Memory Management

Memory Hardware

Physical Memory Storage:

- Random Access Memory (RAM)
- Can be treated as an array of bytes (Addressable Units)
- Each byte has a unique index = physical address
- Minimum addressable unit? = A byte!
- 64-bit: have max of 2^{64} addresses to use.
- A contiguous memory region = An interval of consecutive addresses

We can address bits, but they may have same address. Executable typically contains code for text region and data layout (for data region).

Memory Usage Summary

Transient Data:

- Valid only for a limited duration, e.g. during function call (stack)

Persistent Data:

- Valid for the duration of the program unless explicitly removed (if applicable)
- e,g heap, global, code.

Both types of data sections can grow/shrink during execution.

Operating System: Managing Memory

OS handles the following memory related tasks:

- Allocate memory space to new process
- Manage memory space for process
- Protect memory space of process from each other
- Provides memory related system calls to process
- Manage memory space for internal use

Without Memory Abstraction

Pros:

- Memory access is straightforward
- Address in program == Physical Address, great for performance.
- No conversion/mapping is required
- Address fixed during compilation time

Cons:

- If two processes are occupying the same physical memory:
- Conflicts: Both processes assume memory starts at 0!
- Hard to protect mem space.

Memory Abstraction: Logical Addresses

Embedding physical memory address in program is a bad idea. Give birth to the idea of logical addresses!

- Logical address == how the process view its memory space
- Logical address != Physical address in general, mapping is needed.
- Mapping concept is called indirection: provide mapping from physical to logical address.
- Each process has a self-contained, independent logical memory space.
- of ways to organise this mapping schemes, logical abstraction to physical helps OS avoid clashes in programs while allowing compiler to only care about physical and not do alot of work.

Contiguous Memory Management

Process must be in memory during execution! Each process occupies a contiguous memory region. The physical memory is large enough to contain one or more processes with complete memory space.

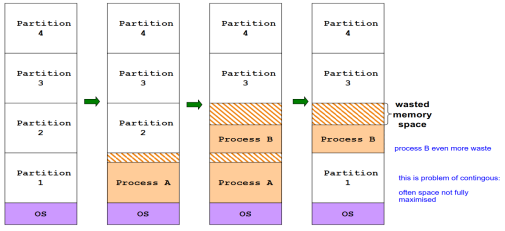
Multitasking, Context Switching, Swapping

Multitasking: Allow multiple processes in the physical memory together,can switch from one process to another. When the physical memory is full, we free up memory by:

- Removing terminated process
- Swapping blocked process to secondary storage

Memory Partition:The contiguous memory region allocated to a single process.

Fixed Partitioning



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

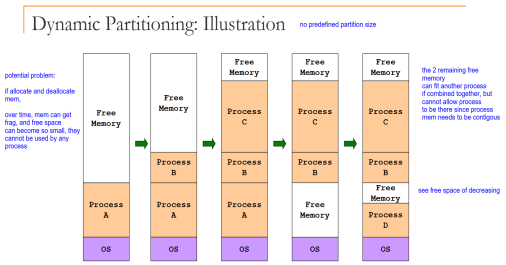
Pros

- Easy to manage
- Fast to allocate since every free partition is the same, no need to choose.

Cons:

- Partition size need be large enough to contain the largest of the processes
- **Internal fragmentation** created when process don't occupy whole partition

Dynamic Partitioning



- Partition is created base on the actual size of process
- OS keep track of the occupied and free memory regions
- Perform splitting and merging when necessary

Free memory space is known as hole. With process creation/termination/swapping, tends to have a large of holes. Known as **External fragmentation**. Merging the holes by moving occupied partitions can create larger hole (more likely to be useful)

Pros:

- Flexible and remove **Internal fragmentation**.
- No waste: process gets exactly memory it asked for.

Cons

- Need to maintain more information in OS.
- Takes more time to locate appropriate region.
- Creates **External fragmentation**.

Allocation for Dynamic

Assuming the OS maintain a list of partitions and holes.

Algorithm to locate partition of size N:

- Search for hole with size M $\rangle$ N.
- Split the hole into N and M-N
- N will be the new partition
- M-N will be the left over space: a new hole

Variants:

- 1. First-Fit: Take the first hole that is large enough
- 2. Best-Fit: Find the smallest hole that is large enough
- 3. Worst-Fit: Find the largest hole

All algorithms take $O(n)$ for allocation and deallocation, and $O(1)$ for merging. To improve: use DS to find your own partition faster: hashtables, so no need search entire list.

