Principia Softwarica
Fundamental Literate System Programs
version 1.0

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Ken Thompson, Rob Pike, Dave Presotto, Phil Winterbottom,
Tom Duff, Andrew Hume, Sape Mullender, Russ Cox,
Richard Miller, Ori Bernstein, and Ron Minnich.

May 28, 2026

The text and figures are Copyright © 2014–2026 Yoann Padioleau.
All rights reserved.

The source code is Copyright © 2021 Plan 9 Foundation.
MIT license.

Contents

1 Introduction **5**
- 1.1 Motivations . . . 5
- 1.2 The ideal teaching operating system . . . 7
- 1.3 The Plan 9 operating system . . . 7
- 1.4 The C language and ARM architecture . . . 9
- 1.5 Literate programming . . . 9
- 1.6 Getting started . . . 10
- 1.7 Requirements . . . 11
- 1.8 Copyright . . . 11
- 1.9 Acknowledgments . . . 11

2 Overview **12**
- 2.1 Code organization . . . 12
- 2.2 Software architecture . . . 12

3 The Literate Programs **16**
- 3.1 The Core system . . . 16
  - 3.1.1 The kernel: `9pi` . . . 16
  - 3.1.2 The core libraries: `libc`, `libthread`, `libbio`, `libflate`, `libregexp` . . . 16
  - 3.1.3 The shell: `rc` . . . 16
- 3.2 The development toolchain . . . 17
  - 3.2.1 The C compiler: `5c` . . . 17
  - 3.2.2 The assembler: `5a` . . . 17
  - 3.2.3 The linker: `5l` . . . 17
  - 3.2.4 The processor emulator: `5i` . . . 17
- 3.3 The developer tools . . . 18
  - 3.3.1 The text editor: `ed` . . . 18
  - 3.3.2 The build system: `mk` . . . 18
  - 3.3.3 The version control system: `git9` . . . 18
  - 3.3.4 The debuggers: `acid`, `ratrace`, `db` . . . 18
  - 3.3.5 The profilers: `prof`, `tprof`, `stats`, `iostats` . . . 19
- 3.4 Graphics . . . 19
  - 3.4.1 The graphics stack: `/dev/draw` . . . 19
  - 3.4.2 The windowing system: `rio` . . . 19
  - 3.4.3 The graphical user interface toolkit: `libpanel` . . . 20
- 3.5 Networking . . . 20
  - 3.5.1 The network stack: `/net` . . . 20
  - 3.5.2 The web browser: `mothra` . . . 20
- 3.6 Utilities: `cat`, `ls`, `grep`, `sed`, `cmp`, `tar`, `gzip`, `bc`, `dc`, `hoc`, `awk`, ... 20
- 3.7 Applications . . . 21

4 Conclusion **22**

A Other Teaching Operating Systems **23**

B Bootstrapping from Scratch **25**
B.1 A basic computer . . . 26
B.2 A booting procedure . . . 26
B.3 Physical programs . . . 27
B.4 Phase 1: Machine code . . . 27
B.5 Phase 2: Assembly . . . 29
B.6 Phase 3: C . . . 32
B.7 Summary of Plan 9 ancestor programs . . . 33

C Literate Program Example: `ed` **34**
C.1 Introduction . . . 34
C.1.1 A line editor . . . 34
C.1.2 A scriptable editor . . . 35
C.1.3 A Unicode editor . . . 36
C.2 Core data structures . . . 37
C.2.1 The backing store: `tfname`, `tfile`, and `tline` . . . 37
C.2.2 The target file: `savedfile` . . . 38
C.2.3 The lines as file offsets: `zero` . . . 38
C.2.4 Cursors: `dot` and `dol` . . . 39
C.2.5 Line input and output buffers: `line` and `linebuf` . . . 39
C.2.6 Other globals: `count`, `col`, etc. . . . 40
C.3 `main()` . . . 40
C.3.1 `ed -` and `vflag` . . . 41
C.3.2 `ed -o` and `oflag` . . . 42
C.3.3 `mktemp()` . . . 42
C.3.4 `init()` . . . 43
C.3.5 `quit()` . . . 43
C.4 Displaying and reading text . . . 44
C.4.1 Displaying text . . . 44
C.4.2 Reading text . . . 46
C.5 `commands()` interpreter loop . . . 48
C.6 reading a file: `r` . . . 48
C.6.1 Reading a `filename()` . . . 49
C.6.2 `setwide()` and `squeeze()` . . . 50
C.6.3 `append()` and `getfile()` . . . 51
C.6.4 `putline()` (simplified) . . . 53
C.6.5 Concrete editing example . . . 54
C.6.6 `exfile()` . . . 55
C.7 writing a file: `w` . . . 55
C.7.1 `putfile()` . . . 56
C.7.2 `getline()` (simplified) . . . 57
C.7.3 Write and quit: `wq` . . . 57
C.8 Main commands . . . 57
C.8.1 printing lines: `p` . . . 58
C.8.2 printing the remembered file: `f` . . . 59
C.8.3 printing line number: `=` . . . 59

C.8.4 append and insert: a, i . . . 59
C.8.5 quitting: q . . . 60
C.8.6 deleting lines: d . . . 60
C.8.7 changing lines: c . . . 61
C.8.8 moving and copying lines: m and t . . . 61
C.9 Command addresses . . . 63
C.9.1 Reading addr1 and addr2 . . . 64
C.9.2 address() . . . 65
C.9.3 Basic addresses: . and $ . . . 66
C.10 Search and replace . . . 66
C.10.1 Search as addresses: /re/ and ?re? . . . 67
C.10.2 Reading and compiling a regexp: compile() . . . 68
C.10.3 match() . . . 69
C.10.4 Reading and compiling a substitution: compsub() . . . 70
C.10.5 substitute: s . . . 71
C.10.6 Advanced substitutions . . . 74
C.10.7 global commands: g/re/cmd and v/re/cmd . . . 77
C.11 Advanced features . . . 79
C.11.1 Running a shell command: ! . . . 79
C.11.2 Advanced listing: l . . . 80
C.11.3 Advanced listing: n . . . 82
C.11.4 joining lines: j . . . 82
C.11.5 browse: b . . . 83
C.11.6 edit remembered file: e . . . 84
C.11.7 Append-only files . . . 84
C.12 Optimizations . . . 85
C.12.1 Optimized putline() . . . 85
C.12.2 getblock() . . . 86
C.12.3 Optimized getline() . . . 88
C.12.4 gdelete() . . . 88
C.13 Error management . . . 89
C.13.1 error() . . . 89
C.13.2 Notes/signals management . . . 90
C.14 Extra code . . . 91
C.15 Index . . . 94

References **97**

Chapter 1

Introduction

Principia Softwarica is a series of books explaining how things work in a computer by describing with full details all the source code of all the essential programs used by a programmer. Among those essential programs are the kernel, the shell, the windowing system, the web browser, the compiler, the linker, the editor, or the debugger. Each program will be covered by a separate book.

The books not only describe the implementations of essential programs, *they are* the implementations of those programs. Indeed, each program in Principia Softwarica comes from a *literate program* [Knu92], which is a document containing both source code and documentation and where the code is organized and presented in a way to facilitate its comprehension. The actual code and the book are derived both automatically from this literate program. See Appendix C for a concrete example of literate program.

The goal of the report you are reading now is to introduce the series and to give a quick overview of the Principia Softwarica programs. They form together the foundation on top of which all applications can be built. Similar to *Principia Mathematica* [WR13], which is a series of books covering the foundations of mathematics, the goal of Principia Softwarica is to cover the fundamental programs. Those programs are mostly all *meta programs*, which are programs in which the input and/or output are other programs. For instance, the kernel is a program that manages other programs; the compiler is a program that generates other programs. Those programs are also sometimes referred as *system software*, in opposition to *application software* (e.g., spreadsheets, word processors, email clients), which I will not cover in Principia Softwarica.

1.1 Motivations

Why did I write those books? The main reason is that I have always been curious and always wanted to understand how things work under the hood, fully, to the smallest detail. Programs are now running the world; it is thus important to understand those programs, to be computer literate, and source code is what defines those programs.

There are already lots of books explaining how computers work, explaining the concepts, theories, and algorithms behind programs such as kernels or compilers. There are also a few books about debuggers. However, all those books rarely explain everything with full details, which is what source code is all about. There are a few books that include the whole source code of the program described, for instance, the books about Minix [Tan87], XINU [Com84], or LCC [FH95]. However, those books cover only a few essential programs, and mostly always either the kernel or the compiler. Moreover, they do not form a coherent set.

Enter Principia Softwarica, a set of books covering all essential programs, in a coherent way. In addition to the kernel and compiler, Principia Softwarica covers also the graphics stack, the networking stack, the windowing system, the assembler, the linker, and many other fundamental programs that have never been fully explained before to the best of my knowledge.

I want to demystify those programs by showing their code, and by showing that they are actually not that complicated. I hope to remove some of the mental barriers people have that prevent them from extending the

tools they use every day: text editors, compilers, even kernels. As a side effect, it will also maybe help people imagine better systems. Indeed, it can be very intimidating to invent something completely new if you have no clue that it is actually possible to build from scratch a complete operating system.

Another motivation for those books, in addition to satisfy my curiosity, as well as your curiosity, is that I think you are a better programmer if you understand how things work under the hood. In my opinion, you become a better C programmer when you understand roughly what code generates the C compiler. A good way to write more efficient code, or to avoid writing really slow code, is to have some ideas of the assembly code generated for your code by the compiler. In the same way, you can better use resources, for instance, memory, if you have some ideas about how the kernel manages for you these resources; you can better fix latency issues if you understand how the networking stack works. The Principia Softwarica books can complement the excellent book *Computer systems: a programmer's perspective* [BO10] by illustrating the many concepts this book introduces with concrete code.

I also think it is easier to debug programs if you better understand the environment in which those programs evolve, if you understand the whole software stack, and how things interact with each other. Indeed, to fully understand certain error messages, e.g., from the kernel, from the networking stack, or from the linker, it is very useful to have some ideas about what those programs do. Moreover, even if in almost all situations the bug or the performance issue is in your program, it could also be sometimes in a core library, in the kernel, in the compiler, or in the linker. It is very rare, but when those situations happen, if you have no clue about the environment in which your program runs, you will never be able to fix your problem.

Finally, by showing code written by great programmers, I hope you will also learn how to write better programs. In other engineering fields it is quite common for the students to learn from the work of the masters of their fields.

Here are a few questions I hope the Principia Softwarica books will answer:

- What happens when the user turns on the computer? Which program gets executed first? What does this program?

- What exactly happens when the user types `ls` in a terminal? What is the set of programs involved in such a command? What is the trace of this command through the different layers of the software stack, from the keyboard interrupt to the display of text glyphs on the screen?

- How does source code get compiled, assembled, linked, and finally loaded in memory? What is the memory image of a program? How does it relate to the original source code? How can a debugger display debugging information from a binary?

- How does a debugger work? Does it rely on special services from the processor or from the kernel to implement breakpoints?

- Which program contains the memory allocator? The kernel or The C library? How `malloc()` is implemented?

- How certain graphical elements (e.g., rectangles, ellipses, characters) are rendered on the screen? How does the graphics card help? How does the kernel help? How things are intercepted by the windowing system to make sure applications can not draw in other windows?

- What happens when you open a connection to another machine, when you type a URL in your web browser? How can you achieve a reliable communication on an unreliable physical network?

- What does it take to port an entire operating system to a new machine, for instance, the Raspberry Pi[1]?

[1] https://www.raspberrypi.org/

1.2 The ideal teaching operating system

The question now is which actual source code to present? The code of the major operating systems (e.g., GNU/Linux, macOS, Windows) is gigantic with hundreds of millions lines of code (LOC). Here by operating system I mean not only the kernel but also the programs (the windowing system, the compiler, etc) that provide the platform for running application software. It is impossible to understand such large codebases.

I think there is hope though, that it is possible to understand fairly well everything, that it is possible to answer all of the questions mentioned in the previous section by focusing on the *essence* of those major operating systems. I think it is possible to design a teaching operating system with capabilities similar to the mainstream operating systems, but with a fraction of their code size. Here are the requirements for the ideal teaching operating system I want to use as the basis for the Principia Softwarica books:

- *Open source*: I want to show code, lots of code, and I want you to be able to play with such code, to modify it, so the ideal operating system must be open source.
- *Small*: I want programs that can be described in books of reasonable size. The program needs to be a bit minimalist.
- *As simple as possible, but not simpler*: I want small code, but I do not want to show toy code. For instance, ARM [Sea01] is a simpler architecture than x86 [Int86], so it makes sense to present the code of an ARM assembler rather than an x86 assembler, but I do not want to show a toy assembler for a toy architecture. In the same way I can present the code of a C compiler without certain advanced state-of-the-art optimizations, to simplify the presentation, but I do not want to show a toy compiler for a toy language.
- *Real*: It has to run on real machines (e.g., an x86 desktop, a Mac laptop, a Raspberry Pi), so you can play with it.
- *Complete*: It needs to not only have a kernel and a compiler but also a graphics stack, a networking stack, a windowing system, etc.
- *Coherent*: The whole set of programs must form a coherent set, so that the interactions between those programs can also be described, and described succinctly.
- *Self hosting*: I want to be able to improve, to recompile, and to run everything under the system itself.

Fortunately, this ideal teaching operating system already exists; it is Plan 9[2]

1.3 The Plan 9 operating system

Plan 9[3] is the successor of UNIX. It was designed from scratch by a small team of great programmers (Rob Pike, Dave Presotto, Phil Winterbottom), including the original creator of UNIX (Ken Thompson). Their goal was to redesign UNIX to better integrate graphics and networking, which both became popular after UNIX was originally invented. In some sense, Plan 9 is a kind of UNIX 2.0.

The choice of Plan 9 as the basis for the Principia Softwarica books may not be obvious, but it is in my opinion the simplest and at the same time fairly complete operating system. If you look at the screenshot of Plan 9 in action in Figure 1.1, you will see many features:

- A screen with basic graphics and multiple windows

[2] I discuss a few other candidates in Appendix A.

[3] plan9.bell-labs.com/plan9/

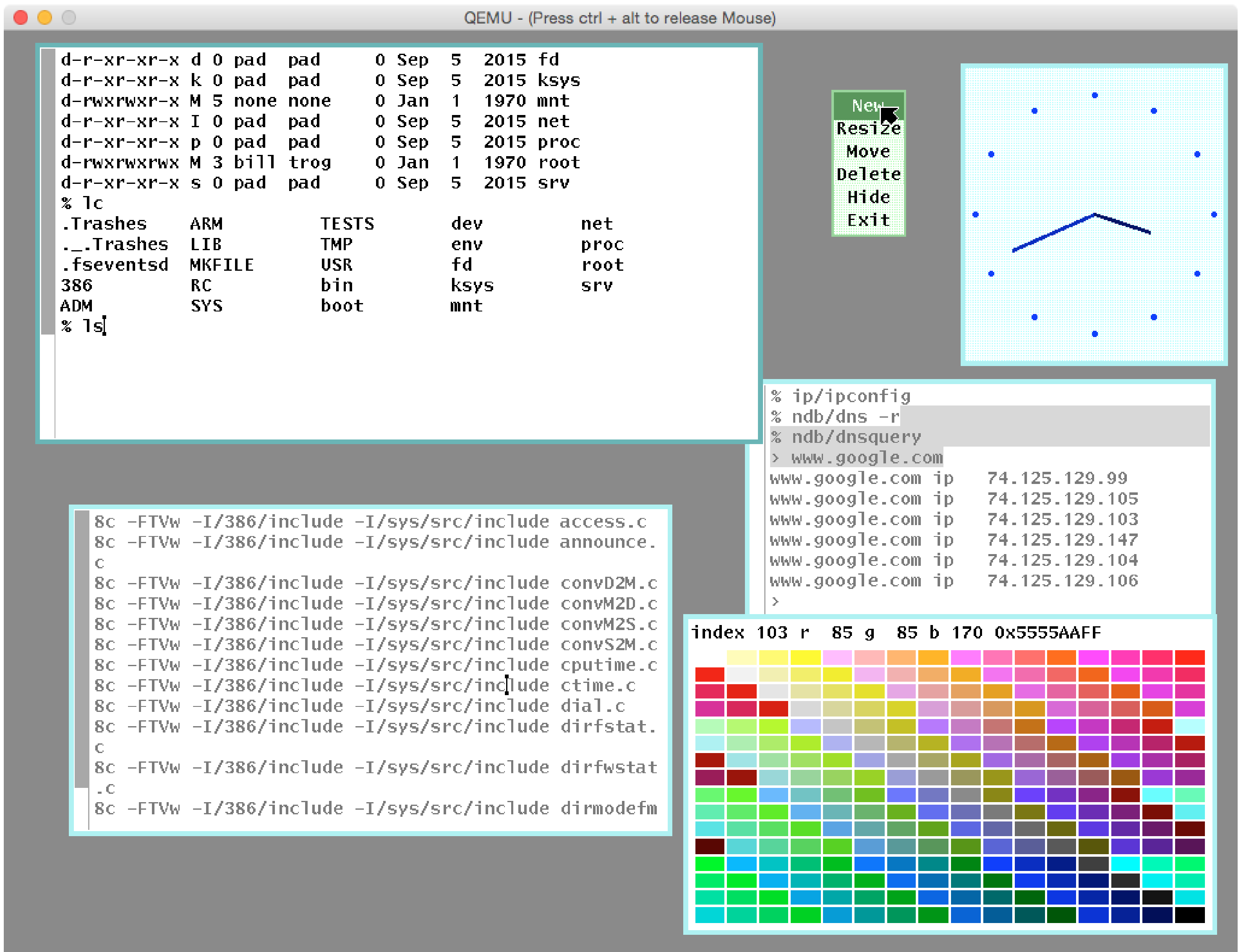


Figure 1.1: Plan 9 in action.

- Multiple shells running independent commands at the same time
- A simple clock graphical application
- A simple program communicating through the network

By comparison, if you look at other systems you will see that Plan 9 *in essence* provides the same core services than GNU/Linux/Xorg, macOS, or Microsoft Windows, without certain bells and whistles, and with a significantly smaller codebase. Indeed, my fork of Plan 9 (whose source code is available at `https://github.com/aryx/principia-softwarica/`), which includes all the essential programs described in all the Principia Softwarica books, and even a few more programs, is less than 365 000 LOC.

The Plan 9 programs are minimalist but powerful, and their source code is simple and small. The creators of Plan 9 were not afraid to rethink everything, not just the kernel, even if it meant not being backward compatible with UNIX. This led to programs with more elegant designs, to the removal of many ugly corners in UNIX, and ultimately to less source code while still providing more services. For instance, thanks to a few novel ideas in the kernel, namely per-process namespace, user-space filesystems, and union mount, every application who wish to interact with the console can just use the uniform `/dev/cons` device file, whether this console is attached directly to the physical terminal, to a remote machine, or whether it is one of the terminal window of the graphical user

interface. Thanks to this design, the Plan 9 windowing system `rio` could be implemented with only 8 800 LOC, including the code to emulate terminals in windows.

By comparison, Linux and X Window introduced the separate concepts of teletype `tty` devices and pseudo `pty` terminals. The code of `xterm`, which is just a terminal for X Window, not X Window itself, has already 88 000 LOC. This is partly because `xterm` carries the historical baggage of standards invented in the 70's (e.g., special control sequences for VT100 terminals).

Plan 9 contains all the essential programs used by a programmer. They form a coherent set because they were all designed from scratch by a small team of great programmers.

1.4 The C language and ARM architecture

The goal of books such as *The Elements* [EucBC] or *Principia Mathematica* [WR13] is to describe the foundation of a field starting from a very small basis, for instance, a logic language with a small set of axioms and inference rules, on top of which all the rest can be derived or built. The book *The Elements of Computing Systems* [NS05] does a similar thing for computers. Starting only from the `nand` logic gate, the book builds gradually the `and`, `or`, and `not` logic gates, a multiplexer, a flip-flop, memory banks, an additioner, an arithmetic and logic unit, and finally a simple processor.

However, in Principia Softwarica we are interested in software, not hardware. Because most of the programs described in the Principia Softwarica books are written in C, our basis in some sense is the C programming language [KR88]. C is a fairly large language, with non trivial semantics, so our basis is unfortunately fairly large too. Note that one of the Principia Softwarica book describes a C compiler that targets the ARM [Sea01] architecture, so in principle our basis could be reduced to the ARM machine language, which is fairly simple. Another Principia Softwarica book describes an ARM emulator. However, both the C compiler and the ARM emulator are written in C itself, which brings us back to C as our basis.

A more elegant alternative, avoiding self-reference, would be to start from a simple machine and build a tower of increasingly powerful languages. Starting from raw binary machine code, we could gradually build more sophisticated languages through a series of *bootstrapping* steps. In fact, such projects exist[4], but with already six bootstrapping steps its author was still far away from a language like C. Appendix B outlines a similar (long) process to bootstrap Plan 9 from scratch.

Another alternative, chosen by Donald Knuth for his encyclopedic books *The Art of Computer Programming* [Knu73], was to pick as a basis a very simple computer he invented called MIX, and a very simple assembly language called MIXAL. Using assembly is maybe OK for describing algorithms, but I think it would not be productive for writing entire programs; this would lead to very long Principia Softwarica books.

I think that starting directly from the ARM machine, a real but fairly simple machine, and the C language, a higher level language than assembly, is maybe less elegant but more practical for Principia Softwarica.

1.5 Literate programming

I want to show source code because it is the ultimate explanation for what a program does. However, I think that showing pages and pages of listings in an appendix, as done for instance in the Minix book [Tan87], even when this appendix is preceded by documentation chapters, is not the best way to explain code. I think the code and its documentation should be mixed together, as done for instance in the Xinu book [Com84], so you do not have to switch back and forth between an appendix and multiple chapters.

Literate programming [Knu92] is a technique invented by Donald Knuth to make it easy to mix code and documentation in a document in order to better develop and better explain programs. Such documents are called *literate programs*. All Principia Softwarica programs are literate programs.

[4] See `http://homepage.ntlworld.com/edmund.grimley-evans/bcompiler.html` or more recently the MES (Maxwell Equations of Software) project `https://www.gnu.org/software/mes/`.

Note that literate programming is different from using API documentation generators such as javadoc[5] or doxygen[6]. Noweb[7], the literate programming tool I used, does not provide the same kind of services. Indeed, literate programming allows programmers to explain their code in the order they think the flow of their thoughts and their code would be best understood, rather than the order imposed by the compiler.

Literate programming allows, among other things, to explain the code piece by piece, with the possibility to present a high-level view first of the code, to switch between top-down and bottom-up explanations, and to separate concerns. For instance, the `Proc` data structure — which you will see in the KERNEL book [Pad14] is a data structure that represents some information about a process — is a huge structure with more than 90 fields. Many of those fields are used only for advanced features of the kernel. The C compiler imposes to define this structure in one place. Noweb allows to present this structure piece by piece, gradually, in different chapters. I can show first the code of the structure with the most important fields, and delay the exposition of other fields to advanced chapters. This greatly facilitates the understanding of the code, by not submerging you with too much details first.

In the same way, the `main()` function in most programs is rather large and mixes together many concerns: command line processing, error management, debugging output, optimizations, and usually a call to the main algorithm. Showing in one listing the whole function would hide behind noise this call to the main algorithm. The main flow of the program though is arguably the most important thing to understand first. Using literate programming, I can show code where the most important parts are highlighted, and where other concerns are hidden and presented later.

In fact, I spent lots of time during the writing of the Principia Softwarica books in transforming the Plan 9 programs in literate programs, and in reorganizing again and again the Plan 9 code to find the best way, the best order, the best separation of concerns in which I think you would more easily understand the code.

See Appendix C for a concrete example of literate program.

1.6 Getting started

To play with the different programs described in the Principia Softwarica books, I recommend to use my fork of Plan 9: `https://github.com/aryx/principia-softwarica`. Section 2.1 will explain the directory structure of this repository. To compile and install this fork of Plan 9, you will need a machine with a C compiler, a UNIX-like operating system, and a kernel that can write on a VFAT filesystem. So, GNU/Linux and macOS are possible host operating systems. Plan 9 is also a valid host as you can build Plan 9 under Plan 9. You can also build Plan 9 using Docker[8].

Because the source code of Plan 9 uses a special dialect of C and some non-standard assembly, you can not use directly popular C compilers and assemblers such as `gcc` and `gas`. To compile Plan 9, you will also need to download my fork of Ken Thompson's C (cross) compilers called Goken at `https://github.com/aryx/goken9cc`. You can *compile* Goken using a regular C compiler (e.g., `gcc`) from your host operating system (e.g., from Linux or macOS). Then you can *cross-compile* the Plan 9 kernel, the Plan 9 C standard library, and then the whole Plan 9 operating system with all its libraries and programs, by using the Goken compiler installed in the previous step. This will build from scratch a Plan 9 distribution.

Finally, you can run this Plan 9 distribution either under Qemu[9], which makes it easy to experiment, or by installing the distribution on a physical machine such as a Raspberry Pi. See `https://www.principia-softwarica.org/getting-started.html` for the full procedure.

[5] `http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html`
[6] `http://www.stack.nl/~dimitri/doxygen/`
[7] `http://www.cs.tufts.edu/~nr/noweb/`
[8] `https://www.docker.com/`
[9] `http://www.qemu.org`

1.7 Requirements

The Principia Softwarica books are not introductions to programming, computer science, or to any subfields of computer science. For instance, the book on the kernel is not an introduction to operating systems. Indeed, I will assume you have already a rough idea of how a kernel works and so that you are already familiar with concepts such as virtual memory, critical regions, interrupts, or system calls. I will present with full details the source code of different programs, but I assume you already know most of the concepts, theories, and algorithms behind those programs. The Principia Softwarica books are there to cover the practice. I assume the readers of Principia Softwarica are students in computer science or programmers who desire to consolidate their knowledge by reading the ultimate computer science explanations: source code.

As I said earlier, because most of the books are made of C source code, you will need a good knowledge of the C programming language [KR88]. Because the programs I describe are Plan 9 programs, and because Plan 9 has a lot in common with its ancestor UNIX, you will also need to be familiar with those systems. I recommend to read [KP84] for UNIX, and to read the Plan 9 tutorials [PPD+95, PPT+93] available in my Plan 9 repository for Plan 9.

1.8 Copyright

All the programs in Principia Softwarica are from Plan 9, with copyrights from the Plan 9 Foundation. They are open source; permission is granted to copy, distribute and/or modify the source code according to the MIT license.

1.9 Acknowledgments

I would like to acknowledge of course Plan 9's authors who wrote in some sense most of the content of the Principia Softwarica books: Ken Thompson, Rob Pike, Dave Presotto, Phil Winterbottom, Russ Cox, and many other people from Bell Labs.

Finally, I would like to thanks Donald Knuth for his early encouragments. It meant everything.

Chapter 2

Overview

Before jumping in the description of the different Principia Softwarica programs in the next chapter, I first give here an overview of how the code is organized, and how the different programs depend on each other.

2.1 Code organization

Table 2.1 presents short descriptions of the main directories in my fork of Plan 9, as well as the corresponding sections in this document in which the program associated with the directory is discussed.

A few Plan 9 programs have some architecture-specific parts, with support for x86 and ARM in my fork of Plan 9. The LOC column in Table 2.1 accounts for the architecture independent part of the code, as well as the x86 and ARM specific code. However, the Principia Softwarica books will present only ARM-specific code. Moreover, even if some directories have a plural form, e.g., `editors/`, the LOC column accounts only for one variant of this program category, e.g., one of the editor, the one I chose to present in Principia Softwarica (`ed`).

