1 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

2 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

3 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?

4 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]

5 Response Time

- **Formula** $T_{response} = T_{firstrun} - T_{arrival}$
- measures the interactive performance between users and the system

6 Turnaround Time

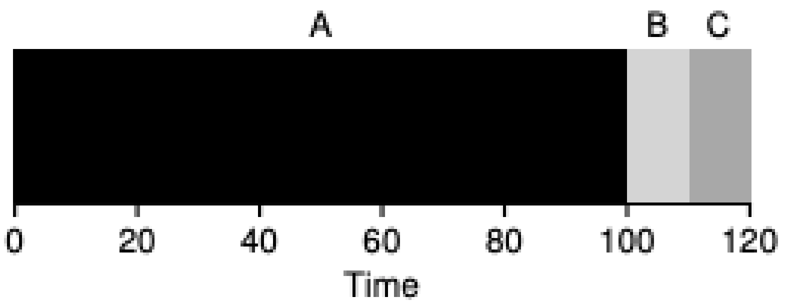
- **Formula** $T_{turnaround} = T_{completion} - T_{arrival}$
- measures the amount of time taken to complete a process

7 Starvation

- Is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time [1]

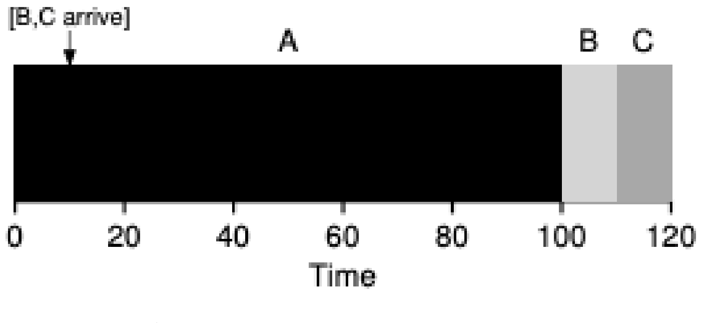
8 Convoy Effect

- Is the problem where number of relatively-short potential consumers of a resource get queued behind a heavy weight consumer

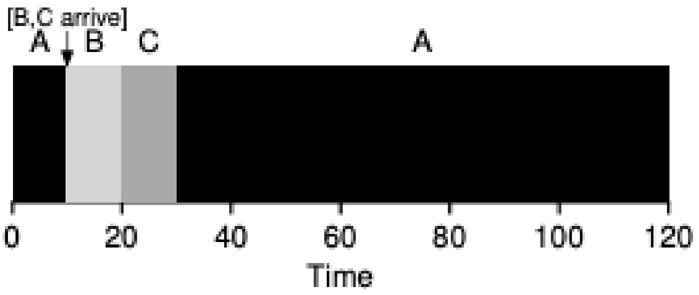


9 Scheduling policies

- Are algorithms for allocating CPU resources to concurrent tasks deployed on (i.e., allocated to) a processor (i.e., computing resource) or a shared pool of processors [5]
- Are sometimes called **Discipline**
- Covers the following algorithms in textbook
  - **First In First Out**
    - Is the most basic scheduling algorithm
    - Is vulnerable to **convoy effect**
    - No **starvation** as long as every process eventually completes
  - **Shortest Job First**
    - Improves average **turnaround time** given processes of uneven length
    - Is a general scheduling principle useful in situation where turnaround time per process matters
    - Is vulnerable to **convoy effect**



    - Is vulnerable to **starvation**
      - When only short-term jobs come in while a long term job is in queue
  - **Shortest Time-to-completion First**
    - Addresses **convoy effect** in **Shortest Job First**
    - Determines which of the remaining+new jobs has least time left, and schedule accordingly at any time

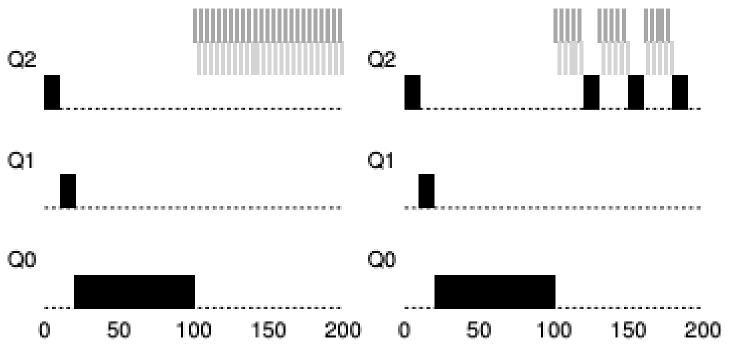


- Is vulnerable to **starvation**
  - When only short-term jobs come in while a long term job is in queue

– **Round Robin**
- Has good **response time** but terrible **turnaround time**
- Runs job for a **time slice** or **quantum**
- Each job gets equal share of CPU time
- Is clock-driven [6]
- Is starvation-free [7]
- Must have the length of a time slice (**quantum**) as multiple of timer-interrupt period

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

– **Multi-level Feedback Queue**
- Is the most well known approaches to shceduling
- Optimizes **turnaround time**, and minimizes **response time**
- Observees the execution of a job and priortizes accordingly without prior knowledge
- Rules
  - **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't)
  - **Rule 2:** If Priority(A) = Priority(B). A & B run in round-robin fashion using the time slice (quantum length) of the given queue
  - **Rule 3:** When a job enters the system, it is placed at the highest priority(the top most queue)
  - **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (it moves down on queue)
  - **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.



10 User Mode

- Is restricted
- Executing code has no ability to *directly* access hardware or reference memory [1]
- Crashes are always recoverable [1]
- Is where most of the code on our computer / applications are executed [3]

11 Kernel Mode

- Is previleged (non-restricted)
- Executing code has complete and unrestricted access to the underlying hardware [3]
- Is generally reserved for the lowest-level, most trusted functions of the operating system [1]
- Is fatal to crash; it will halt the entire PC (i.e the blue screen of death) [3]

12 Interrupt

- i is a signals are sent by hardware (keyboardm mouse, etc.), or software (page fault, protection violation, system call)
- Tells the CPU to stop its current activities and execute the appropriate part of the operating system (**Interrupt Handler**). [2]
- Has three different types [2]
  1) **Hardware Interupts**
     - Are generated by hardware devices to signal that they need some attention from the OS.

  - May be due to receiving some data

    Examples

    * Keystrokes on the keyboard
    * Receiving data on the ethernet card

  - May be due to completing a task which the operating system previous requested

    Examples

    Transfering data between the hard drive and memory

2) **Software Interupts**

  - Are generated by programs when a system call is requested

3) **Traps**

  - Are generated by the CPU itself
  - Indicate that some error or condition occured for which assistance from the operating system is needed

13 Content Switch

- Is switching from running a user level process to the OS kernel and often to other user processes before the current process is resumed
- Happens during a timer interrupt or system call
- Saves the following states for a process during a context switch
  - Stack Pointer
  - Program Counter
  - User Registers
  - Kernel State
- May hinder performance

14 System Call

- Is the programmatic way in which a computer program requests a previleged service from the kernel of the operating system
- i.e. Reading from disk
- Is strictly a subset of software interrupts
- Steps
    1) Setup **trap tables** on boot
    2) Execute system call
    3) Save *Program Counter*, *CPU registers*, *kernal stack* (so process can resume after **return-from-trap** or **context switch**)
    4) Switch from **user mode** to **kernel mode**
    5) Perform previleged operations
    6) Finish and execute **return-from-trap** instruction
    7) Return from **kernel mode** to **user mode** and resume user program

Example

- `yield()`
    - Is a system call
    - Causes the calling thread to relinquish the CPU
    - Places the current thread at the end of the run queue
    - Schedules another thread to run

15 Signals

- Provides a way to communicate with the process
- Can cause job to stop, continue, or terminate
- Can be delivered to an application
    - Stops the application from whatever its doing
    - Runs Signal handler (some code in application to handle the signal)
    - When finished, the process resumes previous behavior

16 CPU-bound process

[8]

- CPU Bound processes are ones that are implementing algorithms with a large number of calculations
- Programs such as simulations may be CPU bound for most of the life of the process.
- Users do not typically expect an immediate response from the computer when running CPU bound programs.
- They should be given a lower priority by the scheduler.

