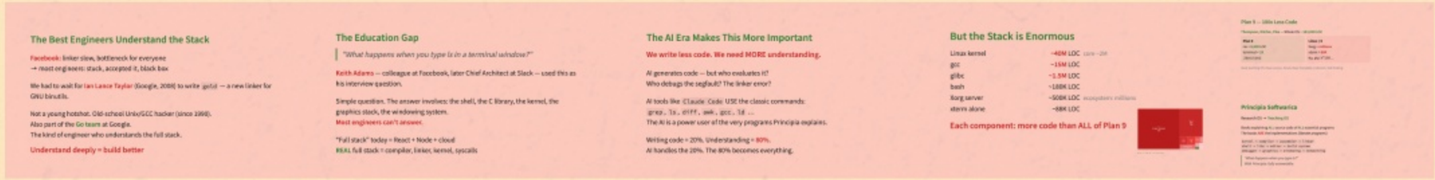
Principia Softwarica

Plan 9 Code Explained

Yoann Padioleau **IWP9**



A Series of Books

Category	Book	Program(s)	LOC	LOE	Pages
Core system	Kernel	9pi	35000	3200	732
	Core libraries	libc libregexp libthread	19000	1600	438
	Shell	rc	6500	1700	166
Development toolchain	C compiler	5c libcc	18500	1900	471
	Assembler	5a	3600	4400	176
	Linker	5l	7500	5400	296
Developer tools	Editor	ed	1600	200	45
	Build system	mk	4350	4050	197
	Debuggers	db acid strace libmach	13100	1000	321
	Profilers	prof time kprof stats iostats	3900	350	102
Graphics	Graphics stack	/dev/draw libdraw libmemdraw libmemlayer	18500	3400	507
	Windowing system	rio libframe libcomplete libplumb	8800	4000	289
Networking	Network stack	/dev/net libip lib9p	18300	4800	457
Misc	CLI utilities	cat ls grep sed diff tar gzip ...	23900	650	493
Total:			182550	36650	4690

Table 1: Principia Softwarica Books and Their Statistics.
(**LOC** = lines of code; **LOE** = lines of explanation; **Pages** = number of typeset pages)

A New Plan 9 Fork

```
.
|-- assemblers/
|   |-- 5a/
|   |-- 8a/
|   |-- data2s.c
|   |-- Assembler.nw
|   `-- mkfile
|-- builders/
|   |-- Builder.nw
|   |-- mk/
|   `-- mkfile
|-- compilers/
|   |-- 5c/
|   |-- 8c/
|   |-- cc/
|   |-- Compiler.nw
|   |-- cpp/
|   `-- mkfile
|-- debuggers/
|   |-- acid/
|   |-- db/
|   |-- Debugger.nw
|   |-- libmach/
|   `-- mkfile
|-- Dockerfile
|-- docs/
|   |-- articles/
|   `-- iwp9/
|-- dosdisk.img
|-- editors/
|   |-- ed/
|   `-- mkfile
|-- include/
|   |-- libc.h
|   |-- ...
|   `-- u.h
|-- linkers/
|   |-- 5l/
|   |-- 8l/
|   |-- Linker.nw
|   `-- mkfile
|-- readme.txt
`-- windows/
    |-- libcomplete/
    |-- libplumb/
    |-- mkfile
    |-- rio/
    `-- Windows.nw

|-- kernel/
|   |-- arch/
|   |-- concurrency/
|   |-- console/
|   |-- devices/
|   |-- files/
|   |-- filesystems/
|   |-- init/
|   |-- memory/
|   |-- Kernel.nw
|   |-- mkfile
|   |-- network/
|   |-- processes/
|   `-- syscalls/
|-- lib_core/
|   |-- libbio/
|   |-- libc/
|   |-- Libcore.nw
|   |-- libthread/
|   `-- mkfile
|-- lib_graphics/
|   |-- Graphics.nw
|   |-- libdraw/
|   |-- libimg/
|   |-- libmemdraw/
|   |-- libmemlayer/
|   `-- mkfile
|-- lib_networking/
|   |-- lib9p/
|   |-- libip/
|   `-- mkfile
|-- lib_strings/
|   |-- libflate/
|   |-- libregexp/
|   |-- libstring/
|   `-- mkfile
|-- Makefile
|-- mkconfig.pc
|-- mkconfig.pi
|-- mkfile
|-- mkfile-host-Cygwin
|-- mkfile-host-Linux
|-- mkfile-host-macOS
|-- mkfile-host-Plan9
`-- MISC/
    |-- pc/
    `-- pi/

|-- mkfiles/
|   |-- 386/
|   |-- arm/
|   |-- mkdirs
|   |-- mkfile.proto
|   |-- mklib
|   `-- mkone
|-- mkfile-target-pc
|-- mkfile-target-pi
|-- networking/
|   |-- arp/
|   |-- dhcp/
|   |-- ftp/
|   |-- http/
|   |-- ip/
|   |-- mkfile
|   |-- ndb/
|   |-- Network.nw
|   |-- snoopy/
|   `-- telnet/
|-- profilers/
|   |-- iostats/
|   |-- mkfile
|   `-- Profiler.nw
|-- shells/
|   |-- mkfile
|   |-- rc/
|   `-- Shell.nw
|-- utilities/
|   |-- archive/
|   |-- byte/
|   |-- calc/
|   |-- compare/
|   |-- files/
|   |-- mkfile
|   |-- pipe/
|   |-- process/
|   |-- text/
|   |-- time/
|   `-- Utilities.nw
`-- ROOT/
    |-- 386/
    |-- arm/
    |-- lib/
    |-- rc/
    |-- tests/
    `-- usr/
```

Figure 1: https://github.com/aryx/principia-softwarica Layout.
(as generated by `tree -L 2 -F` and spread in 3 columns)

A New Way to Build

Goken `Dockerfile` excerpt	Principia Softwarica `Dockerfile`

`# Build goken on Ubuntu using`
`# gcc/binutils (and mk/rc)`
`FROM ubuntu:24.04`
 	
`# Setup a basic C dev environment`
`RUN apt-get install -y gcc libc6-dev \`
`    byacc`
 	
`# Now let's build from source`
`WORKDIR /src`
`COPY . .`
 	
`# Small shell script (not GNU autoconf)`
`# to detect arch and generate`
`# ./mkconfig`
`RUN ./configure`
 	
`# The script below obviously builds`
`# 'mk' (without needing mk) but also:`
`# - 'rc', which is called by 'mk'`
`# - 'ed', for the mkenam script`
`#   run during the build`
`RUN ./scripts/build-mk.sh`
 	
`# copy ./ROOT/<arch>/bin/{mk,rc,ed}`
`# to ./bin/`
`RUN ./scripts/promote-mk.sh`
 	
`# make mk and rc accessible`
`ENV PATH="/src/bin:${PATH}"`
`ENV MKSHELL="/src/bin/rc"`
 	
`# Let's build goken (using mk/rc built`
`# in the previous step)`
`RUN mk`
`RUN mk install`
 	
`ENV PATH="/src/ROOT/amd64/bin:\`
`         /src/ROOT/arm64/bin:\`
`         ${PATH}"`
	`# Build principia on Ubuntu`
	`# Linux (amd64 or arm64)`
	`# for 386 (pc) and arm (pi)`
	`FROM padator/goken`
	
	`WORKDIR /principia`
	`COPY . .`
	
	`# 386`
	`RUN cp mkconfig.pc mkconfig`
	`RUN mk && mk install`
	`RUN mk kernel`
	`# arm`
	`RUN cp mkconfig.pi mkconfig`
	`RUN mk && mk install`
	`RUN mk kernel`
	
	`# VFAT tools`
	`RUN apt-get install -y dosfstools \`
	`      mtools`
	
	`# making dosdisk.img`
	`RUN dd if=/dev/zero of=tmp.img \`
	`       bs=1M count=512`
	`RUN mkfs.vfat tmp.img`
	`RUN mcopy -i tmp.img -s -o \`
	`      ROOT/* ::`
	`RUN cat MISC/pc/bootsector tmp.img \`
	`     > dosdisk.img`
	`RUN rm -f tmp.img`

Table 3: Principia Softwarica Build and Packaging Process.

The Best Engineers Understand the Stack

Facebook: linker slow, bottleneck for everyone

→ most engineers: stuck, accepted it, black box

We had to wait for **Ian Lance Taylor** (Google, 2008) to write `gold` — a new linker for GNU binutils.

Not a young hotshot. Old-school Unix/GCC hacker (since 1990).

Also part of the **Go team** at Google.

The kind of engineer who understands the full stack.

Understand deeply = build better

The Education Gap

> *"What happens when you type ls in a terminal window?"*

Keith Adams — colleague at Facebook, later Chief Architect at Slack — used this as his interview question.

Simple question. The answer involves: the shell, the C library, the kernel, the graphics stack, the windowing system.
Most engineers can't answer.

“Full stack” today = React + Node + cloud
REAL full stack = compiler, linker, kernel, syscalls

The AI Era Makes This More Important

We write less code. We need MORE understanding.

AI generates code — but who evaluates it?
Who debugs the segfault? The linker error?

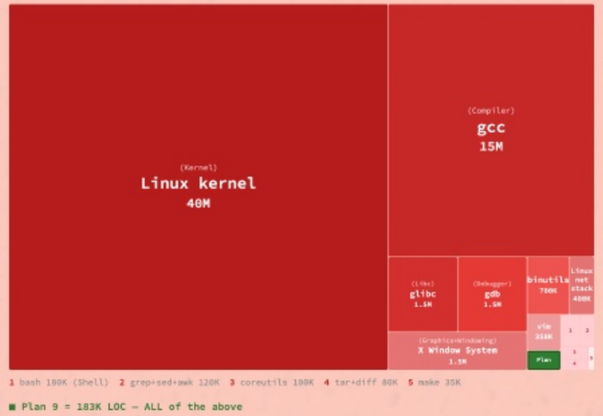
AI tools like `Claude Code` USE the classic commands:
`grep`, `ls`, `diff`, `awk`, `gcc`, `ld` ...
The AI is a power user of the very programs Principia explains.

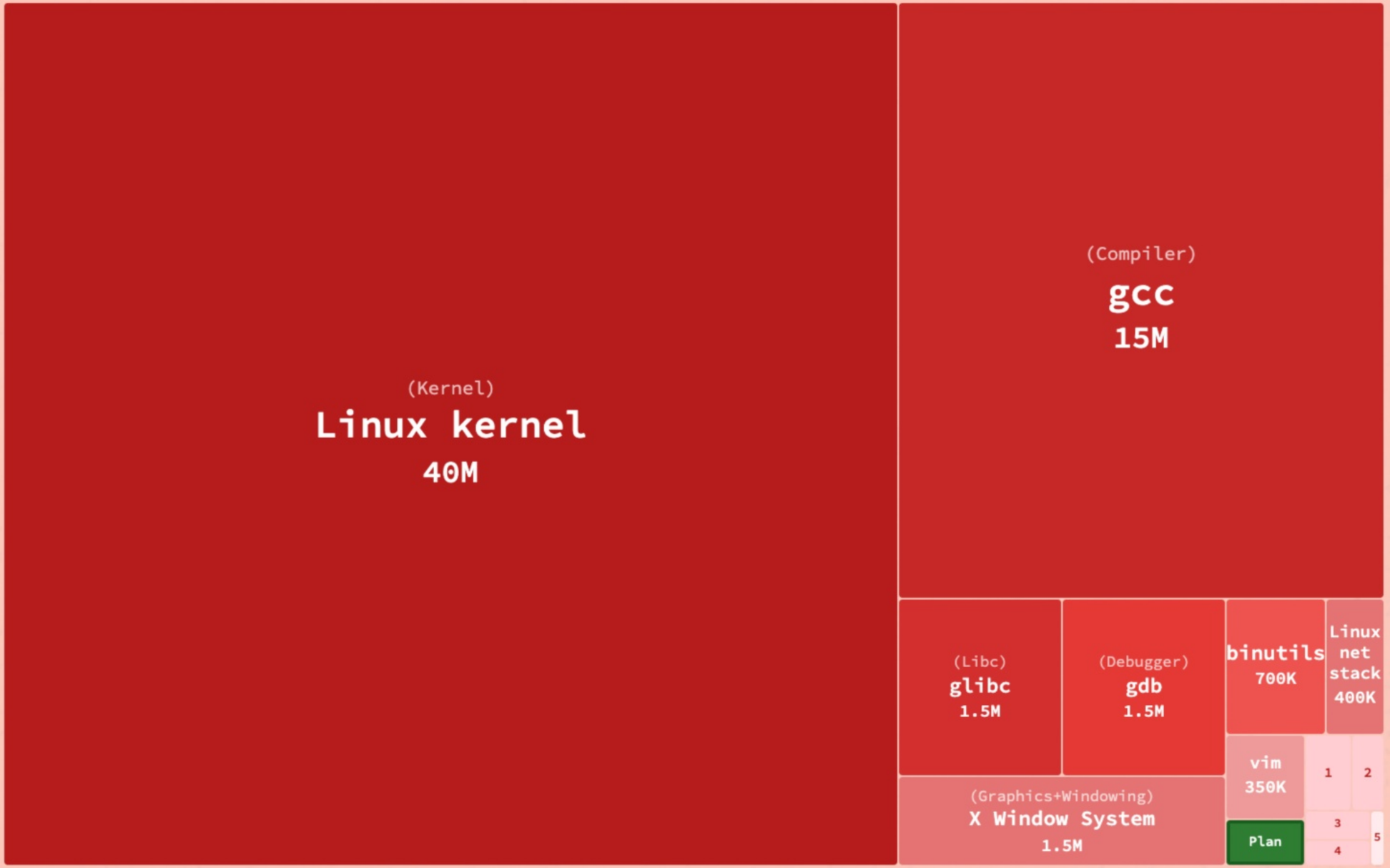
Writing code = 20%. Understanding = **80%**.
AI handles the 20%. The 80% becomes everything.

