CS2106

MEMORY MANAGEMENT

Memory regions

- Text (for instructions)
- Data (for global variables)
- Heap (for dynamic allocation)
- Stack (for function invocations)

Types of data

- Transient: valid only for a limited duration (e.g. function parameters, local variables)
- Persistent: valid for duration of program or until explicitly deallocated (e.g. global variables, dynamically allocated memory)

Memory abstraction

No abstraction

✓ Address in program == physical address
✓ No conversion/mapping required
✓ Address fixed at compile time
× Hard to protect memory space
× Multiple processes might occupy same physical locations

Address relocation

- Recalculate memory address at load time, so processes correctly access memory

× Slow loading time
× Not easy to distinguish memory reference from normal integer constant

Base + Limit registers

- All memory references are compiled as an offset from the base register
- Compiled memory references are checked against the limit regsiter for validity

× Every memory access incurs an addition and a comparison

Logical address

- Is how the process views its memory space
- Is different from physical address, so a mapping is needed
- Each process has a self-contained, independent logical memory space

CONTIGUOUS MEMORY

Memory partitioning

Fixed-size partition

- Physical memory is split into fixed number of partitions of equal size
- A process will occupy one of the partitions

✓ Easy to manage
✓ Fast to allocate, since every free partition is the same
✓ No external fragmentation
× Partition needs to be large enough to contain the largest of processes
× Internal fragmentation occurs when process does not occupy entire partition (i.e. allocated but unused)

Variable-size partitions

- Partition is created based on actual size of process
- OS keeps track of occupied and free regions, performing splitting and merging as necessary

✓ Flexible, no internal fragmentation
× Need to maintain more info
× Takes more time to find appropriate region
× External fragmentation occurs when there are free regions that are too small to be allocated (due to allocate/free patterns)

Dynamic allocation algorithms

Reducing external fragmentation

- (Merging) Merge adjacent free partitions
- (Compaction) Move occupied partitions around to create bigger, consolidated holes

Linear search

✓ No internal fragmentation
× Has external fragmentation

First fit Take first hole that is large enough

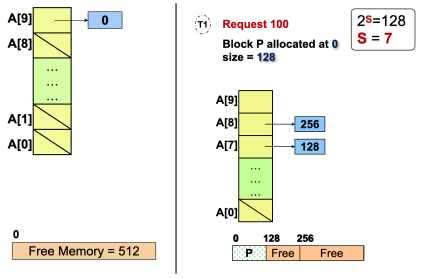
Best fit Take smallest hole that is large enough

Worst fit Take largest hole

Next fit Like first fit, but upon next allocation request, resume searching from the partition where the previous allocation was performed

✓ More uniform distribution of hole sizes
✓ Faster alllocation
× Needs to store an additional pointer to the place of last allocation

Buddy system



Each A[i] is a list of blocks of size 2^i, where each block indicates starting address

Allocate To allocate size N, find smallest S such that $2^S \geq N$, then repeatedly split until there is a block of size 2^S.

Deallocate During deallocation, if buddy is free, merge with buddy and repeat. Otherwise, add the block to the relevant linked list

Check buddy If two blocks are of size 2^S, compare their start values. They are buddies $\iff$ only bit $S+1$ is different

Fragmentation

- Internal fragmentation (since blocks of fixed size 2^K are allocated, but $< 50\%$)
- External fragmentaiton (little but possible)

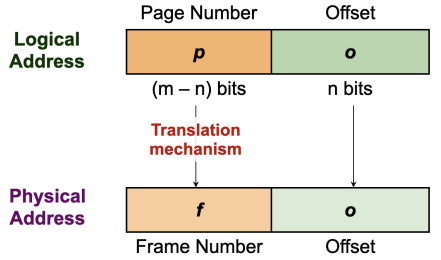
DISJOINT MEMORY

Paging

- Physical memory is split into regions of fixed size (known as physical frames)
- Logical memory is split into regions of same size as physical frames (known as logical page)
- Logical memory space is contiguous, but physical memory region might be disjoint