When an occupied partition is freed:

- Merge with adjacent hole if possible
- Compaction can also be used:
- Move the occupied partition around to create consolidate holes
- Cannot be invoked too frequently as it is very time consuming

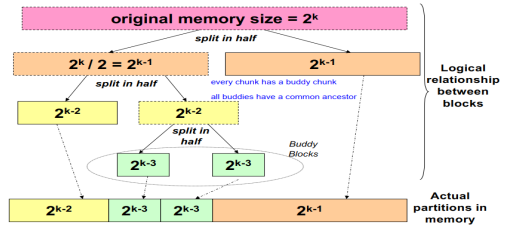
Buddy Allocation

Buddy memory allocation provides efficient:

- 1. Partition splitting.
- 2. Locating good match of free partition (hole).
- 3. Partition de-allocation and coalescing: Only merge Buddies

Main idea:

Free block is split into half repeatedly to meet request. The two halves forms a sibling blocks (buddy blocks). When buddy blocks are both free, can be merged to form larger block.



Buddy Implementation

Keep an array A[0...K], where 2^K is the largest allocatable block size. Each array element A[J] is a linked list which keep tracks of free block(s) of the size 2^J. Each free block is indicated just by the starting address.

Runtime Note: N = num partitions.

- Allocation: $O(lgN)$
- Deallocation: $O(N)$: need to go thru all partitions to find, so at most n.(when all partitions at A[0], yours is at last of A[0])
- Merging $O(lgN)$ could have a case where you create a free buddy that merges with another free buddy and so on and on till top.

Allocation: find smallest block that fits, if do not exist, recursively split bigger blocks until required level has buddy blocks.
Deallocation: Find block and remove, recursively merge until no more to merge.
Merging: 2 blocks B and C are buddy of size 2^S, if the Sth bit of B and C is a complement the leading bits up to Sth bit of B and C are the same.

Disjoint Memory

Process memory space can now be in **disjoint physical memory locations.**

Paging

The **physical memory** is split into regions of fixed size (decided by hardware) known as physical frame. The **logical memory** of a process is similarly split into regions of same size = logical page. At execution time, the pages of a process are loaded into **any available memory frame.**

- Diff parts of process are scattered into diff parts of physical mem, but logically they are contiguous.
- Occupied physical memory region can be disjoined!: Need mapping.

Page Table

Need a general lookup table based approach: logical page maps to physical address:address translation.
Program code uses logical memory address. However, to actually access the value, physical memory address is needed.
To locate a value in physical memory in paging scheme, we need to know:

- **F**, the physical frame
- **Offset**, displacement from the beginning of the physical frame
- Physical Address = F x size of physical frame + Offset

Design Decision Tricks that simplify the address translation calculation:

- 1. Keep frame size (page size) as a power-of-2
- 2. Physical frame size == Logical page size

Given: 1. Page/Frame size of 2n
2.m bits of logical address
Logical Address LA: 1. p = Most significant m-n bits of LA
2. d = Remaining n bits of LA
Use p to find frame f:

- Use page table to find f,
- Physical Address PA: PA = f*2n + d

Paging Evaluation

- Paging removes **External fragmentation** by definition: if there is a free page, anyone can use it, no left-over physical memory region.
- Clear separation of logical and physical address space - Allow great flexibility, Simple address translation.
- Paging can still have **Internal fragmentation**: Logical memory space may not be multiple of page size, but if you need 10 pages, only 10th page will have **Internal fragmentation**. only waste a page per process, so not a very big issue.
- Issues: Require two memory access for every memory reference. 1 to read the page table entry to get frame , 2 to access the actual memory item.
- Qns Assembly code: load r1, [2106] inc r1, store r1, [2106]: with paging, how many ref to memory will the code above have? Answer: 8 or more. Why? For every access (load or store) need to go to table and then do access so now 2*2. but since von neumann archi, need to get inst from memory for load, store, inc: these also need access page table too! For each inst (3 here), need to look at logical addr, then fetch the inst itself, so 2*2*3*3, AND each inst has addr, so may need multiple trips to mem to get inst.

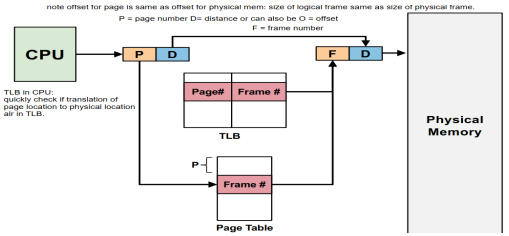
Paging Implementation

OS stores page table information with the process information (e.g. PCB). Page table key part of Memory context of a process.

TLB

Want to solve issue with 2 accesses. Saves the recently accessed page table entries. Modern processors provide specialized on-chip component to support paging: **Translation Look-Aside Buffer (TLB)**: a cache of a few (tens) page table entries. Faster than registers.
Logical address translation with TLB:
Use page to search TLB associatively:

- Entry found (TLB-Hit):Frame is retrieved to generate physical address.
- Entry not found (TLB-Miss):Memory access to access the full page table, Retrieved frame is used to generate physical address and update TLB.



Memory access time Impact: = P(TLB hit)*Latency(hit) + P(TLB miss)*Latency(miss) = 90% x (1ns + 50ns) + 10%(1ns + 50ns + 50ns) = 56ns
TLB hitrate: 99%

TLB, Context Switching

TLB part of hardware context for process!
When a context switch occurs:

- TLB entries are flushed, so that new process will not get incorrect translation.
- OS has special inst to help clear the TLB
- TLB could also be saved and restored later: expensive! Not done often.