But the Stack is Enormous

Linux kernel	**~40M** LOC	core ~2M
gcc	**~15M** LOC	
glibc	**~1.5M** LOC	
bash	~180K LOC	
Xorg server	~500K LOC	ecosystem: millions
xterm alone	~88K LOC	

Each component: more code than ALL of Plan 9





1 bash 180K (Shell) 2 grep+sed+awk 120K 3 coreutils 100K 4 tar+diff 80K 5 make 35K

■ **Plan 9 = 183K LOC — ALL of the above**

Plan 9 — 100x Less Code

Thompson, Ritchie, Pike — Whole OS: **~183,000 LOC**

Plan 9

rio = **8,800** LOC

terminal < **1K**

`/dev/cons`

Linux / X

Xorg = **millions**

xterm = **88K**

tty, pty, VT100…

Ideal teaching OS: Open source, Small, Real, Complete, Coherent, Self-hosting

Principia Softwarica

Research OS → **Teaching OS**

Books explaining **ALL** source code of **ALL** essential programs

The books **ARE** the implementations (literate programs)

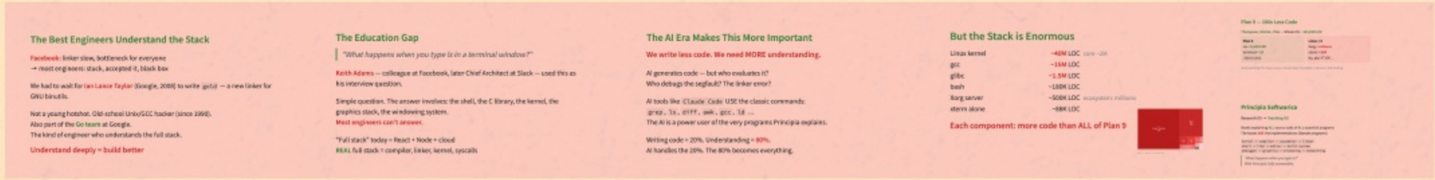
```
kernel → compiler → assembler → linker
shell → libc → editor → build system
debugger → graphics → windowing → networking
```

> *"What happens when you type ls?"*
>
> *With Principia: fully answerable.*

Principia Softwarica

Plan 9 Code Explained

Yoann Padioleau **IWP9**



A Series of Books

Category	Book	Program(s)	LOC	LOE	Pages
Core system	Kernel	9pi	35000	3200	732
	Core libraries	libc libregexp libthread	19000	1600	438
	Shell	rc	6500	1700	166
Development toolchain	C compiler	5c libcc	18500	1900	471
	Assembler	5a	3600	4400	176
	Linker	5l	7500	5400	296
Developer tools	Editor	ed	1600	200	45
	Build system	mk	4350	4050	197
	Debuggers	db acid strace libmach	13100	1000	321
	Profilers	prof time kprof stats iostats	3900	350	102
Graphics	Graphics stack	/dev/draw libdraw libmemdraw libmemlayer	18500	3400	507
	Windowing system	rio libframe libcomplete libplumb	8800	4000	289
Networking	Network stack	/dev/net libip lib9p	18300	4800	457
Misc	CLI utilities	cat ls grep sed diff tar gzip ...	23900	650	493
Total:			182550	36650	4690

Table 1: Principia Softwarica Books and Their Statistics.
(**LOC** = lines of code; **LOE** = lines of explanation; **Pages** = number of typeset pages)

A New Plan 9 Fork

```
.
|-- assemblers/
|   |-- 5a/
|   |-- 8a/
|   |-- data2s.c
|   |-- Assembler.nw
|   `-- mkfile
|-- builders/
|   |-- Builder.nw
|   |-- mk/
|   `-- mkfile
|-- compilers/
|   |-- 5c/
|   |-- 8c/
|   |-- cc/
|   |-- Compiler.nw
|   |-- cpp/
|   `-- mkfile
|-- debuggers/
|   |-- acid/
|   |-- db/
|   |-- Debugger.nw
|   |-- libmach/
|   `-- mkfile
|-- Dockerfile
|-- docs/
|   |-- articles/
|   `-- iwp9/
|-- dosdisk.img
|-- editors/
|   |-- ed/
|   `-- mkfile
|-- include/
|   |-- libc.h
|   |-- ...
|   `-- u.h
|-- linkers/
|   |-- 5l/
|   |-- 8l/
|   |-- Linker.nw
|   `-- mkfile
|-- readme.txt
`-- windows/
    |-- libcomplete/
    |-- libplumb/
    |-- mkfile
    |-- rio/
    `-- Windows.nw

|-- kernel/
|   |-- arch/
|   |-- concurrency/
|   |-- console/
|   |-- devices/
|   |-- files/
|   |-- filesystems/
|   |-- init/
|   |-- memory/
|   |-- Kernel.nw
|   |-- mkfile
|   |-- network/
|   |-- processes/
|   `-- syscalls/
|-- lib_core/
|   |-- libbio/
|   |-- libc/
|   |-- Libcore.nw
|   |-- libthread/
|   `-- mkfile
|-- lib_graphics/
|   |-- Graphics.nw
|   |-- libdraw/
|   |-- libimg/
|   |-- libmemdraw/
|   |-- libmemlayer/
|   `-- mkfile
|-- lib_networking/
|   |-- lib9p/
|   |-- libip/
|   `-- mkfile
|-- lib_strings/
|   |-- libflate/
|   |-- libregexp/
|   |-- libstring/
|   `-- mkfile
|-- Makefile
|-- mkconfig.pc
|-- mkconfig.pi
|-- mkfile
|-- mkfile-host-Cygwin
|-- mkfile-host-Linux
|-- mkfile-host-macOS
|-- mkfile-host-Plan9
`-- MISC/
    |-- pc/
    `-- pi/

|-- mkfiles/
|   |-- 386/
|   |-- arm/
|   |-- mkdirs
|   |-- mkfile.proto
|   |-- mklib
|   `-- mkone
|-- mkfile-target-pc
|-- mkfile-target-pi
|-- networking/
|   |-- arp/
|   |-- dhcp/
|   |-- ftp/
|   |-- http/
|   |-- ip/
|   |-- mkfile
|   |-- ndb/
|   |-- Network.nw
|   |-- snoopy/
|   `-- telnet/
|-- profilers/
|   |-- iostats/
|   |-- mkfile
|   `-- Profiler.nw
|-- shells/
|   |-- mkfile
|   |-- rc/
|   `-- Shell.nw
|-- utilities/
|   |-- archive/
|   |-- byte/
|   |-- calc/
|   |-- compare/
|   |-- files/
|   |-- mkfile
|   |-- pipe/
|   |-- process/
|   |-- text/
|   |-- time/
|   `-- Utilities.nw
`-- ROOT/
    |-- 386/
    |-- arm/
    |-- lib/
    |-- rc/
    |-- tests/
    `-- usr/
```

Figure 1: https://github.com/aryx/principia-softwarica Layout.
(as generated by `tree -L 2 -F` and spread in 3 columns)

A New Way to Build

Goken `Dockerfile` excerpt	Principia Softwarica `Dockerfile`

`# Build goken on Ubuntu using`
`# gcc/binutils (and mk/rc)`
`FROM ubuntu:24.04`
 	
`# Setup a basic C dev environment`
`RUN apt-get install -y gcc libc6-dev \`
`    byacc`
 	
`# Now let's build from source`
`WORKDIR /src`
`COPY . .`
 	
`# Small shell script (not GNU autoconf)`
`# to detect arch and generate`
`# ./mkconfig`
`RUN ./configure`
 	
`# The script below obviously builds`
`# 'mk' (without needing mk) but also:`
`# - 'rc', which is called by 'mk'`
`# - 'ed', for the mkenam script`
`#   run during the build`
`RUN ./scripts/build-mk.sh`
 	
`# copy ./ROOT/<arch>/bin/{mk,rc,ed}`
`# to ./bin/`
`RUN ./scripts/promote-mk.sh`
 	
`# make mk and rc accessible`
`ENV PATH="/src/bin:${PATH}"`
`ENV MKSHELL="/src/bin/rc"`
 	
`# Let's build goken (using mk/rc built`
`# in the previous step)`
`RUN mk`
`RUN mk install`
 	
`ENV PATH="/src/ROOT/amd64/bin:\`
`         /src/ROOT/arm64/bin:\`
`         ${PATH}"`
	`# Build principia on Ubuntu`
	`# Linux (amd64 or arm64)`
	`# for 386 (pc) and arm (pi)`
	`FROM padator/goken`
	
	`WORKDIR /principia`
	`COPY . .`
	
	`# 386`
	`RUN cp mkconfig.pc mkconfig`
	`RUN mk && mk install`
	`RUN mk kernel`
	`# arm`
	`RUN cp mkconfig.pi mkconfig`
	`RUN mk && mk install`
	`RUN mk kernel`
	
	`# VFAT tools`
	`RUN apt-get install -y dosfstools \`
	`      mtools`
	
	`# making dosdisk.img`
	`RUN dd if=/dev/zero of=tmp.img \`
	`       bs=1M count=512`
	`RUN mkfs.vfat tmp.img`
	`RUN mcopy -i tmp.img -s -o \`
	`      ROOT/* ::`
	`RUN cat MISC/pc/bootsector tmp.img \`
	`     > dosdisk.img`
	`RUN rm -f tmp.img`

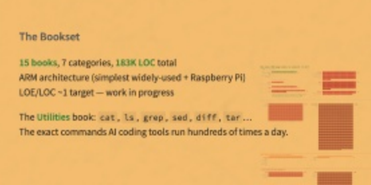
Table 3: Principia Softwarica Build and Packaging Process.

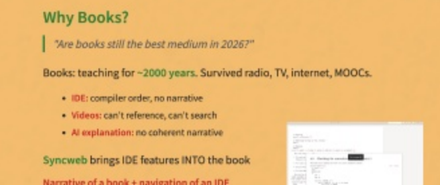
A Series of Books

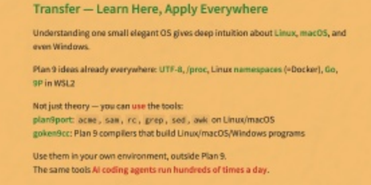
Category	Book	Program(s)	LOC	LOE	Pages
Core system	Kernel	`9pi`	35000	3200	732
	Core libraries	`libc`	19000	1600	438
		`libregexp`			
		`libthread`			
	Shell	`rc`	6500	1700	166
Development toolchain	C compiler	`5c libcc`	18500	1900	471
	Assembler	`5a`	3600	4400	176
	Linker	`5l`	7500	5400	296
Developer tools	Editor	`ed`	1600	200	45
	Build system	`mk`	4350	4050	197
	Debuggers	`db acid`	13100	1000	321
		`strace`			
		`libmach`			
	Profilers	`prof time`	3900	350	102
		`kprof stats`			

Graphics	Graphics stack	`/dev/draw` `libdraw` `libmemdraw` `libmemlayer`	18500	3400	507
	Windowing system	`rio` `libframe` `libcomplete` `libplumb`	8800	4000	289
Networking	Network stack	`/dev/net` `libip` `lib9p`	18300	4800	457
Misc	CLI utilities	`cat` `ls` `grep` `sed` `diff` `tar` `gzip` ...	23900	650	493
Total:			182550	36650	4690

Table 1: Principia Softwarica Books and Their Statistics.
(**LOC** = lines of code; **LOE** = lines of explanation; **Pages** = number of typeset pages)







The Bookset

15 books, 7 categories, **183K LOC** total

ARM architecture (simplest widely-used + Raspberry Pi)

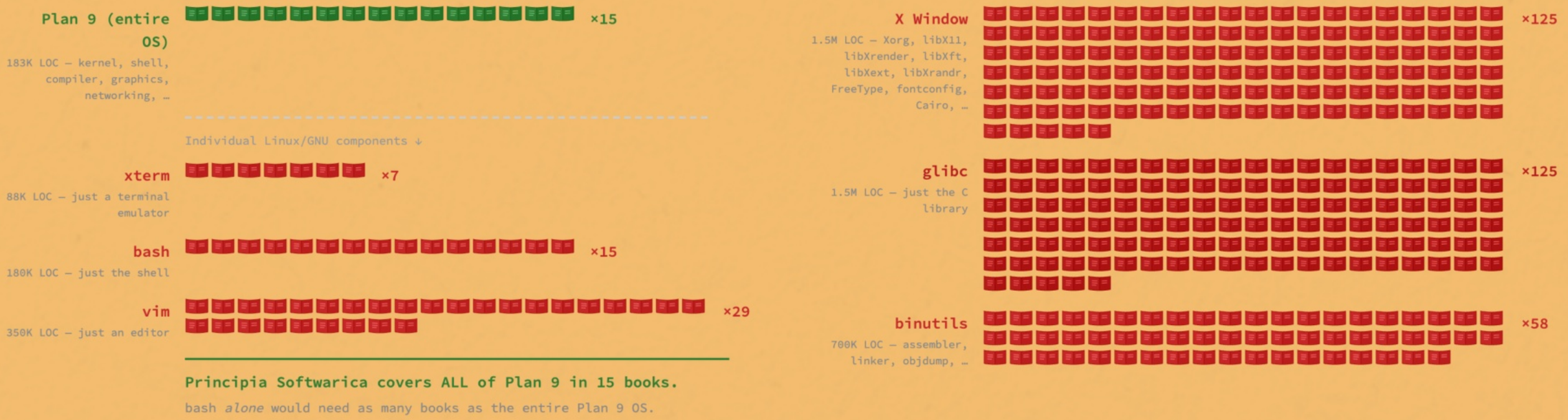
LOE/LOC ~1 target — work in progress

The **Utilities** book: `cat`, `ls`, `grep`, `sed`, `diff`, `tar`...

The exact commands AI coding tools run hundreds of times a day.

How many 400-page books to explain it all?

each 📖 = one 400-page book



Principia Softwarica covers ALL of Plan 9 in 15 books.

bash *alone* would need as many books as the entire Plan 9 OS.

---

same scale, zoomed out ↓

How many 400-page books to explain it all?

each 📘 = one 400-page book

Plan 9 (entire OS) — 183K LOC — kernel, shell, compiler, graphics, networking, … — ×15

Individual Linux/GNU components ↓

xterm — 88K LOC — just a terminal emulator — ×7

bash — 180K LOC — just the shell — ×15

vim — 350K LOC — just an editor — ×29

Principia Softwarica covers ALL of Plan 9 in 15 books.

bash *alone* would need as many books as the entire Plan 9 OS.

X Window — 1.5M LOC — Xorg, libX11, libXrender, libXft, libXext, libXrandr, FreeType, fontconfig, Cairo, … — ×125

glibc — 1.5M LOC — just the C library — ×125

binutils — 700K LOC — assembler, linker, objdump, … — ×58

same scale, zoomed out ↓

gcc — 15M LOC — the compiler — ×1,250

Linux kernel — 40M LOC — the kernel — ×3,333

es a day.

"Clean rewrites" — Wayland replaces X11, Clang replaces gcc

Wayland stack
1.24M LOC — Wayland, Mutter, libinput, FreeType, fontconfig, Cairo, HarfBuzz, pixman, libdrm

×101

Clang
830K LOC — C/C++/ObjC frontend

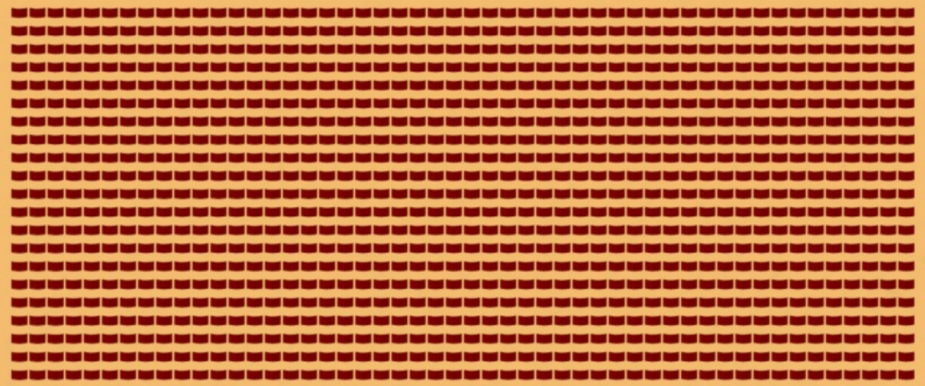
×68

… but they require these massive dependencies ↓

Mesa
4.3M LOC — OpenGL/Vulkan/EGL (required by Wayland compositors)

×350

LLVM
12.8M LOC — backend + optimizer (required by Clang)



×1,050

Literate Programming

Donald Knuth, 1984

Mix code + documentation

Present in order of **UNDERSTANDING**, not compiler order

Piece by piece, separate concerns

Tool: **Noweb**

```
<<chunk-name>>= defines a chunk <<chunk-name>> references a chunk Chunks defined
incrementally, across chapters
```

Literate programming 101

ToyKernel.nw excerpt

```
\subsection{Process}
The [[Proc]] structure is the
most important kernel structure.

<<proc.h>>=
struct Proc {
  int pid;
  Lock l;
  <<[[Proc]] other fields>>
}
@
\subsection{[[main()]]}
The kernel entry point is [[main()]]
which after some initializations calls
the most important function:
[[schedule()]].

<<kernel.c>>=
void main() {
  <<initializations>>
  schedule();
  // never reached
}
@
\subsection{Initializations}
This initializes all the globals:

<<initializations>>=
ginit();
@
Here are the remaining initializations:
<<initializations>>=
cinit();
vinit();
@

\subsection{Advanced features}
Some less important [[Proc]] fields:

<<[[Proc]] other fields>>=
int cnt;
@
```

LaTeX output

3.1 Process

The Proc structure is the most important kernel structure.

⟨proc.h 6a⟩≡
```
struct Proc {
  int pid;
  Lock l;
  ⟨Proc other fields 6e⟩
}
```

3.2 main()

The kernel entry point is main() which after some initializations calls the most important function: schedule().