17 I/O-bound process

[8]

- Processes that are mostly waiting for the completion of input or output (I/O) are I/O Bound.
- Interactive processes, such as office applications are mostly I/O bound the entire life of the process. Some processes may be I/O bound for only a few short periods of time.
- The expected short run time of I/O bound processes means that they will not stay the running the process for very long.
- They should be given high priority by the scheduler.

18 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

19 Address Translation

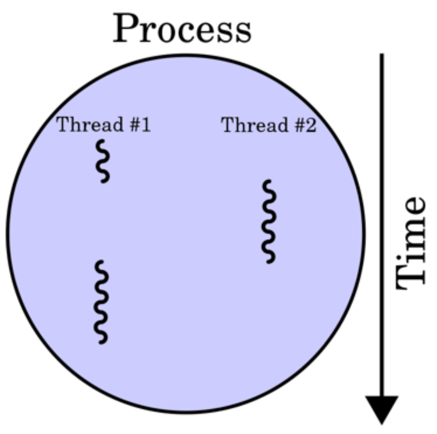
- Is also called **hardware-based address translation**
- Is a mechanism of memory virtualization
- Is the technique of transforming virtual address to physical address

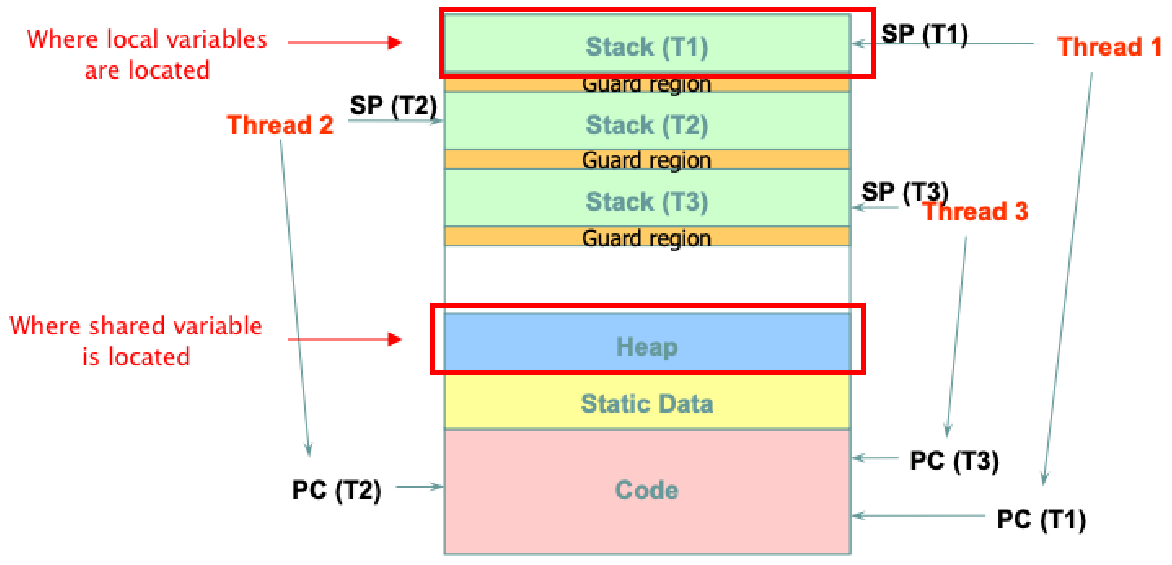
20 Critical Section

- Is a piece of code that accesses a *shared* resource, usually a variable or data structure

21 Thread

- Is a lightweight process that can be managed independently by a schdeduler [4]
- Improves the application performance using parallelism. (e.g peach)





- A thread is bound to a single process
- A process can have multiple threads
- Has two types
  - **User-level Threads:**

    * Are implemented by users and kernel is not aware of the existence of these threads
    * Are represented by a program counter(PC), stack, registers and a small process control block
    * Are small and much faster than kernel level threads
  - **Kernel-level Threads:**
    * Are handled by the operating system directly
    * Thread management is done by the kernel
    * Are slower than user-level threads

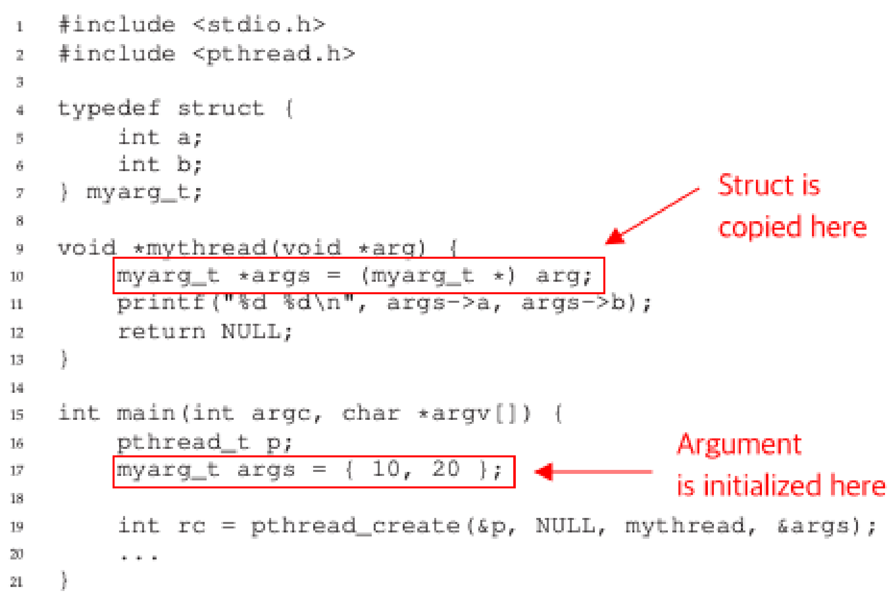
22 Thread API

- `pthread_create`
  - **syntax:**

```c
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void * (*start_routine)(void*),
                   void * arg)
```

    * `thread`
      · is a pointer to a structure of type `pthread_t`
    * `attr`
      · is used to specify any attributes this thread might have
      · is initialized with a separate call `pthread_attr_init()`
      · set default by passing `NULL`
    * `(start_routine)`
      · means which function this thread should start running in?
      · setting void pointer `(void *)` as an argument to function `start_routine` allows us to pass in <u>any</u> type of argument
      · setting void pointer `(void *)` as return type allows us to return <u>any</u> type of result
    * `args`
      · is where to pass the arguments for the function pointer ((start_routine))

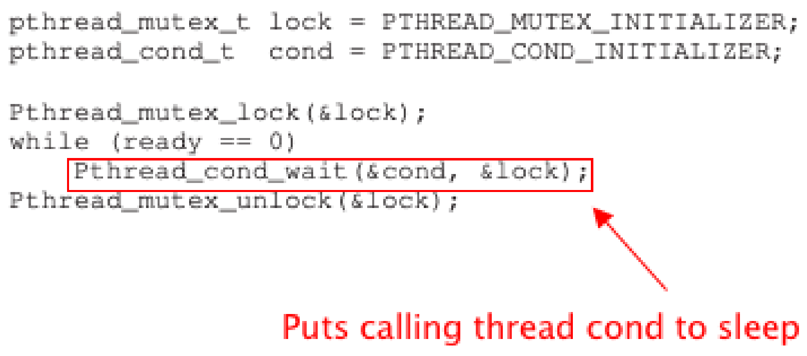
    <u>**Example**</u>



```c
#include <stdio.h>
#include <pthread.h>

typedef struct {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf("%d %d\n", args->a, args->b);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    myarg_t args = { 10, 20 };

    int rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

- `pthread_cond_wait`
  - Puts the calling thread to sleep (a blocked state)
  - Waits for some other thread to signal it

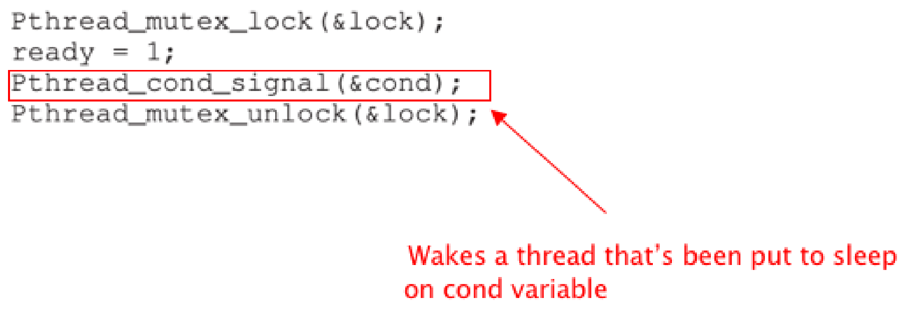
Example



