"The Unix Philosophy"

Dr. Michael S. Brown
Associate Professor
School of Computing
NERD!

http://www.youtube.com/watch?v=dFUIAQZB9Ng
One word summary of Unix?

Simplicity

"UNIX is very simple, it just needs a genius to understand its simplicity."

Dennis Ritchie (Unix co-creator)
Historical Perspective

• First generation of computers (1930-1950)
  – Only a handful of computers in existence
    • E.g. Zuse Z3, Colossus, Harvard Mark 1, ENIAC
  – These computers are glorified calculators
  – Run “programs” in batches
  – Almost all are “government” sponsored machines
Historical Perspective

• Second gen computers (1950-1960)
  – Commercial Computers
    • IBM, Remington Rand, Burroughs, Honeywell
  – Mainly used for calculation and statistics
  – Very expensive

• But things are happening
  – Transistors replace vacuum tubes
  – Disk storage, printers being developed
  – High level languages developed
    Cobol (Common Business-Oriented Language), Fortran (Formula Translator)
Historical Perspective

• Third Generation (1960-1970)
  – More players enter the market
    • Bell Labs, GE, DEC, IBM, HP, Data General, Commodore
  – Uses are going beyond calculations
    • Ivan Sutherland introduces GUI, Graphics
    • Computers are used for Text Editing, type-setting
History of “Unics”

The “Fathers” of Unix

Ken Thompson  
Dennis Ritchie

Engineers at Bell Labs, 1960s
Multics

- **Multiplexed Information and Computing Service**
- A time-sharing OS started in 1964
- Partners: MIT, GE, Bell Labs

- Multics
  - Was growing too complicated and convoluted
  - Bell exited the project in 1969
  - Ken and Dennis worked on this project
  - Ken and Dennis wanted to make a simpler version
    - While working on Multics, they had already started, working on a “small” PDP-7 and experimenting with notions of processes, command-line interpreters, hierarchical file system, and utility programs
Unix (Unics)

• Uniplexed Information and Computing Service (Unics)
  – Some people have purported the Unix was originally a play on Multics, i.e “Unics”
  – Dennis Ritchie states this is a myth, the term was Unix from the beginning . . however, you do wonder how they go to Unix?

• Driving idea of Unix
  – Portable
    • OS up to this point were wedded to the computer used
    • They wrote Unix in C, with minimal assembly code
  – Multi-tasking, multi-user
  – Simple and light-weight OS (Kernel)
  – Use of “flat” plain text-files
  – Hierarchical file system
  – Large number of “software tools”
From the men themselves

http://www.youtube.com/watch?v=7FjX7r5icV8
“Modern” OS features proposed in Unix

(1) Processes
(2) Device/Files Unification
(3) Shell (and pipes)
(4) File system
Multics

• Many of the ideas in Unix, were inspired by Multics
  – Processes
  – Hierarchal file system
  – “Virtual Memory”

• The problem with Multics?
  – The system was too complicated!
    • Tried to do everything, to be too versatile, too flexible, and failed
  – One of the driving philosophies in Unix was to “keep it simple”
Unix Design

Kernel

Controls the systems.
Main tasks:
Process management
Device communication
File system

Shell

User interface to communicate with the system.
Includes a set of basic shell-utilities. Facilitates scripting.

Utilities

The largest set of Unix. Tools developed to assist the user in their tasks, e.g. text editor, plotting, etc..

$ man “kernel”

$ man “shell”

$ man “utilities”
(1) Process Management

- Process management
- Process is an executing program
  - Its code+data
- Each process has a unique ID (PID)
- Multiple users can run multiple processes at once
  - Time-sharing, multi-user system
- Processes have a hierarchy (parent-child)
Unix Processes

• **Process components**

• Kernel manages this information

• Each process gets a small time on the CPU, in a round robin fashion (time-sharing)

```
Unique ID (PID)
Parents ID (PPID)
Code (instructions)
Machine registers
Global data
Run-time data (stack)
Open files (file descriptors)
Environment variables
Credentials for security
```
Process Creation

• A process can create a new process

• Creation strategy?
  
  – fork()
    • makes an exact copy of the process that calls fork()
  
  – exec*()
    • loads a program image to reply current process code
Fork()

Returns a new PID: e.g. pid == 5

Parent

pid = fork()

Child

pid == 0

Data

Shared Program

Data Copied
Fork() + Exec()

Initial process

Fork

fork() returns pid=0 and runs as a cloned parent until execv is called

Returns a new PID

Original process Continues

New Copy of Parent

new_Program (replacement)

exec(new_program)
Process Hierarchy

- **init** – the “mother” process
- **getty** – login process

```
OS kernel
  ↓
Process 0
  ↓ (sched)
Process 1
  ↓ (init)
  ↓
  ↓
  ↓
getty
  ↓
  ↓
login
  ↓
csh
getty
  ↓
  ↓
login
  ↓
bash
```
(2) File/Device Unification

• Another Unix idea was the use of a single (file) interface for both files, devices, and even process communication

• Files
  – Can read and write characters or “blocks” of data

• Devices
  – Character devices (read characters, not “seekable”)
    • /dev/pts10
  – Block devices (read blocks of data, “seekable”)

