7. Memory Abstraction

Physical memory storage:

- Random Access Memory (RAM)
- Can be treated as an array of bytes
- Each byte has a unique index (known as physical address)

Registers

- Closest to ALU
- Use same technology as ALU
- Instruction format (add R1, R2 → R3)
  - Reading two registers, write one register in single instruction
- The register file is multiported
- Registers identified by simple number
- Limits to how many registers you can have

Main Memory

- Off-chip and uses a different technology
  - Goal is density and cost
- Complex address translation needed
- Transactional model
  - Relatively slow: full access to memory cost hundreds of instruction cycles

Memory Usage

- **Transient Data**
  - Valid only for a limited duration. E.g. duration function calls
  - E.g. paramaters, local variables
- **Persistent Data**
  - Valid for the duration of the program unless explicitly removed (if applicable)
  - e.g. global variable, constant variable, dynamically allocated memory
- Both types of data sections can grow/shrink during execution

OS Role: Managing Memory

- Allocate memory space for new processes
- Manage memory space for processes
- Protect memory space of processes from each other
- Provide memory related syscalls to processes
- Manage memory space for internal use **Memory Abstraction**

Without Memory Abstraction

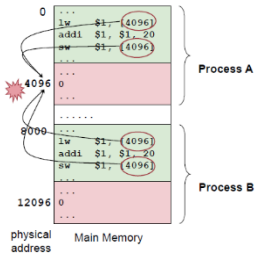
Pros: Memory access is straight forward
Address in program == Physical Address
No conversion/mapping is required
Address fixed during compilation time

Cons: Hard to protect memory space
If two processes occupy the same physical memory
Conflicts: both processes assume memory starts at 0

Fix Attempt: Address Relocation

Recalculate memory references when the process is loaded into memory:

- E.g. add an offset of 8000 to all memory references in Process B (because Process B is loaded at address 8000)



Problems:

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

Fix Attempt 2: Base + Limit Registers

Use a special register as the base of all memory references

- Known as base register
- During compilation time, all memory references are compiled as an offset from this register
- At loading time, the base register is initialized to the starting address of the process memory space

Add another special register to indicate the range of the memory space of the current process

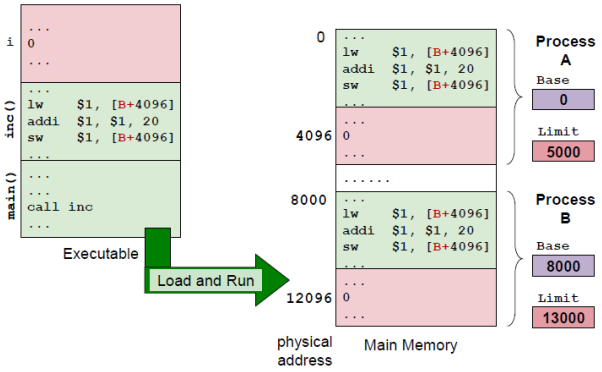
- Known as limit register
- All memory access is checked against the limit to protect memory space integrity

Problems:

- To access adr: Actual = Base + Adr
- Check Actual < Limit for validty
- Every memory access incurs an addition and comparison

Later generalized to segmentation mechanism. It provides a crude memory abstraction: address 4096 in Process A and B no longer the same physical location.

Base and Limit Register Illustration



Memory Abstraction: Logical Address

Embedding physical memory address in program is a bad idea. Gives birth to the idea of logical address:
Logical address == how the process views its memory space

- Logical address != Physical address in general
  - Instead a mapping between logical address and physical address is needed
- Each process has a self-contained, independent logical memory space

Contiguous Memory Management

Process must be in memory during execution

- Stored-program computer
- Load-store memory execution model

Assumptions:

- Each process occupies a contiguous memory region
- Physical memory is large enough to contain one or more processes with complete memory space

To support multitasking

- Allow multiple processes in the physical memory at the same time
- So that we can switch from one process to another

When the physical memory is full:

- Free up memory by:
  - Removing terminated process
  - Swapping blocked process to secondary storage

Memory Partitioning

The contiguous memory region allocated to a single process

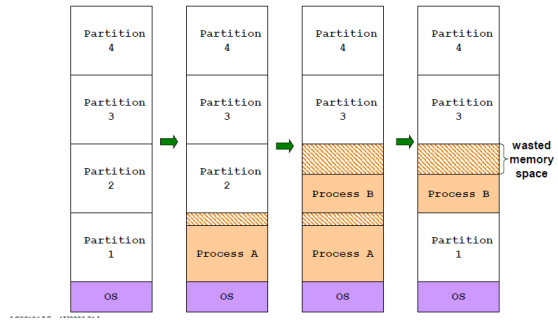
Fixed-Size Partition

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

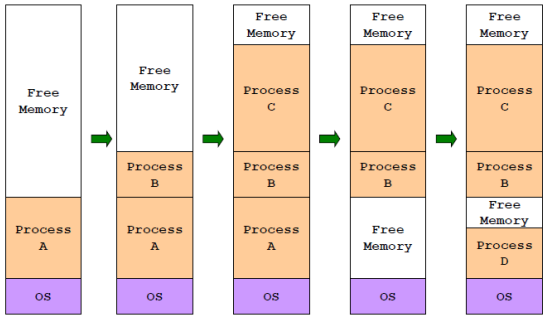
Variable-Size Partition

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

Fixed Partitioning: Illustration



Dynamic Partitioning: Illustration



Fixed Partitioning: Summary

If a process does not occupy the whole partition: any leftover space is wasted – known as internal fragmentation

Pros: Easy to manage
Fast to allocate (every free partition is the same → no need to choose)

Cons: Partition size need be large enough to contain the largest of the processes
Smaller process will waste memory space → **internal fragmentation**

Dynamic Partitioning: Summary

Free memory space is known as hole. With process creation / termination / swapping:

- tend to have a large number 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

Cons: Need to maintain more information in OS
Takes more time to locate appropriate region

Dynamic Partitioning: Allocation Algorithms

Assuming the OS maintains a list of partitions and holes

Algorithm to locate partition of size N:
Search for a hole with size M > N. Several variants:

- **First-Fit:** Take the first hole that is large enough
- **Best-Fit:** Find the smallest hole that is large enough
- **Worst-Fit:** Find the largest hole

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

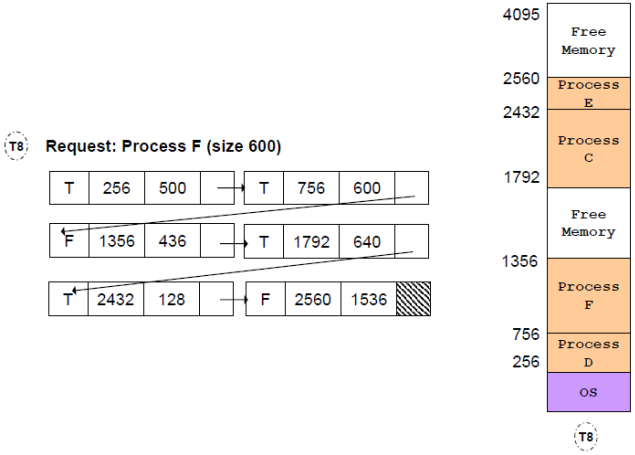
Dynamic Partitioning: Merging and Compaction

When an occupied partition is freed: merge with adjacent hole is possible.

Compaction can also be used to move the occupied partition around to create consolidated holes. But cannot be invoked too frequently as it is very time consuming.

Dynamic Partitioning Illustration

OS maintains a linked list for partition info.

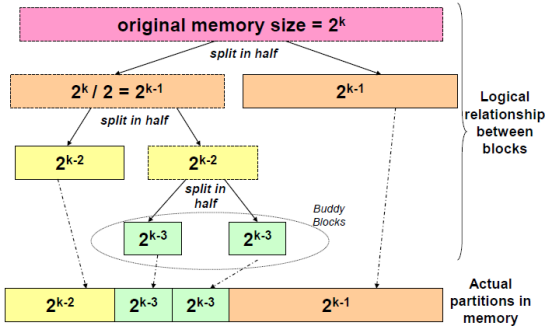


Dynamic Allocation Algorithm: Buddy System

Buddy memory allocation provides efficient:

- partition splitting
- locating good match of free partition (hole)
- partition deallocation and coalescing.

Buddy Blocks Illustration



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

Buddy System Implementation

Keep an array A[0...K], when 2^K is the largest block size that can be allocated

- 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

In actual implementation, there may be a smallest block size that can be allocated as well. A block that is too small is not cost effective to manage

Buddy System: Allocation Algorithm

Allocate a block of size N:

1. Find the smallest `S`, such that 2^S `>= N`
2. Access `A[S]` for a free block
   a. If free block exists:
      - Remove the block from free block list
      - Allocate the block
   b. Else
      - Find the smallest `R` from `S+1 to K`, such that `A[R]` has a free block `B`
      - For (`R-1 to S`)
        - Repeatedly split `B` → `A[S...R-1]` has a new free block
      - Goto Step 2

Buddy System: Deallocation Algorithm

To free a block B:

1. Check in `A[S]` , where 2^S `== size of B`
2. If the buddy `C` of `B` exists (also free)
   a. Remove `B` and C from list
   b. Merge `B` and `C` to get a larger block `B'`
   c. Goto step `1`, where `B` ← `B'`
3. Else (buddy of `B` is not free yet)
   - Insert `B` to the list in `A[S]`

Observe that:

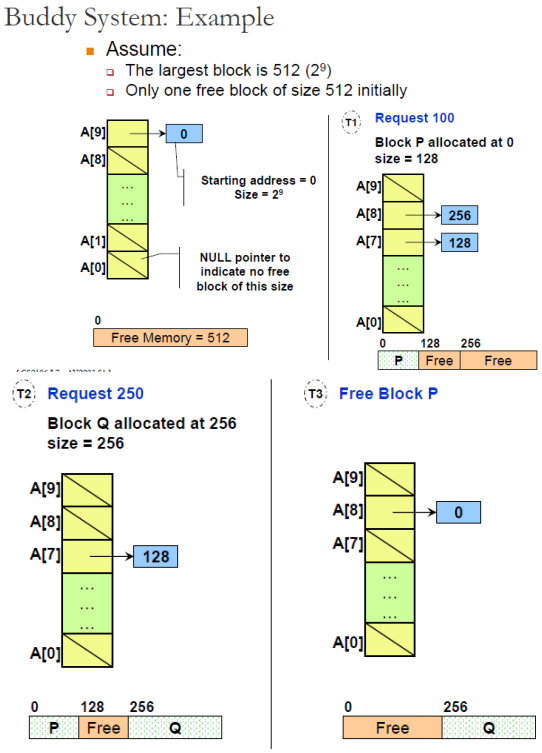
- Given block address `A` is `xxxx00..00`$_2$
- Get 2 blocks of half the size after splitting:
  `B` = `xxxx0..00`$_2$ and `C` = `xxxx1..00`$_2$

Example:

- `A = 0 (000000`$_2$`), size = 32`
- After splitting:
  - `B = 0 (00`**`0000`**$_2$`), size = 16`
  - `C = 16 (01`**`0000`**$_2$`), size = 16`

So, two blocks `B` and `C` are buddy of size 2^`S`, if

- The `S`th bit of `B` and `C` is a complement
- The leading bits up to `S`th bit of `B` and `C` are the same



8. Disjoint Memory Schemes

Physical memory is split into regions of fixed size (decided by hardware)
- Known as **physical frames**

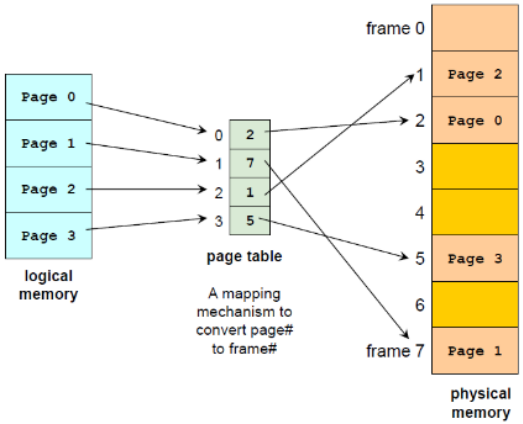
Logical memory of a process is split into regions of the **same size**
- Known as **logical pages**

At execution time, pages of a process are loaded into **any available** memory frame
- Logical memory space remain contiguous
- Occupied physical memory region can be disjoined

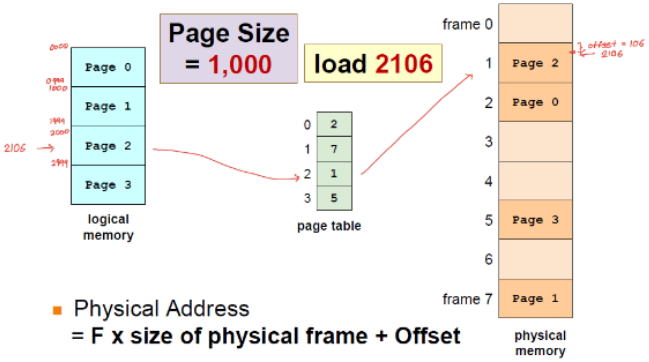
In contiguous memory allocation, it is very simple to keep track of the usage of a process
- Minimally, starting address and size of process

Under paging scheme:
- Logical page ↔ physical frame mapping is no longer straightforward
- Need a lookup table to provide the translation
- Known as the **Page table**

Paging Illustration



Logical Address Translation



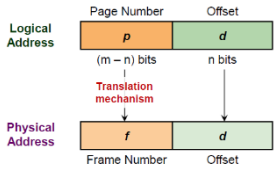
Program code uses logical memory address. However, to actually access the value, physical memory address is needed.