Hence, when a process resume running: Will encounter many TLB misses to fill the TLB, performance suffers. It is possible to place some entries initially, e.g. the code pages to reduce TLB misses. Qns: Upon a context switch from proc A $\longrightarrow$ B: which of the following will be saved in the context of a process?

- Process A's data in physical memory: No! we are not saving any content of the data. no data saving in context switch.
- A's logical mem: this is a trick, data is same as in the physical mem, same data, diff name.
- A's page tables: Yes and No, we are only saving the pointer to the start of the page table since page tables are huge!
- A's TLB content: assumes A has some TLB. No! If we switch to B, we may run into addresses that are similar to B's logical addresses but lead to A's mem, and thus be invalid mem addr! Not good to keep A's TLB for B.

Protection

The basic paging scheme can be easily extended to provide memory protection between processes using:

- 1. Access-Right Bits: whether the page is read write, read only, executable.
- The logical memory range is usually the same for all processes. However, not all process utilize the whole range. Some pages are out-of-range for a particular process. Thus we have:
- 2. Valid Bit: Attached to each page table entry to indicate: Whether the page is valid for the process to access. OS will set the valid bits when a process is running, when page given to process. Memory access is checked against this bit.

Everytime access mem will see this. Memory access is checked against the access right bits.

Page Sharing

Page table can allow several processes to share the same physical memory frame. Use the same physical frame in the page table entries.
Possible Usage:

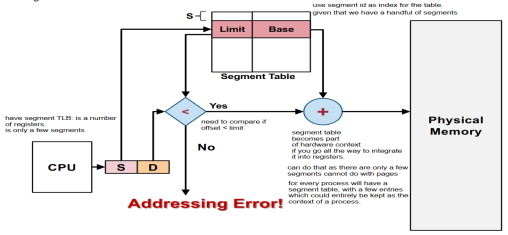
- Shared code pages: same code used by multiple procs.
- Implement Copy-On-Write: parent and child process can share a page until one writes to it.

Segmentation

Segmentation and Paging trying to solve a different problem. Segmentation allows us to have different regions in a contiguous memory space and still allow them grow/shrink freely but still check whether a memory access in a region is in-range or not. **Idea**: Separate the regions into multiple memory segments, each have name, . Logical memory space of a process is now a collection of segments.

Segmentation: Logical Address Translation

Segment name is usually represented as a single . All memory reference is now specified as: Logical address ⟨SegID, Offset⟩ SegID is used to look up ⟨Base, Limit⟩ of the segment in a segment table. Physical Address PA = Base + Offset



Segmentation Summary

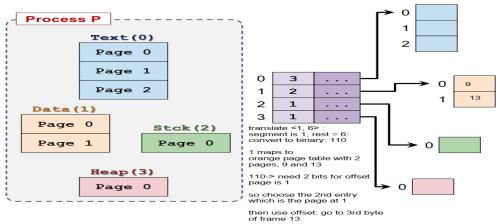
Naturally matches programmer's view of memory. Pros:

- Each segment is an independent contiguous memory space.
- Can grow/shrink, be protected/shared independently.
- **Efficient bookkeeping**: can take a 1TB long segment and describe it with a base and an offset. No need to keep info on many pages. Can integrate into hardware context.
- Protect large regions of memory with 1 bit.

Cons:

- Segmentation requires variable-size contiguous memory regions ⟶ can cause **External fragmentation**.

S + P



Each segment is now composed of several pages instead of a contiguous memory region. Break segment into pieces, can store them in potentially different pieces in memory.

- Essentially, each segment has a page table. Solves the **External fragmentation** problem that we had earlier.
- Segment can grow by allocating new page then add to its page table and similarly for shrinking.

Virtual Memory Management

Split the logical address space into small chunks: Some chunks reside in physical memory (RAM), Other are stored on secondary storage (DISK).

Locality Principles

Most programs exhibit these behaviors, formalized as locality principles:

- Temporal Locality: Memory address which is used is likely to be used again.
- After a page is loaded to physical memory, it is likely to be accessed in near future = Cost of loading page is amortized.
- Spatial Locality: Memory addresses close to a used address is likely to be used.
- A page contains contiguous locations that is likely to be accessed in near future. Later access to nearby locations will not cause page fault.

Most time is spent on a relatively small part of code. In a period of time, accesses are made to a relatively small part of data.

Virtual Page Table Structure

Problems with huge page table: 1. High overhead. 2. Fragmented page table: Page table occupies several memory pages.

Virtual: Direct Paging

Direct Paging: keep all entries in a single table. With 2^p pages in logical memory space, we have:

- p bits to specify one unique page
- 2^p page table entries (PTE), each contains: physical frame , additional information bits.

