Operating Systems

Steven Hand
12 lectures for CST Ia

Easter Term 2000

Part III: Case Studies
Unix: Introduction

• Unix first developed in 1969 at Bell Labs (Thompson & Ritchie)

• Originally written in PDP-7 asm, but then (1973) rewritten in the ‘new’ high-level language C
  ⇒ easy to port, alter, read, etc.

• 6th edition (‘V6’) was widely available (1976).
  – Source avail ⇒ people could write new tools.
  – Nice features of other OSes rolled in promptly.

• By 1978, V7 available (for both the 16-bit PDP-11 and the new 32-bit VAX-11).

• Since then, two main families:
  – Berkeley: “BSD”, currently 4.3BSD/4.4BSD.

• Standardisation efforts (e.g. POSIX, X/OPEN) to homogenise.

• Best known “UNIX” today is probably Linux, but also get FreeBSD, NetBSD, and (commercially) Solaris, OSF/1, IRIX
Unix Family Tree (Simplified)

1969  First Edition
1973  Fifth Edition
1974
1975
1976  Sixth Edition
1977
1978  Seventh Edition
1979
1980
1981
1982  System III
1983  System V
1984  SVR2
1985
1986
1987  SVR3
1988
1989  SVR4
1990
1991
1992
1993

    32V
    3BSD
    4.0BSD
    4.1BSD
    4.2BSD
    SunOS
    SunOS 3
    4.3BSD/Tahoe
    4.3BSD/Reno
    OSF/1
    SunOS 4
    Solaris
    Solaris 2
Design Features

Ritchie and Thompson, CACM, July 74, UNIX (new) features:

1. A hierarchical file system incorporating demountable volumes.

2. Compatible file, device and inter-process I/O.

3. The ability to initiate asynchronous processes.

4. System command language selectable on a per-user basis.

5. Over 100 subsystems including a dozen languages.

6. A high degree of portability.

Features which were not included:

- real time
- multiprocessor support

Fixing the above is hard …
Structural Overview

- Clear separation between user and kernel portions.
- Processes are unit of scheduling and protection.
- All I/O looks like operations on files.
File Abstraction

• A file is an unstructured sequence of bytes.

• Represented in user-space by a file descriptor (fd)

• Operations on files are:
  – $fd = \texttt{open}(\textit{pathname}, \textit{mode})$
  – $fd = \texttt{creat}(\textit{pathname}, \textit{mode})$
  – bytes $= \texttt{read}(fd, buffer, nbytes)$
  – count $= \texttt{write}(fd, buffer, nbytes)$
  – reply $= \texttt{seek}(fd, offset, whence)$
  – reply $= \texttt{close}(fd)$

• Devices represented by special files.

• Hierarchical structure supported by directory files.
Directory Hierarchy

- Directories map names to files (and directories).
- Have distinguished root directory called ‘/’
- Fully qualified pathnames ⇒ perform traversal from root.
- Every directory has ‘.’ and ‘..’ entries: refer to self and parent respectively.
- Shortcut: current working directory (cwd).
- In addition shell provides access to home directory as “username (e.g. “steve/)
Aside: Password File

- /etc/passwd holds list of password entries.
- Each entry roughly of the form:
  
  user-name:encrypted-passwd:home-directory:shell
  
- Use one-way function to encrypt passwords.
- To login:
  1. Get user name
  2. Get password
  3. Encrypt password
  4. Check against version in /etc/password
  5. If ok, instantiate login shell.
- Publicly readable since lots of useful info there.
- Problem: off-line attack.
- Solution: shadow passwords (/etc/shadow)
Inside kernel, a file is represented by a data structure called an index-node or *i-node*.

Holds file *meta-data*:

a) Owner, permissions, reference count, etc.

b) Location on disk of actual data (file contents).

Where is the filename kept?
Directories and Links

- Directory is a file which maps filenames to i-nodes.
- An instance of a file in a directory is a (hard) link.
- (this is why have reference count in i-node).
- Directories can have at most 1 (real) link. Why?
- Also get soft- or symbolic-links: a ‘normal’ file which contains a filename.
On-Disk Structures

- A disk is made up of a boot block and one or more partitions.

- (a partition is just a contiguous range of $N$ fixed-size blocks of size $k$ for some $N$ and $k$).

- A Unix file-system resides within a partition.

