Locks

- Is a variable with two boolean states
  - 1 - (available/unlock/free)
  - 0 - (acquired/locked/held)
- Has two operations
  1. `acquire()`

```
boolean test_and_set(boolean *lock)
{
      boolean old = *lock;
      *lock = True;
      return old;
}
boolean lock;

void acquire(boolean *lock) {
       while(test_and_set(lock));
}
```

  2. `release()`

```
void release(boolean *lock) {
       *lock = false;
}
```

- Is put around critical section to ensure critical section executes as if it's a single atomic instruction

```
1  lock_t mutex; // some globally-allocated lock 'mutex'
2  ...
3  lock(&mutex);
4  balance = balance + 1;
5  unlock(&mutex);
```

- Can only be released by the thread that acquired it
- Is used to protect shared resource (e.g. from race condition in files and data structure) [2]

Semaphore

- Is an abstract data types suitable for synchronization problems [2]
- Has variable `count` that allows arbitrary resource count [1]
- Has two atomic operations
  1. (`wait/P/decrement`) - block until `count > 0` then decrement variable

```
wait(semaphore *s) {
        while (s->count == 0) ;
        s->count -= 1;
}
```

  2. (`signal/V/increment`) - increment `count`, unblock a waiting thread

```
signal(semaphore *s) {
        s->count += 1;
        ......                  //unblock one waiter
}
```

- Can be signaled by any thread [2]

Concurrency

- Is the ability of different parts or units of a program, algorithm, or problem to be executed out of order, without affecting the final outcome. [3]

Concurrency Error

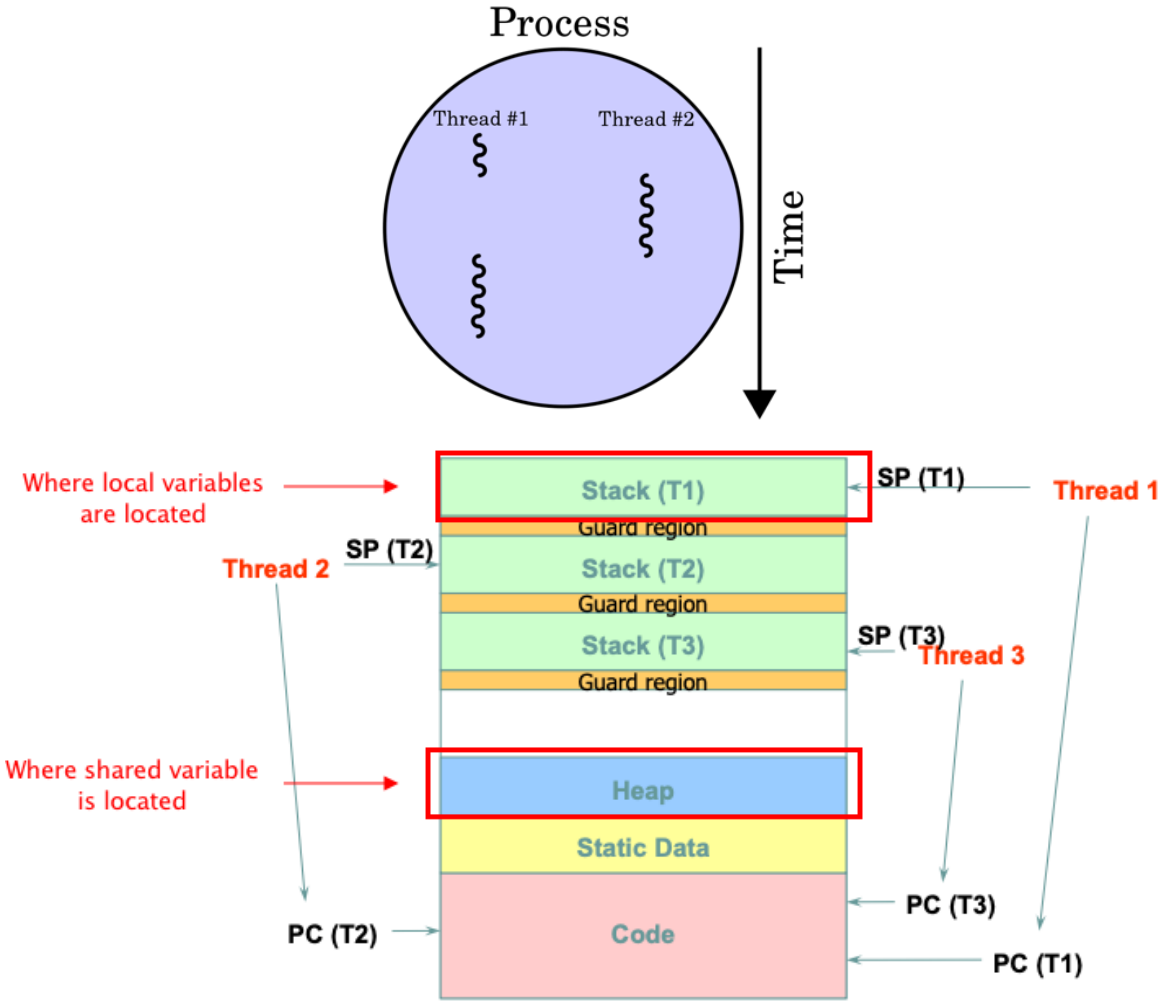
- Two types of concurrency errors [5]
  1. **Deadlock:** A situation wherein two or more processes are never able to proceed because each is waiting for the others to do something

     Key: Circular wait

2. **Race Condition:** a timing dependent error involving shared state
   - **Data Race:** Concurrent accesses to a shared variable and at least one access is a write
   - **Atomicity Bugs:** Code does not enforce the atomicity programmers intended for a group of memory access
   - **Order Bugs:** Code does not enforce the order programmers intended for a group of memory access

Thread

- Is the smallest sequence of programmed instructions that can be managed independently by a schdeduler [4]