Real Life Example:

- Page size: 4KB (12 bits for offset)
- VA 64-bit, 16 ExaBytes of virtual address space
- Physical memory 16GB, PA 34 bits
- How many virtual pages? $2^{64}/2^{12} = 2^{52}$
- How many physical pages? $2^{34}/2^{12} = 2^{22}$
- How many bits for physical page ID? 22 bits = 3Bytes In reality, PTE size = 8Bytes (with other flags)

Page table size = $2^{52} * 8B = 2^{55}B$ per process!

Hierarchical Paging

Not all processes uses the full range of virtual memory space thus Full page table is a waste! **Idea**:

- Split the full page table into regions
- Maintain pointers to each region, use NULL pointer to indicate empty region instead of using empty page tables.
- **Only a few regions are used**, As memory usage grows, new region can be allocated.

Description:

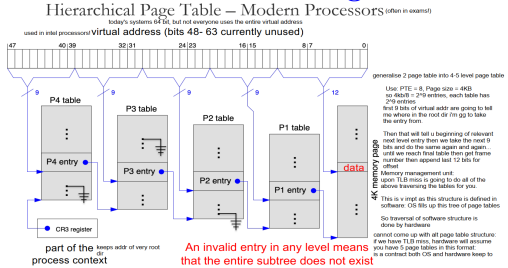
Split page table into smaller page tables, Each with a page table :

- If the original page table has 2^P entries: With 2^M smaller page tables, M bits is needed to uniquely identify one page table. Each smaller page table contains 2^{P-M} entries.
- To keep track of the smaller page tables, A single **page directory** is needed. Page directory contains 2^M indices to locate each of the smaller page tables.

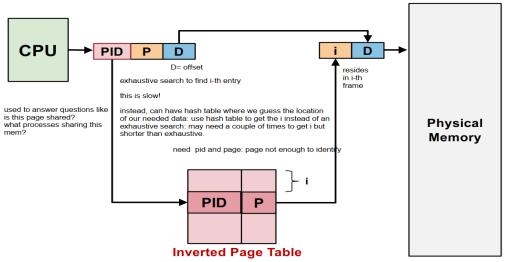
Advantages of Hierarchical Paging

- We can have empty entries in the page directory: The corresponding page tables need not be allocated!
- Assume only 3 page tables are in use. Overhead = 1 page directory + 3 smaller page tables.

We want to make sure size of smaller page tables is same size as page. Why? Cos going beyond the page requires you to store parts with the same page consecutively in memory, which could be a problem. Or it is going to be wasted, if you don't fill it up to the brim will have **Internal fragmentation**.



Inverted Page Table



Page table is a per-process information. With M processes in memory, there are M independent page tables.

- Only N physical memory frames can be occupied
- Out of the M page tables, only N entries is valid
- Huge waste: N valid entries << Overhead of M page tables.

Idea:

- Keep a **single** mapping of physical frame to ⟨pid, page⟩
- page is not unique among processes
- pid + page can uniquely identify a memory page
- Entries are ordered by frame To lookup page X, need to search the whole table

Evaluation: Pros: Huge saving: One table for all processes Cons: Slow translation.

Virtual Page Replacement Algorithms

Suppose there is no free physical memory frame during a page fault: Need to evict (free) a memory page. When a page is evicted:

- Clean page: hasn't been written to at all since the last time it was brought from disk to memory
- Dirty page: modified, need to write back.

Need to consider if it is shared as well! Evicting this page would cause a page fault in many processes. Note for replacement: we do not check what part of the page is accessed: we only care WHICH page is accessed only will see page 8 multiple times, not bit1 of page 8 and so on

OPT

Replace the page that will not be used again for the longest period of time. **Guarantees minimum of page faults**. Not realistic, need future knowledge of mem refs.

FIFO

Memory pages are evicted based on their loading time: Evict the oldest memory page. Simple to implement:

- OS maintain a queue of resident page s
- Remove the first page in queue if replacement is needed
- Update the queue during page fault trap
- No hardware support needed.

Cons:

- If physical frame increases (e.g. more RAM) the page fault increases. (Belady's Anomaly).
- Does not exploit temporal locality!

Least Recently Used

General Idea:

- Make use of **temporal locality**, Replace the page that has not been used in the longest time.
- Expect a page to be reused in a short time window.
- Have not used for some time: most likely will not be used again.

Notes: Attempts to approximate the OPT algorithm. **Gives good results generally**.

LRU Implementation

Implementing LRU is not easy: 1. Need to keep track of the "last access time" somehow. 2. Need substantial hardware support.

Approach A - Use a Counter:

- Use logical "time" counters, which is incremented for every memory reference.
- Page table entry has a "time-of-use" field.
- Store the time counter value whenever reference occurs
- Replace the page with smallest "time-of-use"

Problems: 1. Need to search through all pages. 2. "Time-of-use" is forever increasing (overflow possible!)

