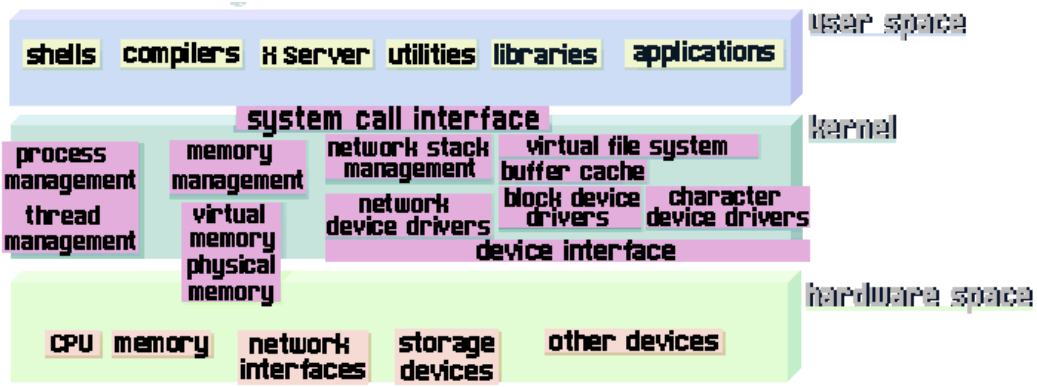
I/O Overview



System Call - Request to the kernel layer to perform a task that must be accomplished by the operating system. The number of system calls depends on the OS version.

Some categories of system calls:

I/O - Creating files and directories, reading/writing files, positioning in files, and reading directories.

Process Control - Creating processes and threads, terminating processes and threads, obtaining process ID(s).

Communication - Manipulating pipes, manipulating sockets, and managing shared memory.

kernel -> Part of OS that allocates machine resources, including memory, disk space, and CPU cycles to all other programs that run on a computer. The kernel includes the low-level hardware interfaces (drivers) and manages processes.

kernelspace -> The part of memory (RAM) where the kernel resides. Code running in kernelspace has full access to hardware and all other processes in memory.

fork -> To create a process. When one process creates another process, it forks a process (a spawn).

process -> The execution of a command by Linux.

parent process -> A process that forks other processes.

child process -> A process that is created by another process (the parent process). Every process is a child process, except for the first process, which is started when Linux begins execution. When you run a command from the shell, the shell spawns a child process to run the command.

User v. Kernel
The kernel is the part of the OS that runs with higher privileges, while user (space) usually means by applications running with low privileges.

User mode and *kernel mode* are terms that may refer specifically to the processor execution mode -> code that runs in kernel mode can fully control the CPU while code that runs in user mode has certain limitations.

(Grossly simplifying- Oh, woe!) The kernel space is the memory area that is reserved to the kernel, while user space is the memory area reserved to a particular user process.

The kernel provides system calls, which are low-level interfaces that user space applications use to request kernel services.
The kernel code itself can be divided into core subsystems (process management) and device drivers which manage hardware-specific operations. Core subsystems often interact with drivers to provide unified functionality.

Linux utilizes monolithic kernel architecture where all OS services run in the kernel space, meaning they all share the same memory space.

Virtual address space refers to the way the CPU sees the memory when the virtual memory module is activated.
(process) address space is part of the virtual address space associated with a process, it's the "memory view" of processes. And kernel (address) space is a "memory view" of the code that runs in kernel mode.
The virtual address space is like a backyard shed, and virtual memory (which can be put in the virtual space) is like a gardening tool being stored in the shed.

Professor:

You can do arithmetic calculations, load/store operations to move memory around your virtual address space, and that's about it. Programs have very little influence; anything beyond that requires the OS to support you, such as printf.

printf -> C standard library function. Within that function is a system call. printf, on your behalf, asks the OS to write something to the terminal. It gives a memory address with bytes for the characters, and a file descriptor number associated with the terminal.

If you're writing for efficiency, you'll want to minimize the system calls you make.

In previous courses, we used standard stream I/O functions (scanf, fgets, printf, fopen, fclose). These functions were intended to make I/O easier than the low level functions by handling:

- data conversions (e.g printf format codes convert from C data types to character output).
- buffer input data, allowing multiple calls to access data that is physically read with a single input request.

- buffer output data, allowing multiple calls to print data although a physical write is done for larger quantities.

- efficiently read/write data.

*Extreme and high-level IO will be used synonymously.

*std i/o || (stream) -> If the function uses a file structure pointer like FILE *stream/file pointer then it's high level.

The OS only understands low level i/o and uses an integer or "file descriptor numbers", like read(int fd. .) is the first file descriptor number.