- A thread is bound to a single process
- A process can have multiple threads

Virtualization of CPU

-

Limited Direct Execution

- Idea: Just run the program you want to run on the CPU, but first make sure to set up the hardware so as to limit what process can do without OS assistance
- baby proofs the CPU by

  1. Setting up trap handlers
  2. Starts an interrupt timer
  3. Run processes in a restricted mode

  Example

  Baby proofing a room:

  - Locking cabinets containing dangerous stuff and covering electrical sockets.
  - When room is readied, let your baby roam free in knowledge that all the dangerous aspect of the room is restricted

Trap Handlers

- Is instruction that tells the hardware what to run when certain exceptions occur

  Example

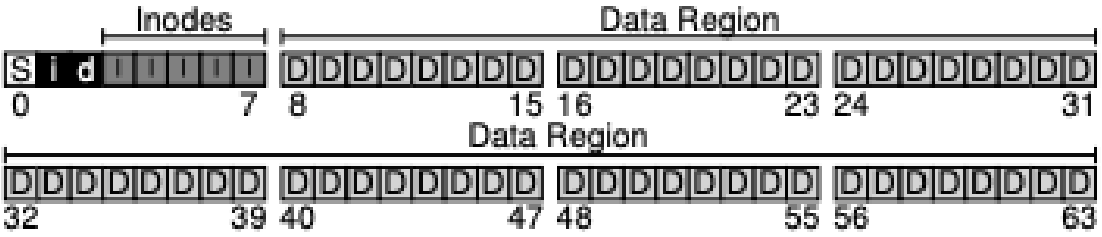
  What code to run when

  1. Hard disk interrupt occurs
  2. Keyboard interrupt occrs
  3. Program makes a system call?

Timer Interrupt

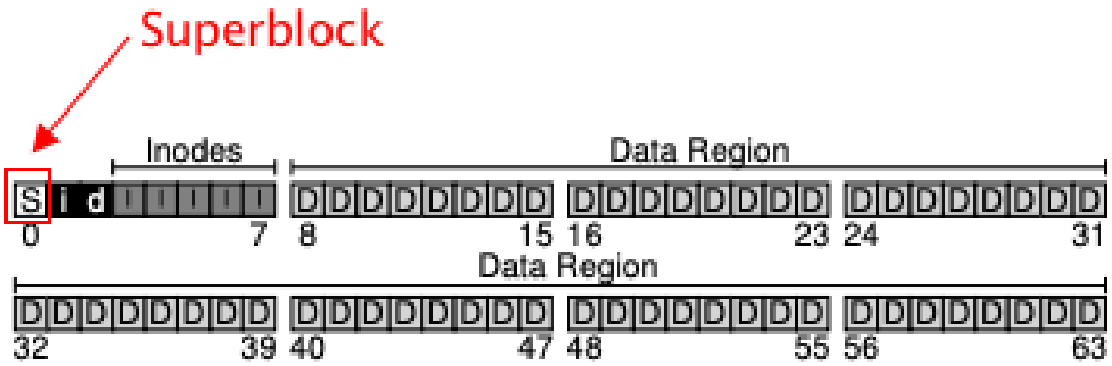
- Is a hardware mechanism that ensures the user program does not run forever
- Is emitted at regular intervals by a timer chip [6]

Index-based File System



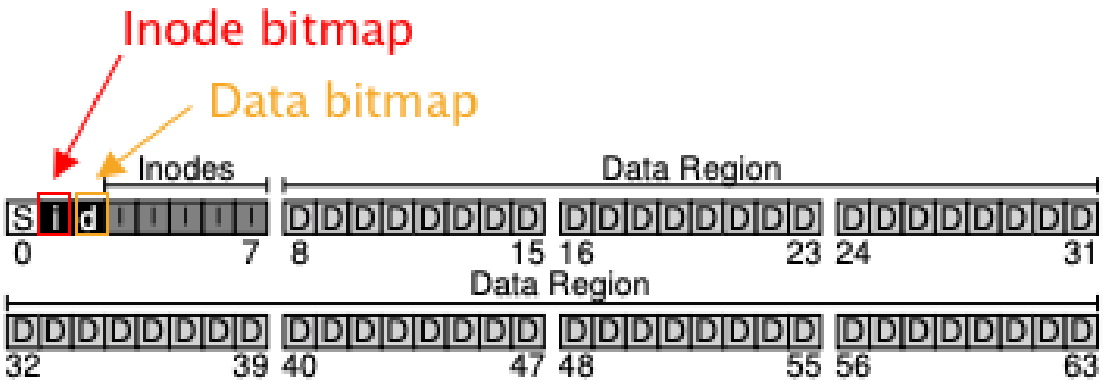
- Has following parts
  - Superblock
  - Inode Bitmap
  - Data Bitmap
  - Inodes
  - Data Region
- Each block in file system is 4KB
- Uses a large amount of metadata per file (especially for large files)