To locate a value in physical memory in the paging scheme, we need to know:
- F, the physical frame number
- Offset, displacement from the beginning of the physical frame

Physical Address = F x size of physical frame + Offset
Two important design decisions:
- Keep frame size (page size) as a power of 2

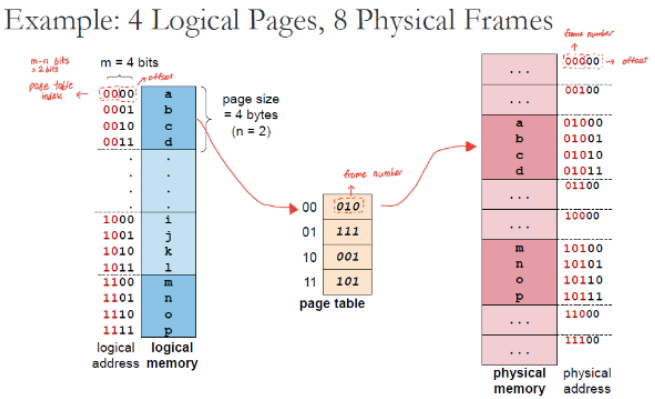
Physical frame size == Logical page size

Logical Address Translation Formula
Given - **page/frame size of 2^n**
- **m bits of logical address**



Logical Address LA:
p = Most significant m-n bits of LA
d = Remaining n bits of LA

Use **p** to find **frame number f** from mapping mechanism like page table.
Physical Address PA = (f x 2^n) + d



Paging: Observations

Paging removes **external fragmentation**
- No leftover physical memory region
- All free frames can be used with no wastage

Paging can still have **internal fragmentation**
- Logical memory space may not be multiple of page size

Clear separation of logical and physical address space
- Allow great flexibility
- Simple address translation

Paging Scheme: Software

Common pure-software solution:
- OS stores page table alongside process information (e.g. PCB)
- Improved understanding: Memory context of a process == Page Table

Issues:
- Require two memory accesses for every memory reference
  - 1st access to read the indexed page table entry to get frame number
  - 2nd access to access the actual memory item

Paging Scheme: Hardware

Modern processors provide specialized on chip component to support paging
- Known as Translation Look Aside Buffer **TLB**
- TLB acts as a cache for a few page table entries

Logical address translation with TLB:
- Use page number to search TLB associatively
- Entry found **(TLB-hit):**
  - Frame number is retrieved to generate physical address
- Entry not found **(TLB-Miss):**
  - Access full page table

Retrieved frame number is used to generate physical address and update TLB

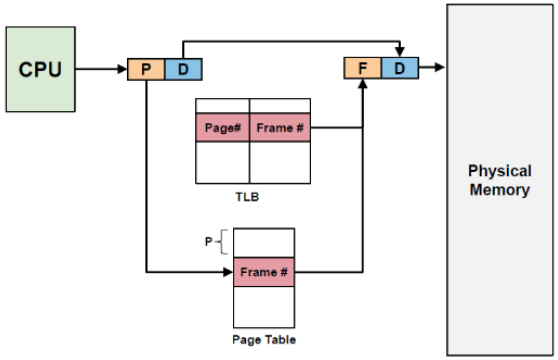
TLB: Impact on Memory Access Time

Suppose: TLB access takes 1ns, Main memory access takes 50ns
What is the average memory access time if TLB contains 40% of the whole page table?

Memory access time = TLB hit + TLB miss
= 40% x (1ns + 50ns) + 60%(1ns + 50ns + 50ns) = 81ns
Overhead of filling in TLB entry and impact of cache ignored
TLB Hit: TLB Access + Access memory at page frame
TLB Miss: TLB Access + Access page table at memory + Access value at memory

TLB and Process Switching

Translation Look-Aside Buffer: Illustration



Improved understanding: TLB is part of the hardware context of a process.

When a context switch occurs, TLB entries are flushed so that new process will not get incorrect translation.

When a process resumes running: it will encounter many TLB misses to fill the TLB. It is possible to place some entries initially (e.g. the code pages, to reduce TLB misses).

Paging Scheme: Protection

The basic paging scheme can be easily extended to provide memory protection between processes using: **Access-Right Bits** & **Valid Bit**.

Access Right Bits:
Each PTE includes several bits to indicate:
- Whether the page is writable, readable, executable
- E.g. page containing code should be executable, page containing data should be readable and writable, etc.

Memory access is checked against the access righit bits.

Observation:
- The logical memory range is usually the same for all processes
- However, not all processes utilize the whole range
    - Some pages are out of range for a particular process

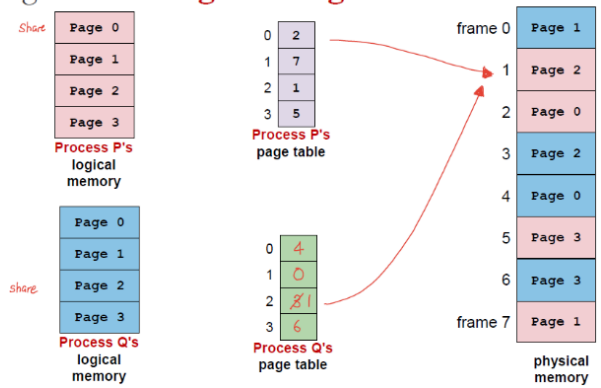
Valid bit:
Attached to each page table entry to indicate:
- Whether the page is valid for the process to access
- OS will set the valid bits when a process is running
- Memory access is checked against this bit: out of range access will be caught by the OS

Paging Scheme: Page sharing

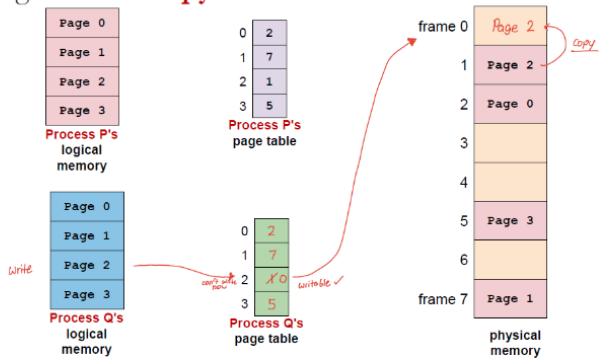
Page table can allow several processes to share the same physical memory frame
- Use the same physical frame number in the page table entries

Shared code page: Some code are being used by many processes, e.g., C standard library, system calls, etc.
Implement Copy-On-Write: Parent and child process can share a page until one tries to change a value in it

Paging Scheme: **Page sharing**



Paging Scheme: **Copy-On-Write**



Segmentation Scheme: Motivation

Memory space of a process is treated as a single entity so far. However, there are several memory regions with different usage in a process.

For example, C program:
- User Code Region
- Global Variables Region
    - Static Data (persistent as long a program executes)
- Heap Region:
    - Dynamic data (persistent as long as user doesn't free)
- Stack Region
- Library Code Region
- etc.

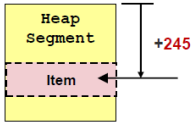
Some regions may grow/shrink at execution time: E.g., the stack region, the heap region, the library code region. It is hard to place different regions in a contiguous memory space and still allow them grow/shrink freely. Also hard to check whether a memory access in a region is within its bound.

Segmentation Scheme: Basic Idea

Separate regions into multiple **memory segments**. Logical memory space of a process is now a collection of segments.

Each memory segment has a **name** (for ease of reference), and a **limit** (to indicate the range of segment).

Memory Access Example:
"*Heap*" + 245



All memory reference is now specified as: **Segment name + Offset**

Segmentation: Logical Address Translation

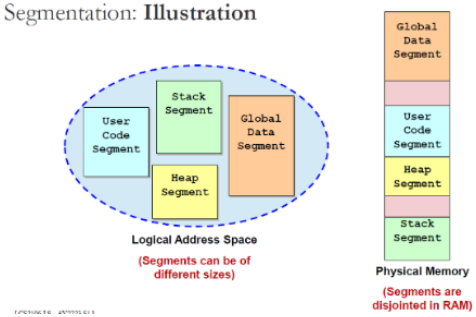
Each segment is mapped to a contiguous physical memory region with a **base address** and a **limit.**

The segmentation name is usually represented as a single number known as **segment id.**

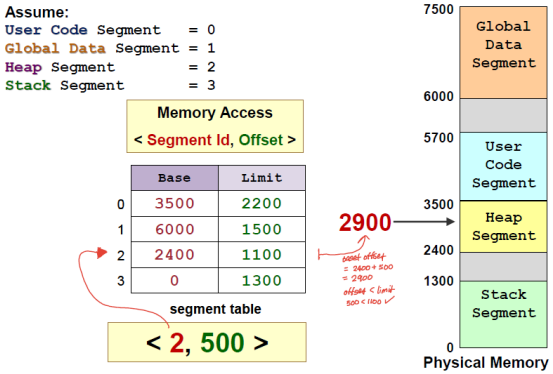
Logical address **<SegID, Offset>**:
SegID used to look up **<Base, Limit>** of the segment in a **segment table**.

- **Physical Address PA = Base + Offset**
- **Offset < Limit for valid access**

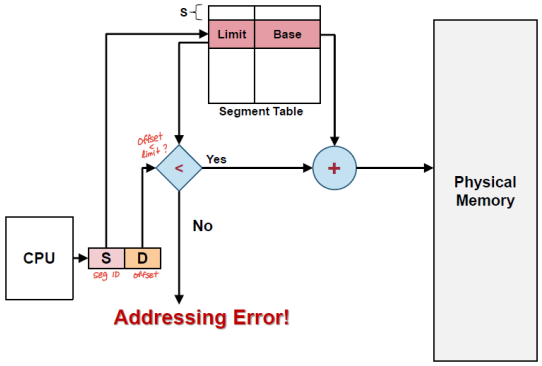
Segmentation: **Illustration**



Logical Address Translation: Illustration



Segmentation: **Hardware Support**



Segmentation Summary

Pros:

- Each segment is an independent contiguous memory space
- Can grow/shrink independently
- Can be protected / shared independently

Cons:

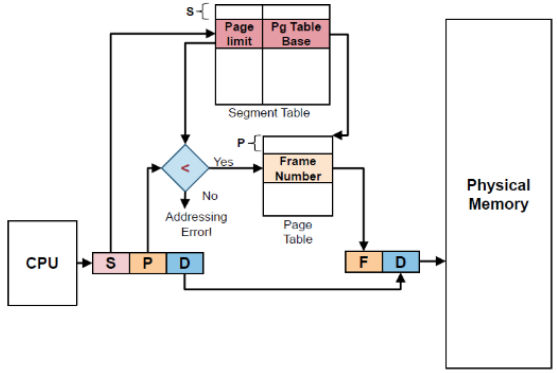
- Segmentation requires variable-size contiguous memory regions
- Can cause external fragmentation

Important observation: *Segmentation is not the same as paging*. Each of them trying to solve a different problem.

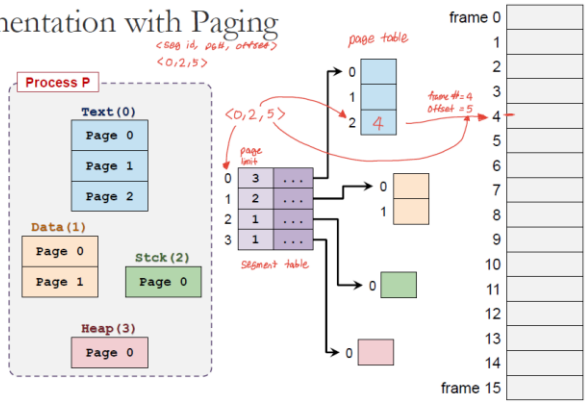
Segmentation with Paging: Basic Idea

Each segment is now composed of several pages instead of a contiguous memory region. Essentially, each segment has a page table. Segment can grow by allocating new pages then add to its page table, and similarly for shrinking.