```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

- `pthread_cond_signal`
  - Is used to unblocks at least one of the threads that are blocked on the specified condition variable cond

Example



```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

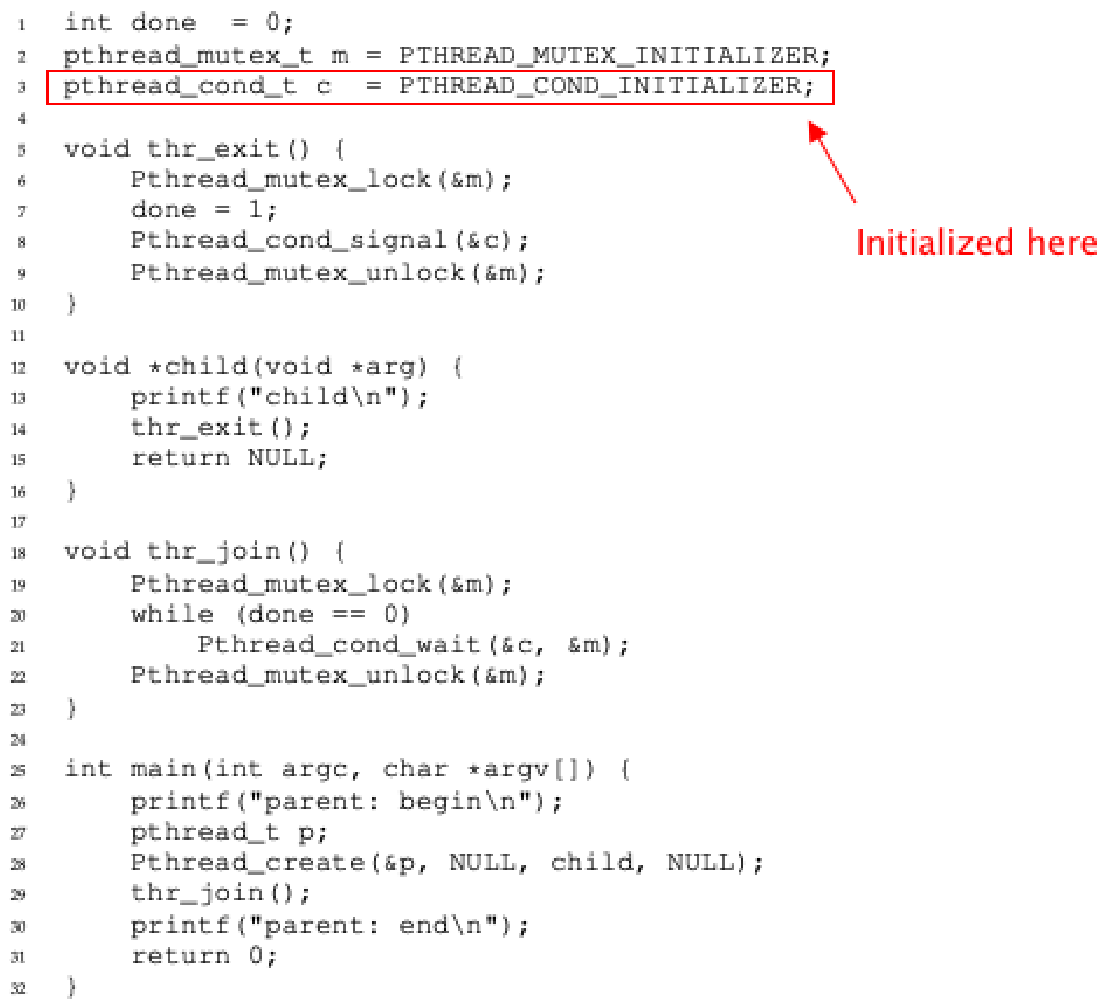
Wakes a thread that's been put to sleep on cond variable

23 Condition Variable

- is an explicit queue that threads can put themselves on when some state of execution is not as desired (so it can be put to sleep)
- when states are changed, one or more of the waiting threads can be awaken and be allowed to continue (done by **signaling** the condition)
- queue is **FIFO**
- `wait()` call is used to put thread to sleep
- `singal()` call is used to awake thread from sleep
- **Syntax (initialization):**

  `pthread_cont_t c = PTHREAD_COND_INITIALIZER`

  Example

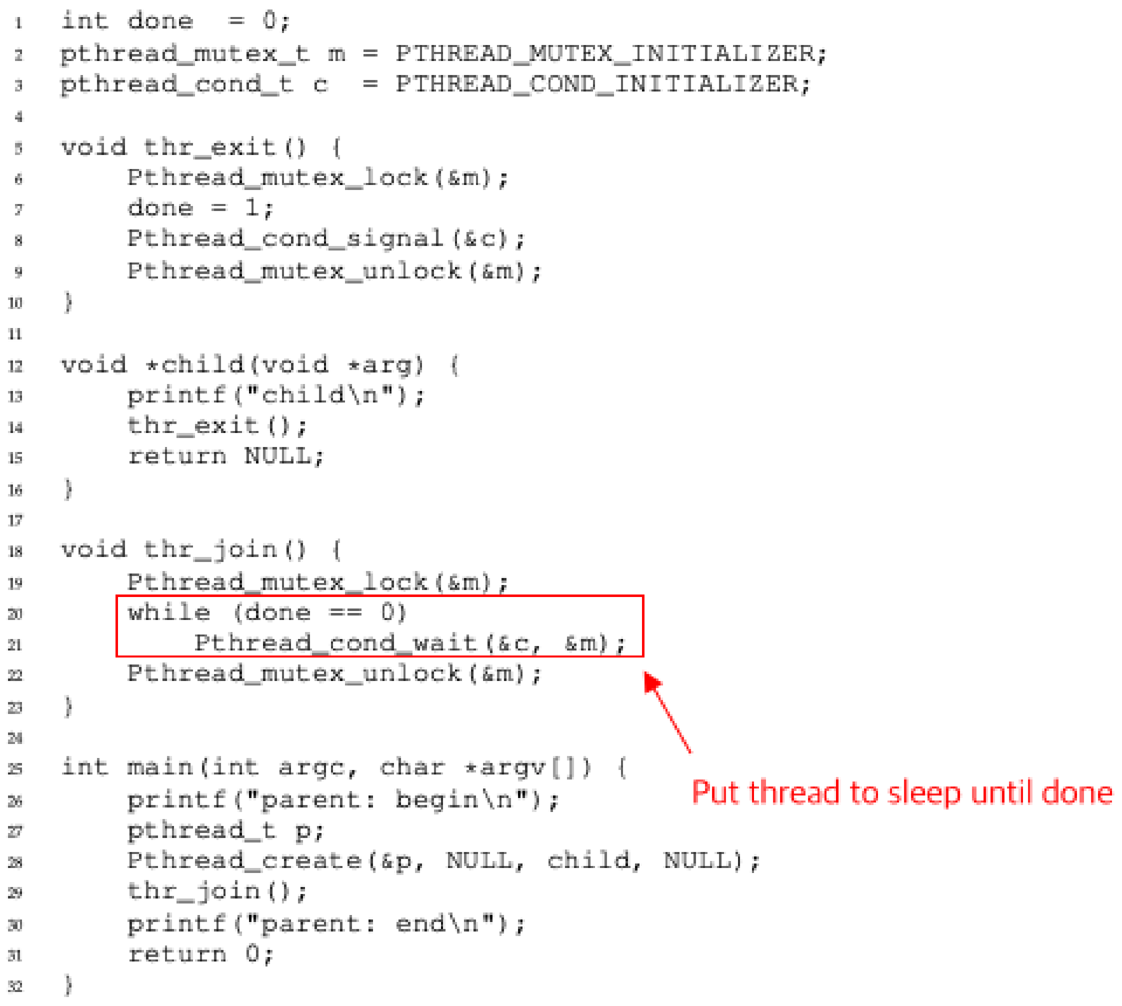