Superblock



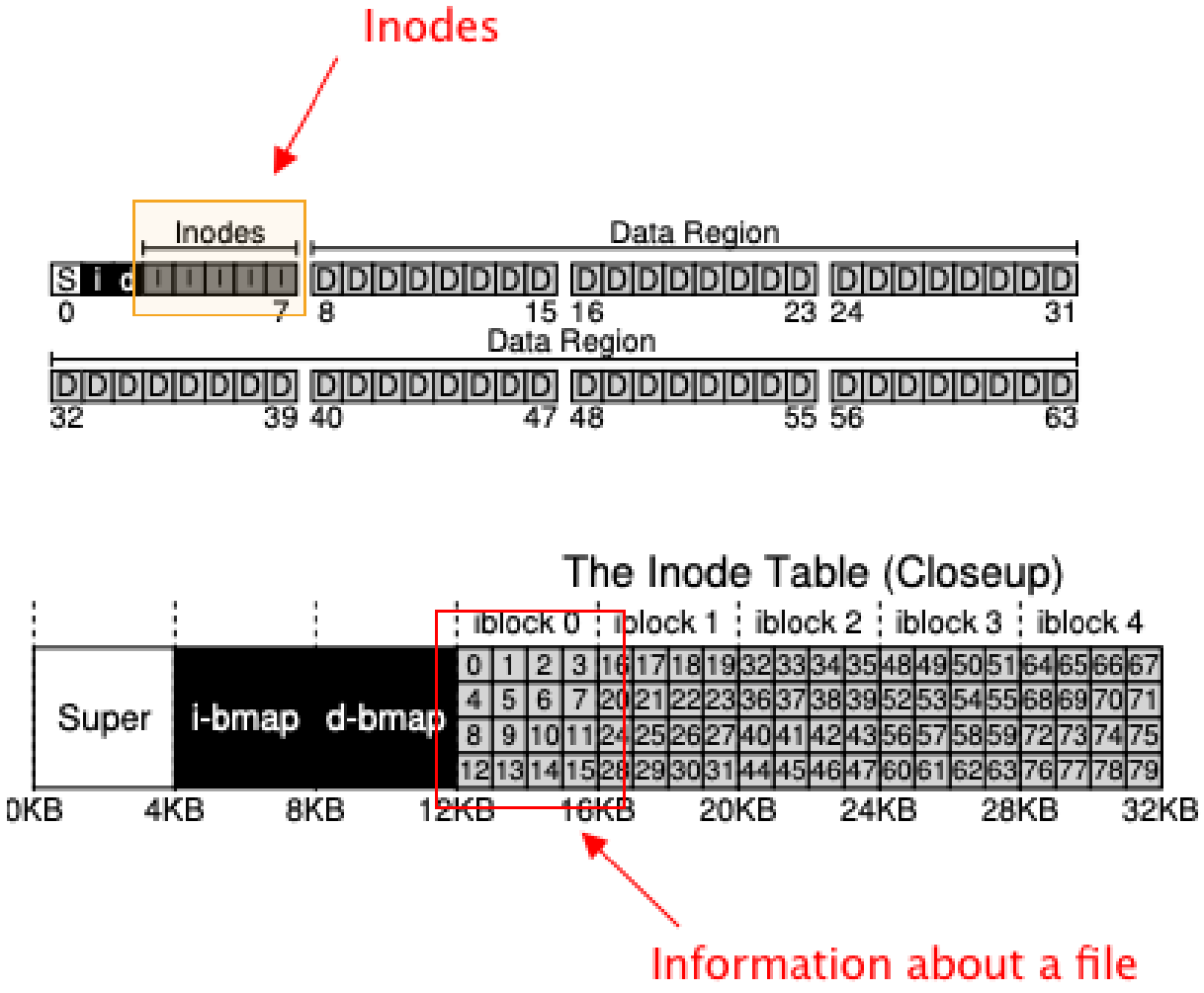
- contains information about the file system, including
  1. the number of inodes and data blocks in a particular file system
  2. the magic number of some kind to identify the file system type (e.g NFS, FFS, VSFS)
- The OS reads superblock first to initialize various parameters, and then attach volume to the file-system tree

Bitmap



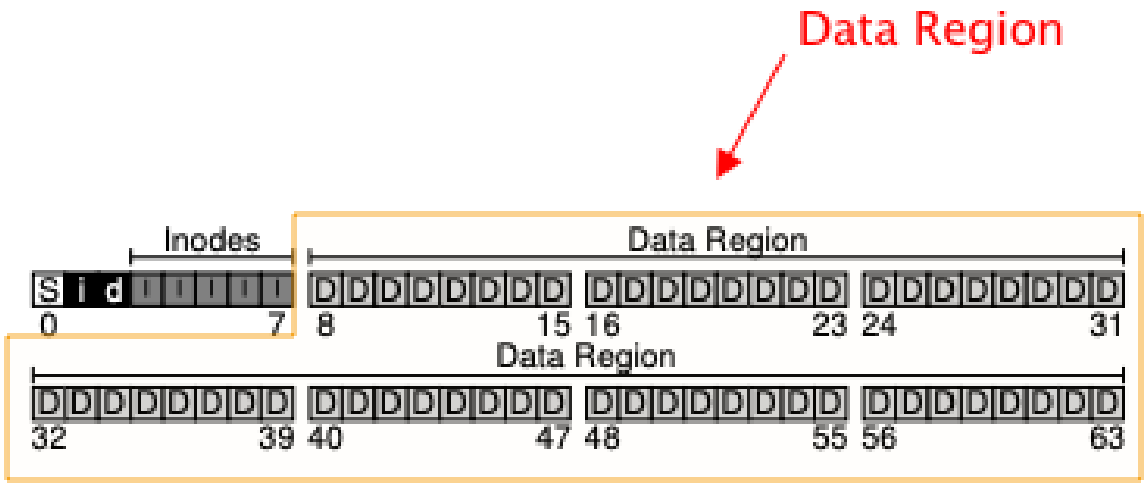
- Tracks whether inode or data blocks are free or allocated
- Is a simple and popuar structure
- Uses each bit
  - `0` means free
  - `1` means in use
- **Data Bitmap** is bitmap for data region
- **Inode Bitmap** is bitmap for inode region

