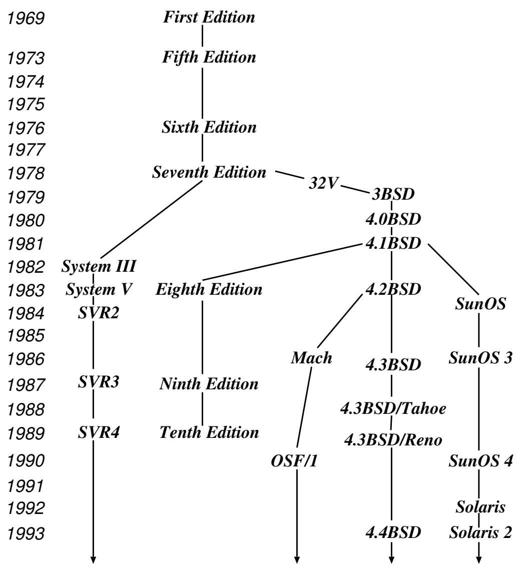
UNIX: Introduction

- 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.

- 6$^{\text{th}}$ 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:
  - AT&T: "System V", currently SVR4.
  - 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.

Unix Family Tree (Simplified)



Design Features

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

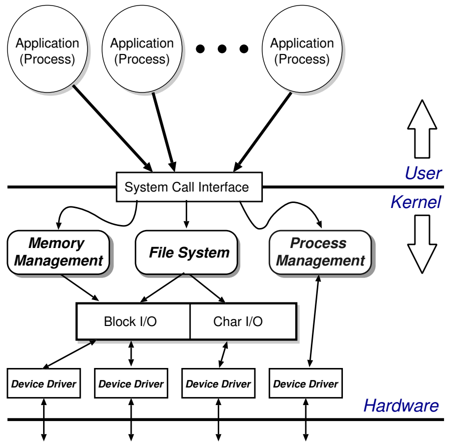
1. A hierarchical file system incorporating demountable volumes.
2. Compatible file, device and inter-process I/O.
3. The ability to initiate asynchronous processes.
4. System command language selectable on a per-user basis.
5. Over 100 subsystems including a dozen languages.
6. A high degree of portability.

Features which were not included:

- real time
- multiprocessor support

Fixing the above is pretty hard.

Structural Overview

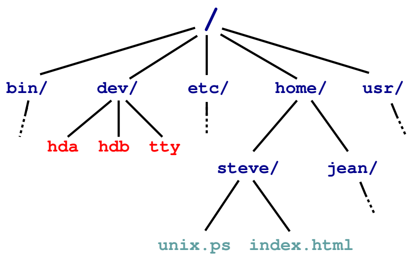


- Clear separation between *user* and *kernel* portions.
- Processes are unit of scheduling and protection.
- All I/O looks like operations on *files*.

File Abstraction

- A file is an unstructured sequence of bytes.
- Represented in user-space by a *file descriptor* (fd)
- Operations on files are:
  - *fd* = **open** (*pathname*, *mode*)
  - *fd* = **creat**(*pathname*, *mode*))
  - bytes = **read**(*fd*, *buffer*, *nbytes*)
  - count = **write**(*fd*, *buffer*, *nbytes*)
  - reply = **seek**(*fd*, *offset*, *whence*)
  - reply = **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*.

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/)

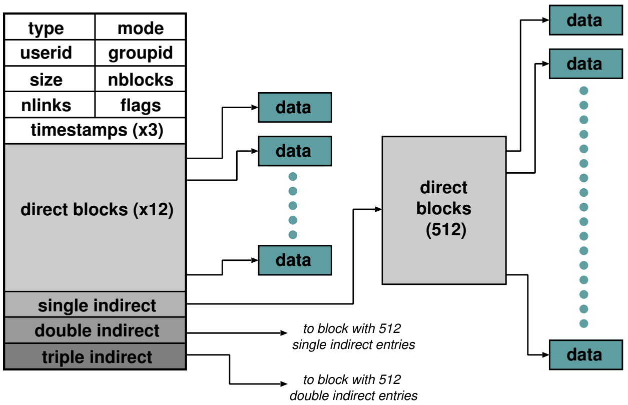
Aside: Password File

- `/etc/passwd` holds list of password entries.
- Each entry roughly of the form:

  user-name:encrypted-passwd:home-directory:shell

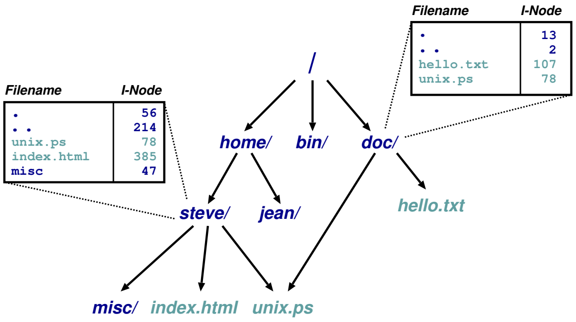
- Use *one-way function* to encrypt passwords.
  - i.e. a function which is easy to compute in one direction, but has a hard to compute inverse.
- To login:
  1. Get user name
  2. Get password
  3. Encrypt password
  4. Check against version in `/etc/password`
  5. If ok, instantiate login shell.
- Publicly readable since lots of useful info there.
- Problem: off-line attack.
- Solution: *shadow passwords* (`/etc/shadow`)

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

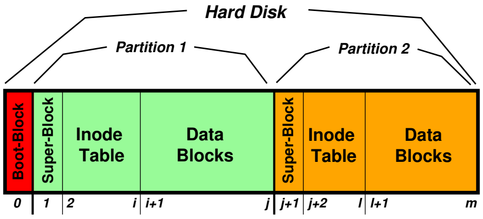


Filename	I-Node
.	56
. .	214
unix.ps	78
index.html	385
misc	47

Filename	I-Node
.	13
. .	2
hello.txt	107
unix.ps	78

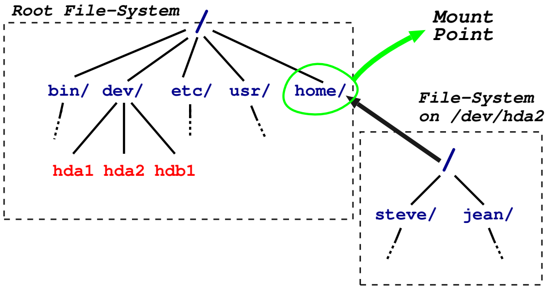
- Directory is a file which maps filenames to i-nodes.
- An instance of a file in a directory is a (hard) *link*.
- (this is why have reference count in i-node).
- Directories can have at most 1 (real) link. Why?
- Also get *soft-* or *symbolic*-links: a 'normal' file which contains a filename.

On-Disk Structures



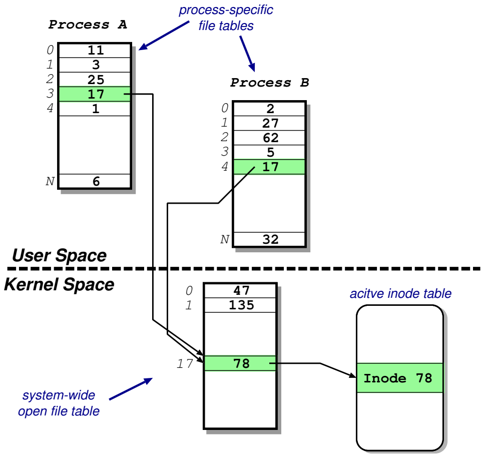
- A disk is made up of a *boot block* followed by one or more *partitions*.
- (a partition is just a contiguous range of N fixed-size blocks of size k for some N and k).
- A Unix file-system resides within a partition.
- *Superblock* contains info such as:
  - number of blocks in file-system
  - number of free blocks in file-system
  - start of the free-block list
  - start of the free-inode list.
  - various bookkeeping information.

Mounting File-Systems



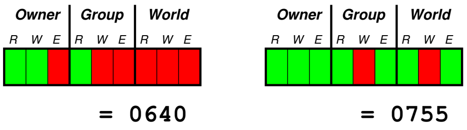
- Entire file-systems can be *mounted* on an existing directory in an already mounted filesystem.
- At very start, only '/' exists ⇒ need to mount a *root file-system*.
- Subsequently can mount other file-systems, e.g. `mount("/dev/hda2", "/home", options)`
- Provides a *unified name-space*: e.g. access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

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



- 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)

  ⇒ any user can run it.
  ⇒ it can update `score` file.
  ⇒ but users can't cheat.

