Scheduling — The `sched()` function
- Chapter 9: Inodes and Superblock — What `iinit()` does

Next: Chapter 5 — Process Management

5. Chapter 5: Process Management

5.1. Overview

Processes are the heart of UNIX. Every running program is a process, and every process except the first is created by another process through `fork()`. This chapter examines how UNIX v4 represents, creates, and manages processes—the fundamental abstraction that makes multitasking possible.

5.2. Source Files

File	Purpose
`usr/sys/proc.h`	Process structure definition
`usr/sys/user.h`	User structure (per-process kernel data)
`usr/sys/ken/slp.c`	`newproc()`, `expand()`
`usr/sys/ken/sys1.c`	`fork()`, `exec()`, `exit()`, `wait()`

5.3. Prerequisites

- Chapter 2: PDP-11 Architecture (memory segments)
- Chapter 4: Boot Sequence (process 0 and 1 creation)

5.4. The Process Table

Every process has an entry in the global process table:

```
/* proc.h */
struct proc {
    char p_stat;      /* Process state */
    char p_flag;      /* Flags */
    char p_pri;       /* Priority (lower = higher priority) */
```

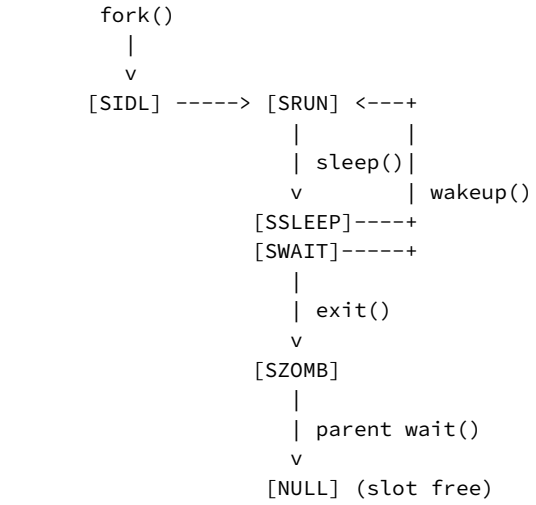
```
    char p_sig;       /* Pending signal */
    char p_null;      /* Unused */
    char p_time;      /* Resident time for scheduling */
    int  p_ttyp;      /* Controlling terminal */
    int  p_pid;       /* Process ID */
    int  p_ppid;      /* Parent process ID */
    int  p_addr;      /* Address of swappable image */
    int  p_size;      /* Size of swappable image (64-byte blocks) */
    int  p_wchan;     /* Wait channel (sleeping on) */
    int  *p_textp;    /* Pointer to text structure */
} proc[NPROC];
```

With NPROC=50, the system supports at most 50 simultaneous processes.

5.4.1. Process States (p_stat)

```
#define SSLEEP  1    /* Sleeping at high priority */
#define SWAIT   2    /* Sleeping at low priority (interruptible) */
#define SRUN    3    /* Runnable */
#define SIDL    4    /* Being created */
#define SZOMB   5    /* Terminated, waiting for parent */
```

State transitions:



5.4.2. Process Flags (p_flag)

```
#define SLOAD   01   /* In memory (not swapped) */
#define SSYS    02   /* System process (process 0) */
#define SLOCK   04   /* Process cannot be swapped */
#define SSWAP   010  /* Being swapped out */
```

5.5. The User Structure

While `proc` holds minimal info for all processes, `u` (the **user structure**) holds extensive per-process data for the *current* process:

```
/* user.h */
struct user {
    int   u_rsav[2];        /* Saved r5, r6 for resume */
    int   u_fsav[25];       /* Floating point save area */
    char  u_segflg;         /* I/O to user/kernel space */
    char  u_error;          /* Error code from syscall */
    char  u_uid;            /* Effective user ID */
    char  u_gid;            /* Effective group ID */
    char  u_ruid;           /* Real user ID */
    char  u_rgid;           /* Real group ID */
    int   u_procp;          /* Pointer to proc entry */
    char  *u_base;          /* I/O base address */
    char  *u_count;         /* I/O byte count */
    char  *u_offset[2];     /* I/O file offset */
    int   *u_cdir;          /* Current directory inode */
    char  u_dbuf[DIRSIZ];   /* Current pathname component */
    char  *u_dirp;          /* Pathname pointer */
    struct {                /* Current directory entry */
        int  u_ino;
        char u_name[DIRSIZ];
    } u_dent;
    int   *u_pdir;          /* Parent directory inode */
    int   u_uisa[8];        /* User segment addresses */
    int   u_uisd[8];        /* User segment descriptors */
    int   u_ofile[NOFILE];  /* Open file table */
    int   u_arg[5];         /* Syscall arguments */
    int   u_tsize;          /* Text size (64-byte blocks) */
    int   u_dsize;          /* Data size */
    int   u_ssize;          /* Stack size */
    int   u_qsav[2];        /* Saved regs for signal return */
    int   u_ssav[2];        /* Saved regs for swap return */
    int   u_signal[NSIG];   /* Signal handlers */
    int   u_utime;          /* User time (ticks) */
    int   u_stime;          /* System time (ticks) */
    int   u_cutime[2];      /* Children's user time */
    int   u_cstime[2];      /* Children's system time */
    int   *u_ar0;           /* Pointer to saved r0 */
    int   u_prof[4];        /* Profiling parameters */
    char  u_nice;           /* Nice value */
    char  u_dsleep;         /* Deep sleep flag */
} u;    /* u = 140000 */
```

The magic comment `u = 140000` means the user structure is always at virtual address `0140000` (octal). This is segment 6, which the kernel remaps for each process.

5.6. fork() — Creating a Process

The `fork()` system call creates a new process:

```
/* sys1.c */
fork()
{
    register struct proc *p1, *p2;

    p1 = u.u_procp;            /* Parent */
    for(p2 = &proc[0]; p2 < &proc[NPROC]; p2++)
        if(p2->p_stat == NULL)
            goto found;
    u.u_error = EAGAIN;        /* No free slots */
    goto out;

found:
    if(newproc()) {
        /* Child: return parent's PID */
        u.u_ar0[R0] = p1->p_pid;
        u.u_cstime[0] = 0;
        u.u_cstime[1] = 0;
        u.u_stime = 0;
        u.u_cutime[0] = 0;
        u.u_cutime[1] = 0;
        u.u_utime = 0;
        return;
    }
    /* Parent: return child's PID */
    u.u_ar0[R0] = p2->p_pid;

out:
    u.u_ar0[R7] =+ 2;          /* Skip over sys fork instruction */
}
```

The key insight: `fork()` returns **twice**—once in the parent (returning child's PID) and once in the child (returning parent's PID). The actual work is in `newproc()`.

5.6.1. newproc() — The Fork Implementation

```
/* slp.c */
newproc()
{
    int a1, a2;
    struct proc *p, *up;
    register struct proc *rpp;
    register *rip, n;

    /* Find free proc slot */
    for(rpp = &proc[0]; rpp < &proc[NPROC]; rpp++)
        if(rpp->p_stat == NULL)
```

```
            goto found;
    panic("no procs");

found:
    /*
     * make proc entry for new proc
     */
    p = rpp;
    rip = u.u_procp;
    up = rip;
    rpp->p_stat = SRUN;
    rpp->p_flag = SLOAD;
    rpp->p_ttyp = rip->p_ttyp;      /* Inherit terminal */
    rpp->p_textp = rip->p_textp;    /* Share text segment */
    rpp->p_pid = ++mpid;            /* Assign new PID */
    rpp->p_ppid = rip->p_pid;       /* Record parent */
    rpp->p_time = 0;
```

The child inherits most fields from the parent, but gets a new PID.

```
    /*
     * make duplicate entries
     * where needed
     */
    for(rip = &u.u_ofile[0]; rip < &u.u_ofile[NOFILE];)
        if((rpp = *rip++) != NULL)
            rpp->f_count++;         /* Bump file ref counts */
    if((rpp=up->p_textp) != NULL) {
        rpp->x_count++;             /* Bump text ref count */
        rpp->x_ccount++;
    }
    u.u_cdir->i_count++;            /* Bump cwd inode ref */
```

Shared resources (open files, text segment, current directory) have their reference counts incremented.

```
    /*
     * swap out old process
     * to make image of new proc
     */
    savu(u.u_rsav);
    rpp = p;
    u.u_procp = rpp;
    rip = up;
    n = rip->p_size;
    a1 = rip->p_addr;
    rpp->p_size = n;
    a2 = malloc(coremap, n);
    if(a2 == NULL) {
        /* No memory: swap out child */
        rip->p_stat = SIDL;
        rpp->p_addr = a1;
        savu(u.u_ssav);
```

```
        xswap(rpp, 0, 0);
        rpp->p_flag =| SSWAP;
        rip->p_stat = SRUN;
    } else {
        /* Copy parent's memory to child */
        rpp->p_addr = a2;
        while(n--)
            copyseg(a1++, a2++);
    }
    u.u_procp = rip;
    return(0);                      /* Return 0 in parent */
}
```

If memory is available, the parent's image is copied block-by-block. If not, the child is created on swap. Either way, the parent returns 0.

The child's return happens later, when the scheduler runs the child and it resumes from the saved context.

5.7. exec() — Running a Program

`exec()` replaces the current process's memory with a new program:

```
/* sys1.c */
exec()
{
    int ap, na, nc, *bp;
    int ts, ds;
    register c, *ip;
    register char *cp;
    extern uchar;

    /*
     * pick up file names
     * and check various modes
     */
    ip = namei(&uchar, 0);      /* Look up pathname */
    if(ip == NULL)
        return;
    bp = getblk(NODEV);         /* Get buffer for args */
    if(access(ip, IEXEC))       /* Check execute permission */
        goto bad;
```

First, the executable file is located and checked for execute permission.

```
    /*
     * pack up arguments into
     * allocated disk buffer
     */
```

```
    cp = bp->b_addr;
    na = 0;    /* Argument count */
    nc = 0;    /* Character count */
    while(ap = fuword(u.u_arg[1])) {
        na++;
        if(ap == -1)
            goto bad;
        u.u_arg[1] =+ 2;
        for(;;) {
            c = fubyte(ap++);
            if(c == -1)
                goto bad;
            *cp++ = c;
            nc++;
            if(nc > 510) {
                u.u_error = E2BIG;
                goto bad;
            }
            if(c == 0)
                break;
        }
    }
```

Arguments are copied from user space into a kernel buffer. There's a 510-byte limit on total argument length.

```
    /*
     * read in first 8 bytes
     * of file for segment sizes:
     * w0 = 407/410 (410 implies RO text)
     * w1 = text size
     * w2 = data size
     * w3 = bss size
     */
    u.u_base = &u.u_arg[0];
    u.u_count = 8;
    u.u_offset[1] = 0;
    u.u_offset[0] = 0;
    u.u_segflg = 1;
    readi(ip);
```

The a.out header is read:

- `0407` — Executable with combined text+data (not shared)
- `0410` — Executable with separate, read-only text (sharable)

```
    /*
     * find text and data sizes
     */
    ts = ((u.u_arg[1]+63)>>6) & 01777;
    ds = ((u.u_arg[2]+u.u_arg[3]+63)>>6) & 01777;
    if(estabur(ts, ds, SSIZE))
```

```
        goto bad;

    /*
     * allocate and clear core
     * at this point, committed to the new image
     */
    u.u_prof[3] = 0;
    xfree();                    /* Free old text segment */
    xalloc(ip);                 /* Allocate new text */
    c = USIZE+ds+SSIZE;
    expand(USIZE);
    expand(c);
    while(--c >= USIZE)
        clearseg(u.u_procp->p_addr+c);
```

Old memory is freed, new memory is allocated and cleared.

```
    /*
     * read in data segment
     */
    estabur(0, ds, 0);
    u.u_base = 0;
    u.u_offset[1] = 020+u.u_arg[1];   /* Skip header + text */
    u.u_count = u.u_arg[2];
    readi(ip);

    /*
     * initialize stack segment
     */
    u.u_tsize = ts;
    u.u_dsize = ds;
    u.u_ssize = SSIZE;
    estabur(u.u_tsize, u.u_dsize, u.u_ssize);
```

The data segment is read from the file. For `0410` executables, the text segment is handled separately through shared text management.

```
    /*
     * Copy arguments to user stack
     */
    cp = bp->b_addr;
    ap = -nc - na*2 - 4;        /* Stack grows down */
    u.u_ar0[R6] = ap;           /* Set stack pointer */
    suword(ap, na);             /* argc */
    c = -nc;
    while(na--) {
        suword(ap=+2, c);       /* argv[i] */
        do
            subyte(c++, *cp);   /* Copy string */
        while(*cp++);
    }
    suword(ap+2, -1);           /* argv terminator */
```

Arguments are copied to the user stack in the standard format:

```
sp → argc
     argv[0] → "program"
     argv[1] → "arg1"
     ...
     NULL
     "program\0arg1\0..."
```

```
    /*
     * set SUID/SGID protections
     */
    if(ip->i_mode&ISUID)
        if(u.u_uid != 0)
            u.u_uid = ip->i_uid;
    if(ip->i_mode&ISGID)
        u.u_gid = ip->i_gid;

    /*
     * clear sigs, regs and return
     */
    for(ip = &u.u_signal[0]; ip < &u.u_signal[NSIG]; ip++)
        if((*ip & 1) == 0)
            *ip = 0;            /* Reset non-ignored signals */
    for(cp = &regloc[0]; cp < &regloc[6];)
        u.u_ar0[*cp++] = 0;     /* Clear registers */
    u.u_ar0[R7] = 0;            /* PC = 0 (entry point) */
```

Setuid/setgid is handled, signals are reset, and execution begins at address 0.

5.8. exit() — Terminating a Process

```
/* sys1.c */
exit()
{
    register int *q, a;
    register struct proc *p;

    /* Ignore all signals */
    for(q = &u.u_signal[0]; q < &u.u_signal[NSIG];)
        *q++ = 1;

    /* Close all open files */
    for(q = &u.u_ofile[0]; q < &u.u_ofile[NOFILE]; q++)
        if(a = *q) {
            *q = NULL;
            closef(a);
        }

    /* Release current directory */
    iput(u.u_cdir);
```

```
    /* Free text segment */
    xfree();
```

First, cleanup: ignore signals, close files, release directory.

```
    /* Save exit status to swap */
    a = malloc(swapmap, 8);
    p = getblk(swapdev, a);
    bcopy(&u, p->b_addr, 256);
    bwrite(p);

    /* Free memory */
    q = u.u_procp;
    mfree(coremap, q->p_size, q->p_addr);
    q->p_addr = a;              /* Now points to swap */
    q->p_stat = SZOMB;          /* Zombie state */
```

The user structure (containing the exit status) is saved to swap, memory is freed, and the process becomes a zombie.

```
loop:
    /* Find parent and wake it up */
    for(p = &proc[0]; p < &proc[NPROC]; p++)
    if(q->p_ppid == p->p_pid) {
        wakeup(&proc[1]);       /* Wake init (adopts orphans) */
        wakeup(p);              /* Wake parent */

        /* Orphan our children to init */
        for(p = &proc[0]; p < &proc[NPROC]; p++)
        if(q->p_pid == p->p_ppid)
            p->p_ppid = 1;

        swtch();                /* Switch away, never return */
    }
```

The parent is woken up, and orphaned children are adopted by init (PID 1). The process then switches away and never returns—it remains a zombie until the parent calls `wait()`.

5.9. wait() — Reaping Children

```
/* sys1.c */
wait()
{
    register f, *bp;
    register struct proc *p;
```

```
    f = 0;

loop:
    for(p = &proc[0]; p < &proc[NPROC]; p++)
    if(p->p_ppid == u.u_procp->p_pid) {
        f++;
        if(p->p_stat == SZOMB) {
            /* Found dead child */
            u.u_ar0[R0] = p->p_pid;

            /* Read exit status from swap */
            bp = bread(swapdev, f=p->p_addr);
            mfree(swapmap, 8, f);

            /* Clear proc slot */
            p->p_stat = NULL;
            p->p_pid = 0;
            p->p_ppid = 0;
            ...

            /* Accumulate child's CPU time */
            u.u_cstime[0] =+ p->u_cstime[0];
            ...

            /* Return exit status */
            u.u_ar0[R1] = p->u_arg[0];
            brelse(bp);
            return;
        }
    }
    if(f) {
        sleep(u.u_procp, PWAIT);    /* Wait for child to exit */
        goto loop;
    }
    u.u_error = ECHILD;             /* No children */
}
```

`wait()` searches for zombie children. If found, it:

1. Reads the exit status from swap
2. Frees the swap space
3. Clears the proc slot
4. Accumulates CPU time statistics
5. Returns PID and exit status

If there are living children but no zombies, it sleeps until one exits.

5.10. sbreak() — Changing Memory Size

```
/* sys1.c */
sbreak()
{
    register a, n, d;

    /*
     * Calculate new data size
     */
    n = (((u.u_arg[0]+63)>>6) & 01777) - nseg(u.u_tsize)*128;
    if(n < 0)
        n = 0;
    d = n - u.u_dsize;          /* Delta */
    n =+ USIZE+u.u_ssize;

    if(estabur(u.u_tsize, u.u_dsize+d, u.u_ssize))
        return;
    u.u_dsize =+ d;

    if(d > 0)
        goto bigger;

    /* Shrinking: move stack down, then shrink */
    a = u.u_procp->p_addr + n - u.u_ssize;
    n = u.u_ssize;
    while(n--) {
        copyseg(a-d, a);
        a++;
    }
    expand(i);
    return;

bigger:
    /* Growing: expand, then move stack up */
    expand(n);
    a = u.u_procp->p_addr + n;
    n = u.u_ssize;
    while(n--) {
        a--;
        copyseg(a-d, a);
    }
    while(d--)
        clearseg(--a);
}
```

`sbreak()` (break) changes the data segment size. The stack must be moved when the data segment grows or shrinks.

5.11. expand() — Growing or Shrinking Process Memory

```
/* slp.c */
expand(newsize)
{
    int i, n;
    register *p, a1, a2;

    p = u.u_procp;
    n = p->p_size;
    p->p_size = newsize;
    a1 = p->p_addr;

    if(n >= newsize) {
        /* Shrinking: just free excess */
        mfree(coremap, n-newsize, a1+newsize);
        return;
    }

    /* Growing: need to allocate new space */
    savu(u.u_rsav);
    a2 = malloc(coremap, newsize);
    if(a2 == NULL) {
        /* No memory: swap out and grow on swap */
        savu(u.u_ssav);
        xswap(p, 1, n);
        p->p_flag =| SSWAP;
        swtch();
        /* no return */
    }

    /* Copy to new location */
    p->p_addr = a2;
    for(i=0; i<n; i++)
        copyseg(a1+i, a2++);
    mfree(coremap, n, a1);
    retu(p->p_addr);
    sureg();
}
```

If growing and memory is available, the process is copied to a new, larger location. If not, it's swapped out.

5.12. Summary

- The **proc structure** holds minimal per-process info; **u** holds the rest
- `fork()` creates a new process by duplicating the parent
- `exec()` replaces a process's memory image with a new program
- `exit()` terminates a process, making it a zombie

- `wait()` reaps zombie children and retrieves their exit status
- `sbreak()` changes the data segment size
- `expand()` handles memory allocation/reallocation for processes

5.13. Key Concepts

5.13.1. Process Creation Pattern

```
if(fork() == 0) {
    /* Child */
    exec("/bin/program", ...);
    exit(1);    /* exec failed */
}
/* Parent continues */
wait(&status);
```

5.13.2. Reference Counting

When `fork()` copies a process, shared resources have their reference counts bumped:

- Open files (`f_count`)
- Text segments (`x_count`)
- Inodes (`i_count`)

When `exit()` cleans up, these counts are decremented.

5.13.3. The Zombie State

A zombie is a process that has exited but hasn't been waited for:

- Uses minimal resources (just a proc slot and swap block)
- Contains exit status for parent
- Cleaned up by parent's `wait()`

5.14. Experiments

1. **Count processes**: Add printfs to trace proc slot allocation in `fork()`.
2. **Argument limit**: Try to exec with more than 510 bytes of arguments.
3. **Fork bomb**: What happens if a process forks in a loop? (The NPROC limit saves you.)

5.15. Further Reading

- Chapter 6: Memory Management — How memory is allocated
- Chapter 8: Scheduling — How processes are selected to run
- Chapter 7: Traps and System Calls — How fork/exec/exit are invoked

---

Next: Chapter 6 — Memory Management

6. Chapter 6: Memory Management

6.1. Overview

UNIX v4 manages memory with elegant simplicity. There's no virtual memory in the modern sense—no page tables, no demand paging, no memory-mapped files. Instead, processes exist entirely in physical memory or entirely on swap. This chapter examines how UNIX allocates, tracks, and swaps memory.

6.2. Source Files

File	Purpose
`usr/sys/dmr/malloc.c`	`malloc()`, `mfree()` — map allocator
`usr/sys/ken/main.c`	Memory discovery, `estabur()`, `sureg()`
`usr/sys/ken/slp.c`	`expand()`, `sched()`, `xswap()`
`usr/sys/systm.h`	`coremap[]`, `swapmap[]` definitions
`usr/sys/param.h`	MAXMEM, CMAPSIZ, SMAPSIZ

6.3. Prerequisites

- Chapter 2: PDP-11 Architecture (MMU, segments)
- Chapter 5: Process Management (process structure)

6.4. The Memory Model

UNIX v4 uses a simple model:

```
Physical Memory:
+------------------+ 0
|     Kernel       | (text, data, bss)
+------------------+
|   User Block     | (u. for current process)
+------------------+
|   Free Memory    | (managed by coremap)
|   Processes      |
|   Buffers        |
+------------------+ maxmem
|   (non-existent) |
+------------------+ 64KB limit

Swap Space:
+------------------+ swplo
|   Swapped procs  | (managed by swapmap)
|   ...            |
+------------------+ swplo+nswap
```

Key characteristics:

- **No paging** — Entire processes are swapped, not individual pages
- **No sharing** — Except for read-only text segments
- **Contiguous allocation** — Each process occupies a contiguous region
- **First-fit algorithm** — Simple but effective

6.5. The Map Allocator

The core of memory management is a general-purpose allocator used for both core memory (`coremap`) and swap space (`swapmap`):

```
/* malloc.c */
struct map {
    char *m_size;    /* Size of this free region */
    char *m_addr;    /* Starting address */
};
```

A map is an array of (size, address) pairs, sorted by address, terminated by a zero-size entry:

```
coremap:
+------+------+
| size | addr |  Free region 1
+------+------+
| size | addr |  Free region 2
+------+------+
|  0   |  -   |  End marker
+------+------+
```

6.5.1. malloc() — Allocate from Map

```
malloc(mp, size)
struct map *mp;
{
    register int a;
    register struct map *bp;

    for (bp = mp; bp->m_size; bp++) {
        if (bp->m_size >= size) {
            a = bp->m_addr;
            bp->m_addr =+ size;
            if ((bp->m_size =- size) == 0)
                /* Remove empty entry by shifting */
                do {
                    bp++;
                    (bp-1)->m_addr = bp->m_addr;
                } while ((bp-1)->m_size = bp->m_size);
            return(a);
        }
    }
    return(0);    /* No space */
}
```

Algorithm (**first-fit**):

1. Scan the map for a region ≥ requested size
2. If found, allocate from the *start* of the region
3. Shrink the region (or remove if empty)
4. Return the starting address (or 0 if no space)

Example — allocating 3 blocks:

```
Before:              After:
Size | Addr          Size | Addr
-----+-----          -----+-----
  5  | 100             2  | 103
  3  | 200    ->       3  | 200
  0  |                 0  |

Returns: 100 (allocated blocks 100–102)
```

6.5.2. mfree() — Free to Map

```
mfree(mp, size, aa)
struct map *mp;
{
    register struct map *bp;
    register int t;
```

```
    register int a;

    a = aa;
    for (bp = mp; bp->m_addr<=a && bp->m_size!=0; bp++);
```

Find where this block fits in the sorted list.

```
    if (bp>mp && (bp-1)->m_addr+(bp-1)->m_size == a) {
        /* Coalesce with previous region */
        (bp-1)->m_size =+ size;
        if (a+size == bp->m_addr) {
            /* Also coalesce with next region */
            (bp-1)->m_size =+ bp->m_size;
            while (bp->m_size) {
                bp++;
                (bp-1)->m_addr = bp->m_addr;
                (bp-1)->m_size = bp->m_size;
            }
        }
    }
```

Try to merge with adjacent free regions.

```
    } else {
        if (a+size == bp->m_addr && bp->m_size) {
            /* Coalesce with next region */
            bp->m_addr =- size;
            bp->m_size =+ size;
        } else if (size) do {
            /* Insert new entry (shift others down) */
            t = bp->m_addr;
            bp->m_addr = a;
            a = t;
            t = bp->m_size;
            bp->m_size = size;
            bp++;
        } while (size = t);
    }
}
```

If can’t merge, insert a new entry (shifting subsequent entries).

Example — freeing 2 blocks at address 105:

```
Before:             After:
Size | Addr         Size | Addr
-----+-----         -----+-----
  2  | 103            4  | 103    (merged: 103-104 + 105-106 = 103-106)
  3  | 200    ->      3  | 200
  0  |               0  |
```

6.5.3. Coalescing

The `mfree()` function handles four cases:

1. **Merge with previous**: freed block is adjacent to end of previous
2. **Merge with next**: freed block is adjacent to start of next
3. **Merge with both**: freed block bridges two regions
4. **No merge**: create new entry

This prevents fragmentation by keeping free regions as large as possible.

6.6. Memory Discovery

At boot, `main()` discovers available memory:

```
/* main.c */
main()
{
    updlock = 0;
    UISA->r[0] = KISA->r[6] + USIZE;
    UISD->r[0] = 077406;
    for(; fubyte(0) >= 0; UISA->r[0]++) {
        clearseg(UISA->r[0]);
        maxmem++;
        mfree(coremap, 1, UISA->r[0]);
    }
    printf("mem = %l\n", maxmem*10/32);
    maxmem = min(maxmem, MAXMEM);
    mfree(swapmap, nswap, swplo);
```

How it works:

1. Map user segment 0 to physical memory after the kernel
2. Try to read byte 0 of that segment with `fubyte()`
3. If successful, memory exists—clear it and add to `coremap`
4. Advance to next 64-byte block and repeat
5. When `fubyte()` fails (returns -1), we've hit non-existent memory

After the loop, `coremap` contains one large free region starting just after the kernel.

6.7. Process Memory Layout

Each process has:

```
Process Memory:
+------------------+ p_addr
|   User Block     | USIZE blocks (1KB)
|   (u.)           |
+------------------+ p_addr + USIZE
|   Data Segment   | u_dsize blocks
|   (+ BSS)        |
+------------------+
|   (free space)   |
+------------------+
|   Stack Segment  | u_ssize blocks
+------------------+ p_addr + p_size

Text Segment:      (if shared, stored separately)
+------------------+ x_caddr
|   Code           | x_size blocks
+------------------+
```

The `p_size` field in `struct proc` is the total size in 64-byte blocks.

6.8. Segment Register Management

6.8.1. estabur() — Establish User Registers

```
/* main.c */
estabur(nt, nd, ns)    /* text, data, stack sizes */
{
    register a, *ap, *dp;

    /* Check limits */
    if(nseg(nt)+nseg(nd)+nseg(ns) > 8 || nt+nd+ns+USIZE > maxmem) {
        u.u_error = ENOMEM;
        return(-1);
    }
```

First, verify the request is feasible:

- No more than 8 segments total
- Total memory ≤ available

```
    /* Text segments (read-only) */
    a = 0;
    ap = &u.u_uisa[0];
    dp = &u.u_uisd[0];
    while(nt >= 128) {
```

```
        *dp++ = (127<<8) | RO;
        *ap++ = a;
        a =+ 128;
        nt =- 128;
    }
    if(nt) {
        *dp++ = ((nt-1)<<8) | RO;
        *ap++ = a;
    }
```

Text is read-only, starting at relative address 0.

```
    /* Data segments (read-write) */
    a = USIZE;    /* After user block */
    while(nd >= 128) {
        *dp++ = (127<<8) | RW;
        *ap++ = a;
        a =+ 128;
        nd =- 128;
    }
    if(nd) {
        *dp++ = ((nd-1)<<8) | RW;
        *ap++ = a;
    }
```

Data is read-write, starting after the user block.

```
    /* Clear unused segments */
    while(ap < &u.u_uisa[8]) {
        *dp++ = 0;
        *ap++ = 0;
    }

    /* Stack segment (expand down) */
    a =+ ns;
    while(ns >= 128) {
        a =- 128;
        ns =- 128;
        *--dp = (127<<8) | RW;
        *--ap = a;
    }
    if(ns) {
        *--dp = ((128-ns)<<8) | RW | ED;
        *--ap = a-128;
    }
    sureg();
    return(0);
}
```

Stack is at the end, with the ED (expand down) bit.

6.8.2. sureg() — Set User Registers

```
/* main.c */
sureg()
{
    register *up, *rp, a;

    a = u.u_procp->p_addr;
    up = &u.u_uisa[0];
    rp = &UISA->r[0];
    while(rp < &UISA->r[8])
        *rp++ = *up++ + a;
```

Copy user segment addresses to hardware, adding the process base address.

The user structure stores *relative* addresses; `sureg()` converts to *absolute* physical addresses.

6.9. Swapping

When memory is tight, processes are **swapped** to disk:

6.9.1. xswap() — Swap Out a Process

```
/* slp.c */
xswap(p, ff, os)
struct proc *p;
{
    register a;

    if(os == 0)
        os = p->p_size;
    a = malloc(swapmap, (p->p_size+7)/8);
    if(a == NULL)
        panic("out of swap");
    xccdec(p->p_textp);
    swap(a, p->p_addr, os, B_WRITE);
    if(ff)
        mfree(coremap, os, p->p_addr);
    p->p_addr = a;
    p->p_flag =| SSWAP;
    p->p_flag =& ~SLOAD;
}
```

1. Allocate swap space
2. Decrement text reference count
3. Write process image to swap

4. Free core memory (if `ff` flag set)
5. Update `p_addr` to point to swap location
6. Clear SLOAD, set SSWAP flags

6.9.2. Swap In

The scheduler (`sched()` in slp.c) swaps processes back in:

```
/* From sched() */
found2:
    if((rp=p1->p_textp) != NULL) {
        if(rp->x_ccount == 0) {
            /* Swap in text if needed */
            if(swap(rp->x_daddr, a, rp->x_size, B_READ))
                goto swaper;
            rp->x_caddr = a;
            a =+ rp->x_size;
        }
        rp->x_ccount++;
    }
    rp = p1;
    if(swap(rp->p_addr, a, rp->p_size, B_READ))
        goto swaper;
    mfree(swapmap, (rp->p_size+7)/8, rp->p_addr);
    rp->p_addr = a;
    rp->p_flag =| SLOAD;
```

6.10. expand() — Change Process Size

```
/* slp.c */
expand(newsize)
{
    int i, n;
    register *p, a1, a2;

    p = u.u_procp;
    n = p->p_size;
    p->p_size = newsize;
    a1 = p->p_addr;

    if(n >= newsize) {
        /* Shrinking */
        mfree(coremap, n-newsize, a1+newsize);
        return;
    }
```

If shrinking, just free the excess.

```
    /* Growing: try to allocate new space */
    savu(u.u_rsav);
    a2 = malloc(coremap, newsize);
    if(a2 == NULL) {
        /* No space: swap out, grow on swap */
        savu(u.u_ssav);
        xswap(p, 1, n);
        p->p_flag =| SSWAP;
        swtch();
        /* no return */
    }

    /* Copy to new location */
    p->p_addr = a2;
    for(i=0; i<n; i++)
        copyseg(a1+i, a2++);
    mfree(coremap, n, a1);
    retu(p->p_addr);
    sureg();
}
```

If growing:

1. Try to allocate larger region
2. If successful, copy process to new location
3. If not, swap out and let scheduler handle it

6.11. Shared Text Segments

Executable programs with magic number `0410` have separate, read-only text segments that can be shared:

```
/* text.h */
struct text {
    int  x_daddr;    /* Disk address of segment */
    int  x_caddr;    /* Core address (if in memory) */
    int  x_size;     /* Size in 64-byte blocks */
    int  *x_iptr;    /* Inode pointer */
    char x_count;    /* Reference count */
    char x_ccount;   /* In-core reference count */
} text[NTEXT];
```

When a process execs a shared-text program:

1. Look for existing text entry for this inode
2. If found, just increment reference count
3. If not, create new entry and load text from disk

When process exits:

1. Decrement reference counts
2. When `x_count` reaches 0, text can be freed

6.12. Memory Limits

From `param.h`:

```
#define MAXMEM   (32*32)   /* Max user memory: 1024 blocks = 64KB */
#define CMAPSIZ  100       /* Coremap entries */
#define SMAPSIZ  100       /* Swapmap entries */
```

The MAXMEM limit exists because:

- 64KB address space per process
- Kernel reserves some segments
- Practical limit on how much to allocate to one process

6.13. Summary

- Memory is managed with a simple first-fit allocator (`malloc`/`mfree`)
- Two maps: `coremap` for physical memory, `swapmap` for swap space
- Processes are allocated contiguous memory regions
- `estabur()` sets up segment registers based on text/data/stack sizes
- `sureg()` loads segment registers into hardware
- Swapping moves entire processes between memory and disk
- Shared text segments reduce memory usage for common programs

6.14. Key Insight: Simplicity

The UNIX v4 memory management is remarkably simple:

- No page tables
- No complex allocation algorithms
- No memory-mapped files
- Just contiguous regions, a free list, and swapping

This simplicity made UNIX portable and maintainable. The more complex virtual memory systems of later UNIX versions added capability but also complexity.

6.15. Experiments

1. **Trace allocation**: Add printfs to `malloc()`/`mfree()` to watch memory allocation patterns.
2. **Fragmentation**: What happens to `coremap` as processes come and go? Does fragmentation occur?
3. **Swap thrashing**: What happens if memory is very tight and processes keep getting swapped in and out?

6.16. Further Reading

- Chapter 5: Process Management — How `expand()` is used
- Chapter 8: Scheduling — How `sched()` decides what to swap
- Chapter 12: Buffer Cache — Another use of memory

Next: Chapter 7 — Traps and System Calls

7. Chapter 7: Traps and System Calls

7.1. Overview

System calls are how user programs request kernel services. This chapter traces the complete path of a system call—from the user's `sys` instruction through the trap handler to the kernel function and back. Understanding this mechanism is key to understanding the user/kernel interface.

7.2. Source Files

File	Purpose
`usr/sys/ken/trap.c`	Trap handler
`usr/sys/ken/sysent.c`	System call table
`usr/sys/conf/mch.s`	Assembly trap entry
`usr/sys/conf/low.s`	Interrupt vectors
`usr/sys/reg.h`	Register offsets

7.3. Prerequisites

- Chapter 2: PDP-11 Architecture (traps, PS word)
- Chapter 5: Process Management (process structure)

7.4. The Trap Mechanism

When a PDP-11 executes a `trap` instruction (or encounters an error), the hardware:

1. Pushes the current PS onto the kernel stack
2. Pushes the current PC onto the kernel stack
3. Loads new PS and PC from the trap vector

4. Execution continues at the new PC (in kernel mode)

7.4.1. Trap Vectors

From `low.s`:

```
. = 0^.
    br   1f
    4

/ trap vectors
    trap; br7+0.     / 4: bus error
    trap; br7+1.     / 10: illegal instruction
    trap; br7+2.     / 14: BPT (breakpoint)
    trap; br7+3.     / 20: IOT trap
    trap; br7+4.     / 24: power fail
    trap; br7+5.     / 30: EMT (emulator trap)
    trap; br7+6.     / 34: system call (TRAP instruction)

. = 240^.
    trap; br7+7.     / 240: programmed interrupt
    trap; br7+8.     / 244: floating point exception
    trap; br7+9.     / 250: segmentation violation
```

Each vector is two words:

- New PC: `trap` (the assembly routine)
- New PS: `br7+N` where N identifies the trap type

The `br7` (octal 340) sets IPL to 7 (block all interrupts) and kernel mode.

7.4.2. Trap Types

Vector	dev	Cause
4	0	Bus error (invalid address)
10	1	Illegal instruction
14	2	BPT (breakpoint trap)
20	3	IOT trap
24	4	Power fail
30	5	EMT (emulator trap)
34	6	**TRAP (system call)**

Vector	dev	Cause
240	7	Programmed interrupt
244	8	Floating point exception
250	9	Segmentation violation

7.5. Assembly Entry Point

From `mch.s`, the `trap` routine:

```
trap:
    mov  PS,-4(sp)        / Save PS in unused stack slot
    tst  nofault
    bne  1f               / If nofault set, handle specially
    mov  SSR0,ssr         / Save MMU status registers
    mov  SSR2,ssr+4
    mov  $1,SSR0          / Re-enable MMU
    jsr  r0,call1; _trap  / Call C trap handler
1:
    mov  $1,SSR0
    mov  nofault,(sp)
    rti
```

The key line is `jsr r0,call1; _trap` which calls the C `trap()` function.

7.5.1. The call1 Routine

```
call1:
    tst  -(sp)            / Make room on stack
    spl  0                / Enable interrupts
    br   1f               / Fall into call

call:
    mov  PS,-(sp)         / Save PS
1:
    mov  r1,-(sp)         / Save r1
    mfpi sp               / Get user SP
    mov  4(sp),-(sp)      / Push dev number
    ...
    jsr  pc,*(r0)+        / Call the C function
    ...
    rti                   / Return from interrupt
```

This saves registers and calls the C function with arguments set up properly.

