Principia Softwarica: Plan 9 Code Explained

Yoann Padioleau
yoann.padioleau@gmail.com

Abstract

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 Plan 9 programs used by a programmer: the kernel, the shell, the assembler, the linker, the C compiler, the editor, the windowing system, the build system, etc. Each program is covered by a separate book and explained using the *literate programming* technique. Plan 9 thus becomes a teaching operating system for students to learn deeply about programming.

Principia Softwarica is also another fork of Plan 9 available at `https://github.com/aryx/principia-softwarica` that can be easily cloned, cross-compiled, modified, and tested from Linux, macOS, as well as Windows, which is convenient for students.

1 Introduction

Plan 9 didn't have the success it deserved. It was innovative, elegant, powerful, small, and designed by great programmers. Those great minds didn't just rethink the kernel; they rethought the whole operating system (OS) in a holistic manner, with a kernel pushing the "everything is a file" UNIX motto to its limit [PPT+93], an integrated graphic and network stack, a windowing system that could even run under itself [Pik89], a compact cross-compiling toolchain (with assemblers, linkers, C compilers, and debuggers for most architectures), minimalist C libraries, a simpler build system and shell, and more.

What is even more impressive is that those programs contain only on the order of thousands of lines of code (LOC). `rio` for example, the Plan 9 windowing system [Pik00], contains just 8 800 LOC and is arguably more powerful than Xorg (an implementation of X Window [SG86]) which contains millions of LOC. The X Window program `xterm`, which is just a terminal, not the windowing system, contains already 88 000 LOC while `rio` contains a built-in terminal in less than 1000 LOC. The *whole* Plan 9 windowing system uses 10x less code than *one* of the many components of X Window. In fact, with only 183 000 LOC, the Plan 9 authors managed to implement from scratch all the major components of an OS.

However, the Linux/GNU/Xorg OS won over Plan 9, partly because it was the first open-source (OSS) system available. Unfortunately, even if the source

code is open, it is very hard to understand most of this code because each component (e.g., the Linux kernel, the `bash` shell, the GNU C library, `gcc`) contains multiple orders of magnitude more code than Plan 9 with millions of LOC. This situation is sad and potentially dangerous, because most programmers rely on a giant software stack that very few people (if any) fully understand. The situation is even worse for students who don't have any clear path to deeply understand the software stack.

Enter *Principia Softwarica*, a new project to repurpose Plan 9 from an old research OS to a new teaching OS (OS in a general sense: not just the kernel but the whole software stack). Concretely, 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 Plan 9 programs used by a programmer. Each program will be covered by a separate book.

The books not only describe the implementations of those programs, *they are* the implementations of those programs. Indeed, each book 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.

The rest of the article is organized as follows. First, I present the full list of Principia Softwarica books in Section 2 and give a brief overview of literate programming in Section 3. Then, in Section 4 I explain the motivation for a new Plan 9 fork and describes its content and key features while Section 5 introduces the associated new Plan 9 distribution. Finally, I discuss future work in Section 6 and related work in Section 7.

2 The bookset

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 *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, while the compiler is a program that generates other programs. Those programs are also sometimes referred to as *system software*, in opposition to *application software* (e.g., spreadsheets, word processors, email clients), which is not covered by Principia Softwarica.

Table 1 presents the full list of books in Principia Softwarica and the corresponding Plan 9 programs or libraries they document.

When a program involves multiple architectures, such as the C compiler or assembler, I chose to present only the ARM [Sea01] variant of the program, here respectively `5c` and `5a`. The LOC column in Table 1 accounts for the generic part of the code that is architecture independent and the ARM-specific code, but not the code for the other architectures (e.g., x86).

I chose the ARM architecture because it is one of the simplest architectures

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

while also being one of the most widely used architectures in the world. Indeed, almost every phone contains an ARM processor. It is also the processor used in the extremely cheap Raspberry Pi[1], a machine used by many electronic hobbyists. Thus, ARM is a great candidate for my teaching purposes. In fact, the kernel book will describe `9pi`[2], the port by Richard Miller of the Plan 9 kernel to the Raspberry Pi.

As you can see from the LOE column in Table 1, which accounts for the number of lines of explanations, Principia Softwarica is still a work in progress and many books have a low number of LOE. A few books are almost complete such as the assembler, the linker, the windowing system, and the build system, which all have a LOE/LOC ratio close to 1 so this should probably be the target for the other books as well.

The Principia Softwarica programs form together the minimal foundation on top of which all applications can be built. Even though there are many 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. Because the programs I just mentioned inevitably have bugs or non optimal parts, you will 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. As I said earlier, it is hard to remove any of the programs from the series.