Consistency Issues

- To delete a file, use the `unlink` system call.
- From the shell, this is `rm <filename>`
- Procedure is:

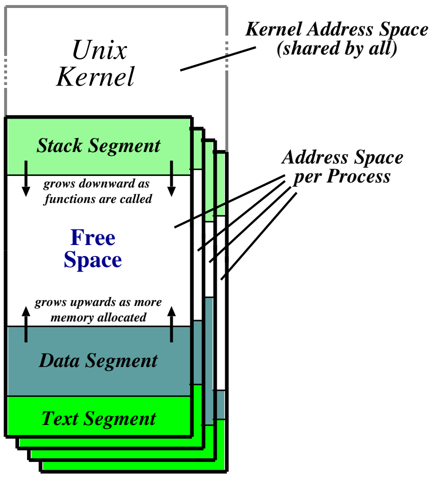
1. check if user has sufficient permissions on the file (must have *write* access).
2. check if user has sufficient permissions on the directory (must have *write* access).
3. if ok, remove entry from directory.
4. Decrement reference count on inode.
5. if now zero:
   (a) free data blocks.
   (b) free 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' 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.

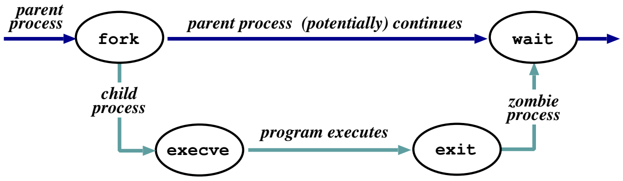
Unix Processes



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
Address Space per Process

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

Unix Process Dynamics



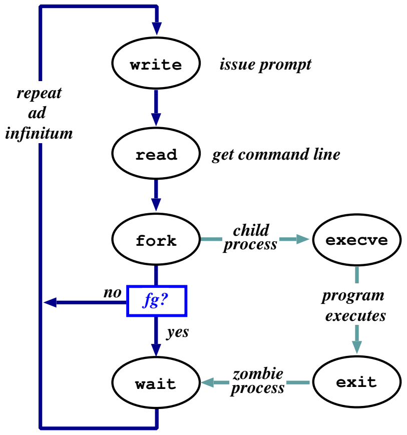
- Process represented by a *process id* (pid)
- Hierarchical scheme: parents create children.
- Four basic primitives:
  - *pid* = **fork** ()
  - reply = **execve**(*pathname*, *argv*, *envp*)
  - **exit**(*status*)
  - *pid* = **wait** (*status*)
- **fork()** nearly *always* followed by **exec()**

⇒ **vfork()** and/or COW.

Start of Day

- Kernel (`/vmunix`) 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 resurrects `/etc/tty` on exit.

The Shell



- 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
cdq/         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?

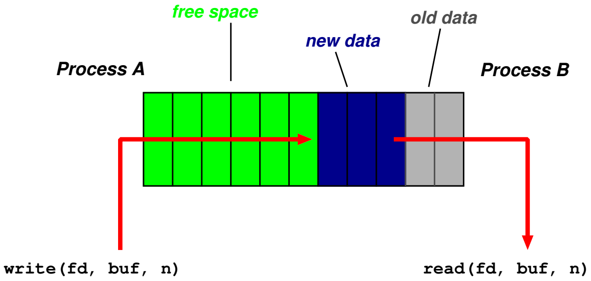
Standard I/O

- 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`.
- Actual file not always appropriate; e.g. 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.
- Redirection can cause some buffering subtleties.

Pipes

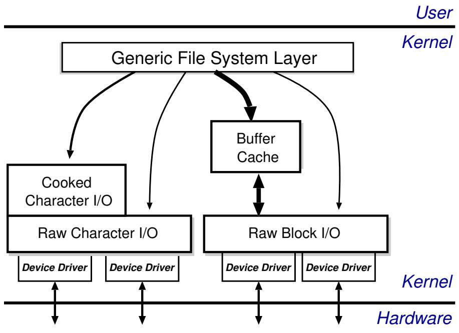


- One of the basic Unix IPC schemes.
- Logically consists of a pair of fds
- e.g. reply = **pipe**(int fds[2])
- Concept of "full" and "empty" pipes.
- Only allows communication between processes with a common ancestor (why?).
- *Named pipes* address this.

Signals

- Problem: pipes need planning ⇒ use *signals*.
- Similar to a (software) interrupt.
- Examples:
  - `SIGINT` : user hit Ctrl-C.
  - `SIGSEGV` : program error.
  - `SIGCHLD` : a death in the family. . .
  - `SIGTERM` : . . . or closer to home.
- Unix allows processes to *catch* signals.
- e.g. Job control:
  - `SIGTTIN`, `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 comlete.

