The Good, the Bad, and the Ugly: The Unix™ Legacy

Rob Pike
Bell Labs
Lucent Technologies
`rob@plan9.bell-labs.com`

Copenhagen

Sept 8-9 2001
+1000000000s

1972

‘‘The number of UNIX installations has grown to 10, with more expected.’’

The *UNIX Programmer’s Manual,* 2nd Edition, June, 1972.

2001

The number of UNIX *variants* has grown to dozens,
with more expected.

A definition

What is Unix™?

A generous definition: Those operating systems derived from or inspired by the Research Unix systems of the 1970s.

Includes commercial Unix systems, Research Unix, NetBSD, OpenBSD, Linux, even Plan 9 and others.

Unix Today

Today, Unix runs everywhere (except a few PCs).

What is the secret of its success?

What *is* its success?

Has it fulfilled its promise?

What is its legacy?

Another definition

What makes Unix *Unix*?

Something like the combination of:

- high-level programming language
- hierarchical file system
- uniform, unformatted files (text)
- separable shell
- distinct tools
- pipes
- regular expressions
- portability
- security

Not all these ideas originate in Unix, but their combination does.

The problem is...

All those things were solidly in place in the 1970s!

To get to 2001, you need to add networking and graphics, but those are not *definitive* of Unix. Quite the opposite: these were added later and badly, with models taken largely from other systems.

A networking example

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
    ...
struct sockaddr_in sock_in;
struct servent *sp;
struct hostent *host;
    ...
    memset(&sock_in, 0, sizeof (sock_in));
    sock_in.sin_family = AF_INET;
    f = socket(AF_INET, SOCK_STREAM, 0);
    if (f < 0)
        error("socket");
    if (bind(f, (struct sockaddr*)&sock_in, sizeof sock_in) < 0){
        error("bind");
    host = gethostbyname(argv[1]);
    if(host){
        sock_in.sin_family = host->h_addrtype;
        memmove(&sock_in.sin_addr, host->h_addr, host->h_length);
```

```
}else{
    sock_in.sin_family = AF_INET;
    sock_in.sin_addr.s_addr = inet_addr(argv[1]);
    if (sock_in.sin_addr.s_addr == -1)
        error("unknown host %s", argv[1]);
}
sp = getservbyname("discard", "tcp");
if (sp)
    sock_in.sin_port = sp->s_port;
else
    sock_in.sin_port = htons(9);

if (connect(f, (struct sockaddr*)&sock_in, sizeof sock_in) < 0){
    error("connect:");
```

This feels too clumsy, too detailed, too not-Unix-like.

On Plan 9

