[12] CASE STUDY: WINDOWS NT

OUTLINE

- Introduction
- Design Principles
- Design
  - Structural
  - HAL, Kernel
  - Processes and Threads, Scheduling
  - Environmental Subsystems
- Objects
  - Manager, Namespace
  - Other Managers: Process, VM, Security Reference, IO, Cache
- Filesystems
  - FAT16, FAT32, NTFS
  - NTFS: Recovery, Fault Tolerance, Other Features
- Summary

INTRODUCTION

- **Introduction**
- Design Principles
- Design
- Objects
- Filesystems
- Summary

PRE-HISTORY

Microsoft and IBM co-developed OS/2 — in hand-written 80286 assembly! As a result, portability and maintainability weren't really strong features so in 1988 Microsoft decided to develop a "new technology" portable OS supporting both OS/2 and POSIX APIs

- Goal: A 32-bit preemptive multitasking operating system for modern microprocessors

Originally, NT was supposed to use the OS/2 API as its native environment, but during development NT was changed to use the Win32 API, reflecting the popularity of Windows 3.0

NEW TECHNOLOGY

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
- Team grew to about 40 by the end, with overall effort of 100 person-years
- July 1993: first version (3.1) introduced. Sucked
- September 1994: NT 3.5 released, providing mainly size and performance optimisations
- May 1995: NT 3.51 with support for the Power PC, and more performance tweaks
- July 1996: NT 4.0 with "new" (Windows 95) look 'n' feel. Saw some desktop use but mostly limited to servers. Various functions pushed back into the kernel, notably graphics rendering

CONTINUED EVOLUTION

- Feb 2000: NT 5.0 aka Windows 2000. Borrows from windows 98 look 'n' feel. Provides server and workstation versions, latter of which starts to get wider use. Big push to finally kill DOS/Win9x family that fails due to internal politicking
- Oct 2001: Windows XP (NT 5.1) launched with home and professional editions. Finally kills Win9x. Several "editions" including Media Center [2003], 64-bit [2005]) and Service Packs (SP1, SP2). 45 million lines of code
- 2003: Server product 2K3 (NT 5.2), basically the same modulo registry tweaks, support contract and of course cost. Comes in many editions
- 2006: Windows Vista (NT 6.0). More security, more design, new APIs
- 2009: Windows 7 (NT 7.0). Focused more on laptops and touch devices
- 2012: Windows 8 (NT 8.0). Radical new UI with tiles, focused on touch at least as much as supporting mouse/keyboard
- 2013: Windows 8.1 (NT 8.1). Back off the UI a bit, more customisation
- 2015: Windows 10 (NT 10.0). More connectivity, for and between devices

DESIGN PRINCIPLES

- Introduction
- **Design Principles**
- Design
- Objects
- Filesystems
- Summary

KEY GOALS

- **Portability**: hence written in C/C++ with the HAL to hide low-level details
- **Security**: new uniform access model implemented via object manager, and certified to US DOD level C2
- **POSIX compliance**: believed this would win sales, but desire to support both POSIX and OS/2 (later WIN32) impacted overall design
- **Multiprocessor support**: most small OSs didn't have this, and traditional kernel schemes are less well suited
- **Extensibility**: because, sometimes, we get things wrong; coupled with the above point, most directly led to the use of a micro-kernel design
- **International support**: sell to a bigger market, meant adopting UNICODE as fundamental internal naming scheme
- **Compatibility with MS-DOS/Windows**: don't want to lose customers, but achieved partial compatibility only...