Segmentation with Paging: Illustration

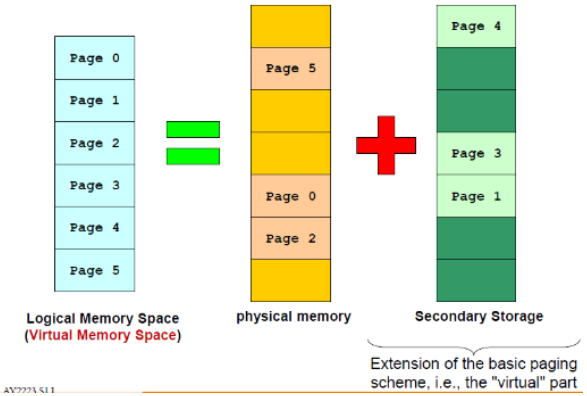


Segmentation with Paging



9. Virtual Memory Management

Virtual Memory: Paging Illustration



Basic Idea

- Obseration: The logical memory space of a process >> physical memory
- Secondary storage has much larger capacity compared to physical memory
- Split the logical address space into pages: Some pages reside in physical memory, other are stored on secondary storage.
- Basic idea remains unchanged: use page table to translate virtual address to physical address

New addition:

To distinguish between two pages types:

Memory resident (pages in physical memory)

Non-memory resident (pages in secondary storage)

Use a (is memory resident?) bit in PTE.

CPU can only access memory resident pages:

- **Page fault:** When CPU tries to access non memory resident page
- OS need to bring a non memory resident page into physical memory

Accessing Page X: General Steps

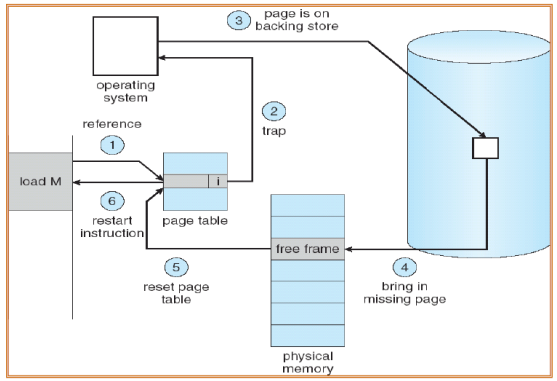
By Hardware

1. Check **page table**:
   - Is page **X *memory resident***?
     - Yes: Access physical memory location. Done.
     - No: Continue to the next step

By OS

2. **Page Fault: Trap** to OS
3. Locate page X in secondary storage
4. Load page X into a physical memory frame
5. Update page table
6. Go to step 1 and retry

Virtual Memory Accessing: Illustration



Virtual memory: Justification

Secondary Storage access time >> Physical memory access time.
If memory access results in page fault most of the time:

- Non-memory resident pages need to be loaded (known as **trashing**)

How do we know that thrashing is unlikely to happen?

Locality Principles

Most programs exhibit these behaviors:

- Most time are spent on a relatively small region of the code
- Within a time period, accesses are made to a relatively small part of the data only

Formalized as **locality principles**

- **Temporal Locality**
  - Memory address which is referenced is likely to be referenced again
- **Spatial Locality**
  - Memory addresses close to a referenced address is likely to be referenced

Virtual Memory and Locality Principle

Exploiting **Temporal Locality**:

- After a page is loaded to physical memory, it is likely to be accessed in near future
  - Cost of loading the page is amortized

Exploiting **Spatial Locality**:

- A page contains contiguous addresses that are likely to be accessed in near future
  - Later access to nearby addresses will not cause page fault

However, there are always exceptions. (Programs that behave badly due to poor design or with malicious intention).

Virtual Memory: Summary

- Completely separate logical memory addresses from physical memory. Amount of physical memory no longer restrict the size of logical memory space.
- More efficient use of physical memory - Page currently not needed can be on secondary storage
- Allow more processes to reside in memory → Improve CPU utilization as there are more processes to choose to run

More on Virtual Memory Management

Large page table with big logical memory space : How to structure the page table for efficiency? → **Page Table Structures**

Each process has limited number of resident memory pages : Which page should be replaced when needed?→ **Page Replacement Algorithms**

Limited physical memory frames : How to distribute among the processes? → **Frame Allocation Policies**

Page Table Structure

Page table information is kept together with the process information and takes up physical memory space. Modern computer systems provide huge logical memory space:

- Huge logical memory space → Large number of pages
- Each page has a page table entry → Large page tables

Problems with large page table

- High overhead
- Fragmented page table: page table occupies several memory pages

Page Table Structure: Direct Paging

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

- **p bits to specify one unique page**
- **2^p page table entries (PTE)**, each contains:
  - Physical frame number
  - Additional information bits (valid, access rights, etc.)

Example: Virtual Address: 32 bits, Page Size = 4KiB, Size of PTE = 2 bytes
P = 32 -12 = 20
*Page Table Size = 2^{20} * 2 bytes = 2MiB*

2-level Paging: Basic Idea

Observation: Process may not use the entire virtual memory space → Full page table is a waste!

- Split the full page table into regions.
- Only a few regions are used
- As memory usage grows, new regions can be allocated
- This idea is similar to split logical memory space into pages.
- Need a directory to keep track of regions (analogues of page table ↔ pages)

2-level Paging: Description

Split page table into smaller page tables

- Each with a **page table number**
- Each **page table size is equal to the page size**

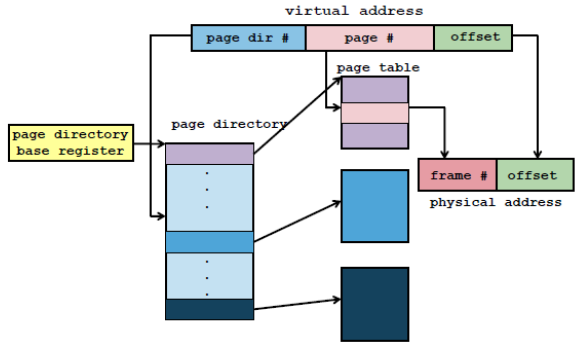
If the original page table has 2^P entries:

- With 2^M smaller page tables, M bits are 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 table

2-level Paging: Illustration

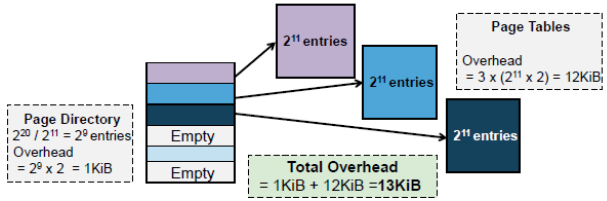


2-Level Paging: Advantages

We can have empty entries in the page directory
→ The corresponding page tables need not be allocated!

Using the same setting as the previous example:

- Assume only 3 page tables are in use
- Overhead = 1 page directory + 3 smaller page tables



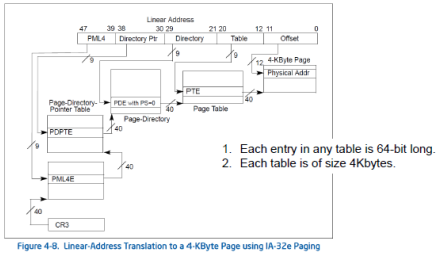


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

Inverted Page Table

Page table is a per process information

- With M processes in memory, there are M independent page tables

Observation

- Difficult to find out which frames are occupied, and by which processes
- Only N physical memory frames can be occupied
- Out of the M page tables, only N entries are valid
- Huge waste: N << Overhead of M page tables

Idea

- Keep a **single** mapping of physical frame to **<pid, page#>**
- pid = process id, page# = page number
- page# is not unique among processes
- pid + page# can uniquely identify a memory page

In a **normal page table**, entries are ordered by page number

- To lookup page X, simply access the Xth entry

In an **inverted page table**, **entries are ordered by frame number**

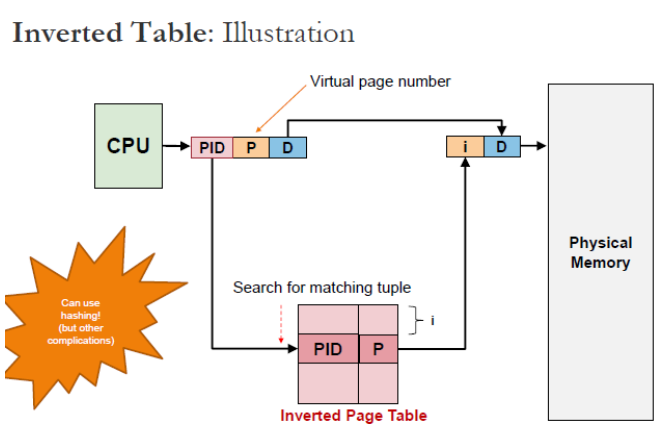
- To lookup page X, need to search the whole table

Advantages:

- Huge saving: one table for all processes
- Frame mangement is easier and faster

Disadvantage:

- Slow translation



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: not modified → no need to write back to storage
- Dirty page: modified → need to write back to storage

Algorithms to find a suitable replacement page:

- Optimum (OPT)
- FIFO
- Least Recently used
- Second-Chance (CLOCK)
- Etc.

Modeling Memory References

In actual memory references: Logical Address = Page Number + Offset. However, to study page replacement algorithms, only **page number** is important.

To simplify discussion, memory references are often modeled as **memory reference strings** , i.e., a sequence of page numbers.

Page Replacement Algorithms: Evaluation

Memory Access

Time: $T_{access} = (1 - p) \times T_{mem} + p * T_{pagefault}$

p = probability of page fault

T_{mem} = access time for memory resident page

$T_{pagefault}$ = access time if page fault occurs

Since $T_{pagefault} \gg T_{mem}$, need to reduce p to keep T_{access} reasonable.

Good algorithm should reduce the total number of page faults.

Page Replacement Algorithms: Optimal Page Replacement (OPT)

Replace the page that **will not** be used again for the **longest period of time.**

- **Guarantees** minimum number of page faults

Unfortunately, not feasible: need future knowledge of memory references.

Still useful: As a base of comparison for other algorithms. The closer to OPT == better algorithm.

Example: OPT (6 Page Faults)

Time	Memory Reference	Frame			Next Use Time			Fault?
		A	B	C				
1	2	2			3	∞	∞	Y
2	3	2	3		3	9	∞	Y
3	2	2	3		6	9	∞	
4	1	2	3	1	6	9	∞	Y
5	5	2	3	5	6	9	8	Y
6	2	2	3	5	10	9	8	
7	4	4	3	5	∞	9	8	Y
8	5	4	3	5	∞	9	11	
9	3	4	3	5	∞	∞	11	
10	2	2	3	5	12	∞	11	Y
11	5	2	3	5	∞	∞	11	
12	2	2	3	5	∞	∞	∞	

Page Replacement Algorithms: FIFO

Memory pages are evicted based on their loading time → Evict the oldest memory page

Implementation:

OS maintains a queue of resident page numbers

- Remove the first page in queue if replacement is needed
- Update the queue during page fault trap

Simple to implement: no hardware support needed

Example: FIFO (9 Page Faults)

Time	Memory Reference	Frame			Loaded at Time			Fault?
		A	B	C				
1	2	2			1			Y
2	3	2	3		1	2		Y
3	2	2	3		1	2		
4	1	2	3	1	1	2	4	Y
5	5	5	3	1	5	2	4	Y
6	2	5	2	1	5	6	4	Y
7	4	5	2	4	5	6	7	Y
8	5	5	2	4	5	6	7	
9	3	3	2	4	9	6	7	Y
10	2	3	2	4	9	6	7	
11	5	3	5	4	9	11	7	Y
12	2	3	5	2	9	11	12	Y

FIFO: Problems

If number of physical frames increases (e.g., more RAM), number of page faults should decrease.

FIFO violates this simple intuition! Use 3 / 4 frames to try: 1 2 3 4 1 2 5 1 2 3 4 5

Opposite behavior (↑ frames → ↑ page faults)

- Known as **Belady's Anomaly**

Reason: FIFO does not exploit **temporal locality** (Memory address which is referenced is likely to be referenced again).