Approach B - Use a "Stack":

- Maintain a stack of page s. If page X is referenced
- Remove from the stack (for existing entry)
- Push on top of stack
- Replace the page at the bottom of stack
- No need to search through all entries

Problems: 1. Not a pure stack: Entries can be removed from any where in the stack. 2. Hard to implement in hardware.

Second-Chance (CLOCK)

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 and page taken as newly loaded. Next FIFO page is selected, go to Step 2.
- Degenerate into FIFO algorithm when all pages has ref bit == 1.

Virtual Mem: Frame Allocation

What is the best way to distribute the N frames among M processes?

Simple Approaches:

- Equal Allocation: Each process get N / M frames
- Proportional Allocation: Processes are different in size (memory usage) Let $size_p$ = size of process p, $size_{total}$ = total size of all processes. Each process get $size_p/size_{total} * N$ frames.

Local Replacement: Victim page are selected among pages of the process that causes page fault.

- Pros: Frames allocated to a process remain constant: Performance is stable between multiple runs.
- Cons: If frame allocated is not enough, might page fault very frequently. If other processes has spare pages, our proc cannot make use of it.

Global Replacement: Victim page can be chosen among all physical frames: Process P can take a frame from Process Q by evicting Q's frame during replacement!

- Pros: Allow self-adjustment between processes: Process that needs more frames can get from other.
- Cons: 1. Badly behave process can affect others. 2. Frames allocated to a process can be different from run to run

Thrashing

Insufficient physical frames for processes decreases throughput of system, as processes continually page fault, and heavy I/O to bring non-resident pages into RAM. This problem is known as thrashing.

Hard to find the right of frames:

- If Global replacement is used: A thrashing process "steals" page from other process $\longrightarrow$ cause other process to thrash (Cascading Thrashing)
- If Local replacement is used: Thrashing can be limited to one process. But that single process can hog the I/O and degrade the performance of other processes.

Working Set Model

Observation:

- The set of pages referenced by a process is relatively constant in a period of time (locality). However, as time passes, the set of pages can change.
- E.g After 1 function terminates, the references will change to another set of pages: may not have same variables and code.

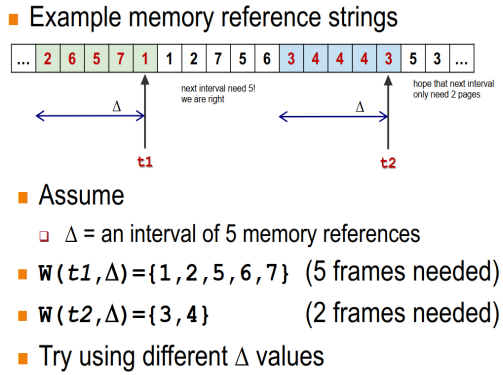
Using these observations on locality:

- With the same set of pages in frame: No/few page fault until process transits to new locality.
- In a new locality: A process will cause page fault for the set of pages

Working Set Model:

- Define Working Set Window δ: An interval of time
- $W(t, \delta)$ = active pages in the interval at time t.
- **Goal**: Allocate enough frames for pages in $W(t, \delta)$ to reduce possibility of page fault.

Transient region:working set changing in size
stable region: working set about the same for a long time.



Accuracy of working set model is directly affected by the choice of D.
Too small: May miss pages in the current locality Too big: May contain pages from different locality

File Management

File System provides:

- An abstraction on top of the physical media, such that we can abstract away physical characteristics, and just read and write and observe the latency.
- A high level resource management scheme
- Protection between processes and users
- Sharing between processes and users

General Criteria:

- Self-Contained: Should be able to "plug-and-play" on another system.
- Persistent: even when power not on, Beyond the lifetime of OS and processes.
- Efficient: Provides good management of free and used space. Minimum overhead for bookkeeping information

Key difficulty of mem management vs file management:

- We are accessing memory at every instruction: to get the instruction then store the result so mem is accessed ALOT multiple times per instruction, so needs to be hyper fast accessed usually without OS involvement.
- File system operations are not as latency critical as mem management operations
- when we are talking to mem, we don't involve OS unless there is some catastrophe, some page fault.
- but when we are talking to FS we are using sys calls, explicit commands to OS to deal with the FS.

File Systems Abstractions

File System:

- Consists of a collection of files and directory structures
- File: An abstract storage of data
- Directory (Folder): Organization of files

File

Represent a logical unit of information created by process. Essentially an ADT have a set of common operations with various possible implementation. Contains:

- Data: Information structured in some ways
- Metadata(attributes): Additional information associated with the file (Name, Identifier, Type, Size, Protection, Time, Date and owner, how to access file)

File Type

Different FS has different naming rule: **Usual**: Name.Extension Each file type has:
1. An associated set of operations.
2. Possibly a specific program for processing.

Common file types:

- Regular files: contains user information. 2 types: 1. ASCII can be read/displayed as/is. e.g text, codes. 2. Binary files: Have a predefined internal structure that can be processed by specific program, e.g executables, pdf, mp3/4, images.
- Directories: system files for FS structure
- Special files: character/block oriented

Distinguishing File Type:

- Use file extension: e.g Windows
- Use embedded information in the file: Unix, stored at the beginning of the file. Commonly known as magic number.