```
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

- **Syntax (Wait):**

  `Pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`

  Example

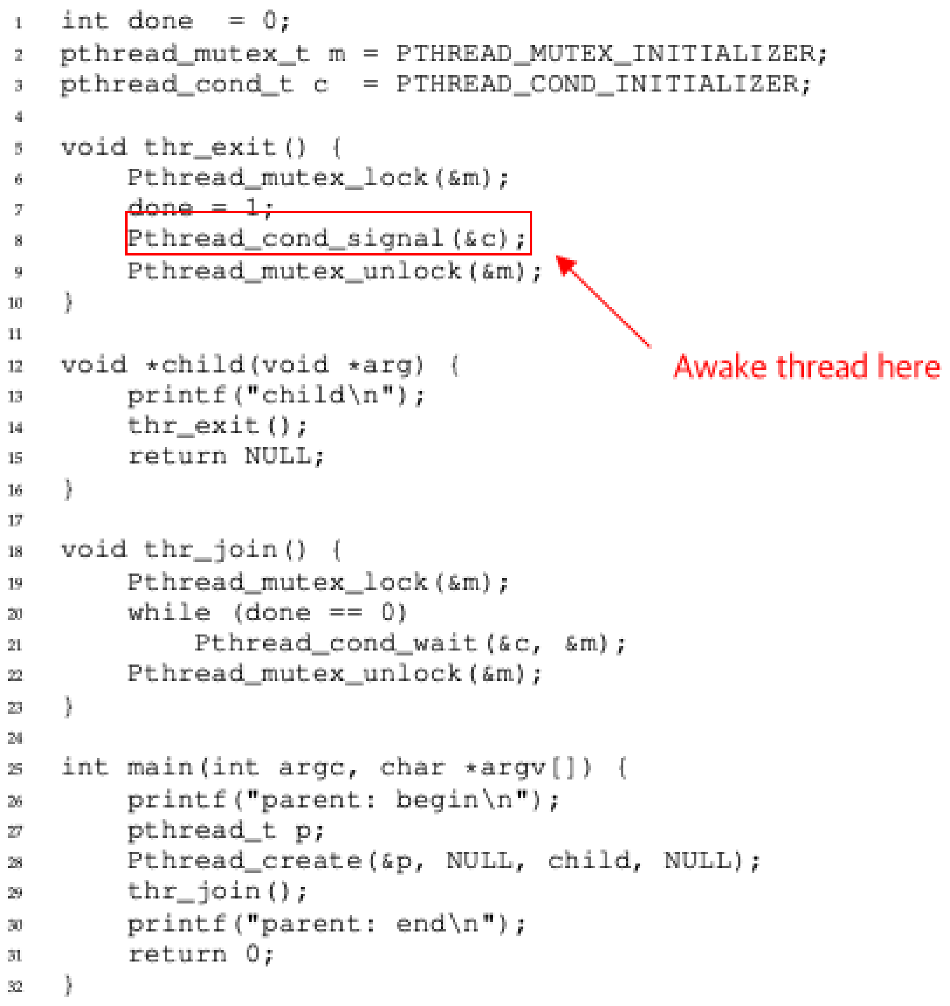


```
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

- **Syntax (Signal):**

  `Pthread_cond_signal(pthread_cond_t *c)`

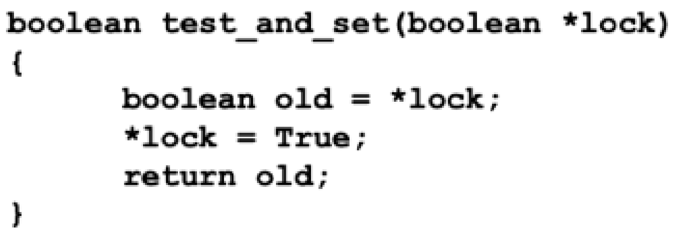
  Example



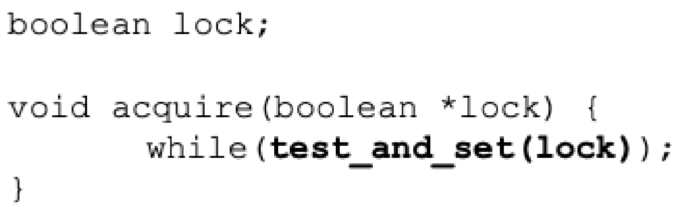
```c
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

24 Spinlock

- Is the simplest lock to build
- Uses a lock variable
  - 0 - (available/unlock/free)
  - 1 - (acquired/locked/held)
- Has two operations
  1. `acquire()`

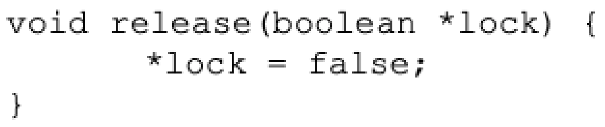


```
boolean test_and_set(boolean *lock)
{
      boolean old = *lock;
      *lock = True;
      return old;
}
```



```
boolean lock;

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

2. `release()`



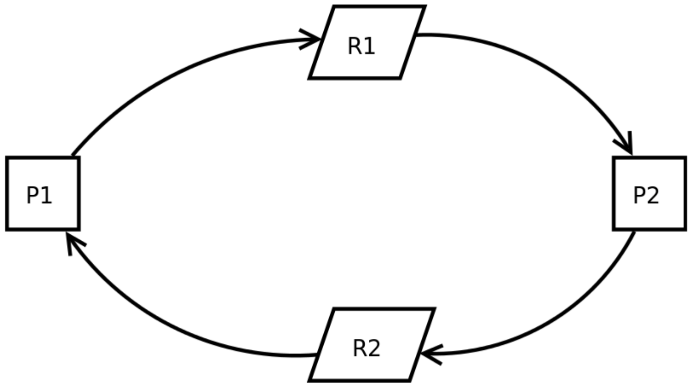
```
void release(boolean *lock) {
       *lock = false;
}
```

- Allows a single thread to enter critical section at a time
- Spins using CPU cycles until the lock becomes available.
- May spin forever

25 Livelock

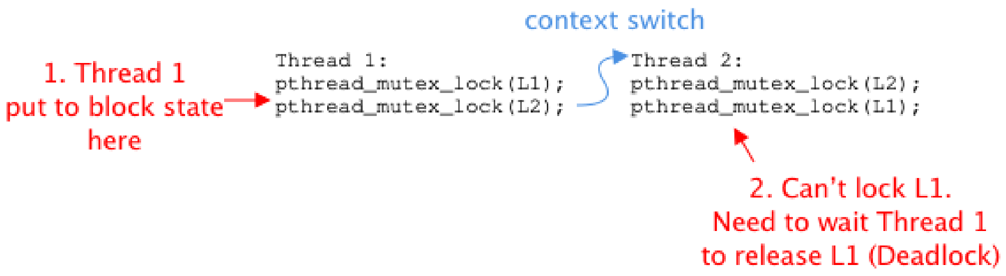
- Two or more threads reapeatedly attempting this code over and over (e.g acquiring lock), but progress is not being made (e.g acquiring lock)
- Solution: Add a random delay before trying again (decrease odd of livelock)

26 Deadlock



- Is a state in which each member of a group is waiting for another member including itself, to take action (e.g. releasing lock)
- Conditions for Deadlock (All four must be met)
  - **Mutual Exclusion**
    - Occurs when threads claim exclusive control of resources that they require (e.g. thread grabing a lock)
  - **Hold-and-wait**
    - Occurs when threads hold resources allocated to them (e.g locks that they have already acquired) while waiting for additional resources (e.g. locks that they wish to acquire)
  - **No Preemption**
    - Occurs when resource cannot be forcibly removed from threads that are holding them
  - **Circular Wait**
    - Occurs when there exists a circular chain of threads such that each threads hold one or more resources (e.g. locks) that are being requested by the next thread in the chain.

Example