Inode



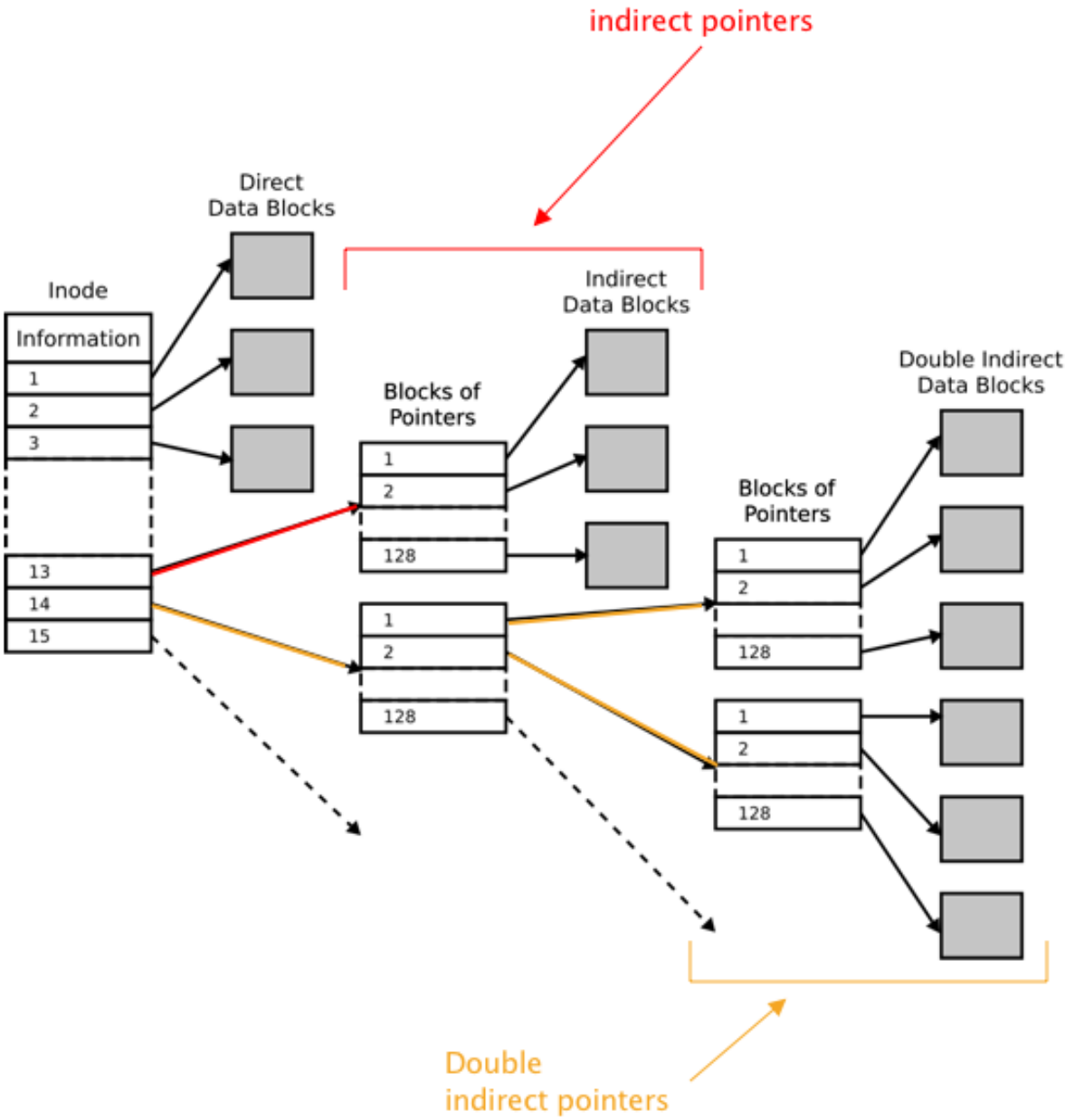
- Is a short form for **index node**
- Contains disk block location of the object's data [7]
- Contains all the information you need about a file (i.e. metadata)
  - File Type
    - e.g. regular file, directory, etc
  - Size
  - Number of blocks allocated to it
  - Protection information
    - such as who owns the file, as well as who can access it
  - Time information
    - e.g. When file was created, modified, or last accessed
  - Location of data blocks reside on disk

Data Region



- Is the region of disk we use for user data

Multi-Level Index



- Is for supporting bigger files
- Uses **indirect pointer** in inode that points to more pointers
- Uses **double indirect pointer** for even larger files
  - is a pointer in inode that points to a block that contains pointers to indirect blocks

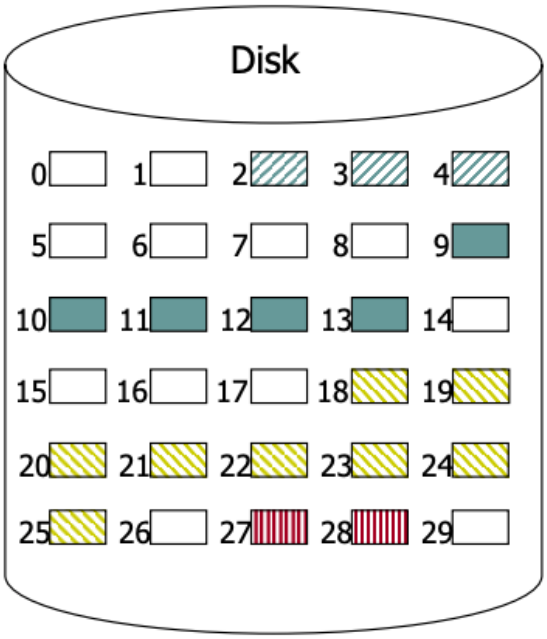
External Fragmentation

- Is various free holes that are generated in either your memory or disk space. [8]
- Are available for allocation, but may be too small to be of any use [8]

Internal Fragmentation

- Is wasted space within each allocated block [8]
- Occurs when more computer memory is allocated than is needed

sectionExtent Based File System



directory

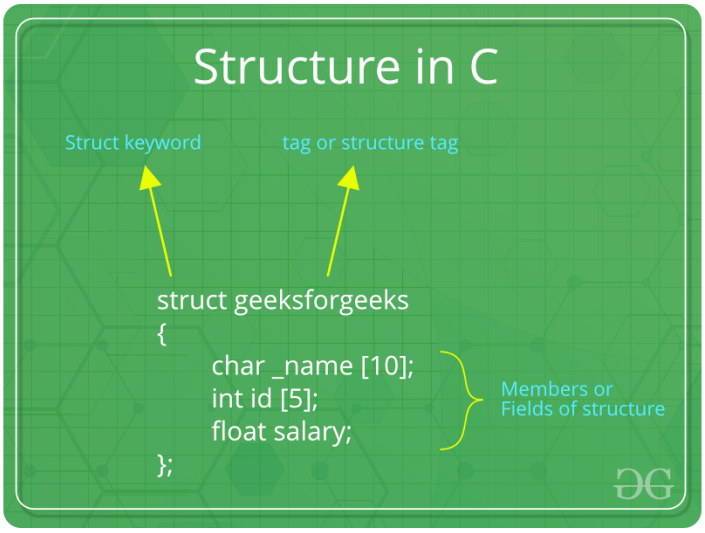
File Name	Start Blk	Length
File A	2	3
File B	9	5
File C	18	8
File D	27	2

