The operating system: why there should be one
(i.e. more than zero)

Stephen Kell

stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge
“An operating system is a collection of things that don’t fit into a language. There shouldn’t be one.”
Smalltalk: a partisan summary

- a programming abstraction
  - objects
  - messages
- a *descriptive* abstraction a.k.a. metasystem
- late binding, pervasively
- uniformity
- big, well-designed libraries
- (… never full enough to replace OS)
Unix: a partisan summary

“Since we are programmers, we naturally designed the system to make it easy to write, test, and run programs.”

- programming abstractions (multiple)
- primitive meta-abstraction for I/O
- late binding in some cases

Missing:

- semantic meta-abstraction (cf. classes)
- uniformity ...
What happened next
Unix achieved world domination

- all modern OSes are Unix-like
- Smalltalk-like systems relegated to running under Unix

Programmers work with fragmented infrastructure

- primitive baseline (byte streams, “plain text”)
- sharing code and data is painful
- integration, migration, communication, “legacy”, ...
- little dynamic inspection and experimentation
Three messages
Unix “file” and Smalltalk “object” naturally converge

- can extrapolate trends in Unix design

The real differences concern diversity and plurality

- the modern versus the postmodern

A way forward is apparent if we look

- metasystem-like facilities abound in modern Unix
- Smalltalk family is coming around to Unix-like ideas
- gradual, compatible evolution is possible
- unifying around a language isn’t
A problem with Unix (1)

$ grep −r 'pattern' $d

■ what kind of thing is $d?
■ what is grep -r going to do with it?
■ what stops it from generalising further?
A problem with Unix (2)

$ grep -r 'pattern' $d

- $d$ is a tree node directory
- grep -r will
  - recursively explore readdir()
  - on leaf nodes, read characters read bytes
  - if match, output a reference filename
- what stops it from generalising further?
  - not all atoms can be named/accessed as files
  - not all compounds can be readdir()’d
  - what should the bytes be?
Unix “file” over time

A process of generalisation

- in the beginning: programs, data, devices
- version 3: IPC channels, a.k.a. pipe()
- BSD 4.2: network sockets, memory objects

Ditto for collections of files (filesystems)

- /proc filesystem (AT&T Eighth Edition)
- kernfs (4.4BSD, c.1994)
- Plan 9: “all shared resources are filesystems”
From file to object

- Unix (original)
- Unix (v7+, BSD, SunOS, ...)
- Plan 9
- Smalltalk

semantics

time
Plan 9: a partisan summary

“build a Unix out of little systems, not a system out of little Unixes”

- Unix-inspired research OS from Bell Labs
- use filesystem abstraction pervasively
- most applications provide and consume “file servers”
- simplifies binding → greater compositionality!
- still missing full metasystem, large/small unification,
  …
Object = file

File(server)s and objects are

- some subdivision of a computation
- implementing a basic protocol…
- consisting of messages (network-transparent)
- … onto which additional meaning can be layered

Newly missing

- documented semantics for files!
- e.g. control files
- what can I `grep`? `cp -r`?
What else might we recursively pattern-search?

$ grep −r ’pattern’ /usr/share/doc/grep
$ ls /usr/share/doc/grep
AUTHORS NEWS.gz README THANKS.gz TODO.gz copyright

$ grep −r ’pattern’ imap://stephen@xylophone/
grep: imap://stephen@xylophone/: No such file or directory

What happened to compositionality?
  ■ hit the limit of Unix (semantics-free) byte-streams
  ■ hit the limit of Unix object model (binding model)
The importance of being meta

usefulness of a metasystem

semantics

Smalltalk

p.17
“An operating system is a collection of things that don’t fit into a language. There shouldn’t be one.”

Specific grumbles:

- “depart from an otherwise consistent framework”
- “very primitive”
- memory, storage, display, input, debugging. . .
The real difference between Unix and VMs

In Smalltalk’s “integrated environment”... there is little distinction between the compiler, interpreter, browser and debugger, [all of which] cooperate through shared data structures.... Pi is an isolated tool in a [Unix] “toolkit environment” [and] interacts with graphics, external data and other processes through explicit interfaces.