OTHER GOALS

- **Reliability**: NT uses hardware protection for virtual memory and software protection mechanisms for operationg system resources
- **Compatibility**: applications that follow the IEEE 1003.1 (POSIX) standard can be compiled to run on NT without changing the source code
- **Performance**: NT subsystems can communicate with one another via high-performance message passing
- **Preemption**: of low priority threads enable sthe system to respond quickly to external events
- Designed for **symmetrical multiprocessing**

THE RESULT

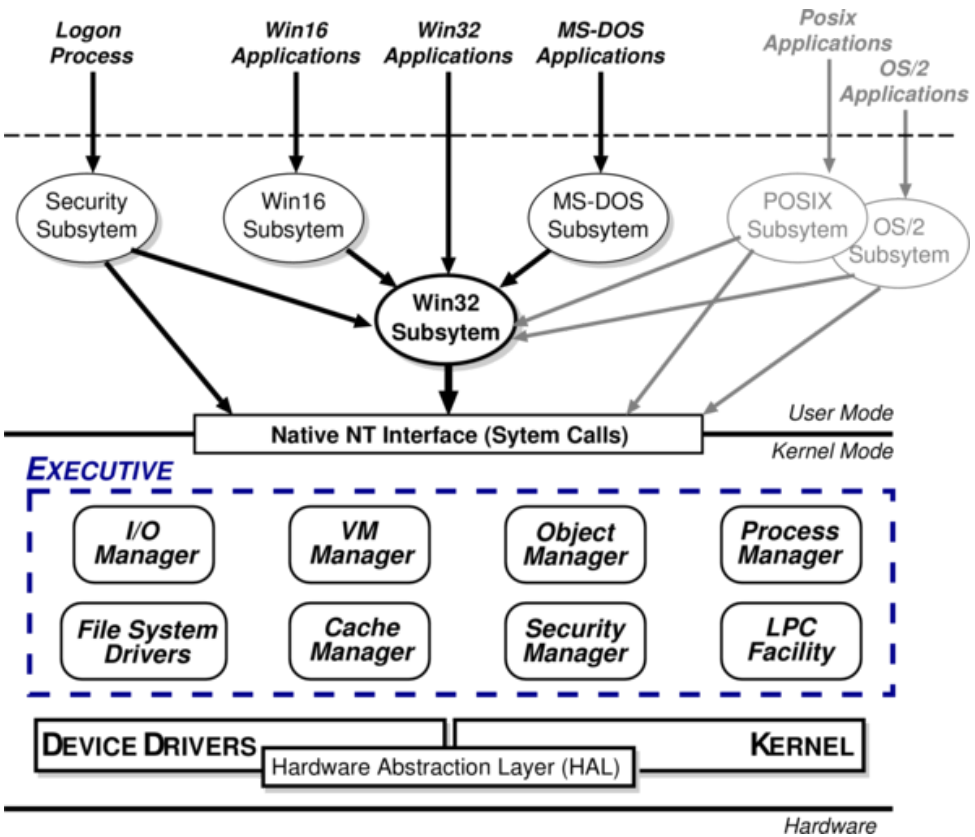
Development of a system which was:

- Written in high-level languages (C and C++)
  - Hence portable to other machines, with
  - Processor-dependent code isolated in a dynamic link library (HAL)
- Based around a micro-kernel
  - Hence extensibility and multiprocessor support
- Constructed in a layered/modular fashion
  - E.g. environmental subsystems

DESIGN

- Introduction
- Design Principles
- **Design**
  - **Structural**
  - **HAL, Kernel**
  - **Processes and Threads, Scheduling**
  - **Environmental Subsystems**
- Objects
- Filesystems
- Summary

STRUCTURAL OVERVIEW



Both layered and modular ("layered system of modules")

Interactions at top are message passing (IPC/LPC); next down is system calls (traps); below is direct invocation

Note that this is a static representation; in practice subsystems are DLLs (plus a few services); also have various threads running below

Kernel Mode: HAL, Kernel, & Executive

User Mode: environmental subsystems, protection subsystem

KERNEL MODE

Hardware Abstraction Layer (HAL): Layer of software (`HAL.DELL`) hiding hardware details, e.g., interrupt mechanisms, DMA controllers, multiprocessor communication mechanisms. Many implementations to the same interface

Kernel: Foundation for the executive and the subsystems, its execution is never preempted (it can be interrupted, but will always resume)

Four main responsibilities:

1. **CPU scheduling**: hybrid dynamic/static priority scheduling
2. **Interrupt and exception handling**: kernel provides trap handling when exceptions and interrupts are generated by hardware or software. If the trap handler can't handle the exception, the kernel's exception dispatcher does. Handle interrupts by either ISR or internal kernel routine
3. **Low-level processor synchronisation**: spin locks that reside in global memory to achieve multiprocessor mutual exclusion, normally provided by HAL
4. **Recovery after a power failure**

KERNEL

Kernel is **object oriented**; all objects either dispatcher objects or control objects

- Dispatcher objects have to do with dispatching and synchronisation, i.e. they are active or temporal things like
  - Threads: basic unit of [CPU] dispatching
  - Events: record event occurrences & synchronise
  - Timer: tracks time, signals "time-outs"
  - Mutexes: mutual exclusion in kernel mode
  - Mutants: as above, but work in user mode too
  - Semaphores: does what it says on the tin
- Control objects represent everything else, e.g.,
  - Process: representing VAS and miscellaneous other bits
  - Interrupt: binds ISR to an interrupt source [HAL]

PROCESSES AND THREADS

NT splits the *virtual processor* into two parts:

- A **process**, the unit of resource ownership. Each has:
  - A security token
  - A virtual address space
  - A set of resources (object handles)
  - One or more threads
- A **thread**, the unit of dispatching. Each has:
  - A scheduling state (ready, running, etc.)
  - Other scheduling parameters (priority, etc.)
  - A context slot
  - An associated process (generally)

Threads have one of **six states**: ready, standby, running, waiting, transition, terminated. They are **co-operative**: all in a process share the same address space & object handles; **lightweight**: less work to create/delete than processes (shared virtual addresss spaces)

CPU SCHEDULING

A process starts via the `CreateProcess` routine, loading any dynamic link libraries that are used by the process and creating a primary thread. Additional threads can be created via the `CreateThread` function

Hybrid static/dynamic priority scheduling:

- Priorities 16—31: "real time" (static) priority
- Priorities 1—15: "variable" (dynamic) priority
- Priority 0 is reserved for the *zero page thread*

Default quantum 2 ticks (~20ms) on Workstation, 12 ticks (~120ms) on Server

CPU SCHEDULING

Some very strange things to remember:

- When thread blocks, it loses 1/3 tick from quantum
- When thread preempted, moves to head of own run queue

Threads have base and current ($\geq$base) priorities.

- On return from IO, 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)
- Foreground thread gets double or triple quantum

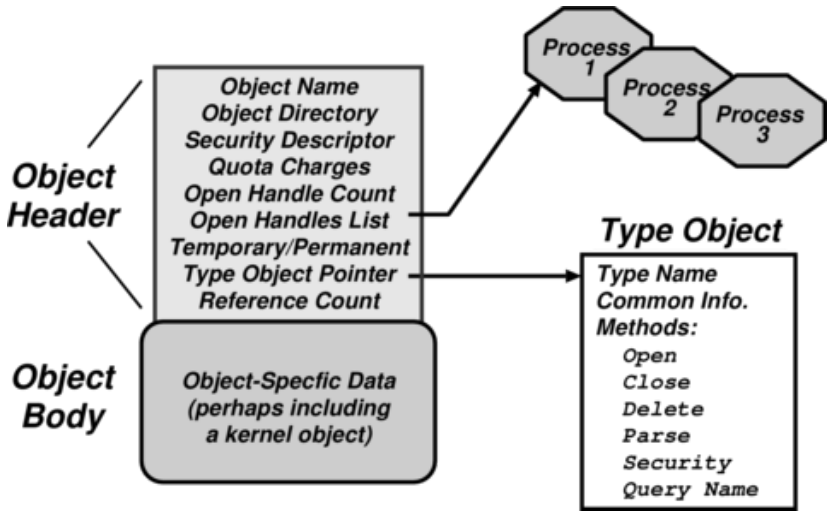
ENVIRONMENTAL SUBSYSTEMS

- 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. It 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
- 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

