# Operating Systems

Steven Hand

*Lent Term 2010*

Handout 3 - Case Studies

Operating Systems — N/H/MW@12

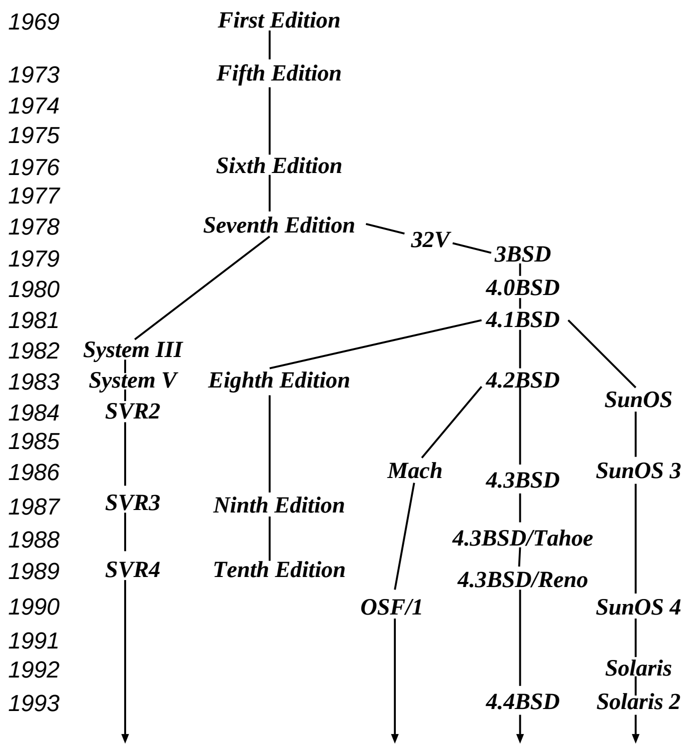
## Unix: Introduction

- 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:
  - 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:

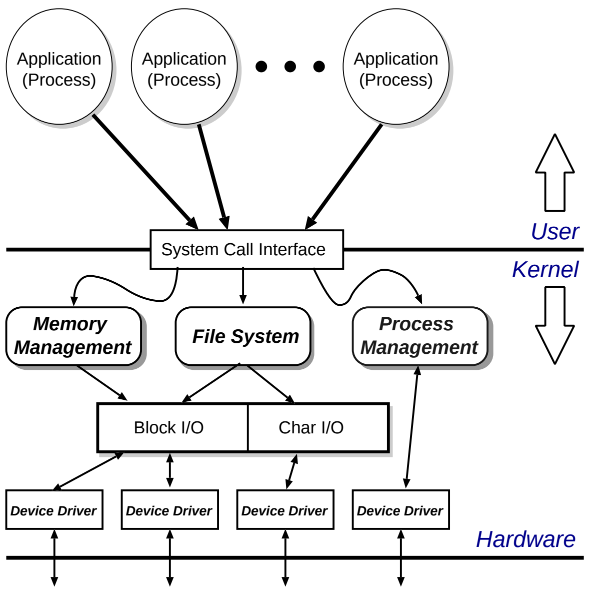
1. A hierarchical file system incorporating demountable volumes.
2. Compatible file, device and inter-process I/O.
3. The ability to initiate asynchronous processes.
4. System command language selectable on a per-user basis.
5. Over 100 subsystems including a dozen languages.
6. A high degree of portability.

Features which were not included:

- real time
- multiprocessor support

Fixing the above is pretty hard.

## Structural Overview

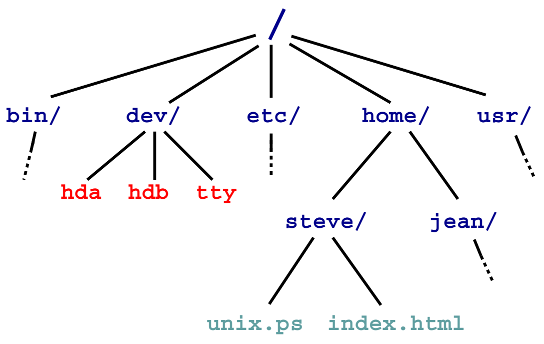


- Clear separation between user and kernel portions.
- Processes are unit of scheduling and protection.
- All I/O looks like operations on files.