7.6. The trap() Function

```
/* trap.c */
trap(dev, sp, r1, nps, r0, pc, ps)
char *sp;
{
    register i, a;

    savfp();              /* Save floating point state */
    u.u_ar0 = &r0;        /* Point to saved registers */
```

The parameters are the saved registers, with dev being the trap type (0-9).

7.6.1. Floating Point Exception (dev == 8)

```
    if(dev == 8) {
        psignal(u.u_procp, SIGFPT);
        if((ps&UMODE) == UMODE)
            goto err;
        return;
    }
```

Floating point errors signal the process.

7.6.2. SETD Instruction Trap (dev == 1)

```
    if(dev==1 && fuword(pc-2)==SETD && u.u_signal[SIGINS]==0)
        return;
```

The SETD instruction (set double mode) traps on some PDP-11 models. If the user hasn't registered a handler, just ignore it.

7.6.3. Kernel Mode Trap

```
    if((ps&UMODE) != UMODE)
        goto bad;         /* Trap in kernel mode = panic */
```

Traps in kernel mode (except floating point) are fatal.

7.6.4. Stack Growth (dev == 9)

```
if(dev==9 && sp<-u.u_ssize*64) {
    if(backup(&r0) == 0)
    if(!estabur(u.u_tsize, u.u_dsize, u.u_ssize+SINCR)) {
        u.u_ssize =+ SINCR;
        expand(u.u_procp->p_size+SINCR);
        /* Move stack up */
        a = u.u_procp->p_addr + u.u_procp->p_size;
        for(i=0; i<u.u_ssize; i++) {
            a--;
            copyseg(a-SINCR, a);
        }
        return;
    }
}
```

A segmentation fault that's just past the stack can be handled by **automatic stack growth**:

1. Back up the instruction
2. Expand the stack segment by SINCR blocks
3. Copy stack to new location
4. Return and retry the instruction

7.6.5. Signal Dispatch

```
u.u_error = 0;
switch(dev) {
case 0:
    i = SIGBUS;
    goto def;
case 1:
    i = SIGINS;
    goto def;
case 2:
    i = SIGTRC;
    goto def;
case 3:
    i = SIGIOT;
    goto def;
case 5:
    i = SIGEMT;
    goto def;
case 9:
    i = SIGSEG;
    goto def;
def:
    psignal(u.u_procp, i);
default:
```

```
        u.u_error = dev+100;
    case 6:;                /* System call - fall through */
    }
```

Most traps cause a signal. Device 6 (system call) falls through to the system call handling code.

7.7. System Call Handling

```
    if(u.u_error)
        goto err;
    ps =& ~EBIT;            /* Clear error bit (optimistic) */
    dev = fuword(pc-2)&077;   /* Get syscall number from instruction */
```

The system call number is encoded in the low 6 bits of the `trap` instruction itself.

7.7.1. Indirect System Calls

```
    if(dev == 0) { /* indirect */
        a = fuword(pc);
        pc =+ 2;
        dev = fuword(a)&077;
        a =+ 2;
    } else {
        a = pc;
        pc =+ sysent[dev].count*2;
    }
```

System call 0 is “indirect”—the next word points to the actual syscall.

7.7.2. Fetch Arguments

```
    for(i=0; i<sysent[dev].count; i++) {
        u.u_arg[i] = fuword(a);
        a =+ 2;
    }
    u.u_dirp = u.u_arg[0];    /* First arg often is pathname */
    trap1(sysent[dev].call);   /* Call the handler */
```

Arguments follow the trap instruction in user memory. They’re fetched into `u.u_arg[]`.

7.7.3. Error Handling

```
    if(u.u_error >= 100)
        psignal(u.u_procp, SIGSYS);
err:
    if(issig())
        psig();
    if(u.u_error != 0) {
        ps =| EBIT;           /* Set error bit in PS */
        r0 = u.u_error;       /* Return error code in r0 */
    }
```

If there's an error, the carry bit (EBIT) is set and the error code goes in r0.

7.7.4. Priority and Reschedule

```
    u.u_procp->p_pri = PUSER + u.u_nice;
    if(u.u_dsleep++ > 15) {
        u.u_dsleep = 0;
        u.u_procp->p_pri++;
        swtch();
    }
    return;
```

After handling the syscall, the process priority is recalculated. If the process has been running for a while (`u_dsleep`), it may be preempted.

7.8. The System Call Table

```
/* sysent.c */
int sysent[]
{
    0, &nullsys,      /*  0 = indir */
    0, &rexit,        /*  1 = exit */
    0, &fork,         /*  2 = fork */
    2, &read,         /*  3 = read */
    2, &write,        /*  4 = write */
    2, &open,         /*  5 = open */
    0, &close,        /*  6 = close */
    0, &wait,         /*  7 = wait */
    2, &creat,        /*  8 = creat */
    2, &link,         /*  9 = link */
    1, &unlink,       /* 10 = unlink */
    2, &exec,         /* 11 = exec */
    1, &chdir,        /* 12 = chdir */
```

```
    0, &gtime,          /* 13 = time */
    3, &mknod,          /* 14 = mknod */
    2, &chmod,          /* 15 = chmod */
    2, &chown,          /* 16 = chown */
    1, &sbreak,         /* 17 = break */
    2, &stat,           /* 18 = stat */
    2, &seek,           /* 19 = seek */
    ...
    0, &dup,            /* 41 = dup */
    0, &pipe,           /* 42 = pipe */
    1, &times,          /* 43 = times */
    4, &profil,         /* 44 = prof */
    ...
    2, &ssig,           /* 48 = sig */
    ...
};
```

Each entry is two words:

- **count** — Number of arguments
- **call** — Pointer to handler function

7.9. Making a System Call (User Side)

From user code, a system call looks like:

```
/ read(fd, buf, count)
    mov  fd,r0
    sys  read; buf; count
    bcs  error
    / r0 = bytes read
```

The `sys read` assembles to `trap 3` (read is syscall 3). Arguments follow inline.

The C library provides wrappers:

```
read(fd, buf, count)
char *buf;
{
    return(syscall(3, fd, buf, count));
}
```

7.10. Complete System Call Flow

```
User Program:
    sys write; buf; count
         |
         v
Hardware:
    Push PS, PC
    Load PS, PC from vector 034
         |
         v
mch.s trap:
    Save registers
    Call _trap(6, ...)
         |
         v
trap.c trap():
    dev = fuword(pc-2) & 077  → 4 (write)
    Fetch arguments
    trap1(sysent[4].call)  → write()
         |
         v
sys2.c write():
    Do the actual write
    Set u.u_error if failed
         |
         v
trap.c trap():
    If error, set EBIT and r0
    Return
         |
         v
mch.s:
    Restore registers
    rti
         |
         v
User Program:
    bcs error    / Check carry bit
    / r0 = return value
```

7.11. Error Handling

System calls report errors by:

1. Setting `u.u_error` to an error code
2. Setting the carry bit (EBIT) in the saved PS
3. Returning the error code in r0

User programs check the carry bit:

```
    sys open; file; 0
    bcs error        / Branch if carry set
    mov r0,fd        / r0 = file descriptor
    ...
error:
    / r0 = error code (ENOENT, EACCES, etc.)
```

7.12. The trap1() Function

```
trap1(f)
int (*f)();
{
    savu(u.u_qsav);
    (*f)();
}
```

`trap1()` saves the registers in `u.u_qsav` before calling the syscall handler. This allows signals to abort syscalls and return to user mode via `aretu(u.u_qsav)`.

7.13. Summary

- System calls use the PDP-11 `trap` instruction (vector 034)
- The trap handler saves state and identifies the syscall number
- Arguments are fetched from user memory following the instruction
- The `sysent[]` table maps syscall numbers to handlers
- Errors are returned via carry bit and r0
- The PS word tracks user/kernel mode and error status

7.14. System Call Reference

#	Name	Args	Description
0	indir	0	Indirect syscall
1	exit	0	Terminate process
2	fork	0	Create child process
3	read	2	Read from file
4	write	2	Write to file

#	Name	Args	Description
5	open	2	Open file
6	close	0	Close file
7	wait	0	Wait for child
8	creat	2	Create file
9	link	2	Create hard link
10	unlink	1	Remove file
11	exec	2	Execute program
12	chdir	1	Change directory
13	time	0	Get time
14	mknod	3	Make device node
15	chmod	2	Change mode
16	chown	2	Change owner
17	break	1	Change memory size (sbrk)
18	stat	2	Get file status
19	seek	2	Seek in file
21	mount	3	Mount filesystem
22	umount	1	Unmount filesystem
23	setuid	0	Set user ID
24	getuid	0	Get user ID
25	stime	0	Set system time
28	fstat	1	Get file status (by fd)
30	smdate	1	Set modification date
31	stty	1	Set terminal parameters
32	gtty	1	Get terminal parameters
34	nice	0	Set process priority
35	sleep	0	Sleep for interval
36	sync	0	Flush filesystem buffers
37	kill	1	Send signal to process
38	switch	0	Get console switches

#	Name	Args	Description
41	dup	0	Duplicate fd
42	pipe	0	Create pipe
43	times	1	Get process times
44	prof	4	Profiling control
46	setgid	0	Set group ID
47	getgid	0	Get group ID
48	signal	2	Set signal handler

7.15. Experiments

1. **Add a syscall**: Add a new syscall that returns a constant. Modify sysent.c and test it.
2. **Trace syscalls**: Add printf to trap() to log every syscall.
3. **Error injection**: Modify a syscall to always fail and watch programs break.

7.16. Further Reading

- Chapter 5: Process Management — fork, exec, exit, wait
- Chapter 10: File I/O — read, write, open, close
- Appendix A: Complete syscall reference

Next: Chapter 8 — Scheduling

8. Chapter 8: Scheduling

8.1. Overview

UNIX v4 uses a simple but effective scheduling algorithm: priority-based preemptive scheduling with aging. This chapter examines how the scheduler decides which process runs, how priorities are calculated, and how context switching works. The elegance is in the simplicity—about 200 lines of code manage all process scheduling.

8.2. Source Files

File	Purpose
`usr/sys/ken/slp.c`	`sched()`, `swtch()`, `sleep()`, `wakeup()`
`usr/sys/ken/clock.c`	Clock interrupt, priority aging
`usr/sys/param.h`	Priority constants
`usr/sys/proc.h`	Process state definitions

8.3. Prerequisites

- Chapter 5: Process Management (process structure)
- Chapter 6: Memory Management (swapping)
- Chapter 7: Traps and System Calls (interrupt handling)

8.4. The Scheduling Model

UNIX v4 scheduling has two levels:

1. **Swapper (process 0)** — Decides which processes are in memory
2. **swtch()** — Chooses among in-memory runnable processes

The swapper runs `sched()` in an infinite loop, swapping processes in and out. The `swtch()` function is called when a process blocks or when the clock decides it's time for preemption.

8.5. Priority Basics

Lower numbers = higher priority:

```
/* param.h */
#define PSWP    -100    /* Swapper */
#define PINOD   -90     /* Waiting for inode */
#define PRIBIO  -50     /* Waiting for buffer I/O */
#define PPIPE   1       /* Waiting for pipe */
#define PWAIT   40      /* Waiting for child (wait syscall) */
#define PSLEP   90      /* Sleeping (sleep syscall) */
#define PUSER   100     /* Base user priority */
```

Negative priorities are for kernel waits and cannot be interrupted by signals. Positive priorities are interruptible.

8.6. The sleep() Function

```
/* slp.c */
sleep(chan, pri)
{
    register *rp, s;

    u.u_dsleep = 0;
    s = PS->integ;
    rp = u.u_procp;
```

`sleep()` puts the current process to sleep waiting for an event (identified by `chan`).

```
    if(pri >= 0) {
        /* Interruptible sleep */
        if(issig())
            goto psig;
        rp->p_wchan = chan;
        rp->p_stat = SWAIT;
        rp->p_pri = pri;
        spl0();
        if(runin != 0) {
            runin = 0;
            wakeup(&runin);
        }
        swtch();
```

```
        if(issig()) {
        psig:
            aretu(u.u_qsav);
            return;
        }
```

For interruptible sleeps (pri >= 0):

1. Check for pending signals first
2. Set wait channel and state SWAIT
3. Context switch away
4. On wakeup, check for signals again

```
    } else {
        /* Uninterruptible sleep */
        rp->p_wchan = chan;
        rp->p_stat = SSLEEP;
        rp->p_pri = pri;
        spl0();
        swtch();
    }
    PS->integ = s;
}
```

For uninterruptible sleeps (pri < 0), the process sleeps until explicitly awakened.

8.7. The wakeup() Function

```
/* slp.c */
wakeup(chan)
{
    register struct proc *p;
    register n, c;

loop:
    c = chan;
    n = 0;
    for(p = &proc[0]; p < &proc[NPROC]; p++)
        if(p->p_wchan == c) {
            if(runout!=0 && (p->p_flag&SLOAD)==0) {
                runout = 0;
                n++;
            }
            p->p_wchan = 0;
            p->p_stat = SRUN;
            runrun++;
        }
    if(n) {
```

```
        chan = &runout;
        goto loop;
    }
}
```

wakeup() marks all processes sleeping on chan as runnable:

1. Scan the process table for matching p_wchan
2. Clear p_wchan, set p_stat = SRUN
3. Increment runrun to trigger rescheduling
4. If any swapped processes were awakened, wake the swapper too

8.8. The swtch() Function

```
/* slp.c */
swtch()
{
    static int *p;
    register i, n;
    register struct proc *rp;

    if(p == NULL)
        p = &proc[0];
    savu(u.u_rsav);          /* Save current context */
    retu(proc[0].p_addr);    /* Switch to process 0's context */
```

First, save the current process's registers and switch to process 0's address space (so we can access the proc table).

```
loop:
    rp = p;
    p = NULL;
    n = 127;                 /* Start with lowest priority */
    for(i=0; i<NPROC; i++) {
        rp++;
        if(rp >= &proc[NPROC])
            rp = &proc[0];
        if(rp->p_stat==SRUN && (rp->p_flag&SLOAD)==SLOAD) {
            if(rp->p_pri < n) {
                p = rp;
                n = rp->p_pri;
            }
        }
    }
```

Search for the highest-priority runnable, in-memory process. The search starts from where we left off (round-robin among equal priorities).

```
    if(p == NULL) {
        p = rp;
        idle();                 /* No runnable process - wait */
        goto loop;
    }
```

If nothing is runnable, call `idle()` to wait for an interrupt.

```
    rp = p;
    retu(rp->p_addr);           /* Switch to new process */
    sureg();                    /* Set up segment registers */
    if(rp->p_flag&SSWAP) {
        rp->p_flag =& ~SSWAP;
        aretu(u.u_ssav);        /* Return from swap */
    }
    return(1);
}
```

Switch to the selected process's address space and return.

8.9. The sched() Function (Swapper)

Process 0 runs `sched()` forever:

```
/* slp.c */
sched()
{
    struct proc *p1;
    register struct proc *rp;
    register a, n;

    /*
     * find user to swap in
     * of users ready, select one out longest
     */
    goto loop;

sloop:
    runin++;
    sleep(&runin, PSWP);

loop:
    spl6();
    n = -1;
    for(rp = &proc[0]; rp < &proc[NPROC]; rp++)
    if(rp->p_stat==SRUN && (rp->p_flag&SLOAD)==0 &&
        rp->p_time > n) {
        p1 = rp;
        n = rp->p_time;
    }
```

```
    if(n == -1) {
        runout++;
        sleep(&runout, PSWP);
        goto loop;
    }
```

Find a swapped-out process that's been waiting longest (`p_time`).

```
    /*
     * see if there is core for that process
     */
    spl0();
    rp = p1;
    a = rp->p_size;
    if((rp=rp->p_textp) != NULL)
        if(rp->x_ccount == 0)
            a =+ rp->x_size;
    if((a=malloc(coremap, a)) != NULL)
        goto found2;
```

Try to allocate memory for it.

```
    /*
     * none found,
     * look around for easy core
     */
    spl6();
    for(rp = &proc[0]; rp < &proc[NPROC]; rp++)
    if((rp->p_flag&(SSYS|SLOCK|SLOAD))==SLOAD &&
        rp->p_stat == SWAIT)
        goto found1;
```

If no memory, look for an easy victim—a process that's sleeping.

```
    /*
     * no easy core,
     * if this process is deserving,
     * look around for
     * oldest process in core
     */
    if(n < 3)
        goto sloop;
    n = -1;
    for(rp = &proc[0]; rp < &proc[NPROC]; rp++)
    if((rp->p_flag&(SSYS|SLOCK|SLOAD))==SLOAD &&
       (rp->p_stat==SRUN || rp->p_stat==SSLEEP) &&
        rp->p_time > n) {
        p1 = rp;
        n = rp->p_time;
    }
    if(n < 2)
        goto sloop;
    rp = p1;
```

If no sleeping process, find the oldest in-memory process. But don't swap out a process that's only been in memory briefly (n < 3 and n < 2 checks).

```
    /*
     * swap user out
     */
found1:
    spl0();
    rp->p_flag =& ~SLOAD;
    xswap(rp, 1, 0);
    goto loop;

    /*
     * swap user in
     */
found2:
    /* ... swap in code ... */
    goto loop;
}
```

The swapper either swaps out a victim or swaps in the waiting process, then loops.

8.10. The Clock Interrupt

The clock ticks 60 times per second:

```
/* clock.c */
clock(dev, sp, r1, nps, r0, pc, ps)
{
    register struct callo *p1, *p2;
    register struct proc *pp;

    *lks = 0115;              /* Restart clock */
    display();                /* Update console display */
```

8.10.1. Callouts

```
    /*
     * callouts - decrement timers
     */
    if(callout[0].c_func == 0)
        goto out;
    p2 = &callout[0];
    while(p2->c_time<=0 && p2->c_func!=0)
        p2++;
    p2->c_time--;
```

```
    if((ps&0340) != 0)        /* If IPL high, don't run callouts */
        goto out;

    spl5();
    if(callout[0].c_time <= 0) {
        /* Run expired callouts */
        p1 = &callout[0];
        while(p1->c_func != 0 && p1->c_time <= 0) {
            (*p1->c_func)(p1->c_arg);
            p1++;
        }
        /* Compact the callout table */
        ...
    }
```

Callouts are timed callbacks. Each tick decrements the first non-zero timer.

8.10.2. Time Accounting

```
out:
    if((ps&UMODE) == UMODE) {
        u.u_utime++;          /* User mode: charge user time */
        if(u.u_prof[3])
            incupc(pc, u.u_prof);   /* Profiling */
    } else
        u.u_stime++;          /* Kernel mode: charge system time */
```

8.10.3. Every Second (60 ticks)

```
    if(++lbolt >= 60) {
        if((ps&0340) != 0)
            return;
        lbolt -= 60;
        if(++time[1] == 0)
            ++time[0];        /* Increment time of day */

        spl1();
        if(time[1]==tout[1] && time[0]==tout[0])
            wakeup(tout);     /* Wake alarm sleepers */
        if((time[1]&03) == 0)
            wakeup(&lbolt);   /* Wake every 4 seconds */
```

8.10.4. Priority Aging

```
        for(pp = &proc[0]; pp < &proc[NPROC]; pp++)
        if(pp->p_time != 127)
            pp->p_time++;     /* Age all processes */
```

`p_time` counts how long a process has been in its current location (memory or swap).

8.10.5. Preemption

```
        if((ps&UMODE) == UMODE) {
            u.u_ar0 = &r0;
            pp = u.u_procp;
            if(issig())
                psig();
            if(pp->p_pri < 105)
                pp->p_pri++;  /* Lower priority (higher number) */
            savfp();
            swtch();          /* Preempt! */
        }
    }
}
```

Once per second, if we're in user mode:

1. Check for signals
2. Decay the process's priority
3. Call `swtch()` to potentially run another process

8.11. Priority Calculation

Priorities in UNIX v4 are simple:

1. **Initial priority**: Set by `sleep()` based on what the process is waiting for
2. **User processes**: Start at `PUSER + u.u_nice` (100 + nice value)
3. **Aging**: Once per second, increment priority (lower priority)
4. **Recalculation**: After a syscall, reset to `PUSER + u.u_nice`

From `trap.c`:

```
    u.u_procp->p_pri = PUSER + u.u_nice;
```

This creates a simple feedback loop:

- Processes that use CPU time get lower priority
- Processes that sleep get reset to high priority when they wake
- I/O-bound processes naturally get better priority than CPU-bound

8.12. Context Switching

The actual context switch uses three assembly functions:

```
savu(u.u_rsav)     /* Save current sp and r5 */
retu(p->p_addr)    /* Switch to process p's memory, restore sp/r5 */
aretu(u.u_qsav)    /* Return to saved context (for signals) */
```

From `mch.s`:

```
_savu:
    spl  7                  / Disable interrupts
    mov  (sp)+,r1           / Return address
    mov  (sp),r0            / Save area pointer
    mov  sp,(r0)+           / Save sp
    mov  r5,(r0)+           / Save r5
    spl  0                  / Enable interrupts
    jmp  (r1)               / Return

_retu:
    spl  7
    mov  (sp)+,r1
    mov  (sp),KISA6         / Set segment 6 to new process
    mov  $_u,r0
    mov  (r0)+,sp           / Restore sp
    mov  (r0)+,r5           / Restore r5
    spl  0
    jmp  (r1)
```

The key is changing `KISA6`—segment 6 points to the user structure, so changing it switches to a different process's context.

8.13. Scheduling Flags

Three flags coordinate scheduling:

- **runrun** — Set when a higher-priority process becomes runnable
- **runin** — Set when swapper should look for something to swap in
- **runout** — Set when swapper should look for something to swap out

When `runrun` is set, `swtch()` is called at the next opportunity.

8.14. Summary

- Scheduling is priority-based: lower number = higher priority

- `sleep()` blocks a process on a “wait channel”
- `wakeup()` makes all processes on a channel runnable
- `swtch()` picks the highest-priority runnable process
- `sched()` (process 0) handles swapping
- The clock provides preemption and priority aging
- Context switching changes segment 6 to point to a different user structure

8.15. The Beauty of Simplicity

The entire scheduler fits in about 200 lines:

- No run queues (just scan the proc table)
- No complex priority inheritance
- No real-time scheduling
- Just: find highest priority, run it, age priorities

This works because:

- Only 50 processes maximum
- Clock provides regular preemption
- I/O-bound processes naturally get good priority

8.16. Experiments

1. **Watch scheduling**: Add printf to `swtch()` to see process switches.
2. **Change priorities**: Modify the priority constants and observe behavior.
3. **Disable preemption**: Remove the `swtch()` call from `clock()` and see what happens.

8.17. Further Reading

- Chapter 5: Process Management — Process states and transitions
- Chapter 6: Memory Management — How swapping interacts with scheduling
- Chapter 7: Traps and System Calls — How syscalls trigger rescheduling

---

End of Part II: The Kernel

Next: Part III — The File System

Part III.

The File System

9. Chapter 9: Inodes and the Superblock

9.1. Overview

The UNIX file system is built on two fundamental abstractions: the **inode** (index node) and the **superblock**. Every file—whether a regular file, directory, or device—is represented by an inode that stores its metadata and block addresses. The superblock contains critical file system parameters and maintains caches of free blocks and free inodes for fast allocation.

This chapter examines how UNIX v4 organizes data on disk and manages the in-memory inode cache—the foundation upon which all file operations are built.

9.2. Source Files

File	Purpose
`usr/sys/inode.h`	In-memory inode structure and flags
`usr/sys/filsys.h`	Superblock structure
`usr/sys/ken/iget.c`	Inode cache operations
`usr/sys/ken/alloc.c`	Block and inode allocation

9.3. Prerequisites

- Chapter 2: PDP-11 Architecture (memory layout)
- Chapter 4: Boot Sequence (`iinit()` called during startup)
- Chapter 12: Buffer Cache (understanding `bread`/`bwrite`)

9.4. Disk Layout

A UNIX v4 file system has this structure:

```
Block 0     Boot block (bootstrap code)
Block 1     Superblock (file system metadata)
Block 2     ┐
  ...       │ Inode area (on-disk inodes)
Block N     ┘
Block N+1   ┐
  ...       │ Data blocks
Block M     ┘
```

Each block is 512 bytes. The number of inode blocks depends on the file system size—the superblock's `s_isize` field records this.

> **Why 512 bytes?** The block size matches the RK05 disk's physical sector size (256 words × 2 bytes = 512 bytes). This makes I/O efficient—one block equals one sector equals one disk operation. The power-of-2 size also makes offset calculations fast (bit shifts instead of division), and with only 64KB of address space, larger buffers would waste precious memory.

> **How many inodes?** The `mkfs` utility calculates inode blocks using: `s_isize = fsize / (43 + fsize/1000)`. For an RK05 with ~4800 blocks, this yields ~100 inode blocks = 1600 inodes (16 per block). The formula allocates roughly 1 inode per 3 data blocks, or about 1 inode per 1.5KB—enough for typical small UNIX files.

9.4.1. On-Disk Inode Format

Each on-disk inode is 32 bytes:

```
Offset  Size  Field
0       2     i_mode    (type/permissions)
2       1     i_nlink   (link count)
3       1     i_uid     (owner)
4       1     i_gid     (group)
5       1     i_size0   (size high byte)
6       2     i_size1   (size low bytes)
8       16    i_addr[8] (block addresses)
24      4     i_atime   (access time)
28      4     i_mtime   (modification time)
```

With 32 bytes per inode, each 512-byte block holds 16 inodes. Inode numbering starts at 1 (inode 0 is unused); inode 1 is the root directory (`ROOTINO`).

9.5. The In-Memory Inode

When a file is opened, its inode is read into memory and cached:

```c
/* inode.h */
struct inode {
    char  i_flag;       /* Flags (ILOCK, IUPD, etc.) */
    char  i_count;      /* Reference count */
    int   i_dev;        /* Device number */
    int   i_number;     /* Inode number on device */
    int   i_mode;       /* Type and permissions */
    char  i_nlink;      /* Link count */
    char  i_uid;        /* Owner user ID */
    char  i_gid;        /* Owner group ID */
    char  i_size0;      /* Size (high byte) */
    char  *i_size1;     /* Size (low 16 bits) */
    int   i_addr[8];    /* Block addresses */
    int   i_lastr;      /* Last block read (for read-ahead) */
} inode[NINODE];
```

The in-memory inode has additional fields not stored on disk:

Field	Purpose
`i_flag`	Lock state, update pending, mount point
`i_count`	Number of references to this cached inode
`i_dev`	Which device this inode is from
`i_number`	Which inode number on that device
`i_lastr`	Enables read-ahead optimization

9.5.1. Inode Flags

```c
#define ILOCK   01      /* Inode is locked */
#define IUPD    02      /* Inode modified, needs write */
#define IACC    04      /* Access time changed */
#define IMOUNT  010     /* Mount point */
#define IWANT   020     /* Process waiting for lock */
#define ITEXT   040     /* Text segment (shared executable) */
```

9.5.2. Mode Bits

The `i_mode` field encodes file type and permissions:

```c
#define IALLOC  0100000   /* Inode is allocated */
#define IFMT    060000    /* Type mask */
#define IFDIR   040000    /* Directory */
#define IFCHR   020000    /* Character device */
```

```
#define IFBLK   060000    /* Block device */
#define ILARG   010000    /* Large file (indirect blocks) */
#define ISUID   04000     /* Set-user-ID */
#define ISGID   02000     /* Set-group-ID */
#define IREAD   0400      /* Owner read */
#define IWRITE  0200      /* Owner write */
#define IEXEC   0100      /* Owner execute */
```

File types encoded in `IFMT`:

- `000000` — Regular file
- `040000` — Directory
- `020000` — Character special
- `060000` — Block special

9.6. The Superblock

The superblock lives in block 1 and describes the file system:

```
/* filsys.h */
struct filsys {
    int   s_isize;      /* Size of inode area in blocks */
    int   s_fsize;      /* Total size in blocks */
    int   s_nfree;      /* Number of free blocks in cache */
    int   s_free[100];  /* Free block cache */
    int   s_ninode;     /* Number of free inodes in cache */
    int   s_inode[100]; /* Free inode cache */
    char  s_flock;      /* Lock during free list manipulation */
    char  s_ilock;      /* Lock during inode list manipulation */
    char  s_fmod;       /* Superblock modified flag */
    char  s_ronly;      /* Read-only flag */
    int   s_time[2];    /* Last modification time */
};
```

The superblock caches up to 100 free block numbers and 100 free inode numbers in memory. This dramatically speeds allocation—most allocations don't require disk I/O.

9.7. iinit() — File System Initialization

Called once during boot to initialize the root file system:

```
/* alloc.c */
iinit()
{
    register *cp, *bp;
```

```
    bp = bread(rootdev, 1);         /* Read superblock */
    cp = getblk(NODEV);             /* Get buffer for in-memory copy */
    if(u.u_error)
        panic("iinit");
    bcopy(bp->b_addr, cp->b_addr, 256);  /* Copy superblock */
    brelse(bp);
    mount[0].m_bufp = cp;           /* Save in mount table */
    mount[0].m_dev = rootdev;
    cp = cp->b_addr;
    cp->s_flock = 0;                /* Clear locks */
    cp->s_ilock = 0;
    cp->s_ronly = 0;
    time[0] = cp->s_time[0];        /* Initialize system time */
    time[1] = cp->s_time[1];
}
```

Key points:

1. The superblock is read from disk block 1
2. It's copied into a dedicated buffer that stays in memory
3. The mount table entry records this buffer
4. System time is initialized from the superblock

9.8. iget() — Getting an Inode

`iget()` retrieves an inode, either from cache or disk:

```
/* iget.c */
iget(dev, ino)
int dev;
int ino;
{
    register struct inode *p;
    register *ip2;
    int *ip1;
    register struct mount *ip;

loop:
    ip = NULL;
    for(p = &inode[0]; p < &inode[NINODE]; p++) {
        if(dev==p->i_dev && ino==p->i_number) {
            /* Found in cache */
            if((p->i_flag&ILOCK) != 0) {
                p->i_flag =| IWANT;
                sleep(p, PINOD);
                goto loop;          /* Retry after wakeup */
            }
```

The first loop searches the inode cache. If found but locked, the process sleeps until the lock is released.

```
            if((p->i_flag&IMOUNT) != 0) {
                /* This is a mount point - cross to mounted fs */
                for(ip = &mount[0]; ip < &mount[NMOUNT]; ip++)
                if(ip->m_inodp == p) {
                    dev = ip->m_dev;
                    ino = ROOTINO;
                    goto loop;
                }
                panic("no imt");
            }
            p->i_count++;
            p->i_flag =| ILOCK;
            return(p);
        }
        if(ip==NULL && p->i_count==0)
            ip = p;                 /* Remember free slot */
    }
```

Mount point handling: if the inode is a mount point, `iget()` transparently crosses to the mounted file system's root.

```
    if((p=ip) == NULL)
        panic("no inodes");
    if (p>maxip)
        maxip = p;
    p->i_dev = dev;
    p->i_number = ino;
    p->i_flag = ILOCK;
    p->i_count++;
    p->i_lastr = -1;
```

If not in cache, use the free slot found during the search. Initialize the in-memory fields.

```
    ip = bread(dev, ldiv(ino+31,16));
    ip1 = ip->b_addr + 32*lrem(ino+31, 16);
    ip2 = &p->i_mode;
    while(ip2 < &p->i_addr[8])
        *ip2++ = *ip1++;
    brelse(ip);
    return(p);
}
```

The disk block containing the inode is calculated:

- `ldiv(ino+31, 16)` gives the block number (16 inodes per block, offset by 2 for boot+super, so +31 adjusts for 1-based inode numbers)
- `32*lrem(ino+31, 16)` gives the byte offset within the block

The on-disk inode (32 bytes from `i_mode` through `i_addr[7]`) is copied into the in-memory structure.

9.9. iput() — Releasing an Inode

`iput()` decrements the reference count and handles cleanup:

```
/* iget.c */
iput(p)
struct inode *p;
{
    register *rp;

    rp = p;
    if(rp->i_count == 1) {
        rp->i_flag =| ILOCK;
        if(rp->i_nlink <= 0) {
            itrunc(rp);             /* Free all data blocks */
            rp->i_mode = 0;         /* Mark as unallocated */
            ifree(rp->i_dev, rp->i_number);
        }
        iupdat(rp);                 /* Write changes to disk */
        prele(rp);
        rp->i_flag = 0;
        rp->i_number = 0;
    }
    rp->i_count--;
    prele(rp);
}
```

When the last reference is released (`i_count` goes to 0):

1. If link count is zero, the file is deleted—`itrunc()` frees data blocks, `ifree()` frees the inode
2. `iupdat()` writes any pending changes to disk
3. The inode slot is cleared for reuse

9.10. iupdat() — Writing Inode to Disk

```
/* iget.c */
iupdat(p)
int *p;
{
    register *ip1, *ip2, *rp;
    int *bp, i;

    rp = p;
    if((rp->i_flag&(IUPD|IACC)) != 0) {
        if(getfs(rp->i_dev)->s_ronly)
            return;                 /* Read-only filesystem */
        i = rp->i_number+31;
        bp = bread(rp->i_dev, ldiv(i,16));
        ip1 = bp->b_addr + 32*lrem(i, 16);
```

```
        ip2 = &rp->i_mode;
        while(ip2 < &rp->i_addr[8])
            *ip1++ = *ip2++;        /* Copy inode to buffer */
        if(rp->i_flag&IACC) {
            *ip1++ = time[0];       /* Update access time */
            *ip1++ = time[1];
        } else
            ip1 =+ 2;
        if(rp->i_flag&IUPD) {
            *ip1++ = time[0];       /* Update modification time */
            *ip1++ = time[1];
        }
        bwrite(bp);                 /* Write synchronously */
    }
}
```

Only writes if `IUPD` or `IACC` flags are set. The times are written after the main inode data.

9.11. itrunc() — Truncating a File

When a file is deleted or truncated, its blocks must be freed:

```
/* iget.c */
itrunc(ip)
int *ip;
{
    register *rp, *bp, *cp;

    rp = ip;
    if((rp->i_mode&(IFCHR&IFBLK)) != 0)
        return;                     /* Devices have no blocks */
    for(ip = &rp->i_addr[0]; ip < &rp->i_addr[8]; ip++)
    if(*ip) {
        if((rp->i_mode&ILARG) != 0) {
            /* Large file: this is an indirect block */
            bp = bread(rp->i_dev, *ip);
            for(cp = bp->b_addr; cp < bp->b_addr+512; cp++)
                if(*cp)
                    free(rp->i_dev, *cp);
            brelse(bp);
        }
        free(rp->i_dev, *ip);
        *ip = 0;
    }
    rp->i_mode =& ~ILARG;
    rp->i_size0 = 0;
    rp->i_size1 = 0;
    rp->i_flag =| IUPD;
}
```

For small files (ILARG not set), `i_addr[0-7]` point directly to data blocks.

For large files (ILARG set), `i_addr[0-7]` point to indirect blocks, each containing 256 block numbers. The indirect blocks are read, their contents freed, then the indirect blocks themselves are freed.

9.12. Block Allocation

9.12.1. alloc() — Allocate a Block

```
/* alloc.c */
alloc(dev)
{
    int bno;
    register *bp, *ip, *fp;

    fp = getfs(dev);
    while(fp->s_flock)
        sleep(&fp->s_flock, PINOD);
    bno = fp->s_free[--fp->s_nfree];
    if(bno == 0) {
        fp->s_nfree++;
        printf("No space on dev %d\n", dev);
        u.u_error = ENOSPC;
        return(NULL);
    }
```

The superblock caches free blocks in `s_free[]`. Allocation pops from this array.

```
    if(fp->s_nfree <= 0) {
        /* Cache empty - reload from linked list */
        fp->s_flock++;
        bp = bread(dev, bno);
        ip = bp->b_addr;
        fp->s_nfree = *ip++;
        bcopy(ip, fp->s_free, 100);
        brelse(bp);
        fp->s_flock = 0;
        wakeup(&fp->s_flock);
    }
    bp = getblk(dev, bno);
    clrbuf(bp);
    fp->s_fmod = 1;
    return(bp);
}
```

When the cache empties, the block just "allocated" is actually a link block—it contains the next batch of 100 free block numbers. This creates a linked list of free block batches across the disk.

9.12.2. free() — Free a Block

```
/* alloc.c */
free(dev, bno)
{
    register *fp, *bp, *ip;

    fp = getfs(dev);
    fp->s_fmod = 1;
    while(fp->s_flock)
        sleep(&fp->s_flock, PINOD);
    if(fp->s_nfree >= 100) {
        /* Cache full - flush to disk */
        fp->s_flock++;
        bp = getblk(dev, bno);
        ip = bp->b_addr;
        *ip++ = fp->s_nfree;
        bcopy(fp->s_free, ip, 100);
        fp->s_nfree = 0;
        bwrite(bp);
        fp->s_flock = 0;
        wakeup(&fp->s_flock);
    }
    fp->s_free[fp->s_nfree++] = bno;
    fp->s_fmod = 1;
}
```

The reverse of `alloc()`: when the cache fills, the current 100 free blocks are written to the block being freed, creating a new link in the chain.