- Is simply a disk pointer plus a length (in blocks)
  - Together, is called **extent**
- Often allows more than one extent
  - resolve problem of finding continuous free blocks
- Is less flexible but more compact
- Works well when there is enough free space on the disk and files can be laid out contiguously

Example

Linux's ext4 file system
sectionFields

- Is the members in a structure



sectionProcess List

- Is a data structure in kernel or OS
- Contains information about all the processes running in the system

sectionProcess Control Block

- Is a data structure in kernel or OS
- Contains all information about a process
- Is where the OS keeps all of a process' hardware execution state
- Generally includes

1. Process state (ready, running, blocked)
2. Process number
3. Program counter: address of the next instruction
4. CPU Registers: is saved at an interrupt
5. CPU scheduling information: process priority
6. Memory management info: page tables
7. I/O status information: list of open files

sectionContext Switch

- is the process of storing the state of a process or thread, so it can be restored and resume execution at a later point [10]
- happens during a timer interrupt or system call
- process context switch needs to save the following [11]
  - Process Number
  - Process State
  - Process Counter,
  - CPU Registers
  - Memory management info
  - I/O status Information
- thread context switch needs to save the following
  - Process Counter,
  - CPU Registers
- May hinder performance

Virtualization

- is an act of creating an illusion that there are as many hardware resource as each program needs.

Virtualizing CPU

- Turns a single CPU into a seemingly infinite number of CPUS, and allows many programs to seemingly run at once
- To implement CPU virtualization, the OS needs low-level machinery called **mechanism** and high level intelligence called **policies**
- Steps
  1. Involve OS to setup hardware hardwre to limit what the process can do without OS assistance (**Limited Direct Execution**)

     This is done so by

     1. Setting up trap handler

    2. Starting an interrupt timer (so process won't last forever)
2. Involve OS to intervene at key points to perform previleged operations or switch out operations when they have monopolized the CPU too long

Limited direct execution

- Is a mechanism of CPU virtualization
- Baby proofs the CPU by setting up hardwre to limit what the process can do without OS assistance
- Runs program in CPU once baby proofed
- Does so by
  - Setting up trap handler
  - Starting an interrupt timer (so process won't last forever)

Scheduling policies

- Are series of polices of CPU virtualization
  - **First In First Out**
  - **Shortest Job First**
  - **Shortest Time-to-completion First**
  - **Round Robin**
  - **Multi-level Feedback Queue**

Virtualizing Memory

- Basic Idea: For the most part, let the program run directly on the hardware; however, at certain key points in time (e.g. system call, timer interrupt), arrange so that the OS gets involved and make sure the 'Right' thing happens.
- Like CPU, many programs are sharing the memory at the same time
- Like CPU, the goal is to create an illusion that it has its own code and data
- Like CPU, the memory also needs low-level machinery called **mechanism**, and high level intelligence called **policies**
- Steps

1. Use **address translation** to transform each memory access, changing **virtual address** provided by instruction to **physical address**
   - Memory access includes instruction fetch, load, or store
   - Is done using hardware
2. Invove OS at key points to **manage memory**

   Memory management includes

   1. Setting up hardware so correct translations take place
   2. Keeping track of which locations are free and which are in use
   3. Judiciously intervening to maintain control over how memory is used

Address Translation

- Is also called **hardware-based address translation**
- Is a mechanism of memory virtualization
- Is the technique of transforming virtual address to physical address

Dynamic (Hardware-based) Relocation

- Is a type of address translation
- Occured in 1950's
- Is also referred as **base and bounds**
- Need **base register** and the **bounds**
  - **base register** transforms virtual address to physical address
  - **bound** ensures addresses are within confines of the address space
- Steps
  1. Write and compile program as if it is loaded at address zero
  2. Allow OS to decide where in physical memory it should be loaded and set base register to that value

     ```
     physical address = virtual address + base
     ```

Block

- Size of each block is 4KB

lseek

- **Syntax:** `off_t lseek(int fildes, off_t offset, int whence)`
  - `fildes` - file descriptor
  - `offset` - file offset to a particular position in file

Kilobyte

- 1 kilobyte is 1024 bytes

file

- is an array of bytes which can be created, read, written and deleted
- low-level name is called **inode number** or **i-number**

Reading a File From Disk

Example

When

```
open("/foo/bar", O_READONLY)
```

is called

- the goal is to find the inode of the file `bar` to read its basic information (i.e. includes permission, information, file size etc)
- done by traversing the pathname and locate the desired inode
- Steps
  1. Begin traversal at the root of the file system, in the **root directory**
  2. Find **inode** of the root directory by looking for `i-number`
     - **i-number** is found in it's parent directoy
     - for root directory, there is no parent directory

   - it's inode number is 2 (for UNIX file systems)
3. Read the **inode** of root directory
4. Once its **inode** is read, look inside to find pointers to data blocks
5. Recursively traverse the pathname until the desired inode is found (e.g `foo` → `bar`)
6. Issue a `read()` system call to read from file
   - `fd` with offset `0` reads the first file block (e.g. `bar data[0]`)
   - `lseek(..., offset_amt * size_of_file_block)` is used to offset/move to desired block in `bar`
7. Trasnfer data to `buf` data block
8. Close `fd`. No I/O is read.