```
#include <u.h>
#include <libc.h>
    ...
    fd = dial(netmkaddr(argv[1], "tcp", "discard"), 0, 0, 0);
    if(fd < 0)
        sysfatal("can’t dial %s: %r", argv[1]);
```

Why the difference?

Why isn’t there a `dial` on Unix? (We’ve written one, using sockets, so we can port our code.)

Similar story for graphics.

Something went wrong.

Today

There have been many advances in the last billion seconds, but lately the pace of Unix-like progress has been slow.

In some areas, things have gotten worse.

To understand the present, we must look to the past with an eye to the future.

The Lesson

What makes the system good at what it’s good at is also what makes it bad at what it’s bad at.

Its strengths are also its weaknesses.

A simple example: flat text files.

Amazing expressive power, huge convenience, but serious problems in pushing past a prototype level of performance or packaging.

Compare the famous `spell` pipeline with an interactive spell-checker.

Two questions

1. What is the best thing about Unix?

2. What is the worst thing about Unix?

Answers at the end.

Languages

C is well understood and has aged surprisingly gracefully. (Good)

Still it’s not very modern: (Bad)

- No garbage collection
- No string handling!

And it has some horrible mistakes: (Ugly)

- The preprocessor
- Conditional compilation

We’re stuck with it

C hasn’t changed much since the 1970s. (ANSI C was mostly a codification; C9X hasn’t happened yet.) And — let’s face it — it’s ugly.

Can’t we do better? C++? (Sorry, never mind.)

Perl? Java?

McBreen, *Software Craftsmanship*:

> Java’s long term value has yet to be proven.

C is the desert island language.

Tools

Collection of Good tools, every one doing one job well.

Integration through pipes, also Good.

But there are problems. (Bad)

Choosing which tool to use is a problem for most users. Therefore when one tool came along that did *everything* — Perl (Ugly) — it took over.

Tool approach works only at a particular level; for instance, integration is limited in scope. Again, compare `spell` vs. spell-checkers;

pipes of commands vs. COM;

pushing data into `troff` vs. importing through OLE, etc.

Tools cont’d.

What Unix does well isn’t what people want.

Integration and hand-holding:

Not pipelines, but `emacs` or IDEs or COM.

The tool approach teaches lessons other communities can learn from.

Try to `diff` two PowerPoint docs.

Then think about regular expressions:

One of Unix’s greatest contributions; unused outside.

Almost no visibility beyond the Unix community!

Does your web browser’s Find command use regexps? *Why not*?

Netscape was written on Unix, is still used on Unix, but doesn’t accept even that one key idea!

Lesson: Perhaps people don’t want to think about problem solving this way. Maybe Unix got it wrong. (Even within Unix, the tool approach is losing ground.)

Browsers

Once, the commonest Unix program was `grep`. Today, it's `emacs` or `mozilla`.

People prefer integrated environments and browsers.

The tyranny of forced interaction:

- No ability to step beyond the model.
- No programmability. (E.g.: Fetching maps, NYT)
- Wasting human time.

The irony:

- Most web pages are synthesized dynamically by Unix tools.
- But the result hides that completely from the user.

The weird solution: programs that interpret HTML to drive web interactions automatically.

- Simple examples: `stock`, `weather`.
- Text-only browsing.

Files

Flat text files are easy to use (Good) but don’t scale well (Bad).

The centerpiece of the tool approach, but as we’ve seen, the tool approach has problems.

Drifting back towards typed, binary files (Ugly):

- `cat .` doesn’t work any more (`readdir` vs. `read`).
- Databases to hold code.
- HTTP/HTML interfaces to system services.

File Systems

Hierarchical file system was a huge improvement when Unix first appeared.

Devices as files

Plan 9 pushed this much farther:

    `/proc`, `/net`, `/fd`, `/dev/draw`, etc.

Spectacular ease of networking: total transparency. (Good) (NFS gets this wrong (Bad)).

Some things don't work well this way: e.g. `fork`ing and `exec`ing.

Mail: `upas/fs` is fine example of both strengths and weaknesses.

- Break mailbox into files
- MIME as hierarchy
- Attachments appear as `.gif` files, etc.
- Much overhead, too many file-system interactions.

Portability

Unix did portability first and, still, best: (Good)

Sources in high-level language

Operating system runs independent of hardware

Interfaces don’t change (much) between platforms.

Unintended consequences: (Bad)

Machine became irrelevant

Therefore cost was only factor

Therefore hardware got lousy, widely variable

Therefore software had to cope, became non-portable.

The success of PCs is in large part due to the fact that, by making all hardware equivalent, good software enabled bad hardware. (Ugly)

Portability cont'd.

Unix's record of portability is mixed.
There are a zillion versions of the system, and widely variable interfaces, leading to:

`#ifdef`
`config`

Abominations that are largely unnecessary and actually *reduce* portability.

Byte order

Don't `#ifdef BIG_ENDIAN`, write code that works independently of byte order!

Proof of concept: The entire Plan 9 source tree, including kernel, libraries, and applications, compiles for all architectures from the same sources with no `#ifdefs`.

Now *that's* portability.

Security

Unix was an early example of thinking about Good security.

Encrypted passwords.

Papers by Ritchie, Thompson and Morris.

Note: user interface didn’t change.

Poor things happened too, e.g. root (Bad) and set-uid (Ugly, and the only patent!). But overall Unix made computing more secure.

Somewhere, things got inverted. (Bad)

Security community now worries about security, not users.

User interface makes security unworkable.

Compare `https:` (Good) to `ssh`. (Ugly.)

`Ssh` fails to connect if there’s no shared protocol. How can there be no shared protocol??

The Interface to Security

Weak security that’s easy to use will help more people than strong security that’s hard to use. E.g.: door locks.

Tip: *User interface is more important than security*.

Bad user interfaces drive people away from security.

Weak security is *much* better than none at all.

Community

Unix has always been available in source code form.

Shared source leads to great progress (Good) but lack of direction (Bad) and incompatible variants (Ugly).

The people in this room represent a single community, yet how many different Unix variants are running on the laptops here?

The lesson we never learned

Microsoft succeeds not because it’s good, but because there’s only one of them.

Division leads to wasted effort and political infighting, limiting the success.

Past:

- 1970s: Variant academic versions:
  Bell Labs, Berkeley, Toronto, Sydney.
- 1980s: Variant commercial versions:
  System V, Xenix, SunOs, DEC Unix, ...
- 1990s: Variant open versions:
  Linux, NetBSD, OpenBSD, Plan 9, ...

Future:

- Unixes of the World, Unite!

Conclusion

The Good things about Unix are also its Bad things. The things that made it successful also limited its success.

C
Tools
Text files
Portability
Open source

The irony is Ugly.

For the next billion seconds, let’s find ways to remove the limitations without losing sight of the original vision.

Two Answers:

1. What is the best thing about Unix?
A: The community.

2. What is the worst thing about Unix?
A: That there are so many communities.