9.12.3. The Free Block List

```
   Superblock          Link Block 1         Link Block 2
+----------------+   +----------------+   +----------------+
| s_nfree = 47   |   | count = 100    |   | count = 100    |
| s_free[0..46] -+-->| blocks[0..99] -+-->| blocks[0..99] -+--> ...
+----------------+   +----------------+   +----------------+
```

This design means:

- Most allocations require no disk I/O (just decrement s_nfree)
- The free list is rebuilt in batches, amortizing disk access

9.13. Inode Allocation

9.13.1. ialloc() — Allocate an Inode

```
/* alloc.c */
ialloc(dev)
{
    register *fp, *bp, *ip;
    int i, j, k, ino;

    fp = getfs(dev);
    while(fp->s_ilock)
        sleep(&fp->s_ilock, PINOD);
loop:
    if(fp->s_ninode > 0) {
        /* Use cached free inode number */
        ino = fp->s_inode[--fp->s_ninode];
        ip = iget(dev, ino);
        if(ip->i_mode == 0) {
            for(bp = &ip->i_mode; bp < &ip->i_addr[8];)
                *bp++ = 0;
            fp->s_fmod = 1;
            return(ip);
        }
        /* Inode was busy - try next */
        printf("busy i\n");
        iput(ip);
        goto loop;
    }
```

Like blocks, free inode numbers are cached in the superblock. If available, pop one and verify it's actually free.

```
    /* Cache empty - scan inode area */
    fp->s_ilock++;
    ino = 0;
    for(i=0; i<fp->s_isize; i++) {
        bp = bread(dev, i+2);       /* Inode blocks start at 2 */
        ip = bp->b_addr;
        for(j=0; j<256; j=+16) {    /* 16 inodes per block, 32 bytes each */
            ino++;
            if(ip[j] != 0)          /* Check i_mode - 0 means free */
                continue;
            /* Skip if currently in use in memory */
            for(k=0; k<NINODE; k++)
            if(dev==inode[k].i_dev && ino==inode[k].i_number)
                goto cont;
            fp->s_inode[fp->s_ninode++] = ino;
            if(fp->s_ninode >= 100)
                break;
        cont:;
        }
```

```
        brelse(bp);
        if(fp->s_ninode >= 100)
            break;
    }
    if(fp->s_ninode <= 0)
        panic("out of inodes");
    fp->s_ilock = 0;
    wakeup(&fp->s_ilock);
    goto loop;
}
```

When the cache empties, the entire inode area is scanned to refill it. This is expensive but rare—the cache holds 100 inodes.

9.13.2. ifree() — Free an Inode

```
/* alloc.c */
ifree(dev, ino)
{
    register *fp;

    fp = getfs(dev);
    if(fp->s_ilock)
        return;                     /* Someone scanning - skip */
    if(fp->s_ninode >= 100)
        return;                     /* Cache full - will be found by scan */
    fp->s_inode[fp->s_ninode++] = ino;
    fp->s_fmod = 1;
}
```

Unlike block freeing, `ifree()` is simple: just add to cache if there's room. If the cache is full, the freed inode will be found on the next `ialloc()` scan.

9.14. getfs() — Finding a File System

```
/* alloc.c */
getfs(dev)
{
    register struct mount *p;

    for(p = &mount[0]; p < &mount[NMOUNT]; p++)
        if(p->m_bufp != NULL && p->m_dev == dev) {
            p = p->m_bufp->b_addr;
            return(p);
        }
    panic("no fs");
}
```

Given a device number, return a pointer to its in-memory superblock. The mount table maps devices to superblock buffers.

9.15. update() — Sync to Disk

`update()` flushes all modified data to disk:

```
/* alloc.c */
update()
{
    register struct inode *ip;
    register struct mount *mp;
    register *bp;

    if(updlock)
        return;
    updlock++;

    /* Write all modified superblocks */
    for(mp = &mount[0]; mp < &mount[NMOUNT]; mp++)
        if(mp->m_bufp != NULL) {
            ip = mp->m_bufp->b_addr;
            if(ip->s_fmod==0 || ip->s_ilock!=0 ||
               ip->s_flock!=0 || ip->s_ronly!=0)
                continue;
            bp = getblk(mp->m_dev, 1);
            ip->s_fmod = 0;
            ip->s_time[0] = time[0];
            ip->s_time[1] = time[1];
            bcopy(ip, bp->b_addr, 256);
            bwrite(bp);
        }
```

First, all modified superblocks are written to their block 1.

```
    /* Write all modified inodes */
    for(ip = &inode[0]; ip < &inode[NINODE]; ip++)
        if((ip->i_flag&ILOCK) == 0) {
            ip->i_flag =| ILOCK;
            iupdat(ip);
            prele(ip);
        }
    updlock = 0;
    bflush(NODEV);
}
```

Then all unlocked inodes with pending changes are written. Finally, `bflush()` writes all dirty buffers. This is called by `sync(2)` and periodically by the update daemon.

9.16. maknode() — Creating a New File

```
/* iget.c */
maknode(mode)
{
    register *ip;

    ip = ialloc(u.u_pdir->i_dev);
    ip->i_flag =| IACC|IUPD;
    ip->i_mode = mode|IALLOC;
    ip->i_nlink = 1;
    ip->i_uid = u.u_uid;
    ip->i_gid = u.u_gid;
    wdir(ip);
    return(ip);
}
```

Allocates a new inode and creates a directory entry for it:

1. `ialloc()` gets a free inode
2. Initialize mode, link count, owner
3. `wdir()` writes the directory entry

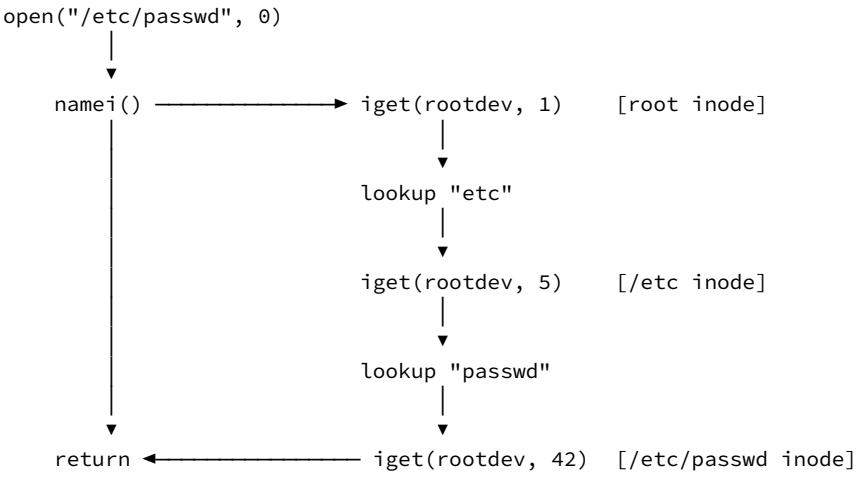
9.17. wdir() — Writing a Directory Entry

```
/* iget.c */
wdir(ip)
int *ip;
{
    register char *cp1, *cp2;

    u.u_dent.u_ino = ip->i_number;
    cp1 = &u.u_dent.u_name[0];
    for(cp2 = &u.u_dbuf[0]; cp2 < &u.u_dbuf[DIRSIZ];)
        *cp1++ = *cp2++;
    u.u_count = DIRSIZ+2;           /* 14 bytes name + 2 bytes ino */
    u.u_segflg = 1;
    u.u_base = &u.u_dent;
    writei(u.u_pdir);
    iput(u.u_pdir);
}
```

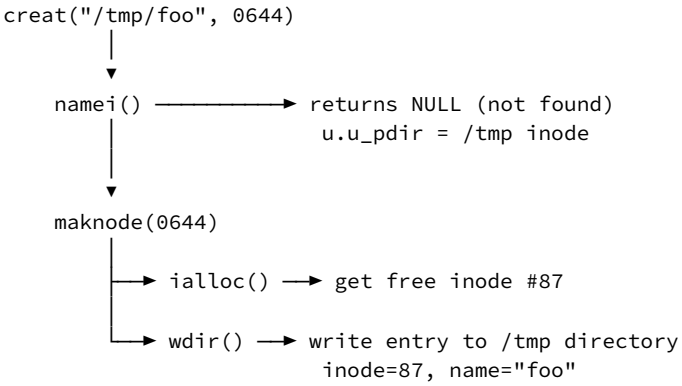
Directory entries are 16 bytes: 2-byte inode number + 14-byte name. The entry is written at the position found by `namei()` (stored in `u.u_offset`).

9.18. How It All Fits Together

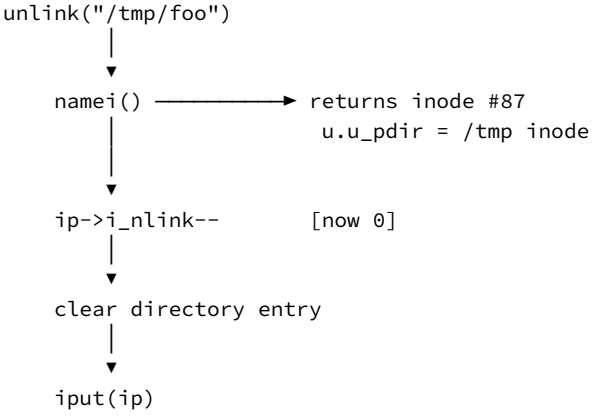
9.18.1. Opening a File

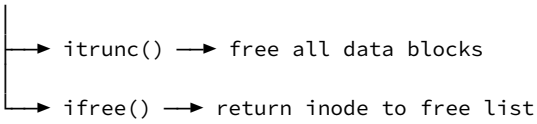


9.18.2. Creating a File



9.18.3. Deleting a File





9.19. Summary

- **Inodes** store all file metadata except the name
- **Superblock** holds file system parameters and free lists
- **iget/iput** manage the in-memory inode cache with reference counting
- **alloc/free** manage disk blocks using a linked-list cache
- **ialloc/ifree** manage inodes using a simple array cache
- **update** syncs everything to disk

The elegance of this design:

1. Most operations hit the in-memory cache
2. Reference counting prevents premature deallocation
3. The free block list amortizes disk I/O
4. Lock flags prevent concurrent modification

9.20. Key Constants

Constant	Value	Meaning
`NINODE`	100	In-memory inode cache size
`ROOTINO`	1	Root directory inode number
`DIRSIZ`	14	Maximum filename length

9.21. Experiments

1. **Trace inode allocation**: Add printf to `ialloc()` to see when the cache is refilled.
2. **Count cache hits**: Track how often `iget()` finds inodes in cache vs. reading from disk.
3. **Free list structure**: Examine the free block list by reading link blocks with od.
4. **Fill the filesystem**: Create files until “No space” appears, then delete some and observe `alloc()` behavior.

9.22. Further Reading

- Chapter 10: File I/O — How data flows through inodes
- Chapter 11: Path Resolution — How `namei()` traverses directories
- Chapter 12: Buffer Cache — The `bread`/`bwrite` interface used here

Next: Chapter 10 — File I/O

10. Chapter 10: File I/O

10.1. Overview

When a user program calls `read()` or `write()`, the request passes through three layers of abstraction: **file descriptors** (per-process), the **open file table** (system-wide), and **inodes** (representing files). This chapter traces the data path from user space through these layers, examining how UNIX v4 translates file positions into disk blocks and moves data between user memory and the buffer cache.

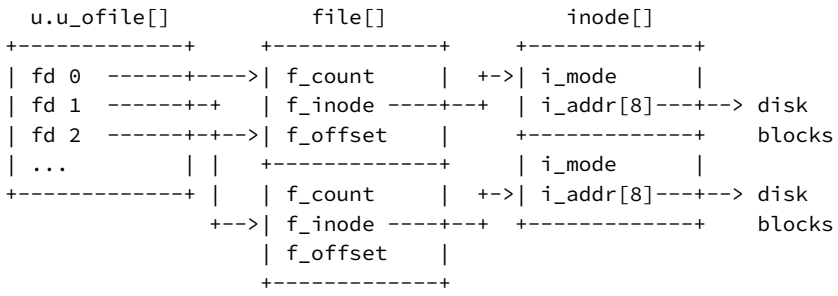
10.2. Source Files

File	Purpose
`usr/sys/file.h`	Open file table structure
`usr/sys/ken/fio.c`	File descriptor operations
`usr/sys/ken/rdwri.c`	`readi()`, `writei()`, `iomove()`
`usr/sys/ken/subr.c`	`bmap()` block mapping

10.3. Prerequisites

- Chapter 9: Inodes and Superblock (inode structure, `i_addr[]`)
- Chapter 12: Buffer Cache (`bread`, b`write`, `brelse`)

10.4. The Three-Level File Abstraction



File descriptors (`u.u_ofile[]`): Per-process array of pointers to open file entries. Small integers (0, 1, 2…) that user programs use.

Open file table (`file[]`): System-wide array of open file structures. Contains the current file offset and a pointer to the inode. Multiple descriptors can point to the same file entry (after `dup()` or `fork()`).

Inodes (`inode[]`): In-memory cache of file metadata. Contains block addresses. Multiple file entries can point to the same inode (multiple opens of the same file).

10.5. The File Structure

```
/* file.h */
struct file {
    char  f_flag;       /* FREAD, FWRITE, FPIPE */
    char  f_count;      /* Reference count */
    int   f_inode;      /* Pointer to inode */
    char  *f_offset[2]; /* Current position (32-bit) */
} file[NFILE];

#define FREAD   01      /* Open for reading */
#define FWRITE  02      /* Open for writing */
#define FPIPE   04      /* This is a pipe */
```

The offset is 32 bits (two 16-bit words) to support files larger than 64KB. With `NFILE=100`, the system supports 100 simultaneous open files across all processes.

10.6. File Descriptor Operations

10.6.1. getf() — Validate File Descriptor

```
/* fio.c */
getf(f)
{
    register *fp, rf;

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

Converts a file descriptor (small integer) to a file pointer. Returns NULL and sets EBADF if invalid.

10.6.2. ufalloc() — Find Free Descriptor

```
/* fio.c */
ufalloc()
{
    register i;

    for (i=0; i<NOFILE; i++)
        if (u.u_ofile[i] == NULL) {
            u.u_ar0[R0] = i;        /* Return fd in r0 */
            return(i);
        }
    u.u_error = EMFILE;             /* Too many open files */
    return(-1);
}
```

Finds the lowest available file descriptor. Places the result in both the return value and `r0` (for system call return).

10.6.3. falloc() — Allocate File Entry

```
/* fio.c */
falloc()
{
    register struct file *fp;
    register i;

    if ((i = ufalloc()) < 0)
```

```
        return(NULL);
    for (fp = &file[0]; fp < &file[NFILE]; fp++)
        if (fp->f_count==0) {
            u.u_ofile[i] = fp;      /* Link descriptor to file */
            fp->f_count++;
            fp->f_offset[0] = 0;    /* Start at beginning */
            fp->f_offset[1] = 0;
            return(fp);
        }
    printf("no file\n");
    u.u_error = ENFILE;             /* File table full */
    return(NULL);
}
```

Allocates both a file descriptor and a file table entry, linking them together. The offset is initialized to zero.

10.6.4. closef() — Close File Entry

```
/* fio.c */
closef(fp)
int *fp;
{
    register *rfp, *ip;

    rfp = fp;
    if(rfp->f_flag&FPIPE) {
        ip = rfp->f_inode;
        ip->i_mode =& ~(IREAD|IWRITE);
        wakeup(ip+1);               /* Wake pipe readers */
        wakeup(ip+2);               /* Wake pipe writers */
    }
    if(rfp->f_count <= 1)
        closei(rfp->f_inode, rfp->f_flag&FWRITE);
    rfp->f_count--;
}
```

Decrements the reference count. When it reaches zero, the underlying inode is closed. Pipes get special handling to wake waiting processes.

10.6.5. closei() — Close Inode

```
/* fio.c */
closei(ip, rw)
int *ip;
{
    register *rip;
```

```
    register dev, maj;

    rip = ip;
    dev = rip->i_addr[0];
    maj = rip->i_addr[0].d_major;
    if(rip->i_count <= 1)
    switch(rip->i_mode&IFMT) {

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

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

For device files, calls the device's close routine. Then releases the inode with `iput()`.

10.6.6. openi() — Open Inode

```
/* fio.c */
openi(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:
        if(maj >= nchrdev)
            goto bad;
        (*cdevsw[maj].d_open)(dev, rw);
        break;

    case IFBLK:
        if(maj >= nblkdev) {
        bad:
            u.u_error = ENXIO;
            return;
        }
        (*bdevsw[maj].d_open)(dev, rw);
    }
}
```

For device files, calls the device's open routine. Regular files don't need any special open processing.

10.7. Permission Checking

10.7.1. access() — Check Permissions

```
/* fio.c */
access(ip, mode)
int *ip;
{
    register *rip, m;

    rip = ip;
    m = mode;
    if(m == IWRITE && getfs(ip->i_dev)->s_ronly != 0) {
        u.u_error = EROFS;          /* Read-only filesystem */
        return(1);
    }
    if(u.u_uid == 0)
        return(0);                  /* Root can do anything */
    if(u.u_uid != rip->i_uid) {
        m =>> 3;                    /* Check group bits */
        if(u.u_gid != rip->i_gid)
            m =>> 3;                /* Check other bits */
    }
    if((rip->i_mode&m) != 0)
        return(0);                  /* Permission granted */
    u.u_error = EACCES;
    return(1);
}
```

The classic UNIX permission algorithm:

1. Check for read-only filesystem (for writes)
2. Root (uid 0) bypasses all checks
3. Check owner bits, group bits, or other bits depending on identity
4. Return 0 for success, 1 for failure

10.7.2. owner() and suser()

```
/* fio.c */
owner(ip)
int *ip;
{
    if(u.u_uid == ip->i_uid)
        return(1);
    return(suser());
}

suser()
```

```
{
    if(u.u_uid == 0)
        return(1);
    u.u_error = EPERM;
    return(0);
}
```

owner() checks if the user owns the file or is root. suser() checks for root privileges.

10.8. Reading Files: readi()

The heart of file reading:

```
/* rdwri.c */
readi(aip)
struct inode *aip;
{
    int *bp;
    int lbn, bn, on;
    register dn, n;
    register struct inode *ip;

    ip = aip;
    if(u.u_count == 0)
        return;
    ip->i_flag =| IACC;             /* Mark access time update */
```

Parameters are passed through the user structure:

- u.u_base — User buffer address
- u.u_count — Bytes to read
- u.u_offset — File position
- u.u_segflg — 0 for user space, 1 for kernel space

```
    if((ip->i_mode&IFMT) == IFCHR) {
        (*cdevsw[ip->i_addr[0].d_major].d_read)(ip->i_addr[0]);
        return;
    }
```

Character devices go directly to their driver's read routine.

```
    do {
        lbn = bn = lshift(u.u_offset, -9);  /* Logical block number */
        on = u.u_offset[1] & 0777;          /* Offset within block */
        n = min(512-on, u.u_count);         /* Bytes this iteration */
```

The 32-bit offset is converted:

- `lbn` = offset / 512 (logical block number)
- on = offset % 512 (byte within block)
- n = bytes to transfer (at most to end of block)

```
        if((ip->i_mode&IFMT) != IFBLK) {
            dn = dpcmp(ip->i_size0, ip->i_size1,
                u.u_offset[0], u.u_offset[1]);
            if(dn <= 0)
                return;                     /* At or past EOF */
            n = min(n, dn);                 /* Don't read past EOF */
            if ((bn = bmap(ip, lbn)) == 0)
                return;                     /* Error in block mapping */
            dn = ip->i_dev;
        } else {
            dn = ip->i_addr[0];             /* Block device: use directly */
            rablock = bn+1;
        }
```

For regular files, check for EOF and call `bmap()` to translate logical to physical block. For block devices, the block number is used directly.

```
        if (ip->i_lastr+1 == lbn)
            bp = breada(dn, bn, rablock);   /* Read-ahead */
        else
            bp = bread(dn, bn);             /* Simple read */
        ip->i_lastr = lbn;
        iomove(bp, on, n, B_READ);
        brelse(bp);
    } while(u.u_error==0 && u.u_count!=0);
}
```

Read-ahead optimization: If reading sequentially (current block = last block + 1), use `breada()` to start fetching the next block while processing this one. `i_lastr` tracks the last block read.

10.9. Writing Files: writei()

```
/* rdwri.c */
writei(aip)
struct inode *aip;
{
    int *bp;
    int n, on;
    register dn, bn;
    register struct inode *ip;

    ip = aip;
    ip->i_flag =| IACC|IUPD;        /* Mark access and update times */
```

```
    if((ip->i_mode&IFMT) == IFCHR) {
        (*cdevsw[ip->i_addr[0].d_major].d_write)(ip->i_addr[0]);
        return;
    }
    if (u.u_count == 0)
        return;
```

Similar setup to `readi()`. Character devices go to their driver.

```
    do {
        bn = lshift(u.u_offset, -9);
        on = u.u_offset[1] & 0777;
        n = min(512-on, u.u_count);
        if((ip->i_mode&IFMT) != IFBLK) {
            if ((bn = bmap(ip, bn)) == 0)
                return;
            dn = ip->i_dev;
        } else
            dn = ip->i_addr[0];
```

Same block calculation as reading. `bmap()` will allocate new blocks if needed.

```
        if(n == 512)
            bp = getblk(dn, bn);    /* Full block: no need to read first */
        else
            bp = bread(dn, bn);     /* Partial: read existing content */
        iomove(bp, on, n, B_WRITE);
```

Optimization: If writing a full 512-byte block, there's no need to read the old contents first—just get an empty buffer.

```
        if(u.u_error != 0)
            brelse(bp);
        else if ((u.u_offset[1]&0777)==0)
            bawrite(bp);            /* Block boundary: async write */
        else
            bdwrite(bp);            /* Delayed write */
```

Write strategy:

- On error, just release the buffer
- At block boundary, use async write (`bawrite`)—starts the I/O but doesn't wait
- Mid-block, use delayed write (`bdwrite`)—buffer stays in cache until needed

```
        if(dpcmp(ip->i_size0, ip->i_size1,
          u.u_offset[0], u.u_offset[1]) < 0 &&
          (ip->i_mode&(IFBLK&IFCHR)) == 0) {
            ip->i_size0 = u.u_offset[0];
```

```
            ip->i_size1 = u.u_offset[1];
        }
        ip->i_flag =| IUPD;
    } while(u.u_error==0 && u.u_count!=0);
}
```

If the write extended the file (offset > size), update the file size.

10.10. Block Mapping: bmap()

bmap ( ) translates a logical block number to a physical disk block:

```
/* subr.c */
bmap(ip, bn)
struct inode *ip;
int bn;
{
    register *bp, *bap, nb;
    int *nbp, d, i;

    d = ip->i_dev;

    if (bn & ~03777) {
        u.u_error = EFBIG;          /* Block number too large */
        return(0);
    }
```

Maximum file size: 03777 (octal) = 2047 blocks = ~1MB.

10.10.1. Small File Algorithm

```
    if((ip->i_mode&ILARG) == 0) {

        /*
         * small file algorithm
         */

        if((bn & ~7) != 0) {
            /*
             * convert small to large
             */
            if ((bp = alloc(d)) == NULL)
                return(0);
            bap = bp->b_addr;
            for(i=0; i<8; i++) {
                *bap++ = ip->i_addr[i];
                ip->i_addr[i] = 0;
```

```
            }
            ip->i_addr[0] = bp->b_blkno;
            bdwrite(bp);
            ip->i_mode =| ILARG;
            goto large;
        }
```

For small files (ILARG not set), `i_addr[0-7]` are direct block pointers. This handles blocks 0-7.

If block 8+ is requested, the file must be converted to large format:

1. Allocate an indirect block
2. Copy the 8 direct pointers into it
3. Point `i_addr[0]` to the indirect block
4. Set ILARG flag

```
        nb = ip->i_addr[bn];
        if(nb == 0 && (bp = alloc(d)) != NULL) {
            bdwrite(bp);
            nb = bp->b_blkno;
            ip->i_addr[bn] = nb;
            ip->i_flag =| IUPD;
        }
        if (bn<7)
            rablock = ip->i_addr[bn+1];
        else
            rablock = 0;
        return(nb);
    }
```

For blocks 0-7: return the direct pointer, allocating if necessary. Set `rablock` for read-ahead.

10.10.2. Large File Algorithm

```
    /*
     * large file algorithm
     */

large:
    i = bn>>8;                      /* Which indirect block */
    if((nb=ip->i_addr[i]) == 0) {
        ip->i_flag =| IUPD;
        if ((bp = alloc(d)) == NULL)
            return(0);
        nb = bp->b_blkno;
        ip->i_addr[i] = nb;
    } else
        bp = bread(d, nb);
```

For large files, `i_addr[0-7]` point to indirect blocks. Each indirect block contains 256 block numbers (512 bytes / 2 bytes per pointer).

- `bn >> 8` = which indirect block (0-7)
- `bn & 0377` = index within that indirect block (0-255)

```
    bap = bp->b_addr;
    i = bn & 0377;
    if((nb=bap[i]) == 0 && (nbp = alloc(d)) != NULL) {
        nb = nbp->b_blkno;
        bap[i] = nb;
        bdwrite(nbp);
        bdwrite(bp);
    } else
        brelse(bp);
    rablock = bap[i+1];
    return(nb);
}
```

Look up the block in the indirect block, allocating if needed.

10.10.3. File Size Limits

```
Small file (ILARG=0):
  i_addr[0-7] → direct blocks
  Max: 8 blocks = 4KB

Large file (ILARG=1):
  i_addr[0-7] → indirect blocks
  Each indirect: 256 block numbers
  Max: 8 × 256 = 2048 blocks = 1MB
```

> **Why two algorithms?** Most UNIX files are small—configuration files, source code, shell scripts. The small file algorithm optimizes for this common case: direct block pointers mean zero extra disk reads to locate data. Large files are rare but must be supported, so the indirect scheme kicks in only when needed, adding just one extra disk read per access. This design keeps inodes compact (only 8 pointers) while still supporting files up to 1MB—a practical tradeoff for 1973 when a 2.4MB RK05 disk was the entire filesystem.

10.11. Data Transfer: iomove()

```
/* rdwri.c */
iomove(bp, o, an, flag)
struct buf *bp;
{
```

```
    register char *cp;
    register int n, t;

    n = an;
    cp = bp->b_addr + o;            /* Buffer address + offset */
```

`iomove()` copies data between a buffer and user space.

```
    if(u.u_segflg==0 && ((n | cp | u.u_base)&01)==0) {
        /* Fast path: user space, all aligned */
        if (flag==B_WRITE)
            cp = copyin(u.u_base, cp, n);
        else
            cp = copyout(cp, u.u_base, n);
        if (cp) {
            u.u_error = EFAULT;
            return;
        }
        u.u_base =+ n;
        dpadd(u.u_offset, n);
        u.u_count =- n;
        return;
    }
```

Fast path: When transferring to/from user space with aligned addresses, use `copyin`/`copyout` for efficient bulk transfer.

```
    if (flag==B_WRITE) {
        while(n--) {
            if ((t = cpass()) < 0)
                return;
            *cp++ = t;
        }
    } else
        while (n--)
            if(passc(*cp++) < 0)
                return;
}
```

Slow path: Byte-by-byte transfer using `cpass()` (get byte from user) and `passc()` (put byte to user). Used for unaligned transfers or kernel-space I/O.

10.11.1. passc() and cpass()

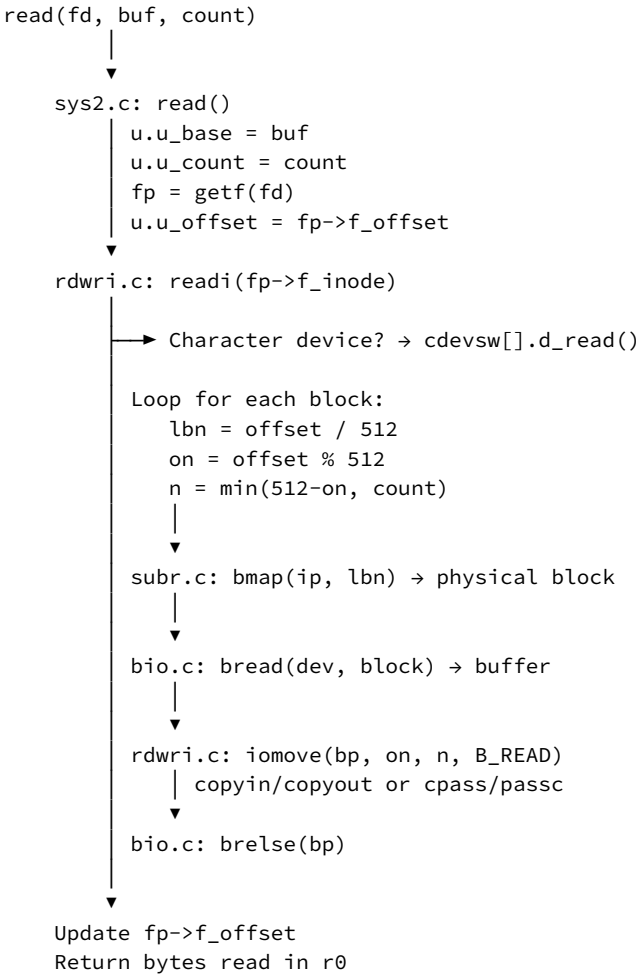
```
/* subr.c */
passc(c)
char c;
{
```

```
    if(u.u_segflg)
        *u.u_base = c;              /* Kernel space: direct */
    else
        if(subyte(u.u_base, c) < 0) {
            u.u_error = EFAULT;
            return(-1);
        }
    u.u_count--;
    if(++u.u_offset[1] == 0)
        u.u_offset[0]++;            /* Handle 32-bit overflow */
    u.u_base++;
    return(u.u_count == 0? -1: 0);
}

cpass()
{
    register c;

    if(u.u_count == 0)
        return(-1);
    if(u.u_segflg)
        c = *u.u_base;              /* Kernel space: direct */
    else
        if((c=fubyte(u.u_base)) < 0) {
            u.u_error = EFAULT;
            return(-1);
        }
    u.u_count--;
    if(++u.u_offset[1] == 0)
        u.u_offset[0]++;
    u.u_base++;
    return(c&0377);
}
```

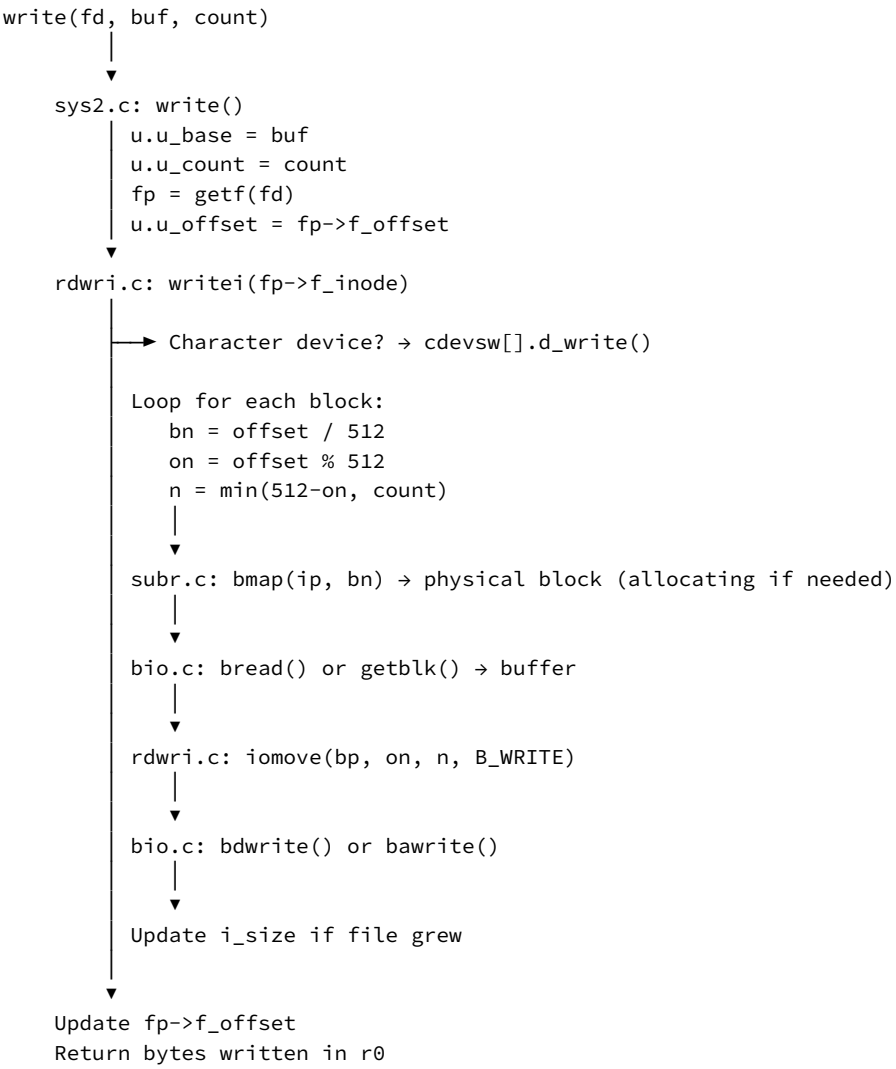
These handle the byte-by-byte case:

- `u.u_segflg=0`: User space, use `subyte`/`fubyte` (store/fetch user byte)
- `u.u_segflg=1`: Kernel space, direct memory access
- Update count, offset, and base pointer after each byte

10.12. The Complete Read Path



10.13. The Complete Write Path



10.14. Summary

- **Three levels**: file descriptors → open file table → inodes
- **readi()/writei()**: Loop over blocks, calling `bmap()` and `iomove()`
- **bmap()**: Translates logical to physical blocks, handles small/large files
- **iomove()**: Transfers data between buffers and user space
- **Read-ahead**: Sequential reads trigger prefetching of the next block
- **Write optimization**: Full blocks don't need to be read first

10.15. Key Design Points

1. **Separation of concerns**: File descriptors handle per-process state, file entries handle sharing, inodes handle storage
2. **Lazy allocation**: Blocks are allocated only when written, not when the file is created or extended
3. **Small file optimization**: Files ≤4KB use direct blocks, avoiding indirect block overhead
4. **Read-ahead**: Sequential access patterns are detected and optimized
5. **Delayed writes**: Data stays in cache, written later, improving performance

10.16. Experiments

1. **Trace bmap()**: Add printf to see small→large file conversion when writing block 8+.
2. **Measure read-ahead**: Compare sequential vs random read performance.
3. **Watch the file table**: Print `file[]` to see sharing after `fork()` or `dup()`.
4. **Fill the file table**: Open files until ENFILE, observe the limit.

10.17. Further Reading

- Chapter 9: Inodes and Superblock — Where `i_addr[]` comes from
- Chapter 11: Path Resolution — How files are found
- Chapter 12: Buffer Cache — The `bread`/`bwrite` layer beneath

Next: Chapter 11 — Path Resolution (namei)

11. Chapter 11: Path Resolution (namei)

11.1. Overview

Every file operation begins with a pathname: `/etc/passwd`, `../foo`, `file.txt`. The `namei()` function translates these human-readable paths into inode pointers the kernel can work with. It handles absolute and relative paths, traverses directories, checks permissions, crosses mount points, and supports three modes: lookup, create, and delete.

This single function is the gateway to the entire file system.

11.2. Source Files

File	Purpose
`usr/sys/ken/nami.c`	`namei()`, `schar()`, `uchar()`
`usr/sys/user.h`	Path resolution state in user structure

11.3. Prerequisites

- Chapter 9: Inodes and Superblock (`iget`/`iput`)
- Chapter 10: File I/O (directory reading)

11.4. Directory Structure

A directory is simply a file containing 16-byte entries:

```
struct {
    int  u_ino;         /* Inode number (2 bytes) */
    char u_name[DIRSIZ]; /* Filename (14 bytes) */
};
```

With `DIRSIZ=14`, filenames are limited to 14 characters. An inode number of 0 indicates an empty (deleted) slot.

Example directory contents:

```
Offset  Inode  Name
0       1      .
16      1      ..
32      47     passwd
48      0      (empty)
64      52     group
...
```

11.5. The User Structure Fields

`namei()` uses several fields in the user structure:

```c
/* user.h */
struct user {
    ...
    int   *u_cdir;              /* Current directory inode */
    char  u_dbuf[DIRSIZ];       /* Component being searched */
    char  *u_dirp;              /* Pointer into pathname */
    struct {
        int  u_ino;             /* Found entry's inode number */
        char u_name[DIRSIZ];    /* Found entry's name */
    } u_dent;
    int   *u_pdir;              /* Parent directory (for create) */
    ...
};
```

Field	Purpose
`u_cdir`	Current working directory inode
`u_dbuf`	Current path component being matched
`u_dirp`	Pointer to next character in pathname
`u_dent`	Last directory entry read
`u_pdir`	Parent directory (set during create mode)

11.6. namei() Function Signature

```
namei(func, flag)
int (*func)();
int flag;
```

func: Function to get the next pathname character - `&uchar` — pathname is in user space - `&schar` — pathname is in kernel space

flag: Operation mode - `0` — Lookup: find existing file - `1` — Create: find parent directory for new file - `2` — Delete: find file to be deleted

Returns: Locked inode pointer, or NULL on error

11.7. namei() Walkthrough

11.7.1. Initialization

```
{
    register struct inode *dp;
    register c;
    register char *cp;
    int eo, *bp;

    /*
     * start from indicated directory
     */
    dp = u.u_cdir;              /* Start at current directory */
    if((c=(*func)()) == '/')
        dp = rootdir;           /* Absolute path: start at root */
    iget(dp->i_dev, dp->i_number);
    while(c == '/')
        c = (*func)();          /* Skip leading/consecutive slashes */
```

The starting point depends on the first character:

- `/etc/passwd` → start at root
- `foo/bar` → start at current directory
- `///foo` → start at root (extra slashes ignored)

`iget()` is called to get a locked, reference-counted copy of the starting directory.