Page Replacement Algorithms: Least Recently Used (LRU)

Make use of **temporal locality**: **Replace the page that has not been accessed in the longest time**.

Expect a page to be reused in a short time window

- Have not accessed for some time → most likely will not be accessed again

Notes:

- Attempts to approximate the OPT algorithm: gives good results generally
- Does not suffer from **Belady's Anomaly**

Example: LRU (7 Page Faults)

Time	Memory Reference	Frame			Last Use Time			Fault?
		A	B	C				
1	**2**	2			1			Y
2	**3**	2	3		1	2		Y
3	**2**	2	3		3	2		
4	**1**	2	3	1	3	2	4	Y
5	**5**	2	5	1	3	5	4	Y
6	**2**	2	5	1	6	5	4	
7	**4**	2	5	4	6	5	7	Y
8	**5**	2	5	4	6	8	7	
9	**3**	3	5	4	9	8	7	Y
10	**2**	3	5	2	9	8	10	Y
11	**5**	3	5	2	9	11	10	
12	**2**	3	5	2	9	11	12	

LRU: Implementation Details

Implementing LRU is not easy

- Need to keep track of the "last access time" somehow

Need substantial hardware support

Approach A – Use a counter

- A logical "time" counter, 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:
  - Need to search through all pages
  - "Time of use" is forever increasing (overflow possible!)

Approach B – Use a "Stack"

- Maintain a stack of page numbers
- If page **X** is referenced
  - Remove from the stack (if entry exists)
  - Push on top of stack
- Replace the page at the bottom of stack
  - No need to search through all entries
- Problems:
  - Not a pure stack: Entries can be removed anywhere in the stack

Hard to implement in hardware

Page Replacement Algorithms: Second-Chance Page Replacement (CLOCK)

Modified FIFO to give a second chance to pages that are accessed

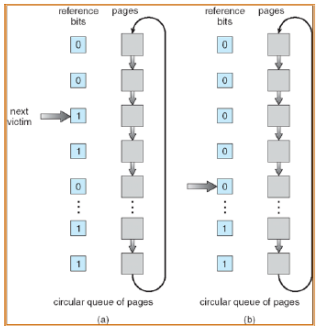
- Each PTE now maintains a "reference bit":
  - 1 = Accessed, 0 = Not Accessed

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
   - Load time reset → page taken as newly loaded
   - Next FIFO page is selected, go to Step 2

**CLOCK degenerates into FIFO Algorithm when all pages have reference bit == 1 (or all == 0).*

Second-Chance: Implementation Details



- 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 to a page with '0' reference bit
  - Clear the reference bit as pointer passes through

Example: CLOCK (6 Page Faults)

Time	Memory Reference	Frame (with Ref Bit)			Fault?
		A	B	C	
1	**2**	►2 (0)			Y
2	**3**	►2 (0)	3 (0)		Y
3	**2**	►2 (1)	3 (0)		
4	**1**	►2 (1)	3 (0)	1 (0)	Y
5	**5**	2 (0)	5 (0)	►1 (0)	Y
6	**2**	2 (1)	5 (0)	►1 (0)	
7	**4**	►2 (1)	5 (0)	4 (0)	Y
8	**5**	►2 (1)	5 (1)	4 (0)	
9	**3**	►2 (0)	5 (0)	3 (0)	Y
10	**2**	►2 (1)	5 (0)	3 (0)	
11	**5**	►2 (1)	5 (1)	3 (0)	
12	**2**	►2 (1)	5 (1)	3 (0)	

► Victim Page

Frame Allocation

- There are N physical memory frames
- There are M processes competing for frames

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

Simple approaches:

Equal Allocation:

- Each process gets **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 gets **$size_p/size_{total}$ * N frames**

Frame Allocation and Page Replacement

The implicit assumption for page replacement algorithms discussed:

- Victim pages are selected **among pages of the process** that causes page fault
- Known as **local replacement**

If 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!
- Known as **global replacement**

Local vs Global Replacement

Local Replacement:

Pros:

- Frames allocated to a process remain constant → Performance is stable between multiple runs

Cons:

- If frames allocated to a process are not enough → hinder the performance of the process

Global Replacement:

Pros:

- Allow dynamic self adjustment between processes
  - Process that needs more frames can get from other

Cons:

- Badly behaved process can negatively affect others

Frames allocated to a process can be different from run to run

Frame Allocation and Thrashing

Insufficient physical frames → Thrashing in process

- Heavy I/O to bring non-resident pages into RAM

Hard to find the right number of frames:

If **global replacement** is used:

- A thrashing process "steals" page from other processes → causes other processes to thrashing (**Cascading Thrashing**)

If **local replacement** is used:

- Thrashing can be limited to one process
- But that single process can hog the I/O and degrades the performance of other processes

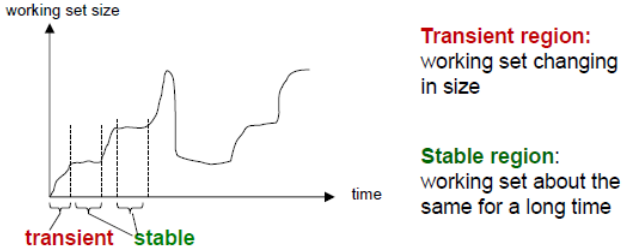
Observation: The set of pages referenced by a process is relatively constant in a period of time (known as **locality)**. However, as time passes, the set of pages can change.

Example: When a function is executing, memory references are likely on:

- local variables, parameters, code in that function
- these pages define the locality for the function

After the function terminates, references will change to another set of pages.

Working Set Model



Using the observation on locality:

- In a new "locality":
  - A process will cause page fault for the set of pages
- With the set of pages loaded in frames:
  - No/few page faults until process transits to new locality

Working Set Model: Defines Working Set Window Δ (an interval of time)

- $W(t, \Delta)$ = active pages in the interval at time t
- Allocate enough frames for pages in $W(t, \Delta)$ to reduce the number of page faults

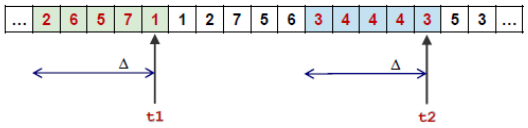
Accuracy of working set model is directly affected by the choice of Δ

Too small → may miss pages in the current locality

Too big → may contains pages from different locality

Working Set Model: Illustration

- Example memory reference strings



- 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)
- Try using different Δ values

L10: FILE MANAGEMENT INTRODUCTION

File System provides:

- Abstraction on top of physical media
- High-level resource management scheme
- Protection between processes and users
- Sharing between processes and users

General Criteria

Self-contained

- Information stored on a media is enough to describe the entire organization
- Should be able to "plug-and-play"

Persistent

- Beyond the lifetime of OS and processes

Efficient

- Provide good management of free and used space
- Minimum overhead for bookkeeping information

	Memory Management	File System Management
Underlying Storage	RAM	Disk
Access Speed	Constant	Variable Disk I/O Time
Unit of addressing	Physical memory address	Disk Sector
Usage	Address space for process **Implicit** when process runs	Non-volatile data **Explicit** access
Organization	Paging/Segmentation: determined by HW & OS	Many different FS: ext* (Linux), FAT* (windows), HFS* (macOS)

Files

Files represent a logical unit of information created by process. As an abstraction, it is essentially an **Abstract Data Type:** a set of common operations with various possible implementations.

Contains:

- **Data:** Information structured in some ways
- **Metadata:** Additional information associated with the file: known as file attributes

File Metadata

Name	Human readable reference to the file
Identifier	Unique ID for the file used internally by FS
Type	Indicate type of of file E.g. executable, txt, obj, dir, etc
Size	Current size of file (in bytes/words/blocks)
Protection	Access permissions, RWX
Time, date, and owner information	Creation, last modification time, owner id etc
Table of content	Information for FS to determine how to access the file

File Name

Different FS has different naming rule to determine valid file name.

Common naming rules:

- Length of file name
- Case sensitivity
- Allowed special symbols
- File extension
  - Usually Name.Extension
  - On some FS used to indicate file type

File Type

OS commonly supports a number of file types. Each file type has:

- An associated set of operations
- Possibly a specific program for processing

Common File Types:

Regular Files: contains user information (ASCII, bin)
Directories: system files for FS structure
Special Files: character/block oriented

ASCII Files:

- Text file, programming src code, etc.
- Can be displayed as is

Binary Files:

- Executable, java class, pdf, mp4, etc.
- Have predefined internal structure that can be processed by specific program

Distinguishing File Type

1. Use file extension as indication
   a. Used by windows OS
   b. XXX.docx → Words Doc
   c. Change of extension implies change in file type
2. Use embedded information in file
   a. Used by unix
   b. Usually stored at beginning of file
   c. Commonly known as magic number

Operations on File Metadata

Rename: Change filename
Change Attributes: File access perms / dates / ownership, etc.
Read Attribute: Get file creation time

File Protection

Controlled access to the information stored in a file.

Types of File Access

Read: Retrieve information from file
Write: Write/Rewrite the file
Execute: Load file into memory and execute it
Append: Add new information to end of file
Delete: Remove the file from FS
List: Read metadata of a file

File Protection

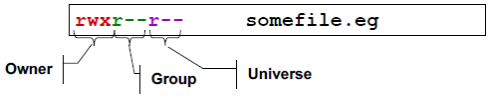
Most common approach – restrict access based on user identity. Most general scheme known as **Access Control List**.

- A list of user identity and the allowed access types
- Pros: Very customizable
- Cons: Too much information associated with file

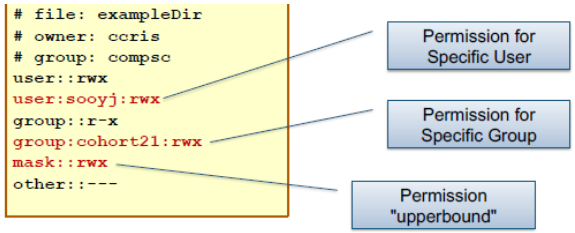
File Protection: Permission Bits

Classified the users into three classes:
Owner: The user who created the file
Group: A set of users who need similar access to a file
Universe: All other users in the system

Example (UNIX)
Define permission of three access types (RWX) for the 3 classes of users. (Use ls -l to see perm bits).



In UNIX, ACL can be a **Minimal ACL** (same as permission bit), or **Extended ACL** (added named users / groups). The *getfacl* command can be used to get ACL information.



File Data Structure

1. **Array of bytes:**

Each byte has an unique **offset** from the file start

2. **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)

3. **Variable Length Records:**

Flexible but harder to locate a record

File Data: Access Methods

1. **Sequential Access:**
   - Data read in order starting from beginning
   - Cannot skip but can be rewound

2. **Random Access**
   - Data can be read in any order
   - Can be provided in two ways
   - **Read(Offset)**: Every read operation explicitly state the position to be accessed
   - **Seek(Offset):** Special operation provided to move to a new location in file

* Unix and Windows uses *Seek*

3. **Direct Access**
   - Used for file that contains fixed-length records
   - Allow random access to any record directly
   - Very useful where there are large amount of records
   - Basic random access method can be viewed as a special case: where each **record == one byte**

File Data Generic Operations

Create	New file is created with no data
Open	Performed before further operations to prepare the necessary information file operations later
Read	Reads data from file, usually from current position
Write	Write data to file, usually starting from current position
Repositioning/Seek	Move the current position to a new location. Now actual R/W is performed
Truncate	Removes data between specified position to EOF

File Operations as System Calls

OS provides file operations as system calls:

- Provide protection, concurrency and efficient access
- Maintain information

**UNIX: open(), read(), write(), lseek(), close()*

Information kept for an opened file:

- **File Descriptor**: Unique ID of the file
- **Disk Location**: Actual file location on disk
- **Open Count**: How many processes has this file opened? (useful to determine when to remove the entry in table)

File Information in the OS

Consider:

- Several processes can open the same file
- Several different files can be opened at any time

Per-process open file table:

- Keep track of the open files for a process
- Each entry points to the system-wide open-file table entries

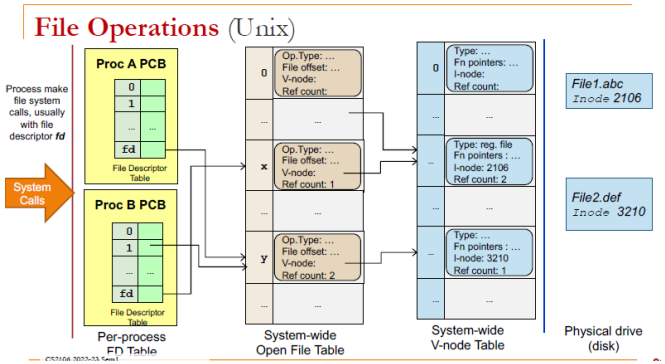
System-wide open-file table:

- Keep track of all the open files in the system
- Each entry points to a V-node entry

System-wide V-node(Virtual node) table

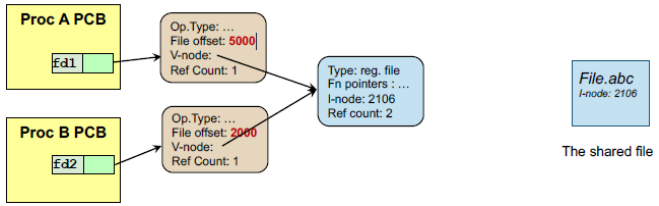
- To link with the file on physical drive
- Contains the information about the physical location of the file

See Appendix for Full Diagram



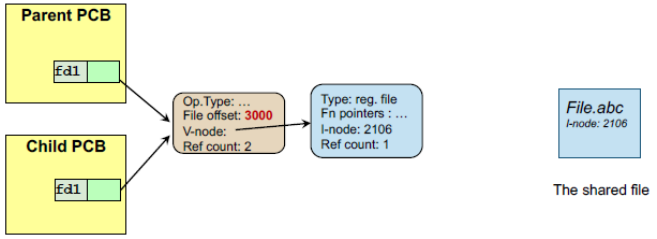
Process Sharing File in Unix : Case 1

A file is opened twice from two processes (I/O can occur at independent offsets)



Process Sharing File in Unix: Case 2

Two processes using the same open file table entry (only one offset → I/O changes the offset for the other process)



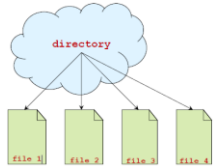
Directory

Directory (folder) is used to

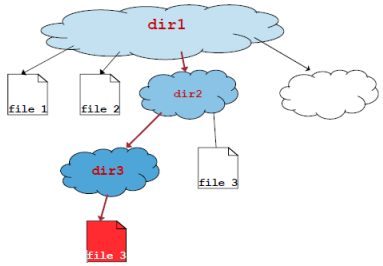
- Provide a logical grouping of files (the user view of directory)
- Keep track of files (actual system usage of directory)

Several ways to structure directory: **single-level, tree-structure, directed acylic graph (DAG), general graph**.

Single Level



Tree Structured



General Idea:

- Directories can be recursively embedded in other directories
- Naturally forms a tree structure

Two ways to refer to a file:

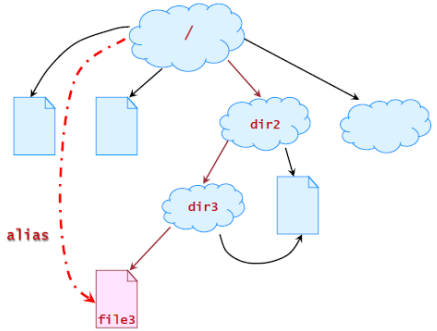
Absolute Pathname:

- Directory names followed from root of tree + final file
- i.e. the path from root directory to 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

Directory Structure: DAG



DAG

If a file can be shared:

- Only one copy of actual content
- "Appears" in multiple directories with a different path names
- Tree Structure → DAG

Implementations in Unix: **Hard Link** (NOT ALLOWED for directories).

Unix Hard Link

Directory A is owner of file F. Directory B wants to share F.

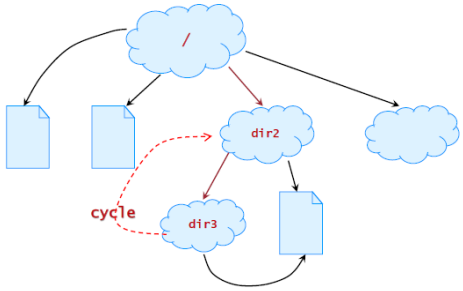
Hard Link:

A and B has separate pointers pointing to the actual file F in disk

Pros: Low overhead, only pointers are added in directory
Cons: Deletion problems: If B deletes F? If A deletes F?

* Unix command: ln

Directory Structure: General Graph



General Graph Directory Structure is not desirable.

- Hard to traverse (need to prevent infinite looping)
- Hard to determine when to remove a file/directory

In Unix: **Symbolic Link** is allowed to linked to directory. General Graphs can be created.

DAG: Unix Symbolic Link / Soft Link

The symbolic link is a **special link file, G**. **G** contains the path name of **F**. When G is accessed: Find out where is F, then access F.

Pros: Simple deletion: If symbolic link is deleted G deleted, not F
If the linked file is deleted, F is gone, G remains (but not working – dangling link)
Cons: Larger overhead: Special link file takes up actual disk space

* Unix: ln -s

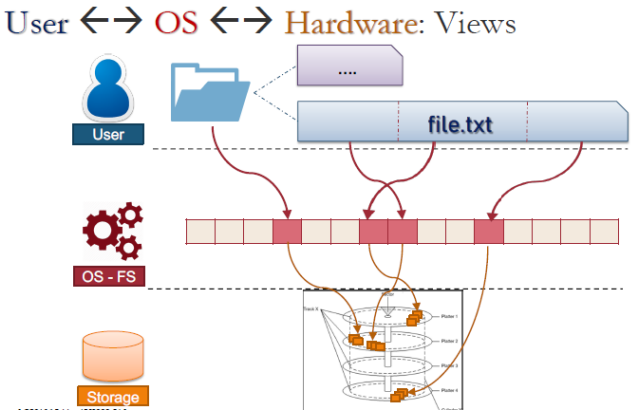
Summary

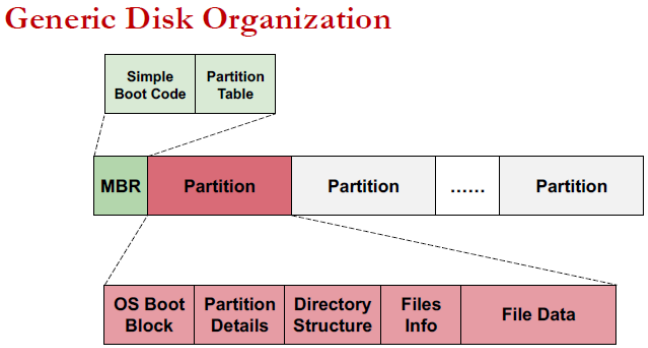
Hard Link: For FILES only
Symbolic Link: FILES or DIRECTORIES

L11: FILE SYSTEM IMPLEMENTATIONS

File systems are stored on storage media. General disk structure:

- 1D array of logical blocks
- Logical block: Smallest accessible unit (usually 512-bytes to 4KB)
- Logical block is mapped into **disk sectors**
- Layout of disk sector is **hardware dependent**





Disk Organization
Master Boot Record (MBR) at sector 0 with partition table, followed by one or more partitions. Each partition can contain an independent file system.

A file system generally contains:

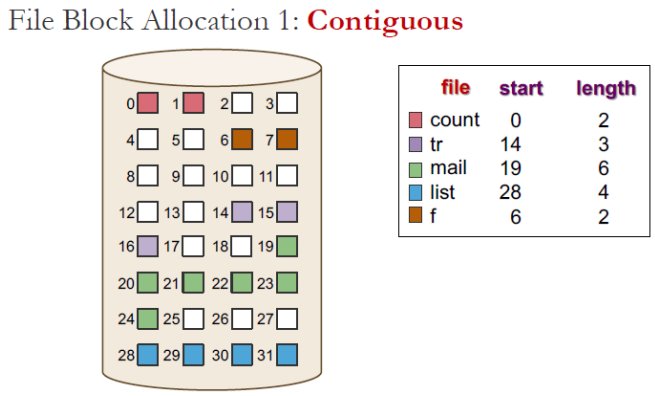
- OS Boot-Up information
- Partition details: total number of blocks, number and location of free disk blocks
- Directory structure
- Files Information
- Actual File Data

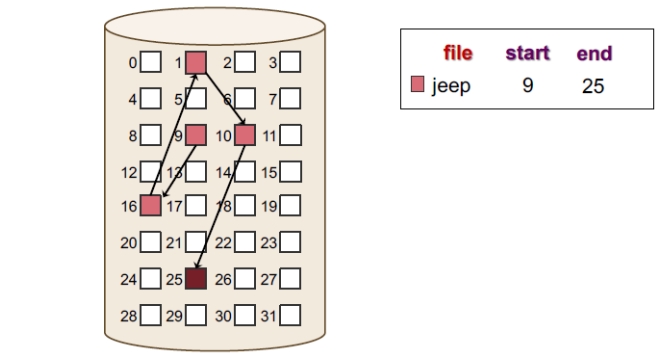
Implementing File
Logical view of a file: A collection of **logical blocks**

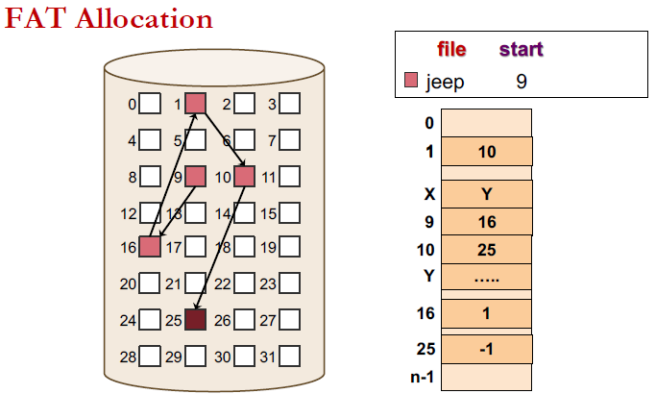
When file size != multiple of logical blocks → last block may have **internal fragmentation**.

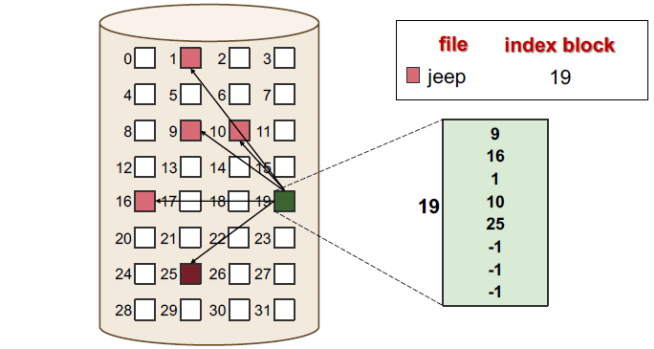
A good file system implementation must'

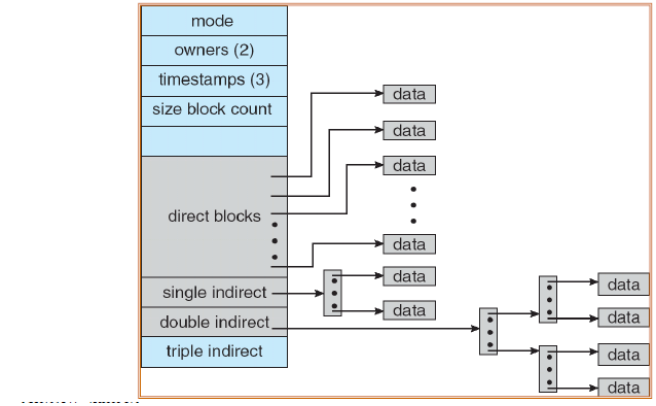
- Keep track of the logical blocks
- Allow efficient access
- Disk space is utilized effectively











File Block Allocation: Contiguous
Allocate consecutive disk blocks to a file.

Pros: Simple to keep track
Fast Access (only need to seek to first block)
Cons: External Fragmentation
File size need to be specified in advance

File Block Allocation: Linked List

Keep a linked list of disk blocks.

Each disk block stores:
- The next disk block number (i.e. act as pointer)
- Actual file data

File information stores
- First and last disk block number

Pros: Solves fragmentation problem
Cons: Random access in a file is very slow
Part of disk block is used for pointer
Less reliable (what if one of the ptr is incorrect?)

File Block Allocation: Linked List V2.0 (FAT)

Move all the block poitners into a single table known as File Allocation Table (FAT). FAT is in memory at all time. It is simple yet efficient (used by MS-DOS).