- Superblock contains info such as:
  - number of blocks in file-system
  - number of free blocks in file-system
  - start of the free-block list
  - start of the free-inode list.
  - various bookkeeping information.
Mounting File-Systems

- Entire file-systems can be *mounted* on an existing directory.
- At very start, only ‘/’ exists ⇒ need to mount a *root file-system*.
- Subsequently can mount other file-systems, e.g. `mount("/dev/hda2", "/home", options)`
- Provides a *unified name-space*: e.g. access `/home/steve/` directly.
- Cannot have hard links across mount points: why?
- What about soft links?
In-Memory Tables

- Recall process sees files as file descriptors
- In implementation these are just indices into process-specific open file table
- Entries point to system-wide open file table. Why?
- These in turn point to (in memory) inode table.
Access Control

<table>
<thead>
<tr>
<th>Owner</th>
<th>Group</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>W</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>W</td>
</tr>
</tbody>
</table>

= 0640

<table>
<thead>
<tr>
<th>Owner</th>
<th>Group</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R</td>
<td>W</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>W</td>
</tr>
</tbody>
</table>

= 0755

- Access control information held in each inode.
- Three bits for each of owner, group and world: read, write and execute.
- What do these mean for directories?
- In addition have setuid and setgid bits:
  - normally processes inherit permissions of invoking user.
  - setuid/setgid allow user to “become” someone else when running a given program.
  - e.g. prof owns both executable test (0711 and setuid), and score file (0600)
    \[\Rightarrow\] anyone user can run it.
    \[\Rightarrow\] it can update score file.
    \[\Rightarrow\] but users can’t cheat.
- And what do these mean for directories?
**Consistency Issues**

- To delete a file, use the `unlink` system call.

- From the shell, this is `rm <filename>`

- Procedure is:
  1. Check if user has sufficient permissions on the file (must have `write` access).
  2. Check if user has sufficient permissions on the directory (must have `write` access).
  3. If ok, remove entry from directory.
  5. If now zero:
     a) Free data blocks.
     b) Free inode.

- If *crash*: must check entire file-system:
  - Check if any block unreferenced.
  - Check if any block double referenced.
Unix File-System: Summary

- Files are unstructured byte streams.

- Everything is a file: ‘normal’, directories, symbolic links, special files.

- Hierarchy built from root (‘/’).

- Unified name-space (multiple file-systems may be mounted).

- Low-level implementation based around inodes.

- Disk contains list of inodes (and of course data blocks).

- Processes see file descriptors: map to system file table.

- Permissions for owner, group and everyone else.

- Setuid/setgid allow for more flexible control.

- Care needed to ensure consistency.
Processes

- Recall: a process is a program in execution.
- Have three *segments*: text, data and stack.
- Unix processes are *heavyweight*. 

```
Unix
Kernel

Kernel Address Space (shared by all)

Stack Segment
- grows downward as functions are called

Free Space
- grows upwards as more memory allocated

Data Segment

Text Segment
```
Unix Process Dynamics

- Process represented by a process id (pid)
- Hierarchical scheme: parents create children.
- Four basic primitives:
  - \( \text{pid} = \text{fork}() \)
  - reply = \( \text{execve}(\text{pathname}, \text{argv}, \text{envp}) \)
  - \( \text{exit}(\text{status}) \)
  - \( \text{pid} = \text{wait}(\text{status}) \)
- \( \text{fork}() \) nearly always followed by \( \text{exec}() \)
  \(\Rightarrow\) \( \text{vfork}() \) and/or COW.
Start of Day

- Kernel (/vmunix) loaded from disk (how?) and execution starts.
- Root file-system mounted.
- Process 1 (/etc/init) hand-crafted.
- init reads conf file /etc/inittab and for each entry:
  1. opens terminal special file (e.g. /dev/tty0)
  2. duplicates the resulting fd twice.
  3. forks an /etc/tty process.
- each tty process next:
  1. initialises the terminal
  2. outputs the string “login:” & waits for input
  3. execve()’s /bin/login
- login then:
  1. outputs “password:” & waits for input
  2. encrypts password and checks it against /etc/passwd.
  3. if ok, sets uid & gid, and execve()’s shell.
- Patriarch init resurrects /etc/tty on exit.
• Shell just a process like everything else.

• Uses path for convenience.

• Conventionally ‘&’ specifies background.

• Parsing stage (omitted) can do lots …
Shell Examples

# pwd
/home/steve
# ls -F
IRAM.micro.ps       gnome_sizes       prog-nc.ps
Mail/               ica.tgz            rafe/
OSDI99_self_paging.ps.gz lectures/    rio107/
TeX/                linbot-1.0/       src/
adag.pdf           manual.ps         store.ps.gz
docs/               past-papers/      wolfson/
emacs-lisp/         pbosch/           xeno_prop/
fs.html             pepsi_logo.tif
# cd src/
# pwd
/home/steve/src
# ls -F
caq/               emacs-20.3.tar.gz   misc/   read_mem.c
emacs-20.3/         ispell/           read_mem*  rio007.tgz
# wc read_mem.c
  95  225  2262  read_mem.c
# ls -lF r*
-rwxrwxr-x 1 steve user 34956 Mar 21 1999 read_mem*
-rw-rw-r-- 1 steve user  2262 Mar 21 1999 read_mem.c
-rw------- 1 steve user 28953 Aug 27 17:40 rio007.tgz
# ls -l /usr/bin/X11/xterm
-rwxr-xr-x 2 root system 164328 Sep 24 18:21 /usr/bin/X11/xterm*

- Prompt is ‘#’.