## 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/)

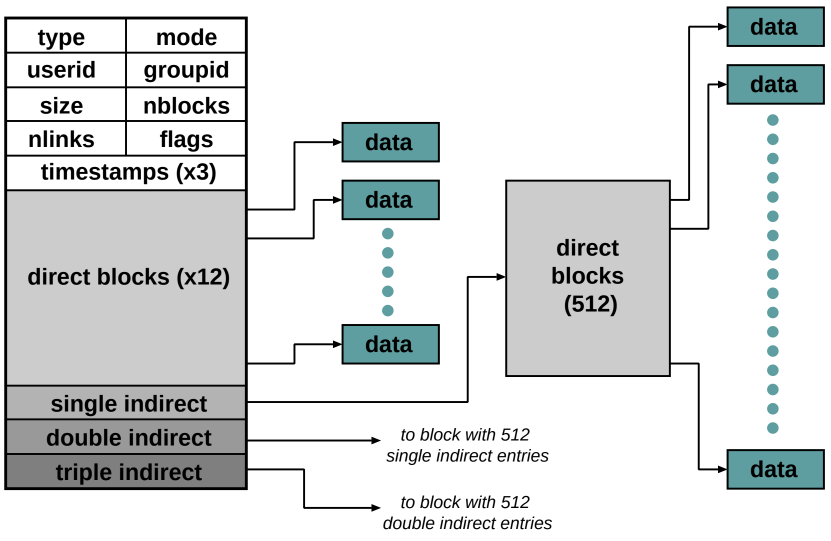
## Aside: Password File

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

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

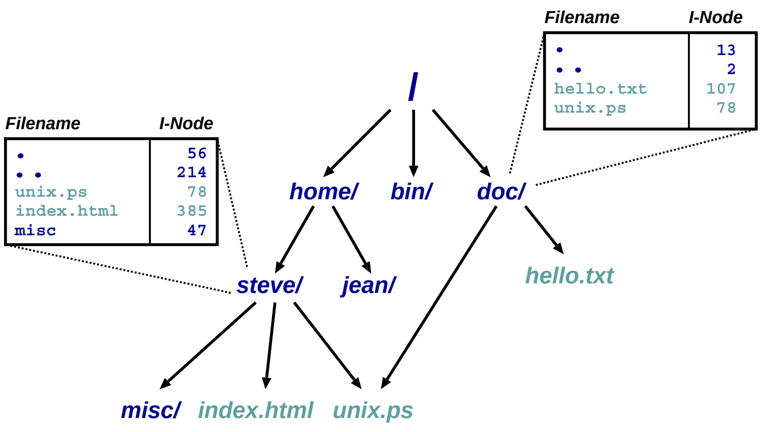
- Use one-way function to encrypt passwords.
  - i.e. a function which is easy to compute in one direction, but has a hard to compute inverse (e.g. person to phone-number lookup).
- To login:
  1. Get user name
  2. Get password
  3. Encrypt password
  4. Check against version in `/etc/password`
  5. 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



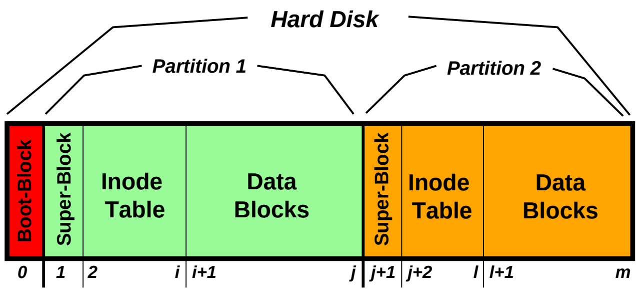
- In kernel, a file is represented by a data structure called an index-node or i-node.
- Holds file meta-data:
  a) Owner, permissions, reference count, etc.
  b) Location on disk of actual data (file contents).
- **Question**: Where is the filename kept?

## Directories and Links



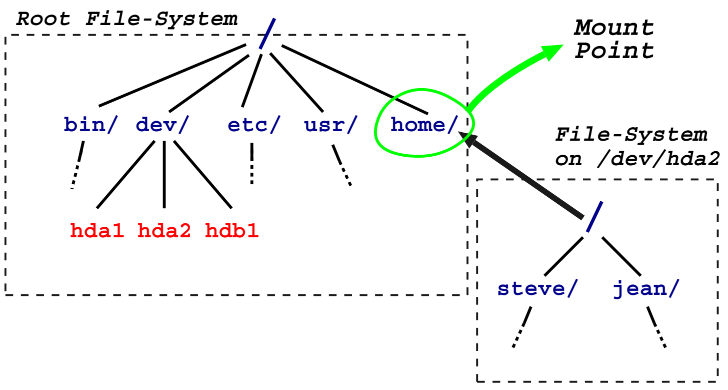
- Directory is a file which maps filenames to i-nodes.
- An instance of a file in a directory is a (hard) link.
- (this is why have reference count in i-node).
- Directories can have at most 1 (real) link. Why?
- Also get soft- or symbolic-links: a 'normal' file which contains a filename.

## On-Disk Structures



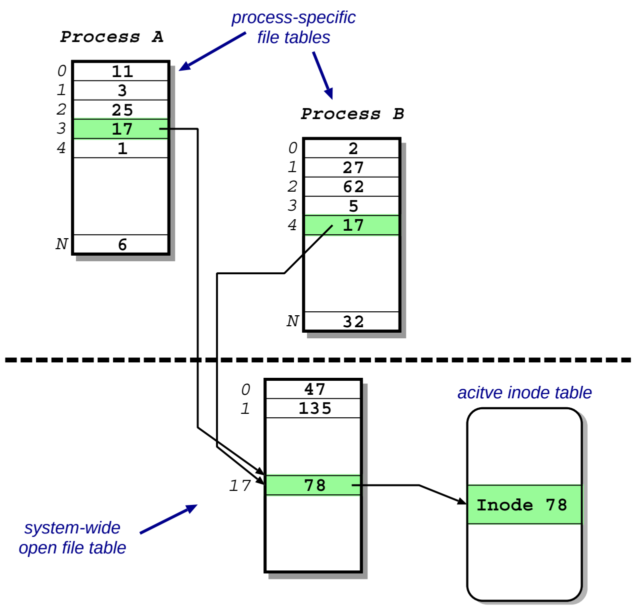
- A disk is made up of a boot block followed by one or more partitions.
- (a partition is a contiguous range of $N$ fixed-size blocks of size $k$ for some $N$, $k$).
- A Unix file-system resides within a partition.
- The file-system superblock contains info such as:
  - number of blocks in file-system
  - 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