11.7.2. Main Loop (cloop)

```
cloop:
    /*
     * here dp contains pointer to last component matched.
     */

    if(u.u_error)
        goto out;
    if(c == '\0')
        return(dp);             /* Path exhausted: success! */
```

If we've consumed the entire path with no errors, return the current inode.

```
    /*
     * if there is another component,
     * dp must be a directory and must have x permission
     */

    if((dp->i_mode&IFMT) != IFDIR) {
        u.u_error = ENOTDIR;
        goto out;
    }

    if(access(dp, IEXEC))
        goto out;
```

To traverse into a directory, it must:

1. Actually be a directory (not a regular file)
2. Have execute permission (the "search" permission for directories)

11.7.3. Parsing the Component

```
    /*
     * gather up name into users' dir buffer
     */

    cp = &u.u_dbuf[0];
    while(c!='/' && c!='\0' && u.u_error==0) {
        if(cp < &u.u_dbuf[DIRSIZ])
            *cp++ = c;
        c = (*func)();
    }
    while(cp < &u.u_dbuf[DIRSIZ])
        *cp++ = '\0';           /* Pad with nulls */
    while(c == '/')
        c = (*func)();          /* Skip trailing slashes */
```

Extract one path component (e.g., “etc” from “/etc/passwd”) into `u_dbuf`. Components longer than 14 characters are silently truncated.

11.7.4. Directory Search Setup

```
    /*
     * search the directory
     */

    u.u_offset[1] = 0;
    u.u_offset[0] = 0;
    u.u_segflg = 1;             /* Reading to kernel space */
    eo = 0;                     /* First empty slot offset */
    u.u_count = ldiv(dp->i_size1, DIRSIZ+2);  /* Entry count */
    bp = NULL;
```

Prepare to scan the directory from the beginning. `u_count` is the number of 16-byte entries.

11.7.5. Directory Search Loop (eloop)

```
eloop:
    if(u.u_count == 0) {
        /* Searched entire directory without finding it */
        if(bp != NULL)
            brelse(bp);
        if(flag==1 && c=='\0') {
            /* Create mode: return parent for new file */
            if(access(dp, IWRITE))
                goto out;
            u.u_pdir = dp;
            if(eo)
                u.u_offset[1] = eo-DIRSIZ-2;  /* Use empty slot */
            else
                dp->i_flag =| IUPD;           /* Append to directory */
            return(NULL);
        }
        u.u_error = ENOENT;     /* File not found */
        goto out;
    }
```

When the search exhausts all entries:

- **Create mode** (`flag==1`) and at final component: Success! Return NULL with `u_pdir` pointing to parent directory. The offset indicates where to write the new entry.
- **Otherwise**: File not found, return ENOENT.

```
    if((u.u_offset[1]&0777) == 0) {
        /* Need to read next directory block */
        if(bp != NULL)
            brelse(bp);
        bp = bread(dp->i_dev,
            bmap(dp, ldiv(u.u_offset[1], 512)));
    }
```

Read directory blocks as needed. Each 512-byte block holds 32 entries.

```
    bcopy(bp->b_addr+(u.u_offset[1]&0777), &u.u_dent, (DIRSIZ+2)/2);
    u.u_offset[1] =+ DIRSIZ+2;
    u.u_count--;
```

Copy the current 16-byte entry into `u_dent` and advance.

```
    if(u.u_dent.u_ino == 0) {
        /* Empty slot - remember for create */
        if(eo == 0)
            eo = u.u_offset[1];
        goto eloop;
    }
```

Empty slots (inode 0) are skipped but remembered—`eo` records the first empty slot for potential reuse during create.

```
    for(cp = &u.u_dbuf[0]; cp < &u.u_dbuf[DIRSIZ]; cp++)
        if(*cp != cp[u.u_dent.u_name - u.u_dbuf])
            goto eloop;
```

Compare the entry's name with the component we're looking for. This is a character-by-character comparison.

11.7.6. Match Found

```
    if(bp != NULL)
        brelse(bp);
    if(flag==2 && c=='\0') {
        /* Delete mode: return current directory with entry info */
        if(access(dp, IWRITE))
            goto out;
        return(dp);
    }
```

For **delete mode** (`flag==2`) at the final component: return the *parent* directory with `u_dent` containing the entry to delete.

```
    bp = dp->i_dev;
    iput(dp);
    dp = iget(bp, u.u_dent.u_ino);
    if(dp == NULL)
        return(NULL);
    goto cloop;
```

For lookup or intermediate components: release the current directory, get the matched inode, and continue parsing.

11.7.7. Cleanup on Error

```
out:
    iput(dp);
    return(NULL);
}
```

On any error, release the current inode and return NULL.

11.8. Character Fetch Functions

11.8.1. uchar() — From User Space

```
uchar()
{
    register c;

    c = fubyte(u.u_dirp++);
    if(c == -1)
        u.u_error = EFAULT;
    return(c);
}
```

Fetches the next character from a user-space pathname using `fubyte()` (fetch user byte). Returns -1 on fault.

11.8.2. schar() — From Kernel Space

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

Fetches directly from kernel memory. Used when the pathname is already in kernel space (e.g., during `exec()` of `/etc/init`).

11.9. Usage Examples

11.9.1. open() — Lookup Mode

```
open()
{
    u.u_dirp = u.u_arg[0];      /* Pathname from user */
    ip = namei(&uchar, 0);      /* flag=0: lookup */
    if(ip == NULL)
        return;                 /* ENOENT already set */
    /* ip is the file's inode */
}
```

11.9.2. creat() — Create Mode

```
creat()
{
    u.u_dirp = u.u_arg[0];
    ip = namei(&uchar, 1);      /* flag=1: create */
    if(ip != NULL) {
        /* File exists - truncate it */
        ...
    } else if(u.u_error == 0) {
        /* File doesn't exist - create it */
        /* u.u_pdir = parent directory */
        /* u.u_offset = where to write entry */
        ip = maknode(mode);
    }
}
```

11.9.3. unlink() — Delete Mode

```
unlink()
{
    u.u_dirp = u.u_arg[0];
    ip = namei(&uchar, 2);      /* flag=2: delete */
    if(ip == NULL)
        return;
    /* ip = parent directory */
    /* u.u_dent = entry to delete */
    u.u_dent.u_ino = 0;         /* Clear the entry */
```

```
    writei(ip);                 /* Write back */
    ...
}
```

11.10. Path Resolution Examples

11.10.1. Absolute Path: `/etc/passwd`

```
namei(&uchar, 0) with u.u_dirp = "/etc/passwd"

1. c='/', dp = rootdir, iget root
2. c='e', parse "etc" into u.u_dbuf
3. Search root directory:
   - Entry ".": skip
   - Entry "..": skip
   - Entry "etc": match! inode=5
4. iput(root), dp = iget(5)
5. c='p', parse "passwd" into u.u_dbuf
6. Search /etc directory:
   - Entry ".": skip
   - Entry "..": skip
   - Entry "passwd": match! inode=47
7. c='\0', return inode 47
```

11.10.2. Relative Path: `../foo`

```
namei(&uchar, 0) with u.u_dirp = "../foo"

1. c='.', dp = u.u_cdir (say inode 10), iget(10)
2. parse ".." into u.u_dbuf
3. Search current directory:
   - Entry ".": skip
   - Entry "..": match! inode=3
4. iput(10), dp = iget(3)
5. parse "foo" into u.u_dbuf
6. Search parent directory:
   - Entry "foo": match! inode=25
7. c='\0', return inode 25
```

11.10.3. Create: `/tmp/newfile`

```
namei(&uchar, 1) with u.u_dirp = "/tmp/newfile"

1. c='/', dp = rootdir
2. parse "tmp", search root, find inode 4
3. dp = iget(4)
4. parse "newfile", search /tmp:
   - Not found!
5. flag==1 and c=='\0':
   - Check write permission on /tmp
```

```
   - u.u_pdir = dp (inode 4)
   - u.u_offset = position for new entry
   - return NULL (success!)
```

11.11. Mount Point Traversal

Note that mount point handling actually happens in `iget()` (Chapter 9), not `namei()`. When `iget()` finds an inode marked `IMOUNT`, it automatically redirects to the mounted filesystem's root.

```
/usr/include/stdio.h
  where /usr is a mount point

namei traverses:

1. / (root)
2. usr → iget() sees IMOUNT, redirects to mounted fs root
3. include
4. stdio.h
```

11.12. Error Handling

Error	Condition
`ENOENT`	Component not found (lookup/delete mode)
`ENOTDIR`	Intermediate component is not a directory
`EACCES`	No search (x) permission on directory
`EFAULT`	Bad user-space pathname pointer

11.13. The "." and ".." Entries

Every directory contains two special entries:

```
.   → inode of the directory itself
..  → inode of the parent directory
```

For the root directory, `..` points to itself. These entries are created by `mkdir` and are traversed by `namei()` just like any other entries.

11.14. Summary

- `namei()` translates pathnames to inodes

- Three modes: lookup (0), create (1), delete (2)
- Parses components left-to-right, searching each directory
- Checks execute permission at each directory
- Uses `u_dbuf` for current component, `u_dent` for matched entry
- Returns locked inode (or NULL with `u_pdir` set for create)

11.15. Key Design Points

1. **Single function**: One function handles all path resolution needs through the flag parameter.
2. **No recursion**: Iterative loop avoids stack overflow on deep paths.
3. **Flexible input**: Function pointer allows user-space or kernel-space pathnames.
4. **Empty slot reuse**: Tracks first empty slot during search for efficient create.
5. **Atomic operation**: Inode locking prevents races during create/delete.

11.16. Experiments

1. **Trace path resolution**: Add printf showing each component and matched inode.
2. **Deep paths**: Create deeply nested directories and observe behavior.
3. **Permission denied**: Remove execute permission from a directory and try to traverse it.
4. **Long names**: Try creating files with names longer than 14 characters.

11.17. Further Reading

- Chapter 9: Inodes and Superblock — `iget()` and mount point handling
- Chapter 10: File I/O — How directories are read
- System calls that use `namei()`: `open`, `creat`, `stat`, `unlink`, `link`, `chdir`, `chmod`, `chown`

Next: Chapter 12 — The Buffer Cache

12. Chapter 12: The Buffer Cache

12.1. Overview

The buffer cache is the kernel's interface to block devices. Every disk read and write passes through this layer, which maintains an in-memory cache of recently-used disk blocks. The cache dramatically improves performance by avoiding redundant disk I/O and provides a uniform interface that hides device-specific details.

This is Dennis Ritchie's code (`usr/sys/dmr/bio.c`)—elegant, compact, and the foundation upon which all file operations rest.

12.2. Source Files

File	Purpose
`usr/sys/buf.h`	Buffer structure and flags
`usr/sys/dmr/bio.c`	Buffer cache implementation

12.3. Prerequisites

- Chapter 9: Inodes and Superblock (uses `bread`/`bwrite`)
- Chapter 10: File I/O (uses buffer cache for all I/O)

12.4. The Buffer Structure

```
/* buf.h */
struct buf {
    int   b_flags;      /* Status flags */
    struct buf *b_forw; /* Hash chain forward */
    struct buf *b_back; /* Hash chain backward */
```

```
    struct buf *av_forw; /* Free list forward */
    struct buf *av_back; /* Free list backward */
    int   b_dev;        /* Device number */
    int   b_wcount;     /* Word count for transfer */
    char  *b_addr;      /* Buffer memory address */
    char  *b_blkno;     /* Block number on device */
    char  b_error;      /* Error code */
    char  *b_resid;     /* Residual count after I/O */
} buf[NBUF];
```

Each buffer has two sets of links:

- `b_forw`/`b_back` — Hash chain (by device + block number)
- `av_forw`/`av_back` — Free list (LRU order)

With `NBUF=15`, the system has 15 buffers, each 514 bytes (512 data + 2 for word count).

12.4.1. Buffer Flags

```
#define B_WRITE   0       /* Write operation */
#define B_READ    01      /* Read operation */
#define B_DONE    02      /* I/O complete */
#define B_ERROR   04      /* Error occurred */
#define B_BUSY    010     /* Buffer in use */
#define B_XMEM    060     /* Extended memory bits */
#define B_WANTED  0100    /* Process waiting for buffer */
#define B_RELOC   0200    /* Relocatable buffer */
#define B_ASYNC   0400    /* Asynchronous I/O */
#define B_DELWRI  01000   /* Delayed write pending */
```

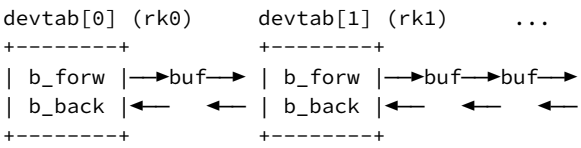
Key flags:

- B_BUSY — Buffer is allocated to someone
- B_DONE — I/O has completed (data is valid)
- B_DELWRI — Buffer has been written but not yet flushed to disk
- B_ASYNC — Don't wait for I/O completion

12.5. Buffer Lists

12.5.1. The Hash Chains

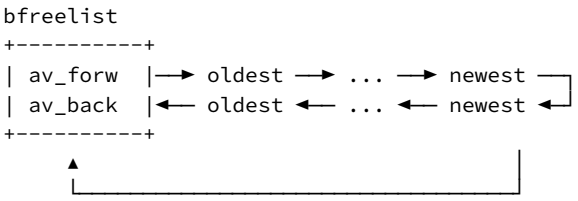
Buffers are organized by device into hash chains:



Each device has a `devtab` structure serving as the list head. To find a cached block, search the appropriate device's chain.

12.5.2. The Free List

Available buffers form a doubly-linked LRU list:



Buffers are taken from the front (oldest) and returned to the back (newest). This implements LRU replacement.

12.6. binit() — Initialization

```
/* bio.c */
binit()
{
    register struct buf *bp;
    register struct devtab *dp;
    register int i;
    struct bdevsw *bdp;

    bfreelist.b_forw = bfreelist.b_back =
        bfreelist.av_forw = bfreelist.av_back = &bfreelist;
```

Initialize the free list as an empty circular list.

```
    for (i=0; i<NBUF; i++) {
        bp = &buf[i];
        bp->b_dev = -1;
        bp->b_addr = buffers[i];
        bp->b_back = &bfreelist;
        bp->b_forw = bfreelist.b_forw;
        bfreelist.b_forw->b_back = bp;
        bfreelist.b_forw = bp;
        bp->b_flags = B_BUSY;
```

```
        brelse(bp);
    }
```

Link each buffer into the free list and assign its data area from the `buffers` array.

```
    i = 0;
    for (bdp = bdevsw; bdp->d_open; bdp++) {
        dp = bdp->d_tab;
        dp->b_forw = dp;
        dp->b_back = dp;
        i++;
    }
    nblkdev = i;
}
```

Initialize each device's hash chain as empty.

12.7. getblk() — Get a Buffer

The heart of the buffer cache:

```
/* bio.c */
getblk(dev, blkno)
{
    register struct buf *bp;
    register struct devtab *dp;

    if(dev.d_major >= nblkdev)
        panic("blkdev");

loop:
    if (dev < 0)
        dp = &bfreelist;        /* NODEV: just get any buffer */
    else {
        dp = bdevsw[dev.d_major].d_tab;
        for (bp=dp->b_forw; bp != dp; bp = bp->b_forw) {
            if (bp->b_blkno!=blkno || bp->b_dev!=dev)
                continue;
```

Search the device's hash chain for the requested block.

```
            spl6();             /* Disable interrupts */
            if (bp->b_flags&B_BUSY) {
                bp->b_flags =| B_WANTED;
                sleep(bp, PRIBIO);
                spl0();
                goto loop;      /* Retry after wakeup */
            }
```

```
            spl0();
            notavail(bp);       /* Remove from free list */
            return(bp);
        }
    }
```

If found but busy, wait for it. If found and free, remove from free list and return.

```
    spl6();
    if (bfreelist.av_forw == &bfreelist) {
        bfreelist.b_flags =| B_WANTED;
        sleep(&bfreelist, PRIBIO);
        spl0();
        goto loop;              /* Retry when buffer freed */
    }
    spl0();
    notavail(bp = bfreelist.av_forw);  /* Take oldest buffer */
```

If not in cache, take the oldest buffer from the free list. If the free list is empty, wait.

```
    if (bp->b_flags & B_DELWRI) {
        bp->b_flags =| B_ASYNC;
        bwrite(bp);             /* Flush dirty buffer */
        goto loop;              /* Retry with different buffer */
    }
```

If the victim buffer has pending writes, flush it first and try again.

```
    bp->b_flags = B_BUSY | B_RELOC;
    bp->b_back->b_forw = bp->b_forw;   /* Remove from old hash chain */
    bp->b_forw->b_back = bp->b_back;
    bp->b_forw = dp->b_forw;           /* Insert in new hash chain */
    bp->b_back = dp;
    dp->b_forw->b_back = bp;
    dp->b_forw = bp;
    bp->b_dev = dev;
    bp->b_blkno = blkno;
    return(bp);
}
```

Move the buffer from its old hash chain to the new one, update device and block number.

12.8. bread() — Read a Block

```
/* bio.c */
bread(dev, blkno)
{
    register struct buf *rbp;

    rbp = getblk(dev, blkno);
    if (rbp->b_flags&B_DONE)
        return(rbp);            /* Already in cache! */
    rbp->b_flags =| B_READ;
    rbp->b_wcount = -256;       /* 256 words = 512 bytes */
    (*bdevsw[dev.d_major].d_strategy)(rbp);
    iowait(rbp);
    return(rbp);
}
```

1. Get a buffer for the block
2. If B_DONE is set, data is already valid—cache hit!
3. Otherwise, initiate a read through the device's strategy routine
4. Wait for completion

The strategy routine is device-specific (e.g., `rkstrategy` for the RK05 disk).

12.9. breada() — Read with Read-Ahead

```
/* bio.c */
breada(adev, blkno, rablkno)
{
    register struct buf *rbp, *rabp;
    register int dev;

    dev = adev;
    rbp = 0;
    if (!incore(dev, blkno)) {
        rbp = getblk(dev, blkno);
        if ((rbp->b_flags&B_DONE) == 0) {
            rbp->b_flags =| B_READ;
            rbp->b_wcount = -256;
            (*bdevsw[adev.d_major].d_strategy)(rbp);
        }
    }
```

If the requested block isn't cached, start reading it.

```
    if (rablkno && !incore(dev, rablkno) && raflag) {
        rabp = getblk(dev, rablkno);
        if (rabp->b_flags & B_DONE)
            brelse(rabp);
        else {
```

```
            rabp->b_flags =| B_READ|B_ASYNC;
            rabp->b_wcount = -256;
            (*bdevsw[adev.d_major].d_strategy)(rabp);
        }
    }
```

If a read-ahead block is specified and not cached, start reading it **asynchronously** (B_ASYNC). This means we don't wait for it—it will be ready when needed.

```
    if (rbp==0)
        return(bread(dev, blkno));  /* Was already cached */
    iowait(rbp);
    return(rbp);
}
```

Wait for the primary block and return it. The read-ahead block completes in the background.

12.10. incore() — Check Cache

```
/* bio.c */
incore(adev, blkno)
{
    register int dev;
    register struct buf *bp;
    register struct devtab *dp;

    dev = adev;
    dp = bdevsw[adev.d_major].d_tab;
    for (bp=dp->b_forw; bp != dp; bp = bp->b_forw)
        if (bp->b_blkno==blkno && bp->b_dev==dev)
            return(bp);
    return(0);
}
```

Returns the buffer if the block is in cache, NULL otherwise. Used by `breada()` to avoid redundant I/O.

12.11. Write Operations

12.11.1. bwrite() — Synchronous Write

```
/* bio.c */
bwrite(bp)
struct buf *bp;
```

```
{
    register struct buf *rbp;
    register flag;

    rbp = bp;
    flag = rbp->b_flags;
    rbp->b_flags =& ~(B_READ | B_DONE | B_ERROR | B_DELWRI);
    rbp->b_wcount = -256;
    (*bdevsw[rbp->b_dev.d_major].d_strategy)(rbp);
    if ((flag&B_ASYNC) == 0) {
        iowait(rbp);
        brelse(rbp);
    } else if ((flag&B_DELWRI)==0)
        geterror(rbp);
}
```

Start the write operation. If not async, wait for completion and release the buffer.

12.11.2. bdwrite() — Delayed Write

```
/* bio.c */
bdwrite(bp)
struct buf *bp;
{
    register struct buf *rbp;

    rbp = bp;
    if (bdevsw[rbp->b_dev.d_major].d_tab == &tmtab)
        bawrite(rbp);           /* Magtape: no delay */
    else {
        rbp->b_flags =| B_DELWRI | B_DONE;
        brelse(rbp);
    }
}
```

Mark the buffer as dirty (`B_DELWRI`) and release it. The actual write happens later, when:

- The buffer is reclaimed by `getblk()`
- `bflush()` is called (by `sync`)

This batches writes for efficiency.

12.11.3. bawrite() — Asynchronous Write

```
/* bio.c */
bawrite(bp)
struct buf *bp;
```

```
{
    register struct buf *rbp;

    rbp = bp;
    rbp->b_flags =| B_ASYNC;
    bwrite(rbp);
}
```

Start the write but don't wait. The buffer is released when I/O completes (in `iodone()`).

12.12. brelse() — Release Buffer

```
/* bio.c */
brelse(bp)
struct buf *bp;
{
    register struct buf *rbp, **backp;
    register int sps;

    rbp = bp;
    if (rbp->b_flags&B_WANTED)
        wakeup(rbp);            /* Wake waiters */
    if (bfreelist.b_flags&B_WANTED) {
        bfreelist.b_flags =& ~B_WANTED;
        wakeup(&bfreelist);     /* Wake buffer-starved processes */
    }
    if (rbp->b_flags&B_ERROR)
        rbp->b_dev.d_minor = -1;  /* Disassociate on error */
```

Wake any processes waiting for this buffer or for any free buffer.

```
    backp = &bfreelist.av_back;
    sps = PS->int;
    spl6();
    rbp->b_flags =& ~(B_WANTED|B_BUSY|B_ASYNC);
    (*backp)->av_forw = rbp;
    rbp->av_back = *backp;
    *backp = rbp;
    rbp->av_forw = &bfreelist;
    PS->int = sps;
}
```

Insert at the back of the free list (newest). The buffer remains on its hash chain—if the same block is needed again soon, it can be found instantly.

12.13. I/O Completion

12.13.1. iowait() — Wait for I/O

```
/* bio.c */
iowait(bp)
struct buf *bp;
{
    register struct buf *rbp;

    rbp = bp;
    spl6();
    while ((rbp->b_flags&B_DONE)==0)
        sleep(rbp, PRIBIO);
    spl0();
    geterror(rbp);
}
```

Sleep until the device sets B_DONE, then check for errors.

12.13.2. iodone() — I/O Complete (Interrupt Handler)

```
/* bio.c */
iodone(bp)
struct buf *bp;
{
    register struct buf *rbp;

    rbp = bp;
    rbp->b_flags =| B_DONE;
    if (rbp->b_flags&B_ASYNC)
        brelse(rbp);
    else {
        rbp->b_flags =& ~B_WANTED;
        wakeup(rbp);
    }
}
```

Called from device interrupt handlers. Sets B_DONE and either releases the buffer (async) or wakes waiting processes.

12.14. notavail() — Remove from Free List

```
/* bio.c */
notavail(bp)
struct buf *bp;
{
    register struct buf *rbp;
    register int sps;

    rbp = bp;
    sps = PS->int;
    spl6();
    rbp->av_back->av_forw = rbp->av_forw;
    rbp->av_forw->av_back = rbp->av_back;
    rbp->b_flags =| B_BUSY;
    PS->int = sps;
}
```

Unlinks a buffer from the free list and marks it busy. The buffer stays on its hash chain.

12.15. bflush() — Flush Dirty Buffers

```
/* bio.c */
bflush(dev)
{
    register struct buf *bp;

loop:
    spl6();
    for (bp = bfreelist.av_forw; bp != &bfreelist; bp = bp->av_forw) {
        if (bp->b_flags&B_DELWRI && (dev == NODEV||dev==bp->b_dev)) {
            bp->b_flags =| B_ASYNC;
            notavail(bp);
            bwrite(bp);
            goto loop;
        }
    }
    spl0();
}
```

Write all dirty buffers for a device (or all devices if NODEV). Called by `sync` system call and `update()`.

12.16. swap() — Swap I/O

```
/* bio.c */
swap(blkno, coreaddr, count, rdflg)
{
```

```
    register int *fp;

    fp = &swbuf.b_flags;
    spl6();
    while (*fp&B_BUSY) {
        *fp =| B_WANTED;
        sleep(fp, PSWP);
    }
    *fp = B_BUSY | rdflg | (coreaddr>>6)&B_XMEM;
    swbuf.b_dev = swapdev;
    swbuf.b_wcount = - (count<<5);
    swbuf.b_blkno = blkno;
    swbuf.b_addr = coreaddr<<6;
    (*bdevsw[swapdev>>8].d_strategy)(&swbuf);
    spl6();
    while((*fp&B_DONE)==0)
        sleep(fp, PSWP);
    if (*fp&B_WANTED)
        wakeup(fp);
    spl0();
    *fp =& ~(B_BUSY|B_WANTED);
    return(*fp&B_ERROR);
}
```

Special buffer (`swbuf`) for swap I/O. Bypasses the normal cache since swapped pages don't need caching—they're only read once when needed.

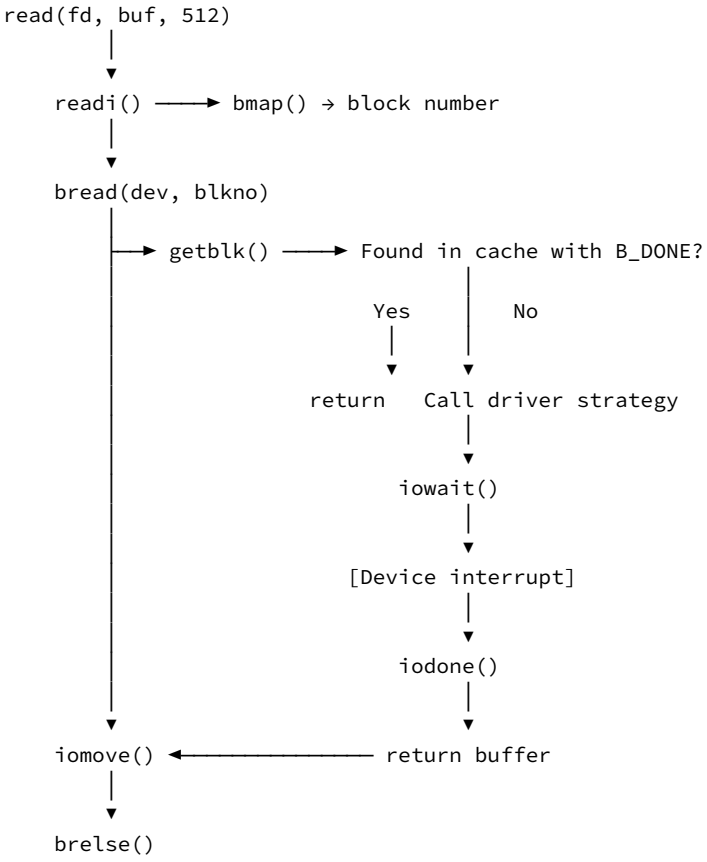
12.17. The Device Strategy Interface

The buffer cache calls device drivers through the strategy routine:

```
(*bdevsw[dev.d_major].d_strategy)(bp);
```

The driver queues the request, starts the device if idle, and returns immediately. When I/O completes, the device interrupt handler calls `iodone(bp)`.

12.18. Complete Read Flow



12.19. Summary

- **15 buffers** cache recent disk blocks
- **Hash chains** enable fast lookup by device + block
- **Free list** implements LRU replacement
- **Delayed write** batches writes for efficiency
- **Read-ahead** prefetches sequential blocks
- **Strategy interface** abstracts device differences

12.20. Key Design Points

1. **Unified interface**: All block I/O goes through `bread`/`bwrite`, hiding device complexity.
2. **Double linking**: Each buffer is on both a hash chain (for lookup) and the free list (for allocation).
3. **Lazy writes**: `bdwrite()` defers writing, improving performance for multiple writes to the same block.

4. **Asynchronous I/O**: B_ASYNC allows the CPU to continue while I/O proceeds.
5. **LRU replacement**: Most recently used blocks stay cached longest.
6. **Cache coherence**: A block can only be in one buffer—no stale copies.

12.21. Experiments

1. **Cache hit rate**: Count B_DONE hits in `bread()` vs disk reads.
2. **Buffer starvation**: Reduce NBUF and observe performance degradation.
3. **Delayed write timing**: Track how long B_DELWRI buffers stay dirty before flush.
4. **Read-ahead effectiveness**: Compare performance with `raflag=0` vs `raflag=1`.

12.22. Further Reading

- Chapter 14: Block Devices — Device drivers and strategy routines
- Chapter 9: Inodes and Superblock — How the file system uses the cache
- Chapter 10: File I/O — The higher-level I/O functions

Part III Complete! Next: Part IV — Device Drivers

Part IV.

Device Drivers

13. Chapter 13: The TTY Subsystem

13.1. Overview

In 1973, users interacted with UNIX through **teletypes**—electromechanical terminals that typed characters on paper. The TTY subsystem handles all this terminal I/O: echoing characters, processing backspace and line-kill, converting between uppercase and lowercase, expanding tabs, generating signals for interrupt and quit, and managing output flow control.

This is surprisingly complex code because it must handle the mismatch between human typing speeds and computer processing, while providing a pleasant interactive experience.

13.2. Source Files

File	Purpose
`usr/sys/tty.h`	TTY structure and constants
`usr/sys/dmr/tty.c`	Common TTY routines
`usr/sys/dmr/kl.c`	KL-11 console driver

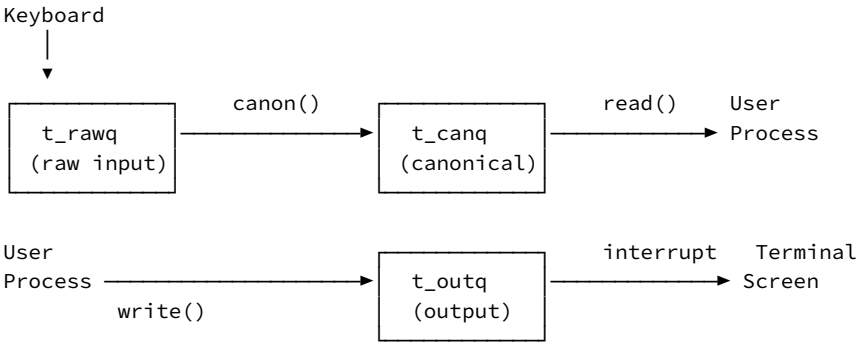
13.3. Prerequisites

- Chapter 10: File I/O (`passc`, `cpass`)
- Character device interface basics

13.4. The TTY Structure

```
/* tty.h */
struct tty {
    struct clist t_rawq;    /* Raw input queue */
    struct clist t_canq;    /* Canonical input queue */
    struct clist t_outq;    /* Output queue */
    int    t_flags;         /* Mode flags */
    int    *t_addr;         /* Device register address */
    char   t_delct;         /* Delimiter count (completed lines) */
    char   t_col;           /* Current column position */
    char   t_intrup;        /* Interrupt character */
    char   t_quit;          /* Quit character */
    char   t_state;         /* State flags */
    char   t_char;          /* (unused) */
    int    t_speeds;        /* Baud rate encoding */
};
```

13.4.1. The Three Queues



t_rawq (raw queue): Characters as they arrive from the keyboard, before any editing.

t_canq (canonical queue): Completed, edited lines ready for the application.

t_outq (output queue): Characters waiting to be displayed.

> **What is canonicalization?** The `canon()` function transforms raw input into "canonical" form—a complete, edited line. This means processing backspace (erase the previous character), kill (erase the entire line), and recognizing end-of-line (newline or EOT). The term comes from "canonical form" in mathematics: a standard, simplified representation. In UNIX, canonical input is line-buffered, edited text ready for applications that expect to read complete lines, as opposed to raw input which is character-by-character with no editing.

13.4.2. Character Lists (clist)

```
struct clist {
    int c_cc;       /* Character count */
    int c_cf;       /* First cblock pointer */
    int c_cl;       /* Last cblock pointer */
};

struct cblock {
    struct cblock *c_next;
    char info[6];   /* 6 characters per block */
};
```

Characters are stored in linked lists of 6-character blocks. This allows queues to grow and shrink dynamically without large fixed buffers.

13.5. Mode Flags

```
/* tty.h */
#define RAW     040     /* No processing, raw I/O */
#define ECHO    010     /* Echo input characters */
#define LCASE   04      /* Map uppercase to lowercase */
#define CRMOD   020     /* Map CR to NL on input, NL to CR+NL on output */
#define XTABS   02      /* Expand tabs to spaces */
#define NODELAY 01      /* No output delays */
```

RAW mode: Characters pass through unprocessed—no line editing, no signals, no echo. Used by screen editors and games.

Cooked mode (default): Full line editing with backspace (#) and line-kill (@), signal generation, and echo.

13.6. State Flags

```
#define ISOPEN   04     /* Device is open */
#define WOPEN    02     /* Waiting for open (modem) */
#define CARR_ON  020    /* Carrier present (modem connected) */
#define BUSY     040    /* Output in progress */
#define TIMEOUT  01     /* Delay in progress */
#define SSTART   010    /* Use special start routine */
```

13.7. Special Characters

```
#define CERASE  '#'     /* Erase one character */
#define CKILL   '@'     /* Kill entire line */
#define CEOT    004     /* End of file (Ctrl-D) */
```

The quit character (`Ctrl-\`) and interrupt character (DEL) are stored in the tty structure and can be changed per-terminal.

13.8. cinit() — Initialize Character Lists

```
/* tty.c */
cinit()
{
    register int ccp;
    register struct cblock *cp;
    register struct cdevsw *cdp;

    ccp = cfree;
    for (cp=(ccp+07)&~07; cp <= &cfree[NCLIST-1]; cp++) {
        cp->c_next = cfreelist;
        cfreelist = cp;
    }
    ccp = 0;
    for(cdp = cdevsw; cdp->d_open; cdp++)
        ccp++;
    nchrdev = ccp;
}
```

Links all cblocks into the free list and counts character devices.

13.9. Input Path

13.9.1. ttyinput() — Receive a Character

Called from the device interrupt handler when a character arrives:

```
/* tty.c */
ttyinput(ac, atp)
struct tty *atp;
{
    register int t_flags, c;
    register struct tty *tp;
```

```
    tp = atp;
    c = ac;
    t_flags = tp->t_flags;
```

```
    if ((c =& 0177) == '\r' && t_flags&CRMOD)
        c = '\n';
```

Strip to 7 bits. If CRMOD is set, convert carriage return to newline.

```
    if ((t_flags&RAW)==0 && (c==tp->t_quit || c==tp->t_intrup)) {
        signal(tp, c==tp->t_intrup? SIGINT:SIGQIT);
        flushtty(tp);
        return;
    }
```

In cooked mode, check for interrupt (DEL) or quit (`Ctrl-\`). Send the appropriate signal and flush all queues.

```
    if (tp->t_rawq.c_cc>=TTYHOG) {
        flushtty(tp);
        return;
    }
```

Prevent buffer overflow—if raw queue exceeds TTYHOG (256), flush everything.

```
    if (t_flags&LCASE && c>='A' && c<='Z')
        c =+ 'a'-'A';
    putc(c, &tp->t_rawq);
```

Convert uppercase to lowercase if LCASE mode. Add character to raw queue.

```
    if (t_flags&RAW || c=='\n' || c==004) {
        wakeup(&tp->t_rawq);
        if (putc(0377, &tp->t_rawq)==0)
            tp->t_delct++;
    }
```

In RAW mode, or when a line delimiter arrives (newline or Ctrl-D), wake up any process waiting for input. The 0377 character marks the end of a line; `t_delct` counts complete lines.

```
    if (t_flags&ECHO) {
        ttyoutput(c, tp);
        ttstart(tp);
    }
}
```

If ECHO is enabled, send the character to output.

13.9.2. canon() — Canonicalize Input

Converts raw input to edited, canonical form:

```
/* tty.c */
canon(atp)
struct tty *atp;
{
    register char *bp;
    char *bp1;
    register struct tty *tp;
    register int c;

    tp = atp;
    spl5();
    while (tp->t_delct==0) {
        if ((tp->t_state&CARR_ON)==0)
            return(0);          /* Carrier lost */
        sleep(&tp->t_rawq, TTIPRI);
    }
    spl0();
```

Wait until at least one complete line is available (`t_delct > 0`).

```
loop:
    bp = &canonb[2];
    while ((c=getc(&tp->t_rawq)) >= 0) {
        if (c==0377) {
            tp->t_delct--;
            break;              /* End of line */
        }
```

Read characters from raw queue until the delimiter (0377).

```
        if ((tp->t_flags&RAW)==0) {
            if (bp[-1]!='\\') {
                if (c==CERASE) {
                    if (bp > &canonb[2])
                        bp--;
                    continue;
                }
                if (c==CKILL)
                    goto loop;  /* Start over */
                if (c==CEOT)
                    continue;   /* Ignore Ctrl-D itself */
            }
```

> **Why octal?** The delimiter 0377 is 255 decimal (0xFF hex). PDP-11 programmers used octal because the machine's architecture favored it: registers are numbered 0-7 (3 bits), instruction fields are 3 bits wide, and the front panel switches were grouped in threes. You could read `MOV R1,R2` directly from its encoding `010102`. DEC's earlier 12-bit machines (PDP-8) divided perfectly into 4 octal digits, establishing the convention. Hexadecimal dominance came later with 8-bit byte-oriented architectures.

Process editing characters:

- # (CERASE): Back up one character
- @ (CKILL): Discard entire line, start over
- `Ctrl-D` (CEOT): Mark end of file but don't include in output

```
            } else
            if (maptab[c] && (maptab[c]==c || (tp->t_flags&LCASE))) {
                if (bp[-2] != '\\')
                    c = maptab[c];
                bp--;
            }
        }
        *bp++ = c;
        if (bp>=canonb+CANBSIZ)
            break;
    }
```

Handle escape sequences: `\{` becomes `{`, `\|` becomes `|`, etc. This allows typing special characters on terminals that lack them.

```
    bp1 = bp;
    bp = &canonb[2];
    c = &tp->t_canq;
    while (bp<bp1)
        putc(*bp++, c);
    return(1);
}
```

Copy the edited line to the canonical queue.