Address translation

To keep things simple, frame size = page size = power of 2



Properties

✓ No external fragmentation
✓ Can have internal fragmentation, when logical memory space is not a multiple of page size, but it is insignificant (max one page per process is not fully utilized)
✓ Clean separation of logical and physical address space

Page table

- Maps logical page number to physical frame number
- Part of process memory context (store pointer to page table)

× Requires two memory accesses for each memory reference

1. Get frame number from page table
2. Access actual item

Translation Look-Aside Buffer (TLB)

- Cache for page table entries (PTEs)
- Very small (tens of entries) and fast (≤ 1 clock cycle)
- Not part of process hardware context

Memory access time Let p be the probability of a TLB hit.

$$\text{Latency(TLB hit)} = \text{TLB} + \text{Mem}$$
$$\text{Latency(TLB miss)} = \text{TLB} + 2\text{Mem}$$

$$\begin{aligned}&\text{Average memory access time}\\ &= p \times \text{Latency(TLB hit)} + (1-p) \times \\ &\quad \text{Latency(TLB miss)}\\ &= p(\text{TLB} + \text{Mem}) + (1-p)(\text{TLB} + 2\text{Mem})\\ &= \text{TLB} + \text{Mem}(2-p)\end{aligned}$$

Protection

Access-right bits

- Each memory access is checked against the PTE's rwx bits in hardware

Valid bit

- Each memory access is checked against the valid bit in hardware
- Out of range access is caught by OS

Page sharing

- Several processes might use the same physical memory frame (e.g. referencing C standard library)
- Can be used to implement copy-on-write

Segmentation

Motivation

Each memory region (text, data, heap, stack)

- May grow/shrink at execution time (hard to achieve if whole process is one piece)
- Hard to check if memory access in a region is in-range or not
- May have different usage, permissions, lifetime, scope, etc.

Idea

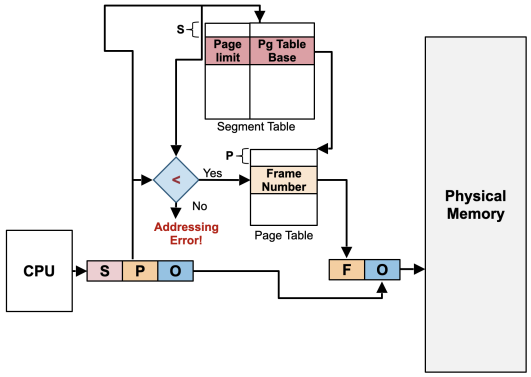
- Each memory segment is a contiguous memory region, specified using `(segment id, base, limit)`
- Memory references specified as `(segment id, offset)`
- Physical address is specified using `(base, offset)`
- `offset < limit` for valid access

× Can cause external fragmentation

Segmentation with paging

Idea

- Each segment has a page table, and is composed of several pages
- Segment can grow by allocating page and adding to page table (same for shrinking)



VIRTUAL MEMORY

Extended paging scheme

Motivation

Logical space can be larger than physical space

Idea

- Split logical memory into fixed size pages
- Some pages are in physical memory, some in secondary storage

Page table On top of storing frame number, page table entry now also needs to include resident bit - whether the page is in physical memory

Page fault

- Occurs when CPU tries to access a non-memory resident page
- OS needs to bring a non-memory resident page into physical memory (counts as I/O)

Issues

Secondary storage access time

- 5 orders of magnitude slower than physical memory access time

Thrashing

- Occurs when page fault happens too often
- Should not happen in programs that behave well, due to temporal/spatial locality

Access virtual address algorithm

- Note: page fault counts as I/O

Algorithm for accessing VA

- Done in hardware

1. VA is decomposed into `<Page#, Offset>`
2. Search TLB for `<Page#>`, and if TLB miss, TRAP to OS (TLB Fault and retry algo)
3. Is `<Page#>` memory resident?, and if not, TRAP to OS (Page Fault and retry algo)
4. Use `<Frame#><Offset>` to access physical memory

TLB Fault Handler

- Done in OS, assumes software managed TLB

1. Access full table of process (in PCB), find `<Page#>`
2. If valid bit is 0, then segmentation fault
3. Else load relevant PTE into TLB
4. Return from TRAP

TLB Fault Handler (Hardware)