Pros: Faster Random Access (The linkedlist traversal now in memory)
Cons: FAT keep tracks of **all disk blocks** in a partition
Can be huge when disk is large
Consume valuable memory space – expensive overhead

File Block Allocation: Indexed Allocation

Instead of keeping track of every file in the system, we maintain blocks for each file. Each file has an **index block**
- An array of disk block addresses
- IndexBlock[N] == Nth block address

Pros: Lesser memory overhead (only index block of opened file needs to be in memory)
Fast direct access
Cons: Limited maximum file size
Max number of blocks == Number of index block entries

Indexed Block Allocation: Variation

Several schemes to allow larger files:
- **Linked Scheme:**
  Keep a linked list of index nodes. Cons: expensive since traversal cost.
- **Multilevel index:**
  Similar idea as multi-level paging. Can be generalized to any number of levels.
- **Combined scheme:**
  Combination of direct indexing and multi-level index scheme.

Free space management

To perform file allocation: need to know which disk block is free

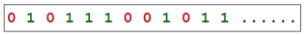
OS Boot Block	Partition Details	Directory Structure	Files Info	File Data

Free space management: Maintain free space information
Allocate:
- Needed when file is created or enlarged

Free:
- Add free disk block to free space list
- Needed when file is deleted or truncated

Free Space Managerment: Bitmap

Each disk block is represented by 1 bit



- Occupied Blocks = 0, 2, 6, 7, 9, …
- Free Blocks = 1, 3, 4, 5, 8, 10, 11, …

Pros: Provide a good set of manipulations
Cons: Need to keep in memory for efficiency reason

Free Space Management: Linked List

Use a linked list of disk blocks. Each disk block contains:
- A number of free disk block numbers, or
- 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

Directory Structure: Overview

Two main tasks:
1. Keeps track of the files in a directory – possibly with the file metadata
2. Map the file name to the file information

File must be opened before use: *open("data.txt");*
The purpose of the open operation: Locate the file information using pathname + file name.

Given a full path name: need to recursively search the directories along the path to arrive at the file information: `/dir2/dir3/data.txt`.

Sub-directory is usually stored as file entry with special file type in a directory.

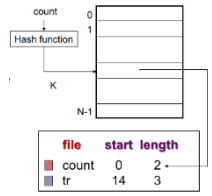
Directory Implementation: Linear List

Directory consists of a linear list. Each entry represents a file:
- Store file name (minimum) and possibly other metadata
- Store **file information** or **pointer** to file information

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.

Directory Implementation: Hash Table

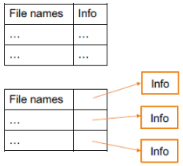
Each directory contains a Hash table of size N. To locate a file by filename:
- File name is hashed into index K from 0 to N-1
- HashTable[K] is inspected to match file name
- Usually, chained collision resolution is used



Pros: Fast Lookup
Cons: Hashtable has limited size
Depends on good hash function

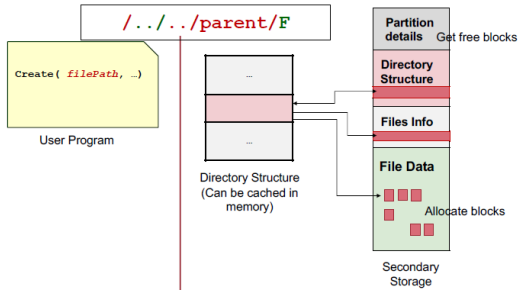
Directory Implementation: File Information

File information consists of:
- File name and other metadata
- Disk blocks information



Two common approaches to store:
1. Store everything in directory entry
2. Store only file name and points to some data structure for other info

File Creation: Illustration



To create a file /.../.../parent/F:

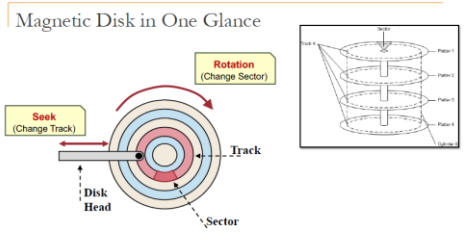
Use full pathname to locate the parent directory
- Search for filename F to avoid duplicates
- If found, file creation terminates with error
- Search could be on the cached directory structure

Use free space list to find free disk block(s)
- Depends on allocation scheme

Add an entry to parent directory
- With relevant file information
- File name, disk block information etc

file	start	length
count	0	2
tr	14	3

Disk I/O Scheduling



3-Stage of Read/Write Process

Time taken to perform a read/write operation:

$= [Seek\ Time] + [Rotational\ Latency] + [Transfer\ Time]$

Position the disk head over the proper track

- Time taken is known as [Seek time] (2ms-10ms)

Wait for the desired sector to rotate under the read/write head

- Time taken is known as [Rotational latency] (4ms-12ms)

Transfer the sector(s)

- Time taken is known as [Transfer time] (in order of microsec)

Disk Scheduling: The Problem

Due to the significant seek and rotational latency, OS should schedule the disk I/O requests.

I/O (disk) scheduling:

- Intention of reducing **overall waiting time**
- As rotational latency is hard to mitigate, we focus on reducing the **seeking time**
- Balance the need for **high throughput** while trying to **fairly share I/O requests** amongst processes

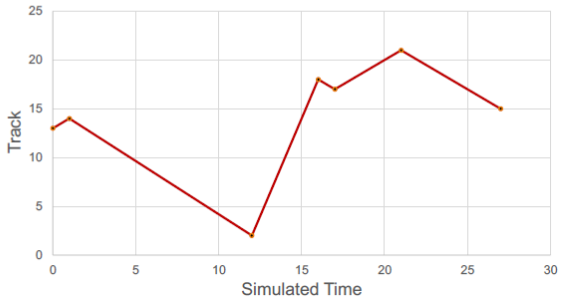
Disk Scheduling: Algorithms

Consider the following disk I/O requests indicated by only the track number (magnetic disks):

A few candidates: **FCFS, SSF** (Shortest Seek First), **SCAN**.

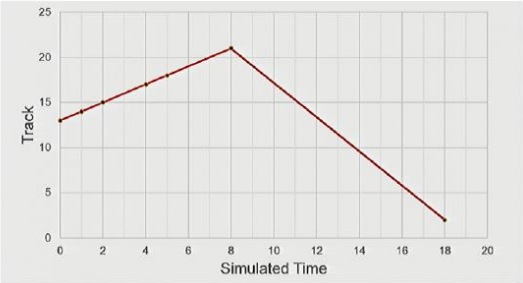
FCFS: Disk Head Movement

- Request: **[13, 14, 2, 18, 17, 21, 15]**



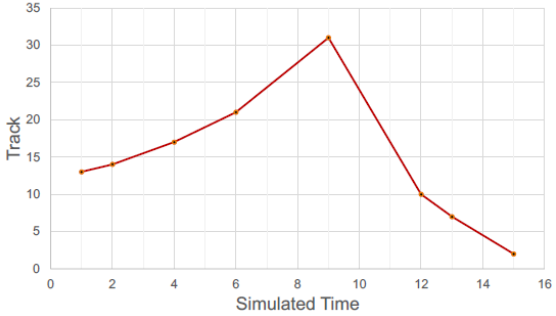
SSF: Disk Head Movement

- Request: **[13, 14, 2, 18, 17, 21, 15]**



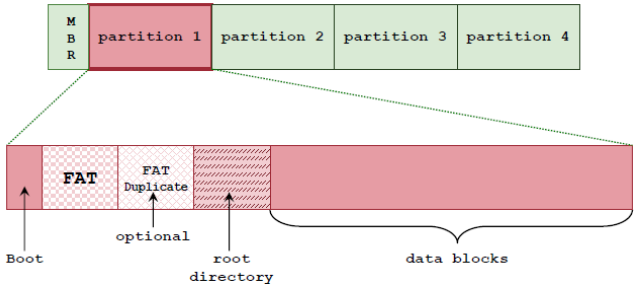
SCAN: Disk Head Movement

- disk I/O requests indicated by only the **track numbe**r **[13, 14, 2, 10, 17, 31, 21, 7]**



L12: FILE SYSTEM CASE STUDIES

Microsoft FAT File System Layout

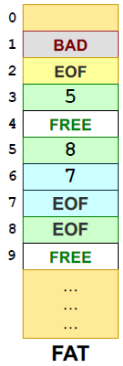


File Data are allocated to:

- A number of data blocks / data block clusters
- Allocation info is kept as a **linked list**
  - All data block pointers kept separately in the File Allocation Table (FAT)

File Allocation Table (FAT)

- One entry per data block/cluster
- Store disk block information (Free? Next block (if occupied)? Damaged?
- OS will cache in RAM to facilitate linked list traversal



FAT Entry contains either:

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

Example:
Block 3 → 5 → 8 → EOF; Block 4 and 9 are free; Block 1 is unusable

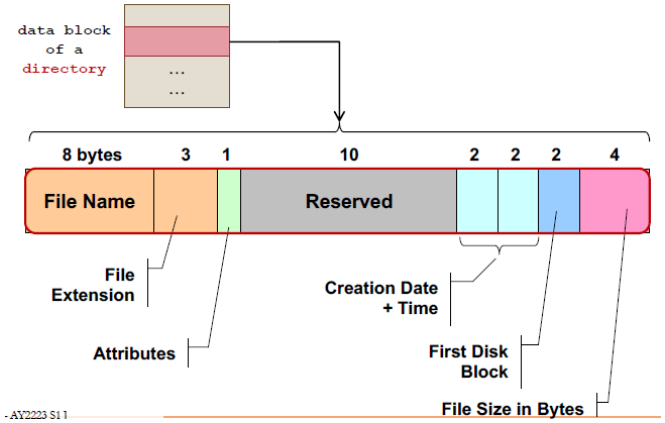
Directory Structure and File Information
Directory (folder) is represented as

- Special type of file
- Root directory is stored in a special location
  - Other directories are stored in the data blocks
- Each file/subdirectory within the folder:
  - Represented as **directory entry**

Directory Entry:

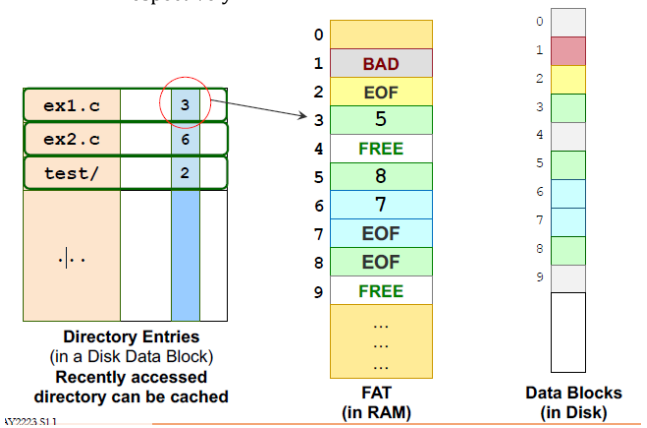
- Fixed-size 32-bytes per entry
- Contains
  - Name + Extension
  - Attributes (Read-Only, Directory/File flag, Hidden etc)
  - Creation Date + Time
  - First disk block + File Size

Directory Entry Illustration



Directory Entry Fields

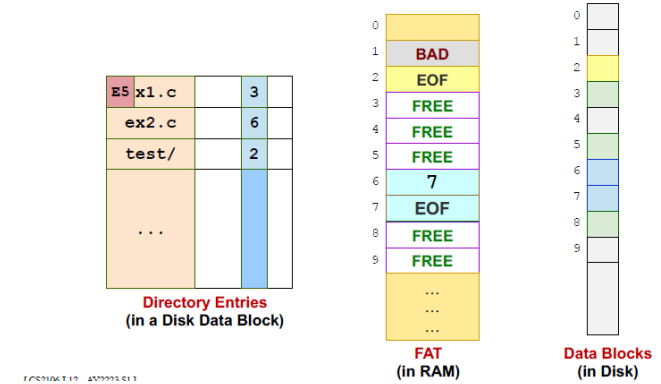
- **File Name + Extension**
  - Limited to 8 + 3 characters
  - The first byte of file name may have special meaning:
    - Deleted, End of directory entries, Parent directory, etc.
- **File Creation Time and Date**
  - Year is limited to 1980 to 2107
  - Accuracy of second is +-2 seconds
- **First Disk Block Index**
  - Different variants uses different number of bits:
  - 12, 16 and 32 bits for FAT12, FAT16, and FAT32 respectively



1. Use first disk block number stored in directory entry to find the starting point of the linked disk blocks
2. Use FAT to find out the subsequent disk blocks number (terminated by special value (EOF))
3. Use disk block number to perform actual disk access on the data blocks

For a directory, the disk blocks contain: Directory entries for the files/subfolders within that directory

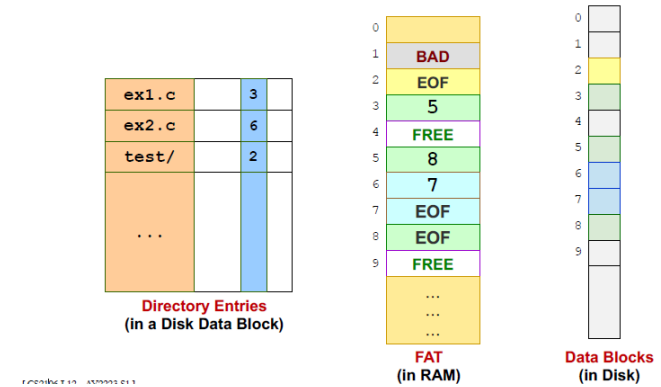
Example of Common Tasks: File Delete



Deletion:

1. Set the first byte of filename to [0xE5]
2. Traverse through FAT table and set all blocks assigned to this file as FREE

Example of Common Tasks: Free Space



Variants: FAT12, FAT16, and FAT32
Main motivation: Support larger hard disk as a single partition

Two major themes:
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 block/cluster => More bits to represent each disk block/cluster

Existing Variants: FAT12, FAT16 and FAT32

FAT12/FAT16/FAT32

Generally, cluster size + FAT size determines the largest usable partition.