OBJECTS

- Introduction
- Design Principles
- Design
- **Objects**
  - **Manager, Namespace**
  - **Other Managers: Process, VM, Security Reference, IO, Cache**
- Filesystems
- Summary

OBJECTS AND MANAGERS



In Unix, everything is a file — in NT, everything is an object

- Every resource in NT is represented by an **(executive) object**
- Kernel objects are re-exported at executive level by encapsulation
- Objects comprise a header and a body, and have a type (approximately 15 types in total)

THE OBJECT MANAGER

Responsible for:

- Creating and tracking objects and object handles. An object handle represents an open object and is process-local, somewhat analogous to an fd
- Performing security checks

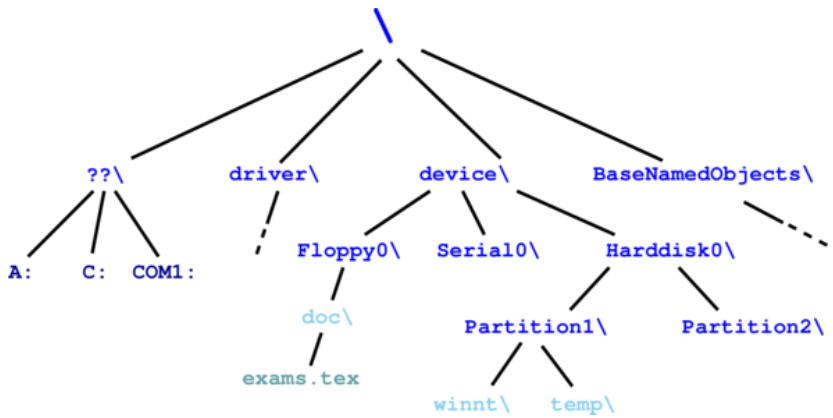
- Objects are manipulated by a standard set of methods, namely `create`, `open`, `close`, `delete`, `query name`, `parse` and `security`. These are usually per type ("class") and hence implemented via indirection through the associated type object. Not all will be valid (specified) for all object types
  - `handle = open(objectname, accessmode)`
  - `result = service(handle, arguments)`

- A process gets an object handle by creating an object, by opening an existing one, by receiving a duplicated handle from another process, or by inheriting a handle from a parent process

THE OBJECT NAMESPACE

Objects (optionally) have a name, temporary or permanent, given via the NT executive

The Object Manger manages a hierarchical namespace, shared between all processes. The namespace is implemented via directory objects analogous to filesystem directories



Each object is protected by an access control list. Naming domains (implemented via `parse`) mean filesystem namespaces can be integrated

Object names structured like file path names in MS-DOS and Unix. Symbolic link objects allow multiple names (aliases) for the same object. Modified view presented at API level: the Win32 model has multiple "root" points (e.g., `C:`, `D:`, etc) so even though was all nice & simple, gets screwed up

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

...thus can support Posix, OS/2 and Win32 models

- It's up to environmental subsystem that owns the process to handle any hierarchical relationships (e.g. inheritance, cascading termination, etc)
- E.g., as noted above, in Win32: a process is started via the `CreateProcess()` function which loads any dynamic link libraries that are used by the process and creates a primary thread; additional threads can be created by the `CreateThread()` function

VIRTUAL MEMORY MANAGER

Assumes that the underlying hardware supports virtual to physical mapping, a paging mechanism, transparent cache coherence on multiprocessor systems, and virtual address aliasing. NT employs **paged virtual memory management**, The VMM provides processes with services to:

- Allocate and free virtual memory via two step process: reserve a portion of the process's address space, then commit the allocation by assigning space in the NT paging file
- Modify per-page protections, in one of six states: valid, zeroed, free, standby, modified and bad
- Share portions of memory using **section objects** (~software segments), based verus non-based, as well as memory-mapped files
- A **section object** is a region of [virtual] memory which can be shared, containing: max size, page protection, paging file (or mapped file if mmap) and based vs non-based (meaning does it need to appear at same address in all process address spaces (based), or not (non-based)?)