⟨kernel.c 6b⟩≡
```
void main() {
  ⟨initializations 6c⟩
  schedule();
  // never reached
}
```

3.3 Initializations

This initializes all the globals:

⟨initializations 6c⟩≡
```
ginit();
```

Here are the remaining initializations:

⟨initializations 6d⟩+≡
```
cinit();
vinit();
```

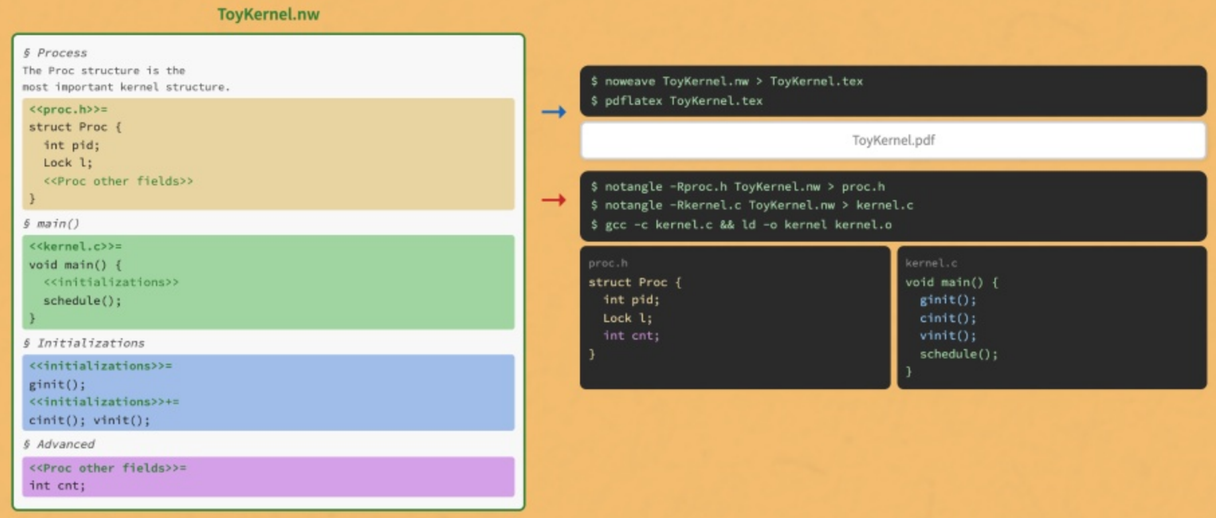
3.4 Advanced features

Some less important Proc fields:

⟨Proc other fields 6e⟩≡
```
int cnt;
```

Table 2: Noweb Source Example and Rendered Output.

Literate Programming in Practice



ToyKernel.nw excerpt

```
\subsection{Process}
The [[Proc]] structure is the
most important kernel structure.

<<proc.h>>=
struct Proc {
  int pid;
  Lock l;
  <<[[Proc]] other fields>>
}
@
\subsection{[[main()]]}
The kernel entry point is [[main()]]
which after some initializations calls
the most important function:
[[schedule()]].
```

LaTeX outpu

3.1 Proce

The Proc stru
nel structure.

⟨proc.h 6a⟩≡

```
struct Proc
  int pid;
  Lock l;
  ⟨Proc other
}
```

3.2 main(

The kernel en
some initializa
function: sche

ToyKernel.nw excerpt

```
\subsection{Process}
The [[Proc]] structure is the
most important kernel structure.

<<proc.h>>=
struct Proc {
  int pid;
  Lock l;
  <<[[Proc]] other fields>>
}
@
\subsection{[[main()]]}
The kernel entry point is [[main()]]
which after some initializations calls
the most important function:
[[schedule()]].

<<kernel.c>>=
void main() {
  <<initializations>>
  schedule();
  // never reached
}
```

LaTeX output

3.1 Process

The `Proc` structure is the most important kernel structure.

⟨`proc.h` 6a⟩≡

```
struct Proc {
  int pid;
  Lock l;
  ⟨Proc other fields 6e⟩
}
```

3.2 `main()`

The kernel entry point is `main()` which after some initializations calls the most important function: `schedule()`.

⟨`kernel.c` 6b⟩≡

```
void main() {
  ⟨initializations 6c⟩
  schedule();
  // never reached
}
```

```
<<kernel.c>>=
void main() {
  <<initializations>>
  schedule();
  // never reached
}
@
\subsection{Initializations}
This initializes all the globals:

<<initializations>>=
ginit();
@
Here are the remaining initializations:
<<initializations>>=
cinit();
vinit();
@

\subsection{Advanced features}
Some less important [[Proc]] fields:

<<[[Proc]] other fields>>=
int cnt;
@
```

⟨kernel.c 6b⟩≡
```
void main() {
  ⟨initializations 6c⟩
  schedule();
  // never reached
}
```

3.3 Initializations

This initializes all the globals:

⟨initializations 6c⟩≡
```
ginit();
```

Here are the remaining initializations:

⟨initializations 6d⟩+≡
```
cinit();
vinit();
```

3.4 Advanced features

Some less important `Proc` fields:

⟨`Proc` other fields 6e⟩≡
```
int cnt;
```

Literate Programming in Practice

ToyKernel.nw

§ Process

The Proc structure is the
most important kernel structure.

```
<<proc.h>>=
struct Proc {
  int pid;
  Lock l;
  <<Proc other fields>>
}
```

§ main()

```
<<kernel.c>>=
void main() {
  <<initializations>>
  schedule();
}
```

§ Initializations

```
<<initializations>>=
ginit();
<<initializations>>+=
cinit(); vinit();
```

§ Advanced

```
<<Proc other fields>>=
int cnt;
```

```
$ noweave ToyKernel.nw > ToyKernel.tex
$ pdflatex ToyKernel.tex
```

ToyKernel.pdf

```
$ notangle -Rproc.h ToyKernel.nw > proc.h
$ notangle -Rkernel.c ToyKernel.nw > kernel.c
$ gcc -c kernel.c && ld -o kernel kernel.o
```

proc.h

```
struct Proc {
  int pid;
  Lock l;
  int cnt;
}
```

kernel.c

```
void main() {
  ginit();
  cinit();
  vinit();
  schedule();
}
```

Why Books?

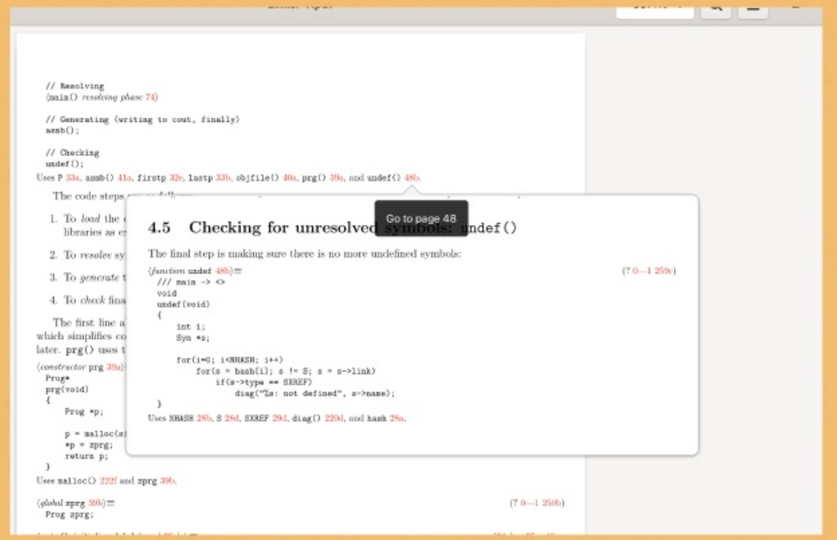
> *"Are books still the best medium in 2026?"*

Books: teaching for **~2000 years**. Survived radio, TV, internet, MOOCs.

- **IDE:** compiler order, no narrative
- **Videos:** can't reference, can't search
- **AI explanation:** no coherent narrative

Syncweb brings IDE features INTO the book

Narrative of a book + navigation of an IDE



```
// Resolving
⟨main() resolving phase 74⟩

// Generating (writing to cout, finally)
asmb();

// Checking
undef();
```

Uses P 33a, asmb() 41a, firstp 32e, lastp 33b, objfile() 40a, prg() 39a, and undef() 48b.

The code steps

1. To *load* the
   libraries as e
2. To *resolve* sy
3. To *generate* t
4. To *check* fina

The first line a
which simplifies co
later. prg() uses t

⟨*constructor* prg 39a⟩

```
Prog*
prg(void)
{
    Prog *p;

    p = malloc(si
    *p = zprg;
    return p;
}
```

Uses malloc() 222f and zprg 39b.

⟨*global* zprg 39b⟩≡ (? 0—1 250b)

```
Prog zprg;
```

Go to page 48

4.5 Checking for unresolved symbols: undef()

The final step is making sure there is no more undefined symbols:

⟨*function* undef 48b⟩≡ (? 0—1 259c)

```
/// main -> <>
void
undef(void)
{
    int i;
    Sym *s;

    for(i=0; i<NHASH; i++)
        for(s = hash[i]; s != S; s = s->link)
            if(s->type == SXREF)
                diag("%s: not defined", s->name);
}
```

Uses NHASH 28b, S 28d, SXREF 29d, diag() 220d, and hash 28a.

Transfer — Learn Here, Apply Everywhere

Understanding one small elegant OS gives deep intuition about **Linux**, **macOS**, and even Windows.

Plan 9 ideas already everywhere: **UTF-8**, **/proc**, Linux **namespaces** (=Docker), **Go**, **9P** in WSL2

Not just theory — you can **use** the tools:
plan9port: `acme`, `sam`, `rc`, `grep`, `sed`, `awk` on Linux/macOS
goken9cc: Plan 9 compilers that build Linux/macOS/Windows programs

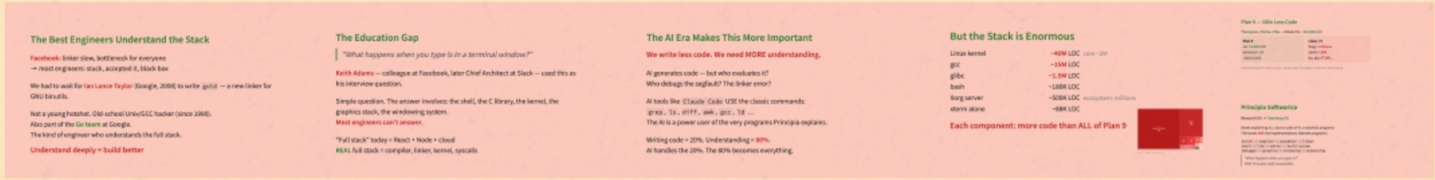
Use them in your own environment, outside Plan 9.
The same tools **AI coding agents run hundreds of times a day**.

Same concepts, 100x more clearly. And you can use them for real.

Principia Softwarica

Plan 9 Code Explained

Yoann Padioleau **IWP9**



A Series of Books

Category	Book	Program(s)	LOC	LOE	Pages
Core system	Kernel	9pi	35000	3200	732
	Core libraries	libc libregexp libthread	19000	1600	438
	Shell	rc	6500	1700	166
Development toolchain	C compiler	5c libcc	18500	1900	471
	Assembler	5a	3600	4400	176
	Linker	5l	7500	5400	296
Developer tools	Editor	ed	1600	200	45
	Build system	mk	4350	4050	197
	Debuggers	db acid strace libmach	13100	1000	321
	Profilers	prof time kprof stats iostats	3900	350	102
Graphics	Graphics stack	/dev/draw libdraw libmemdraw libmemlayer	18500	3400	507
	Windowing system	rio libframe libcomplete libplumb	8800	4000	289
Networking	Network stack	/dev/net libip lib9p	18300	4800	457
Misc	CLI utilities	cat ls grep sed diff tar gzip ...	23900	650	493
Total:			182550	36650	4690

Table 1: Principia Softwarica Books and Their Statistics.
(**LOC** = lines of code; **LOE** = lines of explanation; **Pages** = number of typeset pages)

A New Plan 9 Fork

```
.
|-- assemblers/
|   |-- 5a/
|   |-- 8a/
|   |-- data2s.c
|   |-- Assembler.nw
|   `-- mkfile
|-- builders/
|   |-- Builder.nw
|   |-- mk/
|   `-- mkfile
|-- compilers/
|   |-- 5c/
|   |-- 8c/
|   |-- cc/
|   |-- Compiler.nw
|   |-- cpp/
|   `-- mkfile
|-- debuggers/
|   |-- acid/
|   |-- db/
|   |-- Debugger.nw
|   |-- libmach/
|   `-- mkfile
|-- Dockerfile
|-- docs/
|   |-- articles/
|   `-- iwp9/
|-- dosdisk.img
|-- editors/
|   |-- ed/
|   `-- mkfile
|-- include/
|   |-- libc.h
|   |-- ...
|   `-- u.h
|-- linkers/
|   |-- 5l/
|   |-- 8l/
|   |-- Linker.nw
|   `-- mkfile
|-- readme.txt
`-- windows/
    |-- libcomplete/
    |-- libplumb/
    |-- mkfile
    |-- rio/
    `-- Windows.nw

|-- kernel/
|   |-- arch/
|   |-- concurrency/
|   |-- console/
|   |-- devices/
|   |-- files/
|   |-- filesystems/
|   |-- init/
|   |-- memory/
|   |-- Kernel.nw
|   |-- mkfile
|   |-- network/
|   |-- processes/
|   `-- syscalls/
|-- lib_core/
|   |-- libbio/
|   |-- libc/
|   |-- Libcore.nw
|   |-- libthread/
|   `-- mkfile
|-- lib_graphics/
|   |-- Graphics.nw
|   |-- libdraw/
|   |-- libimg/
|   |-- libmemdraw/
|   |-- libmemlayer/
|   `-- mkfile
|-- lib_networking/
|   |-- lib9p/
|   |-- libip/
|   `-- mkfile
|-- lib_strings/
|   |-- libflate/
|   |-- libregexp/
|   |-- libstring/
|   `-- mkfile
|-- Makefile
|-- mkconfig.pc
|-- mkconfig.pi
|-- mkfile
|-- mkfile-host-Cygwin
|-- mkfile-host-Linux
|-- mkfile-host-macOS
|-- mkfile-host-Plan9
`-- MISC/
    |-- pc/
    `-- pi/

|-- mkfiles/
|   |-- 386/
|   |-- arm/
|   |-- mkdirs
|   |-- mkfile.proto
|   |-- mklib
|   `-- mkone
|-- mkfile-target-pc
|-- mkfile-target-pi
|-- networking/
|   |-- arp/
|   |-- dhcp/
|   |-- ftp/
|   |-- http/
|   |-- ip/
|   |-- mkfile
|   |-- ndb/
|   |-- Network.nw
|   |-- snoopy/
|   `-- telnet/
|-- profilers/
|   |-- iostats/
|   |-- mkfile
|   `-- Profiler.nw
|-- shells/
|   |-- mkfile
|   |-- rc/
|   `-- Shell.nw
|-- utilities/
|   |-- archive/
|   |-- byte/
|   |-- calc/
|   |-- compare/
|   |-- files/
|   |-- mkfile
|   |-- pipe/
|   |-- process/
|   |-- text/
|   |-- time/
|   `-- Utilities.nw
`-- ROOT/
    |-- 386/
    |-- arm/
    |-- lib/
    |-- rc/
    |-- tests/
    `-- usr/
```

Figure 1: https://github.com/aryx/principia-softwarica Layout.
(as generated by `tree -L 2 -F` and spread in 3 columns)

A New Way to Build

Goken `Dockerfile` excerpt	Principia Softwarica `Dockerfile`

`# Build goken on Ubuntu using`
`# gcc/binutils (and mk/rc)`
`FROM ubuntu:24.04`
 	
`# Setup a basic C dev environment`
`RUN apt-get install -y gcc libc6-dev \`
`    byacc`
 	
`# Now let's build from source`
`WORKDIR /src`
`COPY . .`
 	
`# Small shell script (not GNU autoconf)`
`# to detect arch and generate`
`# ./mkconfig`
`RUN ./configure`
 	
`# The script below obviously builds`
`# 'mk' (without needing mk) but also:`
`# - 'rc', which is called by 'mk'`
`# - 'ed', for the mkenam script`
`#   run during the build`
`RUN ./scripts/build-mk.sh`
 	
`# copy ./ROOT/<arch>/bin/{mk,rc,ed}`
`# to ./bin/`
`RUN ./scripts/promote-mk.sh`
 	
`# make mk and rc accessible`
`ENV PATH="/src/bin:${PATH}"`
`ENV MKSHELL="/src/bin/rc"`
 	
`# Let's build goken (using mk/rc built`
`# in the previous step)`
`RUN mk`
`RUN mk install`
 	
`ENV PATH="/src/ROOT/amd64/bin:\`
`         /src/ROOT/arm64/bin:\`
`         ${PATH}"`
	`# Build principia on Ubuntu`
	`# Linux (amd64 or arm64)`
	`# for 386 (pc) and arm (pi)`
	`FROM padator/goken`
	
	`WORKDIR /principia`
	`COPY . .`
	
	`# 386`
	`RUN cp mkconfig.pc mkconfig`
	`RUN mk && mk install`
	`RUN mk kernel`
	`# arm`
	`RUN cp mkconfig.pi mkconfig`
	`RUN mk && mk install`
	`RUN mk kernel`
	
	`# VFAT tools`
	`RUN apt-get install -y dosfstools \`
	`      mtools`
	
	`# making dosdisk.img`
	`RUN dd if=/dev/zero of=tmp.img \`
	`       bs=1M count=512`
	`RUN mkfs.vfat tmp.img`
	`RUN mcopy -i tmp.img -s -o \`
	`      ROOT/* ::`
	`RUN cat MISC/pc/bootsector tmp.img \`
	`     > dosdisk.img`
	`RUN rm -f tmp.img`

Table 3: Principia Softwarica Build and Packaging Process.