2.2 Software architecture

Many of the Principia Softwarica programs are mutually dependent on each other. Indeed, to run a compiler or an editor you need a kernel (and a shell), but to create this kernel in the first place you need an editor and a compiler. In a similar way the C compiler uses code from the core C library, but to create this library you need a C compiler. In fact, the C compiler is written in C itself, so there are even self-dependencies[1]

It is possible though to *layer* things by looking at how things are organized in memory. A first separation to make is between code running in *kernel space* and code running in *user space*, as shown in the left of Figure 2.1. Most of the code running in kernel space is in `kernel/`, as well as some code in `lib_graphics/` and `lib_networking/`. Some of the code in the C library (the memory pool library [2] and a few utility functions and globals) are used both in the kernel and in user programs, but they are the exceptions. The rest of the codebase runs in user space.

The boundary between user programs and the kernel is provided by the *system calls* application programming interface (API). The functions of this API are declared in `include/core/sycall.h` included from `include/core/libc.h` (which contains also many utility functions). The user-space part of the system calls are implemented in `lib_core/libc/9syscall/`. This directory contains one assembly file per system call and each of those files contains mostly the software interrupt instruction with a special value in one of the argument in order to be dispatched to the appropriate code in the kernel. The dispatcher kernel code is in `kernel/syscalls/`. Some of those system calls are process related (e.g., `rfork()`, `exec()`, `exit()` [3]), some are memory related (e.g.,

[1] Appendix B describes how to solve those mutual and self-dependencies issues.

[2] `lib_core/libc/port/pool.c`

[3] with implementation in `kernel/processes/`

```
         +---+    +---+    +---+
         | a |    | l |    | c |                 applications/-+
         | s |    | i |    | o |                 | +---------+ |
         | s |    | n |    | m |                 | | clock.c | |
         | e |    | k |    | p |                 | +---------+ |
         | m |    | e |    | i |                 | +---------+ |
         | b |    | r |    | l |                 | |colors.c | | windows/rio+
         | l |    | s |    | e |                 | +---------+ | |          |
         | e |    | / |    | r |                 |             | |          |
U        | r |    |   |    | s |                 |             | |          |
S        | s |    |   |    | / |                 |             | |          |
E        | / |    |   |    |   |    shells/-+    +-------------+ +----------+
R        +---+    +---+    +---+    |       |   lib_graphics/libdraw/--------+
       +-------------------------+  |       |   |  include/graphics/draw.h   |
S      |include/obj/5.out.h      |  |       |   |                            |
P      +-------------------------+  +-------+   +----------------------------+
A    lib_core/libc/-----------------------------------------------------------+
C    |                        include/core/libc.h                             |
E    |  +-------+ +-------+ +-----+  +-------+ +-------+ +-----+ +-------+    |
     |  | Math  | |Unicode| | Fmt |  |Memory | |Strings| | IO  | |System |    |
     |  +-------+ +-------+ +-----+  +-------+ +-------+ +-----+ +-------+    |
     |9syscall/--------------------------------------------------------------+|
     ||          rfork() | brk() | open() | read() | write() | ...           ||
     ++----------------------------------------------------------------------++
-----------------------------------------------------------------------------------
     kernel/--------------------------------------------------+  +-----+  +-----+
     |  +---+    processes/---------+   devices/-----------+  |  |  l  |  |  l  |
K    |  |   |    |   sysrfork()     |   |  screen/------+  |  |  |  i  |  |  i  |
E    |  |   |    |   sysexec()      |   |  |            |  |  |  |  b  |  |  b  |
R    |  | s |    +------------------+   |  +------------+  |  |  |  _  |  |  _  |
N    |  | y |    memory/------------+   |  keyboard/----+  |  |  |  g  |  |  n  |
E    |  | s |    |  sysbrk()        |   |  |            |  |  |  |  r  |  |  e  |
L    |  | c |    +------------------+   |  +------------+  |  |  |  a  |  |  t  |
     |  | a |    files/-------------+   +------------------+  |  |  p  |  |  w  |
S    |  | l |    |  sysopen()       |   console/-----------+  |  |  h  |  |  o  |
P    |  | l |    |  sysread()       |   |                  |  |  |  i  |  |  r  |
A    |  | s |    |  syswrite()      |   +------------------+  |  |  c  |  |  k  |
C    |  | / |    +------------------+                         |  |  s  |  |  i  |
E    |  |   |    network/-----------+                         |  |  /  |  |  n  |
     |  |   |    |                  |                         |  |     |  |  g  |
     |  |   |    +------------------+                         |  |     |  |  /  |
     |  +---+                                                 |  |     |  |     |
     +--------------------------------------------------------+  +-----+  +-----+
```

Figure 2.1: Software architecture of Plan 9.

Directory	Description	Section	LOC
`kernel/`	The Plan 9 basic kernel (for ARM and x86)	3.1.1	75 000
	+ parts of graphics stack (devices)	3.4.1	11 000
	+ network stack (protocols)	3.5.1	23 000
`include/`	The header files (e.g., `libc.h`)	3.1.2	7 500
`lib_core/`	The core C libraries (for ARM and x86)	3.1.2	33 000
`shells/`	The shell `rc`	3.1.3	7 500
`compilers/`	The C compiler (for ARM and x86)	3.2.1	39 000
`assemblers/`	The ARM and x86 assemblers	3.2.2	6 000
`linkers/`	The ARM and x86 linkers	3.2.3	17 000
`machine/`	The ARM emulator `5i`	3.2.4	4 500
`generators/`	The code generators Lex and Yacc	??	6 500
`editors/`	The editor `ed`	3.3.1	1 600
`builders/`	The build system `mk`	3.3.2	6 000
`version_control/`	The version control system `git9`	3.3.3	10 000
`debuggers/`	The debuggers (for ARM and x86) and tracers	3.3.4	22 000
`profilers/`	The profilers	3.3.5	5 000
`lib_graphics/`	Graphics stack libraries	3.4.1	29 000
`windows/`	The windowing system `rio`	3.4.2	8 800
`lib_gui/`	The widget library `libpanel`	3.4.3	4 000
`lib_networking/`	A small part of the network stack	3.5.1	4 000
`browsers/`	The web browser `mothra`	3.5.2	15 000
`utilities/`	Utilities such as `ls`, `cp`, `mv` `grep`, `gzip`, `tar`, `sed`, `cmp`, `xargs`, `ps`, `awk`, etc	3.6	28 000
Total			365 000

Table 2.1: Main source code directories of my Plan 9 repository.

`brk()` [4]), and other are used for file input and output (IO) (e.g., `open()`, `close()`, `read()`, `write()` [5]).

In fact, those IO system calls provide an extended way for programs to request services from the kernel by using the filesystem hierarchy. Indeed, one of the philosophy of UNIX is that *everything is a file*, including the devices. A process using the `/dev/cons` *device file* will trigger code in the kernel in `kernel/console/devcons.c`; this code will then trigger code that handles the keyboard device in `kernel/devices/keyboard/` (when reading from `/dev/cons`), or the screen device in `kernel/devices/screen/` (when writing to `/dev/cons`).

This philosophy was pushed even further under Plan 9 where *everything is a file server*. The graphics API for instance is accessible though many files under `/dev/draw/`, and triggers code in `kernel/devices/screen/devdraw.c` and `lib_graphics/`. In a similar way, the networking API is accessible through files under `/net/` and triggers code in `kernel/network/`.

A second separation to be made is between *library code* and *application code*. All the user programs in Plan 9 rely first on the core C library in `lib_core/libc/`, exposed via the `include/core/libc.h` header file; all programs are linked with `libc.a`. The other `lib_xxx/` directories contain other general-purpose functions and data structures, which are used by different programs. By using libraries, source code can be reused more easily.

[4] with implementation in `kernel/memory`

[5] with implementation in `kernel/files/`

The shell is a regular program using also the C library. The assembler, linker, and compiler rely on the C library as well as other header files, for instance, `include/obj/5.out.h`, which declares the set of ARM opcodes.

All the graphical applications, for instance, the clock, but also the windowing system, rely also on the `lib_graphics/lib_draw/` library, which is exposed in the `include/graphics/draw.h` header file. The graphics library is essentially a thin wrapper over the protocol used by the `/dev/draw/` device files. In the same way, the `lib_networking/libip/` library, exposed in the `include/net/ip.h` header file, is a thin wrapper over the protocol used by the `/net/` device files.

Chapter 3

The Literate Programs

I now switch to quick descriptions of the 18 programs, and so 18 books, composing the Principia Softwarica series. I organized those programs in six different groups.

3.1 The Core system

The first group, which I call the *core system*, is made of the minimal set of programs that are needed to reach the point where the programmer can interactively launch other programs. In the case of Plan 9, this minimal set is made of the kernel, the shell, and the standard C library.

The C library is necessary because it is used by the shell. Indeed, the C library provides the necessary bridge to call the kernel from user-space programs. The library provides also memory allocation routines and a few general utility functions. In fact, a small part of the C library is also used internally by the kernel.

3.1.1 The kernel: `9pi`

A *kernel* is a program that manages all the other programs. It is arguably the most important program; without the kernel no other program can run. The kernel provides the main abstractions of the computer and multiplex its resources (e.g., the processor, the memory, the input and output devices) to multiple programs at the same time.

The kernel is the biggest program and so the biggest book in the series. The Plan 9 kernel is called simply `9`, but I will focus on a variant of the kernel for the Raspberry Pi called `9pi`.

3.1.2 The core libraries: `libc`, `libthread`, `libbio`, `libflate`, `libregexp`

The Plan 9 C library `libc` defines data structures and functions that are used by all the other programs (including the kernel). In the case of Plan 9, this library contains the memory allocation routines (`malloc()`, `free()`), the bridge to call the kernel (the system calls), the Unicode functions, and many other utility functions. Note that some of the code in the C library is written in assembly.

In addition to `libc`, a few other core libraries are used by many programs: `libthread` for concurrency, `libbio` for buffered IO, `libflate` for compression, and `libregexp` for regular expressions, that I will also present.

3.1.3 The shell: `rc`

A *shell* is a program that allows a user to launch other programs. It is the primary user interface of Plan 9, the so called *command-line interface* (we will see later another interface).

In Plan 9, and also in its ancestor UNIX, the shell is a regular user-space program; it is not part of the kernel. The user can actually change shell without changing the kernel.

The shell will be the first user program I will describe in the Principia Softwarica series. It illustrates many features provided by the kernel, because the shell is using internally many system calls (e.g., `rfork()`, `exec()`, `chdir()`, `pipe()`).

The Plan 9 shell is called `rc` (for `run` command).

3.2 The development toolchain

Once you have a terminal where you can launch programs, you will need tools to produce those binary programs from source code. The *development toolchain* is a set of tools working together to produce executables. This toolchain is made essentially of a compiler, an assembler, and a linker.

3.2.1 The C compiler: `5c`

A *compiler* is a program that transforms source code written in one high-level language (e.g., C), into another lower-level language (e.g., assembly). The C compiler is probably the most important program in Plan 9 after the kernel. Indeed, the Plan 9 kernel itself is written in C and so needs a C compiler to become executable by a machine. In fact, almost all programs in Principia Softwarica are written in C, including the compiler itself[1], so they all need the C compiler to become executables.

I will describe `5c`, a C compiler targeting ARM assembly. The `5` in `5c` comes from the Plan 9 convention to name architectures with a single character (`'0'` is MIPS, `'5'` is ARM, `'8'` is x86, etc). This is why the ARM assembler and linker I will describe later are called respectively `5a` and `5l`.

I chose the ARM architecture because it is one of the simplest architecture while being also one of the most used architecture in the world. Indeed, almost every phone contains an ARM processor. It is also the processor of the extremely cheap Raspberry Pi a machine used by many electronic hobbyists. Thus, ARM is a great candidate for my teaching purpose.

3.2.2 The assembler: `5a`

An *assembler* is a kind of compiler; it translates source code written in an assembly programming language into machine code, or into an object code close to machine code. Some of the kernel code, as well as code in the C library, are written in assembly.

To be consistent with the compiler, I will describe `5a`, the Plan 9 ARM assembler.

3.2.3 The linker: `5l`

A *linker* is a program taking as input multiple files called the *object files*, which contain object code generated either by the assembler or compiler, and creates the final executable. In theory, compiling, assembling, and linking could be done by a single program; the C compiler could take as input multiple C source files and produce directly an executable[2]. However, from a software engineering point of view, it is better to separate concerns and have three separate tools for those three separate tasks.

I will describe `5l`, the Plan 9 ARM linker.

3.2.4 The processor emulator: `5i`

An *emulator* is a program that acts like a computer and can interpret another (binary) program. An emulator is not really a part of the development toolchain. However, because the assembler, linker, and compiler target an architecture, it is useful to describe the instruction set of an architecture (ISA) through its emulator. Moreover,

---

[1] Appendix B describes how to solve this self-reference issue.

[2] Actually `gcc` can be used that way.

an emulator can be extremely useful when transforming a compiler in a *cross compiler*, to test quickly the generated machine code.

I will describe `5i`, an ARM emulator, which we can view as a kind of interpreter for a low-level language, hence the `i`. In fact, `5i` can also be used as a (slow) debugger and profiler.

3.3 The developer tools

The *developer tools* are a set of tools that are not strictly necessary to produce programs, like the software development toolchain I have described in the previous section, but which are useful in the software development process. Those tools are the text editor, the build system, the version control system, the debugger, and finally the profiler.

3.3.1 The text editor: `ed`

A *text editor* is a program to help read and write text, including source code. It is perhaps the most important tool for a programmer. Plan 9 has a few text editors: `ed`, `sam`, and `acme`. I chose to present `ed`, which was originally written by Ken Thompson in the 1970s for UNIX. `ed` is a line editor, as opposed to screen editors like `sam`, `acme`, `vi`, or Emacs.

The main reason I chose `ed` is that it is small enough (under 2000 lines of C) to fit in one appendix. This makes `ed` a perfect candidate to illustrate literate programming in a concrete way, complementing the abstract presentation in Section 1.5. The complete source code of `ed` is presented in Appendix C.

3.3.2 The build system: `mk`

A *build system* is a program to help automate the compilation of programs by calling the appropriate tools from the development toolchain. The whole Plan 9 operating system can be built from scratch with one simple command, `mk all`, thanks to the build system called `mk` and a few configuration files (the `mkfile`s).

3.3.3 The version control system: `git9`

A *version control system* (VCS) is a program to store and retrieve past versions of a file, and to record who made each change, when, where, and why.

The Plan 9 authors didn't use originally a VCS to store the code of Plan 9 and so Plan 9 didn't have any VCS for a long time. They relied instead on powerful snapshot-based filesystems allowing to retrieve easily past versions of a file. However, in the modern age VCSs are useful not only to retrieve past versions of a file but also to help people collaborate with each other and to support concurrent and parallel development. Fortunately, in 2021 Ori Bernstein decided to rewrite Git, the most popular open source VCS, for Plan 9 and redesigned the program to use far less code. I will present his Git port called `git9`, as well as the code of simplified versions of `diff` and `patch` he wrote.

3.3.4 The debuggers: `acid`, `ratrace`, `db`

A *debugger* is a program that controls and inspects another program. Programming is so difficult that invariably we make mistakes when writing code. Having tools to help find those mistakes is essential, and the debugger is the most important of those tools. Even if many programmers, including great programmers [Sei09], use just `printf()` tracing instructions to debug their programs, I think a debugger can greatly accelerate the time it takes to find bugs. Debuggers are also great tools to help understand programs, especially programs written by other people.

Plan 9 has a few tools to help debug programs: `db` the original debugger coming from UNIX, `ratrace` to *trace* other programs (similar to `strace` in Linux), and `acid` a *programmable debugger*. I will describe all of them but mostly focus on `acid` as well as the `libmach` library that abstracts over the executable file format and the architecture-specific machine details (registers, stack frames, binary instructions), and on which all those debuggers rely.

3.3.5 The profilers: `prof`, `tprof`, `stats`, `iostats`

A *profiler* is a program that generates statistics about another program. It is mainly used to optimize code by spotting where the program spends most of its time. It can be useful also to find bugs because unexpected statistics can sometimes be good hints to fix code where the programmer was expecting different results [Ben88]

Profiling is not a single problem but a spectrum of needs: timing a command, identifying hot functions, or monitoring system-wide resource usage. Plan 9 addresses each with a small focused tool: `prof` and `tprof` for function-level profiling, `stats` for live system monitoring, and `iostats` for I/O cost analysis. I will describe all of them.

3.4 Graphics

Up until now, I have mostly described command-line tools interacting with the programmer through simple text-based terminals. However, one of the most important inventions in computer science was the *graphical user interface* (GUI), introduced in the 1970's with the Xerox Alto [TML+79]. Even command-line text-based programs benefit from a graphical user interface as you can run multiple programs in different windows at the same time like in Figure 1.1 on page 8.

3.4.1 The graphics stack: `/dev/draw`

A *graphics stack* provides the basis on top of which graphical applications can be built. GUI elements such as menus, windows, cursors, or texts are ultimately rendered on the screen using simple graphic operations provided by the graphics stack: lines, circles, rectangles, arcs, etc.

Under Plan 9, the graphics services are accessible through the `/dev/draw/` device directory, which is connected to the screen device driver in the kernel. On the user side, applications do not talk to `/dev/draw/` directly but through `libdraw`, which provides drawing primitives and event handling, and `libimg`, which reads and writes standard image file formats (PNG, JPEG, GIF). On the kernel side, the `/dev/draw/` driver itself relies on `libmemdraw` for in-memory bitmap operations and `libmemlayer` for layer stacking and clipping.

3.4.2 The windowing system: `rio`

One of the most important graphical applications is the *windowing system*. In some sense, a windowing system is an extension of the kernel and the shell; it is a program that also manages and launches other programs, which are represented visually by separate windows.

The Plan 9 windowing system is called `rio`. It is essentially a multiplexer of the `/dev/cons`, `/dev/mouse`, and `/dev/draw` device files. `rio` is a file server that internally uses the keyboard, mouse, and screen physical devices, and provides a virtual keyboard, virtual mouse, and virtual screen to its windows using views of those same device files (thanks to the per-process namespace feature in the kernel). In fact, an unusual feature of `rio` is that it can be run inside itself.

3.4.3 The graphical user interface toolkit: `libpanel`

A *GUI toolkit* provides the high-level *widgets* from which graphical applications are built: buttons, menus, sliders, scroll bars, and the layout managers that arrange them on screen. A toolkit sits above the graphics stack and the windowing system: it turns drawing primitives (lines, rectangles, text) and raw input events (mouse clicks, key presses) into reusable interactive elements.

The Plan 9 toolkit is called `libpanel`.

3.5 Networking

Up until now, I have described programs that can run on isolated machines. I will now switch to programs that can communicate with each other on different machines.

Just like the graphical user interface, *networking* (and internetworking, also known as the Internet) was one of the most important inventions in computer science. It has led ultimately to the creation of the Web where networking programs (e.g., web browsers and servers) made possible great things, for instance, the worldwide collaborative project Wikipedia.

3.5.1 The network stack: `/net`

A *network stack* is a part of the kernel, usually fairly large, that provides the necessary abstractions for programs to communicate with each other on different machines, by using different protocols.

Under Plan 9, the network services are accessible through the `/net/` device directory.

3.5.2 The web browser: `mothra`

There are far too many networking applications to present in a single book—Email, Usenet news, FTP, telnet, and IRC, all played a role in the early Internet. I will focus on the one that has eclipsed all the others: the Web browser.

A *Web browser* is a program that fetches *hypertext* documents over the network and renders them on the screen. A modern Web browser can read mail, transfer files, chat, play video, and much more; it has grown so versatile that some treat it as an operating system on its own.

Plan 9 has a few experimental Web browsers but I will present only `mothra`, the simplest one. It is using the `libpanel` GUI library I mentioned earlier. `mothra` is much more limited than Firefox or Chrome—no JavaScript, no style sheets —but its few thousand lines of C are small enough to read end-to-end and to understand how a browser is structured.

3.6 Utilities: `cat`, `ls`, `grep`, `sed`, `cmp`, `tar`, `gzip`, `bc`, `dc`, `hoc`, `awk`, ...

In addition to the programs I mentioned previously, a programmer very often uses small utilities to perform or automate certain tasks. The code of those utilities is usually pretty small because those utilities have often a single and simple function. One of the Principia Softwarica books will be dedicated to describe the code of a few of those utilities, the most important ones for the programmer.

File and directory utilities

The shell has very few *builtins* (e.g., `cd`); to create, delete, or modify files or directories, a programmer needs to use special programs. Those programs are essentially small wrappers around the file and directory system calls provided by the kernel.

Here are the file and directory utilities I will describe:

- `touch` and `mkdir`
- `cat` and `ls`
- `rm`, `cp`, and `mv`
- `chmod` and `chgrp`

Text processing utilities

Source code is stored in text files, which are made of a set of lines, which are a set of strings (words). It is thus normal that string-processing utilities are very often used by programmers to search, modify, or compare source code. Here are the string-processing utilities I will describe:

- `grep`: It is one of the most versatile and useful tool for a programmer. It can be used to find code using regular expressions.
- `sed`: It can be used to help refactor code.
- `awk`: It generalizes the regexp-driven idea of `grep` and `sed` to a small pattern-action language that handles not just strings but also numeric computations, making it a very versatile and useful tool for a programmer.

Process utilities

Source code ultimately is transformed into binary programs, which ultimately become processes when run. A programmer needs a set of utilities to manipulate those processes. Under Plan 9, the `/proc` directory can be used to inspect and manipulate those processes. An alternative is to use command-line programs that are often small wrappers over `/proc`.

Here are a few process utilities I will describe:

- `ps` and `pstree`
- `kill`

Archive utilities

Once a program has been written, a programmer often wants to share it with the world. In this case, he can count on a few utilities to *package* and *compress* code.

Here are a few utilities used to help disseminate programs:

- `tar`
- `gzip`

3.7 Applications

I decided to limit Principia Softwarica to system programs, and to the system programs that are the most relevant to the programmer. I do not cover applications such as spreadsheets, word processors, calendars, email clients, or video games. However, I encourage other people to find applications with small codebases (using minimalist approaches), and to write and publish their literate programs.

Chapter 4

Conclusion

I hope the Principia Softwarica books will greatly consolidate your computer science knowledge, and give you a better and more complete picture of what is going on in your computer.

I think the Principia Softwarica programs form together the minimal foundation on top of which all applications can be built. Even though there are 18 books in the series, I still think it is the minimal foundation. Indeed, it is hard to remove any of those programs because they depend on each other. First, you need to rely on a kernel (hence the name), but the shell and the C library are also essential. However, because those programs are coded in C and assembly, you also need a C compiler and an assembler, and because source code is usually split in many files you also need a linker. To write all this code in the first place you need an editor. Then, with so many source files you need a build system to automate and optimize the compilation process, and a version control system to keep track of changes to those files. Because the programs I just mentioned inevitably have bugs or non optimal parts, you will also need a debugger and a profiler. Finally, nowadays it is inconceivable to not use a graphical user interface and to not work with multiple windows opened at the same time. In the same way it is also not conceivable to work in isolation; programmers collaborate with each other, especially via the Web. This means you need a graphical and networking stack as well as a windowing system and a Web browser. As I said earlier, it is hard to remove any of the programs from the series.

I hope those books will answer many of your questions, even those that seem very simple at first such as "What happens when the user type `ls` in a terminal window?". The answer to this question involves many software layers (the shell, the C library, the kernel, the graphics stack, and the windowing system) and lots of code.

The books in the series can be read mostly in any order. You do not have to read them all. I recommend to pick the program you are the most interested in, for instance, the one you are the most curious about because you have only a vague idea of how they are implemented, and read the corresponding book in the series.

Enjoy!

Appendix A

Other Teaching Operating Systems

Here are a few teaching operating systems that I considered for Principia Softwarica, but which I ultimately discarded:

- UNIX V6[1] (Ken Thompson et al.), fully commented in the classic book by John Lions [Lio77], or its modern incarnation xv6[2], are great resources to fully understand a UNIX kernel. However, this kernel is too simple; there is no support for graphics or networking for instance.

- XINU[3] (Douglas Comer), fully documented in two books [Com84, Com87], has a network stack, but the kernel is still too simple with no virtual memory for instance.

- Minix[4] (Andrew Tannenbaum et al.), also fully documented [Tan87], is fairly small, but it is just a kernel. Minix does not provide for instance its own windowing system; it relies instead on X Window, which is far more complicated than the Plan 9 windowing system.

- Hack[5] (Noam Nisan and Shimon Shocken) is a toy computer introduced in the excellent book *The Elements of Computing Systems* [NS05]. This book is great for understanding processors, assemblers, and even compilers, but the kernel part is really too simple.

- MMIX[6] (Donald Knuth) and its ancestor MIX are computers designed by Donald Knuth and used in his classic book series *The Art of Computer Programming* [Knu73]. Donald Knuth also wrote a book using literate programming, *MMIXware* [Knu99], to explain the full code of the MMIX simulator and assembler. However, similar to Hack, very few programs have been written for this machine. For instance, the book assumes the presence of a kernel called NNIX, but nobody has ever written it.

- STEPS[7] (Alan Kay et al.) is a project to reinvent from scratch programming. It has a far more ambitious goal than Principia Softwarica: write a full operating system in 20 000 LOC. It was unfortunately never finished.

- Oberon[8] (Niklaus Wirth et al.) is a kernel, compiler, and windowing system designed from scratch. It is a great operating system, very compact, and fully documented in a book [WG92]. However, it imposes strong restrictions on the programmer: only applications written in the Oberon programming language can be run. This simplifies many things, but operating systems like UNIX (and Plan 9) are more universal;

[1] http://minnie.tuhs.org/cgi-bin/utree.pl
[2] http://pdos.csail.mit.edu/6.828/2014/xv6.html
[3] http://minnie.tuhs.org/cgi-bin/utree.pl?file=Xinu7
[4] http://minnie.tuhs.org/cgi-bin/utree.pl?file=Minix1.1
[5] http://www.nand2tetris.org/
[6] http://www-cs-faculty.stanford.edu/~uno/mmix-news.html
[7] http://vpri.org/html/writings.php
[8] http://www.projectoberon.com/

they can run any program in any language, as long as the program can be interpreted or compiled into a binary.

- TempleOS[9] (Terry A. Davis) is an operating system single handedly created over a decade. It contains a kernel, a windowing system, a compiler for a dialect of C, and even some games. It has graphics capabilities but there is no network support.

Another candidate for Principia Softwarica was the combination of the GNU system[10], the Linux kernel[11], and the X Window graphical user interface Xorg[12]. However, GNU/Linux/Xorg together are far bigger than Plan 9. If you take the source code of the Linux kernel, the GNU C library (`glibc`), the `bash` shell, the GNU C compiler (`gcc`), the GNU assembler (`gas`) and linker (`ld`) part of the `binutils` package, the GNU Lex and Yacc clones (`flex` and `bison`), the Emacs editor, GNU `make`, the GNU debugger (`gdb`), the GNU profiler (`gprof`), and the X Window system (`Xorg`), you will get orders of magnitude more source code than Plan 9, even though Plan 9 provides in essence the same core services. In fact, almost all of the programs above use individually more source code than the whole Plan 9 system.

Of course, the Linux kernel contains thousands of specific device drivers, `gcc` handles a multitude of different architectures, and `Xorg` supports lots of graphic cards. All of those things could be discarded when presenting the core of those programs. However, their core is still far bigger than the equivalent core in Plan 9 programs.

[9] http://www.templeos.org/
[10] http://www.gnu.org/
[11] http://www.kernel.org/
[12] http://www.freedesktop.org

Appendix B

Bootstrapping from Scratch

It is currently fairly easy to install Plan 9 on a new machine, as explained in Section 1.6. This is because there are already in the world computers running operating systems that include executable C compilers and text editors, with standard formats for data as well as compatible storage devices. This is also because the source code of Plan 9 can easily be downloaded on the storage device of those computers through the network. All of this makes it easy to compile (or cross compile) Plan 9 on one machine and install it on the disk of a new machine that can be booted on. In fact, you can even use an emulator like Qemu and run Plan 9 from the same machine.