File Protection

Controlled access to the information stored in a file. Types of Access:

- Read/Write/Execute
- Append(add new info to EOF)/Delete/List(Read metadata of a file)

Protection, How? Most common: Restrict access base on the user identity.
Access Control List: A list of user identity and the allowed access types.

- In UNIX, Minimal ACL (the same as the permission bits)
- or Extended ACL (added named users / group)

f Pros: Very customizable. Cons Additional information associated with file.

Permission Bits

- Classified the users into three classes: 1. Owner: The user who created the file. 2. Group: A set of users who need similar access to a file. 3. Universe: All other users in the system. Bits appear in that order.
- Example (Unix) Define permission of three access types (Read/Write/Execute) for the 3 classes of users. `ls {l` to see the permission bits for a file

File Data

Operations on File Metadata.

- Rename, Change attributes: file access, dates, ownership.
- Read attributes.

Structure:

- Array of bytes(UNIX): No interpretation of data: just raw bytes. Each byte has a unique offset (distance) from the file start.
- Fixed length records: Array of records, can grow/shrink. Can jump to any record easily: Offset of the Nth record = size of Record * (N-1).
- Variable length records: Flexible but harder to locate a record

Access Methods

- Sequential Access: Data read in order from start. Cannot skip but can be rewound (tape).
- Random Access: Data can be read in any order (disk can move to the position we want, go directly to the point we want) Use 1. Read(offset): explicitly state the position to be accessed. 2. Seek(Offset): A special operation is provided to move to a new location in file.
- Direct Access: Used for file contains fixed-length records. Allow random access to any record directly. Very useful where there is a large amount of records. e.g. In database. Random access = special case.

File Operations

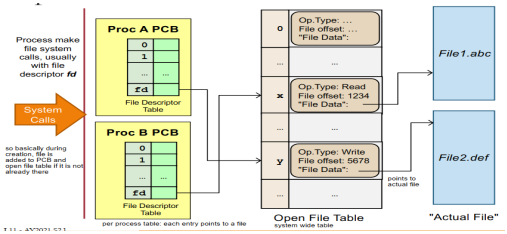
OS provides file operations as system calls:

- Provide protection, concurrent and efficient access.
- Maintain information.

E.g Create, Open, Read, Write, Repositioning: AKA seek Move the current position to a new location No actual Read/Write is performed, Truncate: Removes data between specified position to end of file.

Information kept for an opened file:

- File Pointer: Current location in file
- Disk Location: Actual file location on disk
- Open Count: How many process has this file opened? How many processes are using this file? Useful to determine when to remove the entry in table.



What is a good way to organize the open-file information? Common approach is to have 2 diff tables:

- 1. System-wide open-file table: One entry per unique file.
- 2. Per-process open-file table: keeps track of all files that are open in that particular process: entry in this table points to prev table. One entry per file used in the process. Each entry points to the system-wide table.

Process Sharing Case 1:

- Two processes using different file descriptors: I/O can occur at independent offsets.
- Example: Two process open the same file (fork then open). Same process open the file twice (diff offset, so diff part of same file).

Process Sharing Case 2:

- Two processes using the same file descriptor. Only one offset: I/O changes the offset for the other process
- Example: fork() after file is opened

Directory

Directory (folder) is used to:

- 1. Provide a logical grouping of files: The user view of directory
- 2. Keep track of files: The actual system usage of directory

Ways to structure directory: Single-Level, Tree-Structure, DAG, Graph

Tree-Structure

- Directories can be recursively embedded in other directories: Naturally forms a tree structure
- Two ways to refers to a file: Absolute Pathname: Path from root directory to the file.
- Relative Pathname: Directory names followed from the current working directory (CWD). CWD can be set explicitly or implicitly changed by moving into a new directory under shell prompt

DAG If a file can be shared:

- Only one copy of actual content "Appears" in multiple directories With different path names, then tree structure becomes DAG

2 Implementations:

1. Unix Hard Link

A and B has separate pointers that point to the actual file F in disk, they share the file. Pros: Low overhead, only pointers are added in directory. Cons: Deletion problems: e.g. If B deletes F? If A deletes F?
Command: `ln`

2. Unix Symbolic Link

B creates a special link file, G, contains the path name of F. When G is accessed: Find out where is F, then access F.
Pros: Simple deletion: If B deletes: G deleted, not F. If A deletes: F is gone, G remains (but not working). Cons: Larger overhead: Special link file take up actual disk space. Command: `ln -s`.

General Graph: Have loop (potentially symbolic link)! Not desirable. Hard to traverse, must prevent infinite loops.

File System Implementations

Disk Organisation

- Master Boot Record (MBR) at sector 0 with partition table. Followed by one or more partitions. Each partition can contains an independent file system.
- A file system generally contains: OS Boot-Up information Partition details: Total Number of blocks, Number and location of free disk blocks, Directory Structure, Files Information, Actual File Data.