T.A. Cargill
**Pi: a case study in object-oriented programming**
OOPSLA ’86
Plurality and Unix

Unix has always favoured plurality

- “toolkit” – a plurality of tools
- plurality of implementations – portability!
- shells, commands etc. are just files
- multiple compilers, debuggers, etc.
- downside: fragmentation

- *plurality of programming languages*

Smalltalk achieves uniformity through monism
Plurality creates fragmentation

Unix has too many binding mechanisms!

- file descriptor
- named file (early or late)
- numbered process (signals, \texttt{nice}, \ldots)
- (from BSD 4.2) sockets\ldots
- in-process linkage
- shared memory $\times 2$ \ldots

Processes cannot fully emulate files

- even with pipes!
The lurking metasystem
Messages / objects are ubiquitous, but also metasystems:

■ /proc filesystem, vmmmap, …
■ other synthetic filesystems (/sys, sometime /dev)
■ DWARF debugging
■ \{symbol, version, type\} info in code representations
■ /etc/services
■ filename extensions, MIME. . .
■ HTTP Content-Encoding
■ “any sufficiently complex system” (pacmd demo)
**ABSTRACT**

We identify three design principles for reflection and metaprogramming facilities in object-oriented programming languages. Encapsulation: meta-level facilities must encapsulate their implementation. Stratification: meta-level facilities must be separated from base-level functionality. Ontological correspondence: the ontology of meta-level facilities should correspond to the ontology of the language they manipulate.

Traditional/mainstream reflective architectures do not follow these precepts. In contrast, reflective APIs built around the concept of mirrors are characterized by adherence to these three principles. Consequently, mirror-based architectures have significant advantages with respect to distribution, deployment and general purpose metaprogramming.

**Categories and Subject Descriptors**

D.3.2 [Language Classifications]: Object-oriented languages.

**General Terms**

Design, Languages.

**Keywords**

Reflection, Metaprogramming, Mirrors, Java, Smalltalk.
Three principles

■ encapsulation
  \( \approx \) plurality of implementation

■ stratification
  \( \approx \) optionality / separability

■ ontological correspondence
  \( \approx \) preservation of source features

Unix debugging has all three!

■ \dots \ had them since decades earlier

■ fundamentally different approach to debugging

■ pluralist and descriptive, cf. monist / prescriptive
```plaintext
$ cc -g -o hello hello.c && readelf -wi hello | column

<table>
<thead>
<tr>
<th>TAGCompile_unit</th>
<th>TAG_pointer_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT_language : 1 (ANSI C)</td>
<td>AT_byte_size: 8</td>
</tr>
<tr>
<td>AT_name : hello.c</td>
<td>AT_type : &lt;0x2af&gt;</td>
</tr>
<tr>
<td>AT_low_pc : 0x4004f4</td>
<td>AT_name : main</td>
</tr>
<tr>
<td>AT_high_pc : 0x400514</td>
<td>AT_type : &lt;0xc5&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TAG_base_type</th>
<th>TAG_subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT_byte_size : 4</td>
<td>AT_low_pc : 0x4004f4</td>
</tr>
<tr>
<td>AT_encoding : 5 (signed)</td>
<td>AT_high_pc : 0x400514</td>
</tr>
<tr>
<td>AT_name : int</td>
<td>AT_name : main</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TAG_pointer_type</th>
<th>TAG_formal_parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT_byte_size : 8</td>
<td>AT_type : &lt;0xc5&gt;</td>
</tr>
<tr>
<td>AT_type : &lt;0x2b5&gt;</td>
<td>AT_low_pc : 0x4004f4</td>
</tr>
<tr>
<td>AT_name : argc</td>
<td>AT_high_pc : 0x400514</td>
</tr>
<tr>
<td>AT_type : &lt;0xc5&gt;</td>
<td>AT_location : fbreg - 20</td>
</tr>
<tr>
<td>AT_location : fbreg - 32</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TAG_base_type</th>
<th>TAG_formal_parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT_byte_size : 1</td>
<td>AT_type : &lt;0x7ae&gt;</td>
</tr>
<tr>
<td>AT_encoding : 6 (char)</td>
<td>AT_location : fbreg - 32</td>
</tr>
<tr>
<td>AT_name : char</td>
<td>p.27</td>
</tr>
</tbody>
</table>
```

```
“Found” Smalltalk

“build a
Smalltalk out of lots of fragments,
not a big fragmented system
out of little [insert PL here]s”

Smalltalk is one view among many!

■ languages more \{opinionated, ephemeral\} than OSes
■ languages should be “a view on the wider world”
■ (incompatible: current approaches to implementation!)

Languages are “opinionated”; for an OS, would be bizarre
Evolving Unix

Join the dots to join the metasystems…

■ via minor expansions to (Unix-like) OS interfaces?
■ unify the metasystem, not the system

It’s too late to unify binding mechanisms, but…

■ if you can’t unify mechanisms, “multiply”
■ large/infinite namespaces permit this
Multifying IPC

fopen()
connect()
popen()
dlopen()
shmat()...

creat()
bind(); listen()
ld -o ...
ld -shared -o ...
shmget()...
Multifying IPC

fopen() ← creat()
connect() ← bind(); listen()
popen() ← ld -o ...
dlopen() ← ld -shared -o ...
shmat()... ← shmget()...
Multifying IPC

- fopen()
- connect()
- popen()
- dlopen()
- shmat()...
- creat()
- bind(); listen()
- ld -o ...
- ld -shared -o ...
- shmget()...
A metasystem for large and small alike

Project of mine: liballocs

$ LD_PRELOAD=liballocs.so ./mycmd

- use /proc, ELF, DWARF...
- ... to build a metasystem for in-memory data
- exposes an API for in(tro)spection
- longer-term: also describe files
Conclusions

Unix and Smalltalk have more in common than first appears

- “simple” programming environments
- … built on unifying abstractions (that converge!)

Unix unwittingly secured its own longevity

- by being unopinionated!
- by accommodating diversity (including of language)

We can think of Smalltalk not as a grand design

- … but as something we aspire to (re)discover
- including in Unix

Please ask questions!
A possible provide/require metasystem for IPC

