Unix

CS380L: Mike Dahlin

September 4, 2008

Happily, all of this mechanism meshes very nicely...

D. M. Ritchie and K. Thompson. "The UNIX Time-Sharing System." p. 372. Communications of the ACM*, 17(7), July 1974, pp. 365–375.*

1 Preliminaries

1.1 Review

- THE
  - System structure
    1. Strict hierarchical structure
    2. Sequential processes
    3. Virtual addressing
  - Pearls of wisdom

1.2 Outline

- Unix
  - Theme: Simplicity and elegance
  - File I/O
  - Process management
  - Lessons

1.3 Preview

- Historical perspective: UNIX, HYDRA
- Structure/extensibility: Synthesis, Exokernel, Disco
- Concurrency: Mesa, threads
- Communication: RPC, Active messages, duality of memory and communication

2 Theme: Simplicity and elegance

- The "Third system"
  - Multics: The "second system effect"
    - Visionary
    - Ungainly
  - Unix
    - "Elegance, taste, craftsmanship"
    - Minimum functionality and implementation, yet...
    - Power, and...
    - The pieces fit together seamlessly
    - How many times can one say: "this is just totally obvious"
    - This paper nearly summarizes undergrad OS...if you *understand* it, you understand most of the basics of modern OS's
  - Great companion reading: B. W. Lampson. "Hints for Computer System Design." *Proc. of the 9th ACM Symposium on Operating Systems Principles*, October 1983, pp. 33-48.
    - *"Do one thing at a time, and do it well. An interface should capture the minimum essentials of an abstraction. Dont generalize; generalizations are generally wrong."*

3 File I/O

- Hierarchical name space
  - *strict* hierarchy across directories
  - Can you imagine cases where non-hierarchical structure (e.g., multiple paths to a directory) would be useful? Sure.
  - Basic mechanism could be extended to allow non-hierarchy? Sure.
  - QUESTION: What would get much more complex if you allow non-hierarchy?
  - Disallowing multiple links to directories →
    - Easier search
    - **Easier garbage collection** – no cycles
  - Engineering "taste" – give up a tiny bit of generality for a big savings in complexity
- Directories are files
  - Review: what is an inode? What is in an inode?
  - What is a directory? What is in a directory?
  - How do I find the inumber for file /foo/bar?
  - How do I find the inode for inumber 49824?
  - How do I read the third block of file /foo/bar?

  - What is the difference between a hard link and a symbolic link?

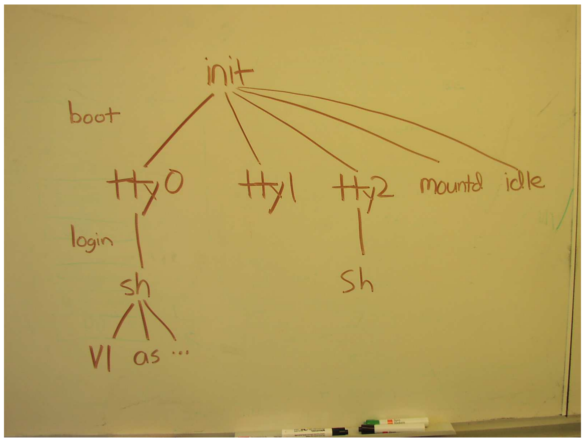
- Untyped data
  - *The structure of files is controlled by the programs which use them, not by the system.* (p. 366)
  - Memory also "untyped": *Another important aspect of programming convenience is that there are not "control blocks" with a complicated structure partially maintained by and depended on by the file system or other system calls. Generally speaking, the contents of a program's address space are the property of the program, and we have tried to avoid placing restrictions on the data structures within that address space.* (p. 374)
- Byte-oriented I/O
- Device-independent I/O
  - Special files
  - *There is a threefold advantage in treating I/O devices this way: file and device I/O are as similar as possible; file and device names have the same syntax and meaning, so that a programming expecting a file name as a parameter can be passed a device name; finally, special files are subject to the same protection mechanism as regular files.* (p. 367)
  - "Everything is a file" has some benefits (one set of system calls) but also some disadvantages (non-obvious "ioctl" interface for device-specific functionality; lose type information.) Today, could this have been more cleanly done with subtyping?
  - Also: redirection
- set-User-ID
  - Coarse-grained sharing – "execute a program as someone else"
  - v. Multix rings – fine grained sharing – "execute a procedure as someone else"
  - minimalist: Need to have process == principal anyhow
  - just add setuid to that basic mechanism rather than invent an orthogonal authentication model
  - DA: more "heavy weight"
  - But, given "psychological acceptability" constraint, are there limits to how fine-grained we can (correctly/conveniently) divide programs?
  - also "Make common case fast. Make uncommon case correct." Common case is – procedure call to same code base. How much extra mechanism do you want (complexity, cost, speed penalty in common case) for uncommon case?
  - Compare power of approach
    * **Question**: suppose you have a "protected subsystem" S in multix that has data D_S that only it can access and to procedures S_{P1} that can be accessed by A and B and S_{P2} that can only be accessed by A
      · How would you arrange this in Multix?