• Pseudo Device
  – /dev/zero (produces zeros)
  – /dev/null (ignores all input, has no output)
  – /dev/random (produces a stream of random output)

• In Unix, files, devices all are interfaced the same
  – This makes it very convenient for writing code
(3) Unix Shell

• Command Line interpreter
  – Is just another program
  – BUT, it can be used to tie many programs together
    • Thanks to process creation and hierarchy
      – Parent of all processes in a shell are the shell!
  – Also, allows “shell scripting”
    • Sequential execution of shell commands
    • Evolved into full programming language

It is surprisingly easy to write your own basic shell using fork+exec. (Even pipes isn’t too hard)
(3) Unix Shell

• Common Shell Commands

- `sort` Sorts lines in ascending, descending and unique order
- `grep` Searches for regular expressions in strings or files
- `basename` Strips the path from a path string to leave just the filename
- `dirname` Removes the file from a path string to leave just the pathname
- `cut` Chops up a text string by characters or fields
- `wc` Count the characters, words, or lines
- `[ (test) ]` Predicate or conditional processor
- `tr 'a' 'b'` Transform characters
- `expr` Simple arithmetic processor
- `bc` Basic Calculator
- `eval` Evaluate variables
- `echo` Output strings
- `date` Create date strings
- `nawk` Manipulate text strings
- `head | tail` Access lines in files

• Flow control

if, for, while, case, basic math
Pipes

- Unix also introduced the notion of “pipes”

```
cat file.txt | wc -l > output.txt
```

Pipes was a powerful way to link the `stdout` of a program to the `stdin` of a other.

This is the cornerstone of Unix getting processes to “work together”.
(4) Unix File System

• Directory Structure
  – Tree-based structure
  – Allows “soft” and “hard links”
  – Some Unix systems allow graph structure (i.e. cycles, this can be dangerous)

• Directory Structure is well known
  – Unix systems are standard
  – Files kept in standard directories
    /usr/bin /sys/bin etc . . .
Unix File System

Boot Block | Super Block | I-nodes | Data Blocks

$ man “boot block”
Boot strap code to get the system booted.

$ man “super block”
Meta-data about the file system, hard drive, tuning parameters.

$ man “inodes”
Meta-data (permissions), and link to data blocks. This is the “file”

$ man “data”
The actual data!

Modern systems allow file sizes up to 8 Zettabytes

<table>
<thead>
<tr>
<th>Unit</th>
<th>Value 1</th>
<th>Value 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>kilobyte (kB)</td>
<td>$10^3$</td>
<td>$2^{10}$</td>
</tr>
<tr>
<td>megabyte (MB)</td>
<td>$10^6$</td>
<td>$2^{20}$</td>
</tr>
<tr>
<td>gigabyte (GB)</td>
<td>$10^9$</td>
<td>$2^{30}$</td>
</tr>
<tr>
<td>terabyte (TB)</td>
<td>$10^{12}$</td>
<td>$2^{40}$</td>
</tr>
<tr>
<td>petabyte (PB)</td>
<td>$10^{15}$</td>
<td>$2^{50}$</td>
</tr>
<tr>
<td>exabyte (EB)</td>
<td>$10^{18}$</td>
<td>$2^{60}$</td>
</tr>
<tr>
<td>zettabyte (ZB)</td>
<td>$10^{21}$</td>
<td>$2^{70}$</td>
</tr>
<tr>
<td>yottabyte (YB)</td>
<td>$10^{24}$</td>
<td>$2^{80}$</td>
</tr>
</tbody>
</table>
Unix Spreads Quickly

- 1972, Unix ported to C language
  - Now, entire OS is almost all in C, only small amount in assembly needs to be modified for a new platform
- Bell (a telephone company) was not allowed to enter the computing market
  - Ken Thompson started sending copies to anyone how asked, full distribution 10MB
- Bell became AT&T, licensed Unix to universities, commercial firms
- AT&T releases System 7+ Unix commercially, 1979+
- Berkeley Software Distribution (BSD) starts its own Unix, 1977
Linus Torvalds

• 1991, Linux is developed for PCs

Finnish student working on “Minix” (a educational only mini-unix by Andrew Tanenbaum), he decided to write his own Unix.

Linux, released under the GNU license.

The rest is History. . .

$ man “Linus”

Masters in CS, U. Helsinki
(Master thesis? LINUX)
Now, only around 2% of existing Linux kernels are from his original code.
His net worth is $20million, due to stock options from Red Hat, etc. .
Now a US citizen.
The many flavors of Unix
The Unix Culture

• Unix created a culture
• People wanted a better, more efficient, computing system
• These people were logical and pragmatic in their thought
• They laid the foundation for:

“The Unix Philosophy”
According to Douglas McIlroy, part of Unix development team at Bell Labs:

(1) Write programs that do one thing and do it well.

(2) Write programs to work together.

(3) Write programs to handle text streams, because that is a universal interface.
Principals and Practices

- Rule 1. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.

- Rule 2. Measure. Don't tune for speed until you've measured.