A New Plan 9 Fork

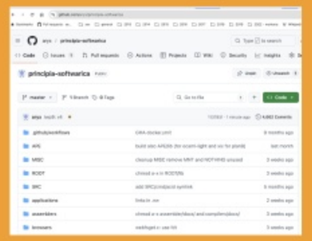
Why a New Fork?

Plan 9 = research OS. Principia = teaching OS. Different needs.

Across all books: introduced `globals.c`, intermediate `.h` files, renamed for clarity, removed dead code, fixed bugs

Result: code extensively reorganized → fork needed

https://github.com/aryx/principia-softwarica



```
.
|-- assemblers/
|   |-- 5a/
|   |-- 8a/
|   |-- data2s.c
|   |-- Assembler.nw
|   `-- mkfile
|-- builders/
|   |-- Builder.nw
|   |-- mk/
|   `-- mkfile
|-- compilers/
|   |-- 5c/
|   |-- 8c/
|   |-- cc/
|   |-- Compiler.nw
|   |-- cpp/
|   `-- mkfile
|-- debuggers/
|   |-- acid/
|   |-- db/
|   |-- Debugger.nw
|   |-- libmach/
|   `-- mkfile
|-- Dockerfile
|-- kernel/
|   |-- arch/
|   |-- concurrency/
|   |-- console/
|   |-- devices/
|   |-- files/
|   |-- filesystems/
|   |-- init/
|   |-- memory/
|   |-- Kernel.nw
|   |-- mkfile
|   |-- network/
|   |-- processes/
|   `-- syscalls/
|-- lib_core/
|   |-- libbio/
|   |-- libc/
|   |-- Libcore.nw
|   |-- libthread/
|   `-- mkfile
|-- lib_graphics/
|   |-- Graphics.nw
|   |-- libdraw/
|   |-- libimg/
|   |-- mkfiles/
|   |   |-- 386/
|   |   |-- arm/
|   |   |-- mkdirs
|   |   |-- mkfile.proto
|   |   |-- mklib
|   |   `-- mkone
|-- mkfile-target-pc
|-- mkfile-target-pi
|-- networking/
|   |-- arp/
|   |-- dhcp/
|   |-- ftp/
|   |-- http/
|   |-- ip/
|   |-- mkfile
|   |-- ndb/
|   |-- Network.nw
|   |-- snoopy/
|   `-- telnet/
|-- profilers/
|   |-- iostats/
|   |-- mkfile
|   `-- Profiler.nw
```

Repository Structure

Why a New Fork?

Plan 9 = research OS. Principia = **teaching OS**. Different needs.

Plan 9 4th edition

sys/src/9/port/

proc.c — everything in one file:

schedinit() sleep() procinit() sched() wakeup() pexit() ~~Qlock~~

page.c

devcons.c

~~dead code…~~

sys/src/9/pc/

trap.c

sys/src/9/bcm/

trap.c (same name, diff arch)

Flat files, mixed concerns, dead code

→

split
rename
clean
reorganize

Principia

Kernel.nw (one book, chapters by concern)

- **Ch. Processes:** procinit() pexit()
- **Ch. Concurrency:** sleep() wakeup()
- **Ch. Scheduling:** schedinit() sched()
- **Ch. Memory:** page functions
- **Ch. Interrupts:** trap() port + arch unified
- **Ch. Advanced:** complex details deferred

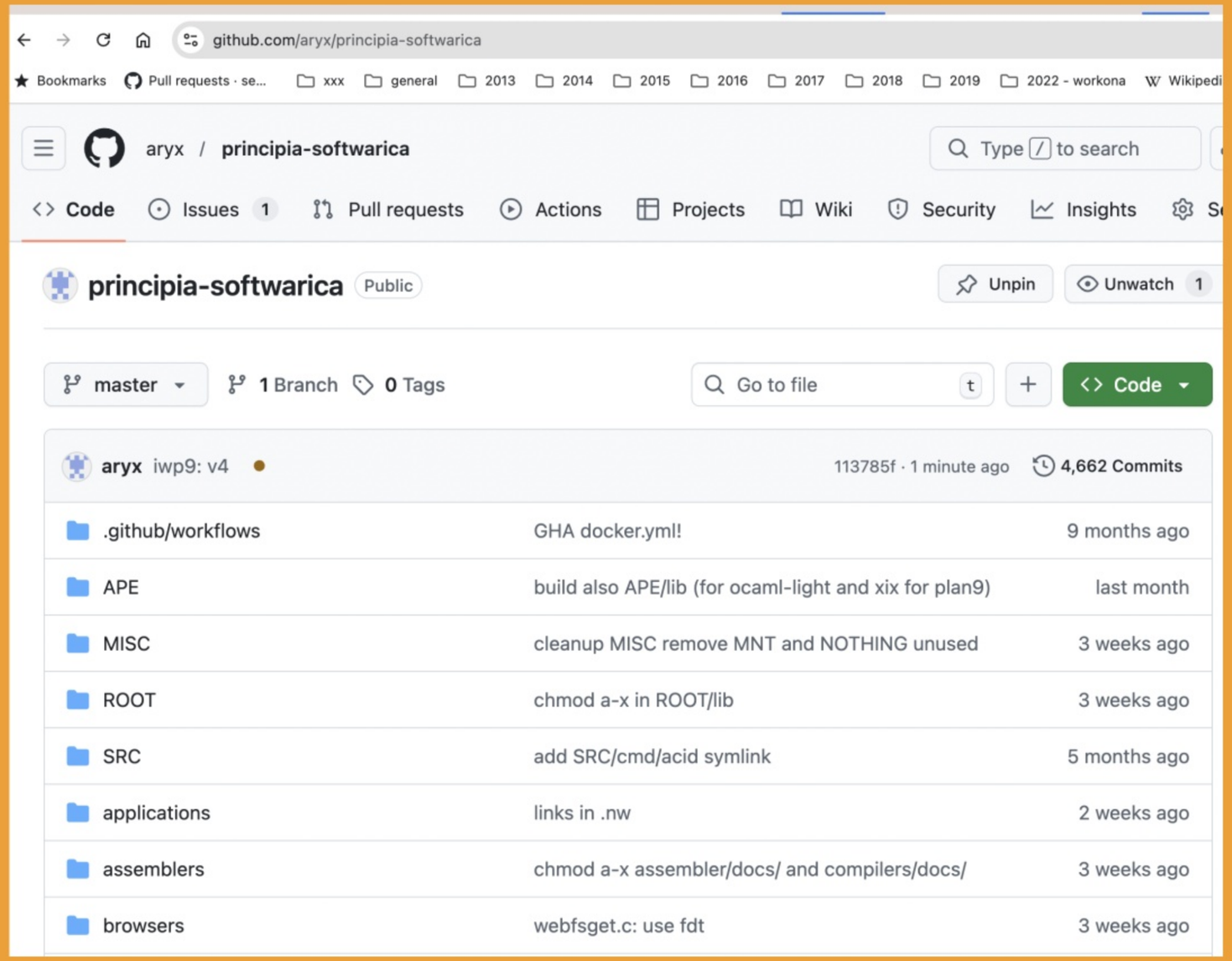
kernel/ processes/ proc.c sysproc.c · concurrency/ qlock.c taslock.c · memory/ page.c segment.c · interrupts/ · scheduling/ …

Files reorganized into topic dirs too

Across all books: introduced `globals.c`, intermediate `.h` files, renamed for clarity, removed dead code, fixed bugs

Result: code extensively reorganized → **fork needed**

https://github.com/aryx/principia-softwarica



```
.
|-- assemblers/
|   |-- 5a/
|   |-- 8a/
|   |-- data2s.c
|   |-- Assembler.nw
|   `-- mkfile
|-- builders/
|   |-- Builder.nw
|   |-- mk/
|   `-- mkfile
|-- compilers/
|   |-- 5c/
|   |-- 8c/
|   |-- cc/
|   |-- Compiler.nw
|   |-- cpp/
|   `-- mkfile
|-- debuggers/
|   |-- acid/
|   |-- db/
|   |-- Debugger.nw
|-- kernel/
|   |-- arch/
|   |-- concurrency/
|   |-- console/
|   |-- devices/
|   |-- files/
|   |-- filesystems/
|   |-- init/
|   |-- memory/
|   |-- Kernel.nw
|   |-- mkfile
|   |-- network/
|   |-- processes/
|   `-- syscalls/
|-- lib_core/
|   |-- libbio/
|   |-- libc/
|   |-- Libcore.nw
|   |-- libthread/
|   `-- mkfile
|-- lib_graphics/
|-- mkfiles/
|   |-- 386/
|   |-- arm/
|   |-- mkdirs
|   |-- mkfile.proto
|   |-- mklib
|   `-- mkone
|-- mkfile-target-pc
|-- mkfile-target-pi
|-- networking/
|   |-- arp/
|   |-- dhcp/
|   |-- ftp/
|   |-- http/
|   |-- ip/
|   |-- mkfile
|   |-- ndb/
|   |-- Network.nw
|   |-- snoopy/
|   `-- telnet/
|-- profilers/
```

Repository Structure

- `assemblers/5a/` not `sys/src/cmd/5a`
- Each dir: `.nw` literate program + generated code
- `kernel/`: `concurrency/` `memory/` `processes/` ...
- `mkconfig.pc` (x86/QEMU) + `mkconfig.pi` (ARM/RPi)

Good strategy to find bugs: try to explain the code in a book.

Syncweb: Bidirectional LP

```
Classic Noweb: .nw ──► code (one way) Syncweb: .nw ◄──► code (both ways)
```

Marks + md5 checksums → detect which side changed

Automatic merge or conflict detection

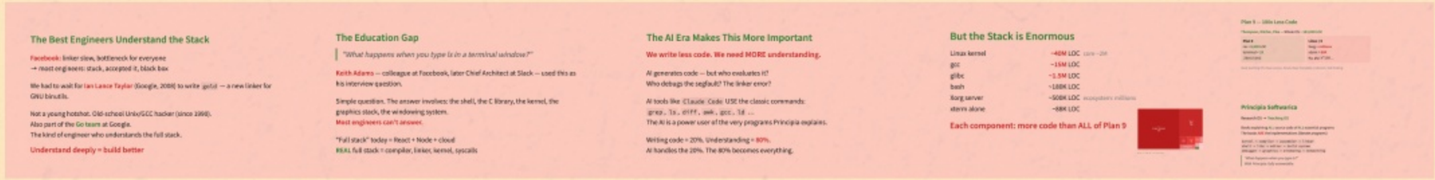
Solves biggest LP complaint

`github.com/aryx/syncweb`

Principia Softwarica

Plan 9 Code Explained

Yoann Padioleau **IWP9**



A Series of Books

Category	Book	Program(s)	LOC	LOE	Pages
Core system	Kernel	9pi	35000	3200	732
	Core libraries	libc libregexp libthread	19000	1600	438
	Shell	rc	6500	1700	166
Development toolchain	C compiler	5c libcc	18500	1900	471
	Assembler	5a	3600	4400	176
	Linker	5l	7500	5400	296
Developer tools	Editor	ed	1600	200	45
	Build system	mk	4350	4050	197
	Debuggers	db acid strace libmach	13100	1000	321
	Profilers	prof time kprof stats iostats	3900	350	102
Graphics	Graphics stack	/dev/draw libdraw libmemdraw libmemlayer	18500	3400	507
	Windowing system	rio libframe libcomplete libplumb	8800	4000	289
Networking	Network stack	/dev/net libip lib9p	18300	4800	457
Misc	CLI utilities	cat ls grep sed diff tar gzip ...	23900	650	493
Total:			182550	36650	4690

Table 1: Principia Softwarica Books and Their Statistics.
(**LOC** = lines of code; **LOE** = lines of explanation; **Pages** = number of typeset pages)

A New Plan 9 Fork

```
.
|-- assemblers/
|   |-- 5a/
|   |-- 8a/
|   |-- data2s.c
|   |-- Assembler.nw
|   `-- mkfile
|-- builders/
|   |-- Builder.nw
|   |-- mk/
|   `-- mkfile
|-- compilers/
|   |-- 5c/
|   |-- 8c/
|   |-- cc/
|   |-- Compiler.nw
|   |-- cpp/
|   `-- mkfile
|-- debuggers/
|   |-- acid/
|   |-- db/
|   |-- Debugger.nw
|   |-- libmach/
|   `-- mkfile
|-- Dockerfile
|-- docs/
|   |-- articles/
|   `-- iwp9/
|-- dosdisk.img
|-- editors/
|   |-- ed/
|   `-- mkfile
|-- include/
|   |-- libc.h
|   |-- ...
|   `-- u.h
|-- linkers/
|   |-- 5l/
|   |-- 8l/
|   |-- Linker.nw
|   `-- mkfile
|-- readme.txt
`-- windows/
    |-- libcomplete/
    |-- libplumb/
    |-- mkfile
    |-- rio/
    `-- Windows.nw

|-- kernel/
|   |-- arch/
|   |-- concurrency/
|   |-- console/
|   |-- devices/
|   |-- files/
|   |-- filesystems/
|   |-- init/
|   |-- memory/
|   |-- Kernel.nw
|   |-- mkfile
|   |-- network/
|   |-- processes/
|   `-- syscalls/
|-- lib_core/
|   |-- libbio/
|   |-- libc/
|   |-- Libcore.nw
|   |-- libthread/
|   `-- mkfile
|-- lib_graphics/
|   |-- Graphics.nw
|   |-- libdraw/
|   |-- libimg/
|   |-- libmemdraw/
|   |-- libmemlayer/
|   `-- mkfile
|-- lib_networking/
|   |-- lib9p/
|   |-- libip/
|   `-- mkfile
|-- lib_strings/
|   |-- libflate/
|   |-- libregexp/
|   |-- libstring/
|   `-- mkfile
|-- Makefile
|-- mkconfig.pc
|-- mkconfig.pi
|-- mkfile
|-- mkfile-host-Cygwin
|-- mkfile-host-Linux
|-- mkfile-host-macOS
|-- mkfile-host-Plan9
`-- MISC/
    |-- pc/
    `-- pi/

|-- mkfiles/
|   |-- 386/
|   |-- arm/
|   |-- mkdirs
|   |-- mkfile.proto
|   |-- mklib
|   `-- mkone
|-- mkfile-target-pc
|-- mkfile-target-pi
|-- networking/
|   |-- arp/
|   |-- dhcp/
|   |-- ftp/
|   |-- http/
|   |-- ip/
|   |-- mkfile
|   |-- ndb/
|   |-- Network.nw
|   |-- snoopy/
|   `-- telnet/
|-- profilers/
|   |-- iostats/
|   |-- mkfile
|   `-- Profiler.nw
|-- shells/
|   |-- mkfile
|   |-- rc/
|   `-- Shell.nw
|-- utilities/
|   |-- archive/
|   |-- byte/
|   |-- calc/
|   |-- compare/
|   |-- files/
|   |-- mkfile
|   |-- pipe/
|   |-- process/
|   |-- text/
|   |-- time/
|   `-- Utilities.nw
`-- ROOT/
    |-- 386/
    |-- arm/
    |-- lib/
    |-- rc/
    |-- tests/
    `-- usr/
```

Figure 1: https://github.com/aryx/principia-softwarica Layout.
(as generated by `tree -L 2 -F` and spread in 3 columns)

A New Way to Build

Goken `Dockerfile` excerpt	Principia Softwarica `Dockerfile`

`# Build goken on Ubuntu using`
`# gcc/binutils (and mk/rc)`
`FROM ubuntu:24.04`
 	
`# Setup a basic C dev environment`
`RUN apt-get install -y gcc libc6-dev \`
`    byacc`
 	
`# Now let's build from source`
`WORKDIR /src`
`COPY . .`
 	
`# Small shell script (not GNU autoconf)`
`# to detect arch and generate`
`# ./mkconfig`
`RUN ./configure`
 	
`# The script below obviously builds`
`# 'mk' (without needing mk) but also:`
`# - 'rc', which is called by 'mk'`
`# - 'ed', for the mkenam script`
`#   run during the build`
`RUN ./scripts/build-mk.sh`
 	
`# copy ./ROOT/<arch>/bin/{mk,rc,ed}`
`# to ./bin/`
`RUN ./scripts/promote-mk.sh`
 	
`# make mk and rc accessible`
`ENV PATH="/src/bin:${PATH}"`
`ENV MKSHELL="/src/bin/rc"`
 	
`# Let's build goken (using mk/rc built`
`# in the previous step)`
`RUN mk`
`RUN mk install`
 	
`ENV PATH="/src/ROOT/amd64/bin:\`
`         /src/ROOT/arm64/bin:\`
`         ${PATH}"`
	`# Build principia on Ubuntu`
	`# Linux (amd64 or arm64)`
	`# for 386 (pc) and arm (pi)`
	`FROM padator/goken`
	
	`WORKDIR /principia`
	`COPY . .`
	
	`# 386`
	`RUN cp mkconfig.pc mkconfig`
	`RUN mk && mk install`
	`RUN mk kernel`
	`# arm`
	`RUN cp mkconfig.pi mkconfig`
	`RUN mk && mk install`
	`RUN mk kernel`
	
	`# VFAT tools`
	`RUN apt-get install -y dosfstools \`
	`      mtools`
	
	`# making dosdisk.img`
	`RUN dd if=/dev/zero of=tmp.img \`
	`       bs=1M count=512`
	`RUN mkfs.vfat tmp.img`
	`RUN mcopy -i tmp.img -s -o \`
	`      ROOT/* ::`
	`RUN cat MISC/pc/bootsector tmp.img \`
	`     > dosdisk.img`
	`RUN rm -f tmp.img`

Table 3: Principia Softwarica Build and Packaging Process.