SECURITY REFERENCE MANAGER

NT's object-oriented nature enables a uniform mechanism for runtime access and audit checks

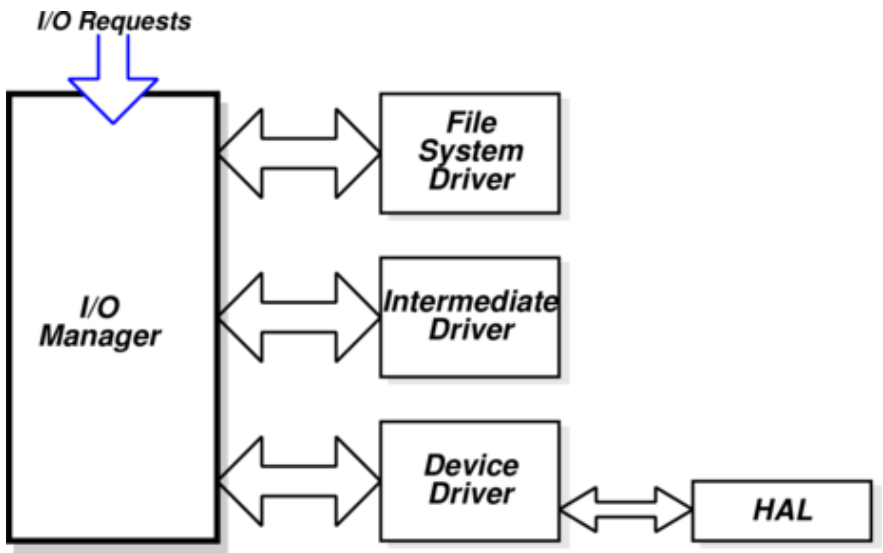
- Every time a process opens handle to an object, check that process's security token and object's ACL
- Compare with Unix (filesystem, networking, window system, shared memory, ...)

LOCAL PROCEDURE CALL FACILITY

Local Procedure Call (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

IO MANAGER



The IO Manager is responsible for file systems, cache management, device drivers

Keeps track of which installable file systems are loaded, manages buffers for IO requests, and works with VMM to provide memory-mapped files

Controls the NT cache manager, which handles caching for the entire IO system (ignore network drivers for now)

IO OPERATIONS

Basic model is asynchronous:

- Each IO operation explicitly split into a request and a response
- **IO Request Packet** (IRP) used to hold parameters, results, etc.

This allows high levels of flexibility in implementing IO type (can implement synchronous blocking on top of asynchronous, other way round is not so easy)

Filesystem & device drivers are stackable (plug'n'play)