- Use man to find out about commands.

- User friendly?
Standard I/O

- Every process has three fds on creation:
  - `stdin`: where to read input from.
  - `stdout`: where to send output.
  - `stderr`: where to send diagnostics.

- Normally inherited from parent, but shell allows redirection to/from a file, e.g.:
  - `ls >listing.txt`
  - `ls >&listing.txt`
  - `sh <commands.sh`

- Consider `ls >temp.txt; wc <temp.txt >results`

- *Pipeline* is better (e.g. `ls | wc >results`)

- Most Unix commands are *filters* ⇒ can build almost arbitrarily complex command lines.

- NB: redirection causes some subtleties …
Pipes

- One of the basic Unix IPC schemes.
- Logically consists of a pair of fds
- e.g. reply = pipe( int fds[2] )
- Concept of “full” and “empty” pipes.
- Only allows communication between processes with a common ancestor (why?).
- Named pipes address this.
Signals

- Problem: pipes need planning $\Rightarrow$ use *signals*.
- Similar to a (software) interrupt.
- Examples:
  - SIGINT: user hit Ctrl-C.
  - SIGSEGV: program error.
  - SIGCHLD: a death in the family ...
  - SIGTERM: ... or closer to home.
- Unix allows processes to *catch* signals.
- E.g. Job control:
  - SIGTTIN, SIGTTOUT sent to bg processes
  - SIGCONT turns bg to fg.
  - SIGSTOP does the reverse.
- Cannot catch SIGKILL.
- Signals also used for timers, window resize, process tracing, ...
• Recall:
  – everything accessed via the file system.
  – two broad categories: block and char.

• Low-level stuff gory and machdep ⇒ ignore.

• Character I/O low rate but complex ⇒ most functionality in the “cooked” interface.

• Block I/O simpler but performance matters ⇒ emphasis on the buffer cache.
The Buffer Cache

• Basic idea: keep copy of some parts of disk in memory for speed.

• On read do:
  1. Locate relevant blocks (from inode)
  2. Check if in buffer cache.
  3. If not, read from disk into memory.
  4. Return data from buffer cache.

• On write do same first three, and then update version in cache, not on disk.

• Q: when does data actually hit disk?

• Can cache metadata too — problems?
Unix Process Scheduling

- Round robin scheduling within discrete priorities
- Same quantum for all processes (100ms)
- Clock interrupts at regular intervals (10ms) — used for accounting
- Priorities are based on usage and nice (negative = higher priority):

\[ P_j(i) = Base_j + \frac{CPU_j(i - 1)}{4} + 2 \times nice_j \]

Gives the priority of process \( j \) at the beginning of interval \( i \) where:

\[ CPU_j(i) = \frac{2 \times load_j(i - 1)}{2(load_j(i - 1) + 1)}CPU_j(i - 1) + nice_j \]

- \( nice_j \) is a user controllable adjustment parameter \( \in [-20, 20] \).
- \( load_j(i) \) is the sampled average length of the run queue in which process \( j \) resides, over the last minute of operation
- Priorities recomputed once per second, at which time a new scheduling decision is made
Summary

• Main Unix features are:
  – file abstraction
  – hierarchical namespace
  – heavy-weight processes
  – IPC: pipes & signals
  – I/O: block and character
  – dynamic priority scheduling.

• But V7 had poor IPC, memory management, concurrency.

• Later systems address these …
Windows NT: History

After OS/2, MS decide they need “New Technology”:

- 1988: Dave Cutler recruited from DEC.
- 1989: team (~ 10 people) starts work on a new OS with a micro-kernel architecture.
- July 1993: first version (3.1) introduced

Bloated and suckful ⇒

- Followed in May 1995 by NT 3.51 (support for the Power PC, and more performance tweaks)
- July 1996: NT 4.0
  - new (windows 95) look ’n feel
  - various functions pushed back into kernel (most notably graphics rendering functions)
  - ongoing upgrades via service packs

NT 5.0 aka Windows 2000 released February 2000 …
NT Design Principles

Key goals for the system were:

- portability
- security
- POSIX compliance
- multiprocessor support
- extensibility
- international support
- compatibility with MS-DOS/Windows applications

The led to the development of a system which was:

- written in high-level languages (C and C++)
- based around a micro-kernel, and
- constructed in a layered/modular fashion.
Structural Overview

- Kernel Mode: HAL, Kernel, & Executive
- User Mode:
  - environmental subsystems
  - protection subsystem
HAL

- Layer of software (HAL.DLL) which hides details of underlying hardware
- e.g. interrupt mechanisms, DMA controllers, multiprocessor communication mechanisms
- Many HALs exist with same interface but different implementation (often vendor-specific)

Kernel

- Foundation for the executive and the subsystems
- Execution is never preempted.
- Four main responsibilities:
  1. CPU scheduling
  2. interrupt and exception handling
  3. low-level processor synchronisation
  4. recovery after a power failure
- Kernel is object-oriented; all objects either dispatcher objects and control objects
Processes and Threads

NT splits the “virtual processor” into two parts:

1. A **process** is the unit of resource ownership. Each process has:
   - a security token,
   - a virtual address space,
   - a set of resources (*object handles*), and
   - one or more *threads*.

2. A **thread** are the unit of dispatching. Each thread has:
   - a scheduling state (ready, running, etc.),
   - other scheduling parameters (priority, etc.),
   - a context slot, and
   - (generally) an associated process.

Threads are:

- co-operative: all threads in a process share the same address space & object handles.

- lightweight: require less work to create/delete than processes (mainly due to shared VAS).
CPU Scheduling

- Hybrid static/dynamic priority scheduling:
  - Priorities 16–31: “real time” (static priority).
  - Priorities 1–15: “variable” (dynamic) priority.

- Default quantum 2 ticks (≈20ms) on Workstation, 12 ticks (≈120ms) on Server.

- Threads have base and current ($\geq$ base) priorities.
  - On return from I/O, current priority is boosted by driver-specific amount.
  - Subsequently, current priority decays by 1 after each completed quantum.
  - Also get boost for GUI threads awaiting input: current priority boosted to 14 for one quantum (but quantum also doubled)
  - Yes, this is true.

- On Workstation also get quantum stretching:
  - “... performance boost for the foreground application” (window with focus)
  - fg thread gets double or triple quantum.
Object Manager

- Every resource in NT is represented by an object

- The Object Manager (part of the Executive) is responsible for:
  - creating objects and object handles
  - performing security checks
  - tracking which processes are using each object

- Typical operation:
  - handle = open(object-name, access-mode)
  - result = service(handle, arguments)
Object Namespace

- Recall: objects (optionally) have a name
- Object Manger manages a hierarchical namespace:
  - shared between all processes \( \Rightarrow \) sharing
  - implemented via directory objects
  - naming domains (implemented via parse) mean file-system namespaces can be integrated too
- Also get symbolic link objects: allow multiple names (aliases) for the same object.
- Modified view presented at API level …
Process Manager

- Provides services for creating, deleting, and using threads and processes.

- Very flexible:
  - No built in concept of parent/child relationships or process hierarchies
  - Processes and threads treated completely orthogonally.

Virtual Memory Manager

- NT employs paged virtual memory management

- The VMM provides processes with services to:
  - allocate and free virtual memory
  - modify per-page protections

- Can also share portions of memory:
  - use *section objects* (∼ software segments)
  - based versus non-based.
  - also used for *memory-mapped files*
I/O Manager

- The I/O Manager is responsible for:
  - file systems
  - cache management
  - device drivers

- Basic model is asynchronous:
  - each I/O operation explicitly split into a request and a response
  - I/O Request Packet (IRP) used to hold parameters, results, etc.

- File-system & device drivers are stackable ...
File System

- The fundamental structure of the NT filing system (NTFS) is a *volume*
  - Created by the NT disk administrator utility
  - Based on a logical disk partition
  - May occupy a portion of a disk, and entire disk, or span across several disks.

- A file in NTFS is not a simple byte stream, as in MS-DOS or UNIX, rather, it is a structured object consisting of *attributes*.

- Every file in NTFS is described by one or more records in an array stored in a special file called the Master File Table (MFT).

- NTFS has a number of advanced features, e.g.
  - security (access checks on open)
  - unicode based names
  - use of a log for efficient recovery
  - support for sparse and compressed files

- (but only recently are features being used)
Summary

- Main Windows NT features are:
  - layered/modular architecture
  - generic use of objects throughout
  - multi-threaded processes
  - multiprocessor support
  - asynchronous I/O subsystem
  - advanced filing system
  - preemptive priority-based scheduling

- HAL, Kernel & Executive: rather decent actually.

- But ...