However, what if we had to start from scratch?

- What if there was no executable C compiler? As mentioned in Section 3.2.1, the C compiler I describe in Principia Softwarica is itself written in C. This self-reference leads to a *chicken and egg* problem: How was compiled the code of the first compiler?

- What if there was no digital text of Plan 9 and no text editor? If the only representation of Plan 9 was the printed Principia Softwarica books, the source code of Plan 9 would have to be entered first in a computer. But without a text editor, how to enter and save text in a computer? How was entered the source code of the first text editor?

- What if there was no kernel? How to interact then with a new machine? If most programs are loaded into memory by the kernel, which can be seen as an interactive program loader, how was loaded in memory the first loader?

In fact, the answer to all those questions is a technique called *bootstrapping*. According to Wikipedia, “bootstrapping in general refers to the starting of a self-sustaining process that is supposed to proceed without external input”. In our case, bootstrapping is a process where you start from something simple and gradually build something self-sustaining and more complex.

In the book *The Knowledge: How to Rebuild our World from Scratch* [Dar14], its author Lewis Dartnell imagines our world after an apocalypse and describes the key knowledge one needs to start rebuilding civilization from scratch. He does not reach though the computer and software age. What if after this apocalypse there was no more software? What if the only programs we had was the printed Principia Softwarica books retrieved from a time capsule? Would that be enough to bootstrap Plan 9? The kernel, compiler, and editor being mutually dependent on each other, it is difficult to imagine where to start. Which program to write first? For which machine? In which language? And how to enter this program in the machine?

In the following sections, I outline a bootstrapping process for Plan 9 where I start from scratch. I think it is an interesting intellectual exercise. By scratch I mean with no existing software, but because in Principia Softwarica we are interested in software and not hardware, I assume the existence of a basic computer though.

B.1 A basic computer

Software and hardware need each other in order to be useful, but hardware is the concrete starting point. Before executing programs, we first need a physical machine, a computer.

The main components of a basic computer are as follows:

- A processor: This is an interpreter for a simple low-level language where instructions are encoded in a binary format.
- Some memory: This contains data but also the code of programs. The *stored-program* concept was one of the major inventions of computer science.
- Input and output devices: This is used for interactivity with the user.

To learn how to build a simple processor and memory, I recommend to read [NS05] in which starting only from the `nand` logic gate, you learn how to build gradually the `and`, `or`, and `not` logic gates, a multiplexer, flip-flops, memory banks, an additioner, an arithmetic and logic unit (ALU), and finally a simple central processing unit (CPU).

Regarding the external devices, a basic computer typically has three:

- An input device, e.g., a keyboard, for reading data. The press of a key can trigger an interrupt in the processor that will trigger the execution of a special program. The value of the key could be read at a certain memory location (memory-mapped IO) or via a special instruction. Before keyboards, *switches* on *panels* or *punched cards* with card readers connected to the computer were the main ways to enter data in a computer.
- An output device, e.g., a screen, for displaying data. Again the device could be memory-mapped, in which case the writing of data at certain memory locations would trigger the display of special characters on the screen. Before screens, diods on panels or printers connected to the computer were the main ways to display data from the computer.
- A storage device, e.g., a magnetic tape with its tape drive, to permanently store data, for instance programs in executable forms and source forms. Again, access to this device could be provided by special instructions or by writing and reading certain codes at certain memory locations.

All those things can be built from scratch. They are not trivial to build, but they do not require any software; they are physical artifacts.

B.2 A booting procedure

Once you have a basic computer, an important question is what happens when a human presses the On/Off button of the computer? The initialization of a computer system is called *booting*, which is the shortened form of the word bootstrapping. The machine needs to initialize itself and execute the first program from memory. If most programs are loaded into memory by other programs already running on the computer, what is the mechanism to load the very first program?

There are multiple ways to load an initial program in memory. The very first computers used switches on panels where the initial configuration of memory could be manually toggled, or at least part of the memory could be toggled. Then the computer when turned on, by construction, would start to execute the code located at a specific memory address, e.g., `0x10`. This initial program, usually small, is called the *boot loader*. Its job is just to load a larger program stored on the main storage device: the kernel (which itself will load other programs).

Later, computers used punched cards and a card reader connected to the computer to load automatically their first program in memory. Modern computers use special read-only memory (ROM) chips to store the initial program. Such programs are also called *firmware*, because they are in the middle between the hard and soft.

In any case, all those techniques are similar: a program is hardwired at a persistent location (panel, punched card, tape, ROM) and loaded in memory at a specific address by the computer when the human presses a special button. The computer then jumps to this memory address and starts executing the program.

B.3 Physical programs

We have just seen different ways to enter programs in a computer without a text editor; switches and punched cards are ways to write data, and so also programs, in the physical world. Thanks to panels and card readers connected to the computer, those physical programs are made machine readable.

Different kinds of punched cards and card readers can be used to write different kinds of information:

- Different holes for different bits can be used to represent binary data such as machine instructions.
- Different holes for different letters or numbers can be used for textual data used by programs.
- Different holes for different instructions of specific languages can be used for the source code of languages such as assembly.

Note that a card reader (or a keypunch machine) could be built in such a way to behave like a primitive mechanical assembler. The holes in the punched card could correspond to different assembly mnemonic instructions and be transmitted to the computer as the assembled binary machine instructions.

Once those physical programs are loaded into memory, they can be stored back on magnetic tapes thanks to special computer instructions.

We now have everything we need to start producing software:

- A basic computer, which can be seen as a machine-language *physical interpreter*. This will allow us to bootstrap the compiler.
- A way to enter programs in the computer, with punched cards, a keypunch machine, and a card reader, which can be seen together as a *physical text (and binary) editor*. This will allow us to bootstrap the text editor.
- A simple booting procedure which will load the boot loader from a fixed *physical location*. This will allow us to bootstrap the kernel.

The following sections describe the process to bootstrap Plan 9 from scratch by writing a series of programs. The process is decomposed in different phases depending on the programming language used.

B.4 Phase 1: Machine code

`BOOT-M-CARD`

The very first program to write is a boot loader, which I call `BOOT-M-CARD`. `M` stands for machine code and `CARD` for punched card. The first program has to be written directly in machine code and entered in the computer via a punched card as there is no assembler and no text editor yet. The bits of the instructions of this program can be encoded as holes in the punched card. This card can then be inserted in the first position of the card reader and loaded at boot time by the computer according to its booting procedure.

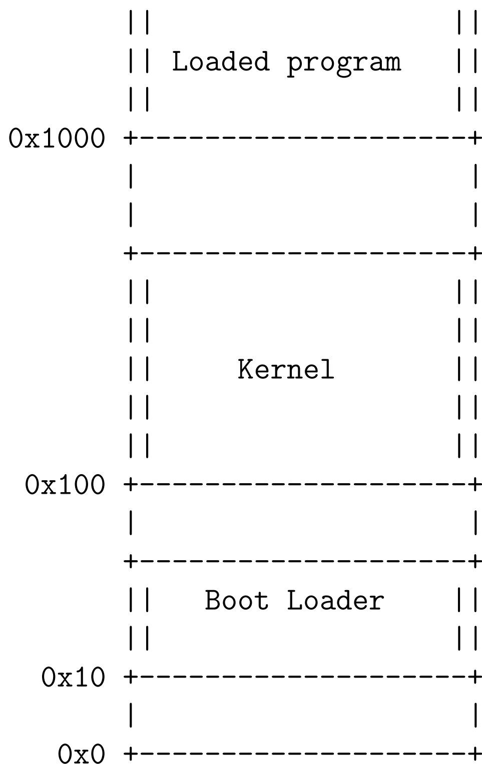
One card should be enough to encode a very simple boot loader. With just a few instructions, this program could load in memory the rest of the punched cards from the card deck (or data from the tape), and so bootstrap an hypothetical kernel. It must load this code after is own code though, to not overwrite itself, e.g., at address `0x100`, and then jump to this address.

To create `BOOT-M-CARD`, we could have created first `BOOT-ASM-PAPER`, the boot loader program written in assembly on paper. Then, thanks to a very powerful (but error-prone) computer and assembler, the human[1], we could have translated this program in binary and then generate `BOOT-M-CARD`.

KERNEL0-M-CARD

The next program to write is a simple kernel, which I call `KERNEL0-M-CARD`. It is essentially an interactive program loader. With this loader, you could type on the keyboard after a prompt the location of a program in order to get this program loaded in memory and jumped to.

The organization of different programs on the tape or in the card reader can be trivial: the different programs could be stored one after the other. The location of a program could then be specified simply by a letter and two numbers: the letter to select the card reader or tape drive, the two numbers for the start and end locations of the program on the tape or in the card deck. Moreover, by convention programs could be loaded at the memory address `0x1000` and executed from there. The last instruction of a program could be to jump to the address `0x100` to give back control to the kernel. The following diagram describes the memory layout after a program was loaded by the kernel:



There is no need yet for a real filesystem or an executable format. The code of this kernel can be very simple; it should require only a few punched cards. Those cards can then be placed just after the punched card of the boot loader in the cards deck.

ASSEMBLER0-M-CARD

The final program to write in machine code is a rudimentary assembler, which I call `ASSEMBLER0-M-CARD`. Again, we could write first `ASSEMBLER0-ASM-PAPER` and then translate this program in machine code by using a human assembler.

Because we do not have yet a text editor, this assembler when loaded could ask first a series of questions to the programmer, and then ask interactively to the programmer to enter via the keyboard the full content of

[1] Not just its brain, but also its powerful IO devices: the hands that can type on a keypunch machine.

the assembly program (without mistakes). The assembler could also take as input the content of an assembly program from cards or the tape, depending on the answer to the initial questions. Another question could be used to specify where to save the generated (assembled) machine-code program.

The punched cards of this assembler program can be put just after the cards of the kernel in the card deck. You can then count the number of cards in the deck to know the location numbers to enter in the prompt of the kernel to load this assembler program in memory.

B.5 Phase 2: Assembly

Thanks to `BOOT-M-CARD`, `KERNEL0-M-CARD`, and `ASSEMBLER0-M-CARD`, we can now enter via the keyboard programs in assembly, a major productivity improvement over punched cards and binary machine-code.

`EDITOR0-ASM0-KBD`

For improving even more productivity, the first assembly program to write is an editor, which I call `EDITOR0-ASM0-KBD`. It is written in the rudimentary assembly `ASM0` supported by the rudimentary assembler `ASSEMBLER0-M-CARD`, and bootstrapped via the keyboard input of the assembler. The assembler can then generate `EDITOR0-M-TAPE`, an executable editor, which can be loaded from tape.

Thanks to this editor, further programs can be entered via the keyboard, edited, and saved on tape. We can make mistake and fix the mistake easily. From now on I assume every programs will be entered via the text editor and saved on tape, so there is no need anymore for the `CARD`, `TAPE`, or `KBD` suffixes in the program names.

`ASSEMBLER1-ASM0`, `ASSEMBLER2-ASM1`, etc

Thanks to `ASSEMBLER0-M` (previously called `ASSEMBLER0-M-CARD`), we can now bootstrap an assembler written in itself: `ASSEMBLER1-ASM0`. In fact, we can write a series of increasingly powerful assemblers. Each assembler can add features, be assembled by the previous generation assembler, which will generate a new binary assembler, e.g., `ASSEMBLER1-M`, enabling now programmers to write assembly programs using those new features, including the assembler program itself. Hopefully at some point there will be no need for more features or optimizations and we will reach a fixpoint with an assembly language I call `ASM` from now on.

`EDITOR1-ASM`

Given this feature-rich assembly language `ASM` and assembler, we could rewrite the editor that was originally using the rudimentary assembly `ASM0`. I call `EDITOR1-ASM` this new editor. Again, we could write a series of increasingly powerful editors where each new feature added could be used to edit more quickly the code of the next version of the editor.

`KERNEL1-ASM`

We can now use a nice editor and program in a powerful assembly language. However, until this point we are still using the basic program loader `KERNEL0-M`. With this kernel, you need to enter the start and end locations on tape of a program to get it executed, which is inconvenient. Indeed, with the multitude of programs we previously created, you need to maintain a physical map of the tape to remember where the programs (in source and binary forms) are located. Moreover, the programs have to be stored contiguously on the tape. Finally, you must take care when using the editor to not create data that would overwrite another program.

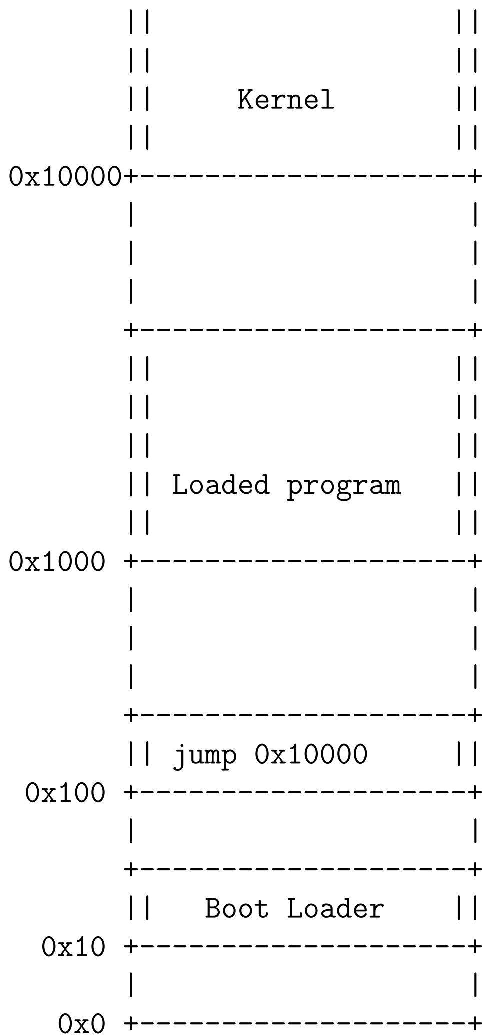
All of this management of data, currently manual and error-prone, can be done instead by the computer. I call `KERNEL1-ASM` a new kernel written in assembly with one important new feature: a *filesystem*. Transitioning to this new kernel can be facilitated by having two tape drives connected to the computer and two programs

`FSINIT-ASM` and `CP-ASM`, which can respectively initialize a new filesystem on a tape, and transfer data from one "raw" tape to another tape managed by a filesystem, and to give a filename to this data.

The start of a tape managed by a filesystem can still be reserved to contain the boot loader and the kernel. By preparing carefully the second tape with the boot loader, the new kernel, the filesystem with the previous programs (assembler, editor, kernel) in source and executable forms appropriately named, you could then switch the first tape with the second and boot the new kernel.

The interactive program loader in `KERNEL1-ASM` would now take after its prompt symbolic filenames instead of tape locations to execute commands. Note though that the copied text editor and assembler programs (`EDITOR1-M` and `ASSEMBLER2-M`) would still use tape locations to operate so you must take care when executing those commands to not corrupt the filesystem. One solution is to now use the second tape as a scratch and use `CP-ASM` to export and import data from this tape; the old editor and assembler could read and write data on this second tape without risking to corrupt the filesystem of the first tape.

Because the code required to manage a filesystem can be large, the kernel might not fit anymore in the original `0x100-0x1000` memory interval. After being loaded by the boot loader at `0x100`, the kernel could copy itself in higher memory and just put at `0x100` a few instructions to jump back to higher memory. That way programs could still be loaded at `0x1000` without overwriting code from the kernel, as in the following diagram:



Note though that very large programs could still overwrite the kernel if their code size would reach the higher memory area where the kernel resides in. We will see later mechanisms to protect the kernel from such programs.

LINKER0-ASM, ASSEMBLER3-ASM-OBJ

Now that we have a filesystem, we can easily create many files. We can now split big programs in multiple files to separate concerns and also to factorize code among those programs in library files.

To create an executable from multiple files, we could create first a program CAT-ASM that would concatenate multiple files into one. We could then use the old program ASSEMBLER2-M on this single file (after it has been exported by CP-ASM on the scratch tape). An alternative is to create intermediate machine-code files, *object files*, and to create a *linker* I call LINKER0-ASM to produce the final executable from those object files. This program is more complicated than CAT-ASM but it enables *separate compilation*, which in the long term will save time.

To create those new object files we need to modify the assembler though and create ASSEMBLER3-ASM-OBJ. Moreover, we can use the opportunity to update also the interface of the assembler to now take filenames as arguments for input and output instead of tape locations.

LIB0-ASMs

For ASSEMBLER3-ASM-OBJ to work with filenames, code in the kernel related to the filesystem had to be copied and integrated in the assembler. Moreover, lots of code dealing with tapes, punched cards, and other devices had to be duplicated in different programs. This is unfortunate. Thanks to the linker we can now factorize code in one library source file and get the resulting object file linked in different programs.

I call LIB0-ASMs a core library written in assembly using multiple files (hence the s in ASMs). This library contains reusable code among all the previous programs. This library can provide functions to interact with devices, the filesystem, files, memory, etc. In some sense this library can also play the role of a kernel as it can abstract hardware and provide low-level functions to other programs. A system call is then simply a library call.

LINKER1-ASMs, ASSEMBLER4-ASMs, EDITOR2-ASMs

We can now reorganize the code of the previous assembly programs by splitting their code in multiple assembly files, to better separate concerns, and by removing the code that was factorized in the LIB0-ASMs library. I call those new programs LINKER1-ASMs, ASSEMBLER4-ASMs, and EDITOR2-ASMs. The interface of all those programs can also be changed to operate on filenames instead of tape locations.

KERNEL2-ASMs, LIB1-ASMs

KERNEL1-ASM has a convenient filesystem but its program loader is still rudimentary. You can load only one program at a time in memory (at 0x1000 by convention). You can load the editor, edit a file, quit, then load the assembler, wait it finishes, then run back the editor again, and so on. It would be nice to run the assembler command from the editor, or at least being able to switch between the editor and assembler without losing the state of the editor, or even better being able to edit files while a time-consuming assembling or linking job is running.

To do so requires a *multi-tasking* kernel where multiple programs loaded in memory at the same time called *processes* can execute concurrently. There are multiple ways to implement multi-tasking but the now dominant way is to write a *preemptive scheduler* using a *timer* interrupt and to leverage *virtual memory*. I call KERNEL2-ASMs a new kernel written in assembly using this technique. The first edition of UNIX was actually written in assembly[2].

Note that the timer and virtual memory requires hardware extensions to the basic computer I described in Section B.1. Thanks to virtual memory, multiple programs can still be loaded at the same (virtual) address, 0x1000, as long as their physical addresses are different. This helps for the transition to the new kernel as old programs would still work with the new kernel. Those programs could also still call code from LIB0-ASMs to interact with devices, the filesystem, etc.

[2] https://github.com/c3x04/Unix-1st-Edition-jun72

The LIB0-ASMs library could be gradually extended to offer process related services, e.g., fork(), exec(), which would allow to program the scenario I mentioned before with the concurrent use of the editor and assembler. The program loader in the kernel could also be extended to provide advanced shell features such as pipes on the command line, job control, etc.

Protection rings are another hardware extension, often associated with virtual memory, which is very useful to have in a computer. They provide the mean to protect the kernel from regular programs and also to protect programs from each other when combined with virtual memory. In practice two rings are enough. The processor then can operate in two different modes:

- *Kernel mode.* In this mode all instructions are allowed, including the ones used to modify the virtual memory mapping, or the ones to interact with devices. Those instructions are called *privileged instructions*. Moreover, in kernel mode all memory accesses are allowed.

- *User mode.* In this mode many privileged instructions and access to certain memory area are disallowed. Such accesses cause *traps* in the kernel when those operations happens.

The only way to go from user mode to kernel mode (other than by causing a trap) is through a special instruction: the *software interrupt.* Calling this instruction is also known as performing a *system call* or *syscall.*

Thanks to protection rings and virtual memory, certain memory area can be marked as protected, e.g., the high memory area where resides the kernel, in which case programs can not mess anymore with the code of the kernel by overwriting its code. Moreover, we can gradually forbid programs to access directly devices such as the screen or the keyboard so that the kernel instead can mediate and control the use of those devices.

To transition to this safer model, we can gradually modify LIB0-ASMs and move routines using privileged instructions (dealing with memory, devices, the filesystem, processes, etc) to the kernel and give access to those routines via a system call (instead of a library call) to user programs. I call LIB1-ASMs this new library, which should be significantly smaller than LIB0-ASMs. Once all the privileged code has been migrated to the kernel, we can turn on the protection rings and forbid the execution of any privileged instructions in user programs.

COMPILERC0-ASMs

The final program to write in assembly will enable the transition to a higher-level language: C. It is a compiler written in assembly for a simple subset of C. I call this program COMPILERC0-ASMs.

B.6 Phase 3: C

Thanks to the previous programs, we now have a basic UNIX environment with a simple C compiler, all written in assembly. It is now time to rewrite this environment in a more succinct and readable way using C.

COMPILERC1-C0, COMPILERC2-C1, etc

Thanks to COMPILERC0-M we can now bootstrap a C compiler written in itself: COMPILERC1-C0. In fact, just like for the assembler and editor, we can write a series of increasingly powerful compilers: each compiler can add features, be compiled by the previous generation compiler generating a new binary compiler, e.g., COMPILERC1-M, enabling now programmers to write C programs using those new features, including the compiler program itself. Hopefully at some point there will be no need for more features and we will reach a fixpoint with a C language I call C from now on.

KERNEL3-C/ASM, LIB2-C/ASM

We can rewrite lots of the core library in C. Indeed, the C language being very flexible and powerful, many of the assembly code idioms can be expressed in C, and usually in a clearer and shorter way. However, some code has

to be kept in assembly, which is why I call this new library LIB2-C/ASM. Some low-level code, such as the system call instructions that allows to jump in the kernel, can not be expressed in C. Moreover, it can be preferable sometimes for heavily used routines to keep code written in highly-optimized assembly. Those routines though can be exposed to other programs with a C interface, e.g., in a libc.h header file. Those other programs can then be written entirely in C.

The kernel itself can also be rewritten in C and assembly in a program I call KERNEL3-C/ASM. The fourth edition of UNIX[3] was the edition where most programs, including the kernel, were rewritten in C. At the time, writing a kernel in a high-level language such a C was controversial.

EDITOR3-C, LINKER2-C, ASSEMBLER5-C

We can rewrite most of the previous assembly programs entirely in C: the editor (EDITOR3-C), the linker (LINKER2-C), and even the assembler can be rewritten in C (ASSEMBLER5-C).

B.7 Summary of Plan 9 ancestor programs

I summarize here the list of programs that can be viewed as ancestors to programs explained in the Principia Softwarica series. Those programs would be necessary to bootstrap Plan 9 from scratch:

- Boot loader: BOOT-ASM (translated in BOOT-M-CARD by a human assembler). The Plan 9 kernel 9 described in Section 3.1.1 includes a boot loader fairly similar to BOOT-ASM.
- Kernels: KERNEL0-ASM (translated in KERNEL0-M-CARD by a human assembler), KERNEL1-ASM, KERNEL2-ASMs. This led finally to KERNEL3-C/ASM, which can be a close ancestor to 9 described in Section 3.1.1. We could derive KERNEL3-C/ASM from the source code of 9 by just removing many features.
- Core library: LIB0-ASMs, LIB1-ASMs. This led to LIB2-C/ASM, which can be a close ancestor to libc described in Section 3.1.2.
- C compilers: COMPILERC0-ASMs, COMPILERC1-C0, COMPILERC2-C1. This led finally to COMPILERC3-C/LY, which can be a close ancestor to 5c described in Section 3.2.1.
- Assemblers: ASSEMBLER0-ASM (translated in ASSEMBLER0-M-CARD by a human assembler), ASSEMBLER1-ASM0, ASSEMBLER2-ASM1, ASSEMBLER3-ASM-OBJ, ASSEMBLER4-ASMs, ASSEMBLER5-C. This led finally to ASSEMBLER6-C/ which can be a close ancestor to 5a described in Section 3.2.2.
- Linkers: LINKER0-ASM, LINKER1-ASMs, This led finally to LINKER2-C, which can be a close ancestor to 5l described in Section 3.2.3.
- Editors: EDITOR0-ASM0, EDITOR1-ASM, EDITOR2-ASMs, EDITOR3-C.

  This led finally to EDITOR3-C, which can be a close ancestor to ed described in Section 3.3.1.

[3] http://minnie.tuhs.org/cgi-bin/utree.pl?file=V4

Appendix C

Literate Program Example: `ed`

This appendix has two purposes. First, it is a concrete example of a literate program—a complement to Section 1.5, which presented literate programming in the abstract. Reading the source of a real program presented in literate form is the best way to see what literate programming actually buys you. Second, it is a full explanation of a real Plan 9 program: `ed`, which is small enough (under 2000 lines of C) to fit in one appendix, yet old enough and famous enough that almost every modern editors and string-processing tools descend from it in some way. Reading its source code is a quick way to see how a non-trivial UNIX tool was structured in the 1970s.

Note that the original source code has been slightly modified to be arguably easier to read. Moreover, I have sometimes added a few comments here and there also in the code. The original comments from Ken Thompson are using the standard C syntax `/**/` while my comments are using the `//` syntax, so they are easy to differentiate.

C.1 Introduction

C.1.1 A line editor

`ed`, short for "editor"[1], is one of the oldest UNIX programs, originally written by Ken Thompson in the 1970s. It is a *line editor*: rather than always displaying a file content on screen like `vi`, Emacs, or Plan 9's `sam` and `acme`, `ed` works mostly one line at a time, accepting commands from standard input and printing results to standard output (possibly displaying lines of the file depending on the input commands). This makes it look primitive compared to modern *screen editors*, but it has a crucial advantage: it can be scripted. You can pipe a sequence of editing commands into `ed` from a shell script, something that is awkward or impossible with most editors.

While you can already create and modify files using `echo` and shell redirection (`>`, `>>`), `ed` offers far more power: searching, substituting, moving lines, and operating on ranges of lines. For a quick hands-on feel, here is a complete `ed` session:

```
$ ed
i
hello
world
.
1,2p
hello
world
f hello.txt
```

[1] Plan 9 and UNIX authors did like short command names: `ed`, `mk`, `rc`, `db`, etc.

```
hello.txt
w
12
q
$ cat hello.txt
hello
world
$
```

The session above launches `ed`, opens *insert mode* with `i`, types two lines, ends the insertion with a single `'.'` on its own line, prints lines 1 to 2 with `1,2p` made of an *address range* and the `p` command (`ed` echoes the 2 lines), sets the remembered filename with `f hello.txt` (`ed` echoes it back), saves the buffer with `w` (`ed` echoes the character count written), and finally quits with `q`.

`ed` is famously silent: it prints no prompt, and every line you type is either a command or part of a text-entry session opened by `i` or `a`. For a full tutorial introduction to `ed`, see Appendix 1 of "The UNIX Programming Environment" [KP84], or Kernighan's two tutorials from the Bell Labs technical reports.

Note that many ideas that seem to "belong" to other tools actually originated in `ed`: the `s/re/replacement/` syntax used by `sed`, the `g/re/p` command that gave `grep` its name ("global regular expression print"), and commands like `w`, `q`, `a`, `i` that `vi` inherited directly. It is thus interesting to see the code of the program that started it all.