- Done in hardware, assumes hardware managed TLB

1. Base address of full page tabel is stored in a dedicated register
2. Processor retrieves missing PTE from memory directly (as it is a simple offset from the base of page table)

Page Fault Handler

- Done in OS

1. Check if global or local replacement
2. Write out the page to be replaced if needed
3. Locate the page in secondray storage
4. Load the page into a physical frame
5. Update PTE
6. Update TLB
7. Return from TRAP

Demand paging

- Process starts with no memory resident page
- Only allocate page when there is a page fault

✓ Fast startup for new process

✓ Small memory footprint

× Process may appear sluggish at the start due to multiple page faults

× Page faults may have cascading effect on other processes (i.e. thrashing)

Page table structure

Assume that there are 2^P pages in logical memory space

Direct paging

- Keep all 2^P entries in a single table

× For efficient retrieval of the pth entry, page tables must be contiguous in OS memory

2-level paging

- Process may not use entire virtual memory space, so it will not fully occupy direct paging table
- Page the page table

✓ Need not be contiguous in OS memory

✓ There can be "empty" pages, so that less memory is used to store the entire table

× Need 2 memory access to get frame number

Page directory contains 2^M entries pointing to 2^M page tables

2nd level (page table) contains 2^{P-M} entries each

Memory Mangement Unit (MMU) cache caches frequent page directory entries (TLB caches PTEs)

Inverted page table

- Single table for all processes
- Each entry contains: `<pid, page#>` (maybe also row number)

× Slow translation, as you need to search the whole table

- In practice, used as auxiliary structure, to answer questions like: who are all the sharers of physical frame X?

Page replacement algorithms

- No free physical memory frame during page fault ⇒ evict a memory page
- Dirty page ⇒ modified ⇒ need to write back

Optimum (OPT)

- Replace page that will not be needed again for the longest period of time

× Not feasible as it requires future knowledge of memory references

FIFO

- Evict oldest memory page

✓ Simple to implement

× Bad performance in practice

× More RAM doesn't necessarily decrease page faults (Belady's Anomaly)

Belady's Anomaly
try 3/4 frames with 1 2 3 4 1 2 5 1 2 3 4 5

Least Recently Used (LRU)

- Replace page that has not been used in the longest time

✓ Exploits temporal locality

✓ Does not suffer from Belady's Anomaly

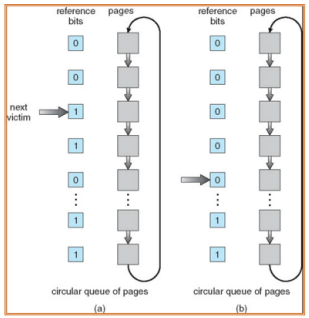
× Need substantial hardware support to implement

Implementation Need to keep track of last access time

1. Store `lastUsedTime` in PTE, updating on every access
   - `lastUsedTime` incremented for every memory reference
   - × `lastUsedTime` may overflow
   - Update: update time on every access
   - Query: search all pages for earliest time
2. Maintain "stack"
   - Update: if entry exists, pop from stack. Then push onto stack
   - Query: remove bottom of stack
   - × Not a pure stack, hard to implement in hardware

Second chance page replacement (CLOCK)

- General Idea:
  - Modified FIFO to give a second chance to pages that are accessed
  - Each PTE now maintains a "reference bit":
    - 1 = Accessed, 0 = Not accessed
  - Algorithm:
    1. The oldest FIFO page is selected
    2. If reference bit == 0 ➔ Page is replaced
    3. If reference bit == 1 ➔ Page is given a 2nd chance
       - Reference bit cleared to 0
       - Arrival time reset ➔ page taken as newly loaded
       - Next FIFO page is selected, go to Step 2
  - Degenerate into FIFO algorithm
    - When all pages has reference bit == 1



- Use **circular queue** to maintain the pages:
  - With a pointer pointing to the oldest page (the **victim page**)
- To find a page to be replaced:
  - Advance until a page with '0' reference bit
  - Clear the reference bit as pointer passes through