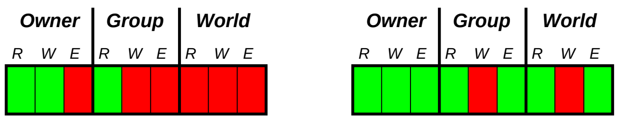
- Entire file-systems can be mounted on an existing directory in an already mounted filesystem.
- At very start, only '/' exists ⇒ need to mount a root file-system.
- Subsequently can mount other file-systems, e.g. `mount("/dev/hda2", "/home", options)`
- Provides a unified name-space: e.g. access `/home/steve/` 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 a 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



= 0640 = 0755

- Access control information held in each inode.
- Three bits for each of owner, group and world: { read, write and execute }
- **Question**: What do these mean for directories?
- In addition have setuid and setgid bits:
  - normally processes inherit permissions of invoking user.
  - setuid/setgid allow the user to "become" someone else when running a particular 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.
- **Question**: and what do *these* mean for directories?

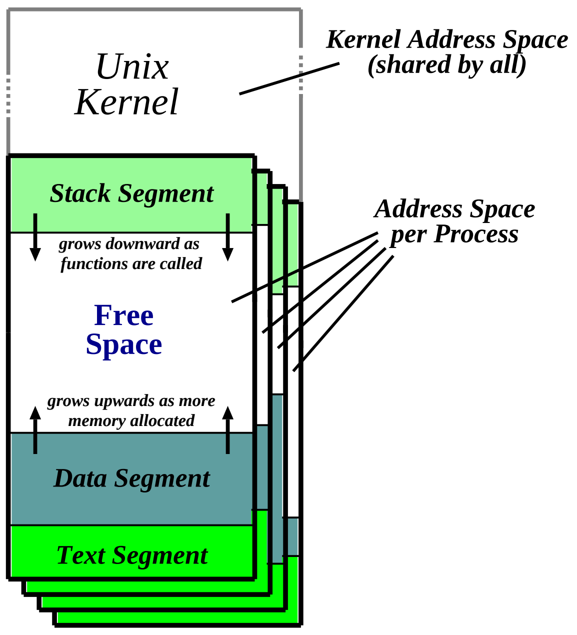
## Consistency Issues

- 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.
  4. Decrement reference count on inode.
  5. if now zero:
     a. free data blocks.
     b. free inode.
- If the system crashes: must check entire file-system:
  - check if any block unreferenced.
  - check if any block double referenced.
- (We'll see more on this later)

## Unix File-System: Summary

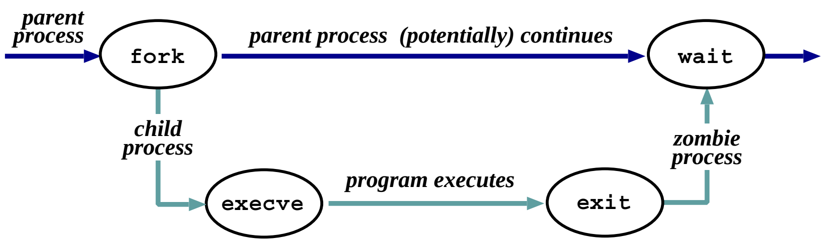
- Files are unstructured byte streams.
- Everything is a file: 'normal' files, directories, symbolic links, special files.
- Hierarchy built from root ('/').
- Unified name-space (multiple file-systems may be mounted on any leaf directory).
- Low-level implementation based around inodes.
- Disk contains 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



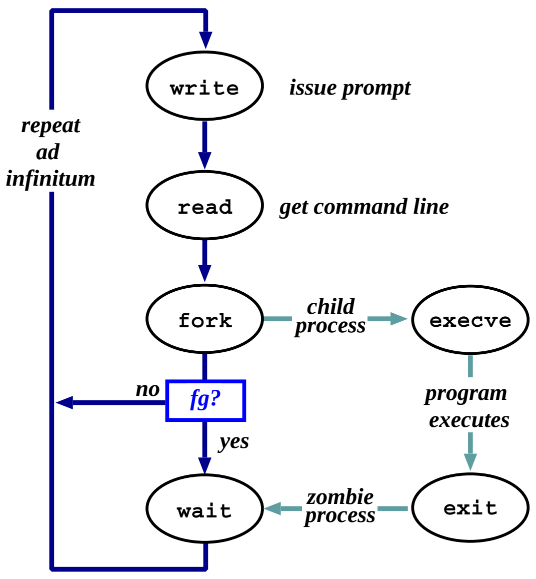
- Process represented by a *process id* (pid)
- Hierarchical scheme: parents create children.
- Four basic primitives:
  - *pid* = **fork** ()
  - reply = **execve**(*pathname*, *argv*, *envp*)
  - **exit**(*status*)
  - *pid* = **wait** (*status*)
- **fork()** nearly *always* followed by **exec()**

  ⇒ **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



- The shell just a process like everything else.
- Uses path (= list of directories to search) for convenience.
- Conventionally '&' specifies run in 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?