Example (**4KiB** Cluster):

FAT12	FAT16	FAT32
2^{12} clusters Largest Partition $= 4KB * 2^{12}$ $= 16MB$	2^{16} clusters Largest Partition $= 4KB * 2^{16}$ $= 256MB$	2^{28} clusters Largest Partition $= 4KB * 2^{28}$ $= 1TB$

*FAT32 only uses 28bits. The other bits are saved for future uses.

Note that the actual size is a little lesser:

- Special values (EOF, FREE, etc) reduces total number of valid data block/cluster indices

Cluster Size

Pros: Larger cluster size → Larger usable partition

Cons: Larger cluster size → Larger internal fragmentation

On **FAT32**, there are further limitations:

- 32-bit sector count → Limited partition size
- Only 28-bit is used in the disk block/cluster index

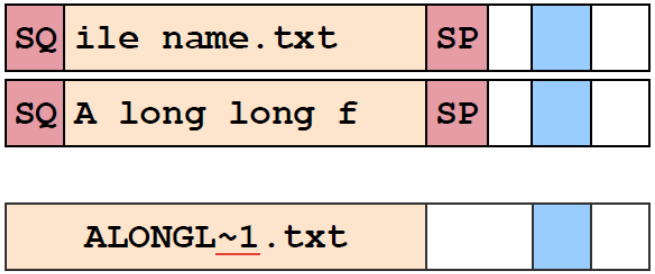
Longer File Name Support

Virtual FAT (VFAT)

- Support long filenames up to 255 characters
- First used in Win95 (FAT16)

A work around rather than redesign:

- Use multiple directory entries for a file with long file name
- Need to keep the 8+3 short version for backward compatibility



SQ → Sequence Number

SP → Special Attribute

Summary

Partition Details → FAT

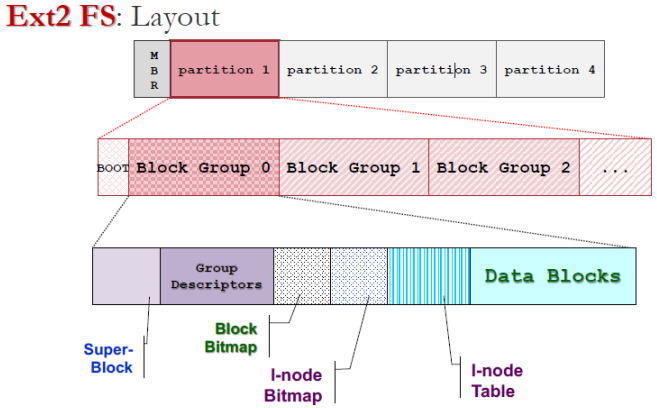
Directory Structure → Data Blocks

File Info → Data Blocks

File Data → Data Blocks

Extended-2 File System (Ext2)

One of the most popular file systems used in Linux



- Disk space is split into **Blocks**
  - Correspond to one or more disk sector
  - Similar to disk cluster in FAT FS
- Blocks are grouped into **Block Groups**
- Each file/directory is described by:
  - A single special structure known as **I-Node** *(Index Node)*
- **I-Node** contains:
  - File metadata (access right, creation time etc)
  - Data block addresses

Partition Information

Superblock

- Describe the whole file system
- Includes:
  - Total I-Nodes number, I-Nodes per group
  - Total disk blocks, Disk Blocks per group
  - Etc
- Duplicated in each block group for redundancy

Group Descriptors

- Describe each of the block group
  - Number of free disk blocks, free I-nodes
  - Location of the bitmaps
- Duplicated in each block group as well

Block Bitmap

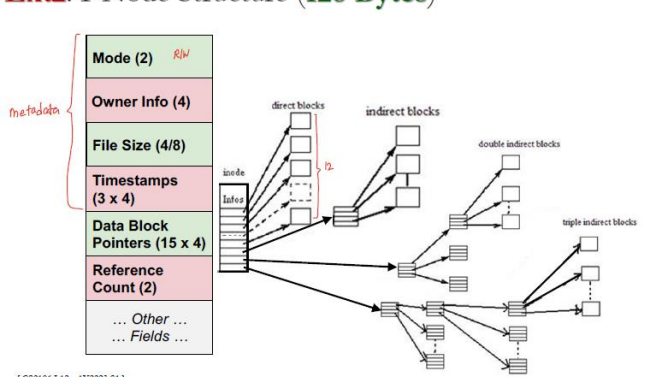
- Keep track of the usage status of blocks of this block group (1=Occupied, 0=Free)

I-Node Bitmap

- Keep track of the usage status of I-Nodes of this block group (1=Occupied, 0=Free)

I-Node Table

- An array of I-Nodes
  - Each I-Node can be accessed by a unique index
- Contains only I-Nodes of this block group



I-Node Data Blocks

- Other than file metadata, I-Node contains
  - 15 disk block pointers (disk block number)
- First 12 pointers:
  - Points to disk blocks that stores actual data
  - Known as the **Direct Blocks**
- 13th Pointer:
  - Points to a disk block that stores direct pointers
  - Known as **Single Indirect Block**
- 14th Pointer:
  - Points to a disk block that contains a number of single indirect block
  - Known as **Double Indirect Block**
- 15th Pointer:
  - Points to a disk block that contains a number of double indirect block
  - Known as **Triple Indirect Block**

I-Node Data Block Example

The design of I-Node allows:

- Fast access to small file
- Flexibility in handling huge file

E.g. (4 bytes block address, 1KB disk block)

Direct blocks:

- 12 data block x 1KiB = 12 KiB

Single Indirect block:

- Number of entries in a disk block = 2^{10} / 4 = 256 entries
- 256 x 1 KiB = 256 KiB

Double Indirect block:

- 256^2 x 1 KiB = 64 MiB

Triple Indirect block:

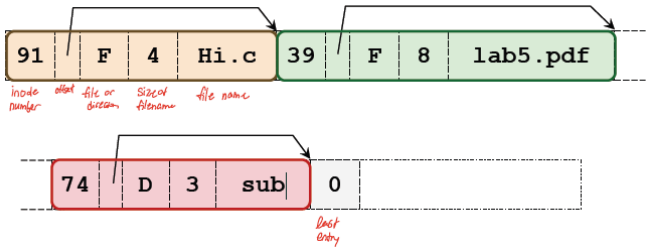
- 256^3 x 1 KiB = 16 GiB

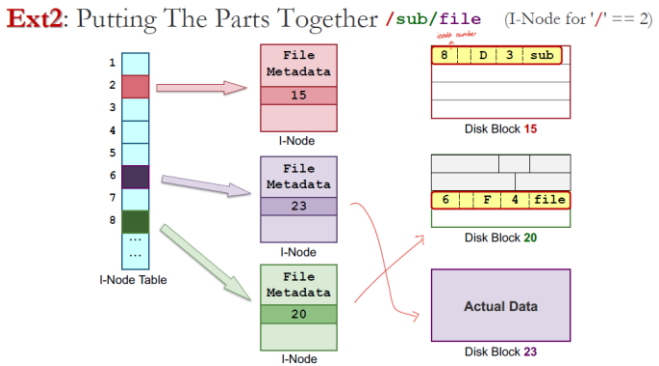
Directory Structure

- Data blocks of a directory stores: **A linked list of directory entries** for file/subdirectories information within this directory

Each directory entry contains:

- I-Node number for that file/subdirectory
- Size of this directory entry – for locating the next directory entry
- Length of the file/subdirectory name
- Type: File or Subdirectory – other special file type is also possible
- File/Subdirectory name (up to 255 characters)

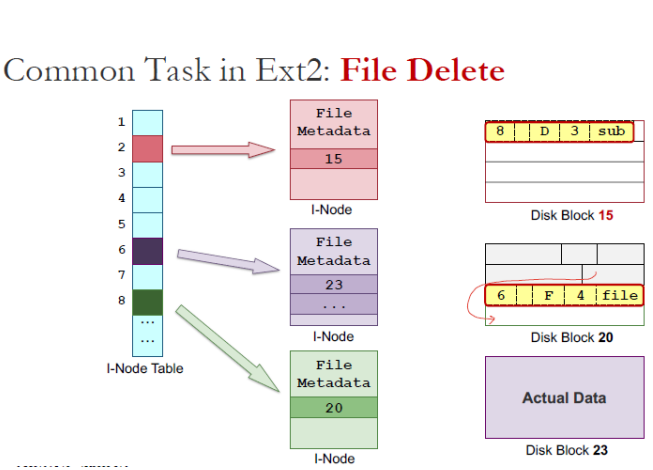




Ext2 FS Summary

Given a pathname, e.g. "/sub/file"

1. Let **CurDir** = "/"
   Root directory usually has fixed I-Node number (e.g. 2)
   Read the actual I-Node
2. Look at the next part in pathname:
   If it is a directory, e.g. "sub/"
   Locate the directory entry in **CurDir**
   Retrieve I-Node number, then read the actual I-Node
   CurDir = next part in pathname
   Goto step 2
   Else it is a *file*
   Locate the directory entry in **CurDir**
   Retrieve I-Node number, then read the actual I-Node



Ext2: Deleting a file

Remove its directory entry from the parent directory:

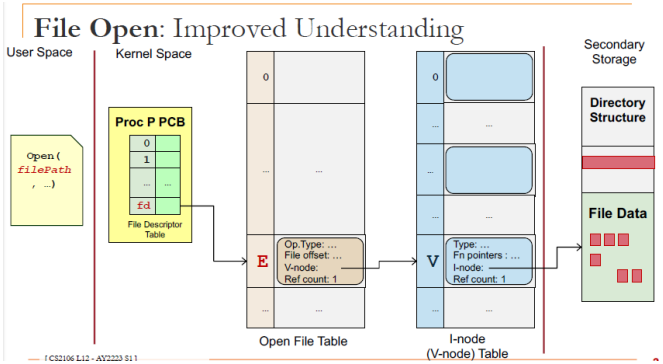
- Point the previous entry to the next entry/end
- To remove first entry: Blank record is needed

Update I-node bitmap:

- Mark the corresponding I-node as free

Update Block bitmap:

- Mark the corresponding block(s) as free



File Operation: Open

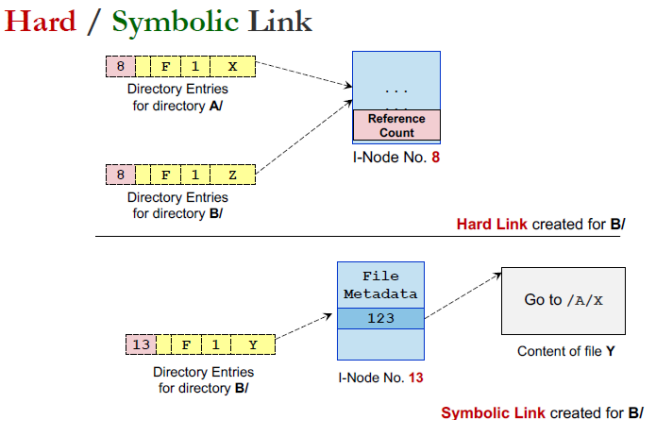
Process P open file /.../.../.../F:

- Use full path name to locate 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 and v in I-node table if not already present
- Creates an entry in P's table to point to E (file descriptor) and pointer from E's entry to V
- Return the file descriptor of this
- The returned file descriptor is used for further read/write operation