13.9.3. ttread() — Read from TTY

```
/* tty.c */
ttread(atp)
struct tty *atp;
{
    register struct tty *tp;
```

```
    tp = atp;
    if (tp->t_canq.c_cc || canon(tp))
        while (tp->t_canq.c_cc && passc(getc(&tp->t_canq))>=0);
}
```

If the canonical queue has characters, or `canon()` can produce some, transfer them to the user's buffer via `passc()`.

13.10. Output Path

13.10.1. ttyoutput() — Process Output Character

```
/* tty.c */
ttyoutput(ac, tp)
struct tty *tp;
{
    register int c;
    register struct tty *rtp;
    register char *colp;
    int ctype;

    rtp = tp;
    c = ac&0177;
```

```
    if (c==004 && (rtp->t_flags&RAW)==0)
        return;                 /* Suppress Ctrl-D in cooked mode */
```

```
    if (c=='\t' && rtp->t_flags&XTABS) {
        do
            ttyoutput(' ', rtp);
        while (rtp->t_col&07);
        return;
    }
```

Expand tabs to spaces if XTABS is set.

```
    if (rtp->t_flags&LCASE) {
        switch (c) {
        case '{':
            c = '(';
            goto esc;
        case '}':
            c = ')';
            goto esc;
        case '|':
```

```
            c = '!';
            goto esc;
        case '~':
            c = '^';
            goto esc;
        case '`':
            c = '\'';
        esc:
            ttyoutput('\\', rtp);
        }
        if ('a'<=c && c<='z')
            c =+ 'A' - 'a';     /* Convert to uppercase */
    }
```

For uppercase-only terminals (LCASE), convert lowercase to uppercase and escape special characters.

```
    if (c=='\n' && rtp->t_flags&CRMOD)
        ttyoutput('\r', rtp);
    if (putc(c, &rtp->t_outq))
        return;
```

Add carriage return before newline if CRMOD. Put character on output queue.

```
    colp = &rtp->t_col;
    ctype = partab[c];
    c = 0;
    switch (ctype&077) {

    case 0:                     /* ordinary */
        (*colp)++;
        break;

    case 1:                     /* non-printing */
        break;

    case 2:                     /* backspace */
        if (*colp)
            (*colp)--;
        break;

    case 3:                     /* newline */
        if (*colp)
            c = max((*colp>>4) + 3, 6);
        *colp = 0;
        break;

    case 4:                     /* tab */
        *colp =| 07;
        (*colp)++;
        break;

    case 6:                     /* carriage return */
        c = 6;
```

```
        *colp = 0;
    }
    if (c && (rtp->t_flags&NODELAY)==0)
        putc(c|0200, &rtp->t_outq);
}
```

Track column position and insert delays. Mechanical terminals need time for:

- Carriage return (6 character times)
- Newline (depends on column position)
- Tab (depends on position)

Delay characters have bit 0200 set.

13.10.2. ttwrite() — Write to TTY

```
/* tty.c */
ttwrite(atp)
struct tty *atp;
{
    register struct tty *tp;
    register int c;

    tp = atp;
    while ((c=cpass())>=0) {
        spl5();
        while (tp->t_outq.c_cc > TTHIWAT) {
            ttstart(tp);
            sleep(&tp->t_outq, TTOPRI);
        }
        spl0();
        ttyoutput(c, tp);
    }
    ttstart(tp);
}
```

Get characters from user space via `cpass()`. If the output queue exceeds TTHIWAT (50 chars), sleep until it drains to TTLOWAT (30). This provides flow control.

13.10.3. ttstart() — Start Output

```
/* tty.c */
ttstart(atp)
struct tty *atp;
{
    register int *addr, c;
```

```
    register struct tty *tp;

    tp = atp;
    addr = tp->t_addr;
    if (tp->t_state&SSTART) {
        (*addr.func)(tp);       /* Special start routine */
        return;
    }
    if ((addr->tttcsr&DONE)==0 || tp->t_state&TIMEOUT)
        return;                 /* Device busy or delay pending */
```

```
    if ((c=getc(&tp->t_outq)) >= 0) {
        if (c<=0177)
            addr->tttbuf = c | (partab[c]&0200);
        else {
            timeout(ttrstrt, tp, c&0177);
            tp->t_state =| TIMEOUT;
        }
    }
}
```

Get a character from the output queue. If it's a real character (≤0177), send it to the device with parity. If it's a delay (>0177), set a timeout.

13.11. The KL-11 Console Driver

The KL-11 is the console terminal interface:

13.11.1. klopen() — Open Console

```
/* kl.c */
klopen(dev, flag)
{
    register *addr;
    register struct tty *tp;

    if(dev.d_minor >= NKL11) {
        u.u_error = ENXIO;
        return;
    }
    tp = &kl11[dev.d_minor];
    tp->t_quit = 034;           /* Ctrl-\ */
    tp->t_intrup = 0177;        /* DEL */
```

Set default control characters.

```
    if (u.u_procp->p_ttyp == 0)
        u.u_procp->p_ttyp = tp;
```

If process has no controlling terminal, this becomes it.

```
    addr = KLADDR;
    if(dev.d_minor)
        addr = KLBASE-8 + 8*dev.d_minor;
    tp->t_addr = addr;
    tp->t_flags = XTABS|LCASE|ECHO|CRMOD;
    tp->t_state = CARR_ON;
    addr->klrcsr =| IENABLE|DSRDY|RDRENB;
    addr->kltcsr =| IENABLE;
}
```

Set device address, default flags, and enable interrupts.

13.11.2. klrint() — Receive Interrupt

```
/* kl.c */
klrint(dev)
{
    register int c, *addr;
    register struct tty *tp;

    tp = &kl11[dev.d_minor];
    addr = tp->t_addr;
    c = addr->klrbuf;
    addr->klrcsr =| RDRENB;     /* Re-enable receiver */
    if ((c&0177)==0)
        addr->kltbuf = c;       /* Hardware botch workaround */
    ttyinput(c, tp);
}
```

Read character from device, re-enable receiver, pass to `ttyinput()`.

13.11.3. klxint() — Transmit Interrupt

```
/* kl.c */
klxint(dev)
{
    register struct tty *tp;

    tp = &kl11[dev.d_minor];
    ttstart(tp);
    if (tp->t_outq.c_cc == 0 || tp->t_outq.c_cc == TTLOWAT)
        wakeup(&tp->t_outq);
}
```

Transmit complete—start next character and wake writers if queue has drained.

13.11.4. klsgtty() — Get/Set TTY Parameters

```
/* kl.c */
klsgtty(dev, v)
int *v;
{
    register struct tty *tp;

    tp = &kl11[dev.d_minor];
    if (v)
        v[2] = tp->t_flags;     /* Get: return flags */
    else {
        wflushtty(tp);
        tp->t_flags = u.u_arg[2]; /* Set: change flags */
    }
}
```

Used by `stty` and `gtty` system calls.

13.12. Flow Control

```
    Output Queue Level

    50 +-- TTHIWAT -- ttwrite() sleeps
       |
    30 +-- TTLOWAT -- klxint() wakes writers
       |
     0 +-- Empty
```

When the output queue exceeds 50 characters, writers sleep. When it drops to 30 or below, they're awakened. This prevents fast writers from flooding slow terminals.

13.13. Uppercase-Only Terminals

Many early terminals only had uppercase letters. LCASE mode provides bidirectional mapping:

Input: `HELLO` → `hello`

Output: `hello` → `HELLO`, { → \ (, } →), etc.

The `maptab[]` array handles escape sequences like \{ → {.

13.14. Signal Generation

```
if ((t_flags&RAW)==0 && (c==tp->t_quit || c==tp->t_intrup)) {
    signal(tp, c==tp->t_intrup? SIGINT:SIGQIT);
    flushtty(tp);
    return;
}
```

In cooked mode:

- **DEL** (0177) → SIGINT (interrupt)
- **Ctrl-** (034) → SIGQIT (quit with core dump)

The `signal()` function sends the signal to all processes with this controlling terminal.

13.15. Summary

- **Three queues**: raw (unedited), canonical (edited), output
- **Cooked mode**: Line editing, echo, signal generation
- **Raw mode**: Unprocessed I/O for special applications
- **Flow control**: TTHIWAT/TTLOWAT prevent output flooding
- **Delays**: Mechanical timing for print head movement
- **LCASE**: Support for uppercase-only terminals

13.16. Key Design Points

1. **Interrupt-driven**: Characters processed in interrupt handlers, minimal latency.
2. **Queue-based**: Decouples user processes from hardware timing.
3. **Modal**: RAW vs cooked mode serves different application needs.
4. **Device-independent**: Common code in tty.c, device-specific in kl.c.
5. **Character blocks**: Dynamic allocation avoids fixed buffer sizes.

13.17. Experiments

1. **RAW mode**: Write a program that reads single characters without echo.
2. **Change control characters**: Use `stty` to change erase from # to backspace.

3. **Overflow behavior**: Send more than TTYHOG characters without newlines.
4. **Flow control**: Write rapidly to a slow terminal and observe sleeping.

13.18. Further Reading

- Chapter 14: Block Devices — Contrasts with character device model
- Chapter 7: Traps and System Calls — Signal delivery mechanism

---

Next: Chapter 14 — Block Devices

14. Chapter 14: Block Devices

14.1. Overview

Block devices transfer data in fixed-size blocks (512 bytes in UNIX v4) and support random access. The primary block device is the disk—the RK05 cartridge disk that holds 2.38 MiB on a removable pack. Block devices go through the buffer cache, providing caching and a uniform interface that hides the complexity of disk geometry and timing.

This chapter examines the RK05 disk driver as a case study in block device implementation.

14.2. Source Files

File	Purpose
`usr/sys/conf.h`	Device switch tables
`usr/sys/dmr/rk.c`	RK05 disk driver
`usr/sys/dmr/bio.c`	Buffer I/O interface

14.3. Prerequisites

- Chapter 12: Buffer Cache (`bread`, `bwrite`, strategy interface)

14.4. The Block Device Switch

```
/* conf.h */
struct bdevsw {
    int (*d_open)();      /* Open device */
    int (*d_close)();     /* Close device */
    int (*d_strategy)();  /* Queue I/O request */
    int *d_tab;           /* Device queue table */
} bdevsw[];
```

Every block device provides these four entry points. The `d_strategy` routine is the key—it accepts buffer requests and handles all I/O.

Example configuration:

```
/* conf/c.c */
struct bdevsw bdevsw[] {
    &nulldev, &nulldev, &rkstrategy, &rktab,   /* 0 = rk */
    &nulldev, &nulldev, &tmstrategy, &tmtab,   /* 1 = tm (tape) */
    0
};
```

14.5. Device Numbers

```
/* conf.h */
struct {
    char d_minor;    /* Unit number within device type */
    char d_major;    /* Index into bdevsw[] */
};
```

A device number encodes:

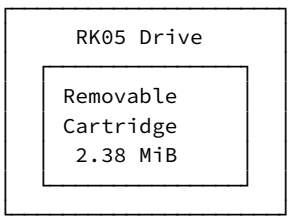
- **Major number**: Which driver (0=rk, 1=tm, etc.)
- **Minor number**: Which unit or partition

For example, device 0407 = major 04, minor 07 = RK disk, unit 7.

14.6. The RK05 Disk

The RK05 is a cartridge disk:

- **Capacity**: 2.38 MiB per pack
- **Geometry**: 203 cylinders × 2 surfaces × 12 sectors
- **Block size**: 512 bytes (256 words)
- **Total blocks**: 4,872 per disk



14.6.1. Hardware Registers

```
#define RKADDR 0177400    /* Base address */

struct {
    int rkds;    /* Drive status */
    int rker;    /* Error register */
    int rkcs;    /* Control/status */
    int rkwc;    /* Word count */
    int rkba;    /* Bus address */
    int rkda;    /* Disk address */
};
```

The disk address register encodes cylinder, surface, and sector:

```
rkda = (drive << 13) | (cylinder << 4) | sector
```

14.7. rkstrategy() — Queue a Request

The strategy routine is called by the buffer cache to perform I/O:

```
/* rk.c */
rkstrategy(abp)
struct buf *abp;
{
    register struct buf *bp;
    register *qc, *ql;
    int d;

    bp = abp;
    d = bp->b_dev.d_minor-7;
    if(d <= 0)
        d = 1;
    if (bp->b_blkno >= NRKBLK*d) {
        bp->b_flags =| B_ERROR;
        iodone(bp);
        return;
    }
```

Validate the block number. If it's beyond the disk capacity, return an error immediately.

```
    bp->av_forw = 0;
    bp->b_flags =& ~B_SEEK;
    if(bp->b_dev.d_minor < 8)
        d = bp->b_dev.d_minor;
    else
        d = lrem(bp->b_blkno, d);
```

Determine which physical drive this request is for.

```
    spl5();
    if ((ql = *(qc = &rk_q[d])) == NULL) {
        *qc = bp;
        if (RKADDR->rkcs&CTLRDY)
            rkstart();
        goto ret;
    }
```

If the drive's queue is empty, add this request and start I/O if the controller is ready.

```
    while ((qc = ql->av_forw) != NULL) {
        if (ql->b_blkno<bp->b_blkno
         && bp->b_blkno<qc->b_blkno
         || ql->b_blkno>bp->b_blkno
         && bp->b_blkno>qc->b_blkno) {
            ql->av_forw = bp;
            bp->av_forw = qc;
            goto ret;
        }
        ql = qc;
    }
    ql->av_forw = bp;
ret:
    spl0();
}
```

Elevator algorithm: Insert the request in sorted order by block number. This minimizes seek time by processing requests in the direction the head is moving, like an elevator.

> **The Elevator Algorithm in CS History.** Also known as SCAN, the elevator algorithm is one of the foundational disk scheduling algorithms. The name comes from its behavior: like an elevator, the disk head services requests in one direction until exhausted, then reverses. First-come-first-served (FCFS) scheduling causes the head to thrash wildly across the platter; elevator ordering dramatically reduces average seek time. Variants include C-SCAN (circular scan, always sweeping in one direction) and LOOK (reversing at the last request rather than the disk edge). With the rise of SSDs—which have no mechanical seek time—elevator scheduling became irrelevant for flash storage, but it remains essential for understanding I/O scheduling principles and is still used in systems with rotational media.

14.8. rkstart() — Initiate Seeks

```
/* rk.c */
rkstart()
{
    register struct buf *bp;
    register int *qp;

    for (qp = rk_q; qp < &rk_q[NRK];) {
        if ((bp = *qp++) && (bp->b_flags&B_SEEK)==0) {
            RKADDR->rkda = rkaddr(bp);
            rkcommand(IENABLE|SEEK|GO);
            if (RKADDR->rkcs<0) {
                bp->b_flags =| B_ERROR;
                *--qp = bp->av_forw;
                iodone(bp);
                rkerror();
            } else
                bp->b_flags =| B_SEEK;
        }
    }
}
```

Overlapped seeks: Start seeks on all drives that have pending requests. While one drive is seeking, another can be transferring data. The RK11 controller supports this parallelism.

14.9. rkaddr() — Compute Disk Address

```
/* rk.c */
rkaddr(bp)
struct buf *bp;
{
    register struct buf *p;
    register int b;
    int d, m;

    p = bp;
    b = p->b_blkno;
    m = p->b_dev.d_minor - 7;
    if(m <= 0)
        d = p->b_dev.d_minor;
    else {
        d = lrem(b, m);
        b = ldiv(b, m);
    }
    return(d<<13 | (b/12)<<4 | b%12);
}
```

Converts a linear block number to RK05 physical address:

- Sector = block % 12
- Cylinder = block / 12
- Pack into: `(drive << 13) | (cylinder << 4) | sector`

14.10. rkintr() — Interrupt Handler

```
/* rk.c */
rkintr()
{
    register struct buf *bp;

    if (RKADDR->rkcs < 0) {
        if (RKADDR->rker&WLO || ++rktab.d_errcnt>10)
            rkpost(B_ERROR);
        rkerror();
    }
```

Check for errors. Write-lock errors are fatal; others retry up to 10 times.

```
    if (RKADDR->rkcs&SEEKCMP) {
        rk_ap = &rk_q[(RKADDR->rkds>>13) & 07];
        devstart(*rk_ap, &RKADDR->rkda, rkaddr(*rk_ap), 0);
    } else
        rkpost(0);
}
```

Two types of interrupts:

1. **Seek complete** (SEEKCMP): Start the data transfer using `devstart()`
2. **Transfer complete**: Call `rkpost()` to finish up

14.11. devstart() — Start Data Transfer

```
/* bio.c */
devstart(bp, devloc, devblk, hbcom)
struct buf *bp;
int *devloc;
{
    register int *dp;
    register struct buf *rbp;
    register int com;

    dp = devloc;
    rbp = bp;
```

```
    *dp = devblk;               /* Block address */
    *--dp = rbp->b_addr;        /* Buffer address */
    *--dp = rbp->b_wcount;      /* Word count */
    com = (hbcom<<8) | IENABLE | GO | rbp->b_flags&B_XMEM;
    if (rbp->b_flags&B_READ)
        com =| RCOM;
    else
        com =| WCOM;
    *--dp = com;
}
```

Programs the device registers and starts the transfer. The PDP-11 DMA controller handles the actual data movement.

14.12. rkpost() — Complete I/O

```
/* rk.c */
rkpost(errbit)
{
    register struct buf *bp;

    if (rk_ap) {
        bp = *rk_ap;
        bp->b_flags =| B_DONE | errbit;
        *rk_ap = bp->av_forw;
        rk_ap = NULL;
        iodone(bp);
        rktab.d_errcnt = 0;
        rkstart();
    }
}
```

Mark the buffer done, remove from queue, call `iodone()` to wake waiting processes, and start the next request.

14.13. Raw I/O: rkread() and rkwrite()

```
/* rk.c */
rkread(dev)
{
    physio(rkstrategy, &rrkbuf, dev, B_READ);
}

rkwrite(dev)
{
```

```
    physio(rkstrategy, &rrkbuf, dev, B_WRITE);
}
```

Raw (character) device interface bypasses the buffer cache, transferring directly to/from user memory. Uses `physio()` from bio.c.

14.14. physio() — Physical I/O

```
/* bio.c */
physio(strat, abp, dev, rw)
struct buf *abp;
int (*strat)();
{
    register struct buf *bp;
    register char *base;
    register int nb;

    bp = abp;
    base = u.u_base;
    /* Validate user buffer address */
    if (base&01 || u.u_count&01 || base>=base+u.u_count)
        goto bad;
```

Check alignment and bounds.

```
    spl6();
    while (bp->b_flags&B_BUSY) {
        bp->b_flags =| B_WANTED;
        sleep(bp, PRIBIO);
    }
    bp->b_flags = B_BUSY | rw;
    bp->b_dev = dev;
    /* Set up transfer parameters */
    bp->b_blkno = lshift(u.u_offset, -9);
    bp->b_wcount = -(u.u_count>>1);
```

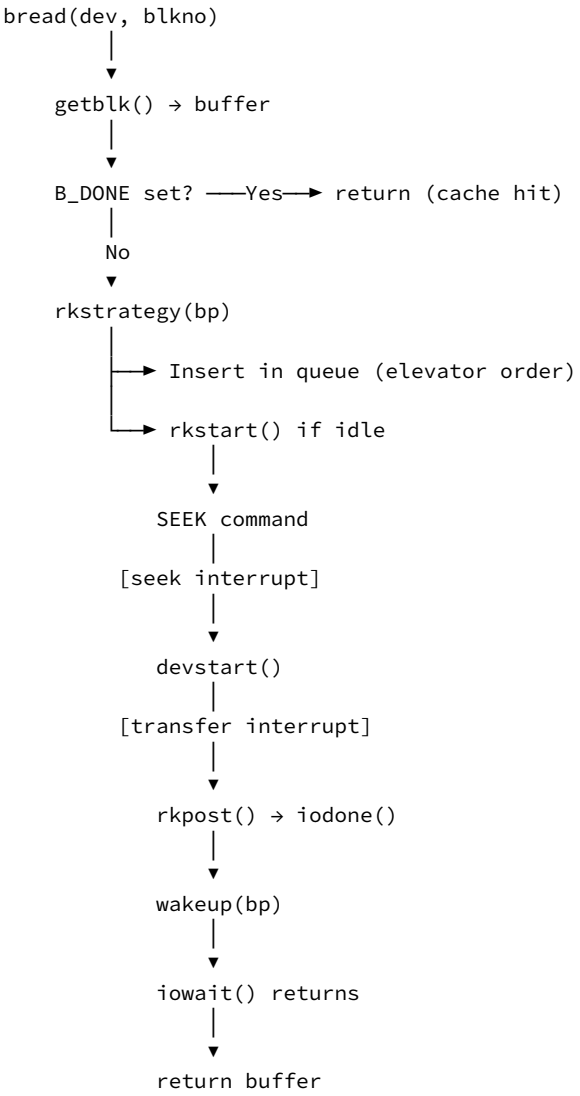
Wait for the raw buffer, then set it up for transfer.

```
    u.u_procp->p_flag =| SLOCK;
    (*strat)(bp);
    spl6();
    while ((bp->b_flags&B_DONE) == 0)
        sleep(bp, PRIBIO);
    u.u_procp->p_flag =& ~SLOCK;
```

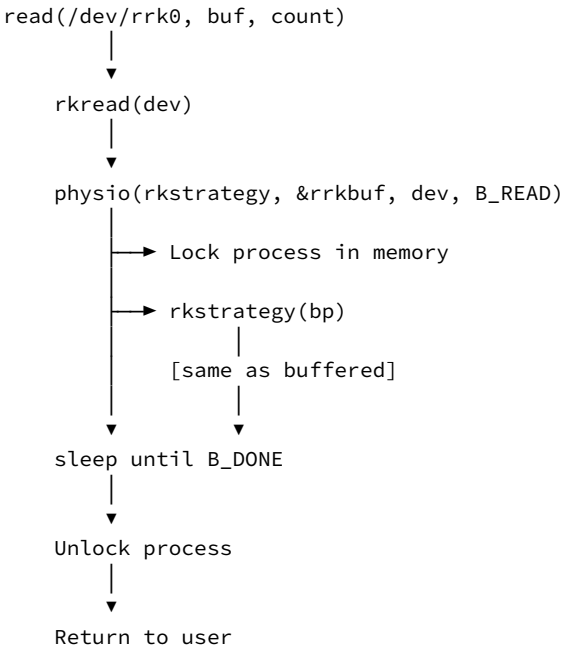
Lock process in memory (can't swap during DMA!), call strategy, wait for completion.

14.15. I/O Flow Summary

14.15.1. Buffered Read



14.15.2. Raw Read

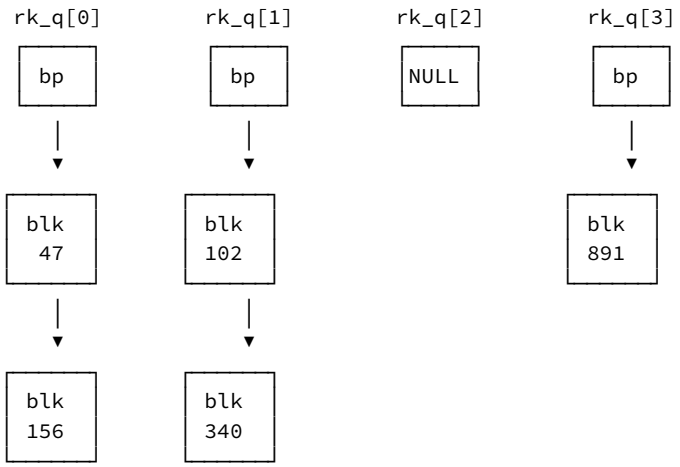


14.16. Error Handling

```
rkerror()
{
    register int *qp;
    register struct buf *bp;

    rkcommand(IENABLE|RESET|GO);
    for (qp = rk_q; qp < &rk_q[NRK];)
        if ((bp = *qp++) != NULL && bp->b_flags&B_SEEK) {
            RKADDR->rkda = rkaddr(bp);
            while ((RKADDR->rkds&(DRY|ARDY)) == DRY);
            rkcommand(IENABLE|DRESET|GO);
        }
}
```

On error: reset controller, recalibrate all drives that were seeking. The strategy routine will retry up to 10 times before giving up.

14.17. Multiple Drives



Each drive has its own queue. Seeks can overlap across drives—while drive 0 seeks to cylinder 47, drive 1 can be transferring block 102.

14.18. Summary

- **Strategy routine**: Main entry point, queues requests
- **Elevator algorithm**: Minimizes seek time
- **Overlapped seeks**: Multiple drives seek simultaneously
- **Interrupt-driven**: CPU free during disk operations
- **Error retry**: Automatic recovery from transient errors
- **Raw I/O**: Bypasses cache for special applications

14.19. Key Design Points

1. **Asynchronous**: `rkstrategy()` returns immediately; completion via interrupt.
2. **Queueing**: Requests accumulate and are processed optimally.
3. **Parallelism**: Controller handles seeks on multiple drives.
4. **Abstraction**: Buffer cache sees only `strategy()`—no geometry details.
5. **DMA**: Data moves without CPU involvement.

14.20. Experiments

1. **Trace seeks**: Add printf to see elevator ordering in action.

2. **Measure throughput**: Compare sequential vs random block access.
3. **Force errors**: Observe retry behavior with a bad block.
4. **Raw vs buffered**: Compare performance for large sequential reads.

14.21. Further Reading

- Chapter 12: Buffer Cache — The interface above block devices
- Chapter 13: TTY Subsystem — Contrasts character device model
- Chapter 15: Character Devices — Non-block device patterns

Next: Chapter 15 — Character Devices

15. Chapter 15: Character Devices

15.1. Overview

Character devices transfer data byte-by-byte without buffering through the block cache. They include terminals (Chapter 13), but also pseudo-devices like `/dev/null` and `/dev/mem`. Unlike block devices that hide behind the buffer cache, character devices interact directly with user processes through their read and write routines.

This chapter examines the memory pseudo-devices—elegant examples of the character device model.

15.2. Source Files

File	Purpose
`usr/sys/conf.h`	Device switch tables
`usr/sys/dmr/mem.c`	Memory devices

15.3. Prerequisites

- Chapter 10: File I/O (`passc`, `cpass`, `u.u_offset`)
- Chapter 13: TTY Subsystem (character device example)

15.4. The Character Device Switch

```
/* conf.h */
struct cdevsw {
    int (*d_open)();     /* Open device */
    int (*d_close)();    /* Close device */
    int (*d_read)();     /* Read from device */
    int (*d_write)();    /* Write to device */
    int (*d_sgtty)();    /* Get/set TTY parameters */
} cdevsw[];
```

Character devices provide five entry points. The `d_sgtty` routine is optional—only terminals use it.

Example configuration:

```
/* conf/c.c */
struct cdevsw cdevsw[] {
    &klopen, &klclose, &klread, &klwrite, &klsgtty,  /* 0 = console */
    &nulldev,&nulldev, &mmread, &mmwrite, &nodev,    /* 1 = mem */
    0
};
```

> **Why "switch"?** The term "switch" in `cdevsw` doesn't mean a toggle—it refers to a dispatch table, like a `switch` statement in C. When the kernel needs to perform an operation on a character device, it uses the major device number as an index into `cdevsw` and "switches" to the appropriate driver function: `(*cdevsw[major].d_read)(dev)`. It's a jump table that routes calls to the right driver.

15.5. Block vs Character Devices

Aspect	Block Device	Character Device
Data unit	512-byte blocks	Bytes
Buffering	Through buffer cache	Direct or driver-managed
Access	Random (any block)	Sequential typical
Interface	`strategy()`	`read()`, `write()`
Examples	Disks, tapes	Terminals, /dev/null

Many devices have both interfaces:

- `/dev/rk0` — Block device (buffered)
- `/dev/rrk0` — Character device (raw, unbuffered)

15.6. The Memory Devices

```
/*
 * Memory special file
 * minor device 0 is physical memory
 * minor device 1 is kernel memory
 * minor device 2 is EOF/RATHOLE
 */
```

Three pseudo-devices in one driver:

- `/dev/mem` (minor 0) — Physical memory
- `/dev/kmem` (minor 1) — Kernel virtual memory
- `/dev/null` (minor 2) — Data sink/source

15.7. mmread() — Read Memory

```
/* mem.c */
mmread(dev)
{
    register c, bn, on;
    int a;

    if(dev.d_minor == 2)
        return;                 /* /dev/null: EOF immediately */
```

Reading from `/dev/null` returns nothing—immediate end-of-file.

```
    do {
        bn = lshift(u.u_offset, -6);    /* Block number (64-byte pages) */
        on = u.u_offset[1] & 077;       /* Offset within page */
        a = UISA->r[0];                 /* Save segment register */
        spl7();
        UISA->r[0] = bn;                /* Map to requested page */
```

The PDP-11 MMU limits direct access to 64KB. To read arbitrary physical memory, we temporarily remap segment 0 to point to the desired page.

```
        if(dev.d_minor == 1)
            UISA->r[0] = KISA->r[(bn>>7)&07] + (bn & 0177);
```

For `/dev/kmem`, translate through the kernel's address space. This allows reading kernel data structures.

```
        c = fubyte(on);                 /* Read the byte */
        UISA->r[0] = a;                 /* Restore segment register */
        spl0();
    } while(u.u_error==0 && passc(c)>=0);
}
```

Read one byte via `fubyte()`, restore the mapping, pass to user via `passc()`. Repeat until done.

15.8. mmwrite() — Write Memory

```
/* mem.c */
mmwrite(dev)
{
    register c, bn, on;
    int a;

    if(dev.d_minor == 2) {
        c = u.u_count;
        u.u_count = 0;
        u.u_base =+ c;
        dpadd(u.u_offset, c);
        return;
    }
```

Writing to `/dev/null`: Accept all data, advance pointers, discard everything. The “rathole” consumes infinite data.

```
    for(;;) {
        bn = lshift(u.u_offset, -6);
        on = u.u_offset[1] & 077;
        if ((c=cpass())<0 || u.u_error!=0)
            break;
        a = UISA->r[0];
        spl7();
        UISA->r[0] = bn;
        if(dev.d_minor == 1)
            UISA->r[0] = KISA->r[(bn>>7)&07] + (bn & 0177);
        subyte(on, c);          /* Write the byte */
        UISA->r[0] = a;
        spl0();
    }
}
```

Same mapping trick as read. Get byte from user via `cpass()`, write to memory via `subyte()`.

15.9. Use Cases

15.9.1. /dev/null — The Bit Bucket

```
$ command > /dev/null    # Discard output
$ cat /dev/null          # Empty file (0 bytes)
```

Writing discards data; reading returns EOF immediately.

15.9.2. /dev/mem — Physical Memory

```
$ od /dev/mem            # Dump physical memory
```

Used by debuggers and system utilities to examine raw memory. Dangerous—can crash the system if misused.

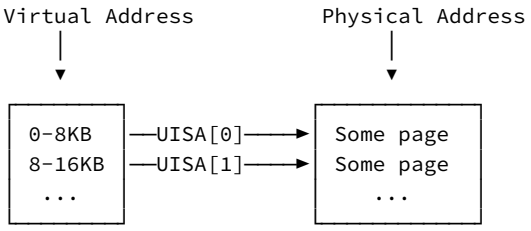
15.9.3. /dev/kmem — Kernel Memory

```
$ ps                     # Reads process table from /dev/kmem
```

Programs like `ps` read kernel data structures (proc table, etc.) through this device. The kernel address translation makes kernel variables accessible.

15.10. The MMU Trick

The PDP-11's memory management unit maps virtual addresses to physical:



To access arbitrary physical memory:

1. Save current UISA[0]

2. Set UISA[0] = target page
3. Access address 0-8KB (maps to target)
4. Restore UISA[0]

This must be done at high priority (spl7) to prevent interrupts from using the corrupted mapping.

15.11. Contrast with TTY

Both are character devices, but very different:

Aspect	TTY	Memory
Hardware	Real device (terminal)	Pseudo-device
Interrupts	Yes (keyboard, transmit)	No
Buffering	Three queues	None
Processing	Echo, line editing	None
Blocking	Waits for input	Never blocks

15.12. Other Character Devices

15.12.1. Line Printer (lp.c)

Output-only device:

- `lpwrite()` — Send characters to printer
- `lpstart()` — Start printing from output queue
- `lpintr()` — Handle printer-ready interrupt

15.12.2. Paper Tape (pc.c)

```
pcread()   /* Read from paper tape reader */
pcwrite()  /* Write to paper tape punch */
```

15.12.3. Raw Disk (rk.c)

```
rkread(dev)
{
    physio(rkstrategy, &rrkbuf, dev, B_READ);
}
```

Character interface to block device. Uses `physio()` to bypass buffer cache—data goes directly between disk and user memory.

15.13. Device Registration

```
/* conf/c.c */
struct cdevsw cdevsw[] {
    &klopen,  &klclose, &klread,  &klwrite, &klsgtty,  /* 0 = kl */
    &nulldev, &nulldev, &mmread,  &mmwrite, &nodev,    /* 1 = mem */
    &nulldev, &nulldev, &rkread,  &rkwrite, &nodev,    /* 2 = rrk */
    &pcopen,  &pcclose, &pcread,  &pcwrite, &nodev,    /* 3 = pc */
    &lpopen,  &lpclose, &nodev,   &lpwrite, &nodev,    /* 4 = lp */
    0
};
```

`nulldev` — Does nothing (for devices that don't need open/close) `nodev` — Returns error (for unsupported operations)

15.14. Creating Device Files

```
# mknod /dev/null c 1 2
#         |       | | |
#         |       | | └── minor number (2 = null)
#         |       | └──── major number (1 = mem driver)
#         |       └────── character device
#         └──────────────── device file name
```

The `mknod` command creates special files that point to device drivers through major/minor numbers.

15.15. Summary

- **Character devices** transfer bytes, not blocks
- **Direct interface**: `read()`, `write()` instead of `strategy()`
- **Varied purposes**: Terminals, pseudo-devices, raw disk access
- **/dev/null**: Discards writes, returns EOF on read
- **/dev/mem**: Access physical memory via MMU tricks
- **/dev/kmem**: Access kernel address space

15.16. Key Design Points

1. **Simplicity**: Most character devices just move bytes—no complex buffering.
2. **Flexibility**: Same interface for hardware (terminals) and pseudo-devices (null).
3. **Direct access**: User process interacts without buffer cache intermediary.
4. **MMU manipulation**: Clever use of memory mapping for /dev/mem.
5. **Dual interfaces**: Block devices often have character (raw) counterparts.

15.17. Experiments

1. **Read /dev/mem**: Write a program to read the first 1KB of physical memory.
2. **Benchmark /dev/null**: Time writing large amounts to /dev/null vs a real file.
3. **Examine kernel**: Read proc table from /dev/kmem (requires knowing the address).
4. **Create devices**: Use mknod to create new device files.

15.18. Further Reading

- Chapter 13: TTY Subsystem — Complex character device
- Chapter 14: Block Devices — The other device model
- Chapter 10: File I/O — How devices fit in the file abstraction

---

Part IV Complete! Next: Part V — User Space

Part V.

User Space

16. Chapter 16: The Shell

16.1. Overview

The shell is the user's interface to UNIX—a program that reads commands, parses them, and executes them. In about 800 lines of C, it implements command execution, I/O redirection, pipes, background processes, and shell scripts. The shell's elegance comes from its simplicity: it's just another user program that happens to orchestrate other programs.

16.2. Source Files

File	Purpose
`usr/source/s2/sh.c`	The complete shell

16.3. Prerequisites

- Chapter 5: Process Management (`fork`, `exec`, `wait`)
- Chapter 7: Traps and System Calls (signals)
- Chapter 10: File I/O (file descriptors, `dup`)

16.4. Shell Overview

The shell is a loop:

1. Print prompt
2. Read a line
3. Parse into a syntax tree
4. Execute the tree
5. Repeat

```
loop:
    if(promp != 0)
        prs(promp);
    peekc = getc();
    main1();
    goto loop;
```

16.5. Data Structures

16.5.1. Global State

```
char *dolp;          /* Dollar expansion pointer */
char **dolv;         /* Argument vector ($1, $2...) */
int  dolc;           /* Argument count */
char *promp;         /* Prompt string (% or #) */
char *linep;         /* Current position in line buffer */
char **argp;         /* Current position in args array */
int  *treep;         /* Current position in tree buffer */
char peekc;          /* Lookahead character */
char error;          /* Syntax error flag */
char uid;            /* User ID (0 = root) */
char setintr;        /* Interactive mode flag */
```

16.5.2. Syntax Tree Nodes

```
/* Tree node layout */
#define dtyp 0       /* Node type */
#define dlef 1       /* Left child or input file */
#define drit 2       /* Right child or output file */
#define dflg 3       /* Flags */
#define dspr 4       /* Subshell pointer */
#define dcom 5       /* Command and arguments start here */

/* Node types */
#define tcom 1       /* Simple command */
#define tpar 2       /* Parenthesized subshell */
#define tfil 3       /* Pipeline */
#define tlst 4       /* List (cmd; cmd) */
```

16.6. main() — Shell Startup

```
main(c, av)
int c;
char **av;
{
    register f;
    register char *acname, **v;

    close(2);
    if((f=dup(1)) != 2)
        close(f);
```

Ensure stderr (fd 2) goes to the same place as stdout.

```
    v = av;
    acname = "/usr/adm/sh_acct";
    promp = "% ";
    if(((uid = getuid())&0377) == 0) {
        promp = "# ";
        acname = "/usr/adm/su_acct";
    }
```

Set prompt: % for normal users, # for root.

```
    if(c > 1) {
        promp = 0;              /* No prompt for scripts */
        close(0);
        f = open(v[1], 0);      /* Open script as stdin */
        if(f < 0) {
            prs(v[1]);
            err(": cannot open");
        }
    }
```

If given a filename argument, run it as a script.

```
    if(**v == '-') {
        setintr++;
        signal(quit, 1);        /* Ignore quit */
        signal(intr, 1);        /* Ignore interrupt */
    }
    dolv = v+1;
    dolc = c-1;
```

Login shells (name starts with −) ignore signals. Set up $1, $2, etc.