C.1.2 A scriptable editor

Here is a real example of `ed` used as a code generator during the build of Plan 9. The `mkenam` script below transforms a header file (`5.out.h`) containing assembler opcode definitions into a C source file (`enam.c`) containing a string array of opcode names:

⟨*mkenam* 35⟩≡

```
ed - ../5l/5.out.h <<'!'
v/^ A/d
1,$s/^ A/ "/
g/ .*$/s///
,s/,*$/",/
1i
char* anames[] =
{
.
$a
};
.
w enam.c
Q
!
```

To understand what this script does, it helps to see a concrete example of input and output. Here is what a snippet of the input file `5.out.h` looks like (left) and what the output `enam.c` is supposed to look like (right):

```
5.out.h (input)          enam.c (generated output)
----------------         ---------------------------
enum    As {              char* anames[] =
    AXXX,                  {
    ANOP,                      "XXX",
    AAND,                      "NOP",
    AORR,                      "AND",
    AEOR,                      "ORR",
    AADD,                      "EOR",
    ASUB,                      "ADD",
    ...                        "SUB",
};                             ...
                          };
```

The transformations from `5.out.h` to `enam.c` are: keep only lines starting with a tab followed by `A` (the opcode names), strip that prefix `A`, wrap each name in double quotes followed by a comma, and add the `char* anames[]` array header and trailing brace. Each time someone modifies `5.out.y`, the `enam.c` can be regenerated automatically thanks to the `mkenam` script called from `5c/mkfile`.

The `mkenam` script uses `ed -` (quiet mode, suppressing character counts) and a shell here-document (`<<'!'`, see the SHELL book [Pad18]) to feed commands. It first deletes lines not starting with `A` (`v/^ A/d`), then uses `s///` substitutions to reshape each remaining line into a quoted C string, and finally uses `i` (`1i`, meaning insert back at line 1) and `a` (`$a`, meaning appending at the end) to wrap the result with the array declaration. By the end of this appendix, you will fully understand every command in this script (and the code behind those commands).

The key advantage of `ed` here is that the editing commands are plain text, easily embedded in a shell script or a `mkfile` rule. Try doing the same in Emacs or Visual Studio Code—you would need an Emacs Lisp program or complex key-sequence macro recording which would not be so easy and fast to replay than `mkenam`.

C.1.3 A Unicode editor

Note that the Plan 9 version of `ed` is Unicode-ready. `ed` can read, edit, and write text that contains non-ASCII characters (e.g., accented letters, chinese characters, emoji), so before we dive into its code it is worth spending a few paragraphs on how Plan 9 represents text. A full treatment lives in the LIBCORE book [Pad16b]; this primer covers just what you need to follow the rest of `ed`.

Unicode is an abstract catalog of characters. A *codepoint* is the numeric ID of a character in that catalog: `U+0041` is "A" (65 in decimal), `U+00E9` is "é" (233), and so on up to `U+10FFFF` for the rarest emoji. *UTF-8* is one way to store/transmit those code points as *bytes*; It is an *encoding*.

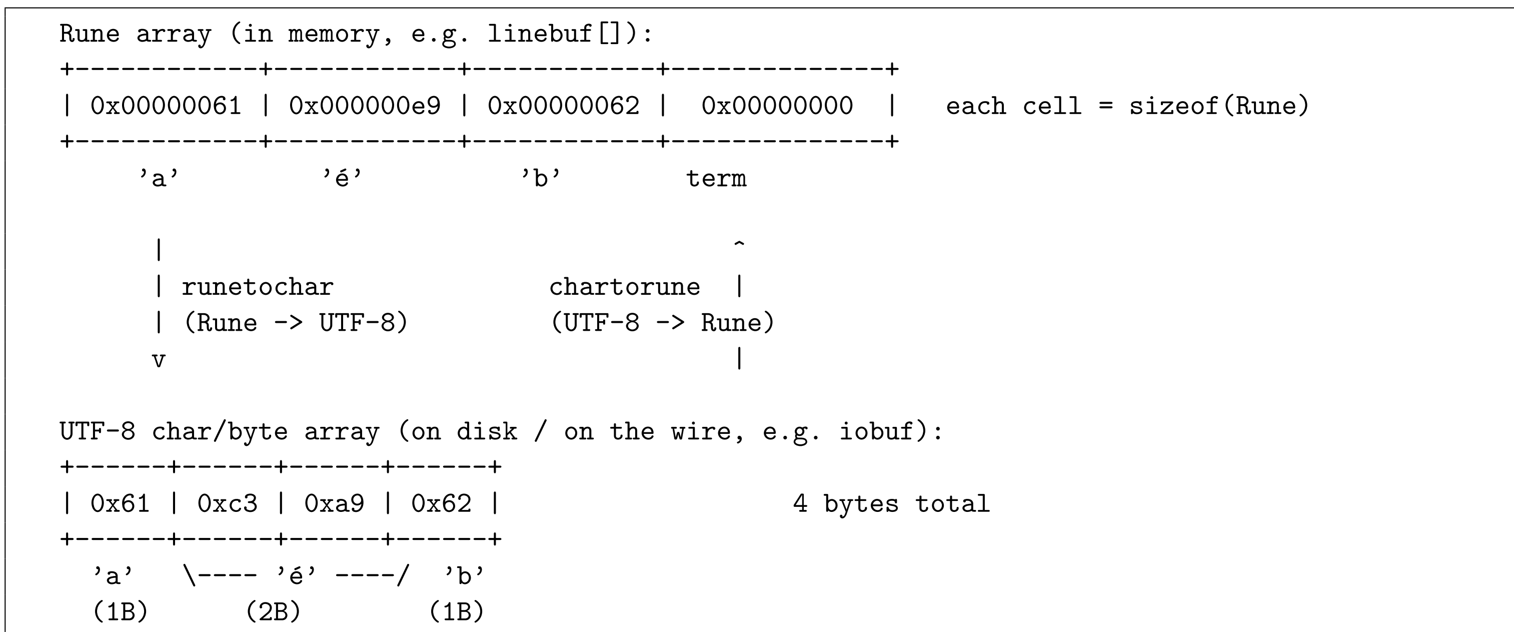
Plan 9 can store codepoints in two complementary formats. A *Rune* is a *fixed-width* unsigned integer (usually 32 bits) holding one codepoint; Rune arrays such as `Rune linebuf[LBSIZE]` area the in-memory format for text manipulation, the Unicode equivalent of `char linebuf[LBSIZE]` in classical old (pre-Unicode) UNIX programs. UTF-8 is a *variable-length* byte encoding where ASCII codepoints take 1 byte, Latin supplements take 2, most other scripts 3, and rare codepoints (including emoji) 4; UTF-8 lives in `char` arrays and is the format for files, terminal I/O, and exec arguments in Plan 9, which is using UTF-8 encoding everywhere text is involved.

The bridge between the two formats is two routines from `libc`:

```
int chartorune(Rune *r/*OUT*/, char *s); // UTF-8 -> Rune; returns # bytes consumed
int runetochar(char *s/*OUT*/, Rune *r); // Rune -> UTF-8; returns # bytes written
```

Both functions move at most `UTFmax` (usually set to 4) bytes per codepoint.

Here is the three-character string "aéb" in both formats. "é" is codepoint `U+00E9`, which UTF-8 encodes as the two bytes `0xc3 0xa9`:



`chartorune()` inspects the top bits of the leading byte to determine how many continuation bytes follow—`0xxxxxxx` is a 1-byte sequence, `110xxxxx` introduces a 2-byte sequence, `1110xxxx` introduces 3 bytes, `11110xxx` introduces 4—and assembles the codepoint. `runetochar()` does the inverse, selecting the encoding by the codepoint's value range.

The modern UNIX equivalent of Plan 9's `Rune` is `wchar_t`, a C standard type since 1995 (declared in `<wchar.h>`). On Linux and macOS `wchar_t` is 32 bits; on Windows it is only 16 bits—a legacy of Microsoft adopting UCS-2 in the early 1990s, before it became clear that 16 bits would not fit all of Unicode.

C.2 Core data structures

Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious.

Fred Brooks

The data structures in `ed` are radically different from modern text editors. There is no linked list of lines, no rope, no gap buffer. In fact, the content of the file being edited is not even kept in memory. Instead, all text is stored in a temporary file on disk (`tfile`[37b]), and the only in-memory data structure is `zero`[38d], an array of integer offsets into that temporary file—one offset per line. Editing operations like delete and move merely shuffle entries in `zero`; the temporary file itself only grows, never shrinks.

C.2.1 The backing store: `tfname`, `tfile`, and `tline`

The temporary file is `ed`'s backing store. `tfname`[37a] holds its path (e.g., `/tmp/e000042`), `tfile`[37b] is the open file descriptor, and `tline`[38a] tracks the current write offset—where the next line will be appended. Every operation that adds text (reading a file, appending user input) writes to `tfile` and records the offset in `zero`[38d].

⟨globals ed.c 37a⟩≡ (91b) 37b ▷
```
char*   tfname; // temporary filename ("/tmp/eXXXXXX")
```

⟨globals ed.c 37b⟩+≡ (91b) ◁ 37a 38a ▷
```
fdt tfile   = -1;
```
Uses `tfile` 37b.

⟨globals ed.c 38a⟩+≡ (91b) ◁37b 38c▷
```
// current write file offset in tfile (in Rune-unit)
int tline;
```

C.2.2 The target file: `savedfile`

`savedfile`[38c] remembers the name of the file being edited—the one specified on the command line or set by the `f` command. When you type `w`, this is the file that gets written.

⟨constants ed.c 38b⟩≡ (91b) 40b▷
```
FNSIZE  = 128,      /* file name */
```

⟨globals ed.c 38c⟩+≡ (91b) ◁38a 38d▷
```
// for w, r, f
char    savedfile[FNSIZE];
```
Uses `FNSIZE-1` 38b.

C.2.3 The lines as file offsets: `zero`

`zero`[38d] is the central data structure: a dynamically-growing array of `int` file offsets into `tfile`[37b], one per line. Lines are 1-indexed: `zero[1]` is the file offset for line 1, `zero[2]` for line 2, etc. The entry `zero[0]` is unused (always 0), which is why probably the array is called "zero"—it starts at the zero-th slot which represents "before line 1."

⟨globals ed.c 38d⟩+≡ (91b) ◁38c 39▷
```
ulong   nlall = 128;
// growing_array<int(even)|int+mark|0>, initial size = (nlall+2+margin)*sizeof(int)
// where the ints are file offsets (Rune-unit) in tfname corresponding to different lines
int*    zero;
```
Uses `nlall` 38d.

The crucial insight is that many editing operations only modify this array of offsets, never the temporary file itself. When you delete a line with `d`, `ed` shifts entries in `zero` down but the text remains in `tfile`. When you read a file with `r`, `ed` appends text to `tfile` and inserts new offsets into `zero`. The temporary file only grows; it is never compacted.

Concretely, suppose the user has the following text in the buffer:

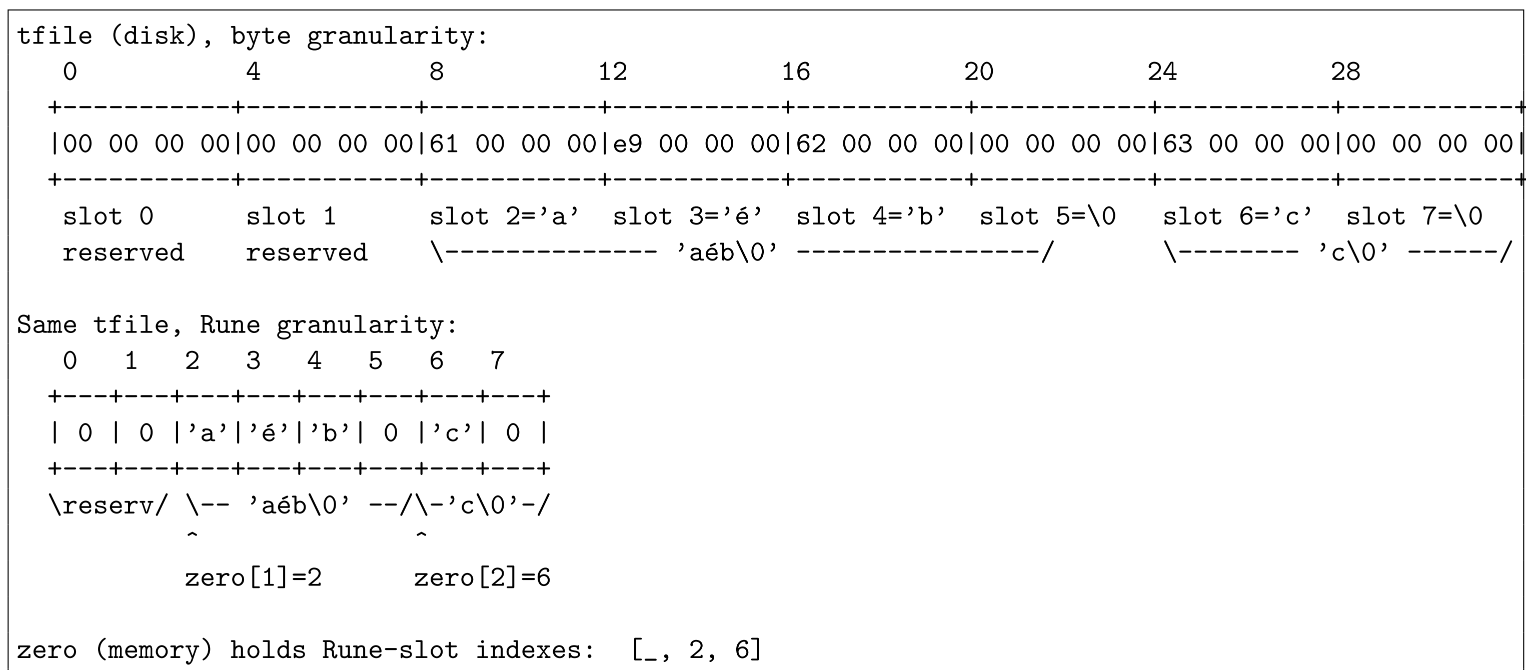
```
tfile (the temporary file on disk)         zero (in memory)
+---------------------------------+        +-------+--------+--------+--------+
|alpha\n                          |        | unused| 0      | 6      | 11     |
|beta\n                           |        +-------+--------+--------+--------+
|gamma\n                          |  index:  0       1        2        3
|a deleted line that's still here\n                line 1   line 2   line 3
+---------------------------------+
```

Three lines are visible to the user, even though the temporary file on disk also contains a fourth line at offset 17 that was previously deleted. Editing operations simply shift entries within `zero`: deleting line 2 (`beta\n`) would move `zero[3]` (the offset 11) into `zero[2]`, leaving the actual text at offset 6 in `tfile` orphaned but harmless.

The actual on-disk layout

The diagram above uses byte offsets and `\n`-terminated text because that is how most people picture a file on disk. The real layout for `tfile` differs in three ways. First, `tfile` is a stream of fixed-width `Rune` cells (see Section C.1.3), not bytes: every codepoint occupies `sizeof(Rune)` bytes (usually 4), so a 5-character line like “alpha” takes 24 bytes on disk—5 character cells plus a terminator. Second, lines are terminated by a null Rune (value 0), not by `\n`. Third, `tline`[38a] starts at 2 (in Rune units), so the first two Rune cells of `tfile` are reserved and remain zero-filled. We will see soon the reason for it.

To make the layout concrete, here is a small two-line buffer with lines “aéb” and “c” (reusing the example from Section C.1.3) drawn at both byte and Rune granularity, with `sizeof(Rune)` = 4 and little-endian Rune storage:



```
tfile (disk), byte granularity:
   0           4           8           12          16          20          24          28
  +-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+
  |00 00 00 00|00 00 00 00|61 00 00 00|e9 00 00 00|62 00 00 00|00 00 00 00|63 00 00 00|00 00 00 00|
  +-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+
   slot 0      slot 1      slot 2='a'  slot 3='é'  slot 4='b'  slot 5=\0   slot 6='c'  slot 7=\0
   reserved    reserved    \-------------- 'aéb\0' ----------------/       \-------- 'c\0' ------/

Same tfile, Rune granularity:
   0   1   2   3   4   5   6   7
  +---+---+---+---+---+---+---+---+
  | 0 | 0 |'a'|'é'|'b'| 0 |'c'| 0 |
  +---+---+---+---+---+---+---+---+
  \reserv/ \-- 'aéb\0' --/\-'c\0'-/
           ^              ^
           zero[1]=2      zero[2]=6

zero (memory) holds Rune-slot indexes:  [_, 2, 6]
```

C.2.4 Cursors: `dot` and `dol`

`dot`[39] and `dol`[39] are pointers into the `zero`[38d] array. Their names match the corresponding ed command syntax: `dot` is the *current line* (what the user refers to as `.`), and `dol` is the *last line* (what the user refers to as `$`, “dollar”). Both are pointers, not indices: `dot - zero` gives the current line number (thanks to pointer arithmetic), and `dol - zero` gives the total number of lines. When the buffer is empty, `dol == zero`, which is why you will see often code like `dol!=zero` to test for non-empty buffer.

⟨globals ed.c 39⟩+≡ (91b) ◁38d 40a▷

```
// ref<int> in zero[], current line pointer
int*    dot;
// ref<int> in zero[], last line pointer
int*    dol;
```

This is where the `1,$` *address range* from the `mkenam` script in Section C.1.2 comes from. The substitute command in that script means “from line 1 to `$` (the last line), substitute . . . ”—and now we know that `$` is just the user spelling for `dol`.

C.2.5 Line input and output buffers: `line` and `linebuf`

There are two distinct line buffers, which can be confusing. `line`[40a] (70 bytes of `char`) is a small output buffer used by `putchr()`[45b] to batch `write()` system calls— UTF-8 encoded characters accumulate in `line` and are

flushed when a newline is encountered or the buffer is nearly full. `linebuf`[40c] (`LBSIZE` entries of `Rune`) is the main working buffer where a single line of edited text is held in Unicode form. Functions like `getfile()`[52b] fill `linebuf` from the input file, `putline()`[53b] writes it to `tfile`, and `getline()`[57a] reads it back. The distinction between `char` for UTF-8 I/O and `Rune` for text manipulation runs throughout the code.

⟨globals ed.c 40a⟩+≡ (91b) ◁39 40c▷
```
char    line[70];
// ref<char> in line (mark end of line usually)
char*   linp    = line;
```
Uses `line` 40a and `linp` 40a.

⟨constants ed.c 40b⟩+≡ (91b) ◁38b 68b▷
```
LBSIZE  = 4096,     /* max line size */
```

⟨globals ed.c 40c⟩+≡ (91b) ◁40a 40d▷
```
Rune    linebuf[LBSIZE];
```
Uses `LBSIZE-2` 40b.

C.2.6 Other globals: `count`, `col`, etc.

`count`[40d] is a general-purpose counter, most often used to track the number of characters read or written (displayed by `exfile()`[55a] in verbose mode). `col`[40e] tracks the current output column, used by the `l` (list) command to wrap long lines.

⟨globals ed.c 40d⟩+≡ (91b) ◁40c 40e▷
```
long count;
```

⟨globals ed.c 40e⟩+≡ (91b) ◁40d 40f▷
```
int col;
```

C.3 `main()`

I now switch from the bottom-up approach of the previous section to a top-down approach, starting with `main()`. The `main()` function initializes buffered console I/O with `Binit()` (see the LIBCORE book [Pad16b] with the `libbio` library and its `Biobuf` data structure), installs a signal handler with `notify()`, processes command-line flags, allocates `zero`[38d], then enters the main loop: `init()`[43a], `commands()`[48a], `quit()`[44b].

⟨globals ed.c 40f⟩+≡ (91b) ◁40e 41a▷
```
// console buffered input
Biobuf  bcons;
```

⟨*function* main*(ed.c)* 40g⟩≡ (91b)
```
void
main(int argc, char *argv[])
{
    char *p1, *p2;

    Binit(&bcons, STDIN, OREAD);
    notify(notifyf);

    ARGBEGIN {
    ⟨main() (ed.c) flags processing cases 42b⟩
    } ARGEND

    USED(argc);
    ⟨main() (ed.c) if - flag 41b⟩
```

```
    ⟨main() (ed.c) if oflag 42c⟩
    else if(*argv) {
        p1 = *argv;
        p2 = savedfile;
        while(*p2++ = *p1++)
            if(p2 >= &savedfile[sizeof(savedfile)])
                p2--;
        globp = L"r";
    }
    zero = malloc((nlall+5)*sizeof(int*)); // BUG should be sizeof(int)
    tfname = mktemp(template);

    init();
    ⟨main() (ed.c) before commands() 89c⟩
    commands();
    quit();
}
```

Uses `bcons` 40f, `commands()` 48a, `globp` 46c, `init()` 43a, `mktemp()` 42e, `nlall` 38d, `notifyf()` 90e, `quit()` 44b, `savedfile` 38c, `template` 42d, `tfname` 37a, and `zero` 38d.

The most tricky part in `main()` above is the `globp = L"r"` line. When a filename argument is given (e.g., `ed foo.txt`), the filename is copied into `savedfile`[38c] and `globp`[46c] is set to the Unicode string `"r"`. The `L""` prefix is standard C syntax for a *wide-string literal* (see the COMPILER book [Pad16a]): where the bare `"r"` would be a `char*` (one byte plus a null terminator), `L"r"` is a `Rune*`—one `Rune` holding the codepoint for `'r'` followed by a null `Rune` terminator. Later, when `commands()` calls `getchr()`[46d], it will read from `globp` before reading from the console, so the first command executed is automatically `r`—as if the user had typed `r foo.txt`. This elegant trick avoids duplicating file-reading logic in `main()`.

The `sizeof(int*)` in the `zero = malloc(...)` line above is actually a bug—it should be `sizeof(int)`, since `zero` holds `int` file offsets, not pointers. The bug is harmless on architectures where pointers and integers have the same size (32-bit systems); on 64-bit systems, pointers are 8 bytes while `int` is still 4, so the allocation simply over-reserves memory without causing any incorrect behavior. Even Ken Thompson wrote slightly buggy C code.

C.3.1 `ed -` and `vflag`

By default `ed` is verbose[2] (`vflag = true`): it prints character counts after `r` and `w` operations. The `-` flag (as in `ed - foo.txt`) suppresses this, which is what you want when `ed` is driven by a script (as in the `mkenam` example in Section C.1.2).

```
⟨globals ed.c 41a⟩+≡                                    (91b) ◁40f 42a▷
  // verbose (a.k.a. interactive) mode
  bool vflag   = true;
```

Uses `vflag` 41a.

```
⟨main() (ed.c) if - flag 41b⟩≡                                    (40g)
  if(*argv && (strcmp(*argv, "-") == 0)) {
      argv++;
      vflag = false;
  }
```

Uses `vflag` 41a.

[2] verbose is a strong word for `ed`; even in verbose mode, `ed` remains very terse (to not say cryptic).

C.3.2 `ed -o` and `oflag`

The `-o` flag turns `ed` into a *filter* (see UTILITIES book [Pad25]): it sets `savedfile`[38c] to `/fd/1` (Plan 9's path for stdout), so that `w` writes to standard output instead of a file. It also sets `globp = L"a"` to start this time in *append mode*, and disables verbose output (since character counts would corrupt the output stream).

⟨globals ed.c 42a⟩+≡ (91b) ◁41a 42d▷
```
// output to standard output (instead of editing a file).
// Useful when ed is used as a filter.
bool oflag;
```

⟨main() *(ed.c) flags processing cases* 42b⟩≡ (40g)
```
case 'o':
    oflag = true;
    vflag = false;
    break;
```
Uses `oflag` 42a and `vflag` 41a.

⟨main() *(ed.c) if* oflag 42c⟩≡ (40g)
```
if(oflag) {
    p1 = "/fd/1";
    p2 = savedfile;
    while(*p2++ = *p1++)
        ;
    globp = L"a";
}
```
Uses `globp` 46c, `oflag` 42a, and `savedfile` 38c.

C.3.3 `mktemp()`

`mktemp()`[42e] called early in `main()` generates a unique temporary filename by replacing the trailing `X`'s in the template with the process ID digits. If the resulting file already exists, it tries replacing the first digit with `a`, `b`, etc.

⟨globals ed.c 42d⟩+≡ (91b) ◁42a 43c▷
```
// in Linux pid can be very long, so better to have at least 6 X (was 5 before)
// the mkstemp man page recommends 6 X
char template[] = "/tmp/eXXXXXX";
```
Uses `template` 42d.

⟨*function* mktemp *(ed.c)* 42e⟩≡ (91b)
```
/// main -> <>
char*
mktemp(char *as)
{
    char *s;
    unsigned pid;
    int i;

    pid = getpid();
    s = as;
    while(*s++)
        ;
    s--;

    while(*--s == 'X') {
        *s = pid % 10 + '0';
        pid /= 10;
```

```
    }
    s++;

    i = 'a';
    while(access(as, 0) != -1) {
        if(i == 'z')
            return "/";
        *s = i++;
    }
    return as;
}
```

C.3.4 init()

init()[43a] creates (or re-creates) the temporary file and resets dot[39] and dol[39] to zero[38d] (empty buffer). It is called both from main() at startup and from the e command, which is why it closes the previous tfile[37b] first.

⟨*function* init *(ed.c)* 43a⟩≡ (91b)
```
/// main | commands('e') -> <>
void
init(void)
{
    ⟨init() (ed.c) locals 77c⟩

    close(tfile);
    ⟨init() (ed.c) initializing globals 43b⟩
    tfile = create(tfname, ORDWR, 0600);
    if(tfile < 0){
        error_1(T);
        exits(nil);
    }
    dot = dol = zero;
}
```
Uses T 43c, dol 39, dot 39, error_1() 90a, tfile 37b, tfname 37a, and zero 38d.

⟨init() *(ed.c) initializing globals* 43b⟩≡ (43a) 72b ▷
```
tline = 2;
```
Uses tline 38a.

Recall that tline is a Rune-unit offset into tfile (the actual byte file offset on disk is tline*sizeof(Rune); see the on-disk layout in Section C.2.3). It starts at 2, not 0, to satisfy two invariants every line offset must maintain: it must be non-zero so 0 stays the "no line" *sentinel*, and it must be even so the low bit of each zero entry stays free for global()[78] and the k command to use as a "matched" flag (see Section C.10.7).

In most of this appendix I will not comment much the error management code, which is often trivial (but important). See Section C.13 for a full discussion about error management in ed and for the code of error_1()[90a] called in init() above.

⟨globals ed.c 43c⟩+≡ (91b) ◁ 42d 44a ▷
```
char    T[] = "TMP";
```
Uses T 43c.

C.3.5 quit()

quit()[44b] implements the "are you sure?" safety net. It is called both from main() after commands()[48a] finished, or from commands() itself when the user types the command q or Q (see Section C.8.5). If there are unsaved changes (fchange == true) and the buffer is non-empty (using the cryptic dol!=zero), the first q prints ?

and resets `fchange`, so a second `q` will actually exit. The `Q` command bypasses this by clearing `fchange` before calling `quit()`.

⟨globals ed.c 44a⟩+≡ (91b) ◁43c 44c▷
```
bool fchange;
```

⟨*function* quit *(ed.c)* 44b⟩≡ (91b)
```
/// main | commands('q') | commands('Q') | ... -> <>
void
quit(void)
{
    if(vflag && fchange && dol!=zero) {
        fchange = false;
        error(Q);
    }
    // else
    remove(tfname);
    exits(nil);
}
```
Uses `Q` 44c, `dol` 39, `error()` 89d, `fchange` 44a, `tfname` 37a, `vflag` 41a, and `zero` 38d.

Note that `error()`[89d] internally uses `longjmp()` to emulate exceptions in C (see Section C.13), which is why the control leaves `quit()` after the call to `error()`.

⟨globals ed.c 44c⟩+≡ (91b) ◁44a 46a▷
```
char    Q[] = "";
```
Uses `Q` 44c.

C.4 Displaying and reading text

We have now seen the code of the `init()`[43a] and `quit()`[44b] functions. It remains to see the most important one, `commands()`[48a] called also directly from `main()`. However, it is useful to see before the routines to display and read text.