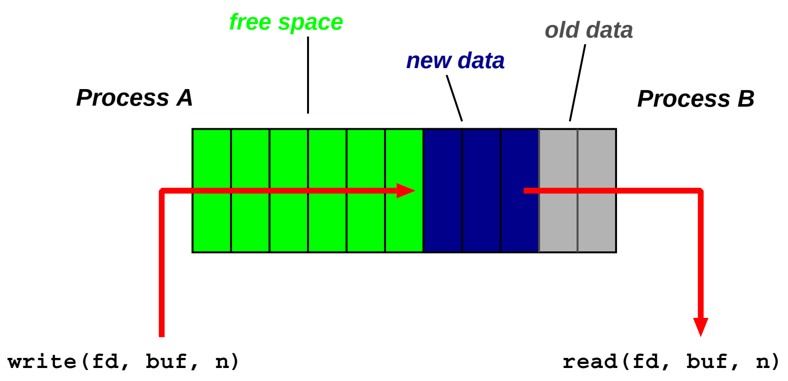
## Standard I/O

- 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, i.e. read from **stdin** and output to **stdout** ⇒ can build almost arbitrarily complex command lines.
- Redirection can cause some buffering subtleties.

## Pipes

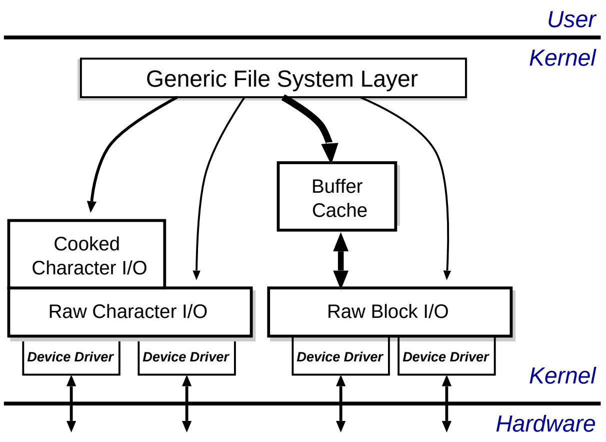


- One of the basic Unix IPC schemes.
- Logically consists of a pair of fds, one for each 'end' of the pipe.
- e.g. reply = **pipe**( int fds[2] )
- Concept of "full" and "empty" pipes.
- Only allows communication between processes with a common ancestor (why?).
- Named pipes address this. . .

## 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 machine dependent ⇒ ignore.
- Character I/O is low rate but complex ⇒ most code 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?

## Unix Process Scheduling

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

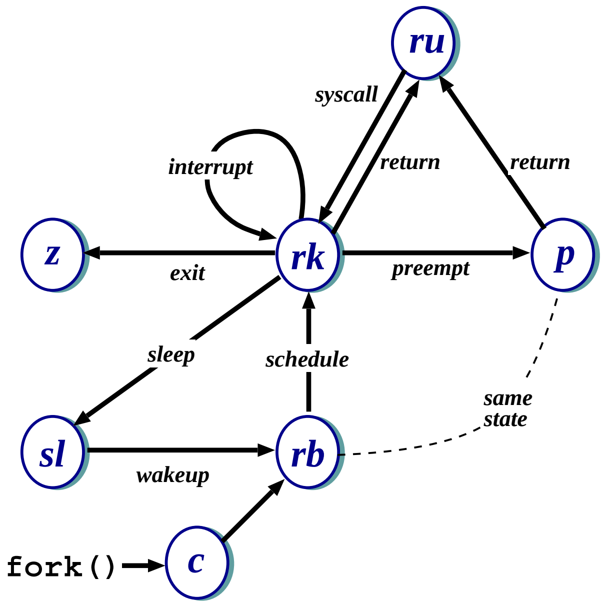
$$P_j(i) = Base_j + \frac{CPU_j(i-1)}{4} + 2 \times nice_j$$

  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 $\Rightarrow$ approximately 90% of 1 seconds CPU usage will be "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 Unix 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 ($\sim$ 10 people) starts work on a new OS (micro-kernel architecture)
- July 1993: first version (3.1) introduced
- (name compatible with windows 3.1)

Bloated and suckful $\Rightarrow$

- 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
  - some desktop users but mostly limited to servers
  - for performance reasons, various functions pushed back into kernel (most notably graphics rendering functions)
  - ongoing upgrades via service packs