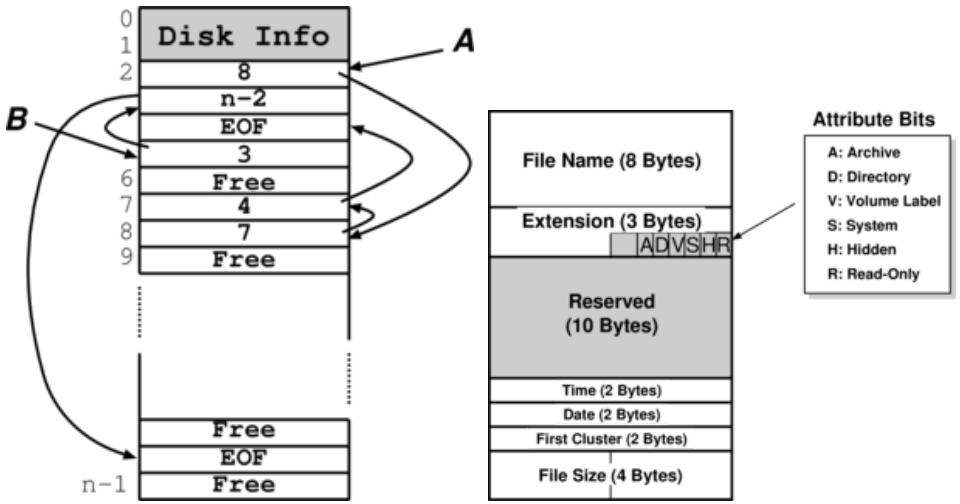
CACHE MANAGER

- Caches "virtual blocks", keeping 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 in virtual address space.
  - Decouples physical & virtual cache systems: e.g. virtually cache in 256kB blocks, physically cluster up to 64kB
- NT virtual memory manager responsible for actually doing the IO
  - Allows lots of FS cache when VM system lightly loaded, little when system is thrashing
- NT/2K also provides some user control:
  - If specify temporary attrib when creating file means it will never be flushed to disk unless necessary
  - If specify write through attrib when opening a file means all writes will synchronously complete

FILESYSTEMS

- Introduction
- Design Principles
- Design
- Objects
- **Filesystems**
  - **FAT16, FAT32, NTFS**
  - **NTFS: Recovery, Fault Tolerance, Other Features**
- Summary

FILE SYSTEMS: FAT16



FAT16 (originally just "FAT") is a floppy disk format from Microsoft (1977) but was used for hard-disks up to about 1996. It's quite a simple file system which basically uses the "chaining in a map" technique described in lectures to manage files

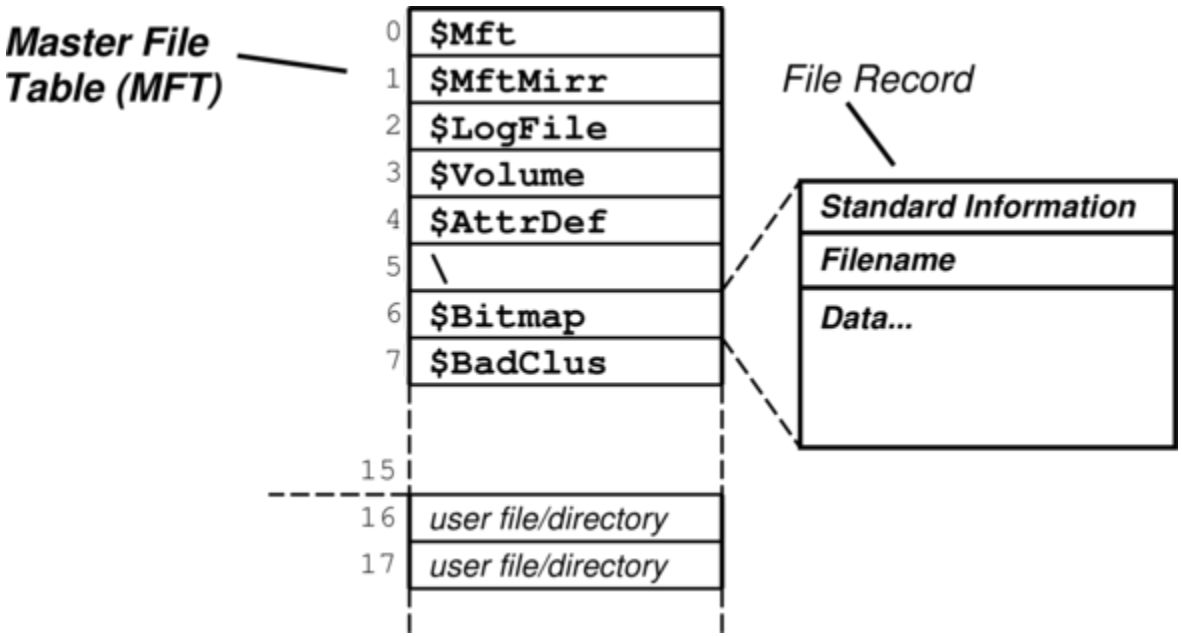
A file is a linked list of clusters: a cluster is a set of 2^n contiguous disk blocks, $n \geq 0$. Each entry in the FAT contains either: the index of another entry within the FAT, or a special value EOF meaning "end of file", or a special value Free meaning "free". Directory entries contain index into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

FILE SYSTEMS: FAT32

- Obvious extetension: instead of using 2 bytes per entry, FAT32 uses 4 bytes per entry, so can support e.g. 8Gb partition with 4kB 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 4kB default cluster size)
  - VFAT on top of FAT32 does long name support: unicode strings of up to 256 characters
  - Want to keep same directory entry structure for compatibility with, e.g., DOS so use multiple directory entries to contain successive parts of name
  - Abuse V attribute to avoid listing these

Still pretty primitive...