Frame allocation

Simple approaches

Equal allocation process gets same amount of frames

Proportional allocation process gets frames proportional to memory usage

Local/Global replacement

Local Evicted page selected from the process

✓ Thrashing limited to the process (but it can use up I/O bandwith and degrade performance of other processes)

× Insufficient frames can hinder progress

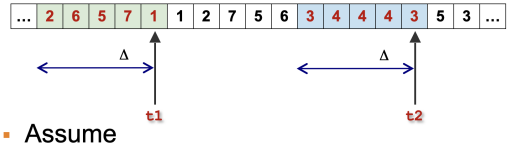
Global Evicted page selected from all frames

✓ Process that needs more frames can get from others that need less

× Can cause thrashing in other processes (cascading thrashing)

Working set model

- Define working set $W(t, \Delta)$ as the set of active pages in the interval at time t
- Allocate sufficient frames for pages in $W(t, \Delta)$ to reduce chance of page fault
- Relatively constant in a period of time (locality)
- Too small: may miss pages in current working set
- Too big: may contain pages from a different working set



- Assume
  - Δ = an interval of 5 memory references
- `W(t1,Δ)={1,2,5,6,7}` (5 frames needed)
- `W(t2,Δ)={3,4}` (2 frames needed)

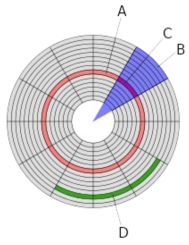
FILE SYSTEM MANAGEMENT

Misc

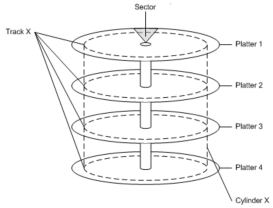
Motivation

- Physical memory is volatile (only maintains data while powered), so we want to use external storage to store persistent info
- Provide an abstraction over external storage
- Provide protection and sharing bewteen processes and users

Hard disk layout



A = Track
B = Geometric Sector
C = Track Sector
D = Cluster



General criteria

Self-contained, Persistent, Efficient

File

- Contains data
- Contains metadata (name, identifier, type, size, protection, time/date/owner info, table of content)

File name

Different FS has different naming rules (length, case sensitivity, special symbols, extension)

File type

- Each file type has an associated set of operations; possibly a specific program for processing
- Regular files, directories, special files

Regular files

- ASCII files: can be printed as is (text file, source code)
- Binary files: have a predefined structure to be processed by a specific program (executable, Java class file, pdf file)

Distinguishing file type

- Windows uses file extension
- Unix uses embedded info in the file (magic number)

File size

Usually does not include file metadata

File protection

Read, Write, Execute, Append, Delete, List (Read metadata of a file)

Permission bits

- Classify users into Owner/Group/Universe
- Unix stores RWX for each of the 3 classes of users

Tut 10/11 DirExp
Think of directory as the list of directory entries

- Read = Can you read this list (ls, tab autocompletion)
- Write = Can you change this list (create, rename, delete entry)
- Execute = Can you use this directory as WD (cd)

Action	`r-x`	`-wx`	`--x`
`ls -l DIR`	ok	no	no
`cd DIR`	ok	ok	ok
After `cd`			
`ls -l`	ok	no	no
`cat curfile`	ok	ok	ok
`touch curfile`	ok	ok	ok
`touch newfile`	nope	ok	nope

File metadata

Rename, Change attributes, Read attributes

File data

- Array of bytes
- Fixed length records (can jump to any record easily)
- Variable length records (flexible but harder to locate a record)

Records

- In C files, each record is a character
- In a SQL database, each record is a row in the table, and it could be fixed length if there are no variable length string attributes
- In video files, each record is a compressed frame (simplified), which has variable length, depending on the compression rate of the individual frames

Access methods

- Sequential: read in order
- Random: read in any order
  - `Read(Offset)`: explicitly state position to access
  - `Seek(Offset)`: special op to move to a new location in file
- Direct: allow random access to any record

Generic operations Create, Open, Read, Write, Repositioning (Seek), Truncate

File operations

OS provides file ops as syscalls:

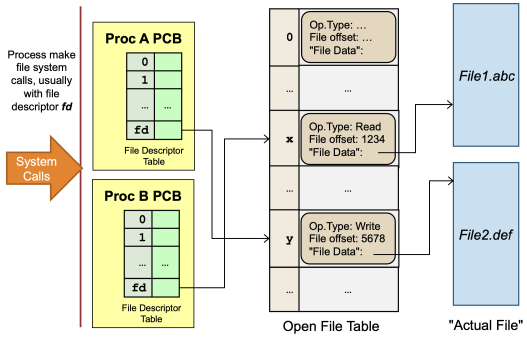
- Provide protection, concurrent and efficient access
- Maintain info

Info to keep for an opened file

File pointer, Disk location, Open count (how many processes has this file opened)

Open-file table

- System-wide file table: one entry per unique file
- Process file table: one entry per file used in process, each entry points to the system-wide table



Process sharing

- Two processes with different FDs pointing to the same file can do I/O independently
- `fork()`: Two processes with the same FD will share 1 FD

Directory

Purpose

Provide a logical grouping of files, Keep track of files

Structure

Single-level, Tree, DAG, General graph

Links

Hard link

- Maintain separate pointers to the same file
- Only for files, creates DAG

Soft link

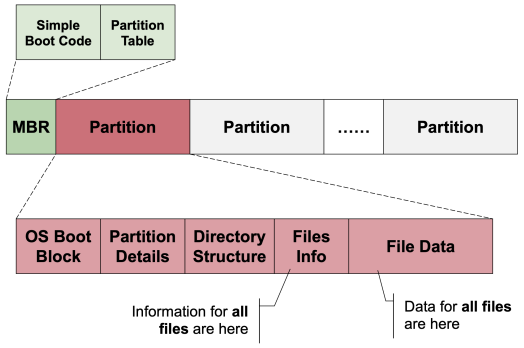
- A file that contains the path name of the file to link to (essentially like a shortcut)
- Files and directories, creates general graph

Unix symbolic link (soft link) Symlinks always have permissions 777 (so anyone may use the symlink), but the actual access permissions are checked against the original file linked to

FILE SYSTEM IMPLEMENTATIONS

Generic disk organization

Note: MBR is Master Boot Record



File block allocation

Note: all can have internal fragmentation, if the last block is not full (at least by CS2106 definitions)

Contiguous

- Allocate consecutive disk blocks to a file
- Each file stores start block and length

✓ Simple to keep track

✓ Fast access (only need to seek to first block)

× External fragmentation

× File size needs to be specified in advance

Linked list

- Each disk block additionally stores next disk block number (i.e. pointer)
- Each file stores first block and last block

✓ No external fragmentation

× Random access in a file is very slow

× Part of disk block is used for pointer

× Less reliable (if one of the pointers is incorrect, you might lose the rest of the data)

File Allocation Table (FAT)

- Linked list v2.0
- Store all disk block pointers in a single table (FAT), which is in memory all the time

✓ Faster random access (than linked list)

× Keeps track of all disk blocks in a partition (can be huge, consumes valuable memory space)

Indexed

- Each file has an index block, an array of disk block addresses of the disks that contain the file data

✓ Lesser memory overhead than FAT (only index block of opened file needs to be in memory)

✓ Fast direct access

× Limited max file size (limited to block size)

× Index block overhead

Indexed + linked list

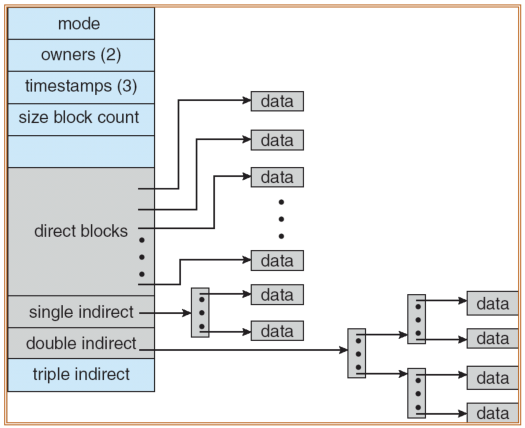
- Keep a linked list of index blocks
- Each index block contains the pointer to the next index block

Multilevel indexing

- Like multilevel paging but for blocks
- Can be generalized to any number of levels, but if a scheme is decided, then there is a limit to the file size