C.4.1 Displaying text

All text output in `ed` goes through a small buffering layer: `putchr()`[45b] is supposed to print a single Rune but actually accumulates the UTF-8 encoded bytes in the `line`[40a] buffer and flushes with a single `write()` call when it sees a newline (or the buffer is nearly full). `putst()`[44d] and `putshst()`[45a] below are wrappers for printing `char` (UTF-8) strings and `Rune` strings respectively and printing a final newline.

⟨*function* putstr *(ed.c)* 44d⟩≡ (91b)
```
/// commands | getfile | error_1 | ... -> <>
void
putst(char *sp)
{
    Rune r;

    col = 0;
    for(;;) {
        sp += chartorune(&r, sp);
        if(r == 0)
            break;
        putchr(r);
    }
    putchr(L'\n');
}
```
Uses `col` 40e and `putchr()` 45b.

⟨*function* `putshst` *(ed.c)* 45a⟩≡ (91b)

```
void
putshst(Rune *sp)
{
    col = 0;
    while(*sp)
        putchr(*sp++);
    putchr(L'\n');
}
```

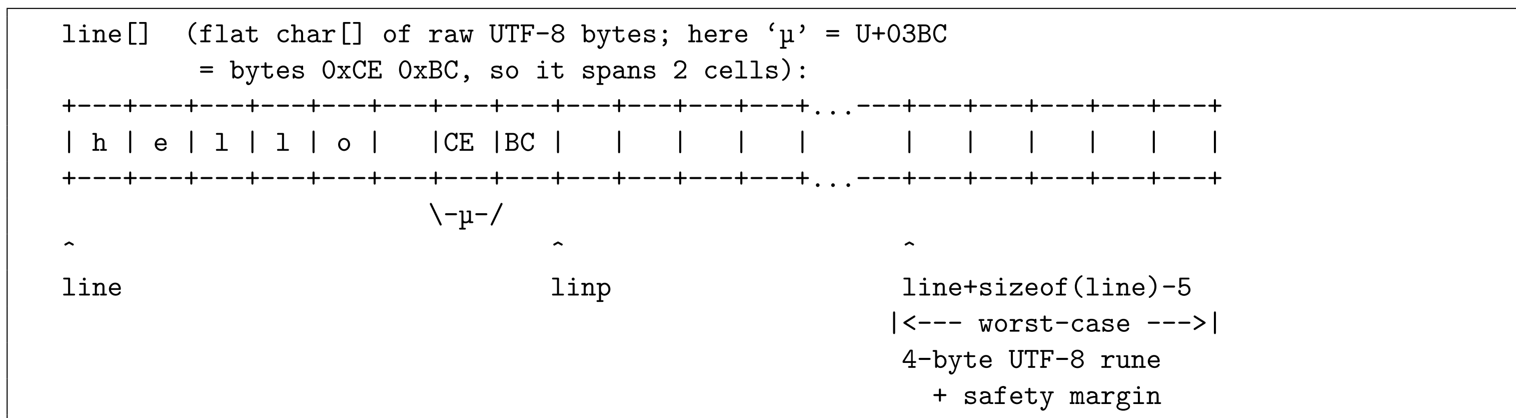
Uses `col` 40e and `putchr()` 45b.

⟨*function* `putchr` *(ed.c)* 45b⟩≡ (91b)

```
/// error | putst | putshst | ... -> <>
void
putchr(int ac)
{
    char *lp;
    int c;
    Rune rune;

    lp = linp;
    c = ac;
    ⟨putchr() if listf 81b⟩

    rune = c;
    lp += runetochar(lp, &rune);

    if(c == '\n' || lp >= &line[sizeof(line)-5]) {
        linp = line;
        // the actual output syscall!
        write(oflag? STDERR: STDOUT, line, lp-line);
        return;
    }
    // else
    linp = lp;
}
```

Uses `line` 40a, `linp` 40a, and `oflag` 42a.

The picture below shows `line` mid-fill during a call to `putshst(L"hello µs")`. The key points: `line` is a flat `char` array holding raw UTF-8 bytes (not `Rune` cells, unlike `tfile`[37b] or `linebuf`[40c]), so a single codepoint may occupy one to four cells; `linp`[40a] is the persistent fill pointer carried across calls (the local `lp` above is initialized from it and advanced by `runetochar()` before the bounds check fires).



`linp` tracks how full the buffer is. The flush check `lp >= &line[sizeof(line)-5]` reserves 5 bytes at the end of the buffer because: (1) runes converted to UTF-8 can be up to 4 bytes each, and (2) one extra byte for safety. Without this margin, a multi-byte rune arriving when the buffer is nearly full could overrun the array.

Note the `oflag` check: when `-o` is active, the edited content goes to `STDOUT`, so diagnostic output from `putchr()` is redirected to `STDERR` to avoid corrupting the data stream.

C.4.2 Reading text

Character input is layered through `getchr()`[46d], which checks three sources in order: (1) `peekc`[46a], a one-character pushback used when a character needs to be "unread" (a classic technique used in many lexers in Plan 9); (2) `globp`[46c], a programmed command string (e.g., `L"r"` or `L"a"` injected by `main()`); (3) actual console input via `Bgetrune()` (from `libbio`, see LIBCORE book [Pad16b]). This layering is what makes the `globp` trick work—`commands()`[48a] does not need to know whether input comes from the user or from a synthetic command.

⟨globals ed.c 46a⟩+≡ (91b) ◁44c 46b▷

```
//option<Rune> (0 = None)
int peekc;
```

⟨globals ed.c 46b⟩+≡ (91b) ◁46a 46c▷

```
//option<Rune> (0 = None)
int lastc;
```

⟨globals ed.c 46c⟩+≡ (91b) ◁46b 48b▷

```
// optional programmed command (e.g., L"r", L"a" or longer string with global())
//option<array<Rune>>
Rune*   globp;
```

⟨*function* getchr *(ed.c)* 46d⟩≡ (91b)

```
int
getchr(void)
{
    lastc = peekc;
    if(lastc) {
        peekc = 0;
        return lastc;
    }
    // else
    if(globp) {
        lastc=*globp++;
        if(lastc != 0)
            return lastc;
        // else
        globp = nil;
        return EOF;
    }
    // else
    lastc = Bgetrune(&bcons);
    return lastc;
}
```

Uses `EOF-9` 90c, `bcons` 40f, `globp` 46c, `lastc` 46b, and `peekc` 46a.

`gettty()`[46e3], reads a line of user input (via `gety()`[47a]) and checks for the special `.\n` terminator that ends text-entry mode in commands like `a` and `i` (see Section C.1.1). `gety()` does the actual character-by-character reading into `linebuf`[40c], stopping at a newline.

⟨*function* gettty *(ed.c)* 46e⟩≡ (91b)

```
/// main -> commands('a') -> add -> <>
int
gettty(void)
{
```

[3] "get from teletype", an old UNIX name for the console

```
    int rc;

    rc = gety();
    if(rc)
        return rc;
    // else
    if(linebuf[0] == '.' && linebuf[1] == 0)
        return EOF;
    return 0; // OK_0 ?
}
```

Uses `EOF-9` 90c, `gety()` 47a, and `linebuf` 40c.

Note that there is no easy way to create a line containing just a `'.'` followed by a newline since this special sequence is recognized by `gettty()` to mark the end of a user input. One needs instead to enter some fake text like `'.A'` and then use a substitute command to remove the `'A'`. Fortunately, it is rare to need text with a single dot by itself in a line (which is probably why this special sequence was chosen).

⟨*function* `gety` *(ed.c)* 47a⟩≡ (91b)

```
/// gettty -> <>
int
gety(void)
{
    int c;
    Rune *p = linebuf;
    ⟨gety() other locals 47b⟩

    for(;;) {
        c = getchr();
        if(c == '\n') {
            *p = 0;
            return 0;
        }
        // else
        if(c == EOF) {
            ⟨get() if gf 47c⟩
            return c;
        }
        // else
        if(c == 0)
            continue;
        // else
        *p++ = c;
        if(p >= &linebuf[LBSIZE-sizeof(Rune)])
            error(Q);
    }
}
```

Uses `EOF-9` 90c, `LBSIZE-2` 40b, `Q` 44c, `error()` 89d, `getchr()` 46d, and `linebuf` 40c.

⟨`gety()` *other locals* 47b⟩≡ (47a)

```
Rune *gf = globp;
```

⟨`get()` *if* `gf` 47c⟩≡ (47a)

```
if(gf)
    peekc = c;
```

Uses `peekc` 46a.

C.5 `commands()` interpreter loop

We are now finally ready to see the code of `commands()`[48a], the main command interpreter loop. In this infinite `for` loop, it reads first an optional *address range* into `addr1`[48b],`addr2`[48b] and then dispatches on the command character `c` via a `switch`. As we will see soon, most command `switch` cases end with a `continue` to loop back; falling through to `error(Q)` below after the switch deals with unknown commands.

⟨*function* `commands` *(ed.c)* 48a⟩≡ (91b)

```
void
commands(void)
{
    int c;
    ⟨commands() other locals 57b⟩

    for(;;) {
        ⟨commands() in for loop, if pflag 81e⟩
        ⟨commands() read addr1 and c via getchr 64d⟩
        switch(c) {
        ⟨commands() switch c cases (ed.c) 48c⟩
        }
        error(Q);
    }
}
```

Uses `Q` 44c and `error()` 89d.

I will defer the explanations about address ranges to Section C.9 but for now it is enough to understand that some commands can operate on ranges (as in `1,3p`) and that the two globals below will store the starting and ending part of the command address range. They both point to `zero`[38d] entries, similar to `dot`[39] and `dol`[39] we saw before.

⟨`globals ed.c` 48b⟩+≡ (91b) ◁46c 48d▷

```
// ref<int> in zero[]
int*    addr1;
// ref<int> in zero[]
int*    addr2;
```

⟨`commands()` *switch* `c` *cases (ed.c)* 48c⟩≡ (48a) 49a▷

```
case EOF:
    return;
```

Uses `EOF-9` 90c.

C.6 reading a file: `r`

Reading a file (as in `r foo.txt`) is the first substantial command, and it involves several cooperating functions: `filename()`[49c] parses the filename argument in `file`[48d], `setwide()`[50d] sets the address range to the whole buffer (which will have for effect to add the content of `foo.txt` to the end of `zero`, after `dol`[39]), `append()`[51b] inserts lines by calling a callback in a loop, `getfile()`[52b] (the callback) reads one line from the input file (e.g., `foo.txt`) into `linebuf`[40c], and `putline()`[53b] writes `linebuf` to `tfile`[37b] and returns the offset to store in `zero`[38d].

⟨`globals ed.c` 48d⟩+≡ (91b) ◁48b 48e▷

```
char    file[FNSIZE];
```

Uses `FNSIZE-1` 38b.

⟨`globals ed.c` 48e⟩+≡ (91b) ◁48d 55c▷

```
fdt io;
Biobuf  iobuf;
```

⟨commands() *switch* c *cases (ed.c)* 49a⟩+≡ (48a) ◁48c 55d▷
```
case 'r':
    // will set file[]
    filename(c);
caseread:
    io=open(file, OREAD);
    ⟨commands() in r case, sanity check io 49b⟩
    ⟨commands() in r case, if append only file 85a⟩
    Binit(&iobuf, io, OREAD);
    // will set addr2 to dol
    setwide();
    squeeze(0);
    c = (zero != dol);
    // getfile() will use iobuf
    append(getfile, addr2);
    exfile(OREAD); // will close io

    fchange = c;
    continue;
```
Uses addr2 48b, append() 51b, dol 39, exfile() 55a, fchange 44a, file 48d, filename() 49c, getfile() 52b, io 48e, iobuf 48e, setwide() 50d, squeeze() 51a, and zero 38d.

The sections below will detail each the code of one of the helper functions used above. The `fchange = c` at the end is subtle: `c` was saved as `(zero != dol)` before the append, so `fchange` is set to true only if the buffer already had content. Reading the first file into an empty buffer does not count as a "change"—this is what lets you `q` without being warned after just opening a file.

⟨commands() *in* r *case, sanity check* io 49b⟩≡ (49a)
```
if(io < 0) {
    lastc = '\n';
    error(file);
}
```
Uses error() 89d, file 48d, io 48e, and lastc 46b.

C.6.1 Reading a `filename()`

`filename()`[49c] reads a filename from user input and stores it in `file`[48d]. If no filename is given (just a newline), it falls back to `savedfile`[38c]—this is how `r` with no argument re-reads the current file. It also updates `savedfile` when used from `e` or `f`, establishing the "remembered filename" for future operations.

⟨*function* filename*(ed.c)* 49c⟩≡ (91b)
```
/// main -> commands('r' | 'e' | 'f') -> <>
void
filename(int comm)
{
    char *p1, *p2;
    Rune rune;
    int c;

    count = 0;
    c = getchr();
    ⟨filename() if c is newline or EOF, use savedfile 50b⟩
    // else
    ⟨filename() read a space and skip extra spaces 50a⟩
    p1 = file;
    do {
        if(p1 >= &file[sizeof(file)-6] || c == ' ' || c == EOF)
            error(Q);
        rune = c;
```

```
        p1 += runetochar(p1, &rune);
    } while((c=getchr()) != '\n');
    *p1 = '\0';

    ⟨filename() set savedfile depending on commands 50c⟩
}
```

Uses EOF-9 90c, Q 44c, count 40d, error() 89d, file 48d, and getchr() 46d.

⟨filename() *read a space and skip extra spaces* 50a⟩≡ (49c)

```
if(c != ' ')
    error(Q);
while((c=getchr()) == ' ')
    ;
if(c == '\n')
    error(Q);
```

Uses Q 44c, error() 89d, and getchr() 46d.

⟨filename() *if* c *is newline or EOF, use* savedfile 50b⟩≡ (49c)

```
if(c == '\n' || c == EOF) {
    p1 = savedfile;
    if(*p1 == '\0' && comm != 'f')
        error(Q);
    p2 = file;
    while(*p2++ = *p1++)
        ;
    return;
}
```

Uses EOF-9 90c, Q 44c, error() 89d, file 48d, and savedfile 38c.

⟨filename() *set* savedfile *depending on commands* 50c⟩≡ (49c)

```
if(savedfile[0] == '\0' || comm == 'e' || comm == 'f') {
    p1 = savedfile;
    p2 = file;
    while(*p1++ = *p2++)
        ;
}
```

Uses file 48d and savedfile 38c.

C.6.2 setwide() and squeeze()

setwide()[50d] sets addr1,addr2 to span the whole buffer (line 1 to dol[39]) when the user did not supply explicit addresses (thx to the given[64a] global).

⟨*function* setwide*(ed.c)* 50d⟩≡ (91b)

```
/// main -> commands('r') -> <>
void
setwide(void)
{
    if(!given) {
        addr1 = zero + (dol>zero);
        addr2 = dol;
    }
}
```

Uses addr1 48b, addr2 48b, dol 39, given 64a, and zero 38d.

squeeze()[51a] validates that the address range is sane—called with 0 when an empty buffer is acceptable (e.g., for the r command) or 1 when at least one line is required (e.g., for the p command).

⟨*function* squeeze *(ed.c)* 51a⟩≡ (91b)
```
/// main -> commands('r') -> <>
void
squeeze(int i)
{
    if(addr1 < zero+i || addr2 > dol || addr1 > addr2)
        error(Q);
}
```
Uses Q 44c, addr1 48b, addr2 48b, dol 39, error() 89d, and zero 38d.

C.6.3 append() and getfile()

append()[51b] is the core insertion function. It takes a callback f (either getfile()[52b] for reading files or gettty()[46e] for user input) and an address a (where to insert after). For each line that f returns via linebuf[40c], it calls putline()[53b] to write it to tfile[37b], then shifts all entries in zero[38d] above the insertion point up by one to make room for the new offset. This shift is the while(a1 > rdot) loop.

⟨*function* append *(ed.c)* 51b⟩≡ (91b)
```
/// main -> commands('r') -> <>
int
append(int (*f)(void), int *a)
{
    //ref<int> in zero[]
    int *a1, *a2, *rdot;
    int nline = 0;
    // file offset in tfile for temporary line just added by putline()
    int tl;

    dot = a;
    // f() (e.g., getfile()) will modify linebuf[]
    while((*f)() == 0) {
        ⟨append() grow zero if zero too small 52a⟩
        // putline() will use linebuf[]
        tl = putline();
        nline++;

        // set zero[] indices
        a1 = ++dol;
        a2 = a1+1;
        rdot = ++dot;
        // shift existing line references up by one
        while(a1 > rdot)
            *--a2 = *--a1;
        // insert the new line reference in zero
        *rdot = tl;
    }
    return nline;
}
```
Uses dol 39, dot 39, and putline() 53b.

When zero is full, realloc() may move it to a new address. Every pointer into the old array—addr1, addr2, dol, dot—must then be adjusted by the same delta (a1 - zero, i.e., new base minus old base). This is a classic C realloc pattern: compute the displacement once, then apply it to all pointers that referred to the

old allocation.

⟨append() *grow* zero *if* zero *too small* 52a⟩≡ (51b)
```
  if((dol-zero) >= nlall) {

      nlall += 512;
      a1 = realloc(zero, (nlall+5)*sizeof(int*)); // BUG: sizeof(int)
      if(a1 == nil) {
          error("MEM?");
          rescue();
      }
      tl = a1 - zero; /* relocate pointers */

      zero += tl;
      addr1 += tl;
      addr2 += tl;
      dol += tl;
      dot += tl;
  }
```
Uses addr1 48b, addr2 48b, dol 39, dot 39, error() 89d, nlall 38d, rescue() 91a, and zero 38d.

getfile() reads one line from iobuf[48e] (the Biobuf initialized from the file to read from the r command) into linebuf, counting characters in count[40d]. It replaces the terminating \n with a null rune (lines in tfile are null-terminated, not newline-terminated as explained in Section C.2.3). It returns 0 on success or EOF when the file is exhausted.

⟨*function* getfile*(ed.c)* 52b⟩≡ (91b)
```
  /// main -> commands(c = 'r') -> append(<>, ...) -> <>
  int
  getfile(void)
  {
      int c;
      Rune *lp;

      lp = linebuf;
      do {
          c = Bgetrune(&iobuf);
          ⟨getfile() if c negative, possibly return EOF 52c⟩
          // else
          ⟨getfile() if overflow linebuf 53a⟩
          // else
          *lp++ = c;
          count++;
      } while(c != '\n');
      lp[-1] = 0;
      return 0; // OK_0
  }
```
Uses count 40d, iobuf 48e, and linebuf 40c.

⟨getfile() *if* c *negative, possibly return* EOF 52c⟩≡ (52b)
```
  if(c < 0) {
      if(lp > linebuf) {
          putst("'\\n' appended");
          c = '\n';
      } else
          return EOF;
  }
```
Uses EOF-9 90c, linebuf 40c, and putst() 44d.

⟨getfile() *if overflow* linebuf 53a⟩≡ (52b)

```
  if(lp >= &linebuf[LBSIZE]) {
      lastc = '\n';
      error(Q);
  }
```

Uses LBSIZE-2 40b, Q 44c, error() 89d, lastc 46b, and linebuf 40c.

C.6.4 putline() (simplified)

Here is a simplified version of putline()[53b] that shows the essential logic: seek to the current write position in tfile[37b], write linebuf[40c] in tfile as a run of fixed-width Rune cells terminated by a null Rune, advance tline[38a] by the number of Runes written (rounded up to an even count), and return the old offset (which append()[51b] stores in zero[38d]).

⟨*function* putline 53b⟩≡ (91b)

```
  int
  putline(void)
  {
      Rune *lp;
      int n, tl;

      ⟨putline() if opti 85e⟩
      // else

      fchange = true;
      tl = tline;                              // current write offset (Rune units)

      // walk linebuf[]; stop at the null Rune or an embedded '\n'
      for(lp = linebuf; *lp; lp++) {
          ⟨putline() if newline char, adjust and break 54⟩
          // else, nothing, continue loop until *lp == 0 == null Rune
      }
      n = lp - linebuf + 1;                    // # Runes including terminator

      if(tline + n >= NBLK * BLKSIZE / (int)sizeof(Rune))
          error(T);

      // tfile stores fixed-width Runes (not UTF-8);
      // convert Rune offset/count to byte offsets for seek/write
      seek(tfile, tline * sizeof(Rune), SEEK__START);
      // syscall!
      write(tfile, linebuf, n * sizeof(Rune));

      // advance past the line just written, rounding up to an even Rune
      // count so tline stays even. zero[] entries are Rune offsets, and
      // global() and 'k' steal the low bit of each entry as a "matched"
      // flag (see global() and anymarks). Padding costs at most one Rune.
      tline += (n + 1) & ~1;
      return tl;
  }
```

Uses BLKSIZE-7 85d, NBLK-8 87, T 43c, error() 89d, fchange 44a, linebuf 40c, tfile 37b, and tline 38a.

The crucial detail: tfile is a stream of Rune-sized cells, not bytes, and tline counts cells, not bytes—which is why every seek and write needs a *sizeof(Rune) conversion.

The other subtlety is the even-padding of tline. Each zero entry holds a Rune offset into tfile (the value returned here), and global()[78] together with the k command steal the low bit of that entry as a “matched”

flag, Keeping every offset even costs at most one extra Rune cell per line and frees the bit for the mark.

⟨putline() *if newline char, adjust and break* 54⟩≡ (53b)
```
// (the latter happens after dosub() injects \n in a multi-line subst)
if(*lp == '\n') {
    *lp = 0;                     // turn '\n' into null terminator
    linebp = lp + 1;             // stash remainder for getsub()
    break;
}
```
Uses linebp 85c.

C.6.5 Concrete editing example

Here is a short example of how zero and tfile evolve. The diagrams below use byte offsets and \n-terminated text for readability, as in the earlier picture of zero in Section C.2.3; the actual on-disk layout is fixed-width Rune cells, but that detail does not affect the editing operations shown here. We do keep one detail from the real layout: offsets start at 2, because tline starts at 2 and so the first two cells of tfile are reserved (kept clear for global()[78]'s mark trick). The first line therefore lands at offset 2, not 0. Suppose the user types:

```
r foo.txt    (file contains: "alpha\nbeta\ngamma\n")
2d           (delete line 2)
1a           (append after line 1)
delta        (the new text)
.
```

After r reads the three lines in foo.txt:

```
tfile (offsets):   0  2      8     13     19
                   +--+------+-----+------+
                   |..|alpha\|beta\|gamma\|
                   +--+------+-----+------+
                    ^^ reserved (bytes 0-1, kept clear)
zero:  [_, 2, 8, 13]    (3 lines, dot=3)
```

After 2d (delete line 2: shift entries down in zero, do nothing on disk):

```
tfile:             unchanged (offset 8 'beta' is orphaned)
zero:  [_, 2, 13]       (2 lines, dot=2)
```

After 1a then delta then . (append "delta" after line 1, putline() writes delta\n to the end of tfile at offset 19, append()[51b] inserts the new offset into zero):

```
tfile:             0  2      8     13     19     25
                   +--+------+-----+------+------+
                   |..|alpha\|beta\|gamma\|delta\|
                   +--+------+-----+------+------+
                                          ^new
zero:  [_, 2, 19, 13]    (3 lines, dot=2)
```

Notice that tfile grew by one line (delta), but the "deleted" beta is still on disk—it just is no longer referenced by zero. This is why ed can do bounded edits in constant time but the temp file can become arbitrarily large during a long session.

C.6.6 exfile()

`exfile()`[55a] ("exit file") finalizes an I/O operation: it flushes the buffer if writing, closes `io`[48e], and in verbose mode prints the character count via `putd()`[55b].

⟨*function* `exfile` *(ed.c)* 55a⟩≡ (91b)
```
/// main -> commands('r') -> <>
void
exfile(int om)
{

    if(om == OWRITE)
        if(Bflush(&iobuf) < 0)
            error(Q);
    close(io);
    io = -1;
    if(vflag) {
        putd();
        putchr(L'\n');
    }
}
```
Uses `Q` 44c, `error()` 89d, `io` 48e, `iobuf` 48e, `putchr()` 45b, `putd()` 55b, and `vflag` 41a.

⟨*function* `putd` *(ex.c)* 55b⟩≡ (91b)
```
void
putd(void)
{
    int r;

    r = count%10;
    count /= 10;
    if(count)
        // recurse
        putd();
    putchr(r + L'0');
}
```
Uses `count` 40d, `putchr()` 45b, and `putd()` 55b.

C.7 writing a file: w

Writing (as in `w foo.txt`) is the dual of reading. `putfile()`[56] iterates over the address range, fetches each line from `tfile` via `getline()`[57a], and writes it to the output file via `Bputrune()` (from `libbio`). The `W` (uppercase) variant opens the file in append mode.

⟨`globals ed.c` 55c⟩+≡ (91b) ◁48e 64a▷
```
// write append
bool wrapp;
```

⟨`commands()` *switch* `c` *cases (ed.c)* 55d⟩+≡ (48a) ◁49a 58a▷
```
case 'W':
    wrapp = true;
    // Fallthrough
case 'w':
    setwide();
    squeeze(dol>zero);

    ⟨commands() when 'w' check for 'q' part1 57c⟩
    filename(c);
```

```
    if(!wrapp ||
      ((io = open(file, OWRITE)) == -1) ||
      ((seek(io, 0L, SEEK__END)) == -1))

        io = create(file, OWRITE, 0666);
        if(io < 0)
            error(file);
    // else and fallthrough

    Binit(&iobuf, io, OWRITE);
    wrapp = false;
    if(dol > zero)
        putfile();

    exfile(OWRITE);

    if(addr1<=zero+1 && addr2==dol)
        fchange = false;
    ⟨commands() when 'w' check for 'q' part2 57d⟩
    continue;
```

Uses addr1 48b, addr2 48b, dol 39, error() 89d, exfile() 55a, fchange 44a, file 48d, filename() 49c, io 48e, iobuf 48e, putfile() 56, setwide() 50d, squeeze() 51a, wrapp 55c, and zero 38d.

C.7.1 putfile()

putfile()[56] is the dual of getfile()[52b]: it writes in iobuf[48e].

⟨*function* putfile*(ed.c)* 56⟩≡ (91b)

```
void
putfile(void)
{
    int *a1;
    Rune *lp;
    long c;

    a1 = addr1;
    do {
        // modifies linebuf[]
        lp = getline(*a1++);
        for(;;) {
            count++;
            c = *lp++;
            if(c == 0) {
                // a null Rune in tfile converts to \n in the written file
                if(Bputrune(&iobuf, '\n') < 0)
                    error(Q);
                break;
            }
            // else
            if(Bputrune(&iobuf, c) < 0)
                error(Q);
        }
    } while(a1 <= addr2);
    if(Bflush(&iobuf) < 0)
        error(Q);
}
```

Uses Q 44c, addr1 48b, addr2 48b, count 40d, error() 89d, getline() 57a, and iobuf 48e.

C.7.2 getline() (simplified)

getline()[57a] is the dual of putline()[53b]: given a line offset tl (in Rune units), it seeks to that position in tfile and reads Rune cells into linebuf[40c] until it hits the null-Rune terminator. As in putline(), the *sizeof(Rune) factor in the seek and the read size is what converts the cell-unit offset into a byte file offset on disk.

⟨*function* getline 57a⟩≡ (91b)
```
Rune*
getline(int tl)
{
    int n;

    ⟨getline() if opti 88b⟩
    // else

    // tfile stores fixed-width Runes; convert Rune offset to bytes
    seek(tfile, tl * sizeof(Rune), SEEK__START);

    // read one Rune at a time until we hit the null-Rune terminator
    n = 0;
    // syscall
    while(read(tfile, &linebuf[n], sizeof(Rune)) == sizeof(Rune)
          && linebuf[n] != 0)
        n++;

    linebuf[n] = 0;
    return linebuf;
}
```
Uses linebuf 40c and tfile 37b.