File Info and Data Implementation

Basically focuses on how to allocate file data on disk.

- Logical view of a file: A collection of logical blocks
- When file size != multiple of logical blocks, Last block may contain wasted space, i.e. **Internal fragmentation**.
- A good file implementation must: Keep track of the logical blocks, Allow efficient access, Disk space is utilized effectively. Prevent creating huge data structures bigger than data to look up files.

System 1: Contiguous

Allocate consecutive disk blocks to a file.

- Pros: Simple to keep track: Each file only needs: Starting block number + Length. Fast access (only need to seek to first block).
- Cons: **External fragmentation**. Think of each file as a variable-size "partition". Over time, with file creation/deletion, disk can have many small "holes". File size need to be specified in advance: hard for it to grow!(limit file size)

System 2: Linked List

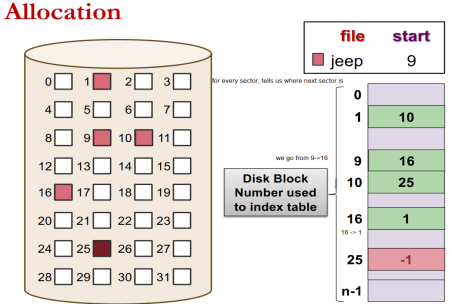
Keep a linked list of disk blocks. Each disk block stores: 1. The next disk block number (i.e. act as pointer). 2. Actual file data. File information stores: First and last disk block number.

- Pros: No **External fragmentation**.
- Cons: Random access in a file is very slow. Part of disk block is used for pointer. Less reliable (case: wrong pointer).

System 3: Linked List 2

Move all the block pointers into a single table known as File Allocation Table (FAT) (contains ONLY pointers, put pointers for all sectors in the system into FAT). FAT is in memory at all time. Simple yet efficient.

FAT Allocation



- Pros: Faster Random Access. The linked list traversal now takes place in memory. Also does not use 1 pointer of space to each block : FAT - potential saving space.
- Cons: Random access in a file is very slow. Part of disk block is used for pointer. Less reliable (case: wrong pointer).

System 4: Indexed Allocation

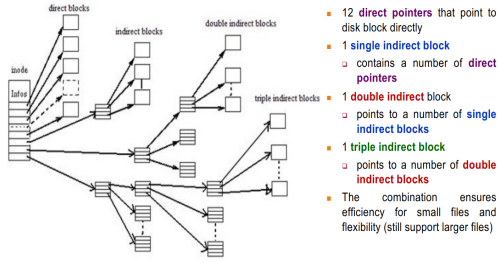
Each file has an index block (per file table). An array of disk block addresses. IndexBlock[N] == Nth Block address. Equivalent to having small direct page table as opposed to FAT which is like an inverted page table.

- Pros: Lesser memory overhead: Only index block (per file table) of opened file needs to be in memory. Fast direct access. Also solve problem of losing rest of list when 1 pointer is lost: we still know the order of blocks even if one in the middle is corrupted.
- Cons: Limited maximum file size. Max number of blocks == Number of index block entries. Index block overhead.

Variations to Indexed Allocations:

- Linked scheme:Keep a linked list of index blocks. Each index block contains the pointer to next index block.
- Multilevel index: Similar idea as multi-level paging. First level index block points to a number of second level index blocks. Each second level index blocks point to actual disk block. Can be generalized to any number of levels.
- Combined scheme: Combination of direct indexing and multi-level index scheme.
- Example:Inode below.

I-Node Unrolled



Free Space Management

To perform file allocation: Need to know which disk block is free. i.e. maintain a free space list. Free space management:

- Maintain free space information.
- **Allocate:** Remove free disk block from free space list. Done when file is created or enlarged (appended)
- **Free**: Add free disk block to free space list. Done when file is deleted or truncated.

Free Space: Bitmap

Each disk block is represented by 1 bit q E.g. 1 == free, 0 == occupied.

- Pros: Provide a good set of manipulations: E.g. can find the first free block, n-consecutive free blocks easily by bit level operation.
- Cons: Need to keep in memory for efficiency reason, can have significant space overhead.

Free Space: Linked List

Use a linked list of disk blocks: Each disk block contains: 1. A number of free disk block numbers. 2. A pointer to the next free space disk block.

- Pros: Easy to locate free block. Only the first pointer is needed in memory, though other blocks can be cached for efficiency.
- Cons: High overhead, can be mitigated by storing the free block list in free blocks! Each free block points to another free block! since they are free blocks, we do not care about their content so no overhead since 'stealing' space from free blocks once we want to use it, will be available to us.

Directory Implementation

The main tasks of a directory structure:

- 1. Keep tracks of the files in a directory. Possibly with the file metadata.
- 2. Map the file name to the file information

Remember:

- File must be opened before use. Something like open("data.txt");
- The purpose of the open operation: Locate the file information using pathname + file name. Pathname = path from root.
- Given pathname: Need to recursively search the directories along the path to arrive at the file information.
- Sub-directory is usually stored as file entry with special type in a directory.