```
Thread 1:
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
```

```
Thread 2:
pthread_mutex_lock(L2);
pthread_mutex_lock(L1);
```

context switch
1. Thread 1 put to block state here
2. Can't lock L1. Need to wait Thread 1 to release L1 (Deadlock)

- Preventions
  - **Circular Wait**
    - Write code such that circular wait is never induced
    - Is the most practical prevention technique
    - Requires deep understanding of the code base
    - **Total Ordering** (Most starightforward)

      Example

      Given two locks in the system (`L1` and `L2`), always acuiqure `L1` before `L2`

    - **Partial Ordering** (Applied to complex systems)

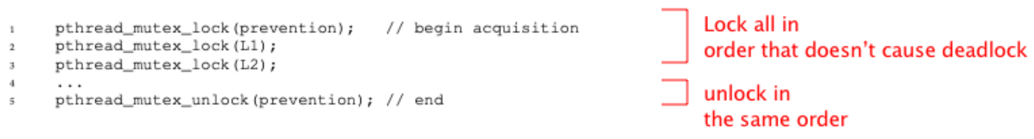
      Example

      Memory mapping code in Linux (has then different groups).

      (Simple) `i_mutex` before `i_mmap_mutex`

      (More complex) `i_mmap_mutex` before `private_lock` before `swap_lock` before `mapping->tree_lock`
  - **Hold-and-wait**
    - Can be avoided by acquiring all locks at once
    - Can be problematic
    - Must know which lock must be held and acquire ahead of time
    - Is likely to decrease concurrency (since all need to be acquired over their needs)

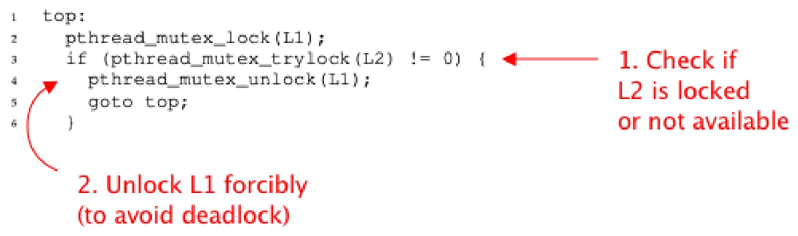
      Example



```
pthread_mutex_lock(prevention);    // begin acquisition
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
...
pthread_mutex_unlock(prevention); // end
```

- **No Preemption**
  - Can be avoided by adding code that force unlock if not available

Thread 1



```
top:
  pthread_mutex_lock(L1);
  if (pthread_mutex_trylock(L2) != 0) {
    pthread_mutex_unlock(L1);
    goto top;
  }
```

Thread 2

```
top:
  pthread_mutex_lock(L2);
  if (pthread_mutex_trylock(L1) != 0) {
    pthread_mutex_unlock(L2);
    goto top;
  }
```

  - `pthread_mutex_trylock` tries to lock the speicied mutex.
  - `pthread_mutex_trylock` returns 0 if lock is available
  - `pthread_mutex_trylock` returns the following error if occupied

    `EBUSY` - Mutex is already locked

    `EINVAL` - Is not initialized mutex

    `EFAULT` - Is in valid pointer
  - May result in **live lock**
- **Mutual Exclusion**
  - Idea: Avoid the mutual exclusion at all
  - Use **lock-free/wait-free** approach: building data structures in a manner that does not require explicit locking using hardware instructions

    Example