      · Can you arrange this in Unix? How?

- Mount
  - Removable storage; expand storage;
  - Engineering simplification: No cross-volume links allowed

4 Process management

- Theme
  - Primitives, not "solutions"
  - "Happily, all of this mechanism meshes very nicely..."
- Building blocks
  - Fork, exec, wait
  - File I/O structure
    - Fork'd child shares parent's open files
    - → pipe, std I/O, redirection, filters
    - Coarse grained sharing of programs: cat foo — grep "bar" — sort — tail -10
  - Shell, bg execution
    - Elegant process structure enables communication



    - Shell pseudo-code

```
shell(){
  while(got = read(STDIN, buffer, ...)){
    command, args, redirection, bg = parse(buffer);
    if(pid = fork()){
      /* I am the child */
      if(redirection){
        close stdin and/or stdout and open specified files
      }
      exec(command, args);
```

```
          /* Only reached if error on exec */
          exit(-1);
        }
        /* I am the parent */
        if(!bg && donePid != pid){
          donePid = wait();
        }
    }
```

∗ HW Question: How to add pipes?

5 Admin

Project proposal – due a week from Friday

- Concrete plan (1-2 week granularity)
- Learn from Ritchie and Thompson – omit needless features (or cut marginally useful features) to achieve big improvements in simplicity; focus on core of what you need to do to accomplish research goal

6 Lessons

How do you teach/learn "elegance"? I don't know. Study case studies, try to learn lessons.

When you build a system, ask yourself "What would Richie and Thompson say about my design?"

Can it be learned at all? Dijkstra attributes much of elegance to "fear", Richie and Thompson to having small machines. (Answer: learn fear?)

6.1 Unix v. multix v. THE

Note: multix design and THE design pre-date Unix. Unix borrows liberally.

"The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system. (p. 374)

- Unix v. Multics
  - Implementation effort: 2 work-years v. ? work-years
  - Implementation latency: 2 years v. inf
  - Key ideas

Feature	Multix	Unix
Key abstraction	Unify file = memory	Unify I/O = file
Protection	rings (jump to memory)	suid (execute file)
Sharing	N segments; arbitrary sharing	3 segments (text, heap, stack) text is shared (RO); Communication between domains via files, pipes

- Coarse grained (file exec) v. fine grained sharing (procedure call)
  - Elegant structure
    - unified file I/O, interprocess I/O (pipes), device I/O
    - cooperating process structure: fork, shared files, shell, redirection, filters
    - setuid
  - Lampson: coarse grained sharing (pipes) is one of great success in CS; fine grained sharing an "abject failure" (my paraphrase of NSF workshop, SOSP keynote talks).
  - Lessons?
    - Simple primitives v. general solutions?
    - "Psychological acceptability" limits us to coarse-grained 2-entity sharing most of the time anyhow?
    - Trade performance and features for simplicity?
- Approach
  - Multics – "we need fine grained sharing; design the 'right' mechanism for it"
  - Unix – "any system we build must have (1) process, (2) ability to read and write files from a process, (3) a user ID associated with a process for access control, (4) be able to read/write tty (and other devices); given those requirements, can we design the basic abstractions in a way that supports sufficient sharing? (Yes. In fact, now that we look at it this way, file IO and device IO are the same thing....)"

- Unix v. THE
  - Similar scale (2-3 work-years)
  - Dates of completion: THE (1967?), Unix (v1 1969, v2 1971)
  - Similar abstractions

THE	Unix
Level 0: sequential execution	process
Level 1: Virtual memory	virtual memory
Level 2: Message demux	File I/O + special files + process
Level 3: Buffer I/O	Buffered I/O
Level 4: User-level process	User-level process
Level 5: Operator	Operator

- Unix 1974 v. Unix 2007
  - Much of multics has been put back in
  - 10K lines of code (not including utility programs) v. 50M lines of code (full distribution)
  - QUESTION: Many have given up on OS security and are resorting to virtual machines for process isolation...

6.2 Quotable UNIX

- *Given the partially antagonistic desires for reasonable efficiency and expressive power, the size constraint has encouraged not only economy but a certain elegance of design.* (p. 374)
  - Both THE and Unix gain success/elegance by ruthless simplification
  - THE motiation "fear"
  - Unix motivation – hardware constraints
  - Today's motivation? (discipline?)
- *Nearly from the start, the system was able to, and did, maintain itself. This fact is more important than it might seem. If designers of a system are forced to use that system, they quickly become aware of its functional and superficial deficiencies and are strongly motivated to crrect them.* (p. 374)
  - "Eat your own dogfood"
  - "worked once" system v. "real system"
  - Goes back to Feinman's rule #1