Unix Process Scheduling

- Priorities 0–127; user processes $\geq$ `PUSER` = 50.
- Round robin within priorities, quantum 100ms.
- Priorities are based on usage and *nice*, i.e.

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

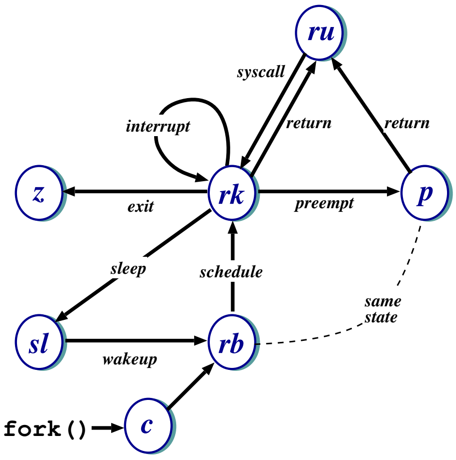
gives the priority of process j at the beginning of interval i where:

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

and $nice_j$ 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.

Unix Process States



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

- Note: above is simplified — see CS section 23.14 for detailed descriptions of all states/transitions.

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.

Windows NT: History

After OS/2, MS decide they need "**N**ew **T**echnology":

- 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 ⇒

- NT 3.5 released in September 1994: mainly size and performance optimisations.
- 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

Windows XP (NT 6.0) coming June 2001. . .

NT Design Principles

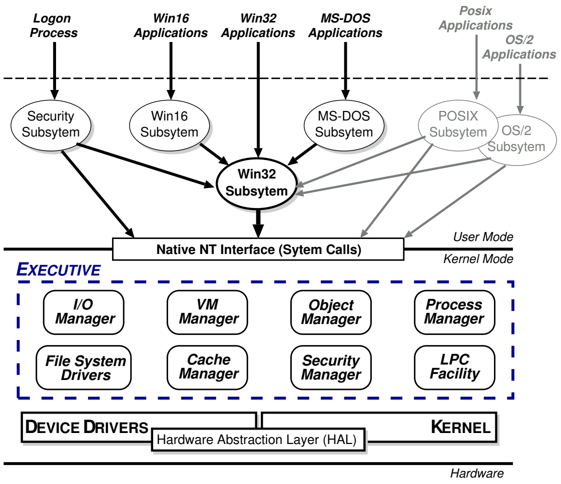
Key goals for the system were:

- 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 objected-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** 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.

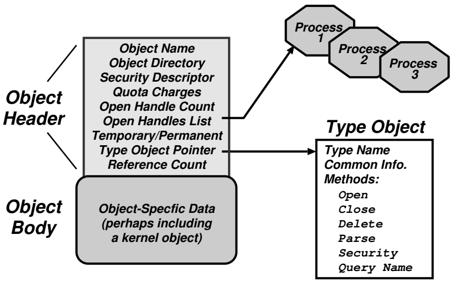
Threads are:

- 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

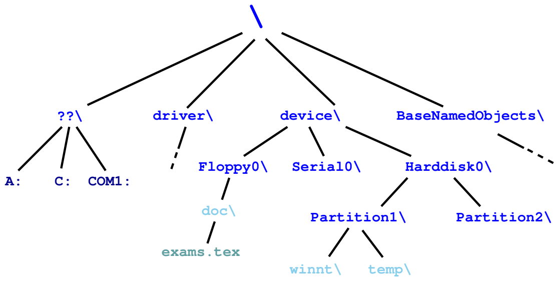
- Hybrid static/dynamic priority scheduling:
  - Priorities 16–31: “real time” (static priority).
  - Priorities 1–15: “variable” (dynamic) priority.
- Default quantum 2 ticks (∼20ms) on Workstation, 12 ticks (∼120ms) on Server.
- Threads have *base* and *current* (≥ base) priorities.
  - On return from 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

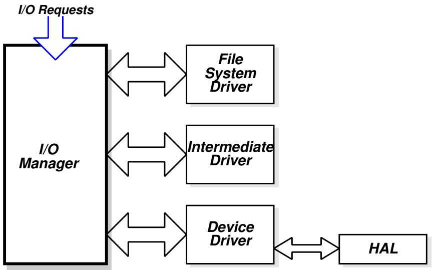
- Provides services for creating, deleting, and using threads and processes.
- Very flexible:
  - no built in concept of parent/child relationships or process hierarchies
  - processes and threads treated orthogonally.

  ⇒ 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*

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
- 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. VXDs)
  - 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