Unix I-node

12 direct pointers, 1 single indirect block, 1 double indirect block, 1 triple indirect block



Free space management

To manage free disk blocks

Bitmap

- Each disk block is represented by 1 bit
- e.g. 1=free, 0=occupied

✓ Powerful bit manipulations

× Need to keep the bitap in memory for efficiency reasons

Linked list

- Linked list of disk blocks
- Each node (disk block) contains some free disk block numbers, and a pointer to the next free space disk block

✓ Easy to find free block

✓ Only need the first pointer in memory (can cache the other blocks for efficiency)

× High overhead (can be mitigated by storing the free block list in free blocks)

Directory

Traversal

- Given a full path name
- Recursively check existence of each directory along the path to arrive at file info
- Stop if not found (or incorrect type)

Linked list

- Each entry represents a file (file name, pointer to file info / actual file info)

× Locating a file requires linear search (but can cache searches)

Hash table

- Each directory has a hash table of size N
- Hash file name to locate file
- Usually use hash table with chaining

✓ Fast lookup

× Hash table has limited size

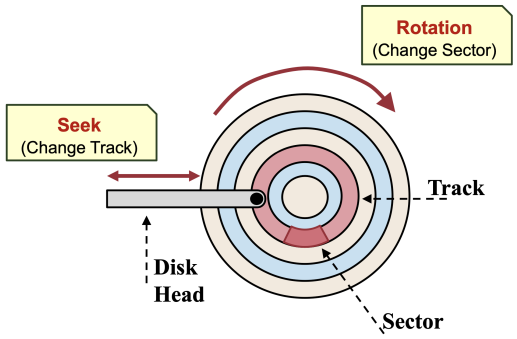
× Depends on good hash function

File information

Consists of file name, metadata, disk block info

1. Store everything in directory entry, or
2. Store file name and point to another DS for other info

Disk scheduling



I/O latency

1. Position disk head over proper track (Seek time)
   - Position the disk head over the proper track
   - Average typically around 2ms to 10ms
   - Expected seek time is $T/3$ where T is the time for max seek distance
2. Wait for desired sector to rotate under the read/write head (Rotational latency)
   - 4800 to 15000 RPM (12.5ms to 4ms per rotation)
   - Assume desired data is halfway around track, then average rotational latency is 6.25ms at 4800RPM, 2ms at 15000RPM
3. Transfer the sector(s) from disk to memory (Transfer time)
   - **Transfer Time** is a function of :
     - Transfer size / Transfer Rate
       - Transfer size: [X KiB / Sector] x [# of Sectors]
       - Transfer rate: 70 to 125 MiB / second
     - E.g. If we read 2 **consecutive sectors** of 512 Bytes
       - Transfer size = 512B x 2 = 1KiB
       - Assuming a transfer rate of 100MiB/second
       - Transfer time = 1KiB / 100MiB per second
         = 2^{10} / 100 x2^{20} = 9.7µs
   - (1) and (2) are 3 orders of magnitude slower than (3)

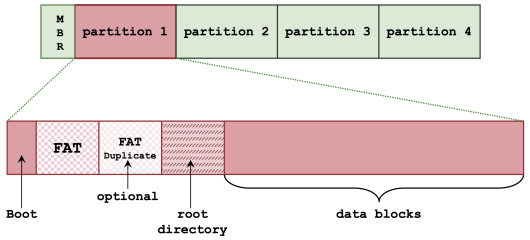
Disk scheduling algorithms

- First come first serve
- Shortest seek first
- SCAN (elevator)
- C-SCAN (outside to inside and repeat)

FILE SYSTEM CASE STUDIES

FAT

Layout



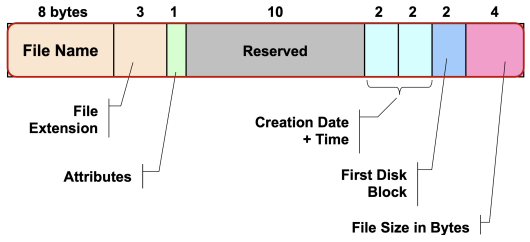
FAT entry

- FREE code (unused block)
- Block number of next block
- EOF code (i.e. NULL pointer)
- BAD block (block is unusable, i.e. disk error)