```
int CompareAndSwap(int *address, int expected, int new) {
  if (*address == expected) {
    *address = new;
    return 1; // success
  }
  return 0; // failure
}
```

- Avoidance
  - **Banker's Algorithm**

27 Process

- Is a program in execution
- Is named by it's process ID or PID
- Can be described by the following states at any point in time
  - Address Space
  - CPU Registers
  - Program Counter
  - Stack Pointer
  - I/O Information

  (wait. this is PCB)
- Exists in one of many different **process states**, including
  1. Running
  2. Ready to Run
  3. Blocked

  - Different events (Getting Scheduled, descheduled, or waiting for I/O) transitions one of these states to the other

References

1) Coding Horror, Understanding User and Kernel Mode, link

2) Kansas State University, Basics of How Operating Systems Work, link

3) Kansas State University, Glossary, link

4) Tutorials Point, User-level threads and Kernel-level threads, link

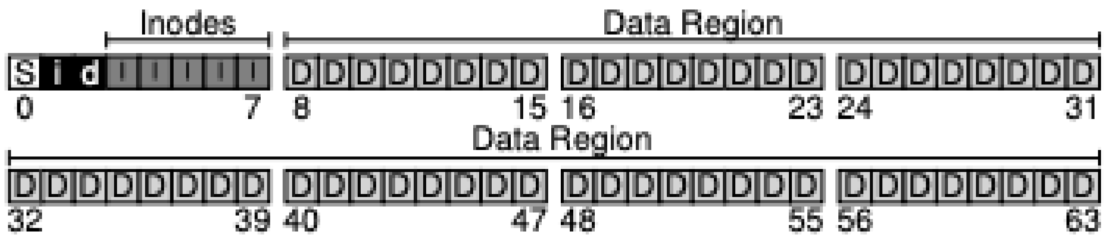
5) Science Direct, Scheduling Policy, link

6) Guru 99: What is CPU Scheduling?, link

7) Wikipedia: Round-robin Scheduling, link

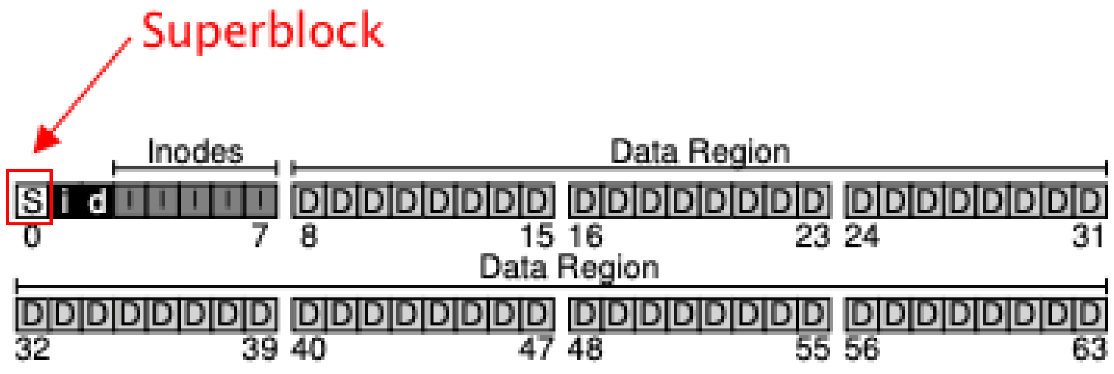
8) Kansas State University, The Process Scheduler, link

Index-based File System



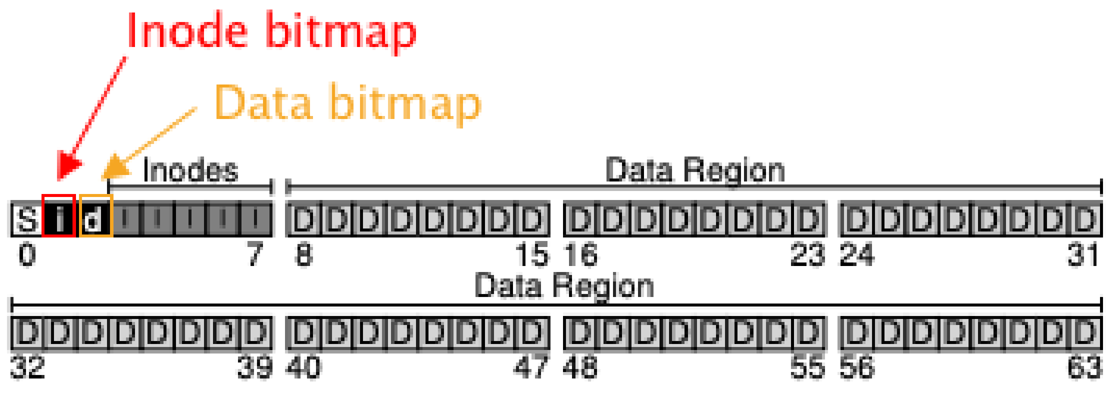
- Has following parts
  - Superblock
  - Inode Bitmap
  - Data Bitmap
  - Inodes
  - Data Region
- Each block in file system is 4KB
- Uses a large amount of metadata per file (especially for large files)

Superblock



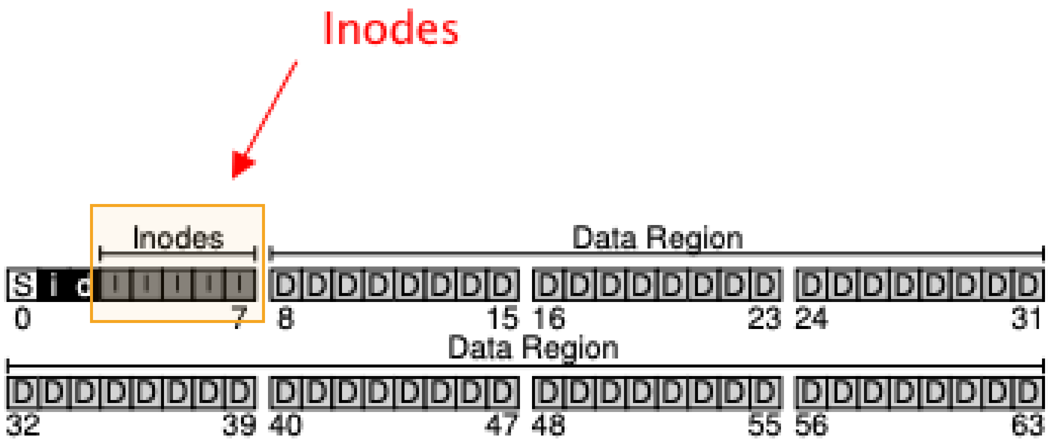
- contains information about the file system, including
  1. the number of inodes and data blocks in a particular file system
  2. the magic number of some kind to identify the file system type (e.g NFS, FFS, VSFS)
- The OS reads superblock first to initialize various parameters, and then attach volume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