A New Way to Build

The Build Problem

Teaching context: students want to modify code, rebuild, and boot — from Linux, macOS, Windows, in their own editor or IDE.

Problem 1: Plan 9 uses a special C dialect with extensions

Architecture layouts, different calling conventions, custom assembler

→ gcc / clang can't compile it as-is

Problem 2: Existing options (9front, 9legacy) have

non-trivial build procedures, packaging not straightforward

Prior art: **Inferno** (Vita Nuova), **plan9port** (Russ Cox), **kencc** (9legacy)

Each made steps toward vizio compiling Plan 9 code from Unix/Linux

goken9cc: Plan 9 compilers, built with gcc/clang

Cross-compile from Linux/macOS/Windows → boot with QEMU or Raspberry Pi

github.com/aryx/goken9cc

5 Commands: Clone, Build, Run

```
$ git clone https://github.com/aryx/goken9cc $ cd goken9cc && docker build -t
"padator/goken" . $ cd .../principia-softwarica $ docker build -t "principia" . $
qemu-system-i386 -smp 4 -m 512 \ -kernel 9qemu -hda dosdisk.img
```

Irony: Docker relies on Linux namespaces — inspired by Plan 9

github.com/aryx/goken9cc

Live Demo: Modify, Rebuild, Boot

```
BEFORE: $ strace ls → open = syscall 6
```

```
CHANGE: $ vi lib_core/libc/9syscall/sys.h #define OPEN 6 → #define OPEN 42
```

```
REBUILD: $ time mk && mk install && mk kernel < 10 seconds
```

```
BOOT: $ qemu-system-i386 -kernel 9qemu -hda dosdisk.img $ strace ls → open =
syscall 42
```

"Rebuilt the entire OS in under a minute. Try that with Linux."

That `ls` went through shell, libc, kernel, graphics, windowing. Every layer — explained in a Principia book.

Goken Dockerfile excerpt	Principia Softwarica Dockerfile

`# Build goken on Ubuntu using`
`# gcc/binutils (and mk/rc)`
`FROM ubuntu:24.04`
 	
`# Setup a basic C dev environment`
`RUN apt-get install -y gcc libc6-dev \`
`  byacc`
 	
`# Now let's build from source`
`WORKDIR /src`
`COPY . .`
 	
`# Small shell script (not GNU autoconf)`
`# to detect arch and generate`
`# ./mkconfig`
`RUN ./configure`
 	
`# The script below obviously builds`
`# 'mk' (without needing mk) but also:`
	`# Build principia on Ubuntu`
	`# Linux (amd64 or arm64)`
	`# for 386 (pc) and arm (pi)`
	`FROM padator/goken`
	
	`WORKDIR /principia`
	`COPY . .`
	
	`# 386`
	`RUN cp mkconfig.pc mkconfig`
	`RUN mk && mk install`
	`RUN mk kernel`
	`# arm`
	`RUN cp mkconfig.pi mkconfig`
	`RUN mk && mk install`
	`RUN mk kernel`
	
	`# VFAT tools`
	`RUN apt-get install -y dosfstools \`

The Build Problem

Teaching context: students want to **modify** code, **rebuild**, and **boot** — from Linux, macOS, Windows, in their own editor or IDE.

Problem 1: Plan 9 uses a special C dialect with extensions

Anonymous structs, different calling conventions, custom assembler

→ `gcc` / `clang` can't compile it as-is

Problem 2: Existing options (9front, 9legacy) have

non-trivial build procedures, packaging not straightforward

Prior art: **Inferno** (Vita Nuova), **plan9port** (Russ Cox), **kencc** (9legacy)

Each made steps toward cross-compiling Plan 9 code from Unix/Linux

goken9cc: Plan 9 compilers, built with gcc/clang

Cross-compile from Linux/macOS/Windows → boot with QEMU or Raspberry Pi

`github.com/aryx/goken9cc`

ant to **modify** code, **rebuild**, and **boot**
ows, in their own editor or IDE.

ecial C dialect with extensions
ling conventions, custom assembler
e it as-is

(9front, 9legacy) have
kaging not straightforward

a), **plan9port** (Russ Cox), **kencc** (9legacy)
compiling Plan 9 code from Unix/Linux

ilers, built with gcc/clang
DS/Windows → boot with QEMU or Raspberry Pi

c

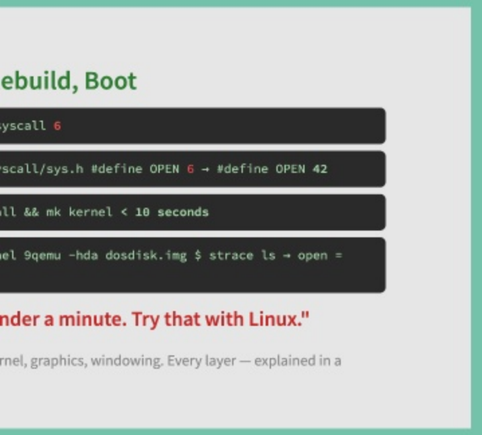
, Run

```
/aryx/goken9cc $ cd goken9cc && docker build -t
cipia-softwarica $ docker build -t "principia" . $
 \ -kernel 9qemu -hda dosdisk.img
```

aces — inspired by Plan 9

c

Goken Dockerfile excerpt	P
`# Build goken on Ubuntu using`	#
`# gcc/binutils (and mk/rc)`	#
`FROM ubuntu:24.04`	#
	F
`# Setup a basic C dev environment`	
`RUN apt-get install -y gcc libc6-dev \`	W
`    byacc`	C
`# Now let's build from source`	#
`WORKDIR /src`	R
`COPY . .`	R
	R




```
# Small shell script (not GNU autoconf)
# to detect arch and generate
# ./mkconfig
RUN ./configure

# The script below obviously builds
# 'mk' (without needing mk) but also:
# - 'rc', which is called by 'mk'
# - 'ed', for the mkenam script
#    run during the build
RUN ./scripts/build-mk.sh

# copy ./ROOT/<arch>/bin/{mk,rc,ed}
# to ./bin/
RUN ./scripts/promote-mk.sh
```

```
RUN mk kerne
# arm
RUN cp mkcon
RUN mk && mk
RUN mk kerne

# VFAT tools
RUN apt-get
      mtools

# making dos
RUN dd if=/d
         bs=1
RUN mkfs.vfa
RUN mcopy -i
      ROOT/*
RUN cat MISC
```

```
# make mk and rc accessible
ENV PATH="/src/bin:${PATH}"
ENV MKSHELL="/src/bin/rc"

# Let's build goken (using mk/rc built
# in the previous step)
RUN mk
RUN mk install

ENV PATH="/src/ROOT/amd64/bin:\
          /src/ROOT/arm64/bin:\
          ${PATH}"
```

```
      > dosdi
RUN rm -f tm
```

Table 3: Principia Softwarica Build and Packaging Pro

Principia Softwarica `Dockerfile`

```
# Build principia on Ubuntu
# Linux (amd64 or arm64)
# for 386 (pc) and arm (pi)
FROM padator/goken

WORKDIR /principia
COPY . .

# 386
RUN cp mkconfig.pc mkconfig
RUN mk && mk install
RUN mk kernel
# arm
RUN cp mkconfig.pi mkconfig
RUN mk && mk install
RUN mk kernel
```

```
# VFAT tools
RUN apt-get install -y dosfstools \
      mtools

# making dosdisk.img
RUN dd if=/dev/zero of=tmp.img \
        bs=1M count=512
RUN mkfs.vfat tmp.img
RUN mcopy -i tmp.img -s -o \
      ROOT/* ::
RUN cat MISC/pc/bootsector tmp.img \
     > dosdisk.img
RUN rm -f tmp.img
```

5 Commands: Clone, Build, Run

```
$ git clone https://github.com/aryx/goken9cc $ cd goken9cc && docker build -t
"padator/goken" . $ cd ../principia-softwarica $ docker build -t "principia" . $
qemu-system-i386 -smp 4 -m 512 \ -kernel 9qemu -hda dosdisk.img
```

Irony: Docker relies on Linux namespaces — inspired by Plan 9

github.com/aryx/goken9cc

Live Demo: Modify, Rebuild, Boot

```
BEFORE: $ strace ls → open = syscall 6
```

```
CHANGE: $ vi lib_core/libc/9syscall/sys.h #define OPEN 6 → #define OPEN 42
```

```
REBUILD: $ time mk && mk install && mk kernel < 10 seconds
```

```
BOOT: $ qemu-system-i386 -kernel 9qemu -hda dosdisk.img $ strace ls → open =
syscall 42
```

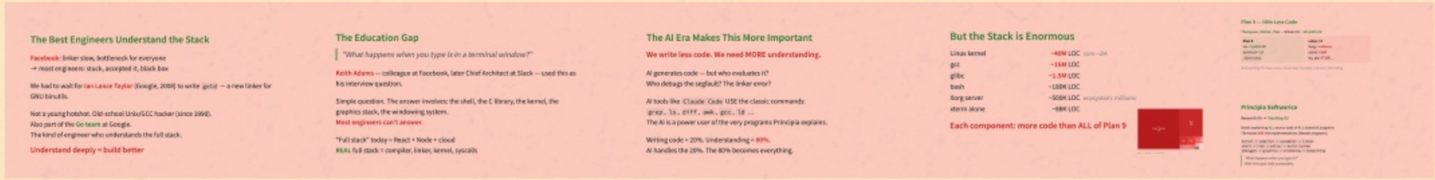
"Rebuilt the entire OS in under a minute. Try that with Linux."

That `ls` went through shell, libc, kernel, graphics, windowing. Every layer — explained in a Principia book.

Principia Softwarica

Plan 9 Code Explained

Yoann Padioleau **IWP9**



A Series of Books

Category	Book	Program(s)	LOC	LOE	Pages
Core system	Kernel	9pi	35000	3200	732
	Core libraries	libc libregexp libthread	19000	1600	438
	Shell	rc	6500	1700	166
Development toolchain	C compiler	5c libcc	18500	1900	471
	Assembler	5a	3600	4400	176
	Linker	5l	7500	5400	296
Developer tools	Editor	ed	1600	200	45
	Build system	mk	4350	4050	197
	Debuggers	db acid strace libmach	13100	1000	321
	Profilers	prof time kprof stats iostats	3900	350	102
Graphics	Graphics stack	/dev/draw libdraw libmemdraw libmemlayer	18500	3400	507
	Windowing system	rio libframe libcomplete libplumb	8800	4000	289
Networking	Network stack	/dev/net libip lib9p	18300	4800	457
Misc	CLI utilities	cat ls grep sed diff tar gzip ...	23900	650	493
Total:			182550	36650	4690

Table 1: Principia Softwarica Books and Their Statistics.
(**LOC** = lines of code; **LOE** = lines of explanation; **Pages** = number of typeset pages)

A New Plan 9 Fork

```
.
|-- assemblers/
|   |-- 5a/
|   |-- 8a/
|   |-- data2s.c
|   |-- Assembler.nw
|   `-- mkfile
|-- builders/
|   |-- Builder.nw
|   |-- mk/
|   `-- mkfile
|-- compilers/
|   |-- 5c/
|   |-- 8c/
|   |-- cc/
|   |-- Compiler.nw
|   |-- cpp/
|   `-- mkfile
|-- debuggers/
|   |-- acid/
|   |-- db/
|   |-- Debugger.nw
|   |-- libmach/
|   `-- mkfile
|-- Dockerfile
|-- docs/
|   |-- articles/
|   `-- iwp9/
|-- dosdisk.img
|-- editors/
|   |-- ed/
|   `-- mkfile
|-- include/
|   |-- libc.h
|   |-- ...
|   `-- u.h
|-- linkers/
|   |-- 5l/
|   |-- 8l/
|   |-- Linker.nw
|   `-- mkfile
|-- readme.txt
`-- windows/
    |-- libcomplete/
    |-- libplumb/
    |-- mkfile
    |-- rio/
    `-- Windows.nw

|-- kernel/
|   |-- arch/
|   |-- concurrency/
|   |-- console/
|   |-- devices/
|   |-- files/
|   |-- filesystems/
|   |-- init/
|   |-- memory/
|   |-- Kernel.nw
|   |-- mkfile
|   |-- network/
|   |-- processes/
|   `-- syscalls/
|-- lib_core/
|   |-- libbio/
|   |-- libc/
|   |-- Libcore.nw
|   |-- libthread/
|   `-- mkfile
|-- lib_graphics/
|   |-- Graphics.nw
|   |-- libdraw/
|   |-- libimg/
|   |-- libmemdraw/
|   |-- libmemlayer/
|   `-- mkfile
|-- lib_networking/
|   |-- lib9p/
|   |-- libip/
|   `-- mkfile
|-- lib_strings/
|   |-- libflate/
|   |-- libregexp/
|   |-- libstring/
|   `-- mkfile
|-- Makefile
|-- mkconfig.pc
|-- mkconfig.pi
|-- mkfile
|-- mkfile-host-Cygwin
|-- mkfile-host-Linux
|-- mkfile-host-macOS
|-- mkfile-host-Plan9
`-- MISC/
    |-- pc/
    `-- pi/

|-- mkfiles/
|   |-- 386/
|   |-- arm/
|   |-- mkdirs
|   |-- mkfile.proto
|   |-- mklib
|   `-- mkone
|-- mkfile-target-pc
|-- mkfile-target-pi
|-- networking/
|   |-- arp/
|   |-- dhcp/
|   |-- ftp/
|   |-- http/
|   |-- ip/
|   |-- mkfile
|   |-- ndb/
|   |-- Network.nw
|   |-- snoopy/
|   `-- telnet/
|-- profilers/
|   |-- iostats/
|   |-- mkfile
|   `-- Profiler.nw
|-- shells/
|   |-- mkfile
|   |-- rc/
|   `-- Shell.nw
|-- utilities/
|   |-- archive/
|   |-- byte/
|   |-- calc/
|   |-- compare/
|   |-- files/
|   |-- mkfile
|   |-- pipe/
|   |-- process/
|   |-- text/
|   |-- time/
|   `-- Utilities.nw
`-- ROOT/
    |-- 386/
    |-- arm/
    |-- lib/
    |-- rc/
    |-- tests/
    `-- usr/
```