inode

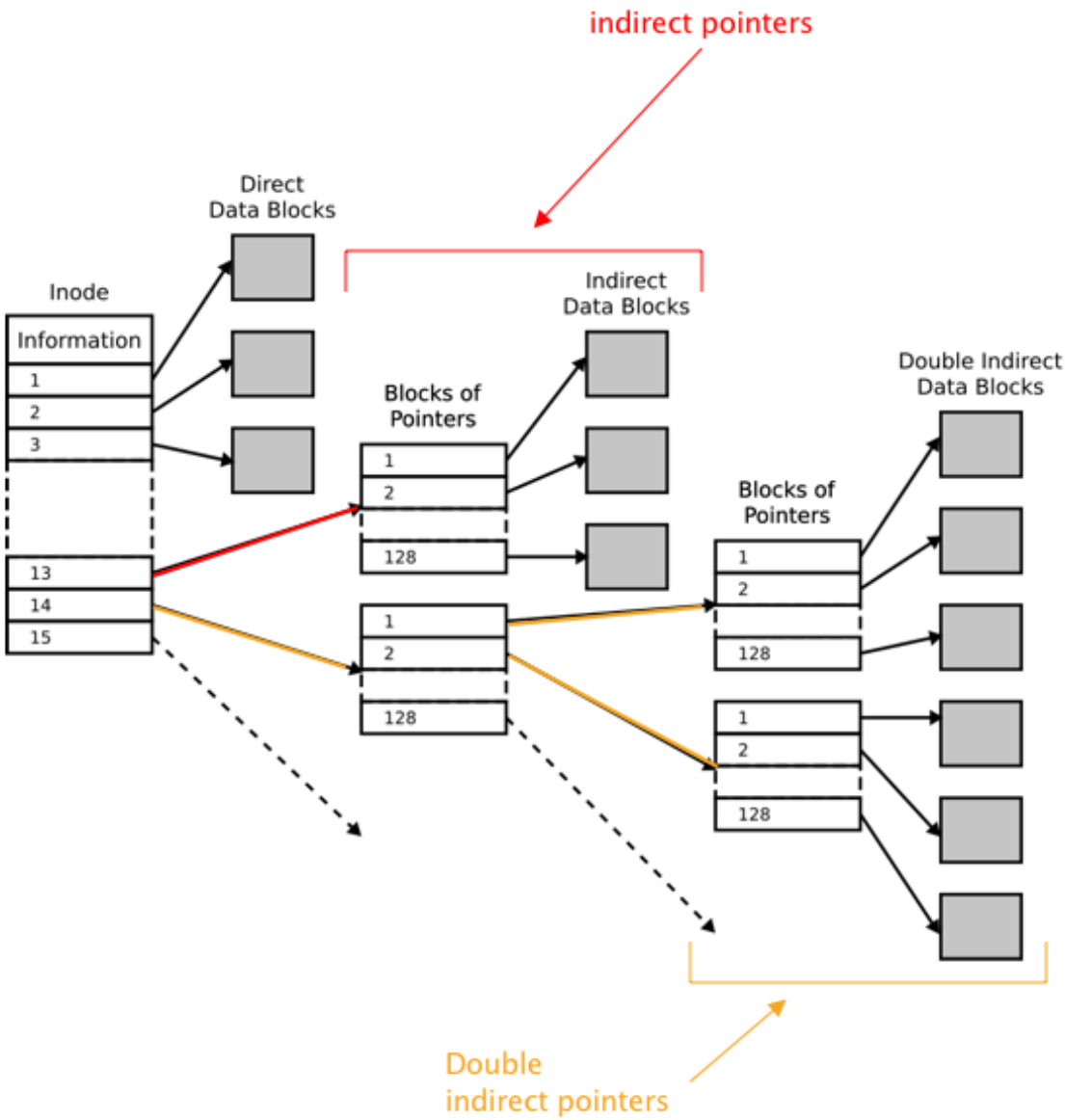
- total size may vary
- inode pointer has size of 4 byte
- Has 12 **direct pointers** to 4KB data blocks
- Has 1 **indirect pointer** [when file grows large enough]
- Has 1 **double indirect Pointer** [when file grows large enough]
- Has 1 **triple indirect Pointer** [when file grows large enough]

Indirect Pointers

- Is allocated to data-block if file grows large enough
- Has total size of 4 KB or 4096 bytes
- Has $4096/4 = 1024$ pointers
- Each pointer points to 4KB data-block
- File can grow to be $(12 + 1024) \times 4\text{K} = 4144\text{KB}$

Double Indirect Pointers

- is allocated when single indirect pointer is not large enough
- each pointer in first pointer block points to another pointer block
- has 1024^2 pointers
- each of 1024^2 pointers point to 4KB data block
- File can grow to be $(12 + 1024 + 1024^2) \times 4\text{K} = 4198448\text{KB}$ or $\approx 4.20\text{GB}$



Triple Indirect Pointers

- is allocated when double indirect pointer is not large enough
- has 1024^3 pointers
- each of 1024^3 pointers point to 4KB data block
- File can grow to be $(12 + 1024 + 1024^2 + 1024^3) \times 4\text{K} = 4299165744\text{KB}$ or $\approx 4.00\text{TB}$