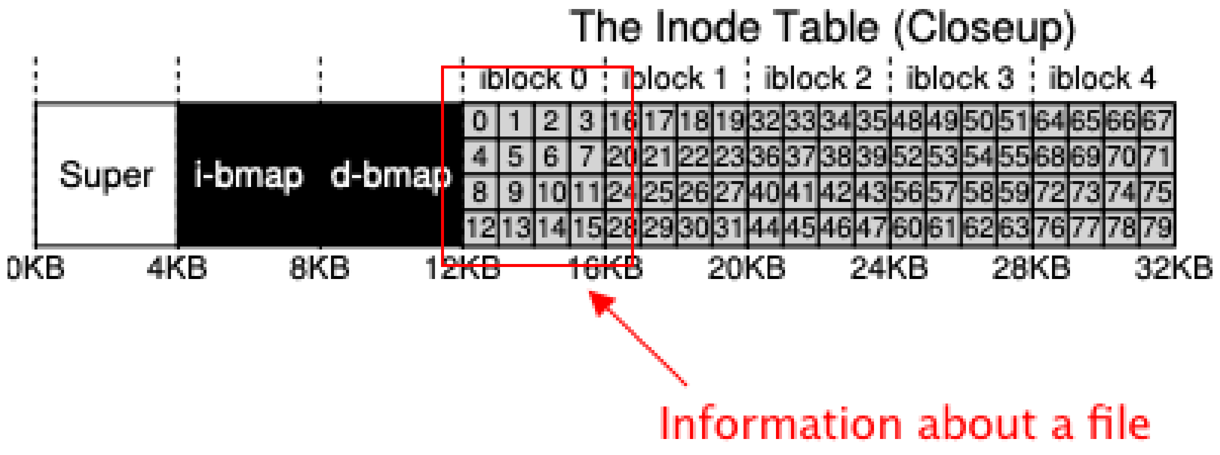
Bitmap



- 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

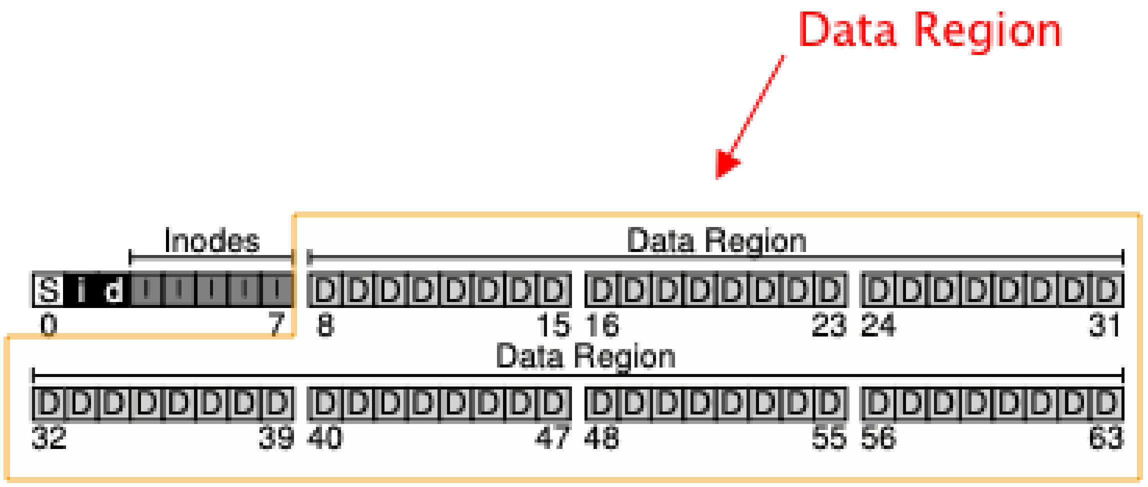
Inode





- 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

28 Block

- Size of each block is 4KB

29 lseek

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

30 Kilobyte

- 1 kilobyte is 1024 bytes

31 file

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

32 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.

33 inode

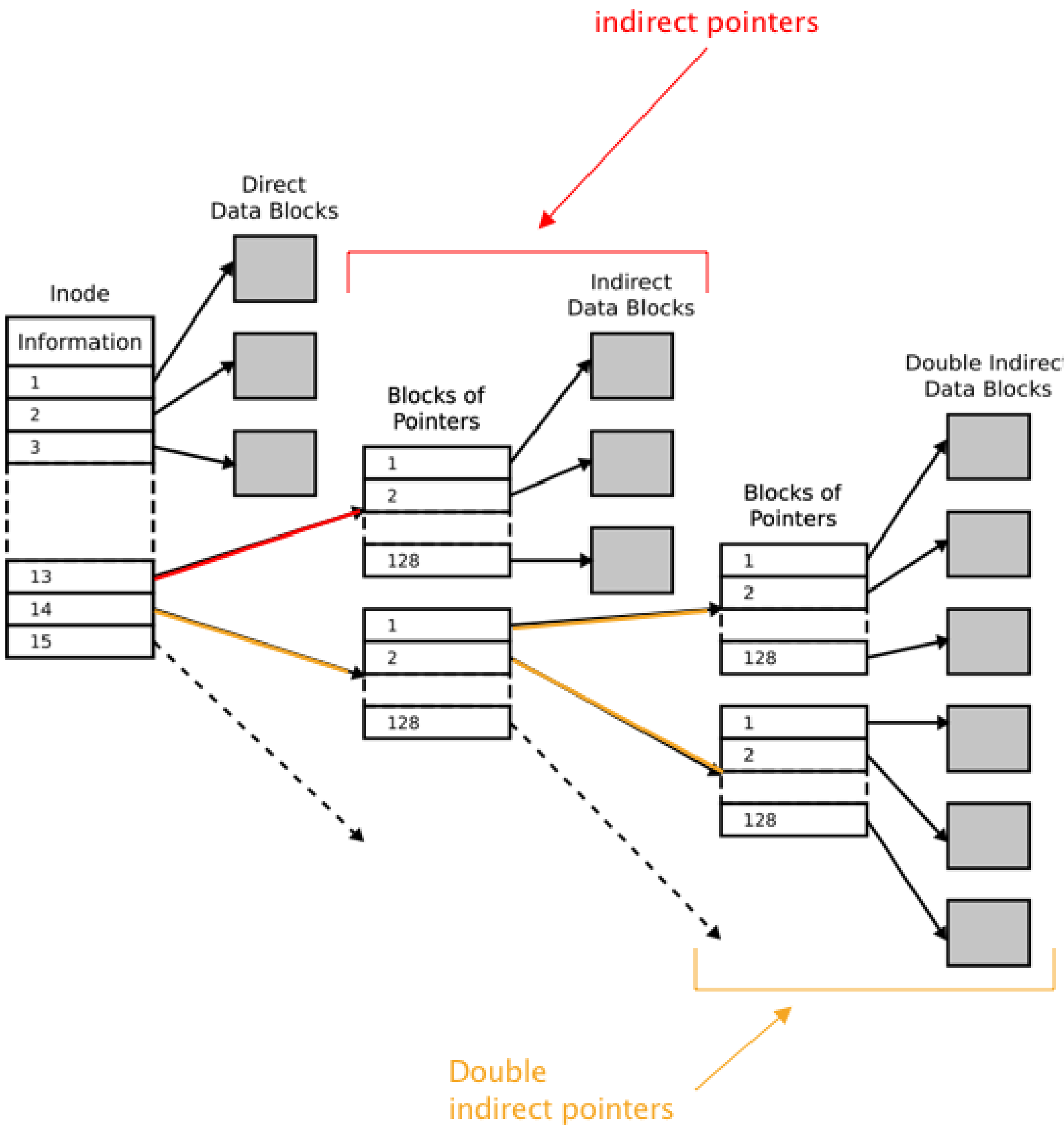
- total size may vary
- inode pointer has size of 4 byte
- Has 12 **direct pointers** to 4KB data blocks
- Has 1 **indirect pointer** [when file grows large enough]
- Has 1 **double indirect Pointer** [when file grows large enough]
- Has 1 **triple indirect Pointer** [when file grows large enough]

34 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 $(12 + 1024) \times 4\text{K} = 4144\text{KB}$

35 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 $\approx 4.20\text{GB}$

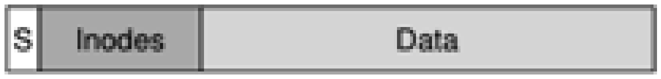


36 Triple Indirect Pointers

- 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 $(12 + 1024 + 1024^2 + 1024^3) \times 4K = 4299165744KB$ or $\approx 4.00TB$

37 Old UNIX File system

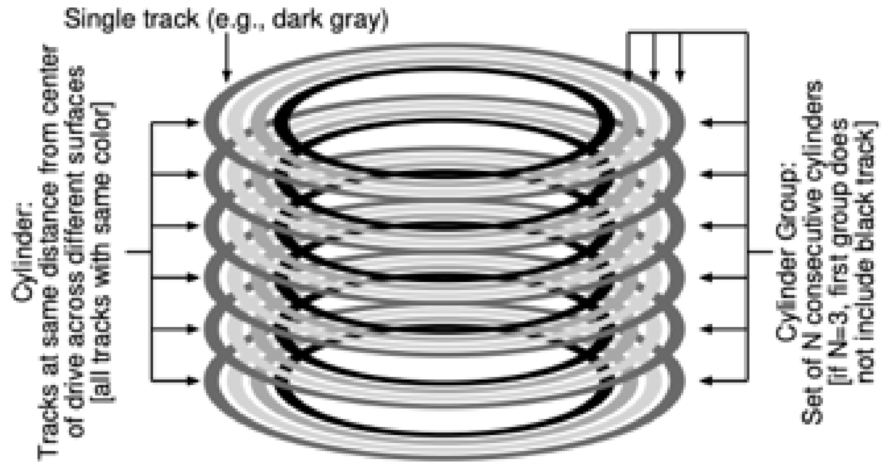
- was simple, and looked like the following on disk



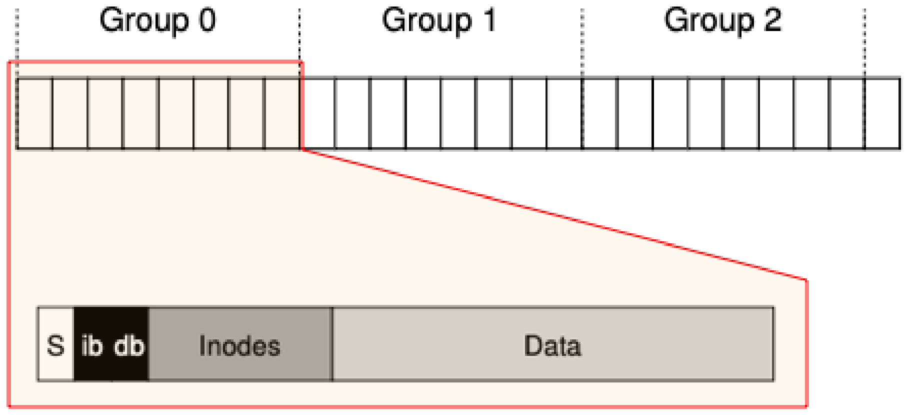
- has terrible performance
- suffers from **external fragmentation**
- had small data block (512 bytes) and transfer of data took too long

38 Fast File System

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


Single track (e.g., dark gray)
Cylinder:
Tracks at same distance from center
of drive across different surfaces
[all tracks with same color]
Cylinder Group:
Set of N consecutive cylinders