Figure 1: https://github.com/aryx/principia-softwarica Layout.
(as generated by `tree -L 2 -F` and spread in 3 columns)

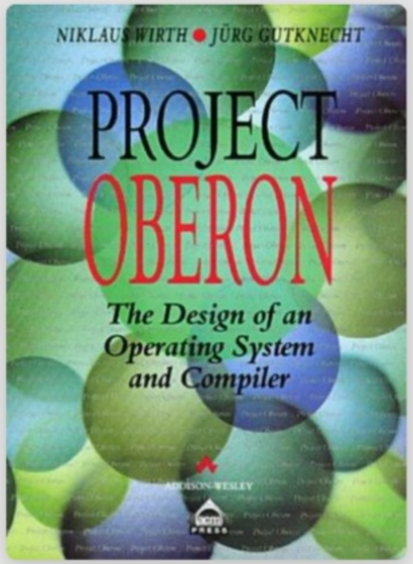
A New Way to Build

Goken `Dockerfile` excerpt	Principia Softwarica `Dockerfile`

`# Build goken on Ubuntu using`
`# gcc/binutils (and mk/rc)`
`FROM ubuntu:24.04`
 	
`# Setup a basic C dev environment`
`RUN apt-get install -y gcc libc6-dev \`
`    byacc`
 	
`# Now let's build from source`
`WORKDIR /src`
`COPY . .`
 	
`# Small shell script (not GNU autoconf)`
`# to detect arch and generate`
`# ./mkconfig`
`RUN ./configure`
 	
`# The script below obviously builds`
`# 'mk' (without needing mk) but also:`
`# - 'rc', which is called by 'mk'`
`# - 'ed', for the mkenam script`
`#   run during the build`
`RUN ./scripts/build-mk.sh`
 	
`# copy ./ROOT/<arch>/bin/{mk,rc,ed}`
`# to ./bin/`
`RUN ./scripts/promote-mk.sh`
 	
`# make mk and rc accessible`
`ENV PATH="/src/bin:${PATH}"`
`ENV MKSHELL="/src/bin/rc"`
 	
`# Let's build goken (using mk/rc built`
`# in the previous step)`
`RUN mk`
`RUN mk install`
 	
`ENV PATH="/src/ROOT/amd64/bin:\`
`         /src/ROOT/arm64/bin:\`
`         ${PATH}"`
	`# Build principia on Ubuntu`
	`# Linux (amd64 or arm64)`
	`# for 386 (pc) and arm (pi)`
	`FROM padator/goken`
	
	`WORKDIR /principia`
	`COPY . .`
	
	`# 386`
	`RUN cp mkconfig.pc mkconfig`
	`RUN mk && mk install`
	`RUN mk kernel`
	`# arm`
	`RUN cp mkconfig.pi mkconfig`
	`RUN mk && mk install`
	`RUN mk kernel`
	
	`# VFAT tools`
	`RUN apt-get install -y dosfstools \`
	`      mtools`
	
	`# making dosdisk.img`
	`RUN dd if=/dev/zero of=tmp.img \`
	`       bs=1M count=512`
	`RUN mkfs.vfat tmp.img`
	`RUN mcopy -i tmp.img -s -o \`
	`      ROOT/* ::`
	`RUN cat MISC/pc/bootsector tmp.img \`
	`     > dosdisk.img`
	`RUN rm -f tmp.img`

Table 3: Principia Softwarica Build and Packaging Process.

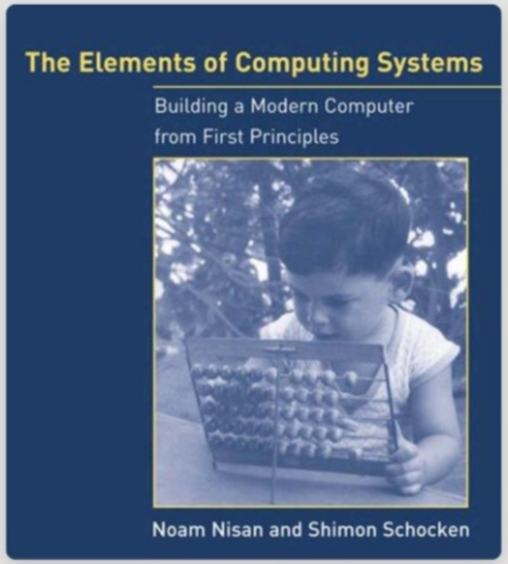
Related Work



Project Oberon

Closest in spirit.

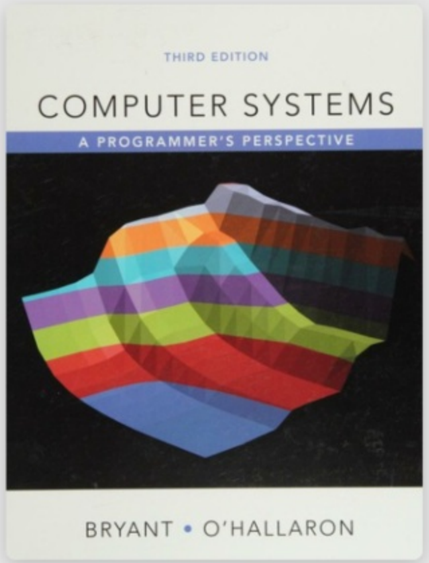
But Oberon-only, no networking.



Nand2Tetris

NAND gates to Tetris.

Principia: real OS, real code.



CS:APP

Explains concepts.

Principia: explains the source code.

Principia complements all three.

Future Work

- More explanations (LOE/LOC → 1)

  AI could help write explanations

- Self-hosting (build Principia from Principia)
- Merge goken + principia codebases

Principia Softwarica

Principia removes the mental barrier.

The stack is not a black box.

These programs are not impossible to understand.

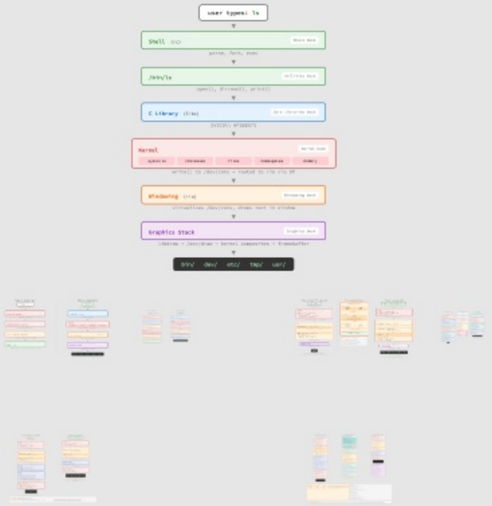
In other fields, students study the masters.

Read the masters' code. Understand everything.

Become a better engineer.

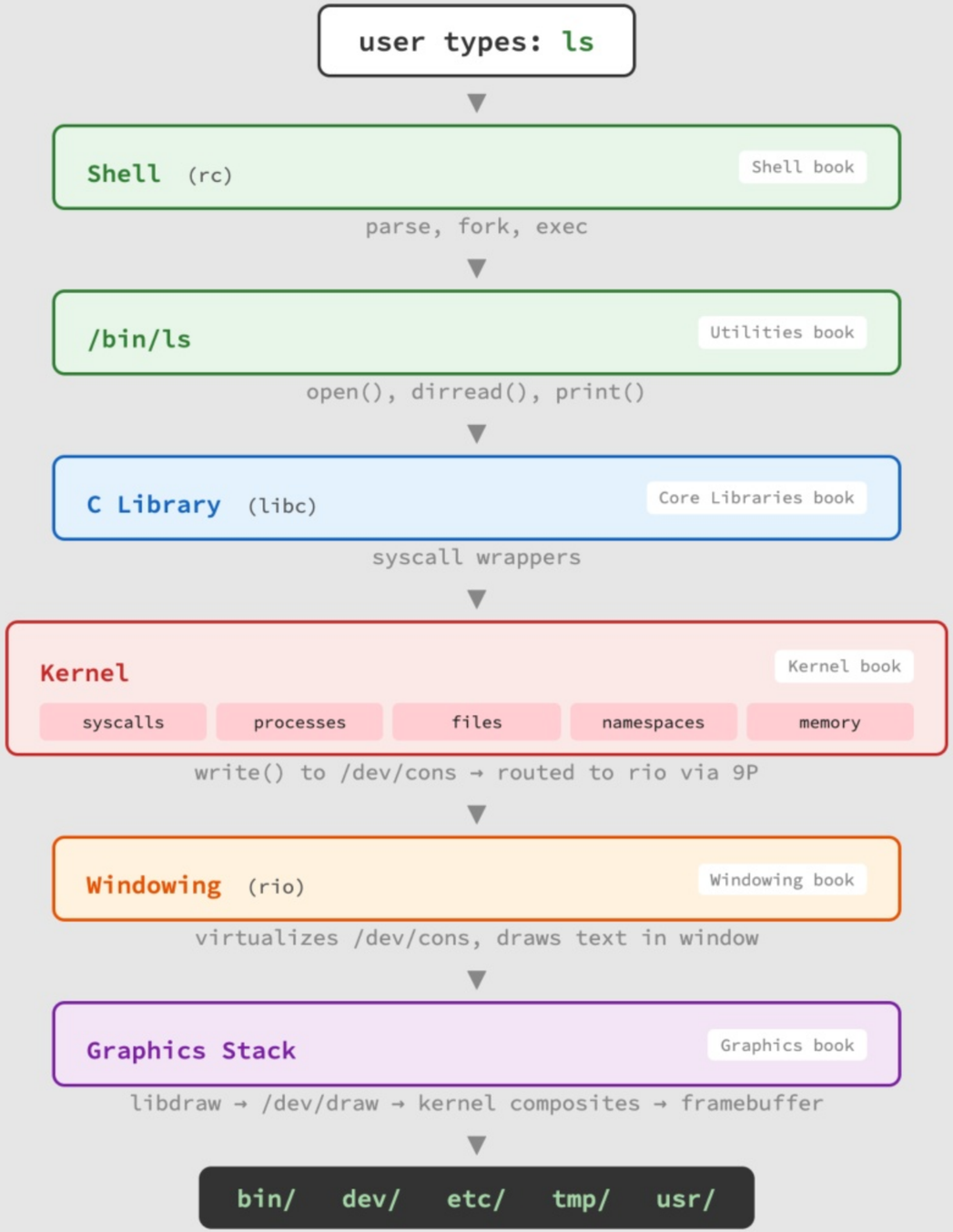
"What happens when you type ls?"

After reading Principia: you'll know. Fully.



Clone it. Build it. Read the books.

`github.com/aryx/principia-softwarica`


user types: ls
Shell (rc)
Shell book
parse, fork, exec
/bin/ls
Utilities book
open(), dirread(), print()
C Library (libc)
Core Libraries book
syscall wrappers
Kernel
Kernel book
syscalls
processes
files
namespaces
memory
write() to /dev/cons → routed to rio via 9P
Windowing (rio)
Windowing book
virtualizes /dev/cons, draws text in window
Graphics Stack
Graphics book
libdraw → /dev/draw → kernel composites → framebuffer
bin/ dev/ etc/ tmp/ usr/

Phase 1: Typing "ls↵"

⌨ keys: l s ↵

hardware interrupt

▼

Interrupt handler — Kernel: Interrupts

scancode → character

▼

Console device /dev/cons — Kernel: Devices

echo char, buffer line

▼

rio virtualizes /dev/cons — Windowing book

↵ newline → flush line to reader

▼

Shell (rc) — Shell book

read() returns "ls\n" → parse → fork + exec

Phase 2: Executing ls

/bin/ls

open(), dirread(), print()

▼

C Library (libc) — Core Libraries book

syscall wrappers

▼

Kernel — Kernel book

syscalls | files | namespaces | processes

write() output → /dev/cons

▼

rio window mgr — Windowing book

draw text in window

▼

Graphics Stack — Graphics book

framebuffer → screen

▼

bin/ dev/ etc/ tmp/ usr/

Phase 1: Typing "ls↵"

⌨ keys: l s ↵

USB/PS2 interrupt

▼

Input subsystem evdev — Kernel

scancode → keycode

▼

Xorg X server — ~500K LOC

KeyPress event → xterm

▼

xterm terminal emulator — 88K LOC

write 'l' to PTY master fd

▼

Kernel: PTY + TTY — Kernel

pty driver | n_tty ldisc | echo

echo back to master + buffer line

▼

xterm VT100 parse + render — 88K LOC

Xft/FreeType → X11 draw request

▼

bash (shell) — 180K LOC

read() returns "ls\n" → parse → fork + exec

Phase 2: Executing ls

/bin/ls

opendir(), readdir(), printf()

▼

glibc (C library) — 1.5M LOC

syscall wrappers

▼

Kernel — ~40M LOC

syscalls | VFS | ext4/btrfs | scheduler

write() output → PTY slave fd

▼

PTY + TTY n_tty ldisc — Kernel

data → master side

▼

xterm VT100 + render — 88K LOC

Xft/FreeType → X11 draw request

▼

Xorg composite + render — ~500K LOC

DRM/KMS → framebuffer

▼

bin/ dev/ etc/ tmp/ usr/

Phase 1: Typing "l" — echo path

⌨ key press: l

hardware interrupt

▼

Kernel — Kernel book

`kbdputc('l')` → circular buffer

clock tick → `echo()`:

`qproduce(kbdq, 'l')` — into queue

raw mode → skip screen echo

read /dev/cons returns 'l' from kbdq

▼

rio — Windowing book

keyboardthread — recv(keyboardctl→c)

sendp(input→ck, 'l') channel

▼

winctl (per window) — event loop

WKey → `wkeyctl(w, 'l')`

→ **winsert**() — insert rune in w→r[]

→ **frinsert**() — **draw 'l' glyph**

draw() via /dev/draw

▼

Graphics Stack — Graphics book

framebuffer → screen

▼

% l

Then 's' and '↵' follow the same path.

On '↵', winctl marks line ready for shell to read.

Rio's internal threads

Thread architecture

Global threads:

keyboard thread	mouse thread

↓ channels ↓

Per window:

winctl

event loop: alt(WKey, WMouse, WCread, WCwrite, WCtl, ...)

↕ channels

9P file server:

filsysproc reads 9P msgs	**xfidctl** worker pool

↕ 9P over /dev/cons

Per window (separate process):

winshell → rc (shell)

Channels (typed message queues):

- ck — keyboard runes → winctl
- mc.c — mouse events → winctl
- consread — shell read req ↔ winctl
- conswrite — shell write req ↔ winctl
- cctl — control msgs → winctl

Phase 2: ls output path

/bin/ls → print("bin/\n...")

write() syscall to virtual /dev/cons

▼

Kernel — Kernel book

sys_write() → devtab→write()

but /dev/cons is **rio's virtual file**

→ 9P write message to rio

9P write to rio's file server

▼

rio — Windowing book

filsysproc — read9pmsg() → dispatch

▼

xfidctl worker → **xfidwrite**(Qcons)

send on conswrite channel

▼

winctl (per window)

WCwrite → recv runes

→ **winsert**() — insert into w→r[]

→ **frinsert**() — **draw text**

draw() via /dev/draw

▼

Graphics Stack — Graphics book

framebuffer → screen

▼

bin/ dev/ etc/ tmp/ usr/

Both echo and output go through

the same winctl → winsert → frinsert path.