## Windows NT: Evolution

- Feb 2000: NT 5.0 aka Windows 2000
  - borrows from windows 98 look 'n feel
  - both server and workstation versions, latter of which starts to get wider use
  - big push to finally kill DOS/Win 9x family (but fails due to internal politicking)
- Windows XP (NT 5.1) launched October 2001
  - home and professional $\Rightarrow$ finally kills win 9x.
  - various "editions" (media center, 64-bit) & service packs (SP1, SP2, SP3)
- Server product Windows Server 2003 (NT 5.2) released 2003
  - basically the same modulo registry tweaks, support contract and of course **cost**
  - a plethora of editions. . .
- Windows Vista (NT 6.0) limped onto the scene Q4 2006
  - new *Aero* UI, new *WinFX* API
  - missing Longhorn bits like *WinFS*, *Msh*
- Windows Server 2008 (also based on NT 6.0, but good) landed Feb 2008
- Windows 7 (NT 6.1, build 7600) released October 2009. . .

## NT Design Principles

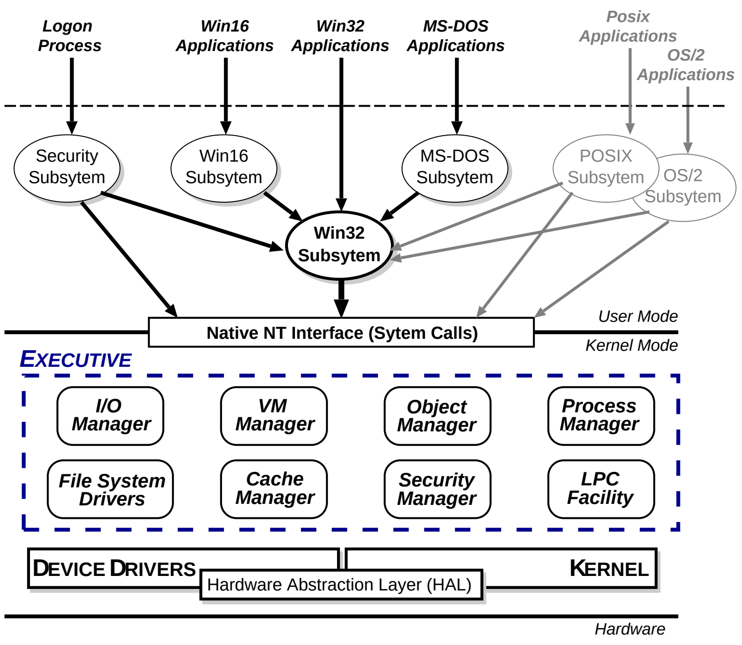
Key goals for the system were:

- 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. low-level interrupt mechanisms, DMA controllers, multiprocessor communication mechanisms
- Several HALs exist with same interface but different implementation (often vendor-specific, e.g. for large cc-NUMA machines)

## 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 are either dispatcher objects (active or temporal things) or control objects (everything else)

## 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.

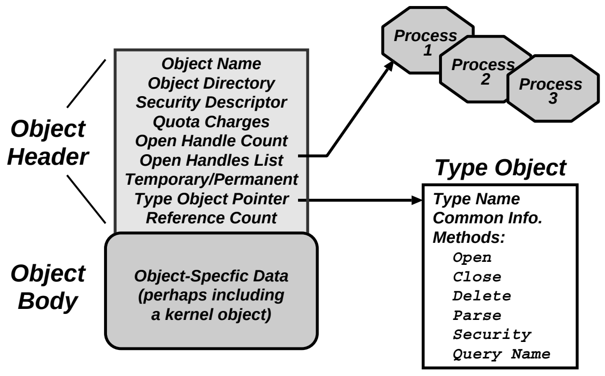
Threads are:

- co-operative: all threads in a process share address space & object handles.
- lightweight: require less work to create/delete than processes (mainly due to shared virtual address space).

## CPU Scheduling

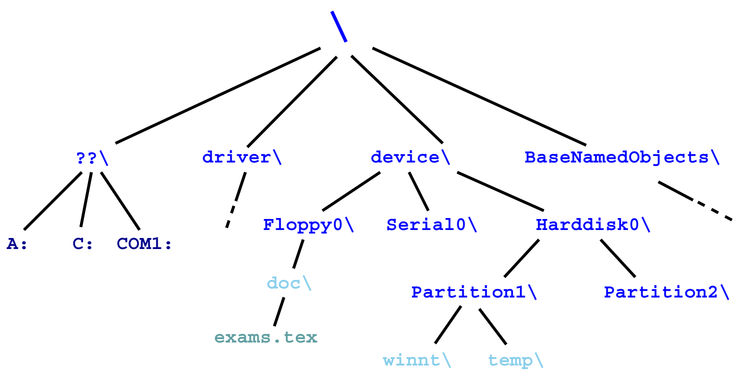
- Hybrid static/dynamic priority scheduling:
  - Priorities 16–31: “real time” (static priority).
  - Priorities 1–15: “variable” (dynamic) priority.
  - (priority 0 is reserved for zero page thread)
- Default quantum 2 ticks (∼20ms) on Workstation, 12 ticks (∼120ms) on Server.
- Threads have base and 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.
- If no runnable thread, dispatch ‘idle’ thread (which executes DPCs).