[if N=3, first group does
not include black track]

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



39 Bitmap

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

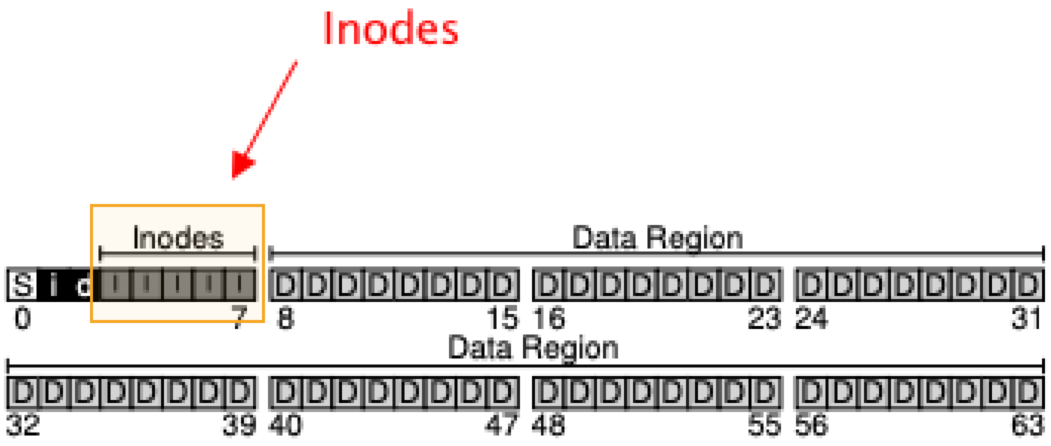
40 FFS Policies: Allocating Files and Directories

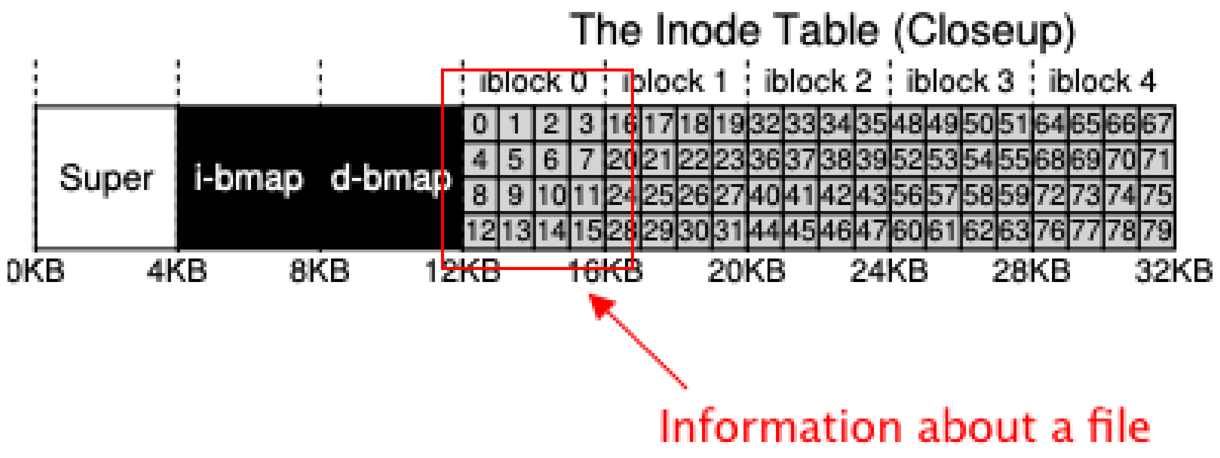
- Basic Idea: keep related stuff together, and keep related stuff far apart
- Directories Step
  1) Find the **cylinder group** with a low number of allocated directories and a high number of free inodes
     - low number of allocated directories → to balance directories across groups
     - high number 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**)

41 Inode



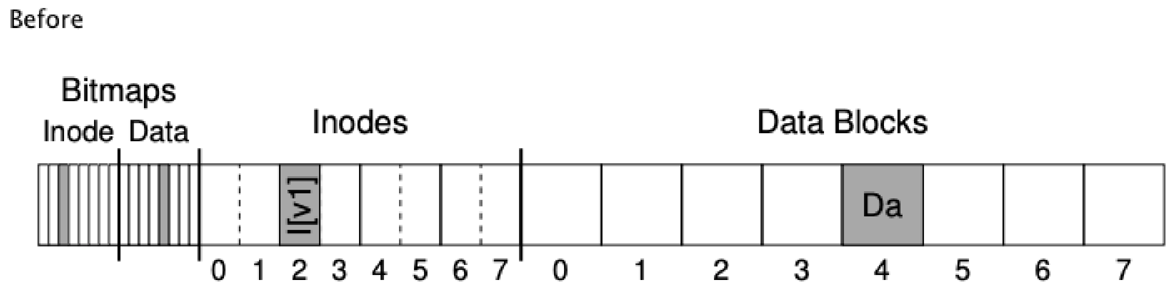


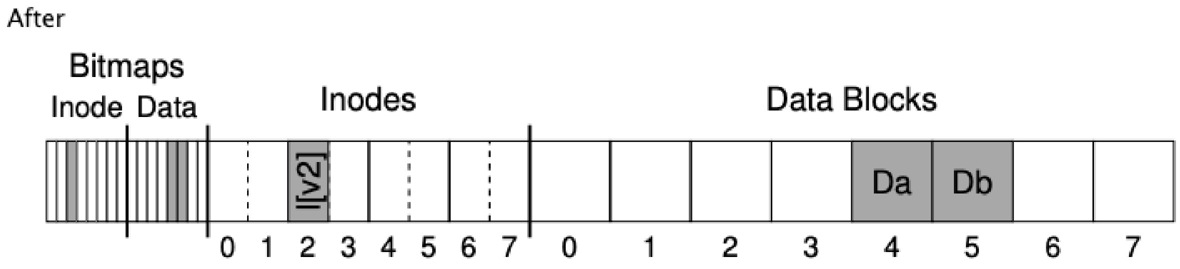
- Is a short form for **index node**
- Contains disk block location of the object's data [1]
- Contains all the information you need about a file (i.e. metadata)
  - File Type
    - e.g. regular file, directory, etc
  - Size
  - Number of blocks allocated to it
  - Protection information
    - such as 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

42 Crash Consistency

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

43 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

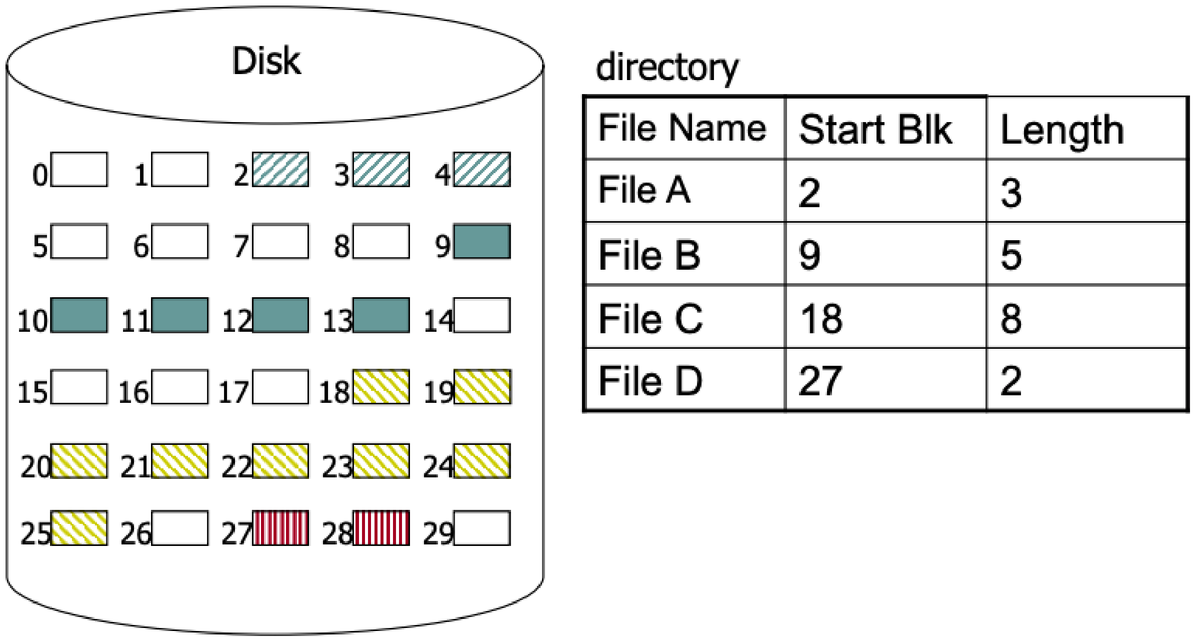
44 External Fragmentation

- 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]

45 Internal Fragmentation

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

sectionExtent Based File System



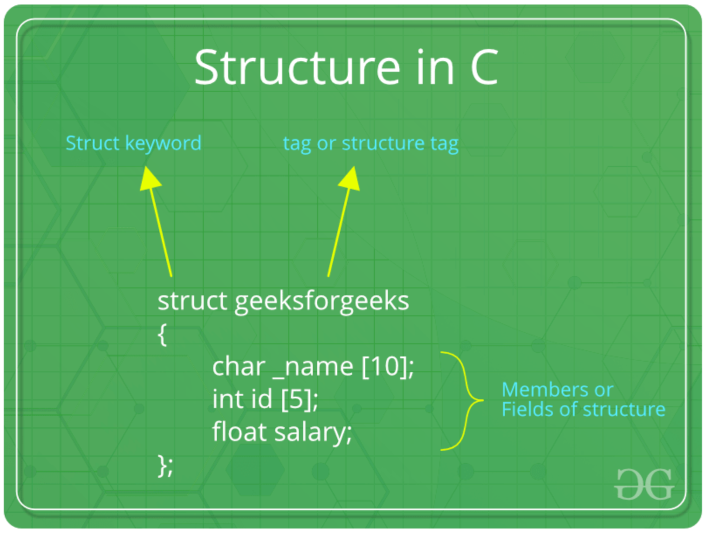
File Name	Start Blk	Length
File A	2	3
File B	9	5
File C	18	8
File D	27	2

- 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



```
struct geeksforgeeks
{
        char _name [10];
        int id [5];
        float salary;
};
```

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