3 Literate programming crash course

Source code is the ultimate explanation for what a program does. However, 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 arguably not the best way to explain code. The code and its documentation should be mixed together, as done for instance in the Xinu book [Com84], so one does 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 explain programs (and arguably also to better develop programs). Such

[1] `https://www.raspberrypi.org/`
[2] `https://9p.io/sources/contrib/miller/9pi.img.gz`

documents are called *literate programs*. All Principia Softwarica programs are literate programs.

Table 2 presents a toy literate program on the left and its rendered LaTeX output on the right to illustrate the main features of Noweb [Ram94][3], the literate programming tool used for Principia Softwarica. The `<<...>>=` syntax allows to define a *chunk* that can be referenced before or after its definition in other chunks using the `<<...>>` syntax. Noweb is similar to macroprocessing languages such as `cpp` or `m4`; chunks are the equivalent of macro constants. However, with Noweb one can also define a macro in multiple parts, for example `initializations` in Table 2. Outside code chunks, one can write explanations using LaTeX commands or regular text.

Note that literate programming is different from using API documentation generators such as javadoc[4] or doxygen[5]. Noweb 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 of the Plan 9 kernel, which 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, however Noweb allows to present this structure piece by piece, gradually, in different chapters, as illustrated in Table 2 although on extremely simplified code. One 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 the reader 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, one can show code where the most important parts are highlighted, and where other concerns are hidden and presented later, as illustrated again in Table 2 on a simplified case.

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

[3] Noweb is available at http://www.cs.tufts.edu/~nr/noweb/. The No-web name comes from Knuth's tool `cweb` which processed "a **web** of C chunks", and because Noweb is **no**t language specific like `cweb` but can work for many programming languages, hence the no part.

[4] http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html

[5] http://www.stack.nl/~dimitri/doxygen/

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.

best separation of concerns to make it easier for the reader to understand the code.

Finally, here are the Noweb commands[6] illustrating how to automatically generate the code from the literate program:

```
$ notangle -Rproc.h ToyKernel.nw > proc.h
$ notangle -Rkernel.c ToyKernel.nw > kernel.c
$ cat proc.h
struct Proc {
  int pid;
  Lock l;
  int cnt;
}
```

4 Yet another Plan 9 fork

To create the `.nw` Noweb literate programs out of the original 4th edition Plan 9 code, I first had to copy and split this code in separate functions and data structures. Then, I distributed those functions and data structures in different chapters and sections and further split big functions and data structures in smaller pieces again and again until the code was simple enough that it could be more easily explained. As I was trying to understand the code (in order to explain it in the books), I sometimes renamed unclear entities, removed what appeared to be dead code, or even sometimes fixed bugs[7] while I was trying hard to explain something that could not be explained (because it was actually buggy). I ended up also renaming many C files from Plan 9 and introduced additional C files and header files to better separate concerns.

The result of all this work is available on GitHub in a new Plan 9 fork at: https://github.com/aryx/principia-softwarica. I also reorganized the original Plan 9 hierarchy to make the source of the programs more easily discoverable; it is arguably easier (and faster) to find the code for the assembler in `assemblers/5a/` than in `sys/src/cmd/5a`. Figure 1 presents the toplevel layout of the Principia Softwarica repository. Note that this fork is a subset of Plan 9 with only the code derived from the Principia Softwarica literate programs (no applications) and with only the code to support the ARM and x86 architectures.

The repository contains not only the literate program files (e.g., `Assembler.nw`) but also the generated code (e.g., `assemblers/5a/`) as shown in Figure 1, which seems redundant. This helps though to remove the need for the student to install Noweb to read (or compile) the generated code.

[6] The `notangle` name comes again from Knuth's literate programming terminology in which one "tangles" or "weaves" a web of chunks.

[7] See http://9legacy.org/9legacy/patch/9-newseg-mapsize.diff for an example of bug-fix as well as other entries with my name at http://9legacy.org/patch.html

In fact, the code is not generated by Noweb as said previously but by a separate tool I made called `syncweb`[8] that introduces special start and end *mark* comments in the generated code and remember the `md5sum` of chunk content in a separate auxiliary file as shown by the commands below:

```
$ syncweb -lang C -md5sum_in_auxfile -less_marks ToyKernel.nw proc.h
$ cat proc.h
/*s: proc.h */
struct Proc {
  int pid;
  Lock l;
  /*s: [[Proc]] other fields */
  int cnt;
  /*e: [[Proc]] other fields */
}
/*e: proc.h */
$ cat .md5sum_proc_h
proc.h |e9c8fb1690357b390261c8e53563504f
[[Proc]] other fields |b537e0da3a721d92f61a6f04c33bad6c
```

Thanks to those marks and checksums, one can modify the generated code or the literate program and propagate ("synchronize") automatically the modifications to respectively the literate program or the code. The checksum acts as a "proxy" to date a chunk. Indeed, if one modifies one chunk in a file `proc.h`, and another chunk in the literate program `ToyKernel.nw`, looking just at the dates of those two files it is impossible to know which file has the latest content (actually both files have parts of the latest content). With the chunk checksums saved in the auxiliary file, `syncweb` can recompute the `md5sum`s of the chunks in the modified files (`proc.h` and `ToyKernel.nw`) and detect for each chunk separately which chunk was modified last and from which file, and automatically merge the changes. However, if one modifies the same chunk in both the generated code and the literate program, `syncweb` can instead detect the conflict and prompt the user to manually merge the changes.