## 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 Manger manages a hierarchical namespace:
  - shared between all processes ⇒ sharing
  - implemented via directory objects
  - each object protected by an access control list.
  - naming domains (using `parse`) mean file-system namespaces can be integrated
- 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 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)
  - section objects are either based (specific base address) or non-based (floating)
  - also used for memory-mapped files

## Security Reference Manager

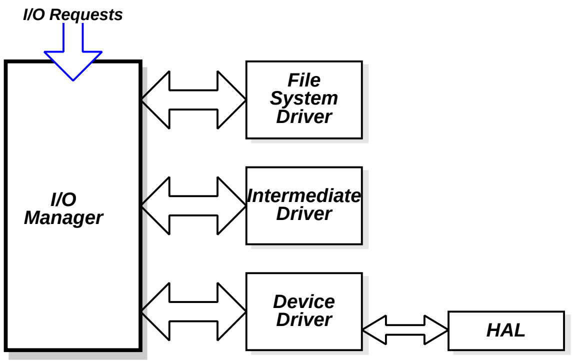
- NT's object-oriented nature enables a uniform mechanism for runtime access and audit checks
  - everytime a process opens handle to an object, check process's security token and object's ACL
  - compare with Unix (file-system, networking, window system, shared memory)

## Local Procedure Call Facility

- LPC (or IPC) passes requests and results between client and server processes within a single machine.
- Used to request services from the various NT environmental subsystems.
- Three variants of LPC channels:

1. small messages ($\leq$ 256 bytes): copy messages between processes
2. zero copy: avoid copying large messages by pointing to a shared memory section object created for the channel.
3. quick LPC: used by the graphical display portions of the Win32 subsystem.

## I/O Manager

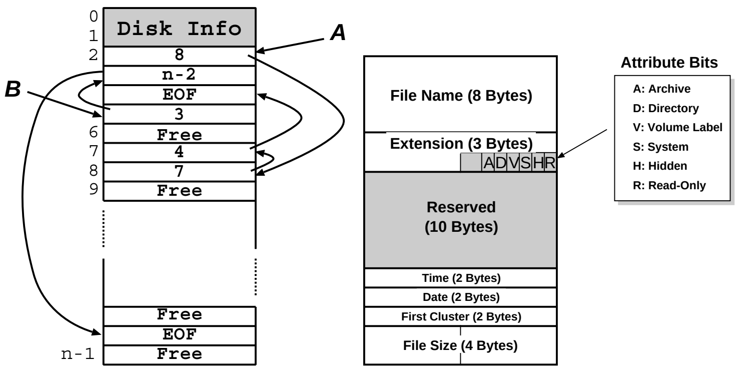


- The I/O Manager is responsible for:
  - file systems
  - cache management
  - device drivers
- Basic model is asynchronous:
  - each I/O operation explicitly split into a request and a response
  - an I/O Request Packet (IRP) used to hold parameters, results, etc.
- File-system & device drivers are stackable. . .

## Cache Manager

- Cache Manager caches "virtual blocks":
  - viz. keeps track of cache "lines" as offsets within a *file* rather than a volume.
  - disk layout & volume concept abstracted away.
  - ⇒ no translation required for cache hit.
  - ⇒ can get more intelligent prefetching
- Completely unified cache:
  - cache "lines" all live in the virtual address space.
  - decouples physical & virtual cache systems: e.g.
    - ∗ virtually cache in 256K blocks,
    - ∗ physically cluster up to 64K.
  - NT virtual memory manager responsible for actually doing the I/O.
  - so lots of FS cache when VM system lightly loaded, little when system thrashing
- NT also provides some user control:
  - if specify `temporary` attrib when creating file ⇒ data will never be flushed to disk unless absolutely necessary.
  - if specify `write_through` attrib when opening a file ⇒ all writes will synchronously complete.

## File Systems: FAT16



- A file is a linked list of clusters (= a set of $2^n$ contiguous disk blocks, $n \geq 0$)
- Each entry in the `FAT` contains either:
  - the index of another entry within the `FAT`, or
  - a special value `EOF` meaning "end of file", or
  - a special value `Free` meaning "free".
- Directory entries contain index into the `FAT`
- `FAT16` could only handle partitions up to $(2^{16} \times c)$ bytes ⇒ max 2Gb partition with 32K clusters (and big cluster size is *bad*)

## File Systems: FAT32

- Obvious extension: instead of using 2 bytes per entry, `FAT32` uses 4 bytes

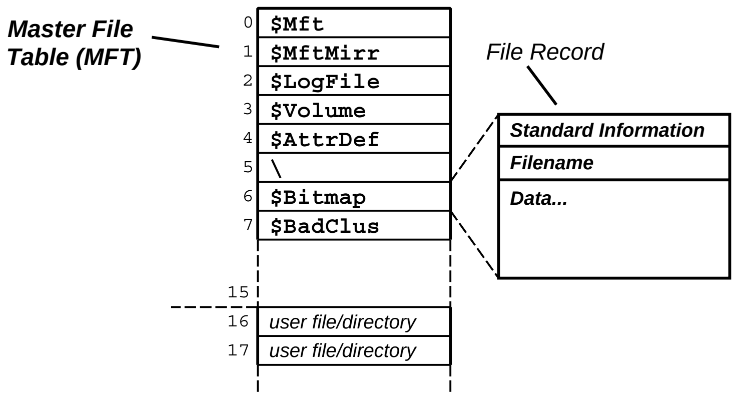
⇒ can support e.g. 8Gb partition with 4K clusters