Directory

- Represented as special type of file
- Root directory is stored in a special location, while other directories are stored in the data blocks
- Each file/subdirectory in the folder is represented as a directory entry

Directory entry



- Fixed size - 32 bits per entry
- Attributes - Read-Only, Directory/File flag, Hidden, etc.
- First byte of file name may have special meaning (Deleted, End of directory entries, Parent directory, etc.)
- Creation time and date - Year limited to 1980 to 2107, accuracy of second is ±2 seconds
- First disk block index - 12, 16, 32 bits for FAT12, FAT16, FAT32 respectively

File deletion

- Set first letter in filename to `0xE5`
- Set FAT entries in linked list to FREE
- Can recover files as actual data blocks are intact

Free space management Must be calculuated by going through FAT

FAT Variants

Disk cluster

Instead of using a single disk block as the smallest allocation unit, use a number of contiguous disk blocks

FAT size

- Bigger FAT ⇒ more disk blocks/clusters ⇒ more bits to represent each disk block/cluster (e.g. FAT12, FAT16, FAT32)

✓ Larger cluster size ⇒ larger usable partition

× Larger cluster size ⇒ larger internal fragmentation

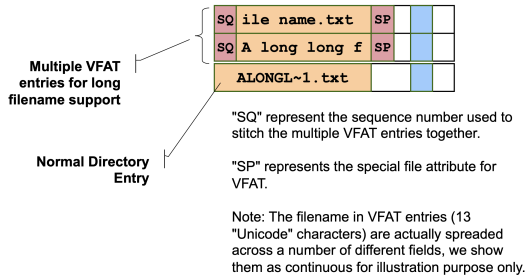
× FAT32: Only 28 out of 32 bits is used in disk block/cluster index (4 bits reserved)

Example (4KiB Cluster):

FAT12	FAT16	FAT32
2^{12} Clusters Largest partition: 4KiB * 2^{12} = 16MiB	2^{16} Clusters Largest partition: 4KiB * 2^{16} = 256MiB	2^{28} Clusters Largest partition: 4KiB * 2^{28} = 1TiB

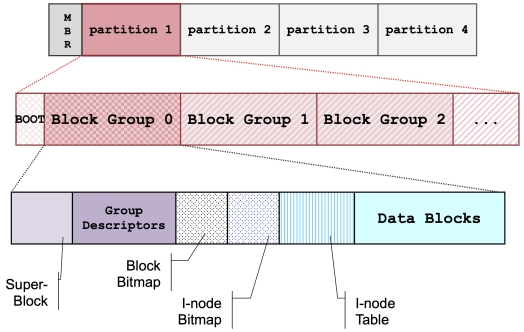
VFAT

- Supports longer filenames (up to 255 chars)
- Workaround, uses multiple directories for a file with long name
- Uses invalid file attribute (so non-VFAT apps can ignore the entries)
- Uses first byte to indicate sequence
  - **"A long long file name.txt"**



Ext2

Layout



Superblock

- Describes entire FS
- Total I-nodes, Total disk blocks, etc.
- Duplicated in each block group for redundancy

Group descriptors

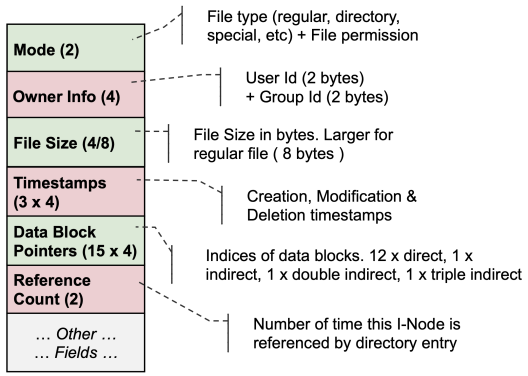
- Describe each block group
- No. of free disk blocks, free I-nodes
- Location of bitmaps
- Duplicated in each block group for redundancy

I-nodes

I-node bitmaps 1 = Occupied, 0 = Free, 1 entry per directory/file

I-node table Array of I-nodes of this block group

I-node structure (also refer to previous page Unix I-node section)