The ability to modify either the code or the literate program is a huge help during development and `syncweb` solved one of the biggest complain people had about the use of literate programming in a project.

5 A new Plan 9 distribution

The original Plan 9[9], as well as forks such as `9legacy`[10] or `9front`[11], can easily be tested thanks to downloadable ISO CD images for PCs or bootable SD card images for Raspberry Pi. You can even run Plan 9 virtualized via QEMU[12]

[8] https://github.com/aryx/syncweb
[9] https://9p.io/plan9/download.html
[10] http://9legacy.org/download.html
[11] https://9front.org/releases/
[12] https://www.qemu.org/

```
.
|-- 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)

from Linux, macOS, and even Windows. But what if a student wants to modify the underlying Plan 9 code? How to rebuild and make an ISO CD image or SD card image from this modified code? How to compile the modified code from Linux, macOS, or Windows?

The teaching context of Principia Softwarica made it necessary to create a new way to build and distribute Plan 9 so that a student can easily modify the code, rebuild, and play with it, from whatever mainstream OS. It is not without challenge though. Indeed, building Plan 9 from another OS is not trivial, because Plan 9 is written in a dialect of C with extensions supported only by the Plan 9 C compiler; one cannot use `gcc` or `clang` to compile it. In the same way, Plan 9 also contains assembly code using a syntax that is only supported by the Plan 9 assembler.

Fortunately, thanks to porting work done for Inferno[13], as well as work in plan9port[14], it is possible to *compile* first some slightly modified versions of the Plan 9 C compiler and assembler (where the use of special C extensions have been removed) using `gcc` or `clang` from Linux, macOS, or Windows. Then, one can use those compiled Plan 9 compiler and assembler to *cross-compile* Plan 9 itself. For Principia Softwarica, I use my own fork (again) of code derived from the Inferno toolchain in a new project called Goken[15] (I submitted a separate work-in-progress paper to this workshop describing the motivations and features of Goken), which contains the code of Plan 9 compilers and assemblers that can build on Linux, macOS, or Windows, and either on amd64 or arm64 machines (most modern machines fit the requirements).

Once all the Plan 9 binaries, including the kernel, have been cross-compiled (from Linux, macOS, or Windows), it remains to *package* them together in a form that can be booted and run. With Principia Softwarica, one can build a disk image using the VFAT (DOS) filesystem (see the `dosdisk.img` file in Figure 1), and either use QEMU to run it, or copy the image on a SD card to run on the Raspberry Pi (hence the `mkconfig.pc` and `mkconfig.pi` in Figure 1).

The motivation for choosing VFAT is that it is a filesystem supported by all the mainstream OSes, making it easy for students to explore the generated artifact. This requires to build a kernel with the `dossrv` program embedded in the kernel binary.

The whole build and packaging process for both Goken and Principia Softwarica can be succinctly specified by the two Docker[16] self-explanatory configuration files in Table 3. Thanks to those Docker files, the following commands are then the only commands needed to build from scratch everything and run Plan 9 with QEMU:

```
# --- build goken ---
$ git clone https://github.com/aryx/goken9cc
$ cd goken9cc
```

[13] https://github.com/inferno-os/inferno-os/tree/master/utils
[14] https://9fans.github.io/plan9port/
[15] https://github.com/aryx/goken9cc
[16] https://www.docker.com/

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.

```
$ docker build -t "padator/goken" -f Dockerfile .
$ docker push "padator/goken" # requires credentials via docker login
$ cd ..

# --- build principia ---
$ git clone https://github.com/aryx/principia-softwarica
$ cd principia-softwarica
$ docker build -t "principia" -f Dockerfile .

