**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, and Tru64.

**Design Features**

Ritchie and Thompson writing in CACM, July 74, identified the following (new) features of UNIX:

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 pretty hard.

**Unix Family Tree (Simplified)**

**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:
  - \( fd = \text{open}( \text{pathname}, \text{mode}) \)
  - \( fd = \text{creat}( \text{pathname}, \text{mode}) \)
  - \( \text{bytes} = \text{read}(fd, \text{buffer}, \text{nbytes}) \)
  - \( \text{count} = \text{write}(fd, \text{buffer}, \text{nbytes}) \)
  - \( \text{reply} = \text{seek}(fd, \text{offset}, \text{whence}) \)
  - \( \text{reply} = \text{close}(fd) \)
- Devices represented by special files:
  - support above operations, although perhaps with bizarre semantics.
  - also have ioctl's: allow access to device-specific functionality.
- Hierarchical structure supported by directory files.

Aside: Password File

- /etc/passwd holds list of password entries.
- Each entry roughly of the form:
  \[ \text{username encrypted-password home-directory:shell} \]
- Use one-way function to encrypt passwords.
  - i.e. a function which is easy to compute in one direction, but has a hard to compute inverse.
- 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)

Directory Hierarchy

- Directories map names to files (and directories).
- Have distinguished root directory called '/'
- Fully qualified pathnames \( \Rightarrow \) 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")

File System Implementation

- Inside kernel, a file is represented by a data structure called an index-node or i-node.
- Holds file meta-data:
  1. Owner, permissions, reference count, etc.
  2. 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.

Mounting File-Systems

- Entire file-systems can be mounted on an existing directory in an already mounted filesystem.
- 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?

On-Disk Structures

- A disk is made up of a boot block followed by 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-i-node list.
  - various bookkeeping information.

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>0640</td>
<td>0755</td>
<td></td>
</tr>
</tbody>
</table>

- 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. root owns both executable test (0711 and setuid), and score file (0600)
    - any user can run it.
    - it can update score file.
    - but users can’t cheat.

Unix File-System: Summary
- Files are unstructured byte streams.
- Everything is a file: ‘normal’ files, directories, symbolic links, special files.
- Hierarchy built from root (‘/’).
- Unified name-space (multiple file-systems may be mounted on any leaf directory).
- Low-level implementation based around inodes.
- Disk contains list of inodes (along with, of course, actual data blocks).
- Processes see file descriptors: small integers which map to system file table.
- Permissions for owner, group and everyone else.
- Setuid/setgid allow for more flexible control.
- Care needed to ensure consistency.

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

- Process represented by a *process id* (pid)
- Hierarchical scheme: parents create children.
- Four basic primitives:
  - `pid = fork()`
  - `reply = execve(pathname, argv, envp)`
  - `exit(status)`
  - `pid = wait(status)`
- `fork()` nearly always followed by `exec()` ⇒ `vfork()` and/or COW.

Start of Day

- Kernel (/vmlinuz) loaded from disk (how?) and execution starts.
- Root file-system mounted.
- Process 1 (/etc/init) hand-crafted.
- init reads 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 resuscitates /etc/tty on exit.

The Shell

- Shell just a process like everything else.
- Uses *path* for convenience.
- Conventionally 'k' specifies *background*.
- Parsing stage (omitted) can do lots...