- Further enhancements with `FAT32` include:
  - can locate the root directory anywhere on the partition (in `FAT16`, the root directory had to immediately follow the `FAT`(s)).
  - can use the backup copy of the `FAT` instead of the default (more fault tolerant)
  - improved support for demand paged executables (consider the 4K default cluster size . . . ).
- `VFAT` on top of `FAT32` adds long name support and internationalization:
  - names now unicode strings of up to 256 characters.
  - want to keep same directory entry structure for compatibility with e.g. DOS

  ⇒ use *multiple* directory entries to contain successive parts of name.
  - abuse `V` attribute to avoid listing these

Still pretty primitive. . .

## File-Systems: NTFS



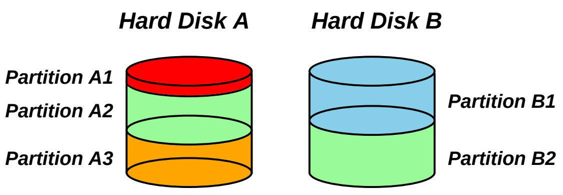
- Fundamental structure of NTFS is a volume:
  - based on a logical disk partition
  - may occupy a portion of a disk, and entire disk, or span across several disks.
- NTFS stores all file records in a special file called the Master File Table (MFT).
- The MFT is indexed by a file reference: a 64-bit unique identifier for a file
- A file itself is a structured object consisting of set of attribute/value pairs of variable length. . .

## NTFS: Recovery

- To aid recovery, all file system data structure updates are performed inside transactions:
  - before a data structure is altered, the transaction writes a log record that contains redo and undo information.
  - after the data structure has been changed, a commit record is written to the log to signify that the transaction succeeded.
  - after a crash, the file system can be restored to a consistent state by processing the log records.
- Does not guarantee that all the user file data can be recovered after a crash — just that metadata files will reflect some prior consistent state.
- The log is stored in the third metadata file at the beginning of the volume (`$Logfile`)
  - in fact, NT has a generic log file service

  ⇒ could in principle be used by e.g. database
- Overall makes for far quicker recovery after crash
- (modern Unix fs [ext3, xfs] use similar scheme)

## NTFS: Fault Tolerance



- `FtDisk` driver allows multiple partitions be combined into a logical volume:
  - e.g. logically concatenate multiple disks to form a large logical volume
  - based on the concept of RAID = **R**edundant **A**rray of **I**nexpensive **D**isks:
  - e.g. RAID level 0: interleave multiple partitions round-robin to form a stripe set:
    - ∗ logical block 0 → block 0 of partition A2, logical block 1 → block 0 of partition B2, logical block 2 → block 1 of partition A2, etc
  - e.g. RAID level 1 increases robustness by using a mirror set: two equally sized partitions on two disks with identical data contents.
  - (other more complex RAID levels also exist)
- `FtDisk` can also handle sector sparing where the underlying SCSI disk supports it
- (if not, NTFS supports cluster remapping in software)

## NTFS: Other Features

- Security:
  - security derived from the NT object model.
  - each file object has a security descriptor attribute stored in its MFT record.
  - this atrribute holds the access token of file owner plus an access control list
- Compression:
  - NTFS can divide a file's data into compression units (sets of 16 contiguous clusters in the file)
  - NTFS also has support for sparse files
    - ∗ clusters with all zeros not actually allocated or stored on disk.
    - ∗ instead, gaps are left in the sequences of VCNs kept in the file record
    - ∗ when reading a file, gaps cause NTFS to zero-fill that portion of the caller's buffer.
- Encryption:
  - Use symmetric key to encrypt files; file attribute holds this key encrypted with user public key
  - Not really that useful: private key pretty easy to obtain; and administrator can bypass entire thing anyhow.

## Environmental Subsystems

- User-mode processes layered over the native NT executive services to enable NT to run programs developed for other operating systems.
- NT uses the Win32 subsystem as the main operating environment
  - Win32 is used to start all processes.
  - Also provides all the keyboard, mouse and graphical display capabilities.
- MS-DOS environment is provided by a Win32 application called the virtual dos machine (VDM), a user-mode process that is paged and dispatched like any other NT thread.
  - Uses virtual 8086 mode, so not 100% compatible
- 16-Bit Windows Environment:
  - Provided by a VDM that incorporates Windows on Windows
  - Provides the Windows 3.1 kernel routines and stub routings for window manager and GDI functions.
- The POSIX subsystem is designed to run POSIX applications following the POSIX.1 standard which is based on the UNIX model.