✓ Allows fast access to small files (using first 12 disk blocks)

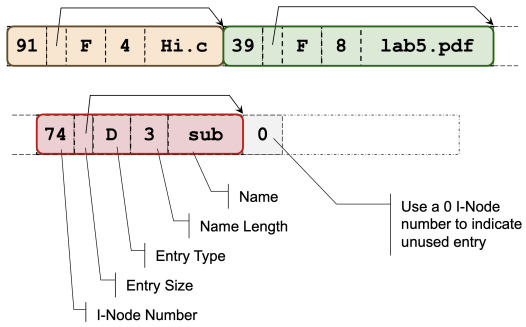
✓ Flexibility in handling huge file (using single/-double/triple indirect blocks)

- Example:
  - Each disk block address is 4 bytes
  - Each disk block is 1KiB
    - So, indirect block can store 1KiB/4 = 256 addresses
  - Maximum File Size:
    = Direct blocks + single indirect + double indirect + triple indirect
    = 12 x 1 KiB + 256 x 1KiB + 256^2 x 1KiB + 256^3 x 1KiB
    = 16843020 KiB (16 GiB)

Directory structure Data blocks of a directory stores a linked list of directory entries

Directory entry

- Store entry size to locate next entry
- File/subdirectory name up to 255 chars



File deletion

- Remove its directory entry from parent directory
  - Point previous entry to next entry/end
  - Blank record for first entry
- Update in I-node and block bitmaps to free

Hard link

- Multiple directory entries point to same I-node
- Maintain a I-node ref count, only delete when count is 0 (i.e. all hard links deleted)

Symbolic link

- Only file pathname is stored
- Involves a search to locate actual I-node number of target file

× Link can be easily invalidated

Misc

FS consistency checks

- Power loss/system crash can render FS in inconsistent state
- Windows: `CHKDSK`, or Linux: `fsck`

Defragmentation

- File data can be scattered across many disjoint blocks on storage media, impacting I/O performance
- Windows: use software to alleviate problem
- Linux
  - Files are allocated further apart
  - Free blocks near to existing data block are used if possible
  - Fragmentation is very low when drive occupancy < 90%

Journaling

- Keep additional info to recover from system crash
- Write info and/or actual data inot a separate log file before performing file op

Other FS

Virtual File System Provides another layer of abstraction on top of existing FS

- Allows app to access file on different FS
- File ops are translated by VFS automatically to the corresponding native FS

Network File System

- Allows files to reside on different physical machines
- File ops are translated into network ops

New Technology File System

- Used in WinXP onwards
- File encryption, file compression
- File versioning
- Hard/Symbolic link

Extended-3/4 FS (Ext3/Ext4)

- Journaling
- In-place upgrade from Ext2
- Expanded max file and file system sizes

Hierarchical File System Plus (HFS+)

- Used in Mac OS X
- Compression, encryption support
- Large FS, file and number of file/folder support
- Metadata journaling

PAST QUESTIONS

1920 S1 Q5 (page table size)

5. [2 marks] The virtual address space supported is 2^{64} **bits** (note bits not bytes). The page size is 1KiB (2^{10} bytes), the size of the physical memory (RAM) is 32KiB, the size of a page table entry is two bytes. Assuming the addressing is at the **byte** level, calculate the size of the page table required for both standard and inverted page tables (the size of the entry in the inverted page table is the same with the entry in the direct page table).
   A. 2^{54} + 32 bytes
   B. 2^{51} + 32 bytes
   C. 2^{54} + 64 bytes
   D. 2^{55} + 64 bytes
   E. 2^{52} + 64 bytes

- Ans: E ($2^{52} + 64$ bytes)
- Assuming 8 bits = 1 byte, then 2^{64} bits = 2^{61} bytes of addressable memory
- 32 KiB / 1 KiB = 32 frames for entire RAM
- 32 × 2 bytes each = 64 bytes for inverted page table
- 2^{10} bytes per page, so $2^{61} \div 2^{10} = 2^{51}$ logical pages required
- $2^{51} \times 2$ bytes = 2^{52} space for standard page table