Phase 1: Typing "l" — echo path

⌨ key press: l

hardware interrupt → IRQ1

▼

Kernel — input subsystem

~40M LOC

i8042 driver (PS/2) or **usbhid** (USB)
→ **input subsystem** (evdev)
→ **/dev/input/event0**

evdev event: EV_KEY, KEY_L, value=1

▼

Xorg X server

~500K LOC

evdev driver reads /dev/input/event0
→ XKB: scancode → keysym → 'l'
→ build X11 **KeyPress** event
→ dispatch to focused window (xterm)

X11 protocol: KeyPress event → xterm

▼

xterm

~88K LOC

XLookupString(KeyPress) → 'l'
→ **write**(master_fd, "l", 1) → PTY master

write 'l' to PTY master side

▼

Kernel — PTY + line discipline

PTY master (/dev/ptmx) → **n_tty** line discipline
→ **ECHO flag set**: copy 'l' back to master read buffer
→ buffer 'l' on slave side (wait for '\n')

echo: xterm reads 'l' back from PTY master

▼

xterm — render echo

read(master_fd) → gets 'l' (echo)
→ VT100 state machine (parse escape sequences)
→ **Xft/FreeType**: rasterize glyph
  fontconfig → find font file (.ttf/.otf)
  FreeType → rasterize outline → bitmap
→ **XDrawString** / XRenderCompositeString

X11 protocol: draw requests → X server

▼

Xorg — render + composite

receive draw request → **Render** extension
→ composite via GPU (DRM/KMS) or software
→ **/dev/dri/card0** → GPU driver → framebuffer

▼

$ l

Linux terminal architecture

The PTY/TTY maze

User space (xterm):

xterm
X11 event loop + VT100 emulator
+ Xft/FreeType font rendering

↕ read/write PTY master fd

Kernel:

/dev/ptmx (master) ↔ **/dev/pts/N** (slave)

↕

n_tty line discipline
echo, line editing, signals,
canonical mode, raw mode,
VEOF, VERASE, VKILL, VINTR...

↕ read/write PTY slave fd

User space (shell):

bash (stdin/stdout = /dev/pts/N)

Historical layers:

VT100 escape codes (1978)
termios/termcap/terminfo
PTY master/slave (BSD, 1980s)
devpts filesystem (Linux 2.2)
X11 protocol (1987)
XKB keyboard extension
Xft + FreeType + fontconfig
DRM/KMS (2000s)

"a maze of historical layers few people fully understand"

Code size comparison:

Linux kernel	~40M LOC
Xorg server	~500K LOC
xterm	~88K LOC
glibc	~1.5M LOC
bash	~180K LOC
FreeType	~120K LOC
Each component alone	**> all of Plan 9**
Plan 9 entire OS	**~183K LOC**

Phase 2: ls output path

/bin/ls → printf("bin/\n...")

write() to stdout (fd 1 = PTY slave)

▼

glibc

~1.5M LOC

stdio buffering → write() syscall

▼

Kernel — PTY + TTY layer

write to /dev/pts/N (slave)
→ **n_tty** line discipline: output processing
  OPOST: '\n' → '\r\n' (if ONLCR)
→ data available on PTY master side
→ wake up xterm's read()

xterm reads output from PTY master

▼

xterm — VT100 + render

~88K LOC

read(master_fd) → "bin/\r\ndev/\r\n..."
→ **VT100 state machine**: parse bytes
  handle escape sequences (colors, cursor, scroll)
→ place chars in screen buffer[80×24]
→ **Xft + FreeType**: rasterize glyphs
  fontconfig → /usr/share/fonts/...
  FreeType: load .ttf → outline → bitmap
→ **XRenderCompositeString**()

X11 protocol: render requests → X server

▼

Xorg X server

~500K LOC

receive Render composite request
→ GPU acceleration (if available)
→ **DRM/KMS** → /dev/dri/card0
→ GPU driver → framebuffer

GPU scanout → monitor

▼

bin/ dev/ etc/ tmp/ usr/

Note: with Wayland, replace Xorg with compositor (e.g. Mutter) — still equally complex. With GPU terminal (Alacritty, kitty), font rendering moves to GPU shaders — even more code.

Phase 1: Typing "l" — echo path

⌨ key press: l

hardware interrupt

▼

Kernel — Kernel book

```
kbdputc('l') → circular buffer
clock tick → echo():
  qproduce(kbdq, 'l') — into queue
  raw mode → skip screen echo
```

read /dev/cons returns 'l' from kbdq

▼

rio — Windowing book

`keyboardthread` — recv(keyboardctl→c)

sendp(input→ck, 'l') channel

▼

`winctl` (per window) — event loop

```
WKey → wkeyctl(w, 'l')
→ winsert() — insert rune in w→r[]
```

winsert calls frinsert on the Frame

▼

libframe — Widgets book

```
frinsert() → layout boxes, line wrap
→ _frdrawtext() → for each text box:
   stringbg(f→b, pt, text, font, "l", back)
```

▼

libdraw — Graphics book

```
_string():
  cachechars() — lookup glyph in font cache
   miss → loadchar() → send 'l' cmd to kernel
  build 'x' cmd: dst, src, font, pt, chars
  bufimage() → accumulate in Display.buf[]
```

flushimage() → write(fd, buf) to /dev/draw

▼

Kernel: /dev/draw — Kernel book

```
drawmesg() — parse 'x' command
→ drawchar() — get glyph metrics from FChar
   → memdraw(dst, src, font→image) — composite
→ dstflush() → arch_flushmemscreen()
```

▼

% l

pixels in VRAM framebuffer

Then 's' and '↵' follow the same path.
On '↵', winctl marks line ready for shell to read.

Phase 2: ls output path

/bin/ls → print("bin/\n...")

write() syscall to virtual /dev/cons

▼

Kernel — Kernel book

```
/dev/cons is rio's virtual file
→ 9P write message to rio
```

▼

rio — Windowing book

`filsysproc → xfidctl → xfidwrite(Qcons)`

send on conswrite channel

▼

`winctl` (per window)

```
WCwrite → recv runes
→ winsert() — insert into w→r[]
```

winsert calls frinsert on the Frame

▼

libframe → libdraw — Graphics + Widgets

```
frinsert → stringbg → _string
(same path as echo — see left)
```

write to /dev/draw

▼

Kernel: /dev/draw — Kernel book

```
drawmesg → drawchar → memdraw
→ dstflush → arch_flushmemscreen
```

▼

bin/ dev/ etc/ tmp/ usr/

pixels in VRAM framebuffer

Both echo and output end the same way:
winsert → frinsert → libdraw → /dev/draw → memdraw → framebuffer
User-space builds commands, kernel composites pixels.

Key insight: rio goes through the kernel twice

1st pass (input): keyboard interrupt → kbdq → /dev/cons → rio
2nd pass (display): rio → libdraw protocol → /dev/draw → kernel composites → framebuffer

Font system: cachechars() keeps glyph cache in user-space (libdraw).
On cache miss, loadchar() sends subfont pixels to kernel via /dev/draw 'l' cmd.
Kernel stores glyph bitmaps in font→image (Memimage in kernel memory).

PHASE 1: TYPING "l" — ECHO PATH

⌨ key press: l

PS/2 scancode → IRQ1 hardware interrupt

Kernel — interrupt handler + console device (Kernel book)

```
i8042intr() → scancode to runeq
→ kbdputc('l') → kbd.istage[] circular buffer
— batched at interrupt time, no queue ops yet —
clock tick (22ms) → kbdputcclock():
  → echo('l'):
    qproduce(kbdq, 'l') — into kernel queue
    kbd.raw == true → SKIP echoscreen()
    /rio set raw mode — kernel does NOT echo
```

libthread keyboard reader: read(/dev/cons) → gets 'l' from kbdq

rio — keyboard input to window (Windowing book)

keyboardthread — recv(keyboardctl→c, 'l')

sendp(input→ck, 'l') — typed channel to active window

winctl (per window) — event loop: alt(WKey, WMouse, WCread, ...)

```
case WKey:
  → wkeyctl(w, 'l') — not raw, not navigation key
  → winsert(w, &'l', 1, w→q0) — insert rune to w→r[] buffer
    runemove(w→r + q0, &r, 1) — shift text, insert
    adjust cursors: w→q0++, w→q1++
```

winsert calls frinsert on the visible Frame

libframe — text widget rendering (Widgets book)

```
frinsert(&w→frm, "l", q0 = w→org)
→ bxscan() — break text into Frbox[] (text runs + tabs + newlines)
→ calculate layout, line wrapping
→ _frdrawtext(f, pt, text_color, back_color):
  for each box: stringbg(f→b, pt, text, ZP, f→font, "l", back, ZP)
```

libdraw — draw protocol + font cache (Graphics book)

```
stringbg → _string(dst, pt, src, font, "l", bg, SoverD):
  cachechars(font, "l") — hash lookup in Font.cache[]
    hit → use cached index
    miss → loadchar() → send 'l' cmd to kernel
      (load subfont glyph bitmap into font→cacheimage)
  build 'x' cmd: dst_id, src_id, font_id, pt, clipr, char_indices
  bufimage(display, n) — append to Display.buf[32KB]
```

flushimage() → write(fd, buf) to /dev/draw

Kernel — /dev/draw graphics device (Kernel book)

```
devdraw write() → drawmesg() — parse binary 'x' command
  unpack: dst, src, font, pt, clipr, char_indices
  fill background: memdraw(dst, r, bg, ...)
  for each char index:
    → drawchar(dst, bg, q, src, font, ci, SoverD)
      fc = font→fchar[ci] — glyph metrics (left, width, miny, maxy)
      → memdraw(dst, r, src, sp, font→image, Pt(fc→minx, fc→miny), SoverD)
        glyph bitmap is the MASK — src color painted through it
→ dstflush(dstid, dst, r) → arch_flushmemscreen(r)
```

% l

VRAM framebuffer → VGA/HDMI scanout → monitor

Same path repeats for 's' and '↵'.
After Enter: winctl has "ls\n" in w→r[]. Line ready for reader.

THE BRIDGE: SHELL READS "ls\n"

Namespace setup (at window creation) (Kernel book)

```
winshell() — rio creates a process for each window:
  1. rfork(RFNAMEG | RFFDG | RFENVG)
     → newpgrp() → new per-process namespace (Pgrp)
     → pgrpcpy() → copy parent's mount table
  2. fd=mount(fd[1], -1, "/mnt/wsys", MREPL, id)
     9P connection to rio's file server → /mnt/wsys/
  → bind("/mnt/wsys", "/dev", MBEFORE)
     union mount: rio's files searched BEFORE real /dev
  3. close(0); open("/dev/cons", OREAD) → stdin
     close(1); open("/dev/cons", OWRITE) → stdout
     /dev/cons now resolves to rio's Qcons, not kernel's!
  4. procexec(pid, "/bin/rc", argv) → launch shell
```

Shell's namespace (per-process)

/dev/cons	/dev/mouse	/dev/draw
→ rio's Qcons (via bind MBEFORE)	→ rio's Qmouse (via bind MBEFORE)	→ kernel devdraw (unchanged)

Now the shell is blocked on read("/dev/cons")...

Shell (rc) — blocked read (Shell book)

```
read(0, buf, n) — stdin = rio's /dev/cons
→ libc: _read() → SYSCALL → kernel sys_read()
```

kernel routes to rio via 9P: Tread → rio's pipe

rio — 9P file server handles read (Windowing book)

filsysproc — read9pmsg() → Tread for Qcons

xfidctl worker → xfidread(Qcons)

send on consread channel — blocks until winctl ready

winctl — case WCread (after '↵' made line ready)

```
recv(crm.c1, &pair) — get buffer from xfidread
copy from w→r[w→q0..] until '\n':
  runetochar(t, &w→r[w→q0++]) → "l", "s", "\n"
send(crm.c2, &pair) — return "ls\n" to xfidread
```

9P Rread → kernel → shell's read() returns 3 bytes: "ls\n"

Shell (rc) — parse + execute (Shell book)

```
yyparse() — parse "ls\n" → AST: simple command
rfork(RFFROC|RFFDG|RFNOTEG) — fork child process
  child inherits namespace → same /dev/cons → same rio window
exec("/bin/ls", ["ls"]) — replace child with ls binary
```

libc: exec() → SYSCALL → kernel sys_exec()

C Library (libc) (Core Libraries book)

```
ls binary linked with libc: open(), dirread(), print()
Each wraps a syscall: libc → SYSCALL instruction → kernel trap
```

syscall wrappers → kernel

/bin/ls — the program (Utilities book)

```
fd = open("/", OREAD) → kernel: namec("/") → root devtab
dirread(fd, dirs) → kernel: devtab→read() → directory entries
for each entry: print("%s\n", name)
  → write(1, "bin/\n", 4) → stdout = rio's /dev/cons
```

write() to /dev/cons → enters the OUTPUT path →

PHASE 2: ls OUTPUT → SCREEN

Kernel — syscall + 9P routing (Kernel book)

```
sys_write(fd=1, "bin/\n", 4)
→ fd 1 → Chan → mounted filesystem check
→ /dev/cons is rio's Qcons (via namespace bind)
→ devmnt: build 9P Twrite message
→ write to rio's 9P pipe
```

9P Twrite → rio's file server pipe

rio — 9P file server handles write (Windowing book)

filsysproc — read9pmsg() → Twrite for Qcons

xfidctl worker → xfidwrite(Qcons)

```
cvttorunes("bin/\n", r) → bytes → Rune[]
```

send on conswrite channel → runes to winctl

winctl — case WCwrite

```
recv(cwm.cw, &pair) — get runes
handle '\b' backspace processing
→ winsert(w, rp, nr, w→q0) — insert output text
  w→q0 += nr — advance read head past output
```

winsert calls frinsert on the Frame

libframe + libdraw — same path as echo (Graphics + Widgets)

```
frinsert() → _frdrawtext() → stringbg()
→ _string() → cachechars() + build 'x' command
→ bufimage() → flushimage() → write to /dev/draw
```

Kernel — /dev/draw (Kernel book)

```
drawmesg() → drawchar() → memdraw()
→ dstflush() → arch_flushmemscreen(r)
```

% ls

bin/ dev/ etc/ tmp/ usr/

VRAM framebuffer → scanout → monitor

Font system — how glyph bitmaps work

```
Font file: /lib/font/bit/lucm/unicode.9.font
  meta-file listing subfont ranges:
  0x0000 0x00FF latin1.9 → covers ASCII
  0x0100 0x024F latinext.9 → extended latin

Subfont: latin1.9 = bitmap image + metrics
  image: all glyphs side by side in one Memimage
  Fontchar[]: (x, top, bottom, left, width) per glyph

Cache: Font.cacheimage (user-space Image)
  cachechars('l') → hash lookup → cache index
  miss → loadchar(): read subfont, send glyph bitmap
    to kernel via 'l' cmd → kernel stores in font→image

Rendering: drawchar(dst, src, font→image)
  memdraw(dst, r, src, sp, mask=font→image, mp, SoverD)
  glyph bitmap = alpha mask
  pixel = src_color × mask + dst_color × (1 − mask)
  1-bit font: binary mask → fast chardraw()
  anti-aliased: grey mask → alphadraw()
```

Rio thread architecture

Global threads:

keyboardthread	mousethread	deletethread

Per window:

winctl — event loop: alt(WKey, WMouse, WCread, WCwrite, WCtl, WMousread, WAread)

9P file server (process):

filsysproc reads 9P msgs	xfidallocthread worker pool mgr	xfidctl (N) per-request workers

Per window (separate process):

winshell → rc (shell) — or any program

Channels (typed message queues): ck keyboard → winctl | mc,c mouse → winctl | consread shell read → winctl | conswrite shell write → winctl | cctl control → winctl

Key insights

Rio goes through the kernel twice:
- 1st: keyboard IRQ → kbdq → /dev/cons → rio
- 2nd: rio → /dev/draw → kernel composites → framebuffer

Namespaces are the key abstraction:
- Shell's /dev/cons ≠ kernel's /dev/cons
- Per-process mount table (Pgrp) → bind("/mnt/wsys", "/dev", MBEFORE)
- Same mechanism that Docker/containers use (Linux namespaces = Plan 9 idea)

Everything is a file + 9P:
- Console I/O = read/write on /dev/cons
- Graphics = read/write on /dev/draw
- Mouse = read on /dev/mouse
- All go through the same syscall path

winctl is the heart of rio:
- Single event loop per window (like a mini server)
- Both echo (WKey) and output (WCwrite) → same winsert → frinsert path

User-space builds commands, kernel renders pixels:
- libdraw accumulates draw protocol in 32KB buffer
- Kernel /dev/draw does actual pixel compositing in memdraw()
- Font glyphs used as alpha masks — elegant Porter-Duff compositing