## Summary

- Main Windows NT features are:
  - layered/modular architecture:
  - generic use of objects throughout
  - multi-threaded processes
  - multiprocessor support
  - asynchronous I/O subsystem
  - NTFS filing system (vastly superior to FAT32)
  - 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
  - "feature interaction", aka huge swathes of complex poorly implemented user-space code written by idiots
- Continues to evolve. . .

## Course Review

- Part I: Computer Organisation
  - "How does a computer work?"
  - The fetch-execute cycle, data representation, etc
  - MIPS assembly language
- Part II: Operating System Functions
  - OS structures: required h/w support, kernel vs. $\mu$-kernel
  - Processes: states, structures, scheduling
  - Memory: virtual addresses, sharing, protection
  - I/O subsytem: polling/interrupts, buffering.
  - Filing: directories, meta-data, file operations.
- Part III: Case Studies
  - Unix: file abstraction, command 'extensibility'
  - Windows NT: layering, objects, asynch. I/O.

## Glossary and Acronyms: A–H

| | |
|---|---|
| **AGP** | Advanced Graphics Port |
| **ALU** | Arithmetic/Logic Unit |
| **API** | Application Programming Interface |
| **ARM** | a 32-bit RISC microprocessor |
| **ASCII** | American Standard Code for Information Interchange |
| **BSD** | Berkeley Software Distribution (Unix variant) |
| **BU** | Branch Unit |
| **CAM** | Content Addressable Memory |
| **COW** | Copy-on-Write |
| **CPU** | Central Processing Unit |
| **DAG** | Directed Acyclic Graph |
| **DMA** | Direct Memory Access |
| **DOS** | **1.** a primitive OS (Microsoft); **2.** Denial of Service |
| **DRAM** | Dynamic RAM |
| **FCFS** | First-Come-First-Served (see also FIFO) |
| **FIFO** | First-In-First-Out (see also FCFS) |
| **Fork** | create a new copy of a process |
| **Frame** | chunk of physical memory (also *page frame*) |
| **HAL** | Hardware Abstraction Layer |

## Glossary and Acronyms: I–N

| | |
|---|---|
| **I/O** | Input/Output (also *IO*) |
| **IA32** | Intel's 32-bit processor architecture |
| **IA64** | Intel's 64-bit processor architecture |
| **IDE** | Integrated Drive Electronics (disk interface) |
| **IPC** | Inter-Process Communication |
| **IRP** | I/O Request Packet |
| **IRQ** | Interrupt ReQuest |
| **ISA** | **1.** Industry Standard Architecture (bus); **2.** Instruction Set Architecture |
| **Interrupt** | a signal from hardware to the CPU |
| **IOCTL** | a system call to control an I/O device |
| **LPC** | Local Procedure Call |
| **MAU** | Memory Access Unit |
| **MFT** | Multiple Fixed Tasks (IBM OS) |
| **MIPS** | **1.** Millions of Instructions per Second; **2.** a 32-bit RISC processor |
| **MMU** | Memory Management Unit |
| **MFT** | Multiple Fixed Tasks (IBM OS) |
| **MVT** | Multiple Variable Tasks (IBM OS) |
| **NT** | New Technology (Microsoft OS Family) |
| **NTFS** | NT File System |

## Glossary and Acronyms: O–SM

| | |
|---|---|
| **OS** | Operating System |
| **OS/2** | a PC operating system (IBM & Microsoft) |
| **PC** | **1.** Program Counter; **2.** Personal Computer |
| **PCB** | **1.** Process Control Block; **2.** Printed Circuit Board |
| **PCI** | Peripheral Component Interface |
| **PIC** | Programmable Interrupt Controller |
| **PTBR** | Page Table Base Register |
| **PTE** | Page Table Entry |
| **Page** | fixed size chunk of virtual memory |
| **Poll** | [repeatedly] determine the status of |
| **Posix** | Portable OS Interface for Unix |
| **RAM** | Random Access Memory |
| **ROM** | Read-Only Memory |
| **SCSI** | Small Computer System Interface |
| **SFID** | System File ID |
| **Shell** | program allowing user-computer interaction |
| **Signal** | event delivered from OS to a process |
| **SJF** | Shortest Job First |
| **SMP** | Symmetric Multi-Processor |

## Glossary and Acronyms: SR–X

| | |
|---|---|
| **SRAM** | Static RAM |
| **SRTF** | Shortest Remaining Time First |
| **STBR** | Segment Table Base Register |
| **STLR** | Segment Table Length Register |
| **System V** | a variant of Unix |
| **TCB** | **1.** Thread Control Block; **2.** Trusted Computing Base |
| **TLB** | Translation Lookaside Buffer |
| **UCS** | Universal Character Set |
| **UFID** | User File ID |
| **UTF-8** | UCS Transformation Format 8 |
| **Unix** | the first kernel-based OS |
| **VAS** | Virtual Address Space |
| **VLSI** | Very Large Scale Integration |
| **VM** | **1.** Virtual Memory; **2.** Virtual Machine |
| **VMS** | Virtual Memory System (Digital OS) |
| **VXD** | Virtual Device Driver |
| **Win32** | API provided by modern Windows OSes |
| **XP** | a recent OS from Microsoft |
| **x86** | Intel familty of 32-bit CISC processors |