- Rule 3. Fancy algorithms are slow when $n$ is small, and $n$ is usually small. Fancy algorithms have big constants. Until you know that $n$ is frequently going to be big, don't get fancy. (Even if $n$ does get big, use Rule 2 first.)

- Rule 4. Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures.

- Rule 5. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. **Data structures, not algorithms, are central to programming.**

Doug McIlroy

$\text{man "Pike"}$

Education cannot be determined
Bell Labs, Developed early GUI
terminal for Unix in 1982 (not X)
Now at Google
(Did not receive the 1980 Olympic Medal for Archery)
Some selective rules

(http://www.faqs.org/docs/artu/ch01s06.html)
Rule of Modularity

Write simple parts connected by clean interfaces.

Software can be very complicated, especially for large systems. It’s even hard for the programming to cope with the complexity.

Take a modular approach and build many small, but simple components.

These components work together through simple and clean interfaces.

**Object oriented programming uses this concept too.**
Rule of Clarity

Clarity is better than cleverness.

Write code for humans to read, first!

```c
void primes(int m, int t, int c) {
    int i, j;
    i = t / m;
    j = t % m;
    (i <= 1) ? primes(m, t+1, c) : (!j) ? primes(m, t+1, j) : (j == i && !c) ?
            (printf("%d\t", i), primes(m, t+1, c)) : (j > 1 && j < i) ?
            primes(m, t+1, c + !(i & j)) : (t < m * m) ? primes(m, t+1, c) : 0;
}

int main(void) {
    primes(100, 0, 0);
}
```
Rule of Composition

Design programs to be connected with other programs

THINK PIPE!

Stick with text streams for input and output.

Avoid the use of user input (for example, prompt for “yes”), use command line flags instead.

To make programs composable, make them independent. A program on one end of a text stream should care as little as possible about the program on the other end.
Rule of Simplicity

Design for simplicity; add complexity only where you must.

The notion of “intricate and beautiful complexities” is almost an oxymoron. Complexity leads to bugs.

Keep programs as simple as possible. Sometimes less is more.

“Worse” is “better”.

If other processing is required, consider writing another programming and exploiting composibility . . i.e. piping.
Rule of Transparency

Design for visibility to make inspection and debugging easier.

Debugging dominates development. Unix is made to have programs work together. Make it easy to understand how your program works (i.e. make it transparent).

For a program to demonstrate its own correctness, it needs to be using input and output formats sufficiently simple so that the proper relationship between valid input and correct output is easy to check.

A good ‘manual page’ with examples also helps!
Rule of Robustness

Robustness is the child of transparency and simplicity.

Program should work well. Minimize when it fails, and when it does fail, fail gracefully.

Consider situations, just as accepting empty lists/strings/etc., even in places where a human would seldom or never supply an empty string, avoids having to special-case such situations when generating the input mechanically.
Rule of Lease Surprise

In interface design, always do the least surprising thing.

The easiest programs to use are those that demand the least new learning from the user.

Stick with convention (+ means add).

Consider your audience.
  Program used by sys administrators may have different common practices than a program used by programmers.

Avoid excessive features, novelty.
Rule of Silence

When a program has nothing surprising to say, it should say nothing.

I think that the terseness of Unix programs is a central feature of the style. When your program's output becomes another's input, it should be easy to pick out the needed bits. And for people it is a human-factors necessity — important information should not be mixed in with verbosity about internal program behavior. If all displayed information is important, important information is easy to find.

-- Ken Arnold

Ken Arnold

$ man “McIlory”
BS Berkeley
Contributor to the original BSD Unix distribution.
Also wrote the game “rouge”.
Rule of Repair

Repair what you can — but when you must fail, fail noisily and as soon as possible.

Basically, if it is something minor, fix it (or consider handling it), but if you “fail” then make sure you cause a detectable failure to the other programs.

Also, if possible, make it clear why it failed.
Rule of Optimization

Prototype before polishing. Get it working before you optimize it.

“90% of the functionality delivered now is better than 100% of it delivered never!”
- Kernighan & Plauger
  (Kernighan is co-author of the C Programming Book)

“Premature optimization is the root of all evil”
– Donald Knuth (Computing “Grandfather”)

Brian Kernighan  Don Knuth
Unix Influence

• Apple OS
  – Built on BSD

• Chrome/OS
  – Google has hired many of the Unix people

• Unix inspirations
  – Xinu (Xinu Is Not Unix)
    Extremely light-weight OS for embedded systems
Unix Usage Today

• Hollywood (Lucas Film, ILM)
  – Processing ray-tracing renderings
  – Scripting

• Bank Industry
  – Database manipulation
  – Report creation
  – Scripting

• Super computers/clouds
  – Light-weight OS, better utilities resources than Windows
Still need windows?

• Try cygwin
  Pseudo-Unix for Windows
  Provides a great deal of functionality of Unix, on your Windows machine
Summary

• When Unix first came out, it represented a huge leap in OS design
• Established many of the modern ideas used today
• Windows has slowly copied these ideas
• There is always room for improvement
  – Don’t become complacent
  – Cloud computing, very light-weight systems
• Maybe Google . . . .