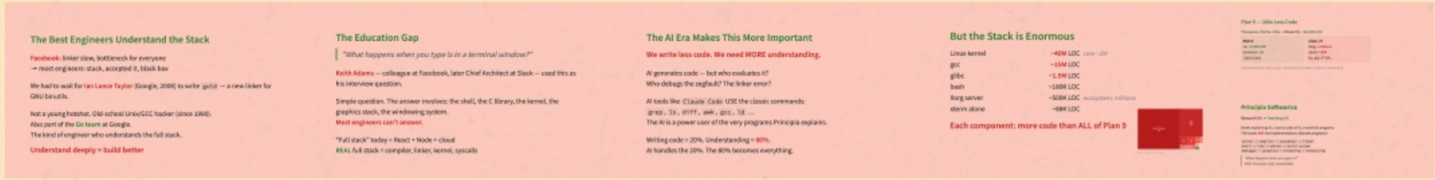
Every layer is explained in a Principia book

Kernel — interrupts, syscalls, /dev/draw, /dev/cons, namespaces, processes | Windowing — rio threads, 9P server, winctl, channels | Shell — rc parser, fork, exec | Utilities — ls, cat, grep, ... | Core Libraries — libc syscall wrappers | Graphics — libdraw, libmemdraw, font system | Widgets — libframe text rendering | Fonts — subfont bitmaps, cache, compositing

Principia Softwarica

Plan 9 Code Explained

Yoann Padioleau **IWP9**



A Series of Books

Category	Book	Program(s)	LOC	LOE	Pages
Core system	Kernel	9pi	35000	3200	732
	Core libraries	libc libregexp libthread	19000	1600	438
	Shell	rc	6500	1700	166
Development toolchain	C compiler	5c libcc	18500	1900	471
	Assembler	5a	3600	4400	176
	Linker	5l	7500	5400	296
Developer tools	Editor	ed	1600	200	45
	Build system	mk	4350	4050	197
	Debuggers	db acid strace libmach	13100	1000	321
	Profilers	prof time kprof stats iostats	3900	350	102
Graphics	Graphics stack	/dev/draw libdraw libmemdraw libmemlayer	18500	3400	507
	Windowing system	rio libframe libcomplete libplumb	8800	4000	289
Networking	Network stack	/dev/net libip lib9p	18300	4800	457
Misc	CLI utilities	cat ls grep sed diff tar gzip ...	23900	650	493
Total:			182550	36650	4690

Table 1: Principia Softwarica Books and Their Statistics.
(**LOC** = lines of code; **LOE** = lines of explanation; **Pages** = number of typeset pages)

A New Plan 9 Fork

```
.
|-- assemblers/
|   |-- 5a/
|   |-- 8a/
|   |-- data2s.c
|   |-- Assembler.nw
|   `-- mkfile
|-- builders/
|   |-- Builder.nw
|   |-- mk/
|   `-- mkfile
|-- compilers/
|   |-- 5c/
|   |-- 8c/
|   |-- cc/
|   |-- Compiler.nw
|   |-- cpp/
|   `-- mkfile
|-- debuggers/
|   |-- acid/
|   |-- db/
|   |-- Debugger.nw
|   |-- libmach/
|   `-- mkfile
|-- Dockerfile
|-- docs/
|   |-- articles/
|   `-- iwp9/
|-- dosdisk.img
|-- editors/
|   |-- ed/
|   `-- mkfile
|-- include/
|   |-- libc.h
|   |-- ...
|   `-- u.h
|-- linkers/
|   |-- 5l/
|   |-- 8l/
|   |-- Linker.nw
|   `-- mkfile
|-- readme.txt
`-- windows/
    |-- libcomplete/
    |-- libplumb/
    |-- mkfile
    |-- rio/
    `-- Windows.nw

|-- kernel/
|   |-- arch/
|   |-- concurrency/
|   |-- console/
|   |-- devices/
|   |-- files/
|   |-- filesystems/
|   |-- init/
|   |-- memory/
|   |-- Kernel.nw
|   |-- mkfile
|   |-- network/
|   |-- processes/
|   `-- syscalls/
|-- lib_core/
|   |-- libbio/
|   |-- libc/
|   |-- Libcore.nw
|   |-- libthread/
|   `-- mkfile
|-- lib_graphics/
|   |-- Graphics.nw
|   |-- libdraw/
|   |-- libimg/
|   |-- libmemdraw/
|   |-- libmemlayer/
|   `-- mkfile
|-- lib_networking/
|   |-- lib9p/
|   |-- libip/
|   `-- mkfile
|-- lib_strings/
|   |-- libflate/
|   |-- libregexp/
|   |-- libstring/
|   `-- mkfile
|-- Makefile
|-- mkconfig.pc
|-- mkconfig.pi
|-- mkfile
|-- mkfile-host-Cygwin
|-- mkfile-host-Linux
|-- mkfile-host-macOS
|-- mkfile-host-Plan9
`-- MISC/
    |-- pc/
    `-- pi/

|-- mkfiles/
|   |-- 386/
|   |-- arm/
|   |-- mkdirs
|   |-- mkfile.proto
|   |-- mklib
|   `-- mkone
|-- mkfile-target-pc
|-- mkfile-target-pi
|-- networking/
|   |-- arp/
|   |-- dhcp/
|   |-- ftp/
|   |-- http/
|   |-- ip/
|   |-- mkfile
|   |-- ndb/
|   |-- Network.nw
|   |-- snoopy/
|   `-- telnet/
|-- profilers/
|   |-- iostats/
|   |-- mkfile
|   `-- Profiler.nw
|-- shells/
|   |-- mkfile
|   |-- rc/
|   `-- Shell.nw
|-- utilities/
|   |-- archive/
|   |-- byte/
|   |-- calc/
|   |-- compare/
|   |-- files/
|   |-- mkfile
|   |-- pipe/
|   |-- process/
|   |-- text/
|   |-- time/
|   `-- Utilities.nw
`-- ROOT/
    |-- 386/
    |-- arm/
    |-- lib/
    |-- rc/
    |-- tests/
    `-- usr/
```

Figure 1: https://github.com/aryx/principia-softwarica Layout.
(as generated by `tree -L 2 -F` and spread in 3 columns)

A New Way to Build

Goken `Dockerfile` excerpt	Principia Softwarica `Dockerfile`

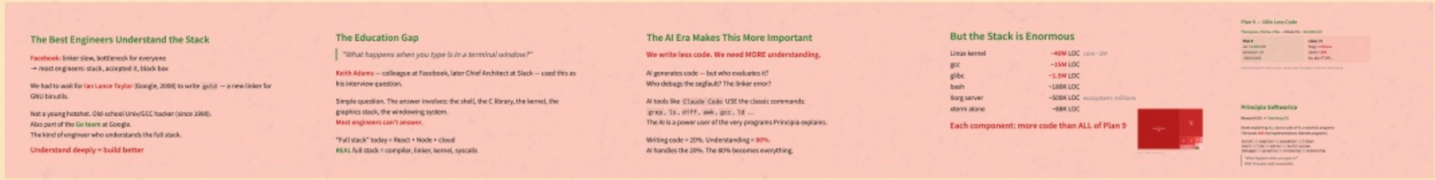
`# Build goken on Ubuntu using`
`# gcc/binutils (and mk/rc)`
`FROM ubuntu:24.04`
 	
`# Setup a basic C dev environment`
`RUN apt-get install -y gcc libc6-dev \`
`    byacc`
 	
`# Now let's build from source`
`WORKDIR /src`
`COPY . .`
 	
`# Small shell script (not GNU autoconf)`
`# to detect arch and generate`
`# ./mkconfig`
`RUN ./configure`
 	
`# The script below obviously builds`
`# 'mk' (without needing mk) but also:`
`# - 'rc', which is called by 'mk'`
`# - 'ed', for the mkenam script`
`#   run during the build`
`RUN ./scripts/build-mk.sh`
 	
`# copy ./ROOT/<arch>/bin/{mk,rc,ed}`
`# to ./bin/`
`RUN ./scripts/promote-mk.sh`
 	
`# make mk and rc accessible`
`ENV PATH="/src/bin:${PATH}"`
`ENV MKSHELL="/src/bin/rc"`
 	
`# Let's build goken (using mk/rc built`
`# in the previous step)`
`RUN mk`
`RUN mk install`
 	
`ENV PATH="/src/ROOT/amd64/bin:\`
`         /src/ROOT/arm64/bin:\`
`         ${PATH}"`
	`# Build principia on Ubuntu`
	`# Linux (amd64 or arm64)`
	`# for 386 (pc) and arm (pi)`
	`FROM padator/goken`
	
	`WORKDIR /principia`
	`COPY . .`
	
	`# 386`
	`RUN cp mkconfig.pc mkconfig`
	`RUN mk && mk install`
	`RUN mk kernel`
	`# arm`
	`RUN cp mkconfig.pi mkconfig`
	`RUN mk && mk install`
	`RUN mk kernel`
	
	`# VFAT tools`
	`RUN apt-get install -y dosfstools \`
	`      mtools`
	
	`# making dosdisk.img`
	`RUN dd if=/dev/zero of=tmp.img \`
	`       bs=1M count=512`
	`RUN mkfs.vfat tmp.img`
	`RUN mcopy -i tmp.img -s -o \`
	`      ROOT/* ::`
	`RUN cat MISC/pc/bootsector tmp.img \`
	`     > dosdisk.img`
	`RUN rm -f tmp.img`

Table 3: Principia Softwarica Build and Packaging Process.

Principia Softwarica

Plan 9 Code Explained

Yoann Padioleau **IWP9**



A Series of Books

Category	Book	Program(s)	LOC	LOE	Pages
Core system	Kernel	9pi	35000	3200	732
	Core libraries	libc libregexp libthread	19000	1600	438
	Shell	rc	6500	1700	166
Development toolchain	C compiler	5c libcc	18500	1900	471
	Assembler	5a	3600	4400	176
	Linker	5l	7500	5400	296
Developer tools	Editor	ed	1600	200	45
	Build system	mk	4350	4050	197
	Debuggers	db acid strace libmach	13100	1000	321
	Profilers	prof time kprof stats iostats	3900	350	102
Graphics	Graphics stack	/dev/draw libdraw libmemdraw libmemlayer	18500	3400	507
	Windowing system	rio libframe libcomplete libplumb	8800	4000	289
Networking	Network stack	/dev/net libip lib9p	18300	4800	457
Misc	CLI utilities	cat ls grep sed diff tar gzip ...	23900	650	493
Total:			182550	36650	4690

Table 1: Principia Softwarica Books and Their Statistics.
(**LOC** = lines of code; **LOE** = lines of explanation; **Pages** = number of typeset pages)

A New Plan 9 Fork

```
.
|-- assemblers/
|   |-- 5a/
|   |-- 8a/
|   |-- data2s.c
|   |-- Assembler.nw
|   `-- mkfile
|-- builders/
|   |-- Builder.nw
|   |-- mk/
|   `-- mkfile
|-- compilers/
|   |-- 5c/
|   |-- 8c/
|   |-- cc/
|   |-- Compiler.nw
|   |-- cpp/
|   `-- mkfile
|-- debuggers/
|   |-- acid/
|   |-- db/
|   |-- Debugger.nw
|   |-- libmach/
|   `-- mkfile
|-- Dockerfile
|-- docs/
|   |-- articles/
|   `-- iwp9/
|-- dosdisk.img
|-- editors/
|   |-- ed/
|   `-- mkfile
|-- include/
|   |-- libc.h
|   |-- ...
|   `-- u.h
|-- linkers/
|   |-- 5l/
|   |-- 8l/
|   |-- Linker.nw
|   `-- mkfile
|-- readme.txt
`-- windows/
    |-- libcomplete/
    |-- libplumb/
    |-- mkfile
    |-- rio/
    `-- Windows.nw

|-- kernel/
|   |-- arch/
|   |-- concurrency/
|   |-- console/
|   |-- devices/
|   |-- files/
|   |-- filesystems/
|   |-- init/
|   |-- memory/
|   |-- Kernel.nw
|   |-- mkfile
|   |-- network/
|   |-- processes/
|   `-- syscalls/
|-- lib_core/
|   |-- libbio/
|   |-- libc/
|   |-- Libcore.nw
|   |-- libthread/
|   `-- mkfile
|-- lib_graphics/
|   |-- Graphics.nw
|   |-- libdraw/
|   |-- libimg/
|   |-- libmemdraw/
|   |-- libmemlayer/
|   `-- mkfile
|-- lib_networking/
|   |-- lib9p/
|   |-- libip/
|   `-- mkfile
|-- lib_strings/
|   |-- libflate/
|   |-- libregexp/
|   |-- libstring/
|   `-- mkfile
|-- Makefile
|-- mkconfig.pc
|-- mkconfig.pi
|-- mkfile
|-- mkfile-host-Cygwin
|-- mkfile-host-Linux
|-- mkfile-host-macOS
|-- mkfile-host-Plan9
`-- MISC/
    |-- pc/
    `-- pi/

|-- mkfiles/
|   |-- 386/
|   |-- arm/
|   |-- mkdirs
|   |-- mkfile.proto
|   |-- mklib
|   `-- mkone
|-- mkfile-target-pc
|-- mkfile-target-pi
|-- networking/
|   |-- arp/
|   |-- dhcp/
|   |-- ftp/
|   |-- http/
|   |-- ip/
|   |-- mkfile
|   |-- ndb/
|   |-- Network.nw
|   |-- snoopy/
|   `-- telnet/
|-- profilers/
|   |-- iostats/
|   |-- mkfile
|   `-- Profiler.nw
|-- shells/
|   |-- mkfile
|   |-- rc/
|   `-- Shell.nw
|-- utilities/
|   |-- archive/
|   |-- byte/
|   |-- calc/
|   |-- compare/
|   |-- files/
|   |-- mkfile
|   |-- pipe/
|   |-- process/
|   |-- text/
|   |-- time/
|   `-- Utilities.nw
`-- ROOT/
    |-- 386/
    |-- arm/
    |-- lib/
    |-- rc/
    |-- tests/
    `-- usr/
```

Figure 1: https://github.com/aryx/principia-softwarica Layout.
(as generated by `tree -L 2 -F` and spread in 3 columns)

A New Way to Build

Goken `Dockerfile` excerpt	Principia Softwarica `Dockerfile`

`# Build goken on Ubuntu using`
`# gcc/binutils (and mk/rc)`
`FROM ubuntu:24.04`
 	
`# Setup a basic C dev environment`
`RUN apt-get install -y gcc libc6-dev \`
`    byacc`
 	
`# Now let's build from source`
`WORKDIR /src`
`COPY . .`
 	
`# Small shell script (not GNU autoconf)`
`# to detect arch and generate`
`# ./mkconfig`
`RUN ./configure`
 	
`# The script below obviously builds`
`# 'mk' (without needing mk) but also:`
`# - 'rc', which is called by 'mk'`
`# - 'ed', for the mkenam script`
`#   run during the build`
`RUN ./scripts/build-mk.sh`
 	
`# copy ./ROOT/<arch>/bin/{mk,rc,ed}`
`# to ./bin/`
`RUN ./scripts/promote-mk.sh`
 	
`# make mk and rc accessible`
`ENV PATH="/src/bin:${PATH}"`
`ENV MKSHELL="/src/bin/rc"`
 	
`# Let's build goken (using mk/rc built`
`# in the previous step)`
`RUN mk`
`RUN mk install`
 	
`ENV PATH="/src/ROOT/amd64/bin:\`
`         /src/ROOT/arm64/bin:\`
`         ${PATH}"`
	`# Build principia on Ubuntu`
	`# Linux (amd64 or arm64)`
	`# for 386 (pc) and arm (pi)`
	`FROM padator/goken`
	
	`WORKDIR /principia`
	`COPY . .`
	
	`# 386`
	`RUN cp mkconfig.pc mkconfig`
	`RUN mk && mk install`
	`RUN mk kernel`
	`# arm`
	`RUN cp mkconfig.pi mkconfig`
	`RUN mk && mk install`
	`RUN mk kernel`
	
	`# VFAT tools`
	`RUN apt-get install -y dosfstools \`
	`      mtools`
	
	`# making dosdisk.img`
	`RUN dd if=/dev/zero of=tmp.img \`
	`       bs=1M count=512`
	`RUN mkfs.vfat tmp.img`
	`RUN mcopy -i tmp.img -s -o \`
	`      ROOT/* ::`
	`RUN cat MISC/pc/bootsector tmp.img \`
	`     > dosdisk.img`
	`RUN rm -f tmp.img`

Table 3: Principia Softwarica Build and Packaging Process.

Take this with you.
Revisit anytime.

Missed something? Want to explore further?
Scan or click below to open this presentation.
Anytime, anywhere.

View presentation