# --- extract 9qemu and dosdisk.img and run ---
$ docker run -u $(id -u):$(id -g) --rm -v "$PWD:/out" principia \
   sh -c "cp kernel/COMPILE/9/pc/9qemu /out && cp dosdisk.img /out"
$ qemu-system-i386 -smp 4 -m 512 -kernel ./9qemu -hda ./dosdisk.img
```

It is an incredible feeling to modify in `lib_core/libc/9syscall/sys.h` one of the Plan 9 system calls, for instance `#define OPEN 6` to 42, and recompile *everything* from scratch using some of the commands above, and get the compilation done in less than 10 seconds on a modern machine. I have no idea which files in the Linux kernel or GNU C library I would have to modify to have the same effect, and how much time I would need to wait to recompile the Linux/GNU/Xorg OS from scratch.

Note that it is not necessary to use a Docker container to build Principia Softwarica; one can also compile Principia Softwarica on "native" Linux, macOS, or Windows. However, Docker files are a convenient and portable way to specify the commands to run in a fresh environment to build a project. Moreover, the GitHub continuous integration (CI) system GitHub Actions[17] can also run Docker containers; this is used in the Goken and Principia Softwarica repositories to automatically check for build and test regressions after each commit.

6 Future work

As I mentioned earlier in Section 2, an important future work is to add lines of explanations to many of the books in Table 1 to reach a LOE/LOC ratio close to 1.

Another future work is to make Principia Softwarica self-hosted; right now one can build Principia from the mainstream OSes (which is very useful in a teaching context) but not from the produced and booted VFAT image itself.

Finally, it would be cleaner to merge some of the code in the Principia Softwarica and Goken repositories. The code in the two projects was forked from different Plan 9 derivatives, respectively Plan 9 4th edition and Kencc[18] (which itself derived from the Inferno and Plan 9 toolchain), because the motivations and needs for the two projects were different. However, the code could probably now be merged back together.

[17] https://github.com/features/actions
[18] https://code.google.com/p/ken-cc/

7 Related work

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 like in Principia Softwarica.

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

- UNIX V6[19] (Ken Thompson et al.), fully commented in the infamous book by John Lions [Lio77], or its modern incarnation xv6[20], 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[21] (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[22] (Andrew Tanenbaum 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[23] (Noam Nisan and Shimon Schocken) 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.

- Oberon[24] (Niklaus Wirth et al.) is a kernel, compiler, and windowing system designed from scratch. It is a great OS, 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 OS like UNIX (and Plan 9) are more universal; they can run any program in any language, as long as the program can be interpreted or compiled into a binary.

[19] http://minnie.tuhs.org/cgi-bin/utree.pl
[20] http://pdos.csail.mit.edu/6.828/2014/xv6.html
[21] http://minnie.tuhs.org/cgi-bin/utree.pl?file=Xinu7
[22] http://minnie.tuhs.org/cgi-bin/utree.pl?file=Minix1.1
[23] http://www.nand2tetris.org/
[24] http://www.projectoberon.com/

- TempleOS[25] (Terry A. Davis) is an OS 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[26], the Linux kernel[27], and the X Window graphical user interface Xorg[28]. However, GNU/Linux/Xorg together is 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 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.

8 Conclusion

I hope the Principia Softwarica books will greatly consolidate student computer science knowledge and give students a better and more complete picture of what is going on in a computer. Hopefully, it will also give more exposure to the hidden gem that is Plan 9 code; this code is small, elegant, powerful, open source, and was written by incredible programmers who deserve to have their art pieces (their programs) fully exposed.

I hope those books will answer many students' questions, even those that seem very simple at first such as "What happens when the user types `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 involves actually a non-trivial amount of code.

References

[Com84] Douglas Comer. *Operating System Design: The XINU Approach.* Prentice Hall, 1984. cited page(s) 4, 13

[Com87] Douglas Comer. *Operating System Design: Volume II Internetworking with XINU.* Prentice Hall, 1987. cited page(s) 13

[25] http://www.templeos.org/
[26] http://www.gnu.org/
[27] http://www.kernel.org/
[28] http://www.freedesktop.org

[FH95] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995. cited page(s) 13

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 2, 4

[Lio77] John Lions. *Lions Commentary on UNIX 6th Edition, with Source Code*. Peer to Peer Communications, 1977. cited page(s) 13

[NS05] Noam Nisan and Shimon Shocken. *The Elements of Computing Systems*. MIT Press, 2005. cited page(s) 13

[Pik89] Rob Pike. A concurrent window system. In *Computing Systems*, pages 133–154, 1989. . cited page(s) 1

[Pik00] Rob Pike. Rio: Design of a concurrent window system. Technical report, Bell Labs, 2000. . cited page(s) 1

[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. . cited page(s) 1

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, September 1994. cited page(s) 5

[Sea01] David Seal. *ARM, Architecture Reference Manual*. Addison-Wesley, 2001. cited page(s) 2

[SG86] Robert W. Scheifler and Jim Gettys. The X Window system. In *ACM Transaction of Graphics*, pages 79–109, 1986. cited page(s) 1

[Tan87] Andrew S. Tanenbaum. *Operating Systems Design and Implementation*. Prentice Hall, 1987. cited page(s) 4, 13

[WG92] Niklaus Wirth and Jurg Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, 1992. cited page(s) 13

[WR13] Alfred N. Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1910, 1912, 1913. cited page(s) 2