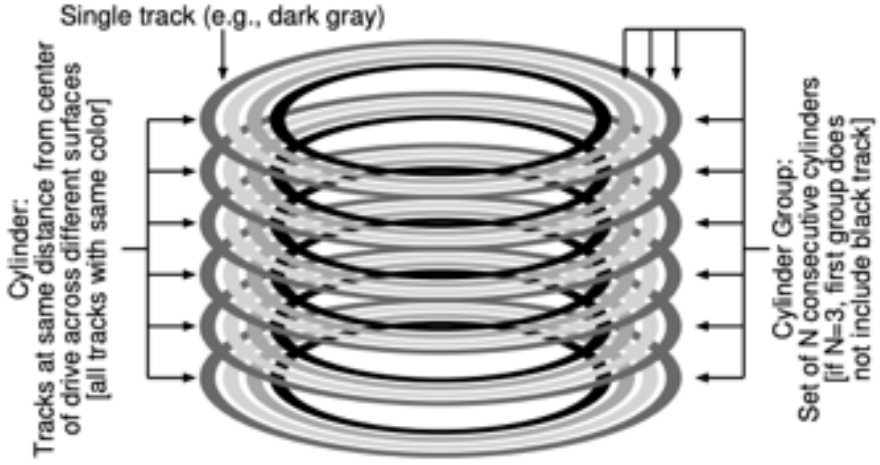
Old UNIX File system

- was simple, and looked like the following on disk

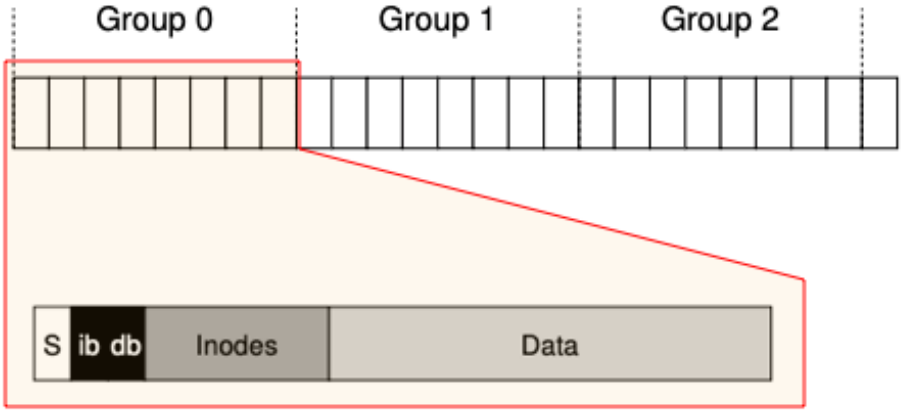
- has terrible performance
- suffers from **external fragmentation**
- had small data block (512 bytes) and transfer of data took too long

Fast File System

- modern file system has same APIS (`read()`, `write()`, `open()`, `close()`)
- divides into a number of **cylinder groups**



- each **block group** or **cylinder group** is consecutive portion of disk's address



Bitmap

- Are excellent way to manage free space
- tracks whether inodes/data block of the group are allocated

FFS Policies: Allocating Files and Directories

- Basic Idea: keep related stuff together, and keep related stuff far apart
- Directories Step

  1) Find the **cylinder group** with a low number of allocated directories and a high number of free inodes
     - low number of allocated directories → to balance directories across groups
     - high number of free nodes → to subsequently be able to allocate a bunch offiles
  2) Put directory data and inode to the **cylinder group**

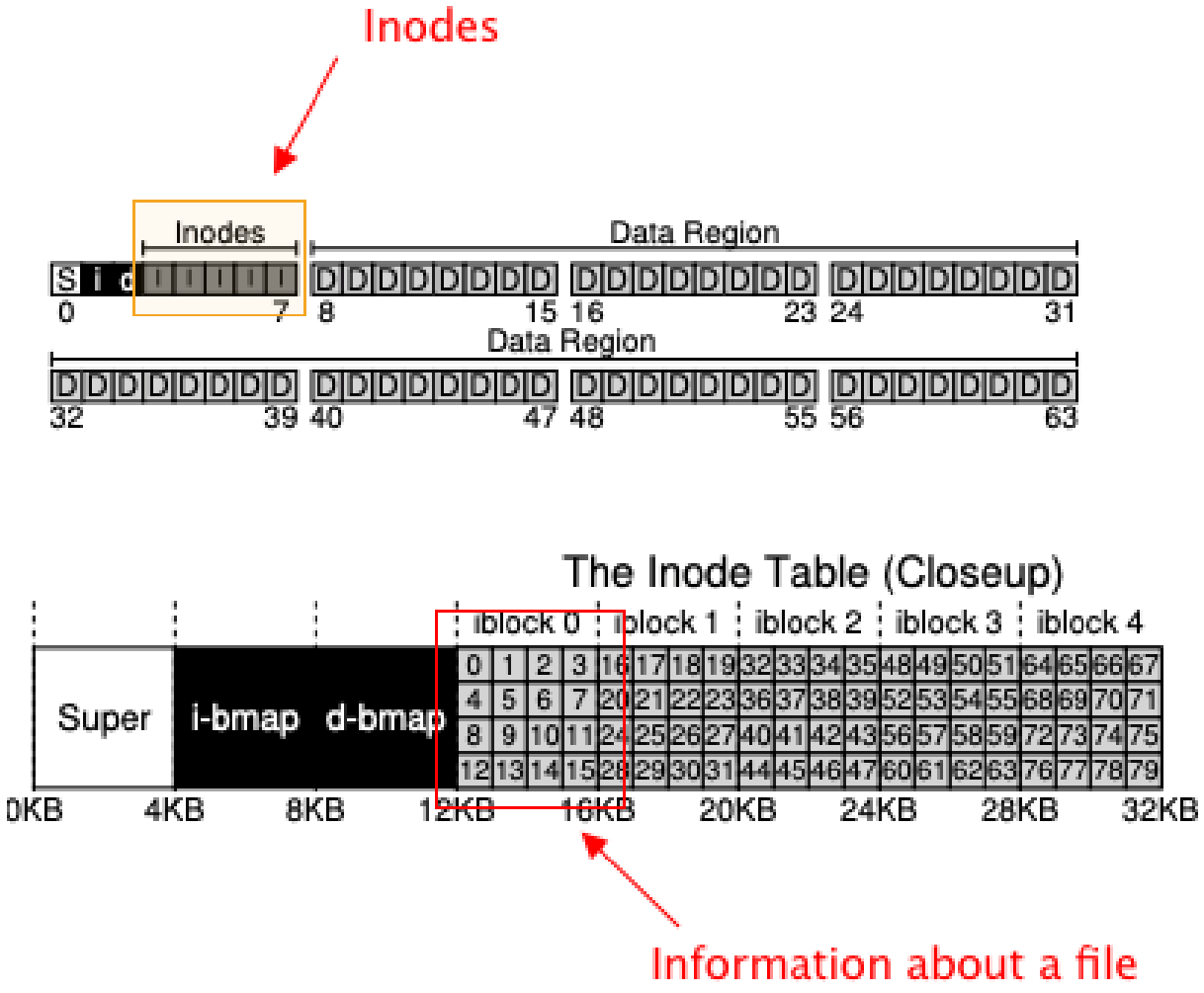
- Files Step

  1) Allocate the data blocks of a file in the same **cylinder group** as its inode
  2) Place all files in the same directory in the cylinder group of the directory they are in