FILESYSTEMS: 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 FORMAT

NTFS uses clusters as the underlying unit of disk allocation:

- A cluster is a number of disk sectors that is a power of two
- Because the cluster size is smaller than for the 16-bit FAT file system, the amount of internal fragmentation is reduced
- NTFS uses logical cluster numbers (LCNs) as disk addresses
- The NTFS name space is organized by a hierarchy of directories; the index root contains the top level of the B+ tree

An array of file records is stored in a special file called the **Master File Table** (MFT), 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:

- Each file on an NTFS volume has a unique ID called a file reference: a 64-bit quantity that consists of a 16-bit file number and a 48-bit sequence number
- used to perform internal consistency checks
- MFT indexed by file reference to get file record

NTFS: RECOVERY

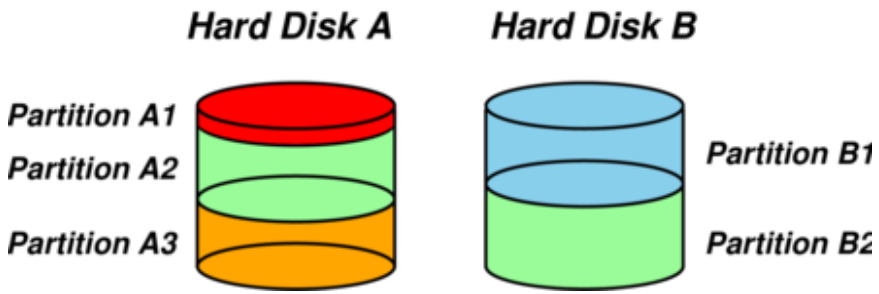
To aid recovery, all file system data structure updates are performed inside transactions:

- 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`):

- NT has a generic log file service that could be used by e.g. databases
- Makes for far quicker recovery after crash
- Modern Unix filesystems eg., `ext3`, `xfs` use a similar scheme

NTFS: FAULT TOLERANCE



`FtDisk` driver allows multiple partitions be combined into a logical volume:

- Logically concatenate multiple disks to form a large logical volume, a volume set
- Based on the concept of RAID = Redundant Array of Inexpensive Disks
- E.g., RAID level 0: interleave multiple partitions round-robin to form a stripe set
- 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 s/w cluster remapping

NTFS: OTHER FEATURES (I)

Security

- Security derived from the NT object model
- Each file object has a security descriptor attribute stored in its MFT record
- This atrribute contains the access token of the owner of the file plus an access control list

Compression

- NTFS can divide a file's data into compression units (blocks of 16 contiguous clusters) and supports sparse files
  - Clusters with all zeros not allocated or stored
  - 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

NTFS: OTHER FEATURES (I)

Encryption

- Use symmetric key to encrypt files; file attribute holds this key encrypted with user public key
- Problems:
  - Private key pretty easy to obtain; and
  - Administrator can bypass entire thing anyhow

SUMMARY

- Introduction
- Design Principles
- Design
- Objects
- Filesystems
- **Summary**

SUMMARY

Main Windows NT features are:

- Layered/modular architecture
- Generic use of objects throughout
- Multi-threaded processes & multiprocessor support
- Asynchronous IO 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
- Continues to evolve: Singularity, Drawbridge, Windows 10, ...

SUMMARY

- Introduction
- Design Principles
- Design
  - Structural
  - HAL, Kernel
  - Processes and Threads, Scheduling
  - Environmental Subsystems
- Objects
  - Manager, Namespace
  - Other Managers: Process, VM, Security Reference, IO, Cache
- Filesystems
  - FAT16, FAT32, NTFS
  - NTFS: Recovery, Fault Tolerance, Other Features
- Summary