Shell Examples

```bash
# get /home/stem
# ls -l
INARstore.psNull/INARstore.ps_PgB.png-Pg.ps.gz
file.png.ps.gz

# cd/home/stem/src
# ls -l
src/depens.c
src/depsrc.c
src/expand.c
src/file.c
src/fork.c
src/generate.c
src/generate.h
src/generate.i
src/generate.o
src/generate.s
src/generate.y
src/generate.z
src/hexdump.c
src/hexdump.h
src/hexdump.o
src/hexdump.s
src/hexdump.y
src/hexdump.z
src/init.c
src/init.h
src/init.o
src/init.s
src/init.y
src/init.z
src/pause.c
src/pause.h
src/pause.o
src/pause.s
src/pause.y
src/pause.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
src/rgen.c
src/rgen.h
src/rgen.o
src/rgen.s
src/rgen.y
src/rgen.z
`
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`.
- Actual file not always appropriate; e.g. consider:
  - `ls >temp.txt`;
  - `wc <temp.txt >results`
- `Pipe` is better (e.g. `ls | wc >results`)
- Most Unix commands are `filters` => can build almost arbitrarily complex command lines.
- `Redirection` can cause some buffering 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 => use `signals`.
- Similar to a (software) interrupt.
- Examples:
  - `SIGINT` : user hit Ctrl-C.
  - `SIGSEGV` : program error.
  - `SIGCHILD` : a death in the family...
  - `SIGTERM` : ... or closer to home.
- Unix allows processes to `catch` signals.
- e.g. Job control:
  - `SIGTIN, SIGTTOU` sent to bg processes
  - `SIGCONT` turns bg to fg.
  - `SIGSTOP` does the reverse.
- Cannot catch `SIGKILL` (hence `kill -9`)
- Signals can also be used for timers, window resize, process tracing, ...

I/O Implementation

- 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.
- “Typically” prevents 85% of implied disk transfers.
- Question: when does data actually hit disk?
- Answer: call sync every 30 seconds to flush dirty buffers to disk.
- Can cache metadata too — problems?
- need mutual exclusion and condition synchronisation
  e.g. WAIT for a buffer
  e.g. WAIT for full (data transfer complete).

Unix Process States

- ru — running (user-mode)
- rk — running (kernel-mode)
- z — zombie
- p — pre-empted
- sl — sleeping
- rb — runnable
- c — created

Unix Process Scheduling

- Priorities 0–127; user processes ≥ PUSER = 50.
- Round robin within priorities, quantum 100ms.
- Priorities are based on usage and nice, i.e.
  \[
  P_j(i) = \text{Base}_j + \frac{CPU_j(i-1)}{4} + 2 \times \text{nice}_j
  \]
  gives the priority of process \( j \) at the beginning of interval \( i \) where:
  \[
  CPU_j(i) = \frac{2 \times \text{load}_j}{(2 \times \text{load}_j) + 1} CPU_j(i-1) + \text{nice}_j
  \]
  and \( \text{nice}_j \) is a (partially) user controllable adjustment parameter \( \in [-20,20] \).
- load \( j \) is the sampled average length of the run queue in which process \( j \) resides, over the last minute of operation
- so if e.g. load is 1 \( \Rightarrow \sim 90\% \) of 1 seconds CPU usage “forgotten” within 5 seconds.

Summary

- Main Unix features are:
  - file abstraction
    - a file is an unstructured sequence of bytes
    - (not really true for device and directory files)
  - hierarchical namespace
    - directed acyclic graph (if exclude soft links)
    - can recursively mount filesystems
  - heavy-weight processes
  - IPC: pipes & signals
  - I/O: block and character
  - dynamic priority scheduling
    - base priority level for all processes
    - priority is lowered if process gets to run
    - over time, the past is forgotten
- But V7 had inflexible IPC, inefficient memory management, and poor kernel concurrency.
- Later versions address these issues.

Note: above is simplified — see CS section 23.14 for detailed descriptions of all states/transitions.
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 slow:
- 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)
- Feb 2000: NT 5.0 aka Windows 2000
  - big push to finally kill DOS/Win 9x family

NT Design Principles

Key goals for the system were:
- portability
- security
- POSIX compliance
- multiprocessor support
- extensibility
- international support
- compatibility with MS-DOS/Windows applications

This 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
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** is 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 (≥ 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(objectname, accessmode)`
  - `result = service(handle, arguments)`

### Object Namespace

- Recall: objects (optionally) have a name
- Object Manager manages a hierarchical namespace:
  - shared between all processes ⇒ sharing
  - implemented via *directory objects*
  - each object protected by an access control list
  - *naming domains* (implemented via parse) mean file-system namespaces can be integrated
- Also get *symbolic link objects*: allow multiple names (aliases) for the same object.
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 orthogonally
  ⇒ can support Posix, OS/2 and Win32 models.

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*

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)

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*.

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
- Design essentially more advanced than Unix.
- Implementation of lower levels (HAL, kernel & executive) actually rather decent.
- But: has historically been crippled by
  - almost exclusive use of Win32 API
  - legacy device drivers (e.g. VXD’s)
  - lack of demand for “advanced” features
- Windows XP + Luna might finally break free.
Course Review

- **Part I: Computer Organisation**
  - fetch-execute cycle, data representation, etc
  - mainly for getting up to speed for h/w courses
- **Part II: Operating System Functions**
  - OS structures: h/w support, kernel vs. μ-kernel
  - Processes: states, structures, scheduling
  - Memory: virtual addresses, sharing, protection
  - Filing: directories, meta-data, file operations.
- **Part III: Concurrency Control**
  - multithreaded processes
  - mutual exclusion and condition synchronisation
  - implementation of concurrency control
- **Part IV: Case Studies**
  - UNIX and Windows NT/2000