C.7.3 Write and quit: wq

The w command peeks at the next character to detect wq (write and quit) or wQ (write and force-quit). If the next character is neither q nor Q, it is pushed back (“unread”) with peekc[46a] so filename()[49c] called after can consume it.

⟨commands() *other locals* 57b⟩≡ (48a) 64b ▷
```
int temp;
```

⟨commands() *when* 'w' *check for* 'q' *part1* 57c⟩≡ (55d)
```
temp = getchr();
if(temp != 'q' && temp != 'Q') {
    peekc = temp;
    temp = 0;
}
```
Uses getchr() 46d and peekc 46a.

⟨commands() *when* 'w' *check for* 'q' *part2* 57d⟩≡ (55d)
```
if(temp == 'Q')
    fchange = false;
if(temp)
    quit();
```
Uses fchange 44a and quit() 44b.

C.8 Main commands

With the reading and writing machinery in place, we can now look at the basic editing commands. Most of them are short—often just a few lines calling the helper functions we have already seen.

C.8.1 printing lines: p

The p command prints lines in the address range. printcom()[58c] iterates from addr1 to addr2, calling getline()[57a] to fetch each line from tfile and putshst()[45a] to display it.

⟨commands() *switch* c *cases (ed.c)* 58a⟩+≡ (48a) ◁55d 58e▷

```
⟨commands before 'p' case 81a⟩
case 'p':
case 'P':
    newline();
    printcom();
    continue;
```

Uses newline() 58b and printcom() 58c.

⟨*function* newline *(ed.c)* 58b⟩≡ (91b)

```
void
newline(void)
{
    int c;

    c = getchr();
    if(c == '\n' || c == EOF)
        return;
    ⟨newline() if special chars pln 82a⟩
    // else
    error(Q);
}
```

Uses EOF-9 90c, Q 44c, error() 89d, and getchr() 46d.

⟨*function* printcom *(ed.c)* 58c⟩≡ (91b)

```
void
printcom(void)
{
    int *a1;

    nonzero();
    a1 = addr1;
    do {
        ⟨printcom() if listn 82e⟩
        putshst(getline(*a1++));
    } while(a1 <= addr2);
    dot = addr2;
    ⟨printcom() reset flags 82b⟩
}
```

Uses addr1 48b, addr2 48b, dot 39, getline() 57a, nonzero() 58d, and putshst() 45a.

⟨*function* nonzero *(ed.c)* 58d⟩≡ (91b)

```
void
nonzero(void)
{
    squeeze(1);
}
```

Uses squeeze() 51a.

Pressing just Enter (newline with no command) prints the next line—the '\n' case below advances dot[39] by one and prints.

⟨commands() *switch* c *cases (ed.c)* 58e⟩+≡ (48a) ◁58a 59a▷

```
case '\n':
    if(a1==nil) {
```

```
        a1 = dot+1;
        addr2 = a1;
        addr1 = a1;
    }
    if(lastsep==';')
        addr1 = a1;

    printcom();
    continue;
```

Uses addr1 48b, addr2 48b, dot 39, and printcom() 58c.

C.8.2 printing the remembered file: f

⟨commands() *switch* c *cases (ed.c)* 59a⟩+≡ (48a) ◁58e 59c▷

```
case 'f':
    setnoaddr();
    filename(c);
    putst(savedfile);
    continue;
```

Uses filename() 49c, putst() 44d, savedfile 38c, and setnoaddr() 59b.

This command does not accept any explicit address range before, hence the call to setnoaddr()[59b].

⟨*function* setnoaddr *(ed.c)* 59b⟩≡ (91b)

```
void
setnoaddr(void)
{
    if(given)
        error(Q);
}
```

Uses Q 44c, error() 89d, and given 64a.

C.8.3 printing line number: =

⟨commands() *switch* c *cases (ed.c)* 59c⟩+≡ (48a) ◁59a 59d▷

```
case '=':
    setwide();
    squeeze(0);
    newline();
    count = addr2 - zero;
    putd();
    putchr(L'\n');
    continue;
```

Uses addr2 48b, count 40d, newline() 58b, putchr() 45b, putd() 55b, setwide() 50d, squeeze() 51a, and zero 38d.

C.8.4 append and insert: a, i

Both a (append) and i (insert) use add()[60b], which calls append(gettty, addr2). The difference is that i decrements addr1/addr2 by one, so the new text goes *before* the current line instead of after it.

⟨commands() *switch* c *cases (ed.c)* 59d⟩+≡ (48a) ◁59c 60a▷

```
case 'a':
    add(0);
    continue;
```

Uses add() 60b.

⟨commands() *switch* c *cases (ed.c)* 60a⟩+≡ (48a) ◁59d 60c▷

```
case 'i':
    add(-1);
    continue;
```

Uses add() 60b.

⟨*function* add *(ed.c)* 60b⟩≡ (91b)

```
void
add(int i)
{
    if(i && (given || dol > zero)) {
        addr1--;
        addr2--;
    }
    squeeze(0);
    newline();
    append(gettty, addr2);
}
```

Uses addr1 48b, addr2 48b, append() 51b, dol 39, gettty() 46e, given 64a, newline() 58b, squeeze() 51a, and zero 38d.

We saw before a call to `append(getfile, ...)` for the `r` command in Section C.6. This time `add()` passes instead `gettty()`[46e] (that we also saw before) for the callback to `append()`[51b].

C.8.5 quitting: q

⟨commands() *switch* c *cases (ed.c)* 60c⟩+≡ (48a) ◁60a 60d▷

```
case 'Q':
    fchange = false;
    // fallthrough:
case 'q':
    setnoaddr();
    newline();
    quit();
```

Uses fchange 44a, newline() 58b, quit() 44b, and setnoaddr() 59b.

C.8.6 deleting lines: d

`rdelete()`[60e] ("range delete") removes lines by shifting the `zero`[38d] entries above `addr2`[48b] down to overwrite the deleted range—the mirror image of the upward shift in `append()`[51b]. The text remains in `tfile`[37b] (it is never reclaimed), only the offsets in `zero` change.

⟨commands() *switch* c *cases (ed.c)* 60d⟩+≡ (48a) ◁60c 61a▷

```
case 'd':
    nonzero();
    newline();
    rdelete(addr1, addr2);
    continue;
```

Uses addr1 48b, addr2 48b, newline() 58b, nonzero() 58d, and rdelete() 60e.

⟨*function* rdelete *(ed.c)* 60e⟩≡ (91b)

```
void
rdelete(int *ad1, int *ad2)
{
    int *a1, *a2, *a3;

    a1 = ad1;
    a2 = ad2+1;
```

```
    a3 = dol;
    dol -= a2 - a1;
    do {
        *a1++ = *a2++;
    } while(a2 <= a3);
    a1 = ad1;
    if(a1 > dol)
        a1 = dol;
    dot = a1;
    fchange = true;
}
```

Uses `dol` 39, `dot` 39, and `fchange` 44a.

C.8.7 changing lines: `c`

The `c` (change) command is simply delete followed by append: it calls `rdelete()`[60e] then `append(gettty, addr1-1)`.

⟨`commands()` *switch* `c` *cases (ed.c)* 61a⟩+≡ (48a) ◁60d 61b▷

```
case 'c':
    nonzero();
    newline();
    rdelete(addr1, addr2);
    append(gettty, addr1-1);
    continue;
```

Uses `addr1` 48b, `addr2` 48b, `append()` 51b, `gettty()` 46e, `newline()` 58b, `nonzero()` 58d, and `rdelete()` 60e.

C.8.8 moving and copying lines: `m` and `t`

The `m` (move) and `t` (copy, since `c` was taken) commands share the `move()`[61d] function. For `t`, the lines are first duplicated to the end of the buffer via `append(getcopy, ...)`, then moved.

⟨`commands()` *switch* `c` *cases (ed.c)* 61b⟩+≡ (48a) ◁61a 61c▷

```
case 'm':
    move(0);
    continue;
```

Uses `move()` 61d.

⟨`commands()` *switch* `c` *cases (ed.c)* 61c⟩+≡ (48a) ◁61b 71c▷

```
case 't':
    move(1);
    continue;
```

Uses `move()` 61d.

⟨*function* `move` *(ed.c)* 61d⟩≡ (91b)

```
void
move(int cflag)
{
    int *adt, *ad1, *ad2;

    nonzero();
    if((adt = address())==0)    /* address() guarantees addr is in range */
        error(Q);
    newline();

    if(cflag) {
        int *ozero, delta;
```

```
        ad1 = dol;
        ozero = zero;
        append(getcopy, ad1++);
        ad2 = dol;
        delta = zero - ozero;
        ad1 += delta;
        adt += delta;
    } else {
        ad2 = addr2;
        for(ad1 = addr1; ad1 <= ad2;)
            *ad1++ &= ~01;
        ad1 = addr1;
    }
    ad2++;
    if(adt<ad1) {
        dot = adt + (ad2-ad1);
        if((++adt)==ad1)
            return;
        reverse(adt, ad1);
        reverse(ad1, ad2);
        reverse(adt, ad2);
    } else
    if(adt >= ad2) {
        dot = adt++;
        reverse(ad1, ad2);
        reverse(ad2, adt);
        reverse(ad1, adt);
    } else
        error(Q);
    fchange = true;
}
```

Uses `Q` 44c, `addr1` 48b, `addr2` 48b, `address()` 65a, `append()` 51b, `dol` 39, `dot` 39, `error()` 89d, `fchange` 44a, `getcopy()` 62a, `newline()` 58b, `nonzero()` 58d, `reverse()` 62b, and `zero` 38d.

⟨*function* `getcopy` *(ed.c)* 62a⟩≡ (91b)

```
int
getcopy(void)
{
    if(addr1 > addr2)
        return EOF;
    getline(*addr1++);
    return 0;
}
```

Uses `EOF-9` 90c, `addr1` 48b, `addr2` 48b, and `getline()` 57a.

⟨*function* `reverse` *(ed.c)* 62b⟩≡ (91b)

```
void
reverse(int *a1, int *a2)
{
    int t;

    for(;;) {
        t = *--a2;
        if(a2 <= a1)
            return;
        *a2 = *a1;
        *a1++ = t;
    }
}
```

C.9 Command addresses

Until now I have glossed over how `addr1`[48b] and `addr2`[48b] are set. In `ed`, most commands can be preceded by one or two addresses: a line number (`3p`), a range (`1,5d`), the current line (`.p`), the last line (`$a`), or even a search (`/foo/p`). The address-parsing code runs before the command character is read, so every command benefits from the same rich addressing syntax.

Here is a cheat sheet of the address primitives and how they eventually show up in the code:

```
syntax      meaning                      handled by
------      --------------------------   -------------------
N           absolute line N              getnum() in address()
.           dot (current line)           case '.' in address()
$           dol (last line)              case '$' in address()
'x          line previously marked x     case '\'' (see 'k')
/re/        next line matching re        case '/'  in address()
?re?        previous line matching re    case '?'  in address()
A+N  A-N    A offset by N                + and - operators
A,B         range (addr1=A, addr2=B)     top-level in commands()
A;B         range with side effect dot=A ';' case in commands()
```

All these forms feed a single recursive-descent parser. The two address separators `,` and `;` are parsed in `commands()`[48a] itself; everything to the left or right of a separator is parsed by `address()`[65a].

A worked example makes this concrete. Suppose the user types `.+2,/foo/p` when `dot` points to line 5 and the next line matching `foo` is line 11. The parse proceeds as follows:

```
outer loop in commands():
  a1 = address()       -> parses ".+2"
     step 1: c='.'   -> a=dot (line 5), opcnt=1
     step 2: c='+'   -> sign=+, nextopand=2, loop
     step 3: c='2'   -> a += 2 (line 7), opcnt=2
     step 4: c=','   -> push back, return a (line 7)
  c = ','              -> separator, loop again
  addr1 = 7
  a1 = address()       -> parses "/foo/"
     step 1: c='/'   -> compile(/foo/); scan from dot
                          until match -> a=line 11
     step 2: c='p'   -> push back, return a (line 11)
  c = 'p'              -> not a separator, exit loop
  addr2 = 11; given = true
  switch(c='p') -> print lines 7..11
```

Two things are worth noticing. First, the address parser is strictly left-to-right with no precedence—every operator is applied in sequence, so `1+2-3` means `((1+2)-3)`, never `1+(2-3)`. Second, the search forms `/re/` and `?re?` are resolved *during* parsing: `address()` actually scans the buffer and returns a concrete line pointer, which is why compiling the regexp happens inline in `address()` rather than later at execution time.

Returning to the `mkenam` script in Section C.1.2, you can now read the address parts of each line. The substitution line is two addresses (`1` and `$`) followed by the `s` command; the `v` line is a global-not-match command (no address part, matches operate on the whole buffer); `1i` is a single address (line 1) followed by `i` (insert before); `$a` is a single address (last line with `$`) followed by `a` (append after).

C.9.1 Reading addr1 and addr2

```
⟨globals ed.c 64a⟩+≡                                         (91b) ◁55c 68a▷
  bool given;
```

```
⟨commands() other locals 64b⟩+≡                              (48a) ◁57b 64c▷
  int *a1;
```

```
⟨commands() other locals 64c⟩+≡                              (48a) ◁64b 84b▷
  char lastsep; // '\n' or ',' or ';'
```

The address-parsing loop reads addresses separated by `,` or `;`. If no addresses are given, `addr1 = addr2 = dot` (the current line). If only one address is given, both `addr1` and `addr2` point to it. The `given` flag records whether the user typed an explicit address, so commands like `setwide()`[50d] can choose appropriate defaults.

```
⟨commands() read addr1 and c via getchr 64d⟩≡                      (48a)
  c = '\n';
  addr1 = nil;

  for(;;) {
      lastsep = c;
      a1 = address();
      c = getchr();

     if(c != ',' && c != ';')
         break;

      // else
      if(lastsep == ',')
          error(Q);
      if(a1 == nil) {
          a1 = zero+1; // line 1
          if(a1 > dol)
              a1--;
      }
      addr1 = a1;
      ⟨commands() in address parsing, if separator is ';' 64e⟩
  }
  ⟨commands() after address parsing, use defaults if missing addresses 64f⟩
```

Uses `Q` 44c, `addr1` 48b, `address()` 65a, `dol` 39, `error()` 89d, `getchr()` 46d, and `zero` 38d.

```
⟨commands() in address parsing, if separator is ';' 64e⟩≡          (64d)
  if(c == ';')
      dot = a1;
```

Uses `dot` 39.

```
⟨commands() after address parsing, use defaults if missing addresses 64f⟩≡   (64d)
  if(lastsep != '\n' && a1 == nil)
      a1 = dol;

  if((addr2=a1) == nil) {
      given = false;
      addr2 = dot;
  } else
      given = true;

  if(addr1 == nil)
      addr1 = addr2;
```

Uses `addr1` 48b, `addr2` 48b, `dol` 39, `dot` 39, and `given` 64a.

C.9.2 address()

address()[65a] parses a single address expression. It starts from dot and applies offsets: a bare number is relative to line 0 (i.e., absolute), . and $ set the base, + and - adjust the sign for subsequent operands, and /re/ or ?re? search forward or backward.

⟨*function* address *(ed.c)* 65a⟩≡ (91b)

```
int*
address(void)
{
    int sign, *a, opcnt, nextopand, *b, c;

    nextopand = -1;
    sign = 1;
    opcnt = 0;
    a = dot;
    do {
        do {
            c = getchr();
        } while(c == ' ' || c == '\t');

        if(c >= '0' && c <= '9') {
            peekc = c;
            if(!opcnt)
                a = zero;
            a += sign*getnum();
        } else

        switch(c) {
        // will return in default: case
        ⟨address() (ed.c) switch c cases 65b⟩
        }

        sign = 1;
        opcnt++;
    } while(zero <= a && a <= dol);

    error(Q);
    return nil;
}
```

Uses Q 44c, dol 39, dot 39, error() 89d, getchr() 46d, getnum() 66a, peekc 46a, and zero 38d.

The nested do-do structure above is subtle. The inner loop just skips whitespace. The outer loop chains operators: each iteration consumes one operand (., $, a number, or a search), and then loops back to see if another + or - follows. The variable nextopand tracks whether a trailing operator like + or - was seen without a following operand, in which case the default: branch adds an implicit 1 (so 3+ means 3+1, i.e., line 4).

The default: case below handles the return: if no operator is pending, push the character back and return the computed address.

⟨address() *(ed.c) switch* c *cases* 65b⟩≡ (65a) 66b ▷

```
default:
    if(nextopand == opcnt) {
        a += sign;
        if(a < zero || dol < a)
            continue;       /* error(Q); */
    }

    if(c != '+' && c != '-' && c != '^') {
        peekc = c;
        if(opcnt == 0)
```

```
            a = nil;

        // finally returning!
        return a;
    }
    sign = 1;
    if(c != '+')
        sign = -sign;
    nextopand = ++opcnt;
    continue;
```

Uses `dol` 39, `peekc` 46a, and `zero` 38d.

⟨*function* `getnum` *(ed.c)* 66a⟩≡ (91b)

```
int
getnum(void)
{
    int r = 0;
    int c;

    for(;;) {
        c = getchr();
        if(c < '0' || c > '9')
            break;
        r = r*10 + (c-'0');
    }
    peekc = c;
    return r;
}
```

Uses `getchr()` 46d and `peekc` 46a.

C.9.3 Basic addresses: . and $

⟨`address()` *(ed.c) switch* `c` *cases* 66b⟩+≡ (65a) ◁65b 67▷

```
case '$':
    a = dol;
    // Fallthrough
case '.':
    if(opcnt)
        error(Q);
    break;
```

Uses `Q` 44c, `dol` 39, and `error()` 89d.

C.10 Search and replace

Search and replace is where `ed`'s real power lies. The `/re/` and `?re?` address forms search forward and backward, the `s` command performs substitutions, and the `g` command applies commands to all matching lines. The actual regular expression engine is in `libregexp` (see LIBCORE book [Pad16b]); `ed` just reads and compiles patterns via `compile()`[68c] and tests matches via `match()`[69d].

Before diving in, it helps to see the buffers and pointers that the substitute machinery juggles together. A single `s/re/rhs/` command touches five distinct pieces of state:

```
             +-----------------------------+
pattern -->|  libregexp Reprog  (NFA)     |   compiled RE
             +-----------------------------+

             +-----------------------------+
rhsbuf   :  |  replacement with ESCFLG     |   parsed by compsub()
            |  markers for \1..\9 and &    |
             +-----------------------------+

             +-----------------------------+
linebuf  :  |  current line from tfile     |
             +--^-----------------^--------+
                |                 |
                loc1 (match start) loc2 (match end)
                     set by match() via subexp[0]

             +-----------------------------+
genbuf   :  |  scratch: being assembled    |
            |  [linebuf..loc1) + rhs       |
            |  expanded + [loc2..end)      |
             +-----------------------------+
```

The flow of a substitution is always the same: (1) `getline()`[57a] fills `linebuf`[40c] from `tfile`; (2) `match()` runs `rregexec()` on `linebuf` and records the hit as `loc1`/`loc2` pointers *into* `linebuf`; (3) `dosub()`[72d] walks `linebuf` and `rhsbuf` together, emitting the result into `genbuf`, then copies `genbuf` back into `linebuf`; (4) `putline()`[53b] writes `linebuf` to `tfile` and the new offset replaces the old one in `zero`. The two-buffer dance is what makes the `g` flag work: after `dosub()` the new `loc2` points into the updated `linebuf`, so a fresh `match(nil)` can resume scanning right where the previous hit ended.

C.10.1 Search as addresses: `/re/` and `?re?`

When `/re/` or `?re?` appears as an address, `address()`[65a] compiles the pattern and scans through the buffer line by line, wrapping around from `dol` to `zero` (or vice versa for `?re?` backward search), until a match is found or we return to where we started.

⟨`address()` *(ed.c) switch* `c` *cases* 67⟩+≡ (65a) ◁66b 77a▷

```
  case '?':
      sign = -sign;
      // Fallthrough
  case '/':
      // read the pattern and compile it in Reprog pattern[]
      compile(c);
      b = a;
      for(;;) {
          // direction
          a += sign;
          // wrap around
          if(a <= zero)
              a = dol;
          if(a > dol)
              a = zero;

          if(match(a))
              break;
```

```
        // reached back where we were without finding anything
        if(a == b)
            error(Q);
    }
    break;
```

Uses Q 44c, compile() 68c, dol 39, error() 89d, match() 69d, and zero 38d.

C.10.2 Reading and compiling a regexp: compile()

compile()[68c] reads a regular expression delimited by the eof parameter (typically / or ?), handles backslash escapes, and passes the result to regcomp() (from libregexp, see Libcore book [Pad16b]). If the delimiter is given with no pattern in between, the previously compiled pattern is reused—this is how // means "repeat the last search."

⟨globals ed.c 68a⟩+≡ (91b) ◁64a 69c▷

```
  Reprog  *pattern;
```

⟨constants ed.c 68b⟩+≡ (91b) ◁40b 69b▷

```
  ESIZE   = 256,      /* max size of reg exp */
```

⟨*function* compile*(ed.c)* 68c⟩≡ (91b)

```
  /// (main -> address) | (commands('g' | 'v') -> global) -> <>
  void
  compile(int eof)
  {
      Rune c;
      char *ep;
      // UTF8 string
      char expbuf[ESIZE];

      c = getchr();
      if(c == '\n') {
          peekc = c;
          c = eof;
      }
      if(c == eof) {
          if(!pattern)
              error(Q);
          return;
      }
      // else
      if(pattern) {
          free(pattern);
          pattern = nil;
      }
      ep = expbuf;
      do {
          if(c == '\\') {
              ⟨compile() (ed.c) sanity check ep inside expbuf 69a⟩
              // else
              ep += runetochar(ep, &c);
              c = getchr();
              if(c == '\n') {
                  error(Q);
                  return;
              }
          }
          ⟨compile() (ed.c) sanity check ep inside expbuf 69a⟩
```

```
        // else
        ep += runetochar(ep, &c);
    } while((c = getchr()) != eof && c != '\n');
    if(c == '\n')
        peekc = c;
    *ep = 0;

    // lib_regexp call
    pattern = regcomp(expbuf);
}
```
Uses Q 44c, error() 89d, getchr() 46d, pattern 68a, and peekc 46a.

⟨compile() *(ed.c) sanity check* ep *inside* expbuf 69a⟩≡ (68)
```
if(ep >= expbuf+sizeof(expbuf)) {
    error(Q);
    return;
}
```
Uses Q 44c and error() 89d.

C.10.3 match()

match()[69d] tests whether the given line matches pattern[68a]. When addr is non-null, it calls getline()[57a] to load the line into linebuf first; when addr is null, it retries the match from loc2 (the end of the previous match on the same line), which is how s///g finds successive matches. On success, loc1 and loc2 bracket the matched region in linebuf.

⟨constants ed.c 69b⟩+≡ (91b) ◁68b 70g▷
```
MAXSUB  = 9,        /* max number of sub reg exp */
```

⟨globals ed.c 69c⟩+≡ (91b) ◁68a 70b▷
```
Resub   subexp[MAXSUB];
```
Uses MAXSUB-4 69b.

⟨*function* match *(ed.c)* 69d⟩≡ (91b)
```
bool
match(int *addr)
{
    ⟨match() (ed.c) return if no pattern 69e⟩
    if(addr){
        ⟨match() (ed.c) return if addr is zero 70a⟩
        subexp[0].s.rsp = getline(*addr);
    } else
        ⟨match() (ed.c) when null addr use loc2 not getline 70e⟩
    subexp[0].e.rep = nil;

    if(rregexec(pattern, linebuf, subexp, MAXSUB)) {
        ⟨match() (ed.c) set loc1 and loc2 with matched string 70c⟩
        return true;
    }
    // else
    ⟨match() (ed.c) reset loc1 and loc2 70d⟩
    return false;
}
```
Uses MAXSUB-4 69b, getline() 57a, linebuf 40c, pattern 68a, and subexp 69c.

⟨match() *(ed.c) return if no* pattern 69e⟩≡ (69d)
```
if(!pattern)
    return false;
```
Uses pattern 68a.

⟨match() *(ed.c) return if* addr *is* zero 70a⟩≡ (69d)
```
  if(addr == zero)
      return false;
```
Uses zero 38d.

⟨globals ed.c 70b⟩+≡ (91b) ◁69c 70f▷
```
  Rune*   loc1;
  Rune*   loc2;
```

⟨match() *(ed.c) set* loc1 *and* loc2 *with matched string* 70c⟩≡ (69d)
```
  loc1 = subexp[0].s.rsp;
  loc2 = subexp[0].e.rep;
```
Uses loc1 70b, loc2 70b, and subexp 69c.

⟨match() *(ed.c) reset* loc1 *and* loc2 70d⟩≡ (69d)
```
  loc1 = loc2 = nil;
```
Uses loc1 70b and loc2 70b.

⟨match() *(ed.c) when null* addr *use* loc2 *not* getline 70e⟩≡ (69d)
```
  subexp[0].s.rsp = loc2;
```
Uses loc2 70b and subexp 69c.

C.10.4 Reading and compiling a substitution: compsub()

compsub()[70h] reads the full s/re/replacement/ command. It first calls compile()[68c] to handle the left-hand side (the pattern), then reads the right-hand side into rhsbuf[70f], handling backslash escapes and the ESCFLG marker for group references like \1. It returns true if the trailing g flag is present (global substitution on the line).

⟨globals ed.c 70f⟩+≡ (91b) ◁70b 72a▷
```
  Rune    rhsbuf[LBSIZE/sizeof(Rune)];
```
Uses LBSIZE-2 40b.

⟨constants ed.c 70g⟩+≡ (91b) ◁69b 77g▷
```
  ESCFLG  = Runemax,  /* escape Rune - user defined code */
```

⟨*function* compsub *(ed.c)* 70h⟩≡ (91b)
```
  bool
  compsub(void)
  {
      int seof, c;
      Rune *p;

      seof = getchr();
      if(seof == '\n' || seof == ' ')
          error(Q);

      // read (and compile) the regexp (left hand side)
      compile(seof);

      // read the subst (right hand side)
      p = rhsbuf;
      for(;;) {
          c = getchr();
          ⟨compsub() (ed.c) if match group 74c⟩
          else
          ⟨compsub() (ed.c) if newline and no globp 71b⟩
          else
```

```
            if(c == seof)
                break;
          // else
          *p++ = c;
          ⟨compsub() (ed.c) sanity check p inside rhsbuf 71a⟩
      }
      *p = 0;
      ⟨compsub() (ed.c) peek to check for 'g' 75e⟩
      // else
      newline();
      return false;
  }
```

Uses Q 44c, compile() 68c, error() 89d, getchr() 46d, newline() 58b, and rhsbuf 70f.

⟨compsub() *(ed.c) sanity check* p *inside* rhsbuf 71a⟩≡ (74c 70h)

```
  if(p >= &rhsbuf[LBSIZE/sizeof(Rune)])
      error(Q);
```

Uses LBSIZE-2 40b, Q 44c, error() 89d, and rhsbuf 70f.

⟨compsub() *(ed.c) if newline and no* globp 71b⟩≡ (70h)

```
  if(c == '\n' && (!globp || !globp[0])) {
      peekc = c;
      pflag = true;
      break;
  }
```

Uses globp 46c, peekc 46a, and pflag 81d.

C.10.5 substitute: s

We are now finally ready to see the code to perform substitutions. `substitute()`[71d] is the most complex command. For each line in the address range, it calls `match()`[69d] to find the pattern, then `dosub()`[72d] to build the replacement in `genbuf`[72c], copying everything before `loc1`, the replacement text (expanding `&` and `\1`–`\9` references), and everything after `loc2`. The result is copied back to `linebuf` and written to `tfile` via `putline()`[53b], replacing the old `zero` entry. With the suffix `g` flag, the match/dosub loop repeats on the same line via `match(nil)`; the optional skip count (`s3/re/.../`) skips the first `n-1` matches.