     Example

     On putting `/a/c`, `/a/d`, `/b/f`, FFS would place
     - `/a/c`, `/a/d` as close as possible in the same **cylinder group**,
     - `/b/f` located far away (in some other **cylinder group**)

Inode

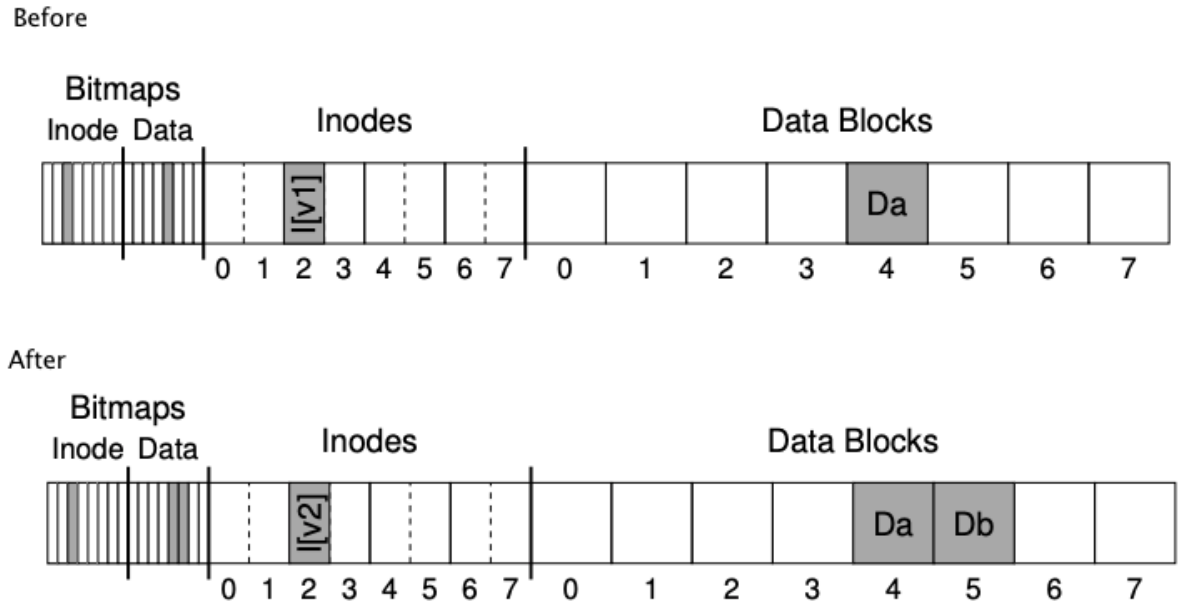


- Is a short form for **index node**
- Contains disk block location of the object's data [1]
- Contains all the information you need about a file (i.e. metadata)
  - File Type
    - e.g. regular file, directory, etc
  - Size
  - Number of blocks allocated to it
  - Protection information
    - such as who owns the file, as well as who can access it
  - Time information
    - e.g. When file was created, modified, or last accessed
  - Location of data blocks reside on disk

Crash Consistency

- Goal: How to update persistent data structures despite the presence of a **power loss** or **system crash**?

Crash Scenarios



1) Just the data block (Db) is written to disk

- No inode that points to it
- No bitmap that says the block is allocated
- It is as if the write never occured
- There is no problem here. All is well. (In file system's point of view)

2) Just the updated inode (I[v2]) is written to disk

- Inode points to the disk where `Db` is about to be written
- No bitmap that says the block is allocated
- No `Db` is written
- Garbage data will be read
- Also creates **File-system Inconsistency**

  - Caused by on-disk bitmap telling us Db 5 is not allocated, but inode saying it does

3) Just the updated bitmap (B[v2]) is written to disk

- Bitmap indicates tht block 5 is allocated
- No inode exists at block 5
- Creates **file-system inconsistency**
- Creates **space-leak** if left as is
  - block 5 can never be used by the file system

4) Inode (I[v2]) and bitmap (B[v2]) are written to disk, and not data

- File system metadata is completely consistent (in perspective of file system)
- Garbage data will be read

5) Inode (I[v2]) and data block (Db) are written, but not the bit map

- Creates **file-system inconsistency**
- Needs to be resolved before using file system again

6) Bitmap (B[v2]) and data block (Db) are written, but not the inode (I[v2])

- Creates **file-system inconsistency** between inode and data bitmap
- Creates **space-leak** if left as is
  - Inode block is lost for future use
- Creates **data-leak** if left as is
  - Data block is lost for future use

References

1) Wikipedia, Semaphore (programming), link

2) Stack Overflow, Difference between binary semaphore and mutex, link

3) Wikipedia, Concurrency (computer science), link

4) Wikipedia, Thread, link

5) Columbia University, Concurrency Errors, link

6) Wikibooks, Operating System Design/Processes/Interrupt, link

7) Wikipedia: inode, link

8) Washington University, Explain the difference between external fragmentation and internal fragmentation, link

9) Wikipedia, Inode pointer structure, link

10) Wikipedia: Context switch, link

11) University of Washington, CSE 451, Spring 2000 Solutions to Homework 2, link