Hard/Symbolic Link with I-Node

Scenario:

- Directory A contains file X, with I-Node# XN
- Directory B wants to share X



Ext2 FS: Hard Link and Symbolic Link

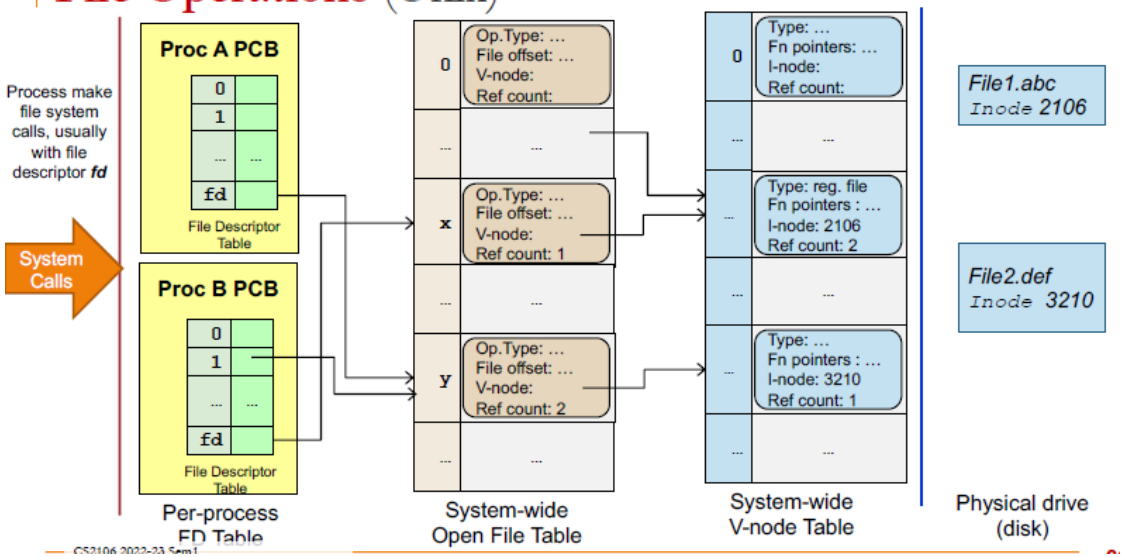
Hard Link problems:

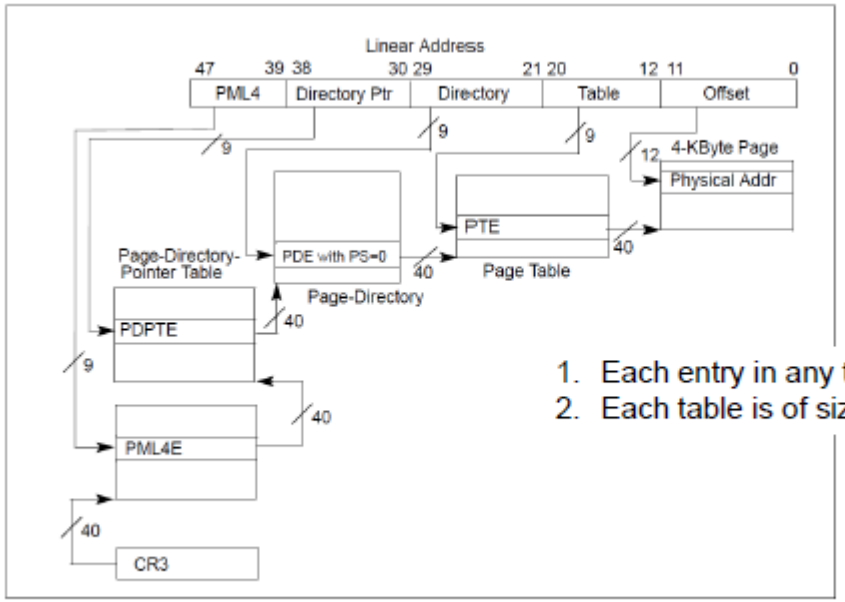
- With multiple references of a I-Node, it is hard to determine when to delete an I-Node
  - Maintain a I-Node reference count
  - Decremented for every deletion

Symbolic Link problems:

- As **only the pathname is stored**, the link can be easily invalidated:
  - File name changes, file deletion etc
- Involve a search to locate actual I-Node number of target file
  - Just like a normal file opening operation

File Operations (Unix)





1. Each entry in any table is 64-bit long.
2. Each table is of size 4Kbytes.

Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

Barrier solution

```
rendezvous

mutex.wait()
    count = count + 1
mutex.signal()

if count == n: barrier.signal()

barrier.wait()
barrier.signal()

critical point
```

Reusable barrier solution

```
# rendezvous

mutex.wait()
    count += 1
    if count == n:
        turnstile2.wait()       # lock the second
        turnstile.signal()      # unlock the first
mutex.signal()

turnstile.wait()                # first turnstile
turnstile.signal()

# critical point

mutex.wait()
    count -= 1
    if count == 0:
        turnstile.wait()         # lock the first
        turnstile2.signal()      # unlock the second
mutex.signal()

turnstile2.wait()                # second turnstile
turnstile2.signal()
```

Queue solution (leaders)

```
mutex.wait()
if followers > 0:
    followers--
    followerQueue.signal()
else:
    leaders++
    mutex.signal()
    leaderQueue.wait()

dance()
rendezvous.wait()
mutex.signal()
```

Queue solution (followers)

```
mutex.wait()
if leaders > 0:
    leaders--
    leaderQueue.signal()
else:
    followers++
    mutex.signal()
    followerQueue.wait()

dance()
rendezvous.signal()
```

FIFO barbershop solution (customer)

```
self.sem = Semaphore(0)
mutex.wait()
    if customers == n:
        mutex.signal()
        balk()
    customers += 1
    queue.append(self.sem)
mutex.signal()

customer.signal()
self.sem.wait()

# getHairCut()

customerDone.signal()
barberDone.wait()

mutex.wait()
    customers -= 1
mutex.signal()
```

And the code for barbers:

FIFO barbershop solution (barber)

```
customer.wait()
mutex.wait()
    sem = queue.pop(0)
mutex.signal()

sem.signal()

# cutHair()

customerDone.wait()
barberDone.signal()
```

Notice that the barber has to get `mutex` to access the queue.
This solution is in `sync_code/barber2.py` (see 3.2).

Santa problem solution (Santa)

```
santaSem.wait()
mutex.wait()
    if reindeer >= 9:
        prepareSleigh()
        reindeerSem.signal(9)
        reindeer -= 9
    else if elves == 3:
        helpElves()
mutex.signal()
```

Santa problem solution (reindeer)

```
mutex.wait()
    reindeer += 1
    if reindeer == 9:
        santaSem.signal()
mutex.signal()

reindeerSem.wait()
getHitched()
```

Santa problem solution (elves)

```
elfTex.wait()
mutex.wait()
    elves += 1
    if elves == 3:
        santaSem.signal()
    else
        elfTex.signal()
mutex.signal()

getHelp()

mutex.wait()
    elves -= 1
    if elves == 0:
       elfTex.signal()
mutex.signal()
```

Theoretical ext2 limits under Linux[12]

Block size:	1 KiB	2 KiB	4 KiB	8 KiB
max. file size:	16 GiB	256 GiB	2 TiB	2 TiB
max. filesystem size:	4 TiB	8 TiB	16 TiB	32 TiB

Then, the directory permission can be understood as:
Read = Can you read this list? (impact: ls, <tab> auto-completion)

Write = Can you change this list? (impact: create, rename, delete file/subdir). Note the interaction with the execute permission bit.

Execute = Can you use this directory as your working directory? (impact: cd).

Since modifying the directory entries (i.e. with write permission) requires us to set the current working directory, execute bit is needed for the write-related operations under the target directory.

Some interesting scenarios:
a. Directory permission is independent from the file permission. So, you still can modify a file under a "read only" directory if the file allows write.
b. If you want to allow outsider to access a particular deeply nested file, e.g. A/B/C/D/file, you only need execute bit on A, B, C, D directory (i.e. read permission is not needed). This is a great way to hide the content of the directory and only allow access to specific file given the full pathname.

2. **[File System Overhead]** Let us find out the overhead of FAT16 and ext2 file systems. To have a
meaningful comparison, we assume that there is a total of 2^n 1KB data blocks. Let us find out how much bookkeeping information is needed to manage these data blocks in the two file systems.
Express the overhead in terms of number of data blocks.
a. Overhead in FAT16: Give the size of the two copies of file allocation table.
b. Overhead in ext2 is a little more involved. For simplicity, we will ignore the super block and group descriptors overhead.

Below are some known restrictions:
• The two bitmaps (data block and inode) each occupies a single disk block.
• There are **184** inodes per block group.

Calculate the following:
i. Number of data block per block group.
ii. Size of the inode table per block group.
iii. Number of block groups in order to manage 2^n data blocks.
iv. Combine (i – iii) to give the total overhead.
c. Comment on the runtime overhead of the two file systems, i.e. how much memory space is needed to support file system operations during runtime.

a. Each FAT table is 2^{16} x 16bit (2 bytes) = 2^{17} bytes.
2 copies of FAT table = 2 x 2^{17} = 2^{18} bytes. This takes 2^{18} / 2^{10} = 2^8 disk blocks.

Note that due to the special codes (free, bad, eof etc), this setup actually handles less than 2^{16} data blocks. We ignore this fact for ease of comparison.

b.
i. Each bitmap is in a 1KB block→ 8K bits → 8K data blocks per block group.
ii. From restrictions, 184 inode x 128 bytes each / 1024 bytes data block = 23 disk blocks.
Note that 184 inodes per block group is an arbitrary number.
iii. From (i) 2^{16} blocks / 2^{13} per block group = 8 block groups.
iv. Overhead per block group = 2 blocks of bitmaps + 23 blocks of inodes = 25 blocks.
Total overhead = 8 x 25 blocks = 200 blocks.

c. FAT: The entire FAT table is in memory, i.e. 2^{17} bytes.
Ext2: Nothing is needed. Though for efficiency, a couple of the recently accessed I-Node may be cached. As a comparison, with the overhead of FAT, we can cache 2^{17}/ 128 bytes = 2^{10} I-node in memory.

---

2. [Page table structure]
Given the following information:

- Virtual Memory Address Space = 32 bits
- Physical memory = 512MB
- Page Size = 4 KB
- PTE Size = 4 bytes

Suppose there are 4 processes, each using 512MB virtual memory. Answer the following:

a. Direct paging is used. What is the total space needed for all page tables used?
b. 2-level paging is used. What is the total space needed for this scheme?
c. Inverted table is used. What is the total space needed? You can assume each inverted table entry takes 8 bytes.

a. Number of page table entries
= Total Memory / Page Size
= 2^{32} / 2^{12}
= 2^{20}
Size of page table
= # of page table entries * size of page table entries
= 2^{20} * 2^2
= 2^{22} (4 MB)
Each process has its own page table, so total space needed
= Page table size * # of processes
= 2^{22} * 2^2
= 2^{24} (16MB)

b. The page table will be split into smaller page tables, each fitting into a page.
Number of PTE per smaller page table
= Page Size / PTE size
= 2^{12} / 2^2
= 2^{10} entries per smaller page table
Total Number of page table entries (PTE)
= Total Memory / Page Size
= 2^{32} / 2^{12}
= 2^{20} page table entries (from prev)
Total number of smaller page tables
= Total number of PTE / Number of PTE per smaller page table
= 2^{20} / 2^{10}
= 2^{10} smaller page tables

Assuming that the same size of entries is used in the page directory,
Size of page directory
= Number of smaller page tables * Size of PTE
= 2^{10} * 2^2
= 2^{12} bytes

A process uses 512 MB of memory.
Number of pages used by process
= Memory used / Size of page
= 2^{29} / 2^{12} (512MB / 4 KB)
= 2^{17} pages
Number of smaller page tables used
= Number of pages used / entries per smaller page table
= 2^{17} / 2^{10}
= 2^7 (128) smaller page tables

Overhead for a single process is equal to 1 page directory + 128 page table portions
Overhead for a single process
= Size of page directory +
size of smaller page table (i.e, page size) * number of smaller page table
= 2^{12} + 2^{12} * 2^7
= 2^{12} + 2^{19}
= 4KB + 512 KB
= 516 KB
Total Overhead
= 516 KB * 4 processes
= 2064 KB

c. Number of physical frames is = 2^{29} (512 MB) / 2^{12} = 2^{17} frames
A single frame table is used system wide, overhead = 2^{17} * 8 B = 2^{20} = 1 MB