⟨commands() *switch* c *cases (ed.c)* 71c⟩+≡ (48a) ◁61c 75d▷

```
  case 's':
      nonzero();
      substitute(globp != nil);
      continue;
```

Uses globp 46c, nonzero() 58d, and substitute() 71d.

⟨*function* substitute *(ed.c)* 71d⟩≡ (91b)

```
  void
  substitute(int inglob)
  {
      int *a1;
      bool gsubf; // s/.../.../g  global subst
      int n = 0;
      ⟨substitute() (ed.c) other locals 75g⟩

      ⟨substitute() (ed.c) read optional n 75f⟩

      gsubf = compsub();

      for(a1 = addr1; a1 <= addr2; a1++) {
```

```
        // will internally set linebuf[]
        if(match(a1)){

            int *ozero;
            int m = n;

            do {
                int span = loc2-loc1;

                if(--m <= 0) {
                    // will modify linebuf[]
                    dosub();

                    if(!gsubf)
                        break;
                    // else
                    if(span == 0) { /* null RE match */
                        if(*loc2 == 0)
                            break;
                        loc2++;
                    }
                }
            } while(match(nil));

            if(m <= 0) {
                inglob |= 01;
                // will use linebuf[]
                subnewa = putline();
                *a1 &= ~01;
                ⟨substitute() (ed.c) after putline() if anymarks 77b⟩
                ⟨substitute() (ed.c) after putline() set subolda 75c⟩
                *a1 = subnewa;

                ⟨substitute() (ed.c) after putline() call append() 76a⟩
            }
        }
    }
    if(inglob == 0)
        error(Q);
}
```

Uses Q 44c, addr1 48b, addr2 48b, compsub() 70h, dosub() 72d, error() 89d, loc2 70b, match() 69d, putline() 53b, and subnewa 72a.

⟨globals ed.c 72a⟩+≡ (91b) ◁70f 72c▷

```
int subnewa;
```

⟨init() (ed.c) initializing globals 72b⟩+≡ (43a) ◁43b 76f▷

```
subnewa = 0;
```

Uses subnewa 72a.

⟨globals ed.c 72c⟩+≡ (91b) ◁72a 75b▷

```
Rune    genbuf[LBSIZE];
```

Uses LBSIZE-2 40b.

`dosub()` builds the replacement in `genbuf`: it copies `linebuf`[40c] up to `loc1` (the match start), expands the replacement string from `rhsbuf`[70f] (handling `&` for the whole match and `\1`–`\9` for subgroups via `place()`[74b]), then appends the rest of `linebuf` after `loc2`. Finally it copies `genbuf` back to `linebuf`.

⟨*function* dosub (ed.c) 72d⟩≡ (91b)

```
void
```

```
dosub(void)
{
    Rune *lp, *sp, *rp;
    int c, n;

    lp = linebuf;
    sp = genbuf;
    rp = rhsbuf;
    while(lp < loc1)
        *sp++ = *lp++;
    while(c = *rp++) {
        ⟨dosub() (ed.c) if c == '&' 74a⟩
        ⟨dosub() (ed.c) if match group 75a⟩
        // else
        *sp++ = c;
        if(sp >= &genbuf[LBSIZE])
            error(Q);
    }
    lp = loc2;
    loc2 = (sp - genbuf) + linebuf;
    while(*sp++ = *lp++)
        if(sp >= &genbuf[LBSIZE])
            error(Q);
    lp = linebuf;
    sp = genbuf;
    while(*lp++ = *sp++)
        ;
}
```

Uses `LBSIZE-2` 40b, `Q` 44c, `error()` 89d, `genbuf` 72c, `linebuf` 40c, `loc1` 70b, `loc2` 70b, and `rhsbuf` 70f.

The line `loc2 = sp - genbuf + linebuf` is the trickiest part: after assembly, `sp` points into `genbuf` where the tail (post-match text) was copied. This arithmetic converts that `genbuf` offset back into a `linebuf` pointer, so that the next call to `match(nil)` in the `g` flag loop will resume searching from the correct position in the updated `linebuf`.

A concrete trace of `s/world/earth/` on `linebuf = "hello world!"` shows the four phases of `dosub()`:

```
initial state (after match() on linebuf):
  linebuf: "hello world!\0"
            ^     ^    ^
            |     loc1 loc2
            lp starts here
  rhsbuf:  "earth\0"
  genbuf:  (empty, sp at start)

phase 1: copy linebuf[..loc1) -> genbuf
  genbuf:  "hello "
                 ^sp

phase 2: expand rhsbuf into genbuf
  (no & or \N, so plain copy)
  genbuf:  "hello earth"
                      ^sp

phase 3: the famous pointer shuffle
  loc2 = (sp - genbuf) + linebuf
```

```
      =    11         + linebuf
      -> points into linebuf at offset 11 (the '!')
   This is where the next match(nil) would start.

 phase 4: copy linebuf[loc2..] -> genbuf, then genbuf -> linebuf
   genbuf:  "hello earth!\0"
   linebuf: "hello earth!\0"
```

Phase 3 is subtle because `loc2` must end up pointing into the *new* `linebuf`, not the old one. At the moment of the assignment, `linebuf` still holds the pre-substitution text, but we know the new prefix ("hello earth") will occupy bytes `linebuf[0..11)` once phase 4 runs. So `sp-genbuf` (the length of that prefix) plus `linebuf` (the base) is the correct future location. It is a deliberate compute-first, copy-later trick—if the assignment were reordered after the final copy, `sp` would already have advanced and the arithmetic would break.

C.10.6 Advanced substitutions

The code of `dosub()`[72d], `compsub()`[70h], and `substitute()`[71d] got simplified before to show just the core code path. Many additional search and replace features require additional code as shown in the following sections.

Matched string reference: `s/.../...&.../`

⟨`dosub()` *(ed.c) if* `c == '&'` 74a⟩≡ (72d)

```
if(c == '&'){
    sp = place(sp, loc1, loc2);
    continue;
}
```

Uses `loc1` 70b, `loc2` 70b, and `place()` 74b.

⟨*function* `place` *(ed.c)* 74b⟩≡ (91b)

```
Rune*
place(Rune *sp, Rune *l1, Rune *l2)
{

    while(l1 < l2) {
        *sp++ = *l1++;
        if(sp >= &genbuf[LBSIZE])
            error(Q);
    }
    return sp;
}
```

Uses `LBSIZE-2` 40b, `Q` 44c, `error()` 89d, and `genbuf` 72c.

Matched groups: `s/.../...\1.../`

⟨`compsub()` *(ed.c) if match group* 74c⟩≡ (70h)

```
if(c == '\\') {
    c = getchr();
    *p++ = ESCFLG;
    ⟨compsub() (ed.c) sanity check p inside rhsbuf 71a⟩
}
```

Uses `ESCFLG-5` 70g and `getchr()` 46d.

⟨dosub() *(ed.c) if match group* 75a⟩≡ (72d)

```
if(c == ESCFLG && (c = *rp++) >= '1' && c < MAXSUB+'0') {
    n = c-'0';
    if(subexp[n].s.rsp && subexp[n].e.rep) {
        sp = place(sp, subexp[n].s.rsp, subexp[n].e.rep);
        continue;
    }
    error(Q);
}
```

Uses `ESCFLG-5` 70g, `MAXSUB-4` 69b, `Q` 44c, `error()` 89d, `place()` 74b, and `subexp` 69c.

undo substitution: u

The `u` command provides a minimal undo: it remembers the previous `zero` entry (`subolda`[75b]) before the last substitution replaced it with `subnewa`[72a], and swaps it back. This only undoes the most recent substitution on a single line—far from a general undo facility.

⟨globals ed.c 75b⟩+≡ (91b) ◁72c 76d▷

```
int subolda;
```

⟨substitute() *(ed.c) after* putline() *set* subolda 75c⟩≡ (71d)

```
subolda = *a1;
```

Uses `subolda` 75b.

⟨commands() *switch* c *cases (ed.c)* 75d⟩+≡ (48a) ◁71c 76c▷

```
case 'u':
    nonzero();
    newline();
    if((*addr2&~01) != subnewa)
        error(Q);
    *addr2 = subolda;
    dot = addr2;
    continue;
```

Uses `Q` 44c, `addr2` 48b, `dot` 39, `error()` 89d, `newline()` 58b, `nonzero()` 58d, `subnewa` 72a, and `subolda` 75b.

global subst: s/.../.../g

⟨compsub() *(ed.c) peek to check for* 'g' 75e⟩≡ (70h)

```
peekc = getchr();
if(peekc == 'g') {
    peekc = 0;
    newline();
    return true;
}
```

Uses `getchr()` 46d, `newline()` 58b, and `peekc` 46a.

Skipped matches: s3/.../.../

⟨substitute() *(ed.c) read optional* n 75f⟩≡ (71d)

```
n = getnum();    /* OK even if n==0 */
```

Uses `getnum()` 66a.

Append and subst

⟨substitute() *(ed.c) other locals* 75g⟩≡ (71d)

```
int *mp, nl;
```

⟨substitute() *(ed.c) after* putline() *call* append() 76a⟩≡ (71d)

```
  ozero = zero;
  nl = append(getsub, a1);
  addr2 += nl;
  nl += zero-ozero;
  a1 += nl;
```

Uses addr2 48b, append() 51b, getsub() 76b, and zero 38d.

⟨*function* getsub *(ed.c)* 76b⟩≡ (91b)

```
  int
  getsub(void)
  {
      Rune *p1, *p2;

      p1 = linebuf;
      if((p2 = linebp) == 0)
          return EOF;
      while(*p1++ = *p2++)
          ;
      linebp = 0;
      return 0;
  }
```

Uses EOF-9 90c, linebp 85c, and linebuf 40c.

mark: k

The k command *marks* a line with a lowercase letter (ka through kz). The mark is stored in names[c-'a'] as the line's zero entry (with the low bit cleared—since that bit is used by global()[78]). The ' (apostrophe) address retrieves a marked line by scanning zero for the matching offset.

⟨commands() *switch* c *cases (ed.c)* 76c⟩+≡ (48a) ◁75d 77e▷

```
  case 'k':
      nonzero();
      c = getchr();
      if(c < 'a' || c > 'z')
          error(Q);
      newline();
      names[c-'a'] = *addr2 & ~01;
      anymarks |= 01;
      continue;
```

Uses Q 44c, addr2 48b, anymarks 76e, error() 89d, getchr() 46d, names 76d, newline() 58b, and nonzero() 58d.

⟨globals ed.c 76d⟩+≡ (91b) ◁75b 76e▷

```
  int names[26];
```

⟨globals ed.c 76e⟩+≡ (91b) ◁76d 80b▷

```
  int anymarks;
```

⟨init() *(ed.c) initializing globals* 76f⟩+≡ (43a) ◁72b 77d▷

```
  anymarks = 0;
```

Uses anymarks 76e.

⟨address() *(ed.c) switch* c *cases* 77a⟩+≡ (65a) ◁67

```
case '\'':
    c = getchr();
    if(opcnt || c < 'a' || c > 'z')
        error(Q);
    a = zero;
    do {
        a++;
    } while(a <= dol && names[c-'a'] != (*a & ~01));
    break;
```

Uses Q 44c, dol 39, error() 89d, getchr() 46d, names 76d, and zero 38d.

⟨substitute() *(ed.c) after* putline() *if* anymarks 77b⟩≡ (71d)

```
if(anymarks) {
    for(mp=names; mp<&names[26]; mp++)
        if(*mp == *a1)
            *mp = subnewa;
}
```

Uses anymarks 76e, names 76d, and subnewa 72a.

⟨init() *(ed.c) locals* 77c⟩≡ (43a)

```
int *markp;
```

⟨init() *(ed.c) initializing globals* 77d⟩+≡ (43a) ◁76f 86b▷

```
for(markp = names; markp < &names[26]; )
    *markp++ = 0;
```

Uses names 76d.

C.10.7 global commands: g/re/cmd and v/re/cmd

The g/re/cmd command (and its inverse v/re/cmd) works in two phases. In phase 1, it scans all lines in the address range and *marks* those that match (or don't match, for v) the pattern by setting the low bit of their zero entry (*a1 |= 01—this is the "bit-stealing trick" that tline starting at 2 makes possible). In phase 2, it iterates over zero, and for each marked line, clears the mark, sets globp[46c] to the command string (stored in the local globuf), and calls commands()[48a] recursively. The two-phase design is necessary because executing commands can change the buffer (e.g., g/re/d deletes lines), so we cannot match and execute at the same time.

⟨commands() *switch* c *cases (ed.c)* 77e⟩+≡ (48a) ◁76c 77f▷

```
case 'g':
    global(1);
    continue;
```

Uses global() 78.

⟨commands() *switch* c *cases (ed.c)* 77f⟩+≡ (48a) ◁77e 80a▷

```
case 'v':
    global(0);
    continue;
```

Uses global() 78.

⟨constants ed.c 77g⟩+≡ (91b) ◁70g 85d▷

```
GBSIZE  = 256,      /* max size of global command */
```

⟨*function* global *(ed.c)* 78⟩≡ (91b)

```
void
global(int k)
{
    Rune *gp, globuf[GBSIZE];
    int c, *a1;

    if(globp)
        error(Q);

    setwide();
    squeeze(dol > zero);

    // readding a '/' or '?'
    c = getchr();
    if(c == '\n')
        error(Q);

    // reading the whole pattern until corresponding ending '/' or '?'
    compile(c);

    gp = globuf;
    while((c=getchr()) != '\n') {
        if(c == EOF)
            error(Q);
        if(c == '\\') {
            c = getchr();
            if(c != '\n')
                *gp++ = '\\';
        }
        *gp++ = c;
        if(gp >= &globuf[GBSIZE-2])
            error(Q);
    }
    if(gp == globuf)
        *gp++ = 'p';
    *gp++ = '\n';
    *gp = 0;

    for(a1=zero; a1<=dol; a1++) {
        *a1 &= ~01;
        if(a1 >= addr1 && a1 <= addr2 && match(a1) == k)
            *a1 |= 01;
    }

    ⟨global() (ed.c) if g/.../d command, call optimized gdelete() 88d⟩

    for(a1=zero; a1<=dol; a1++) {
        if(*a1 & 01) {
            *a1 &= ~01;
            dot = a1;
            globp = globuf;
            // recurse!
            commands();
            a1 = zero; // zero may have grown and move, need update a1
        }
    }
}
```

Uses EOF-9 90c, GBSIZE-6 77g, Q 44c, addr1 48b, addr2 48b, commands() 48a, compile() 68c, dol 39, dot 39, error() 89d,

getchr() 46d, globp 46c, match() 69d, setwide() 50d, squeeze() 51a, and zero 38d.

There is an important subtlety in the phase 2 loop: after each `commands()` call, `a1` is reset to `zero` rather than just incrementing. This is because `commands()` during that recursive call may call `append()`[51b], which may `realloc` the `zero` array, moving it to a new address. All saved pointers into the old array become dangling. Resetting `a1 = zero` picks up the (possibly new) base, and the loop re-scans from the beginning—this is safe because each executed line has its mark cleared, so it won't be processed again.

Visually, a `g/foo/d` on a four-line buffer goes through the following two phases. Bit 0 of each `zero` entry is stolen as the "matched" mark; I show it as `M` when set, and `'.'` when clear:

```
initial:                 after phase 1 (mark):
zero[1] = 0x40           zero[1] = 0x40   .   "alpha"
zero[2] = 0x48           zero[2] = 0x49   M   "foobar"   <- matched
zero[3] = 0x56           zero[3] = 0x56   .   "beta"
zero[4] = 0x62           zero[4] = 0x63   M   "foo"      <- matched

phase 2 iteration:
  a1 = zero+1          .  skip
  a1 = zero+2          M  clear mark; dot=a1;
                           globp = "d\n"; commands();
                           (runs 'd' -> rdelete shifts entries down)
                           a1 = zero (restart from base)
  a1 = zero+1          .  skip (old alpha)
  a1 = zero+2          .  skip (was beta, shifted down)
  a1 = zero+3          M  clear mark; delete; a1 = zero
  a1 = zero+1          .  skip
  a1 = zero+2          .  skip
  dol is now zero+2    loop exits
```

Three consequences follow from this design. (1) The marks must live *inside* `zero` rather than in a parallel bitmap, because `rdelete()`[60e] shifts `zero` entries around and a parallel bitmap would go out of sync instantly. Putting the mark in the low bit of the offset means delete, move, and substitute all carry marks along for free. (2) The low bit is safe to steal because `tline` starts at 2 and grows in multiples of 2 (aligned `Rune` boundaries), so legitimate offsets never have bit 0 set. (3) The restart-from-`zero` loop is $O(n^2)$ in the worst case (one full pass per matched line), which is why `gdelete()`[89a] exists as a special case for `g/.../d` (see Section C.12.4).

With this in hand, the origin of `grep` is now obvious. The classic `ed` one-liner `g/re/p` means "for every line matching the regular expression `re`, execute the `p` command (print)"—i.e., global / regular expression / print. This was such a common idiom that Ken Thompson and the early UNIX folks pulled it out into its own standalone tool, named literally after the ed command: `g` - `re` - `p` → `grep`.

C.11 Advanced features

The previous sections covered the core editing commands, the addressing system, and search-and-replace. This section collects the remaining commands of `ed`. None of those commands introduce new mechanisms—they are mostly small additions to the command dispatcher in `commands()`[48a].

C.11.1 Running a shell command: !

The `!` command forks a child process, executes the rest of the line via `/bin/rc -c`, and waits for it to finish. In verbose mode it prints `!` when the command completes. This command is less useful on Plan 9 where you

typically have multiple windows, but was essential back in the old UNIX days on a single-terminal system.

⟨commands() *switch* c *cases (ed.c)* 80a⟩+≡ (48a) ◁77f 82d▷
```
  case '!':
      callunix();
      continue;
```
Uses callunix() 80c.

⟨globals ed.c 80b⟩+≡ (91b) ◁76e 80d▷
```
  bool waiting;
```

⟨*function* callunix *(ed.c)* 80c⟩≡ (91b)
```
  void
  callunix(void)
  {
      int c, pid;
      Rune rune;
      char buf[512];
      char *p;

      setnoaddr();

      p = buf;
      while((c=getchr()) != EOF && c != '\n')
          if(p < &buf[sizeof(buf) - 6]) {
              rune = c;
              p += runetochar(p, &rune);
          }
      *p = '\0';

      pid = fork();
      if(pid == 0) {
          // child
          execl("/bin/rc", "rc", "-c", buf, nil);
          // should not be reached
          exits("execl failed");
      }
      // else, parent
      waiting = true;
      while(waitpid() != pid)
          ;
      waiting = false;
      if(vflag)
          putst("!");
  }
```
Uses EOF-9 90c, getchr() 46d, putst() 44d, setnoaddr() 59b, vflag 41a, and waiting 80b.

C.11.2 Advanced listing: l

The l (list) command is like p but makes non-printing characters visible: tabs become \t, backspaces \b, and other control or high characters are shown as \xNNNN hex escapes. Long lines are folded at column 72. The listf flag is checked inside putchr()[45b] to trigger this special rendering.

⟨globals ed.c 80d⟩+≡ (91b) ◁80b 81c▷
```
  // for displaying special chars, 'l' list flag
  bool listf;
```

⟨commands *before* 'p' *case* 81a⟩≡ (58a)

```
  case 'l':
      listf = true;
      // fallthrough:
```

Uses listf 80d.

⟨putchr() *if* listf 81b⟩≡ (45b)

```
  if(listf) {
      if(c == '\n') {
          if(linp != line && linp[-1] == ' ') {
              *lp++ = '\\';
              *lp++ = 'n';
          }
      } else {
          if(col > (72-6-2)) {
              col = 8;
              *lp++ = '\\';
              *lp++ = '\n';
              *lp++ = '\t';
          }
          col++;
          if(c=='\b' || c=='\t' || c=='\\') {
              *lp++ = '\\';
              if(c == '\b')
                  c = 'b';
              else
              if(c == '\t')
                  c = 't';
              col++;
          } else
          if(c<' ' || c>='\177') {
              *lp++ = '\\';
              *lp++ = 'x';
              *lp++ =  hex[c>>12];
              *lp++ =  hex[c>>8&0xF];
              *lp++ =  hex[c>>4&0xF];
              c     =  hex[c&0xF];
              col += 5;
          }
      }
  }
```

Uses col 40e, hex 81c, line 40a, linp 40a, and listf 80d.

⟨globals ed.c 81c⟩+≡ (91b) ◁80d 81d▷

```
  char    hex[]   = "0123456789abcdef";
```

Uses hex 81c.

⟨globals ed.c 81d⟩+≡ (91b) ◁81c 82c▷

```
  bool pflag;
```

The pflag mechanism lets commands request an automatic print after execution. For example, s/foo/bar/p sets pflag, and at the top of the commands()[48a] loop, if pflag is set, dot is printed before reading the next command. The suffixes p, l, and n can be appended to many commands (handled in newline()[58b]).

⟨commands() *in for loop, if* pflag 81e⟩≡ (48a)

```
  if(pflag) {
      pflag = false;
      addr1 = addr2 = dot;
      printcom();
  }
```

Uses addr1 48b, addr2 48b, dot 39, pflag 81d, and printcom() 58c.

⟨newline() *if special chars* pln 82a⟩≡ (58b)
```
  if(c == 'p' || c == 'l' || c == 'n') {
      pflag = true;
      if(c == 'l')
          listf = true;
      else
      if(c == 'n')
          listn = true;
      c = getchr();
      if(c == '\n')
          return;
  }
```
Uses getchr() 46d, listf 80d, listn 82c, and pflag 81d.

⟨printcom() *reset flags* 82b⟩≡ (58c)
```
  listf = false;
  listn = false;
  pflag = false;
```
Uses listf 80d, listn 82c, and pflag 81d.

C.11.3 Advanced listing: n

The n suffix prepends the line number before each printed line. It is useful when the user is working with addresses and wants to remember which line is which without typing = repeatedly. Like l, n is implemented as a flag set in newline()[58b] and consumed by printcom()[58c].

⟨globals ed.c 82c⟩+≡ (91b) ◁81d 83c▷
```
  // 'n' flag
  bool listn;
```

⟨commands() *switch* c *cases (ed.c)* 82d⟩+≡ (48a) ◁80a 82f▷
```
  case 'n':
      listn = true;
      newline();
      printcom();
      continue;
```
Uses listn 82c, newline() 58b, and printcom() 58c.

⟨printcom() *if* listn 82e⟩≡ (58c)
```
  if(listn) {
      count = a1-zero;
      putd();
      putchr(L'\t');
  }
```
Uses count 40d, listn 82c, putchr() 45b, putd() 55b, and zero 38d.

C.11.4 joining lines: j

The j command concatenates multiple lines into one. It copies all lines in the range into genbuf end-to-end, writes the result back as a single line via putline()[53b], then deletes the extra lines with rdelete()[60e].

⟨commands() *switch* c *cases (ed.c)* 82f⟩+≡ (48a) ◁82d 83b▷
```
  case 'j':
      if(!given)
          addr2++;
      newline();
      join();
      continue;
```
Uses addr2 48b, given 64a, join() 83a, and newline() 58b.

⟨*function* join *(ed.c)* 83a⟩≡ (91b)
```
void
join(void)
{
    Rune *gp, *lp;
    int *a1;

    nonzero();
    gp = genbuf;
    for(a1=addr1; a1<=addr2; a1++) {
        lp = getline(*a1);
        while(*gp = *lp++)
            if(gp++ >= &genbuf[LBSIZE-sizeof(Rune)])
                error(Q);
    }
    lp = linebuf;
    gp = genbuf;
    while(*lp++ = *gp++)
        ;
    *addr1 = putline();
    if(addr1 < addr2)
        rdelete(addr1+1, addr2);
    dot = addr1;
}
```
Uses `LBSIZE-2` 40b, `Q` 44c, `addr1` 48b, `addr2` 48b, `dot` 39, `error()` 89d, `genbuf` 72c, `getline()` 57a, `linebuf` 40c, `nonzero()` 58d, `putline()` 53b, and `rdelete()` 60e.

C.11.5 browse: b

The `b` (browse) command is a paging facility: it prints `bpagesize` lines (default 20) starting from the current address. You can go forward or backward with `b+` or `b-`, and change the page size with `b30`.

⟨`commands()` *switch* `c` *cases (ed.c)* 83b⟩+≡ (48a) ◁82f 84a▷
```
case 'b':
    nonzero();
    browse();
    continue;
```
Uses `browse()` 83d and `nonzero()` 58d.

⟨`globals ed.c` 83c⟩+≡ (91b) ◁82c 85b▷
```
int bpagesize = 20;
```
Uses `bpagesize` 83c.

⟨*function* browse *(ed.c)* 83d⟩≡ (91b)
```
void
browse(void)
{
    int forward, n;
    static int bformat, bnum; /* 0 */

    forward = 1;
    peekc = getchr();
    if(peekc != '\n'){
        if(peekc == '-' || peekc == '+') {
            if(peekc == '-')
                forward = 0;
            getchr();
        }
```

```
        n = getnum();
        if(n > 0)
            bpagesize = n;
    }
    newline();
    if(pflag) {
        bformat = listf;
        bnum = listn;
    } else {
        listf = bformat;
        listn = bnum;
    }
    if(forward) {
        addr1 = addr2;
        addr2 += bpagesize;
        if(addr2 > dol)
            addr2 = dol;
    } else {
        addr1 = addr2-bpagesize;
        if(addr1 <= zero)
            addr1 = zero+1;
    }
    printcom();
}
```

Uses `addr1` 48b, `addr2` 48b, `bpagesize` 83c, `dol` 39, `getchr()` 46d, `getnum()` 66a, `listf` 80d, `listn` 82c, `newline()` 58b, `peekc` 46a, `pflag` 81d, `printcom()` 58c, and `zero` 38d.

C.11.6 edit remembered file: `e`

The `e` command re-initializes the editor (calling `init()`[43a] to reset `tfile`[37b] and `zero`[38d]) and reads a new file, effectively starting a fresh editing session. `E` is the force variant that skips the unsaved-changes check.

⟨`commands()` *switch* `c` *cases (ed.c)* 84a⟩+≡ (48a) ◁83b

```
case 'E':
    fchange = false;
    c = 'e';
    // Fallthrough
case 'e':
    setnoaddr();
    if(vflag && fchange) {
        fchange = false;
        error(Q);
    }
    filename(c);
    init();
    addr2 = zero;
    goto caseread;
```

Uses `Q` 44c, `addr2` 48b, `error()` 89d, `fchange` 44a, `filename()` 49c, `init()` 43a, `setnoaddr()` 59b, `vflag` 41a, and `zero` 38d.

C.11.7 Append-only files

Plan 9 supports append-only files (the `DMAPPEND` mode bit), used for log files that can only grow. `ed` checks for this flag when opening a file for reading and prints a warning, since writing back to such a file would fail.

⟨`commands()` *other locals* 84b⟩+≡ (48a) ◁64c

```
Dir *d;
```

⟨commands() *in* r *case, if append only file* 85a⟩≡ (49a)

```
  if((d = dirfstat(io)) != nil){
      if(d->mode & DMAPPEND)
          print("warning: %s is append only\n", file);
      free(d);
  }
```

Uses file 48d and io 48e.

C.12 Optimizations

The simplified putline()[53b] and getline()[57a] shown earlier do a seek+write/read for every single line. The original ed versions use a (userspace) *block cache*: getblock()[86c] maintains two BLKSIZE-byte buffers (ibuff for reads, obuff for writes) and only performs actual I/O when the requested block is not already cached. This was essential on early UNIX systems that lacked a kernel buffer cache, and it remains a significant optimization when editing large files.