16.7. Lexical Analysis: word()

```
word()
{
    register char c, c1;

    *argp++ = linep;

loop:
    switch(c = getc()) {

    case ' ':
    case '\t':
        goto loop;              /* Skip whitespace */

    case '\'':
    case '"':
        c1 = c;
        while((c=readc()) != c1) {
            if(c == '\n') {
                error++;
                peekc = c;
                return;
            }
            *linep++ = c|quote;  /* Mark as quoted */
        }
        goto pack;
```

Quoted strings: characters inside quotes are marked with the `quote` bit (0200) so metacharacters aren't treated specially.

```
    case '&':
    case ';':
    case '<':
    case '>':
    case '(':
    case ')':
    case '|':
    case '^':
    case '\n':
        *linep++ = c;
        *linep++ = '\0';
        return;
    }
```

Metacharacters are returned as single-character tokens.

```
    peekc = c;

pack:
    for(;;) {
```

```
        c = getc();
        if(any(c, " '\"\t;&<>()|^\n")) {
            peekc = c;
            if(any(c, "\"'"))
                goto loop;
            *linep++ = '\0';
            return;
        }
        *linep++ = c;
    }
}
```

Regular words: collect characters until a metacharacter or whitespace.

16.8. getc() — Character Input with Expansion

```
getc()
{
    register char c;

    if(peekc) {
        c = peekc;
        peekc = 0;
        return(c);
    }
```

Handle lookahead character.

```
getd:
    if(dolp) {
        c = *dolp++;
        if(c != '\0')
            return(c);
        dolp = 0;
    }
```

If expanding a $n variable, return characters from it.

```
    c = readc();
    if(c == '\\') {
        c = readc();
        if(c == '\n')
            return(' ');        /* Line continuation */
        return(c|quote);        /* Escaped character */
    }
    if(c == '$') {
        c = getc();
        if(c>='0' && c<='9') {
```

```
            if(c-'0' < dolc)
                dolp = dolv[c-'0'];
            goto getd;          /* Expand $n */
        }
    }
    return(c&0177);
}
```

Handle \ escapes and $1, $2, etc. parameter expansion.

16.9. Parsing: Recursive Descent

The parser builds a syntax tree using recursive descent—a parsing technique where each grammar rule becomes a function, and functions call each other to match the input structure. The beauty of recursive descent is that the code mirrors the grammar directly.

Parser Call Hierarchy:

```
                    syntax()
                       |
                       v
syn1() ———————— handles ; and & (command lists)
                       |
                       v
syn2() ———————— handles | (pipelines)
                       |
                       v
syn3() ———————— simple commands, (subshells), redirects
```

Each level handles operators of a specific precedence. Lower precedence operators (`;`, &) are parsed first at the top, so they become the root of the tree. Higher precedence operators (`|`) bind tighter and appear deeper.

Example Parse Tree for `ls | grep foo; echo done`:

```
                tlst (;)
               /        \
           tfil (|)      tcom
          /      \      "echo"
       tcom     tcom    "done"
       "ls"    "grep"
               "foo"
```

The tree is executed bottom-up, left-to-right: first the pipeline `ls | grep foo` runs, then `echo done`.

16.9.1. syntax() — Top Level

```
syntax(p1, p2)
char **p1, **p2;
{
    while(p1 != p2) {
        if(any(**p1, ";&\n"))
            p1++;
        else
            return(syn1(p1, p2));
    }
    return(0);
}
```

Skip separators, parse command list.

16.9.2. syn1() — Command Lists

```
/*
 * syn1
 *    syn2
 *    syn2 & syntax
 *    syn2 ; syntax
 */
syn1(p1, p2)
char **p1, **p2;
{
    register char **p;
    register *t, *t1;
    int l;

    l = 0;
    for(p=p1; p!=p2; p++)
    switch(**p) {

    case '(':
        l++;
        continue;

    case ')':
        l--;
        continue;

    case '&':
    case ';':
    case '\n':
        if(l == 0) {
            t = tree(4);
            t[dtyp] = tlst;
            t[dlef] = syn2(p1, p);
            t[dflg] = 0;
```

```
            if(**p == '&') {
                t1 = t[dlef];
                t1[dflg] =| fand|fint;  /* Background */
            }
            t[drit] = syntax(p+1, p2);
            return(t);
        }
    }
    return(syn2(p1, p2));
}
```

Handle ; (sequential) and & (background). Track parentheses depth.

16.9.3. syn2() — Pipelines

```
/*
 * syn2
 *    syn3
 *    syn3 | syn2
 */
syn2(p1, p2)
char **p1, **p2;
{
    char **p;
    int l, *t;

    l = 0;
    for(p=p1; p!=p2; p++)
    switch(**p) {

    case '(':
        l++;
        continue;

    case ')':
        l--;
        continue;

    case '|':
    case '^':
        if(l == 0) {
            t = tree(4);
            t[dtyp] = tfil;
            t[dlef] = syn3(p1, p);
            t[drit] = syn2(p+1, p2);
            return(t);
        }
    }
    return(syn3(p1, p2));
}
```

Handle | (pipe). Note ^ is also pipe (older syntax).

16.9.4. syn3() — Simple Commands

```
/*
 * syn3
 *    ( syn1 ) [ < in ] [ > out ]
 *    word word* [ < in ] [ > out ]
 */
syn3(p1, p2)
char **p1, **p2;
{
    /* ... parse redirections and command words ... */

    if(lp != 0) {
        /* Parenthesized subshell */
        t = tree(5);
        t[dtyp] = tpar;
        t[dspr] = syn1(lp, rp);
        goto out;
    }
    /* Simple command */
    t = tree(n+5);
    t[dtyp] = tcom;
    for(l=0; l<n; l++)
        t[l+dcom] = b[l];
out:
    t[dflg] = flg;
    t[dlef] = i;           /* Input redirect */
    t[drit] = o;           /* Output redirect */
    return(t);
}
```

Parse I/O redirections (<, >, >>) and collect command words.

16.10. Execution: execute()

```
execute(t, pf1, pf2)
int *t, *pf1, *pf2;
{
    int i, f, pv[2];
    register *t1;

    if(t != 0)
    switch(t[dtyp]) {

    case tcom:
        /* Simple command */
```

16.10.1. Built-in Commands

```
        cp1 = t[dcom];
        if(equal(cp1, "chdir")) {
            if(t[dcom+1] != 0) {
                if(chdir(t[dcom+1]) < 0)
                    err("chdir: bad directory");
            }
            return;
        }
        if(equal(cp1, "shift")) {
            dolv[1] = dolv[0];
            dolv++;
            dolc--;
            return;
        }
        if(equal(cp1, "login")) {
            execv("/bin/login", t+dcom);
            return;
        }
        if(equal(cp1, "wait")) {
            pwait(-1, 0);
            return;
        }
        if(equal(cp1, ":"))
            return;
```

Built-ins execute in the shell process itself (no fork).

16.10.2. External Commands

```
    case tpar:
        f = t[dflg];
        i = 0;
        if((f&fpar) == 0)
            i = fork();
        if(i == -1) {
            err("try again");
            return;
        }
        if(i != 0) {
            /* Parent */
            if((f&fand) != 0) {
                prn(i);
                prs("\n");
                return;
            }
            if((f&fpou) == 0)
                pwait(i, t);
            return;
        }
```

Fork a child process. Parent waits (unless &).

```
        /* Child process */
        if(t[dlef] != 0) {
            close(0);
            i = open(t[dlef], 0);   /* Input redirect */
        }
        if(t[drit] != 0) {
            if((f&fcat) != 0) {
                i = open(t[drit], 1);
                seek(i, 0, 2);       /* Append */
            } else
                i = creat(t[drit], 0666);
            close(1);
            dup(i);
            close(i);
        }
```

Handle input/output redirection by manipulating file descriptors before exec.

```
        execv(t[dcom], t+dcom);
        /* Try /bin/ and /usr/bin/ */
        cp1 = linep;
        cp2 = "/usr/bin/";
        while(*cp1 = *cp2++)
            cp1++;
        cp2 = t[dcom];
        while(*cp1++ = *cp2++);
        execv(linep+4, t+dcom);  /* /bin/cmd */
        execv(linep, t+dcom);    /* /usr/bin/cmd */
```

Try to execute the command. If not found, try `/bin/` and `/usr/bin/` prefixes.

16.10.3. Pipelines

```
    case tfil:
        f = t[dflg];
        pipe(pv);
        t1 = t[dlef];
        t1[dflg] =| fpou | (f&(fpin|fint));
        execute(t1, pf1, pv);
        t1 = t[drit];
        t1[dflg] =| fpin | (f&(fpou|fint|fand));
        execute(t1, pv, pf2);
        return;
```

Create a pipe, execute left side with output to pipe, right side with input from pipe.

16.10.4. Command Lists

```
    case tlst:
        f = t[dflg]&fint;
        if(t1 = t[dlef])
            t1[dflg] =| f;
        execute(t1);
        if(t1 = t[drit])
            t1[dflg] =| f;
        execute(t1);
        return;
```

Execute left side, then right side.

16.11. I/O Redirection

The shell implements redirection by manipulating file descriptors:

```
/* Input: cmd < file */
close(0);
open(t[dlef], 0);     /* Opens as fd 0 */

/* Output: cmd > file */
i = creat(t[drit], 0666);
close(1);
dup(i);               /* Duplicates to fd 1 */
close(i);

/* Append: cmd >> file */
i = open(t[drit], 1);
seek(i, 0, 2);        /* Seek to end */
close(1);
dup(i);
```

> **Classic UNIX Technique.** The `close(1); dup(i);` idiom works because `dup()` always returns the lowest available file descriptor. After `close(1)` frees stdout, `dup(i)` duplicates `i` into slot 1—effectively redirecting stdout to the file. The order is critical: `dup(i); close(1);` would duplicate to some random slot, then close stdout, achieving nothing useful. This "lowest available" guarantee is baked into the UNIX design and appears throughout early system code.

16.12. Pipes

```
/* cmd1 | cmd2 */
pipe(pv);              /* Create pipe: pv[0]=read, pv[1]=write */

/* Execute cmd1 with stdout → pipe */
t1[dflg] =| fpou;
execute(t1, pf1, pv);

/* Execute cmd2 with stdin ← pipe */
t1[dflg] =| fpin;
execute(t1, pv, pf2);
```

In the child processes:

```
/* Writer (cmd1) */
close(1);
dup(pv[1]);            /* stdout → pipe write end */
close(pv[0]);
close(pv[1]);

/* Reader (cmd2) */
close(0);
dup(pv[0]);            /* stdin ← pipe read end */
close(pv[0]);
close(pv[1]);
```

16.13. Background Processes

```
if(**p == '&') {
    t1[dflg] =| fand|fint;
}

/* In execute(): */
if((f&fand) != 0) {
    prn(i);
    prs("\n");         /* Print PID */
    return;            /* Don't wait */
}
```

Background processes:

- Print PID and return immediately
- Have stdin redirected from `/dev/null`
- Ignore interrupt signals (`fint` flag)

16.14. Glob (Wildcard Expansion)

```
scan(t, &tglob);
if(gflg) {
    t[dspr] = "/etc/glob";
    execv(t[dspr], t+dspr);
}
```

If wildcards (`*`, `?`, `[`) are found, the shell execs `/etc/glob` to expand them. Glob is a separate program that does pattern matching.

16.15. Signal Handling

```
/* In main(): Login shells ignore signals */
if(**v == '-') {
    signal(quit, 1);
    signal(intr, 1);
}

/* In execute(): Restore signals for children */
if((f&fint) == 0 && setintr) {
    signal(intr, 0);
    signal(quit, 0);
}
```

Interactive shells ignore interrupt/quit so they survive Ctrl-C. Child processes have signals restored unless running in background.

16.16. Summary

The shell in ~800 lines:

- **Lexer** (`word`): Tokenizes input, handles quotes and escapes
- **Parser** (`syntax`, `syn1`, `syn2`, `syn3`): Builds syntax tree
- **Executor** (`execute`): Runs commands via fork/exec
- **Built-ins**: `chdir`, `shift`, `wait`, `login`, `:`
- **Redirection**: Via file descriptor manipulation
- **Pipes**: Via `pipe()` system call
- **Background**: Fork without wait, print PID
- **Glob**: Delegated to `/etc/glob`

16.17. Key Design Points

1. **Simplicity**: No complex features—just the essentials.
2. **Fork/exec model**: Every external command is a new process.
3. **File descriptors**: Redirection works because children inherit fds.
4. **Recursive descent**: Clean, readable parser structure.
5. **External glob**: Pattern matching is a separate program.

16.18. Experiments

1. **Add a built-in**: Implement pwd as a built-in command.
2. **Trace execution**: Add printf to see the syntax tree structure.
3. **Pipeline depth**: Create long pipelines and observe process creation.
4. **Signal behavior**: Compare Ctrl-C handling in foreground vs background.

16.19. Further Reading

- Chapter 5: Process Management — `fork`, `exec`, `wait` internals
- Chapter 7: Traps and System Calls — Signal mechanism
- Chapter 10: File I/O — File descriptor operations

Next: Chapter 17 — Core Utilities

17. Chapter 17: Core Utilities

17.1. Overview

UNIX comes with dozens of small, focused utilities that work together through pipes and files. This chapter examines four representative programs spanning the spectrum from tiny assembly routines to substantial C applications. Together they demonstrate the UNIX philosophy: simple tools that do one thing well.

17.2. Source Files

File	Lines	Language	Purpose
`usr/source/s1/echo.c`	10	C	Print arguments
`usr/source/s1/cat.s`	65	Assembly	Concatenate files
`usr/source/s1/cp.c`	80	C	Copy files
`usr/source/s1/ls.c`	428	C	List directory

17.3. Prerequisites

- Chapter 2: PDP-11 Architecture (for assembly)
- Chapter 7: System Calls (`open`, `read`, `write`, `stat`)

17.4. echo — The Simplest Utility

```
main(argc, argv)
int argc;
char *argv[];
{
    int i;

    argc--;
    for(i=1; i<=argc; i++)
        printf("%s%c", argv[i], i==argc? '\n': ' ');
}
```

Ten lines. Print each argument separated by spaces, end with newline. This is the essence of UNIX utilities—do exactly one thing.

Usage:

```
$ echo hello world
hello world
```

17.5. cat — Concatenate Files (Assembly)

`cat` is written in assembly for efficiency—it's used constantly.[1]

```
/ cat -- concatenate files

    mov  (sp)+,r5        / r5 = argc
    tst  (sp)+           / skip argv[0]
    mov  $obuf,r2        / r2 = output buffer pointer
    cmp  r5,$1
    beq  3f              / no args: read stdin

loop:
    dec  r5
    ble  done            / no more files
    mov  (sp)+,r0        / r0 = next filename
    cmpb (r0),$'-
    bne  2f
    clr  fin             / "-" means stdin
    br   3f
2:
    mov  r0,0f
    sys  open; 0:..; 0   / open file
    bes  loop            / error: skip file
    mov  r0,fin
```

[1] The branch targets like `3f` and `2b` are local labels in PDP-11 assembly. The number (1-9) is the label name; `f` means "forward" (find the next occurrence ahead) and `b` means "backward" (find the previous occurrence behind). So `beq 3f` branches to the next `3:` label, and `br 1b` loops back to the previous `1:`. This lets you reuse simple numeric labels without inventing unique names.

The main loop opens each file argument (or uses stdin for "-").

```
3:
    mov  fin,r0
    sys  read; ibuf; 512.  / read up to 512 bytes
    bes  3f                / error or EOF
    mov  r0,r4             / r4 = bytes read
    beq  3f                / EOF
    mov  $ibuf,r3
4:
    movb (r3)+,r0          / get byte
    jsr  pc,putc           / output it
    dec  r4
    bne  4b                / loop until done
    br   3b                / read more
3:
    mov  fin,r0
    beq  loop              / stdin: don't close
    sys  close
    br   loop
```

Read 512 bytes at a time, output byte by byte through `putc`.

```
putc:
    movb r0,(r2)+          / store in output buffer
    cmp  r2,$obuf+512.
    blo  1f                / buffer not full
    mov  $1,r0
    sys  write; obuf; 512. / flush buffer
    mov  $obuf,r2
1:
    rts  pc
```

Output is buffered—write 512 bytes at a time for efficiency.

```
done:
    sub  $obuf,r2
    beq  1f                / nothing to flush
    mov  r2,0f
    mov  $1,r0
    sys  write; obuf; 0:.. / flush remaining
1:
    sys  exit

    .bss
ibuf: .=.+512.             / input buffer
obuf: .=.+512.             / output buffer
fin:  .=.+2                / current input fd
```

Key points: - Buffered I/O for performance - Handles multiple files - "-" means stdin - ~65 lines of tight assembly

17.6. cp — Copy Files

```
main(argc,argv)
char **argv;
{
    int buf[256];
    int fold, fnew, n, ct;
    char *p1, *p2, *bp;
    int mode;

    if(argc != 3) {
        write(1, "Usage: cp oldfile newfile\n", 26);
        exit(1);
    }
```

Basic argument checking.

```
    if((fold = open(argv[1], 0)) < 0) {
        write(1, "Cannot open old file.\n", 22);
        exit(1);
    }
    fstat(fold, buf);
    mode = buf[2];          /* Preserve file mode */
```

Open source file and get its mode (permissions).

```
    if((fnew = creat(argv[2], mode)) < 0){
        stat(argv[2], buf);
        if((buf[2] & 060000) == 040000) {
            /* Destination is a directory */
            p1 = argv[1];
            p2 = argv[2];
            bp = buf;
            while(*bp++ = *p2++);
            bp[-1] = '/';
            p2 = bp;
            while(*bp = *p1++)
                if(*bp++ == '/')
                    bp = p2;
            /* Now buf = "dir/basename" */
            if((fnew = creat(buf, mode)) < 0) {
                write(1, "Cannot creat new file.\n", 23);
                exit(1);
            }
        } else {
            write(1, "Cannot creat new file.\n", 23);
            exit(1);
        }
    }
```

Create destination. If it's a directory, append the source filename.

```
    while(n = read(fold, buf, 512)) {
        if(n < 0) {
            write(1, "Read error\n", 11);
            exit(1);
        }
        if(write(fnew, buf, n) != n){
            write(1, "Write error.\n", 13);
            exit(1);
        }
    }
    exit(0);
}
```

Copy loop: read 512 bytes, write them, repeat until EOF.

Key points: - Preserves file permissions - Handles “cp file dir/” case - 512-byte block copies - Error checking at each step

17.7. ls — List Directory

`ls` is the most complex utility here at 428 lines. It demonstrates:

- Option parsing
- Directory reading
- stat() for file info
- Sorting
- Formatted output

17.7.1. Option Parsing

```
main(argc, argv)
char **argv;
{
    if (--argc > 0 && *argv[1] == '-') {
        argv++;
        while (****argv) switch (**argv) {
        case 'a':
            aflg++;         /* Show hidden files */
            continue;
        case 's':
            sflg++;         /* Show sizes */
            statreq++;
            continue;
        case 'l':
            lflg++;         /* Long format */
            statreq++;
            uidfil = open("/etc/passwd", 0);
```

```
            continue;
        case 'r':
            rflg = -1;      /* Reverse sort */
            continue;
        case 't':
            tflg++;         /* Sort by time */
            statreq++;
            continue;
        /* ... more options ... */
        }
    }
```

Classic UNIX option style: single dash, single letters, can be combined (`ls -la`).

17.7.2. Reading Directories

```
readdir(dir)
char *dir;
{
    static struct {
        int   dinode;
        char  dname[14];
    } dentry;

    if (fopen(dir, &inf) < 0) {
        printf("%s unreadable\n", dir);
        return;
    }
    for(;;) {
        p = &dentry;
        for (j=0; j<16; j++)
            *p++ = getc(&inf);      /* Read 16-byte entry */
        if (dentry.dinode==0        /* Empty slot */
         || aflg==0 && dentry.dname[0]=='.')
            continue;               /* Skip hidden */
        if (dentry.dinode == -1)
            break;                  /* End of directory */
        ep = gstat(makename(dir, dentry.dname), 0);
        /* ... store entry ... */
    }
}
```

Directories are just files with 16-byte entries (2-byte inode + 14-byte name).

17.7.3. Getting File Information

```
gstat(file, argfl)
char *file;
{
    struct ibuf statb;
    register struct lbuf *rep;

    if (stat(file, &statb)<0) {
        printf("%s not found\n", file);
        return(0);
    }
    rep->lnum = statb.inum;
    rep->lflags = statb.iflags;
    rep->luid = statb.iuid;
    rep->lsize = statb.isize;
    rep->lmtime[0] = statb.imtime[0];
    rep->lmtime[1] = statb.imtime[1];
    /* ... */
}
```

The `stat()` system call fills in file metadata: type, permissions, owner, size, times.

17.7.4. Formatting Output

```
pentry(ap)
struct lbuf *ap;
{
    if (iflg)
        printf("%5d ", p->lnum);       /* Inode number */
    if (lflg) {
        pmode(p->lflags);              /* -rwxr-xr-x */
        printf("%2d ", p->lnl);        /* Link count */
        /* ... owner, size, date ... */
    }
    printf("%.14s\n", p->lname);       /* Filename */
}
```

17.7.5. Permission Display

```
int m0[] { 3, DIR, 'd', BLK, 'b', CHR, 'c', '-'};
int m1[] { 1, ROWN, 'r', '-' };
int m2[] { 1, WOWN, 'w', '-' };
int m3[] { 2, SUID, 's', XOWN, 'x', '-' };
/* ... */

pmode(aflag)
```

```
{
    register int **mp;
    flags = aflag;
    for (mp = &m[0]; mp < &m[10];)
        select(*mp++);
}
```

Clever table-driven approach to print -rwxr-xr-x style permissions.

17.8. System Call Patterns

17.8.1. Error Handling

```
/* Typical pattern */
if((fd = open(file, 0)) < 0) {
    write(2, "error message\n", n);
    exit(1);
}
```

17.8.2. Reading Files

```
/* Block-at-a-time */
while((n = read(fd, buf, 512)) > 0) {
    /* process n bytes in buf */
}
```

17.8.3. Writing Output

```
/* Direct write */
write(1, string, length);

/* Using printf (links with library) */
printf("%s\n", string);
```

17.9. Assembly vs C Trade-offs

Aspect	Assembly (cat)	C (ls)
Size	~300 bytes	~4KB
Speed	Optimal	Good
Maintenance	Difficult	Easy
Portability	PDP-11 only	Somewhat portable

Assembly was used for:

- Frequently-used utilities (cat, echo)
- Performance-critical code
- Tiny programs where every byte mattered

C was used for:

- Complex logic (ls, cp with directory handling)
- Maintainability requirements
- Less performance-critical utilities

17.10. The UNIX Philosophy

These utilities embody key principles:

1. **Do one thing well**: `cat` concatenates, `cp` copies, `echo` echoes
2. **Text streams**: Programs read/write text, enabling pipes
3. **Composability**: `cat file | grep pattern | wc -l`
4. **No unnecessary output**: Silent on success
5. **Meaningful exit codes**: 0 for success, non-zero for error

17.11. Summary

- **echo**: 10 lines—print arguments
- **cat**: 65 lines assembly—buffered file concatenation
- **cp**: 80 lines—copy with directory handling
- **ls**: 428 lines—full-featured directory listing

17.12. Experiments

1. **Add an option**: Add `-n` to `cat` to number lines.
2. **Trace system calls**: Count `read`/`write` calls in `cat` for various file sizes.
3. **Benchmark**: Compare `cat` performance with a C version.
4. **Extend ls**: Add color coding for file types.

17.13. Further Reading

- Chapter 16: The Shell — How utilities are invoked
- Chapter 7: System Calls — The `open`/`read`/`write` interface
- Chapter 11: Path Resolution — How files are found

Next: Chapter 18 — The C Compiler

18. Chapter 18: The C Compiler

18.1. Overview

The C compiler that compiled UNIX was itself written in C—a bootstrapping achievement that demonstrated the language's power. In roughly 4,000 lines across two passes, it translates C source code into PDP-11 assembly language. This chapter examines the compiler's architecture, showing how a complete language implementation fits in such compact form.

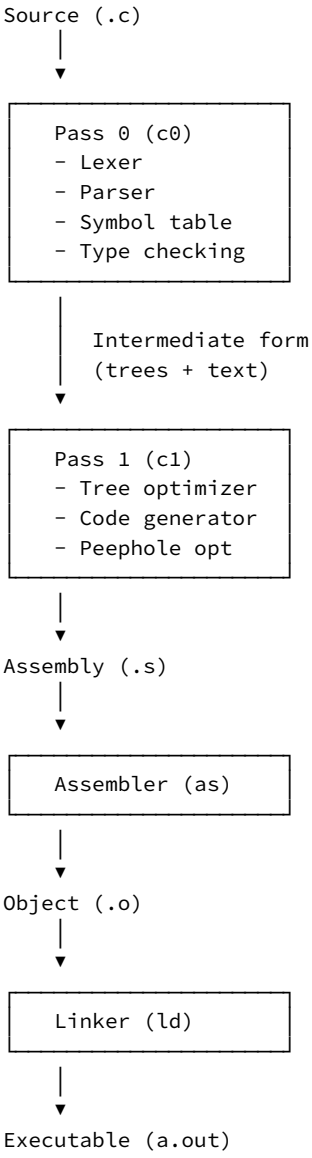
18.2. Source Files

File	Lines	Purpose
`usr/c/c00.c`	744	Lexer, symbol table, expression parser
`usr/c/c01.c`	~600	Statements, declarations
`usr/c/c02.c`	~400	Expression building
`usr/c/c03.c`	~300	Type checking
`usr/c/c04.c`	~200	Output utilities
`usr/c/c10.c`	942	Code generator main
`usr/c/c11.c`	~500	Code generation helpers
`usr/c/c12.c`	~400	More code generation

18.3. Prerequisites

- Chapter 2: PDP-11 Architecture (target machine)
- Understanding of basic compiler concepts

18.4. Two-Pass Architecture



18.5. Pass 0: Lexical Analysis

18.5.1. Keyword Table

```
struct kwtab {
    char *kwname;
    int  kwval;
} kwtab[] {
    "int",      INT,
    "char",     CHAR,
    "float",    FLOAT,
    "double",   DOUBLE,
    "struct",   STRUCT,
    "auto",     AUTO,
    "extern",   EXTERN,
    "static",   STATIC,
    "register", REG,
    "goto",     GOTO,
    "return",   RETURN,
    "if",       IF,
    "while",    WHILE,
    "else",     ELSE,
    "switch",   SWITCH,
    "case",     CASE,
    "break",    BREAK,
    "continue", CONTIN,
    "do",       DO,
    "default",  DEFAULT,
    "for",      FOR,
    "sizeof",   SIZEOF,
    0,          0,
};
```

All 22 C keywords in one table. Keywords are installed in the symbol table at startup with class KEYWC.

18.5.2. The Lexer: symbol()

```
symbol() {
    register c;
    register char *sp;

    if (peeksym>=0) {
        c = peeksym;
        peeksym = -1;
        return(c);
    }
    /* ... get character ... */
loop:
    switch(ctab[c]) {
```

```
    case SPACE:
        c = getchar();
        goto loop;

    case NEWLN:
        line++;
        c = getchar();
        goto loop;

    case PLUS:
        return(subseq(c,PLUS,INCBEF));  /* + or ++ */

    case MINUS:
        return(subseq(c,subseq('>',MINUS,ARROW),DECBEF));

    case ASSIGN:
        /* Handle =, ==, +=, etc. */
        ...

    case DIVIDE:
        if (subseq('*',1,0))
            return(DIVIDE);
        /* Skip comment */
        ...

    case LETTER:
        /* Collect identifier */
        while(ctab[c]==LETTER || ctab[c]==DIGIT) {
            if (sp<symbuf+ncps) *sp++ = c;
            c = getchar();
        }
        csym = lookup();
        if (csym->hclass==KEYWC)
            return(KEYW);
        return(NAME);
    }
}
```

The lexer uses a character classification table (`ctab[]`) for fast dispatch. Multi-character tokens like `++`, `->`, and `==` are handled by `subseq()`.

18.5.3. Symbol Table

```
struct hshtab {
    char  name[ncps];    /* Symbol name (8 chars) */
    char  hclass;        /* Storage class */
    char  htype;         /* Type encoding */
    int   hoffset;       /* Offset or value */
    int   dimp;          /* Dimension pointer */
};

struct hshtab *lookup()
```

```
{
    int ihash;
    register struct hshtab *rp;

    /* Hash the symbol name */
    ihash = 0;
    for (sp=symbuf; sp<symbuf+ncps;)
        ihash =+ *sp++ & 0177;
    rp = &hshtab[ihash%hshsiz];

    /* Linear probe for match or empty slot */
    while (*(np = rp->name)) {
        for (sp=symbuf; sp<symbuf+ncps;)
            if ((*np++&0177) != *sp++)
                goto no;
        return(rp);         /* Found */
    no:
        if (++rp >= &hshtab[hshsiz])
            rp = hshtab;    /* Wrap around */
    }
    /* Install new symbol */
    ...
    return(rp);
}
```

Simple hash table with linear probing. Symbol names limited to 8 characters.

18.6. Pass 0: Parsing

18.6.1. Expression Parser

The expression parser uses operator precedence parsing:

```
tree() {
    int *op, opst[SSIZE], *pp, prst[SSIZE];
    register int andflg, o;

    op = opst;
    pp = prst;
    *op = SEOF;
    *pp = 06;         /* Lowest precedence */
    andflg = 0;       /* Expecting operand? */

advanc:
    switch (o=symbol()) {

    case NAME:
        /* Push operand */
        *cp++ = block(2,NAME,cs->htype,...);
        goto tand;
```

```
    case CON:
        *cp++ = block(1,CON,INT,0,cval);
        goto tand;

tand:
        if (andflg)
            goto syntax;    /* Two operands in a row */
        andflg = 1;
        goto advanc;

    case PLUS:
    case MINUS:
        if (!andflg) {
            o = NEG;        /* Unary minus */
        }
        andflg = 0;
        goto oponst;
    }

oponst:
    p = (opdope[o]>>9) & 077;    /* Get precedence */
    /* Reduce higher-precedence operators on stack */
    while (p <= *pp) {
        /* Pop and build tree node */
        build(*op--);
        --pp;
    }
    /* Push this operator */
    ***op = o;
    ***pp = p;
    goto advanc;
}
```

The opdope[] table encodes operator properties: precedence, associativity, whether binary or unary.

18.6.2. Tree Building

```
build(op) {
    register struct tnode *p1, *p2;

    p2 = *--cp;
    if (opdope[op] & BINARY)
        p1 = *--cp;
    /* Type check and convert */
    ...
    /* Build node */
    *cp++ = block(2, op, type, 0, p1, p2);
}
```

18.7. Pass 1: Code Generation

18.7.1. Main Loop

```
main(argc, argv)
char *argv[];
{
    while ((c=getc(ascbuf)) > 0) {
        if(c=='#') {
            /* Expression tree follows */
            tree = getw(binbuf);
            table = tabtab[getw(binbuf)];
            tree = optim(tree);     /* Optimize */
            rcexpr(tree, table, 0); /* Generate code */
        } else
            putchar(c);             /* Copy through */
    }
}
```

Pass 1 reads the intermediate file, optimizes expression trees, and generates code.

18.7.2. Table-Driven Code Generation

```
char *match(tree, table, nrleft)
struct tnode *tree;
struct table *table;
{
    op = tree->op;
    d1 = dcalc(tree->tr1, nrleft);  /* Difficulty of left */
    d2 = dcalc(tree->tr2, nrleft);  /* Difficulty of right */

    /* Find matching table entry */
    for (; table->op==op; table++)
        for (opt = table->tabp; opt->tabdeg1!=0; opt++) {
            if (d1 > (opt->tabdeg1&077))
                continue;
            if (d2 > (opt->tabdeg2&077))
                continue;
            /* Check type compatibility */
            if (notcompat(tree->tr1, opt->tabtyp1))
                continue;
            return(opt);    /* Match found */
        }
    return(0);
}
```

Code templates are stored in tables. The generator matches tree patterns against templates and emits the corresponding code.

18.7.3. Code Template Example

A template might specify:

```
ADD instruction:
  Operand 1: register (difficulty ≤ 12)
  Operand 2: any (difficulty ≤ 12)
  Output: "add A2,R\n"
```

The `cexpr()` function interprets template strings:

- A — address of operand 1
- B — address of operand 2
- R — result register
- M — instruction mnemonic

18.7.4. Optimization

```
optim(tree) {
    if ((dope&COMMUTE)!=0) {
        /* Reorder commutative operations */
        tree = acommute(tree);
    }
    if (tree->tr2->op==CON && op==MINUS) {
        /* Convert x-c to x+(-c) */
        tree->op = PLUS;
        tree->tr2->value = -tree->tr2->value;
    }
    if (tree->tr1->op==CON && tree->tr2->op==CON) {
        /* Fold constants */
        const(op, &tree->tr1->value, tree->tr2->value);
        return(tree->tr1);
    }
    ...
}
```

Optimizations include:

- Constant folding
- Strength reduction (multiply by power of 2 → shift)
- Common subexpression handling
- Register allocation

18.8. Type System

Types are encoded in a small integer:

```
/* Base types */
#define INT     0
#define CHAR    1
#define FLOAT   2
#define DOUBLE  3
#define STRUCT  4

/* Type modifiers (in higher bits) */
#define PTR     010     /* Pointer to */
#define FUNC    020     /* Function returning */
#define ARRAY   030     /* Array of */
```

So `int *x` has type PTR | INT, and `int (*f)()` has type PTR | FUNC | INT.

18.9. Generated Code Example

Source:

```
int x, y;
x = y + 1;
```

Generated assembly:

```
.globl _x
.globl _y
.comm _x,2
.comm _y,2
        mov     _y,r0
        inc     r0
        mov     r0,_x
```

18.10. Compilation Flow

```
$ cc foo.c

1. Preprocessor (cpp)
   foo.c → /tmp/ctm1

2. Pass 0 (c0)
   /tmp/ctm1 → /tmp/ctm2 (text) + /tmp/ctm3 (trees)

3. Pass 1 (c1)
```

```
   /tmp/ctm2 + /tmp/ctm3 → /tmp/ctm4.s

4. Assembler (as)
   /tmp/ctm4.s → foo.o

5. Linker (ld)
   foo.o + libc.a → a.out
```

18.11. Summary

The C compiler in ~4,000 lines:

- **Lexer**: Character classification, symbol table with hashing
- **Parser**: Operator precedence for expressions, recursive descent for statements
- **Type system**: Compact encoding of C's type hierarchy
- **Code generator**: Table-driven pattern matching
- **Optimizer**: Constant folding, strength reduction

18.12. Key Design Points

1. **Two passes**: Separates analysis from code generation cleanly.
2. **Table-driven**: Code templates make the generator compact and maintainable.
3. **Bootstrapped**: The compiler compiles itself—proving the language works.
4. **PDP-11 targeted**: Code generation exploits the architecture's features.
5. **No preprocessor**: `#include` handled by a separate `cpp` program.

18.13. Experiments

1. **Trace compilation**: Add printf to see token stream and parse trees.
2. **New operator**: Add a simple operator and trace through both passes.
3. **Optimization effect**: Compare generated code with/without optimization.
4. **Type encoding**: Decode type integers to understand the encoding.

18.14. Further Reading

- Chapter 2: PDP-11 Architecture — Target instruction set

- Chapter 19: The Assembler — Next stage in compilation
- Original C Reference Manual by Dennis Ritchie

Next: Chapter 19 — The Assembler

19. Chapter 19: The Assembler

19.1. Overview

The UNIX assembler translates PDP-11 assembly language into executable object code. Written in assembly language itself, it demonstrates a classic two-pass design: pass 1 builds the symbol table, pass 2 generates code. The assembler is the final stage in the compilation pipeline before linking.

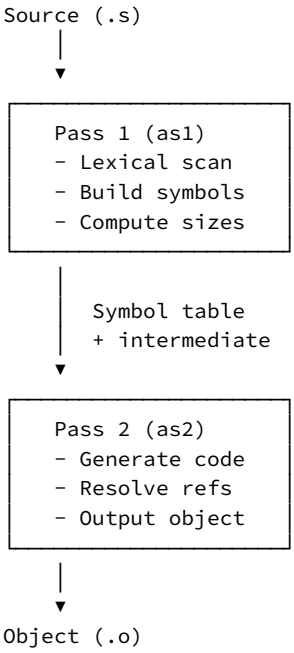
19.2. Source Files

File	Purpose
`usr/source/s1/as11.s` - `as19.s`	Pass 1 (lexer, parser, symbol table)
`usr/source/s1/as21.s` - `as29.s`	Pass 2 (code generation)

19.3. Prerequisites

- Chapter 2: PDP-11 Architecture (instruction set)
- Chapter 18: C Compiler (produces assembly input)

19.4. Two-Pass Architecture



19.4.1. Why Two Passes?

Forward references are the problem:

```
        jmp     later       / Can't know address yet
        ...
later:  mov     r0,r1       / Defined here
```

Pass 1 computes the address of `later`. Pass 2 uses it.