```c
fd = open(leaf_path, mode);
// then read bytes; hope they're chars

dirfd = open(branch_path, mode);
// then read dirents; fail if not dir!
```
A possible provide/require metasystem for IPC

```c
fd = open(leaf_path, mode);
// then read bytes; hope they’re chars

dirfd = open(branch_path, mode);
// then read dirents; fail if not dir!

open(leaf_path, flags, TEXT_UTF8);
// read bytes; declare that they *will be treated as* UTF-8

open(branch_path, flags, DIRECTORY);
// read dirents, encoding whatever the tree *happens* to be
```

What is the structure of the rightmost arguments?

- something like DWARF, but for files too
Concrete suggestion #3

- fopen()
- connect()
- popen()
- dlopen()
- shmat()
- creat()
- bind(); listen()
- ld -o ...
- ld -shared -o ...
- shmget()

File / socket / program / library namespaces...

- ... are large/infinite
- → can be extended transparently

Needn’t be $n^2$ underneath

- everything is really a dlopen() (ask me)

Sensible behaviour requires meta-info on files/sockets/...

p.34
Late binding in Unix

stdin, stdout, stderr

- fully interposable (a.k.a. redirectable)!

/path/to/some/file

- not interposable: fopen("/path/to/some/file", "r")
- interposable: fopen(filename_supplied_by_user, "r")
- hacks for fixing the first case
Valiant attempts

- stdin, stdout, stderr
- pipes
- popen()
- mmap()
- fmemopen()

Mechanisms are okay; what abstraction can cover them?
The Right Thing vs Worse Is Better

“Many of the problems of composition in modern Unices are really due to misuse and hacks more than anything else.”

- but “doing it right” is too much to ask!
- foresight is not 20/20
- reinterpret the hack once you’ve understood it
- generalise after, not before (Unix vs Multics)
Concrete suggestion #4: programmatic integration
Make integration a job for synthesis or DSLs…

![Diagram](A) ![Diagram](B) ?

… not manual coding!

- needs *provides* and *requires* meta-info
- disclosure: this was my PhD topic…

By expanding OS-level facilities, we can do more and better

- including the inter-process case
Composition in Plan 9

“A system could import a TCP stack to a computer that didn’t have TCP or even Ethernet, and over that network connect to a machine with a different CPU architecture, import its /proc tree, and run a local debugger to do breakpoint debugging of the remote process. This sort of operation was workaday on Plan 9... such things fell out of the design.”

Still missing:

- general-purpose metasystem
- parity of small vs large objects
- late binding by default