Directory: Linked List

Directory consists of a list: Each entry represents a file. We store:

- File name (minimum) and possibly other metadata.
- File information or pointer to file information (start block and the length of file).

Locate a file using list:

- Requires a linear search: Inefficient for large directories and/or deep tree traversal.
- Common solution: Use cache to remember the latest few searches. User usually move up/down a path.

Directory: Hash Table

Each directory contains a: Hash table of size N. To locate a file by file name:

- File name is hashed into index K from 0 to N-1
- HashTable[K] is inspected to match file name. Usually chained collision resolution is used. i.e., file names with same hash value is chained together to form a linked list with list head at HashTable[K].

Evaluation:

- Pros: Fast lookup, no linear search.
- Cons: Hash table has limited size (need to provide in advance). Depends on good hash function.

Can also combine: small directories use linked-list, big use hash table.

Directory: File Information

File information consists of: 1. File name and other metadata, 2. Disk blocks information. Two common approaches:

- 1. Store everything in directory entry. A simple scheme is to have a fixed size entry. All files have the same amount of space for information.
- 2. Store only file name and points to some data structure for other info.

If other file info is bulky and big, then better to store elsewhere: is **good to keep directory small**.

File System Together

At runtime, when user interacts with file:

- Run-time information is needed
- Maintained by OS in memory

[Recap] Common in-memory information:

- System-wide open-file table: Contain a copy of file information for each open file + other info.
- Per-process open-file table: Contains pointer to system-wide table + other info.
- Buffers for disk blocks read from/written to disk.

File Operation: Create

Want to create: ".../**parent**/F".

- 1. We start from root, then look for directories and see if have access then keep looking into directories until get to **parent**
- 2. Make sure no duplicates of F in parent.
- 2.5. Quite alot of operations to find parent huge number of disk operations to create new file: what helps: directory struct being cached in mem, which is easier to cache if directory is small: this is why we want to sep file data and info from directory struct.
- 3. Use free space list to find free disk block(s).
- 4. Add an entry to parent directory: With relevant file information. File name, disk block information etc

Disadv is that need to look in many places for info on file if sep file data and info from directory struct.

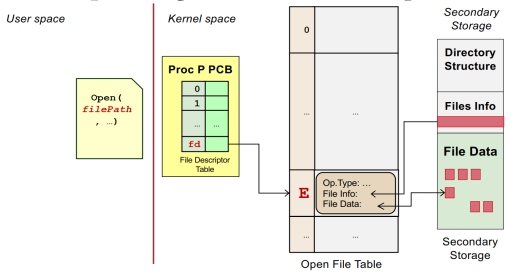
File Operation: Open

Want: Process P open file "/.../.../.../F":

- 1. Search system-wide table for existing entry E. If found: Creates an entry in P's table to point to E. Return a pointer to this entry. Else: continue to next step.
- 2. Use full pathname to locate file F. If not found, open operation terminates with error. When F is located, its file information is loaded into a new entry E in system-wide table. Creates an entry in P's table to point to E
- 3. Return a pointer to this entry.

The returned pointer is used for further read/write operation.

File Open: Improved Understanding



Disk IO Scheduling

Simpler Approaches

- FCFS: do as requests arrive, wasteful.
- SSF (Shortest Seek First) "SJF" modified for the disk context Greedy!
- SCAN (Elevator) Bi-Direction, moves from innermost to outer and vice versa.
- C-SCAN: Only from Outermost to innermost.
- For both scans sort requests by how near they are to current: E.g for [13, 14, 2, 10, 17, 21, 7]: assume we are at 13: 14 is close, 2 is not on the way, 10 is already past, so go 17 next then next 21 then no one else after 21: no need to go all the way to end of tracks. Now turn and go back: pickup locations missed previously: 10, 7, 2, then go back up again.

Newer Approaches

- Deadline - 3 queues for I/O requests(optimisation to add better response time, make sure individual requests are served better not just overall time of all requests): 1. Sorted. 2. Read FIFO - read requests stored chronologically. 3. Write FIFO - write requests stored chronologically.
- noop (No-operation) - Deadline but no Sorted queue.
- cfq (Completely Fair Queueing) - time slice and perprocess sorted queues. Like lottery scheduling decide how long each process gets to use disk.
- bfq (Budget Fair Queuing) (Multi-queue) - fair sharing based on the number of sectors requested. Dont fair-share time, share based on amount of data written/read want to make sure everyone gets to read same number of blocks.

Deadline method: Starvation solved by having 2 queues for read and write: We then have a deadline to serve every request and read and write can have diff deadlines but we try to serve requests by sorted order but periodically check if 2 other queues to see if deadline coming soon, if yes then go serve that request immediately, else default to optimal ordering(sorted). We always have oldest req at head of read and write queues.