19.5. Pass 1 Structure

```
/ as11.s - Main entry

start:
    jsr     pc,assem        / Main assembly loop
    movb    pof,r0
    sys     write; outbuf; 512.
    ...
    sys     exec; fpass2; ...   / Chain to pass 2
```

Pass 1 processes the source, building the symbol table, then exec's pass 2.

19.5.1. Assembly Loop

```
assem:
    jsr     pc,readop       / Get next token
    cmp     r4,$5           / End of file?
    beq     1f
    jsr     pc,checkeos     / End of statement?
    br      assem

1:  rts     pc
```

19.5.2. Symbol Table

Symbols are stored in a simple table:

```
Entry format:
  Bytes 0-7: Symbol name (8 chars, null-padded)
  Byte 8:    Type/flags
  Bytes 9-10: Value (address or constant)
```

Types include:

- Undefined (forward reference)
- Absolute (constant)
- Text segment
- Data segment
- BSS segment
- External

19.5.3. Location Counter

The assembler tracks the current address with `.` (dot):

```
.text               / Switch to text segment
        mov r0,r1   / . = 0, instruction at 0
        add r2,r3   / . = 2, instruction at 2
.data               / Switch to data segment
foo:    .word 42    / . = 0 in data, foo = 0
```

19.6. Instruction Encoding

PDP-11 instructions are encoded in 16-bit words:

Single Operand (CLR, INC, TST, etc.):

```
 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|        opcode         |  mode  |     reg     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Double Operand (MOV, ADD, CMP, etc.):

```
 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  opcode   | src mode  |src reg| dst mode  |dst|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Branch (BEQ, BNE, BR, etc.):

```
 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|        opcode         |        offset         |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Jump/Subroutine (JSR, JMP):

```
 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|     opcode      | reg |  mode  |   dst reg   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

19.6.1. Addressing Modes

```
Mode  Syntax          Meaning
0     Rn              Register
1     (Rn)            Register indirect
2     (Rn)+           Autoincrement
3     @(Rn)+          Autoincrement indirect
4     -(Rn)           Autodecrement
5     @-(Rn)          Autodecrement indirect
6     X(Rn)           Index
7     @X(Rn)          Index indirect
```

Special cases with PC (R7):

```
Mode 2: #n      Immediate (literal follows)
Mode 3: @#n     Absolute
Mode 6: n       Relative
Mode 7: @n      Relative indirect
```

19.7. Pass 2 Structure

Pass 2 reads the intermediate output and symbol table from pass 1:

```
/ as21.s - Pass 2 main

start2:
    / Read symbol table from temp file
    mov     $usymtab,r1
    sys     read; ...

loop2:
    jsr     pc,readop       / Get opcode
    jsr     pc,opline       / Process operands
    jsr     pc,outw         / Output word
    br      loop2
```

19.7.1. Code Generation

For each instruction:

1. Look up opcode in table
2. Parse operands
3. Encode addressing modes
4. Output instruction word(s)

```
opline:
    mov     optab(r0),r1    / Get opcode template
    jsr     pc,addres       / Parse first operand
    swab    r3              / Shift to source field
    bis     r3,r1
    jsr     pc,addres       / Parse second operand
    bis     r3,r1           / Add to destination field
    mov     r1,outbuf       / Store result
    rts     pc
```

19.7.2. Relocation

The assembler generates relocation information for the linker:

```
Object file format:
  Header:
    - Magic number
    - Text size
    - Data size
    - BSS size
    - Symbol table size
    - Entry point
```

```
    - Relocation size

  Text segment
  Data segment
  Text relocation
  Data relocation
  Symbol table
```

Relocation entries indicate which words need adjustment when the program is loaded at a different address.

19.8. Directives

```
.globl  sym         / Make symbol global
.text               / Switch to text segment
.data               / Switch to data segment
.bss                / Switch to BSS segment
.byte   1,2,3       / Output bytes
.word   1,2,3       / Output words
.even               / Align to word boundary
.=.+n               / Advance location counter
```

19.9. Assembly Language Features

19.9.1. Labels

```
foo:    mov     r0,r1       / Define label
        jmp     foo         / Reference label
```

19.9.2. Local Labels

```
1:      mov     r0,r1
        bne     1b          / Back to 1:
        br      1f          / Forward to next 1:
1:      clr     r0
```

`1b` means “label 1, searching backward”; `1f` means forward.

19.9.3. Expressions

```
        mov     $foo+4,r0
        .word   bar-baz
        .=.+100
```

The assembler evaluates expressions involving +, −, *, /, &, |, symbols, and constants.

19.10. Example Assembly

Source:

```
.globl  _main
.text
_main:
        mov     $1,r0
        sys     write; 1f; 2f-1f
        clr     r0
        sys     exit
.data
1:      <hello\n>
2:
```

Object code (hex):

```
15c0 0001       mov $1,r0
8904 000c 0006  sys write; L1; 6
0a00            clr r0
8901            sys exit
6865 6c6c 6f0a  "hello\n"
```

19.11. Error Handling

```
filerr:
    mov     (r5)+,r4        / Get filename
    mov     $1,r0
    sys     write; ...      / Print filename
    sys     write; "?\n"; 2 / Print "?"
```

Errors are terse: typically just the filename and “?”. Debug by examining the source line.

19.12. Summary

The UNIX assembler:

- **Two passes**: Build symbols, then generate code
- **Self-hosting**: Written in the assembly language it processes
- **PDP-11 specific**: Encodes all addressing modes and instructions
- **Relocation**: Generates position-independent code for linker
- **Minimal**: No macros, simple expression evaluation

19.13. Key Design Points

1. **Simplicity**: No macro processor—that's separate (`m4`).
2. **Two passes**: Clean separation of symbol resolution from code generation.
3. **Tables**: Instruction encodings stored in lookup tables.
4. **Temp files**: Pass 1 writes intermediate data for pass 2.
5. **exec chain**: Pass 1 directly exec's pass 2, avoiding shell overhead.

19.14. Experiments

1. **Trace assembly**: Add print statements to see symbol table construction.
2. **New instruction**: Add a pseudo-instruction like `.ascii`.
3. **Object dump**: Write a program to decode a.out format.
4. **Forward reference**: Trace how a forward branch is resolved.

19.15. Further Reading

- Chapter 2: PDP-11 Architecture — Target instruction set
- Chapter 18: C Compiler — Producer of assembly input
- PDP-11 Processor Handbook — Instruction encoding details

Part VI.

Appendices

20. Appendix A: System Call Reference

This appendix provides a complete reference for all system calls implemented in UNIX Fourth Edition. Each entry includes the system call number, C library interface, arguments, return value, and a brief description.

20.1. Overview

UNIX v4 implements 35 active system calls (out of 64 slots in the system call table). System calls are invoked via the `sys` instruction on the PDP-11, which causes a trap to kernel mode. The kernel looks up the system call number in `sysent[]` (defined in `usr/sys/ken/sysent.c`) and dispatches to the appropriate handler.

20.1.1. Argument Passing

- Arguments are passed in the `u.u_arg[]` array in the user structure
- Return values are placed in registers r0 (and sometimes r1)
- Errors are indicated by setting `u.u_error` to an error code

20.1.2. Error Codes

Code	Name	Meaning
1	EPERM	Not owner
2	ENOENT	No such file or directory
3	ESRCH	No such process
4	EINTR	Interrupted system call
5	EIO	I/O error
6	ENXIO	No such device or address

Code	Name	Meaning
7	E2BIG	Argument list too long
8	ENOEXEC	Exec format error
9	EBADF	Bad file number
10	ECHILD	No children
11	EAGAIN	No more processes
12	ENOMEM	Not enough memory
13	EACCES	Permission denied
14	EFAULT	Bad address
15	ENOTBLK	Block device required
16	EBUSY	Mount device busy
17	EEXIST	File exists
18	EXDEV	Cross-device link
19	ENODEV	No such device
20	ENOTDIR	Not a directory
21	EISDIR	Is a directory
22	EINVAL	Invalid argument
23	ENFILE	File table overflow
24	EMFILE	Too many open files
25	ENOTTY	Not a typewriter
26	ETXTBSY	Text file busy
27	EFBIG	File too large
28	ENOSPC	No space left on device
29	ESPIPE	Illegal seek

20.2. System Call Reference

20.2.1. 0 - indir (indirect system call)

```
syscall(number, args...)
```

Description: Execute an indirect system call. The first argument is the system call number, followed by that call's arguments. Primarily used for implementing system call stubs.

Implementation: `nullsys()` - does nothing in UNIX v4

20.2.2. 1 - exit

```
exit(status)
int status;
```

Arguments: - `status` - Exit status (passed in r0, shifted left 8 bits)

Returns: Does not return

Description: Terminate the calling process. All open file descriptors are closed, the current directory inode is released, and the process enters the zombie state until its parent calls `wait()`. The exit status is saved for retrieval by the parent.

Implementation: `rexit()` in `usr/sys/ken/sys1.c`

20.2.3. 2 - fork

```
pid = fork()
int pid;
```

Arguments: None

Returns: - In parent: PID of child process - In child: PID of parent process (note: parent's PID, not 0) - On error: -1

Description: Create a new process. The child is an exact copy of the parent, including all open file descriptors and current file positions. The child inherits the parent's memory image but has a separate copy.

Notes: - Fork returns twice - once in each process - The parent's PC is advanced by 2 so it skips to the instruction after fork - Process times (user, system) are reset to 0 in the child

Implementation: `fork()` in `usr/sys/ken/sys1.c`

20.2.4. 3 - read

```
nread = read(fd, buffer, nbytes)
int fd;
char *buffer;
int nbytes;
```

Arguments: - `fd` - File descriptor (in r0) - `buffer` - Address of buffer to read into - `nbytes` - Number of bytes to read

Returns: Number of bytes actually read, or -1 on error

Description: Read data from a file. For regular files, reads from the current file position and advances it. For pipes, blocks if no data available. For device files, behavior depends on the device driver.

Implementation: `read()` in `usr/sys/ken/sys2.c`

20.2.5. 4 - write

```
nwritten = write(fd, buffer, nbytes)
int fd;
char *buffer;
int nbytes;
```

Arguments: - `fd` - File descriptor (in r0) - `buffer` - Address of buffer to write from - `nbytes` - Number of bytes to write

Returns: Number of bytes actually written, or -1 on error

Description: Write data to a file. For regular files, writes at the current position and advances it. For pipes, blocks if the pipe buffer is full.

Implementation: `write()` in `usr/sys/ken/sys2.c`

20.2.6. 5 - open

```
fd = open(name, mode)
char *name;
int mode;
```

Arguments: - `name` - Pathname of file to open - `mode` - Access mode: 0=read, 1=write, 2=read/write

Returns: File descriptor, or -1 on error

Description: Open an existing file for reading and/or writing. The file must exist (use `creat()` to create files). Access permissions are checked based on the mode.

Implementation: `open()` in `usr/sys/ken/sys2.c`

20.2.7. 6 - close

```
close(fd)
int fd;
```

Arguments: - `fd` - File descriptor (in r0)

Returns: 0 on success, -1 on error

Description: Close an open file descriptor. Releases the file table entry and decrements the inode reference count.

Implementation: `close()` in `usr/sys/ken/sys2.c`

20.2.8. 7 - wait

```
pid = wait()
int pid;
/* Status returned in r1 */
```

Arguments: None

Returns: - r0: PID of terminated child - r1: Exit status of child - -1 if no children exist

Description: Wait for a child process to terminate. If a child has already terminated (zombie state), return immediately with its status. Otherwise, block until a child terminates.

Implementation: `wait()` in `usr/sys/ken/sys1.c`

20.2.9. 8 - creat

```
fd = creat(name, mode)
char *name;
int mode;
```

Arguments: - `name` - Pathname of file to create - `mode` - Permission bits (masked with 07777)

Returns: File descriptor opened for writing, or -1 on error

Description: Create a new file or truncate an existing file. If the file exists, it is truncated to zero length and opened for writing. If it doesn't exist, a new file is created with the specified permissions (modified by umask).

Implementation: `creat()` in `usr/sys/ken/sys2.c`

20.2.10. 9 - link

```
link(name1, name2)
char *name1, *name2;
```

Arguments: - `name1` - Pathname of existing file - `name2` - New pathname (link) to create

Returns: 0 on success, -1 on error

Description: Create a hard link. The new pathname refers to the same inode as the existing file. Both pathnames must be on the same filesystem. The link count of the inode is incremented. Only superuser can link directories.

Errors: - EEXIST - name2 already exists - EXDEV - Cross-device link attempted

Implementation: `link()` in `usr/sys/ken/sys2.c`

20.2.11. 10 - unlink

```
unlink(name)
char *name;
```

Arguments: - `name` - Pathname to remove

Returns: 0 on success, -1 on error

Description: Remove a directory entry. Decrements the link count of the inode. When the link count reaches zero and no processes have the file open, the file's blocks are freed. Only superuser can unlink directories.

Implementation: `unlink()` in `usr/sys/ken/sys3.c`

20.2.12. 11 - exec

```
exec(name, argv)
char *name;
char *argv[];
```

Arguments: - `name` - Pathname of executable file - `argv` - Array of argument strings, NULL-terminated

Returns: Does not return on success, -1 on error

Description: Execute a program. The current process image is replaced with the new program. Open file descriptors remain open (unless close-on-exec is set). Signals are reset to default.

Executable Format: - Word 0: Magic number (0407 or 0410) - Word 1: Text size - Word 2: Data size - Word 3: BSS size

Implementation: `exec()` in `usr/sys/ken/sys1.c`

20.2.13. 12 - chdir

```
chdir(dirname)
char *dirname;
```

Arguments: - `dirname` - Pathname of new current directory

Returns: 0 on success, -1 on error

Description: Change the current working directory. The specified pathname must be a directory and the process must have execute permission.

Implementation: `chdir()` in `usr/sys/ken/sys3.c`

20.2.14. 13 - time

```
time()
/* Returns time in r0 (high) and r1 (low) */
```

Arguments: None

Returns: Current time in seconds since epoch (Jan 1, 1970) as a 32-bit value split across r0 (high 16 bits) and r1 (low 16 bits)

Description: Get the current system time.

Implementation: `gtime()` in `usr/sys/ken/sys3.c`

20.2.15. 14 - mknod

```
mknod(name, mode, dev)
char *name;
int mode;
int dev;
```

Arguments: - `name` - Pathname for new node - `mode` - File type and permissions - `dev` - Device number (major/minor) for device files

Returns: 0 on success, -1 on error

Description: Create a special file (device node). Only superuser can create device nodes. The mode specifies the file type:

- 040000 (IFDIR) - Directory
- 020000 (IFCHR) - Character device

- 060000 (IFBLK) - Block device

Implementation: `mknod()` in `usr/sys/ken/sys2.c`

20.2.16. 15 - chmod

```
chmod(name, mode)
char *name;
int mode;
```

Arguments: - `name` - Pathname of file - `mode` - New permission bits

Returns: 0 on success, -1 on error

Description: Change file permissions. Only the file owner or superuser can change permissions.

Implementation: `chmod()` in `usr/sys/ken/sys3.c`

20.2.17. 16 - chown

```
chown(name, owner)
char *name;
int owner;
```

Arguments: - `name` - Pathname of file - `owner` - New owner UID

Returns: 0 on success, -1 on error

Description: Change file owner. Only superuser can change file ownership. When a non-superuser changes ownership, the setuid bit is cleared.

Implementation: `chown()` in `usr/sys/ken/sys3.c`

20.2.18. 17 - break (sbrk)

```
brk(addr)
char *addr;
```

Arguments: - `addr` - New end of data segment

Returns: 0 on success, -1 on error

Description: Change the program break (end of data segment). Used to allocate or deallocate memory for the heap. The break address is rounded up to a 64-byte boundary.

Implementation: `sbreak()` in `usr/sys/ken/sys1.c`

20.2.19. 18 - stat

```
stat(name, buf)
char *name;
struct stat *buf;
```

Arguments: - `name` - Pathname of file - `buf` - Buffer for stat structure

Returns: 0 on success, -1 on error

Description: Get file status. Fills in a stat structure with information about the file.

Stat Structure (36 bytes):

```
struct stat {
    int  st_dev;    /* Device */
    int  st_ino;    /* Inode number */
    int  st_mode;   /* Mode and permissions */
    char st_nlink;  /* Link count */
    char st_uid;    /* Owner UID */
    char st_gid;    /* Group GID */
    char st_size0;  /* Size high byte */
    int  st_size1;  /* Size low word */
    int  st_addr[8];/* Block addresses */
    int  st_atime[2]; /* Access time */
    int  st_mtime[2]; /* Modification time */
};
```

Implementation: `stat()` in `usr/sys/ken/sys3.c`

20.2.20. 19 - seek (lseek)

```
seek(fd, offset, whence)
int fd;
int offset;
int whence;
```

Arguments: - `fd` - File descriptor (in r0) - `offset` - Position offset - `whence` - Base for offset: - 0: From beginning - 1: From current position - 2: From end - 3: Like 0, but offset is in 512-byte blocks - 4: Like 1, but offset is in 512-byte blocks - 5: Like 2, but offset is in 512-byte blocks

Returns: 0 on success, -1 on error

Description: Reposition file offset. Cannot seek on pipes.

Implementation: `seek()` in `usr/sys/ken/sys2.c`

20.2.21. 20 - (unimplemented)

Reserved for getpid (not implemented in v4)

20.2.22. 21 - mount

```
mount(special, dir, rwflag)
char *special;
char *dir;
int rwflag;
```

Arguments: - `special` - Pathname of block device - `dir` - Pathname of mount point - `rwflag` - 0=read-/write, 1=read-only

Returns: 0 on success, -1 on error

Description: Mount a filesystem. The block device is mounted on the specified directory. Only superuser can mount filesystems.

Implementation: `smount()` in `usr/sys/ken/sys3.c`

20.2.23. 22 - umount

```
umount(special)
char *special;
```

Arguments: - `special` - Pathname of block device to unmount

Returns: 0 on success, -1 on error

Description: Unmount a filesystem. Fails if any files on the filesystem are open.

Implementation: `sumount()` in `usr/sys/ken/sys3.c`

20.2.24. 23 - setuid

```
setuid(uid)
int uid;
```

Arguments: - `uid` - New user ID (in r0, low byte only)

Returns: 0 on success, -1 on error

Description: Set user ID. If the caller is superuser or the new UID matches the real UID, both effective and real UID are changed.

Implementation: `setuid()` in `usr/sys/ken/sys3.c`

20.2.25. 24 - getuid

```
uid = getuid()
/* Returns real UID in low byte of r0, effective UID in high byte */
```

Arguments: None

Returns: Real UID in r0 low byte, effective UID in r0 high byte

Description: Get user ID. Returns both the real and effective user IDs.

Implementation: `getuid()` in `usr/sys/ken/sys3.c`

20.2.26. 25 - stime

```
stime()
/* Time passed in r0 (high) and r1 (low) */
```

Arguments: Time value in r0/r1

Returns: 0 on success, -1 on error

Description: Set system time. Only superuser can set the time.

Implementation: `stime()` in `usr/sys/ken/sys3.c`

20.2.27. 26-27 - (unimplemented)

Reserved

20.2.28. 28 - fstat

```
fstat(fd, buf)
int fd;
struct stat *buf;
```

Arguments: - `fd` - File descriptor (in r0) - `buf` - Buffer for stat structure

Returns: 0 on success, -1 on error

Description: Get status of an open file. Like stat(), but operates on a file descriptor instead of a pathname.

Implementation: `fstat()` in `usr/sys/ken/sys3.c`

20.2.29. 29 - (unimplemented)

Reserved

20.2.30. 30 - smdate

```
smdate(name)
char *name;
```

Arguments: - `name` - Pathname (1 word in sysent)

Description: Set modification date. This syscall is defined in the system call table but implemented as `nullsys()`, meaning it does nothing and returns immediately. Likely a placeholder for planned functionality that was never completed in v4.

Implementation: `nullsys()` (no-op)

20.2.31. 31 - stty

```
stty(fd, argp)
int fd;
int *argp;
```

Arguments: - `fd` - File descriptor of terminal (in r0) - `argp` - Pointer to 3-word structure

Returns: 0 on success, -1 on error

Description: Set terminal parameters. The structure contains:

- Word 0: Input modes
- Word 1: Output modes
- Word 2: Erase and kill characters

Implementation: `stty()` in `usr/sys/dmr/tty.c`

20.2.32. 32 - gtty

```
gtty(fd, argp)
int fd;
int *argp;
```

Arguments: - `fd` - File descriptor of terminal (in r0) - `argp` - Pointer to 3-word buffer

Returns: 0 on success, -1 on error

Description: Get terminal parameters.

Implementation: `gtty()` in `usr/sys/dmr/tty.c`

20.2.33. 33 - (unimplemented)

Reserved

20.2.34. 34 - nice

```
nice(incr)
int incr;
```

Arguments: - `incr` - Priority adjustment (in r0)

Returns: Previous nice value

Description: Change process priority. Positive values decrease priority (make process nicer). Values are clamped to 0-20. Only superuser can decrease the nice value (increase priority).

Implementation: `nice()` in `usr/sys/ken/sys3.c`

20.2.35. 35 - sleep

```
sleep(seconds)
int seconds;
```

Arguments: - `seconds` - Number of seconds to sleep (in r0)

Returns: 0

Description: Suspend execution for the specified number of seconds.

Implementation: `sslep()` in `usr/sys/ken/sys2.c`

20.2.36. 36 - sync

```
sync()
```

Arguments: None

Returns: 0

Description: Flush all filesystem buffers to disk. Writes all modified buffer cache blocks and superblocks.

Implementation: `sync()` in `usr/sys/ken/sys3.c`

20.2.37. 37 - kill

```
kill(pid, sig)
int pid;
int sig;
```

Arguments: - `pid` - Process ID to signal (in r0) - `sig` - Signal number

Returns: 0 on success, -1 on error

Description: Send a signal to a process. The sender must have the same controlling terminal as the target, or be superuser.

Implementation: `kill()` in `usr/sys/ken/sys4.c`

20.2.38. 38 - switch (getcsw)

```
csw = getcsw()
```

Arguments: None

Returns: Console switch register value

Description: Read the console switch register. Used for debugging and system configuration on the PDP-11.

Implementation: `getswit()` in `usr/sys/ken/sys3.c`

20.2.39. 39-40 - (unimplemented)

Reserved

20.2.40. 41 - dup

```
newfd = dup(fd)
int fd;
```

Arguments: - `fd` - File descriptor to duplicate (in r0)

Returns: New file descriptor, or -1 on error

Description: Duplicate a file descriptor. Returns the lowest available file descriptor number that refers to the same open file.

Implementation: `dup()` in `usr/sys/ken/sys3.c`

20.2.41. 42 - pipe

```
pipe()
/* Returns read fd in r0, write fd in r1 */
```

Arguments: None

Returns: - r0: Read end file descriptor - r1: Write end file descriptor - -1 on error

Description: Create a pipe. Data written to the write end can be read from the read end. Used for inter-process communication.

Implementation: `pipe()` in `usr/sys/dmr/pipe.c`

20.2.42. 43 - times

```
times(buffer)
int *buffer;
```

Arguments: - `buffer` - Pointer to 6-word buffer

Returns: 0

Description: Get process times. Fills buffer with:

- Word 0-1: User time of current process
- Word 2-3: System time of current process
- Word 4-5: Sum of children's user and system times

Implementation: `times()` in `usr/sys/ken/sys4.c`

20.2.43. 44 - profil

```
profil(buff, bufsiz, offset, scale)
int *buff;
int bufsiz;
int offset;
int scale;
```

Arguments: - `buff` - Buffer for profile counters - `bufsiz` - Size of buffer - `offset` - PC offset for profiling - `scale` - Scaling factor for PC

Returns: 0

Description: Enable execution profiling. The kernel periodically samples the PC and increments a counter in the buffer based on where the process was executing.

Implementation: `profil()` in `usr/sys/ken/sys4.c`

20.2.44. 45 - (unimplemented)

Reserved (tiu - was used for TIU hardware)

20.2.45. 46 - setgid

```
setgid(gid)
int gid;
```

Arguments: - `gid` - New group ID (in r0, low byte only)

Returns: 0 on success, -1 on error

Description: Set group ID. Like setuid, but for group ID.

Implementation: `setgid()` in `usr/sys/ken/sys3.c`

20.2.46. 47 - getgid

```
gid = getgid()
/* Returns real GID in low byte of r0, effective GID in high byte */
```

Arguments: None

Returns: Real GID in r0 low byte, effective GID in r0 high byte

Description: Get group ID.

Implementation: `getgid()` in `usr/sys/ken/sys3.c`

20.2.47. 48 - signal

```
old = signal(sig, func)
int sig;
int (*func)();
```

Arguments: - `sig` - Signal number - `func` - Handler: 0=default, 1=ignore, or address of handler function

Returns: Previous handler value

Description: Set signal handler. Cannot change handler for signal 9 (KILL).

Signals in UNIX v4:

Number	Name	Default Action
1	SIGHUP	Terminate
2	SIGINT	Terminate
3	SIGQIT	Core dump
4	SIGINS	Core dump
5	SIGTRC	Core dump
6	SIGIOT	Core dump
7	SIGEMT	Core dump
8	SIGFPT	Core dump
9	SIGKIL	Terminate (cannot catch)
10	SIGBUS	Core dump
11	SIGSEG	Core dump
12	SIGSYS	Core dump
13	SIGPIP	Terminate

Implementation: `ssig()` in `usr/sys/ken/sys4.c`

20.2.48. 49-63 - (unimplemented)

Reserved for future use

20.3. System Call Summary Table

#	Name	Args	Description
0	indir	0	Indirect system call
1	exit	0	Terminate process
2	fork	0	Create child process
3	read	2	Read from file
4	write	2	Write to file
5	open	2	Open file
6	close	0	Close file
7	wait	0	Wait for child
8	creat	2	Create file
9	link	2	Create hard link
10	unlink	1	Remove directory entry
11	exec	2	Execute program
12	chdir	1	Change directory
13	time	0	Get system time
14	mknod	3	Create device node
15	chmod	2	Change permissions
16	chown	2	Change owner
17	break	1	Change data segment size
18	stat	2	Get file status
19	seek	2	Seek in file
21	mount	3	Mount filesystem
22	umount	1	Unmount filesystem
23	setuid	0	Set user ID
24	getuid	0	Get user ID
25	stime	0	Set system time
28	fstat	1	Get open file status
30	smdate	1	Set modification date (stub)

#	Name	Args	Description
31	stty	1	Set terminal params
32	gtty	1	Get terminal params
34	nice	0	Set priority
35	sleep	0	Sleep
36	sync	0	Flush buffers
37	kill	1	Send signal
38	switch	0	Read console switches
41	dup	0	Duplicate fd
42	pipe	0	Create pipe
43	times	1	Get process times
44	profil	4	Execution profiling
46	setgid	0	Set group ID
47	getgid	0	Get group ID
48	signal	2	Set signal handler

20.4. See Also

- Chapter 7: Traps and System Calls
- `usr/sys/ken/sysent.c` - System call table
- `usr/sys/ken/sys1.c` - Process system calls
- `usr/sys/ken/sys2.c` - File I/O system calls
- `usr/sys/ken/sys3.c` - File system calls
- `usr/sys/ken/sys4.c` - Miscellaneous system calls

21. Appendix B: File Formats

This appendix documents the binary file formats used in UNIX Fourth Edition, including the a.out executable format, archive format, and the on-disk filesystem layout.

21.1. a.out Executable Format

The `a.out` format is the executable file format used by UNIX v4. The name comes from "assembler output," as it's the default output file produced by the assembler.

21.1.1. Header Structure

Every a.out file begins with an 8-word (16-byte) header:

```
Offset  Size    Field           Description
------  ----    -----           -----------
0       2       a_magic         Magic number (0407 or 0410)
2       2       a_text          Size of text segment (bytes)
4       2       a_data          Size of initialized data (bytes)
6       2       a_bss           Size of uninitialized data (bytes)
8       2       a_syms          Size of symbol table (bytes)
10      2       a_entry         Entry point (usually 0)
12      2       a_unused        Unused
14      2       a_flag          Relocation flags
```

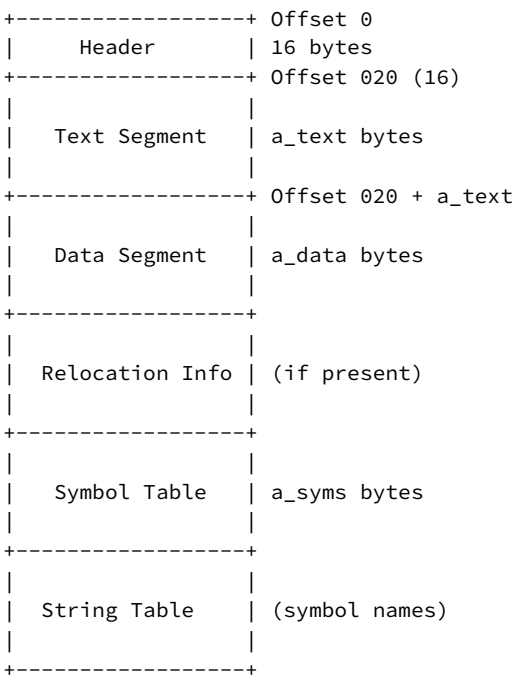
21.1.2. Magic Numbers

Magic	Octal	Description
0407	0407	Normal executable (text not read-only)
0410	0410	Read-only text (shared text segment)

0407 Format: - Text and data are combined into a single segment - Cannot share text between processes - Simpler memory layout

0410 Format: - Separate text and data segments - Text is read-only and can be shared between processes - Used for larger programs to save memory

21.1.3. File Layout



21.1.4. Memory Layout at Execution

When `exec()` loads a 0407 file:

```
Address 0
+------------------+
|   Text + Data    | Combined segment
+------------------+
|       BSS        | Zero-initialized
+------------------+
|                  |
|      Stack       | Grows downward
+------------------+ Address 64KB
```

When `exec()` loads a 0410 file:

```
Address 0
+------------------+
|      Text        | Read-only, shared
+------------------+
|      Data        | Per-process
+------------------+
|       BSS        | Zero-initialized
+------------------+
|                  |
|      Stack       | Grows downward
+------------------+ Address 64KB
```

21.1.5. Relocation

When the file contains relocation information (not stripped), the relocation entries follow the data segment. Each relocation entry is 8 bytes:

```
struct reloc {
    int    r_vaddr;    /* Address to relocate */
    int    r_symndx;   /* Symbol index or segment */
    int    r_type;     /* Relocation type */
};
```

21.1.6. Symbol Table

Symbol table entries are 12 bytes each:

```
struct sym {
    char   s_name[8];  /* Symbol name (truncated to 8 chars) */
    int    s_type;     /* Type and storage class */
    int    s_value;    /* Value (address) */
};
```

Symbol types:

Value	Meaning
0	Undefined
1	Absolute
2	Text
3	Data
4	BSS
037	File name

21.1.7. Example: Examining an a.out File

```
$ od -o a.out | head
0000000 000407 000062 000004 000000
0000020 ...
```

Breaking down the header:

- 000407 = Magic (normal executable)
- 000062 = Text size (50 bytes)
- 000004 = Data size (4 bytes)
- 000000 = BSS size (0 bytes)

21.2. Archive Format (.a files)

Archives are used by the linker to package multiple object files into a single library. The `ar` command creates and manipulates archives.

21.2.1. Archive Structure

```
+------------------+
|  Archive Header  | Magic string
+------------------+
|  Member Header   |
+------------------+
|  Member Content  | (object file)
+------------------+
|  Member Header   |
+------------------+
|  Member Content  |
+------------------+
       ...
```

21.2.2. Archive Magic

Archives begin with the magic string:

```
!<arch>\n
```

(8 bytes: `!<arch>` followed by newline)

21.2.3. Member Header

Each archive member is preceded by a 60-byte header:

```
struct ar_hdr {
    char ar_name[16];   /* Member name, blank padded */
    char ar_date[12];   /* Modification time (decimal) */
    char ar_uid[6];     /* User ID (decimal) */
    char ar_gid[6];     /* Group ID (decimal) */
    char ar_mode[8];    /* File mode (octal) */
    char ar_size[10];   /* Size in bytes (decimal) */
    char ar_fmag[2];    /* Magic: "`\n" */
};
```

Notes: - All fields are ASCII, not binary - Names longer than 16 characters are truncated - The `ar_fmag` field contains the string `` `\n `` (backquote, newline) - Member content follows immediately after the header - If member size is odd, a padding newline is added

21.2.4. Symbol Table (__.SYMDEF)

If the archive contains a symbol table (created by `ranlib`), it appears as the first member named `__.SYMDEF`:

```
struct ranlib {
    int  ran_off;       /* Offset of symbol name in string table */
    int  ran_foff;      /* File offset of archive member */
};
```

21.3. Filesystem Format

The UNIX v4 filesystem uses a simple and elegant on-disk layout. This section describes the physical structure of data on disk.

21.3.1. Disk Layout

```
Block 0: Boot Block
Block 1: Superblock
Blocks 2-N: Inode List
Blocks N+1-end: Data Blocks
```

21.3.2. Boot Block (Block 0)

The first 512-byte block is reserved for the boot loader. On a bootable disk, it contains code to load and execute the kernel. On non-bootable filesystems, it may be unused.

21.3.3. Superblock (Block 1)

The superblock contains filesystem metadata. It is defined in `usr/sys/filsys.h`:

```
struct filsys {
    int   s_isize;      /* Size of inode list in blocks */
    int   s_fsize;      /* Size of filesystem in blocks */
    int   s_nfree;      /* Number of entries in s_free */
    int   s_free[100];  /* Free block list cache */
    int   s_ninode;     /* Number of entries in s_inode */
    int   s_inode[100]; /* Free inode cache */
    char  s_flock;      /* Lock for free list manipulation */
    char  s_ilock;      /* Lock for inode list manipulation */
    char  s_fmod;       /* Superblock modified flag */
    char  s_ronly;      /* Read-only flag */
    int   s_time[2];    /* Last modification time */
};
```

Total size: 412 bytes

Field descriptions:

Field	Description
s_isize	Number of blocks in inode list (starting at block 2)
s_fsize	Total blocks in filesystem
s_nfree	Count of block numbers in s_free[] (0-100)
s_free[]	Cache of free block numbers
s_ninode	Count of inode numbers in s_inode[]
s_inode[]	Cache of free inode numbers
s_flock	Prevents concurrent free list modification
s_ilock	Prevents concurrent inode list modification
s_fmod	Set when superblock needs writing
s_ronly	Set for read-only mounted filesystems
s_time	Time of last modification

21.3.4. Free Block List

Free blocks are managed using a linked list of block groups. The superblock caches up to 100 free block numbers in `s_free[]`. When this cache is exhausted, `s_free[0]` contains a pointer to a block that contains another 100 free block numbers, and so on.

```
Superblock s_free[]:
+----+----+----+----+...+----+
|ptr | b1 | b2 | b3 |...| b99|
+----+----+----+----+...+----+
  |
  v
Block containing more free block numbers:
+----+----+----+----+...+----+
|ptr | b1 | b2 | b3 |...| b99|
+----+----+----+----+...+----+
  |
  v
  ...
```

21.3.5. Inode List

Inodes are stored sequentially starting at block 2. Each inode is 32 bytes, so 16 inodes fit per 512-byte block.

Inode number to block calculation:

```
block = (inode_number + 31) / 16;
offset = ((inode_number + 31) % 16) * 32;
```

Note: Inode numbers start at 1 (inode 0 is unused). The +31 accounts for this offset.

21.3.6. On-Disk Inode Structure

Each inode is 32 bytes on disk:

```
Offset  Size    Field       Description
------  ----    -----       -----------
0       2       di_mode     Type, permissions, flags
2       1       di_nlink    Number of hard links
3       1       di_uid      Owner user ID
4       1       di_gid      Owner group ID
5       1       di_size0    File size high byte
6       2       di_size1    File size low word
8       16      di_addr[8]  Block addresses (8 x 16-bit)
24      4       di_atime    Access time
28      4       di_mtime    Modification time
```

Total: 32 bytes

21.3.7. Inode Mode Field

The `di_mode` field encodes file type and permissions:

```
 15   14   13   12   11   10    9    8    7    6    5    4    3    2    1    0
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
| A  |  type   | L  | U  | G  |    | r  | w  | x  | r  | w  | x  | r  | w  | x  |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
  |    \__(2)__/  |    |    |       \___(3)____/   \___(3)____/   \___(3)____/
  |         |     |    |    |            |              |              |
  |         |     |    |    |          owner          group          other
  |         |     |    |    |
  |         |     |    |    +---- ISGID  (02000)   set-group-ID
  |         |     |    +--------- ISUID  (04000)   set-user-ID
  |         |     +-------------- ILARG  (010000)  large file
  |         +-------------------- IFMT   (060000)  file type
  +------------------------------ IALLOC (0100000) allocated
```

Bits	Value	Meaning
15	IALLOC (0100000)	Inode is allocated
14-13	IFMT (060000)	File type
	00	Regular file
	01	Character device (020000)
	10	Directory (040000)
	11	Block device (060000)
12	ILARG (010000)	Large file (indirect blocks)
11	ISUID (04000)	Set user ID on execution

Bits	Value	Meaning
10	ISGID (02000)	Set group ID on execution
9	(01000)	Unused[1]
8-6		Owner permissions (rwx)
5-3		Group permissions (rwx)
2-0		Other permissions (rwx)

21.3.8. Block Addressing

The `di_addr[]` array holds 8 block addresses as 16-bit integers[2] (16 bytes total):

```
di_addr storage (16 bytes = 8 addresses x 2 bytes each):

 byte:   0    1    2    3    4    5        14   15
       +----+----+----+----+----+----+    +----+----+
       | addr[0] | addr[1] | addr[2] |....| addr[7] |
       +----+----+----+----+----+----+    +----+----+
         blk 0     blk 1     blk 2          blk 7
```

Small files (ILARG not set): - di_addr[0-7] point directly to data blocks - Maximum file size: 8 * 512 = 4KB

Large files (ILARG set): - di_addr[0-6] point to indirect blocks - Each indirect block contains 256 block numbers (512 bytes / 2 bytes per number) - di_addr[7] points to a doubly-indirect block - Maximum file size: (7 * 256 + 256 * 256) * 512 = approximately 33MB

```
Small file:
di_addr[0] --> data block
di_addr[1] --> data block
...

Large file:
di_addr[0] --> indirect block --> 256 data blocks
di_addr[1] --> indirect block --> 256 data blocks
...
di_addr[7] --> double indirect --> 256 indirect blocks --> 65536 data blocks
```

21.3.9. Directory Format

Directories are regular files with a specific internal format. Each directory entry is 16 bytes:

[1] Bit 9 became ISVTX (sticky bit) in UNIX Seventh Edition (1979). On executables, it kept the program text in swap space after exit. On directories, it restricted file deletion to owners.