C.12.1 Optimized putline()

The real putline()[53b] writes linebuf to tfile through getblock()[86c], handling block boundaries: when the current block is full (nleft == 0), it advances to the next block. The offset arithmetic with tline uses block-aligned addresses.

⟨globals ed.c 85b⟩+≡ (91b) ◁83c 85c▷

```
  int nleft;
```

⟨globals ed.c 85c⟩+≡ (91b) ◁85b 86a▷

```
  Rune*   linebp;
```

⟨constants ed.c 85d⟩+≡ (91b) ◁77g 87▷

```
  BLKSIZE = 4096,     /* block size in temp file */
```

⟨putline() *if* opti 85e⟩≡ (53b)

```
  if(opti) return putline_opti();
```

Uses opti 91b and putline_opti() 85f.

⟨*function* putline_opti *(ed.c)* 85f⟩≡ (91b)

```
  /// main -> commands('r') -> append -> <>
  int
  putline_opti(void)
  {
      Rune *lp, *bp;
      int nl, tl;

      fchange = true;
      lp = linebuf;
      tl = tline;

      bp = getblock(tl, OWRITE);
      nl = nleft;
      tl &= ~((BLKSIZE/sizeof(Rune))-1);
      while(*bp = *lp++) {
          if(*bp++ == '\n') {
              bp[-1] = 0;
              linebp = lp;
              break;
          }
```

```
        nl -= sizeof(Rune);
        if(nl == 0) {
            tl += BLKSIZE/sizeof(Rune);
            bp = getblock(tl, OWRITE);
            nl = nleft;
        }
    }

    nl = tline;
    tline += ((lp-linebuf) + 03) & 077776;
    return nl;
}
```

Uses BLKSIZE-7 85d, fchange 44a, getblock() 86c, linebp 85c, linebuf 40c, nleft 85b, and tline 38a.

C.12.2 getblock()

getblock()[86c] is the heart of the block cache. It converts a line offset atl into a block number and an offset within that block.

⟨globals ed.c 86a⟩+≡ (91b) ◁85c 89b▷

```
// index in ibuff
int iblock;
// index in obuff
int oblock;

int ichanged;
```

⟨init() *(ed.c) initializing globals* 86b⟩+≡ (43a) ◁77d

```
iblock = -1;
oblock = -1;
ichanged = 0;
```

Uses iblock 86a, ichanged 86a, and oblock 86a.

⟨*function* getblock*(ed.c)* 86c⟩≡ (91b)

```
/// putline | getline -> <>
Rune*
getblock(int atl, int iof)
{
    int bno; // block number
    int off; // offset

    static uchar ibuff[BLKSIZE];
    static uchar obuff[BLKSIZE];

    bno = atl / (BLKSIZE/sizeof(Rune));
    /* &~3 so the ptr is aligned to 4 (?) */
    off = (atl*sizeof(Rune)) & (BLKSIZE-1) & ~3;
    if(bno >= NBLK) {
        lastc = '\n';
        error(T);
    }
    nleft = BLKSIZE - off;
    if(bno == iblock) {
        ichanged |= iof;
        return (Rune*)(ibuff+off);
    }
    if(bno == oblock)
        return (Rune*)(obuff+off);
    if(iof == OREAD) {
```

```
        if(ichanged)
            blkio(iblock, ibuff, write);
        ichanged = 0;
        iblock = bno;
        blkio(bno, ibuff, read);
        return (Rune*)(ibuff+off);
    }
    if(oblock >= 0)
        blkio(oblock, obuff, write);
    oblock = bno;
    return (Rune*)(obuff+off);
}
```

Uses BLKSIZE-7 85d, NBLK-8 87, T 43c, blkio() 88a, error() 89d, iblock 86a, ichanged 86a, lastc 46b, nleft 85b, and oblock 86a.

```
⟨constants ed.c 87⟩+≡                                (91b) ◁85d 90c▷
  NBLK    = 8191,     /* max size of temp file */
```

If the requested block is already in `ibuff` (the read cache) or `obuff` (the write cache), it returns a pointer directly into the cached buffer—no I/O needed. Otherwise, it evicts the oldest cached block (flushing `ibuff` to disk via `blkio()`[88a] if it was modified), loads the new block, and returns a pointer into it.

The `ichanged` flag is the dirty bit for `ibuff`: when a write request (`iof == OWRITE`) hits the `ibuff` cache, `ichanged` is set via `ichanged |= iof`. Later, if `ibuff` must be evicted for a read, it is flushed first. The `obuff` path has no such flag—it is always flushed on eviction because it is only used by writes, so it is always dirty.

```
tfile (on disk):
+-----------+-----------+-----------+----------+-----...
|  block 0  |  block 1  |  block 2  |  block 3  |
|  4096 B   |  4096 B   |  4096 B   |  4096 B   |
+-----------+-----------+-----------+----------+-----...

getblock(atl, iof):
    bno = atl / (BLKSIZE/sizeof(Rune))     -> which block
    off = (atl*sizeof(Rune)) & (BLKSIZE-1) -> byte offset within block

In-memory caches (ed.c globals):

    ibuff[4096]  <- iblock (currently cached read block)
    obuff[4096]  <- oblock (currently cached write block)
    ichanged                (dirty bit for ibuff)

Three cases:

    1. bno == iblock      -> cache hit on read buffer
                             return &ibuff[off]
                             if iof==OWRITE, set ichanged

    2. bno == oblock      -> cache hit on write buffer
                             return &obuff[off]

    3. miss               -> evict the appropriate buffer
                             (flush if ichanged or oblock>=0),
                             read or allocate the new block,
                             return its address
```

The split between `ibuff` and `obuff` avoids ping-pong: a typical `ed` session keeps reading lines from one part of `tfile` (cached in `ibuff`) while writing modified lines to the growing tail of `tfile` (cached in `obuff`). With a single buffer the cache would constantly be evicted as the program alternates reads and writes; with two, both stay warm.

⟨*function* `blkio` *(ed.c)* 88a⟩≡ (91b)
```
void
blkio(int b, uchar *buf, long (*iofcn)(int, void *, long))
{
    seek(tfile, b*BLKSIZE, SEEK__START);
    if((*iofcn)(tfile, buf, BLKSIZE) != BLKSIZE) {
        error(T);
    }
}
```
Uses `BLKSIZE-7` 85d, `T` 43c, `error()` 89d, and `tfile` 37b.

C.12.3 Optimized `getline()`

⟨`getline()` *if* `opti` 88b⟩≡ (57a)
```
if(opti) return getline_opti(tl);
```
Uses `getline_opti()` 88c and `opti` 91b.

⟨*function* `getline_opti` *(ed.c)* 88c⟩≡ (91b)
```
/// printcom | putfile -> <>
Rune*
getline_opti(int tl)
{
    Rune *lp, *bp;
    int nl;

    lp = linebuf;
    bp = getblock(tl, OREAD);
    nl = nleft;
    tl &= ~((BLKSIZE/sizeof(Rune)) - 1);
    while(*lp++ = *bp++) {
        nl -= sizeof(Rune);
        if(nl == 0) {
            tl += BLKSIZE/sizeof(Rune);
            bp = getblock(tl, OREAD);
            nl = nleft;
        }
    }
    return linebuf;
}
```
Uses `BLKSIZE-7` 85d, `getblock()` 86c, `linebuf` 40c, and `nleft` 85b.

C.12.4 `gdelete()`

The generic `g/re/cmd` loop calls `commands()`[48a] once per marked line, which for `g/re/d` would result in $O(n^2)$ shifting in `rdelete()`[60e]. `gdelete()`[89a] below special-cases this: it makes a single pass over `zero`, copying unmarked entries down and skipping marked ones, achieving $O(n)$ deletion.

⟨`global()` *(ed.c) if* `g/.../d` *command, call optimized* `gdelete()` 88d⟩≡ (78)
```
/*
 * Special case: g/.../d (avoid n^2 algorithm)
 */
if(globuf[0] == 'd' && globuf[1] == '\n' && globuf[2] == 0) {
```

```
    gdelete();
    return;
}
```

Uses gdelete() 89a.

⟨*function* gdelete *(ed.c)* 89a⟩≡ (91b)

```
void
gdelete(void)
{
    int *a1, *a2, *a3;

    a3 = dol;
    for(a1=zero; (*a1&01)==0; a1++)
        if(a1>=a3)
            return;
    for(a2=a1+1; a2<=a3;) {
        if(*a2 & 01) {
            a2++;
            dot = a1;
        } else
            *a1++ = *a2++;
    }
    dol = a1-1;
    if(dot > dol)
        dot = dol;
    fchange = true;
}
```

Uses dol 39, dot 39, fchange 44a, and zero 38d.

C.13 Error management

ed's error handling is based on setjmp/longjmp (see the LIBCORE book [Pad16b]), which act as a poor man's exception mechanism.

⟨globals ed.c 89b⟩+≡ (91b) ◁86a 90d▷

```
jmp_buf savej;
```

C.13.1 error()

Before calling commands()[48a], main() calls setjmp(savej) to save the execution context.

⟨main() *(ed.c) before* commands() 89c⟩≡ (40g)

```
setjmp(savej);
```

Uses savej 89b.

When error()[89d] is called anywhere in the program, it prints ? followed by a short message, resets global state, and calls longjmp(savej, 1) to jump back to just before commands(), ready to read the next command.

⟨*function* error *(ed.c)* 89d⟩≡ (91b)

```
void
error(char *s)
{
    error_1(s);
    longjmp(savej, 1);
}
```

Uses error_1() 90a and savej 89b.

⟨*function* `error_1` *(ed.c)* 90a⟩≡ (91b)
```
void
error_1(char *s)
{
    int c;

    ⟨error_1() (ed.c)) reset globals 90b⟩
    putchr(L'?');
    putst(s);
}
```
Uses `putchr()` 45b and `putst()` 44d.

⟨`error_1()` *(ed.c)) reset globals* 90b⟩≡ (90a)
```
wrapp = false;
listf = false;
listn = false;
count = 0;

seek(STDIN, 0, SEEK__END);
pflag = false;

if(globp)
    lastc = '\n';
globp = nil;

peekc = lastc;
if(lastc)
    for(;;) {
        c = getchr();
        if(c == '\n' || c == EOF)
            break;
    }

if(io > 0) {
    close(io);
    io = -1;
}
```
Uses `EOF-9` 90c, `count` 40d, `getchr()` 46d, `globp` 46c, `io` 48e, `lastc` 46b, `listf` 80d, `listn` 82c, `peekc` 46a, `pflag` 81d, and `wrapp` 55c.

⟨`constants ed.c` 90c⟩+≡ (91b) ◁87
```
EOF = -1,
```

C.13.2 Notes/signals management

Plan 9 uses *notes* instead of UNIX*signals* (see the Kernel book [Pad14]). On an “interrupt” note, `ed` prints a newline and jumps back to the command loop via `notejmp` (the Plan 9 equivalent of `siglongjmp`). On a “hangup” note (terminal disconnected), it calls `rescue()`[91a], which saves the buffer to `ed.hup` before exiting—a safety net so you do not lose unsaved work.

⟨`globals ed.c` 90d⟩+≡ (91b) ◁89b
```
bool rescuing;
```

⟨*function* `notifyf` *(ed.c)* 90e⟩≡ (91b)
```
void
notifyf(void *a, char *s)
{
    if(strcmp(s, "interrupt") == 0){
```

```
        if(rescuing || waiting)
            noted(NCONT);
        putchr(L'\n');
        lastc = '\n';
        error_1(Q);
        notejmp(a, savej, 0);
    }
    if(strcmp(s, "hangup") == 0){
        if(rescuing)
            noted(NDFLT);
        rescue();
    }
    fprint(STDERR, "ed: note: %s\n", s);
    abort();
}
```

Uses `Q` 44c, `error_1()` 90a, `lastc` 46b, `putchr()` 45b, `rescue()` 91a, `rescuing` 90d, `savej` 89b, and `waiting` 80b.

⟨*function* `rescue` *(ed.c)* 91a⟩≡ (91b)

```
void
rescue(void)
{
    rescuing = true;
    if(dol > zero) {
        addr1 = zero+1;
        addr2 = dol;
        io = create("ed.hup", OWRITE, 0666);
        if(io > 0){
            Binit(&iobuf, io, OWRITE);
            putfile();
        }
    }
    fchange = false;
    quit();
}
```

Uses `addr1` 48b, `addr2` 48b, `dol` 39, `fchange` 44a, `io` 48e, `iobuf` 48e, `putfile()` 56, `quit()` 44b, `rescuing` 90d, and `zero` 38d.

C.14 Extra code

⟨`ed/ed.c` 91b⟩≡

```
/*
 * Editor
 */
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <regexp.h>

enum
{
    ⟨constants ed.c 38b⟩
};

⟨globals ed.c 37a⟩
bool opti = false;

// forward declarations
void    add(int);
int*    address(void);
```

```
int     append(int(*)(void), int*);
void    browse(void);
void    callunix(void);
void    commands(void);
void    compile(int);
int     compsub(void);
void    dosub(void);
void    error(char*);
int     match(int*);
void    exfile(int);
void    filename(int);
Rune*   getblock(int, int);
int     getchr(void);
int     getcopy(void);
int     getfile(void);
Rune*   getline(int);
int     getnum(void);
int     getsub(void);
int     gettty(void);
void    global(int);
void    init(void);
void    join(void);
void    move(int);
void    newline(void);
void    nonzero(void);
void    notifyf(void*, char*);
Rune*   place(Rune*, Rune*, Rune*);
void    printcom(void);
void    putchr(int);
void    putd(void);
void    putfile(void);
int     putline(void);
void    putshst(Rune*);
void    putst(char*);
void    quit(void);
void    rdelete(int*, int*);
void    regerror(char *);
void    reverse(int*, int*);
void    setnoaddr(void);
void    setwide(void);
void    squeeze(int);
void    substitute(int);

⟨function main(ed.c) 40g⟩

⟨function commands(ed.c) 48a⟩

// Command helpers
⟨function printcom(ed.c) 58c⟩

⟨function address(ed.c) 65a⟩
⟨function getnum(ed.c) 66a⟩

// ???
⟨function setwide(ed.c) 50d⟩
⟨function setnoaddr(ed.c) 59b⟩
⟨function nonzero(ed.c) 58d⟩
⟨function squeeze(ed.c) 51a⟩
⟨function newline(ed.c) 58b⟩
⟨function filename(ed.c) 49c⟩
```

```
// Writing files
⟨function exfile(ed.c) 55a⟩

// Error management
⟨function error_1(ed.c) 90a⟩
⟨function error(ed.c) 89d⟩
⟨function rescue(ed.c) 91a⟩

// Note management
⟨function notifyf(ed.c) 90e⟩

// Reading characters
⟨function getchr(ed.c) 46d⟩
⟨function gety(ed.c) 47a⟩
⟨function gettty(ed.c) 46e⟩

// Reading and writing files
⟨function getfile(ed.c) 52b⟩
⟨function putfile(ed.c) 56⟩

⟨function append(ed.c) 51b⟩
⟨function add(ed.c) 60b⟩

⟨function browse(ed.c) 83d⟩

⟨function callunix(ed.c) 80c⟩

⟨function quit(ed.c) 44b⟩
⟨function onquit(ed.c) 94b⟩

// Delete
⟨function rdelete(ed.c) 60e⟩
⟨function gdelete(ed.c) 89a⟩

// Get/Put lines
⟨function getline_opti(ed.c) 88c⟩
⟨function putline_opti(ed.c) 85f⟩
⟨function blkio(ed.c) 88a⟩
⟨function getblock(ed.c) 86c⟩

⟨function init(ed.c) 43a⟩

⟨function global(ed.c) 78⟩

⟨function join(ed.c) 83a⟩

// Get/Put lines simplified versions (not in original ed.c)
⟨function getline 57a⟩
⟨function putline 53b⟩

// Search and replace
⟨function substitute(ed.c) 71d⟩
⟨function compsub(ed.c) 70h⟩
⟨function getsub(ed.c) 76b⟩
⟨function dosub(ed.c) 72d⟩
⟨function place(ed.c) 74b⟩

⟨function move(ed.c) 61d⟩
⟨function reverse(ed.c) 62b⟩
```

```
  ⟨function getcopy(ed.c) 62a⟩

  ⟨function compile(ed.c) 68c⟩
  ⟨function match(ed.c) 69d⟩

  // Printing text
  ⟨function putd(ex.c) 55b⟩
  ⟨function putstr(ed.c) 44d⟩
  ⟨function putshst(ed.c) 45a⟩
  ⟨function putchr(ed.c) 45b⟩

  ⟨function mktemp(ed.c) 42e⟩

  ⟨function regerror(ed.c) 94a⟩
```

Uses opti 91b.

⟨*function* regerror(*ed.c*) 94a⟩≡ (91b)

```
  void
  regerror(char *s)
  {
      USED(s);
      error(Q);
  }
```

Uses Q 44c and error() 89d.

⟨*function* onquit(*ed.c*) 94b⟩≡ (91b)

```
  void
  onquit(int sig)
  {
      USED(sig);
      quit();
  }
```

Uses quit() 44b.

C.15 Index

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

add(): 59d, 60a, 60b
addr1: 48b, 50d, 51a, 52a, 55d, 56, 58c, 58e, 60b, 60d, 61a, 61d, 62a, 64d, 64f, 71d, 78, 81e, 83a, 83d, 91a
addr2: 48b, 49a, 50d, 51a, 52a, 55d, 56, 58c, 58e, 59c, 60b, 60d, 61a, 61d, 62a, 64f, 71d, 75d, 76a, 76c, 78, 81e, 82f, 83a, 83d, 84a, 91a
address(): 61d, 64d, 65a
anymarks: 76c, 76e, 76f, 77b
append(): 49a, 51b, 60b, 61a, 61d, 76a
bcons: 40f, 40g, 46d
blkio(): 86c, 88a
BLKSIZE-7: 53b, 85d, 85f, 86c, 88a, 88c
bpagesize: 83c, 83c, 83d
browse(): 83b, 83d
callunix(): 80a, 80c
col: 40e, 44d, 45a, 81b
commands(): 40g, 48a, 78
compile(): 67, 68c, 70h, 78

compsub(): 70h, 71d
count: 40d, 49c, 52b, 55b, 56, 59c, 82e, 90b
dol: 39, 43a, 44b, 49a, 50d, 51a, 51b, 52a, 55d, 60b, 60e, 61d, 64d, 64f, 65a, 65b, 66b, 67, 77a, 78, 83d, 89a, 91a
dosub(): 71d, 72d
dot: 39, 43a, 51b, 52a, 58c, 58e, 60e, 61d, 64e, 64f, 65a, 75d, 78, 81e, 83a, 89a
EOF-9: 46d, 46e, 47a, 48c, 49c, 50b, 52c, 58b, 62a, 76b, 78, 80c, 90b, 90c
error(): 44b, 47a, 48a, 49b, 49c, 50a, 50b, 51a, 52a, 53a, 53b, 55a, 55d, 56, 58b, 59b, 61d, 64d, 65a, 66b, 67, 68c, 69a, 70h, 71a, 71d, 72d, 74b, 75a, 75d, 76c, 77a, 78, 83a, 84a, 86c, 88a, 89d, 94a
error_1(): 43a, 89d, 90a, 90e
ESCFLG-5: 70g, 74c, 75a
ESIZE-3: 68b
exfile(): 49a, 55a, 55d
fchange: 44a, 44b, 49a, 53b, 55d, 57d, 60c, 60e, 61d, 84a, 85f, 89a, 91a
file: 48d, 49a, 49b, 49c, 50b, 50c, 55d, 85a
filename(): 49a, 49c, 55d, 59a, 84a
FNSIZE-1: 38b, 38c, 48d
GBSIZE-6: 77g, 78
gdelete(): 88d, 89a
genbuf: 72c, 72d, 74b, 83a
getblock(): 85f, 86c, 88c
getchr(): 46d, 47a, 49c, 50a, 57c, 58b, 64d, 65a, 66a, 68c, 70h, 74c, 75e, 76c, 77a, 78, 80c, 82a, 83d, 90b
getcopy(): 61d, 62a
getfile(): 49a, 52b
getline(): 56, 57a, 58c, 62a, 69d, 83a
getline_opti(): 88b, 88c
getnum(): 65a, 66a, 75f, 83d
getsub(): 76a, 76b
gettty(): 46e, 60b, 61a
gety(): 46e, 47a
given: 50d, 59b, 60b, 64a, 64f, 82f
global(): 77e, 77f, 78
globp: 40g, 42c, 46c, 46d, 71b, 71c, 78, 90b
hex: 81b, 81c, 81c
iblock: 86a, 86b, 86c
ichanged: 86a, 86b, 86c
init(): 40g, 43a, 84a
io: 48e, 49a, 49b, 55a, 55d, 85a, 90b, 91a
iobuf: 48e, 49a, 52b, 55a, 55d, 56, 91a
join(): 82f, 83a
lastc: 46b, 46d, 49b, 53a, 86c, 90b, 90e
LBSIZE-2: 40b, 40c, 47a, 53a, 70f, 71a, 72c, 72d, 74b, 83a
line: 40a, 40a, 45b, 81b
linebp: 54, 76b, 85c, 85f
linebuf: 40c, 46e, 47a, 52b, 52c, 53a, 53b, 57a, 69d, 72d, 76b, 83a, 85f, 88c
linp: 40a, 40a, 45b, 81b
listf: 80d, 81a, 81b, 82a, 82b, 83d, 90b
listn: 82a, 82b, 82c, 82d, 82e, 83d, 90b
loc1: 70b, 70c, 70d, 72d, 74a

loc2: 70b, 70c, 70d, 70e, 71d, 72d, 74a
main-10(): 40g
match(): 67, 69d, 71d, 78
MAXSUB-4: 69b, 69c, 69d, 75a
mktemp(): 40g, 42e
move(): 61b, 61c, 61d
names: 76c, 76d, 77a, 77b, 77d
NBLK-8: 53b, 86c, 87
newline(): 58a, 58b, 59c, 60b, 60c, 60d, 61a, 61d, 70h, 75d, 75e, 76c, 82d, 82f, 83d
nlall: 38d, 38d, 40g, 52a
nleft: 85b, 85f, 86c, 88c
nonzero(): 58c, 58d, 60d, 61a, 61d, 71c, 75d, 76c, 83a, 83b
notifyf(): 40g, 90e
oblock: 86a, 86b, 86c
oflag: 42a, 42b, 42c, 45b
onquit(): 94b
opti: 85e, 88b, 91b, 91b
pattern: 68a, 68c, 69d, 69e
peekc: 46a, 46d, 47c, 57c, 65a, 65b, 66a, 68c, 71b, 75e, 83d, 90b
pflag: 71b, 81d, 81e, 82a, 82b, 83d, 90b
place(): 74a, 74b, 75a
printcom(): 58a, 58c, 58e, 81e, 82d, 83d
putchr(): 44d, 45a, 45b, 55a, 55b, 59c, 82e, 90a, 90e
putd(): 55a, 55b, 55b, 59c, 82e
putfile(): 55d, 56, 91a
putline(): 51b, 53b, 71d, 83a
putline_opti(): 85e, 85f
putshst(): 45a, 58c
putst(): 44d, 52c, 59a, 80c, 90a
Q: 44b, 44c, 44c, 47a, 48a, 49c, 50a, 50b, 51a, 53a, 55a, 56, 58b, 59b, 61d, 64d, 65a, 66b, 67, 68c, 69a, 70h, 71a, 71d, 72d, 74b, 75a, 75d, 76c, 77a, 78, 83a, 84a, 90e, 94a
quit(): 40g, 44b, 57d, 60c, 91a, 94b
rdelete(): 60d, 60e, 61a, 83a
regerror(): 94a
rescue(): 52a, 90e, 91a
rescuing: 90d, 90e, 91a
reverse(): 61d, 62b
rhsbuf: 70f, 70h, 71a, 72d
savedfile: 38c, 40g, 42c, 50b, 50c, 59a
savej: 89b, 89c, 89d, 90e
setnoaddr(): 59a, 59b, 60c, 80c, 84a
setwide(): 49a, 50d, 55d, 59c, 78
squeeze(): 49a, 51a, 55d, 58d, 59c, 60b, 78
subexp: 69c, 69d, 70c, 70e, 75a
subnewa: 71d, 72a, 72b, 75d, 77b
subolda: 75b, 75c, 75d
substitute(): 71c, 71d
T: 43a, 43c, 43c, 53b, 86c, 88a
template: 40g, 42d, 42d

tfile: 37b, 37b, 43a, 53b, 57a, 88a
tfname: 37a, 40g, 43a, 44b
tline: 38a, 43b, 53b, 85f
vflag: 41a, 41a, 41b, 42b, 44b, 55a, 80c, 84a
waiting: 80b, 80c, 90e
wrapp: 55c, 55d, 90b
zero: 38d, 40g, 43a, 44b, 49a, 50d, 51a, 52a, 55d, 59c, 60b, 61d, 64d, 65a, 65b, 67, 70a, 76a, 77a, 78, 82e, 83d, 84a, 89a, 91a
__anon_enum_1: 91b

Bibliography

[Ben88] Jon Bentley. *More Programming Pearls*. Addison-Wesley, 1988. cited page(s) 19

[BO10] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2010. cited page(s) 6

[Com84] Douglas Comer. *Operating System Design: The XINU Approach*. Prentice Hall, 1984. cited page(s) 5, 9, 23

[Com87] Douglas Comer. *Operating System Design: Volume II Internetworking with XINU*. Prentice Hall, 1987. cited page(s) 23

[Dar14] Lewis Dartnell. *The Knowledge: How to Rebuild Our World from Scratch*. Penguin Press, 2014. cited page(s) 25

[EucBC] Euclid. *The Elements*. 300 BC. cited page(s) 9

[FH95] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995. cited page(s) 5

[Int86] Intel. *Intel 80386 Programmer's Reference Manual*. Intel Corporation, 1986. cited page(s) 7

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume 1, 2, 3*. Addison-Wesley, 1973. cited page(s) 9, 23

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 5, 9

[Knu99] Donald E. Knuth. *MMIXWare, a RISC Computer for the Third Millenium*. Springer, 1999. cited page(s) 23

[KP84] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall, 1984. cited page(s) 11, 35

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 9, 11

[Lio77] John Lions. *Lions Commentary on UNIX 6th Edition, with Source Code*. Peer to Peer Communications, 1977. cited page(s) 23

[NS05] Noam Nisan and Shimon Shocken. *The Elements of Computing Systems*. MIT Press, 2005. cited page(s) 9, 23, 26

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 10, 90

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 41

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 36, 40, 46, 66, 68, 89

[Pad18] Yoann Padioleau. *Principia Softwarica: The Plan 9 Shell rc*. 2018. cited page(s) 36

[Pad25] Yoann Padioleau. *Principia Softwarica: The Plan 9 Utilities*. 2025. cited page(s) 42

[PPD+95] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs. In *Computing Systems*, pages 221–254, 1995. Also available at `docs/articles/9.ps`. cited page(s) 11

[PPT+93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. In *Operating Systems Review*, pages 72–76, 1993. Also available at `docs/articles/names.ps`. cited page(s) 11

[Sea01] David Seal. *ARM, Architecture Reference Manual*. Addison-Wesley, 2001. cited page(s) 7, 9

[Sei09] Peter Seibel. *Coders at Work, Reflections on the Craft of Programming*. Apress, 2009. cited page(s) 18

[Tan87] Andrew S. Tanenbaum. *Operating Systems Design and Implementation*. Prentice Hall, 1987. cited page(s) 5, 9, 23

[TML+79] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. Alto: A personal computer. Technical report, Xerox PARC, 1979. CSL-79-11. cited page(s) 19

[WG92] Niklaus Wirth and Jurg Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, 1992. cited page(s) 23

[WR13] Alfred N. Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1910, 1912, 1913. cited page(s) 5, 9