[2] UNIX v4 stores block addresses as simple 16-bit integers with no packing. Starting in v6, addresses were stored in a packed 3-byte (24-bit) format using the `l3tol()` and `ltol3()` conversion functions to support larger filesystems. Some documentation incorrectly attributes this 3-byte packing to v4.

```c
struct direct {
    int   d_ino;        /* Inode number (2 bytes) */
    char  d_name[14];   /* Filename (14 bytes, null-padded) */
};
```

Notes: - Filenames are limited to 14 characters - A `d_ino` of 0 indicates an empty (deleted) directory entry - The `.` and `..` entries are always present

21.3.10. Special Inodes

Inode	Purpose
1	Root directory (/)
2	Usually /lost+found or reserved

21.3.11. Device Files

For device files (character or block special files), the `di_addr[0]` field contains the device number:

```
di_addr[0]:
+--------+--------+
| major  | minor  |
+--------+--------+
  8 bits   8 bits
```

- Major number: Identifies the device driver
- Minor number: Identifies the specific device instance

21.3.12. Example Filesystem Calculation

For a filesystem on an RK05 disk (4872 blocks):

```
Block 0:      Boot block
Block 1:      Superblock
Blocks 2-41:  Inode list (40 blocks = 640 inodes)
Blocks 42-4871: Data blocks (4830 blocks)
```

Settings in superblock:

```
s_isize = 40      (blocks in inode list)
s_fsize = 4872    (total filesystem blocks)
```

21.4. Object File Format

Object files (produced by the assembler before linking) use the same basic structure as executables, with different magic numbers and additional relocation information.

21.4.1. Object File Header

Same as a.out header, but with different magic values:

- 0407: Relocatable object with relocation info
- 0410: Pure (read-only text) relocatable object

21.4.2. Relocation Entries

Object files contain relocation entries that tell the linker how to adjust addresses when combining multiple object files:

```
struct reloc {
    int r_address;    /* Address to patch */
    int r_symbolnum;  /* Symbol or segment reference */
    int r_type;       /* Type of relocation */
};
```

21.5. Summary

UNIX v4's file formats are characterized by:

1. **Simplicity** - Minimal headers, straightforward layouts
2. **Efficiency** - Packed formats to save space
3. **Fixed limits** - 14-character filenames, 64KB address space

These constraints reflect the limited resources of the PDP-11 era while still providing the foundation for a fully functional operating system.

21.6. See Also

- Chapter 9: Inodes and Superblock
- Chapter 11: Path Resolution
- Chapter 18: The C Compiler
- Appendix C: PDP-11 Quick Reference

22. Appendix C: PDP-11 Quick Reference

This appendix provides a concise reference for the PDP-11 architecture as used by UNIX Fourth Edition. It covers registers, addressing modes, instruction set, and other essential details for understanding the source code.

22.1. Registers

The PDP-11 has eight 16-bit general-purpose registers:

Register	Name	UNIX Usage
r0	General	Return value, temporary
r1	General	Return value (high word), temporary
r2	General	Temporary, preserved across calls
r3	General	Temporary, preserved across calls
r4	General	Temporary, preserved across calls
r5	General	Frame pointer (fp), preserved
r6	sp	Stack pointer
r7	pc	Program counter

Calling Convention: - r0-r1: Used for return values, caller-saved - r2-r4: Callee-saved (must be preserved by called function) - r5: Frame pointer, callee-saved - r6/sp: Stack pointer - r7/pc: Program counter

22.1.1. Processor Status Word (PSW)

Located at address 0177776:

```
Bit  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |CM|PM|  |RS|  |  |  |  |PR|PR|PR| T| N| Z| V| C|
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Bits	Name	Description
15-14	CM	Current mode (00=kernel, 11=user)
13-12	PM	Previous mode
11	RS	Register set (not used in 11/40)
7-5	PR	Processor priority (0-7)
4	T	Trace trap
3	N	Negative (result < 0)
2	Z	Zero (result = 0)
1	V	Overflow
0	C	Carry

22.2. Addressing Modes

The PDP-11 uses a flexible addressing mode system. Each operand uses 6 bits: 3 for mode and 3 for register.

22.2.1. Mode Encoding

Mode	Syntax	Name	Description
0	Rn	Register	Operand is in register
1	(Rn)	Register deferred	Register contains address
2	(Rn)+	Autoincrement	Use address, then increment register

Mode	Syntax	Name	Description
3	@(Rn)+	Autoincrement deferred	Double indirection with increment
4	-(Rn)	Autodecrement	Decrement register, then use address
5	@-(Rn)	Autodecrement deferred	Double indirection with decrement
6	X(Rn)	Index	Address is register + offset X
7	@X(Rn)	Index deferred	Double indirection with index

22.2.2. PC-Relative Modes (using r7)

Mode	Syntax	Name	Description
27	#n	Immediate	Operand follows instruction
37	@#n	Absolute	Address follows instruction
67	n	Relative	PC-relative address
77	@n	Relative deferred	Indirect through PC-relative address

22.2.3. Examples

```
mov  r0,r1        ; Register to register
mov  (r0),r1      ; Memory[r0] to r1
mov  (r0)+,r1     ; Memory[r0] to r1, r0 += 2
mov  -(sp),r1     ; Push: sp -= 2, Memory[sp] to r1
mov  4(r5),r1     ; Memory[r5+4] to r1
mov  $100,r0      ; Immediate: 100 to r0
mov  *$addr,r0    ; Absolute: Memory[addr] to r0
```

22.3. Instruction Set

22.3.1. Data Movement

Instruction	Operation	Description
mov src,dst	dst = src	Move word
movb src,dst	dst = src	Move byte
clr dst	dst = 0	Clear word
clrb dst	dst = 0	Clear byte
com dst	dst = ~dst	Complement (bitwise NOT)
comb dst	dst = ~dst	Complement byte
neg dst	dst = -dst	Negate (two’s complement)
negb dst	dst = -dst	Negate byte
inc dst	dst++	Increment
incb dst	dst++	Increment byte
dec dst	dst–	Decrement
decb dst	dst–	Decrement byte
swab dst		Swap bytes in word

22.3.2. Arithmetic

Instruction	Operation	Description
add src,dst	dst += src	Add
sub src,dst	dst -= src	Subtract
cmp src,dst	src - dst	Compare (set flags only)
cmpb src,dst	src - dst	Compare bytes
tst src	src - 0	Test (set flags only)
tstb src	src - 0	Test byte
adc dst	dst += C	Add carry
sbc dst	dst -= C	Subtract carry

Instruction	Operation	Description
mul src,reg	reg = reg * src	Multiply (result in reg:reg+1)
div src,reg	reg = reg:reg+1 / src	Divide
ash shift,reg		Arithmetic shift
ashc shift,reg		Arithmetic shift combined

22.3.3. Logical

Instruction	Operation	Description
bit src,dst	src & dst	Bit test (set flags only)
bitb src,dst	src & dst	Bit test byte
bic src,dst	dst &= ~src	Bit clear
bicb src,dst	dst &= ~src	Bit clear byte
bis src,dst	dst	= src
bisb src,dst	dst	= src
xor reg,dst	dst ^= reg	Exclusive OR

22.3.4. Rotate/Shift

Instruction	Operation	Description
asr dst	dst »= 1	Arithmetic shift right
asrb dst	dst »= 1	Arithmetic shift right byte
asl dst	dst «= 1	Arithmetic shift left
aslb dst	dst «= 1	Arithmetic shift left byte
ror dst		Rotate right through carry
rorb dst		Rotate right byte
rol dst		Rotate left through carry
rolb dst		Rotate left byte

22.3.5. Branches

All branches are PC-relative with an 8-bit signed offset (range: -128 to +127 words).

Instruction	Condition	Description
br addr	Always	Branch always
bne addr	Z=0	Branch if not equal
beq addr	Z=1	Branch if equal
bpl addr	N=0	Branch if plus
bmi addr	N=1	Branch if minus
bvc addr	V=0	Branch if overflow clear
bvs addr	V=1	Branch if overflow set
bcc addr	C=0	Branch if carry clear
bcs addr	C=1	Branch if carry set
bge addr	N^V=0	Branch if greater or equal (signed)
blt addr	N^V=1	Branch if less than (signed)
bgt addr	Z	(N^V)=0
ble addr	Z	(N^V)=1
bhi addr	C	Z=0
blos addr	C	Z=1

22.3.6. Jumps and Subroutines

Instruction	Operation	Description
jmp dst	pc = dst	Jump
jsr reg,dst	tmp=dst; -(sp)=reg; reg=pc; pc=tmp	Jump to subroutine
rts reg	pc=reg; reg=(sp)+	Return from subroutine
mark n		Mark stack (for complex returns)
sob reg,addr	if (–reg) br addr	Subtract one and branch

Common calling patterns:

```
; Call function
jsr  pc,func      ; Push old PC, jump to func
; ... or ...
jsr  r5,func      ; Push old r5, jump (used with mark)

; Return
rts  pc           ; Pop return address into PC
; ... or ...
rts  r5           ; Restore r5, return
```

22.3.7. Stack Operations

The stack grows downward (toward lower addresses). SP (r6) always points to the top item.

```
; Push
mov  r0,-(sp)     ; Decrement SP, store r0

; Pop
mov  (sp)+,r0     ; Load r0, increment SP

; Push multiple
mov  r2,-(sp)
mov  r3,-(sp)
mov  r4,-(sp)

; Pop multiple
mov  (sp)+,r4
mov  (sp)+,r3
mov  (sp)+,r2
```

22.3.8. Traps and Interrupts

Instruction	Vector	Description
trap n	034	Trap (n in low byte of instruction)
emt n	030	Emulator trap
bpt	014	Breakpoint trap
iot	020	I/O trap
rti		Return from interrupt
rtt		Return from trap (trace trap)
halt		Halt processor

Instruction	Vector	Description
wait		Wait for interrupt
reset		Reset UNIBUS

22.3.9. System Calls (UNIX-specific)

```
sys  n            ; System call number n
; Arguments in words following sys instruction
; or in registers depending on call
```

Example:

```
sys  write        ; sys 4
fout              ; file descriptor
buf               ; buffer address
count             ; byte count
```

22.3.10. Condition Code Operations

Instruction	Description
clc	Clear C
clv	Clear V
clz	Clear Z
cln	Clear N
ccc	Clear all flags
sec	Set C
sev	Set V
sez	Set Z
sen	Set N
scc	Set all flags
nop	No operation

22.4. Memory Map

22.4.1. PDP-11/40 with 28K words (56KB)

```
Address (octal)  Contents
000000-000377    Interrupt vectors
000400-037777    User text/data (when in user mode)
040000-157777    User text/data continued
160000-167777    Kernel/User stack or I/O page
170000-177777    I/O page and device registers
```

22.4.2. I/O Page (160000-177777)

Address	Device/Register
177560	Console receiver status
177562	Console receiver data
177564	Console transmitter status
177566	Console transmitter data
177570	Console switch register
177572	Memory management registers
177776	Processor status word (PSW)

22.5. Interrupt Vectors

Vector (octal)	Interrupt Source
000	Reserved
004	Bus timeout, illegal instruction
010	Illegal instruction
014	BPT (breakpoint)
020	IOT
024	Power fail

Vector (octal)	Interrupt Source
030	EMT
034	TRAP
060	Console input
064	Console output
100	Line clock
104	Programmable clock
200	RK disk

Each vector location contains two words:

- Vector + 0: New PC
- Vector + 2: New PSW

22.6. Assembly Syntax (UNIX as)

22.6.1. Directives

Directive	Description
.text	Switch to text segment
.data	Switch to data segment
.bss	Switch to BSS segment
.globl name	Declare global symbol
.byte val,…	Emit bytes
.even	Align to word boundary
.comm name,size	Define common block

22.6.2. Expressions

```
label:            ; Define label at current address
.               ; Current location counter
label + 4         ; Arithmetic on labels
<expr            ; Force 8-bit value
>expr            ; Force 16-bit value
```

22.6.3. Numeric Constants

```
10.               ; Decimal 10
10                ; Octal 10 (= decimal 8)
0x10              ; (not standard, use octal)
'a                ; Character constant (ASCII value)
"str              ; String (each char is a word)
```

22.6.4. Common Patterns

```
; Function prologue
func:
    mov  r5,-(sp)     ; Save frame pointer
    mov  sp,r5        ; Set new frame pointer
    ; ... function body ...
    mov  (sp)+,r5     ; Restore frame pointer
    rts  pc           ; Return

; Access argument (first arg at 4(r5) after prologue)
    mov  4(r5),r0     ; First argument
    mov  6(r5),r1     ; Second argument

; Local variables
    sub  $4,sp        ; Allocate 4 bytes
    ; -2(r5) is first local, -4(r5) is second
```

22.7. PDP-11 Models Used with UNIX v4

Model	Memory	Notes
11/20	28KB	Original UNIX development
11/40	64KB	Memory management
11/45	256KB	Separate I/D space

UNIX v4 was primarily developed on the PDP-11/40 with memory management enabled.

22.8. Quick Reference Card

22.8.1. Most Common Instructions

```
mov  src,dst      ; Copy word
add  src,dst      ; dst += src
sub  src,dst      ; dst -= src
cmp  src,dst      ; Compare (set flags)
tst  src          ; Test (set flags)
beq  label        ; Branch if equal
bne  label        ; Branch if not equal
jsr  pc,func      ; Call function
rts  pc           ; Return from function
```

22.8.2. Register Usage Summary

```
r0    Return value, scratch
r1    Return value (pair), scratch
r2-r4 Preserved across calls
r5    Frame pointer (preserved)
sp    Stack pointer
pc    Program counter
```

22.8.3. Stack Frame Layout

```
        +-----------------+
        | Argument n      |
        +-----------------+
        | ...             |
        +-----------------+
        | Argument 1      | 4(r5)
        +-----------------+
        | Return address  | 2(r5)
        +-----------------+
r5 -->  | Old r5          | 0(r5)
        +-----------------+
        | Local 1         | -2(r5)
        +-----------------+
        | Local 2         | -4(r5)
        +-----------------+
sp -->  | ...             |
        +-----------------+
```

22.9. See Also

- Chapter 2: PDP-11 Architecture
- Chapter 4: Boot Sequence
- Chapter 7: Traps and System Calls
- Appendix A: System Call Reference

23. Appendix D: Glossary

This glossary defines key terms, data structures, functions, and concepts used throughout the UNIX Fourth Edition source code and this commentary.

23.1. A

a.out The default output filename produced by the assembler and linker. Also refers to the executable file format used by UNIX v4. See Appendix B.

address space The range of memory addresses accessible to a process. On the PDP-11, each process has a 64KB (16-bit) address space divided into text, data, and stack segments.

alloc() Function in `alloc.c` that allocates a free disk block from the filesystem's free list.

APR (Active Page Register) PDP-11 memory management register that defines the mapping between virtual and physical addresses for each memory segment.

23.2. B

bio.c Buffer I/O source file (`usr/sys/dmr/bio.c`) containing the buffer cache implementation including `bread()`, `bwrite()`, and `brelse()`.

bmap() Function in `subr.c` that maps a logical file block number to a physical disk block number, handling both direct and indirect block addressing.

bread() "Block read" - reads a disk block into a buffer, returning a pointer to the buffer. Blocks until I/O completes.

brelse() "Buffer release" - returns a buffer to the free list after use. The buffer remains in the cache for potential reuse.

BSS "Block Started by Symbol" - the uninitialized data segment of a program, zero-filled at load time.

buffer cache A pool of memory buffers used to cache disk blocks, reducing the need for disk I/O. Managed by `bio.c`.

bwrite() "Block write" - writes a buffer's contents to disk. May be synchronous or asynchronous depending on flags.

23.3. C

callout A deferred function call, typically scheduled by `timeout()` to execute after a specified delay.

cdevsw[] Character device switch table - an array of function pointers for character device operations (open, close, read, write).

clist Character list - a linked list of small buffers (c-blocks) used for TTY input and output queuing.

clock() Clock interrupt handler in `clock.c`, called 60 times per second. Updates time, handles scheduling, and processes callouts.

context switch The process of saving one process's state and restoring another's, performed by `swtch()` in `slp.c`.

copyin()/copyout() Functions to safely copy data between user and kernel address spaces.

core Physical memory. "Core map" refers to the data structure tracking physical memory allocation.

coremap[] Array tracking allocation of physical memory (core) in 64-byte blocks.

23.4. D

data segment The portion of a process's address space containing initialized global and static variables.

device driver Kernel code that manages a hardware device, providing a standard interface (open, close, read, write, strategy) to the rest of the kernel.

device number A number identifying a device, consisting of a major number (device type/driver) and minor number (specific device instance).

device switch table Arrays (`bdevsw[]`, `cdevsw[]`) mapping device major numbers to driver functions.

direct block A disk block address stored directly in an inode's `i_addr[]` array, as opposed to an indirect block.

directory A special file containing a list of (inode number, filename) pairs. See `namei()`.

dmr Dennis M. Ritchie - author of device driver code in `usr/sys/dmr/`.

23.5. E

effective UID/GID The user/group ID used for permission checking, which may differ from the real UID/GID due to setuid/setgid bits.

estabur() "Establish user registers" - configures memory management registers for a process's text, data, and stack segments.

exec() System call that replaces the current process image with a new program from an executable file.

expand() Function to change a process's memory allocation, either growing or shrinking its address space.

23.6. F

falloc() Allocate a free entry in the system file table.

file descriptor A small integer (0-14 in UNIX v4) that identifies an open file within a process. Index into `u.u_ofile[]`.

file structure Kernel structure (`struct file`) representing an open file, containing the file offset and pointer to the inode.

file table System-wide array of `struct file` entries, shared among all processes.

filsys structure The superblock structure defined in `filsys.h`, containing filesystem metadata.

fork() System call that creates a new process as a copy of the calling process.

free list Linked list of available resources (disk blocks, inodes, buffers, etc.).

fubyte()/fuword() "Fetch user byte/word" - safely read a byte/word from user space.

23.7. G

getblk() Get a buffer for a specified device and block number. May return a cached buffer or allocate a new one.

getf() Get file structure pointer from a file descriptor.

geterror() Extract error status from a buffer after I/O completion.

23.8. H

hash chain Linked list of buffers or inodes with the same hash value, used for quick lookup.

23.9. I

i-list The contiguous area on disk (blocks 2 through N) containing all inodes for a filesystem.

i-node See inode.

ialloc() Allocate a free inode from the filesystem.

ifree() Free an inode, returning it to the filesystem's free inode list.

iget() Get a locked, in-memory copy of an inode, reading from disk if necessary.

ILARG Inode flag indicating "large file" - the inode uses indirect block addressing.

indirect block A disk block containing block numbers rather than file data, used to address large files.

init Process 1, the ancestor of all user processes. Created by the kernel during boot, it spawns `getty` processes and adopts orphaned processes.

inode "Index node" - data structure containing all metadata about a file except its name. Stored both on disk and cached in memory.

interrupt Hardware signal that causes the CPU to suspend current execution and transfer control to an interrupt handler.

interrupt vector Memory location containing the address of an interrupt handler.

iomove() Copy data between a buffer and user space during file I/O.

iput() Release an inode obtained via `iget()`, decrementing its reference count and writing to disk if modified.

itrunc() Truncate a file to zero length, freeing all its data blocks.

iupdat() Update an inode on disk if it has been modified.

23.10. K

ken Ken Thompson - author of core kernel code in `usr/sys/ken/`.

kernel mode Privileged processor mode with full access to hardware and memory. Also called supervisor mode.

kernel stack Per-process stack used when executing in kernel mode, stored in the user structure.

KL11 The console terminal interface on the PDP-11, driven by `kl.c`.

23.11. L

link count Number of directory entries (hard links) referring to an inode. When it reaches zero and no processes have the file open, the file is deleted.

lock Mechanism to ensure exclusive access to a resource. In UNIX v4, typically a flag that causes processes to sleep until cleared.

23.12. M

magic number The first word(s) of a file identifying its format. For executables: 0407 (combined text/-data) or 0410 (separate text).

major device number Upper byte of device number, identifying the device driver.

maknode() Create a new inode in a directory.

malloc() Allocate contiguous blocks from a resource map (core memory or swap space). Not the C library malloc.

memory management Hardware and software mechanisms for mapping virtual addresses to physical addresses and protecting memory regions.

mfree() Return blocks to a resource map.

minor device number Lower byte of device number, identifying a specific device instance to the driver.

MMU (Memory Management Unit) Hardware that translates virtual addresses to physical addresses and enforces memory protection.

mount Attach a filesystem to a directory in the existing file hierarchy.

mount table Array of `struct mount` entries tracking mounted filesystems.

23.13. N

namei() "Name to inode" - converts a pathname to an inode, following the directory hierarchy. The heart of pathname resolution.

newproc() Create a new process structure and copy the parent's context. Called by `fork()`.

nice value Process priority adjustment. Higher nice values mean lower scheduling priority.

23.14. O

open file table Per-process array (`u.u_ofile[]`) of pointers to file structures for open files.

openi() Perform device-specific open operations when opening a special file.

23.15. P

panic() Kernel function called for unrecoverable errors. Prints a message and halts the system.

PDP-11 Digital Equipment Corporation minicomputer family for which UNIX v4 was written. See Appendix C.

physio() Perform physical (raw) I/O directly between a device and user memory, bypassing the buffer cache.

pipe Inter-process communication mechanism allowing one process to write data that another can read. Implemented in `pipe.c`.

priority Scheduling priority determining which process runs next. Lower values mean higher priority.

proc structure Per-process data structure (`struct proc`) containing process state, priority, memory allocation info, etc.

proc[] System-wide array of process structures.

process An executing program with its own address space, resources, and execution context.

process ID (PID) Unique identifier assigned to each process.

prele() Release a locked inode's lock without decrementing its reference count.

PSW (Processor Status Word) PDP-11 register containing condition codes and processor mode/priority.

23.16. R

raw device Character device interface to a block device, bypassing the buffer cache. Typically named with an 'r' prefix (e.g., `/dev/rrk0`).

read-ahead Optimization where the system reads additional blocks beyond what was requested, anticipating future reads.

readi() Read data from an inode into the user area, handling block mapping and buffer cache.

real UID/GID The actual user/group ID of the process owner, as opposed to the effective UID/GID.

reference count Count of pointers/handles to a resource (inode, file structure). Resource is freed when count reaches zero.

register Fast CPU storage location. The PDP-11 has 8 general registers (r0-r7).
resource map Data structure for managing allocation of contiguous blocks (memory, swap space).
RK05 DEC disk drive with 2.4MB capacity, commonly used with UNIX v4. Driven by `rk.c`.
root directory The top-level directory of the filesystem, referenced by inode 1 and accessed as `/`.

23.17. S

sched() The scheduler function in `slp.c`, also known as process 0 or the swapper.
segment A region of the address space (text, data, or stack) with specific permissions and mapping.
setrun() Mark a process as runnable after it was sleeping.
signal Software notification sent to a process, causing it to execute a handler or terminate.
sleep() Put the current process to sleep waiting on a channel (event). Process is awakened by `wakeup()` on that channel.
slp.c Source file containing process switching, sleep/wakeup, and scheduler code.
special file A file representing a device rather than data on disk. Block special files use `bdevsw[]`; character special files use `cdevsw[]`.
stack segment Region of address space for the process stack, growing downward from high addresses.
strategy() Block device driver function that queues I/O requests. Called by buffer cache code.
subyte()/suword() "Store user byte/word" - safely write a byte/word to user space.
superblock Block 1 of a filesystem, containing filesystem metadata (size, free lists, etc.).
sureg() "Set user registers" - load memory management registers with process-specific values.
swap device Disk (or partition) used for swapping process images in and out of memory.
swap() Move a process image between memory and the swap device.
swapmap Resource map tracking allocation of swap space.
swapper Process 0, which moves processes between memory and swap space when memory is scarce.
switch register Console panel switches on the PDP-11, readable via the `switch` system call.
swtch() Perform a context switch to the highest-priority runnable process.
sysent[] System call entry table mapping system call numbers to handler functions.
system call Request from user program to kernel for a service. Invoked via the `sys` (trap) instruction.

23.18. T

text segment Read-only portion of address space containing program instructions. May be shared between processes running the same program.

text structure Kernel structure tracking shared text segments among processes.

time System time in seconds since January 1, 1970 (the UNIX epoch). Stored as two 16-bit words.

timeout() Schedule a function to be called after a specified number of clock ticks.

trap Exception or interrupt caused by executing a special instruction (like `sys`) or error condition.

trap() Kernel trap handler in `trap.c`, dispatching system calls and handling faults.

TTY (teletype) Terminal device. The TTY subsystem handles input/output processing for character-based terminals.

23.19. U

u (user structure) Per-process kernel data structure containing open files, current directory, signal handlers, saved registers, and the kernel stack. Always mapped at a fixed kernel address.

u.u_ar0 Pointer to saved user registers in the user structure.

u.u_base I/O transfer address for current system call.

u.u_cdir Pointer to inode of current working directory.

u.u_count I/O transfer count for current system call.

u.u_error Error code from most recent system call.

u.u_offset[] File offset for current I/O operation.

u.u_ofile[] Array of pointers to open file structures.

u.u_procp Pointer to the current process's proc structure.

u.u_signal[] Array of signal handler addresses.

ufalloc() Allocate a free file descriptor in the current process.

USIZE Size of user structure in 64-byte blocks (typically 16, or 1KB).

user mode Unprivileged processor mode for running user programs, with restricted access to hardware and memory.

23.20. V

vector See interrupt vector.

23.21. W

wait channel (wchan) Address used to identify what event a sleeping process is waiting for. Processes are awakened when `wakeup()` is called with their wait channel.

wait() System call to wait for a child process to terminate and retrieve its exit status.

wakeup() Wake all processes sleeping on a specified channel (address).

wdir() Write a directory entry.

working directory The current directory for pathname resolution. Changed by `chdir()`.

writei() Write data from the user area to an inode, handling block mapping and buffer cache.

23.22. X

xalloc() Allocate shared text segment for a process.

xfree() Free a process's reference to its shared text segment.

23.23. Z

zombie A terminated process whose parent has not yet called `wait()`. The process structure remains allocated to hold the exit status.

23.24. Numeric/Symbol

0407 Magic number for normal (non-pure) executable files.

0410 Magic number for pure (shared text) executable files.

/dev Directory containing device special files.

/etc Directory containing system configuration files.

/etc/init The init program, first user process executed after boot.

/etc/passwd User account database.

23.25. Source File Quick Reference

File	Location	Contents
alloc.c	ken/	Disk block allocation
bio.c	dmr/	Buffer cache
clock.c	ken/	Clock interrupt handler
fio.c	ken/	File descriptor operations
iget.c	ken/	Inode operations
kl.c	dmr/	Console driver
main.c	ken/	Kernel entry point
mem.c	dmr/	Memory device driver
nami.c	ken/	Path resolution (namei)
pipe.c	dmr/	Pipe implementation
prf.c	ken/	printf functions
rdwri.c	ken/	readi/writei
rk.c	dmr/	RK05 disk driver
sig.c	ken/	Signal handling
slp.c	ken/	Sleep/wakeup, scheduler
subr.c	ken/	bmap and utilities
sys1.c	ken/	fork, exec, exit, wait
sys2.c	ken/	open, read, write, close
sys3.c	ken/	unlink, chdir, chmod, etc.
sys4.c	ken/	signal, kill, times, etc.
sysent.c	ken/	System call table
text.c	ken/	Shared text segments
trap.c	ken/	Trap handler
tty.c	dmr/	TTY line discipline

23.26. See Also

- Appendix A: System Call Reference
- Appendix B: File Formats
- Appendix C: PDP-11 Quick Reference

24. Appendix E: Running UNIX v4

This appendix explains how to run UNIX v4 on a modern computer using the OpenSIMH PDP-11 simulator. Running the actual system lets you experiment with the code discussed throughout this book.

24.1. Resources

The UNIX v4 restoration and emulation documentation is maintained at:

- **squoze.net** — http://squoze.net/UNIX/v4/ — Complete restoration by Angelo Papenhoff
- **Internet Archive** — https://archive.org/details/utah_unix_v4_raw — Original tape image
- **Turnkey version** — http://squoze.net/UNIX/v4/turnkey/ — Pre-built, ready to boot

For the most up-to-date instructions, consult squoze.net. The instructions below provide a quick start.

24.2. Prerequisites

24.2.1. macOS (MacPorts)

```
sudo port install opensimh
```

OpenSIMH executables have a `simh-` prefix (e.g., `simh-pdp11`).

24.2.2. Debian/Ubuntu

```
sudo apt install simh
```

The executable is named `pdp11`.

24.3. Quick Start with Turnkey Version

The fastest way to get running:

```
# Download pre-built disk and boot script
curl -O http://squoze.net/UNIX/v4/turnkey/disk.rk
curl -O http://squoze.net/UNIX/v4/turnkey/boot.ini

# Boot UNIX v4
simh-pdp11 boot.ini
```

At the boot prompt:

```
k
unix
```

Login as `root` (no password).

24.4. Installation from Tape

For the full installation experience:

24.4.1. Step 1: Download Files

```
curl -O http://squoze.net/UNIX/v4/unix_v4.tap
curl -O http://squoze.net/UNIX/v4/install.ini
curl -O http://squoze.net/UNIX/v4/boot.ini
```

24.4.2. Step 2: Run Installation

```
simh-pdp11 install.ini
```

At the = prompt, enter:

```
=mcopy
'p' for rp; 'k' for rk
k
disk offset
0
tape offset
75
count
4000
```

```
=uboot
k
unix
```

You should see:

```
mem = 64530

login: root
```

24.4.3. Step 3: Verify Installation

```
# ls
bin
dev
etc
lib
mnt
tmp
unix
usr
# sync
# sync
```

24.5. Basic Usage

24.5.1. Important: Use `chdir` not `cd`

UNIX v4 predates the `cd` alias. You must use `chdir` to change directories:

```
# chdir /usr      # correct
# cd /usr         # wrong - no such command
```

24.5.2. Essential Commands

```
# ls              # list files
# ls -l           # long listing
# pwd             # print working directory
# chdir /usr      # change directory
# cat file        # display file
# ps              # show processes
# who             # show logged-in users
# date            # show date (will show 1974!)
# df              # disk free space
```

24.5.3. Text Editing with ed

The only editor is ed, a line editor:

```
# ed filename
a                 # append mode
Type your text here
.                 # period alone exits append mode
w                 # write (save)
q                 # quit
```

24.5.4. Compiling C Programs

```
# ed hello.c
a
main()
{
    printf("Hello from UNIX v4!\n");
}
.
w
q
# cc hello.c      # compile
# a.out           # run
Hello from UNIX v4!
```

24.6. Shutdown Procedure

Critical: Always follow this procedure to prevent data loss.

```
# sync            # flush buffers
# sync            # run twice to ensure completion
```

Then press `Ctrl+E` to enter the simulator prompt:

```
Simulation stopped, PC: 002040 (MOV (SP)+,177776)
sim> quit
Goodbye
```

Why sync twice? The first `sync` starts the buffer flush but doesn't guarantee completion. The second ensures all data is written.

24.7. Optional: Rebuilding the Kernel

If `ps` doesn't work or you want to customize the kernel:

24.7.1. Step 1: Download Kernel Build Files

```
curl -O http://squoze.net/UNIX/v4/sys.tp
```

24.7.2. Step 2: Update boot.ini

```
set cpu 11/45
set tc en
att rk0 disk.rk
att tc1 sys.tp
d sr 2
boot rk
```

24.7.3. Step 3: Extract and Build

```
# chdir /usr/sys
# tp 1x           # extract from DECtape
# sh run          # build kernel
# mv a.out /nunix
# sync
# sync
```

24.7.4. Step 4: Boot New Kernel

Restart the simulator and at the boot prompt:

```
k
nunix            # boot new kernel instead of "unix"
```

If successful, replace the old kernel:

```
# mv /unix /unix.old
# mv /nunix /unix
```

24.8. Optional: Installing Manual Pages

UNIX v4 didn't include manual pages. The restoration provides them from v6:

```
curl -O http://squoze.net/UNIX/v4/nroff.tp
curl -O http://squoze.net/UNIX/v4/man.tap
```

See squoze.net for detailed installation instructions. Note that `nroff` has a known bug where it resets the terminal to uppercase mode—use output redirection (`man cat > temp && cat temp`) to work around this.

24.9. Troubleshooting

Stuck at = prompt: Type `uboot` then k then `unix`

Lost data after reboot: Always run `sync` twice before exiting

ps command doesn't work: Rebuild the kernel with the updated files from sys.tp

Terminal stuck in UPPERCASE: Run `stty -lcase` or logout and login again

24.10. Experiments to Try

With a running UNIX v4 system, you can verify the code discussed in this book:

1. **Examine the filesystem:**

```
# ls -l /dev
# cat /etc/passwd
# od /unix | head
```

2. **Watch process creation:**

```
# ps a
# sh &
# ps a
```

3. **Explore the C compiler:**

```
# chdir /usr/c
# ls
# cat c00.c | head -50
```

4. **Look at the kernel source:**

```
# chdir /usr/sys/ken
# ls
# cat main.c
```

5. **Try the assembler:**

```
# chdir /usr/source/s1
# ls *.s
# cat cat.s
```

24.11. File Descriptions

File	Description
unix_v4.tap	Original tape in SIMH format
disk.rk	RK05 disk image (your filesystem)
install.ini	SIMH config for installation
boot.ini	SIMH config for booting
sys.tp	DECtape with kernel build files
nroff.tp	Text formatting system from v6
man.tap	Manual pages

24.12. Notes

- UNIX v4 has no password for root by default
- The system uses 64KB of memory
- RK05 disk images are approximately 2.4MB
- Press `Ctrl+E` at any time for the SIMH prompt
- The date will show 1974—this is correct for the tape

For comprehensive documentation and the latest updates, visit squoze.net/UNIX/v4.

25. About the Author

Briam Rodriguez grew up in Little Havana, Miami, Florida. He attended Miami Senior High School, where he discovered Linux and recognized that serious systems programming required mastering C. His mother purchased a copy of *The C Programming Language* by Brian Kernighan and Dennis Ritchie at a time when the family could barely afford it. That book became the foundation of his career.

He enrolled at Miami Dade Community College but left before completing his degree to support his family. He continued his education independently, teaching himself systems programming from source code and technical literature.

His early professional work included writing Linux kernel device drivers at Tangent POS, developing keyboard and display drivers for point-of-sale systems based on the Intel SA1110 StrongARM platform during the Linux 2.4 era. He went on to lead development of Congressional Moments for iPad, the Lexee voice assistant at Angel.com, and the Salesforce Cloud CTI integration. His experience spans kernel development, iOS applications, and full-stack web engineering, with production work in C, Python, Objective-C, Perl, PHP, C++, and assembly.

He holds five U.S. patents:

- *Systems and Methods for Maintaining Internet Protocol Address During and After Failover* (US-20240121697-A1, 2024)
- *Conversation Assistant* (US-10129720-B1, 2012)
- *Display Screen with Graphical User Interface* (US-D763863-S, US-D771643, US-D802613-S; 2016)

He currently serves as Senior Vice President of Engineering for the POTS In A Box offering at DataRemote, Inc.

This book represents both a personal debt and a professional aspiration. The debt is to Ritchie, Thompson, Kernighan, and the Bell Labs engineers whose work enabled his career. The aspiration is to provide what he once searched for and could not find: a guide to UNIX internals written with the clarity and concision that distinguished *The C Programming Language* — explaining not merely what the code does, but why it was designed that way.

The original UNIX authors wrote systems of remarkable elegance under severe constraints. This commentary is an attempt to make their thinking accessible to a new generation of programmers.

The UNIX Fourth Edition Source Code Commentary

In 1974, a magnetic tape containing UNIX Fourth Edition was sent from Bell Labs to the University of Utah. Fifty years later, it was rediscovered in a storage closet—the only surviving copy of the first UNIX written in C.

This book is a complete guide to that code.

At just 10,000 lines, UNIX v4 is small enough for one person to understand completely—yet it contains a fully functional operating system with multiprocessing, a hierarchical filesystem, device drivers, and a shell. Every concept that defines modern computing is here, in its purest form.

What you'll learn:

- How an operating system boots and initializes itself
- How processes are created, scheduled, and terminated
- How the filesystem stores and retrieves data
- How system calls bridge user programs and the kernel
- How device drivers interface hardware to software
- How the shell parses and executes commands
- How the C compiler transforms source into machine code

Who this book is for:

- Systems programmers who want to understand operating systems at the deepest level
- Computer science students seeking a comprehensible, complete OS to study
- Historians of computing exploring UNIX's origins
- Anyone curious how 10,000 lines of code changed the world

The code is elegant. The design is timeless. The ideas are foundational.

"UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity."
– Dennis Ritchie