High level

Function	Category	Purpose
scanf/fscanf/sscanf	std i/o, (stream)	stream input using format codes *note that sscanf gets its data from a string variable
gets/fgets	std i/o, (stream)	stream input of text lines
printf/fprintf	std i/o, (stream)	stream output using format codes
getc/fgetc	std i/o, (stream)	get next char from a stream
putc/fputc	std i/o, (stream)	put a char to a stream
fopen	std i/o	open a file for buffered i/o
fclose	std i/o	close a file opened by fopen
fread	binary read	binary input of one or more logical records
fwrite	binary write	binary output of one or more logical records
fseek	binary position	changes file position to a location relative to a number of bytes from the beginning of the file

Low level

Function	Category	Purpose
open	unix low level i/o	opens a file
close	unix low level i/o	closes a file
read	unix low level i/o	reads a specified number of bytes at the current position
write	unix low level i/o	writes a specified number of bytes at the current position
lseek	unix low level i/o	similar to fseek
stat, fstat	unix low level i/o	returns the stat structure for a file which includes inodeNr, file type, file mode, number of links, size, etc.
opendir	unix low level i/o	opens the specified directory for reading
readdir	unix low level i/o	reads the next directory entry
closedir	unix low level i/o	closes a directory

What's the advantage?

1. convenience

2. performance

Data conversions -> Makes them super simple
"If I had a floating point number in memory, and I wanted to write that to the terminal, I could do that- but it's a whole lot easier to do it via format string with the proper format specifiers. With f/printf %f %lf and pass the value of the floats, convert it to a string, then convert it to the target string.

Part of the contents of the structure is the buffer. It buffers input and output.

Example: Let's say I wanna read 10 bytes, maybe for some reason I want to read a file 10 bytes at a time. If you're using high level I/O it'll buffer the input data (the i/o block size is 1 MB on fox servers) every time you ask for another 10 bytes, it doesn't have to perform a system call again or hit the storage device- it just reads it from your local memory.

Aside from having to hit the storage device every time, the system call is a big performance inpetitive. At that point, your program is blocked and you can't do anything.
Your program has to block and wait because presumptively I'm going to use the 10 bytes when it's available.

When you use the system call you voluntarily give up the CPU and be put in a waiting queue for the device controller for the storage device you're trying to access. Then when it's finished reading the data, it sends an interrupt, that's handled, etc. Even at that point your process doesn't get the CPU back right away and you're put in a ready queue to get the CPU back again.

If you did a lot of system calls and they weren't buffered, it'd be extremely slow.

Files

Linux/Unix manages file storage as blocks (typically 4096 bytes). One file can consist of many bollocks which are not usually contiguous. For a text file, many text lines can be in a block. Some text lines might actually span across one or more blocks.

inode

Linux/Unix needs a mechanism to keep track of the blocks that make up a file. Why don't the blocks simply have a pointer to the next block? That would work for simple stream files. But you don't want metadata and userdata sitting adjacent to eachother, because you can overflow into the metadata; A customer could have access to other data that they shouldn't be able to access.

There are 2 distinct components in an inode:
stat structure/information & everything below that is a pointer.

An inode contains:

- stat info
  - size of the file in bytes
  - device ID of the device containing the file
  - user ID of the owner
  - group ID
  - file mode containing the file type and the permissions mode (3 octal value)
  - timestamps for when the inode was last changed, content last modified, and last access
  - link count.
- index of pointers:
  - 12 direct pointers to data blocks
  - 1 indirect pointer
  - 1 double indirect pointer
  - 1 triple indirect pointer

An inode doesn't include the name of the file

So for pointers:

`1 indirect pointer` -> `block of data pointers`

`1 double indirect pointer` -> `a block of pointers` -> `blocks of data pointers`

`1 triple indirect pointer` -> `block of pointers` -> `blocks of pointers` -> `blocks of pointers` -> data blocks

All the pointers will be the same size, just like in memory it doesn't matter what direction the pointer is in- they're all the same size.
sizeof(int *) is the same as sizeof(int **)
sizeof(int *) == sizeof(int **) == sizeof(int ***)
Because the pointer is just a memory address. Same with the levels of indirection, it doesn't matter how many levels the block number or address that's in the same address space.

Every inode has a unique number that identifies it on that particular file system.

`man 2 stat`
These functions return information about a file.

All these system calls return a stat structure which contains the following fields:

man 2 stat:

```
      struct stat {
            dev_t st_dev;         /* ID of device containing file*/
            ino_t st_ino;         /* inode number */
            mode_t st_mode;       /* protection */
            nlink_t st_nlink;     /* num of hard links */
            uid_t st_uid;         /* user ID of owner */
            gid_t st_gid;         /* group ID of owner */
            dev_t st_rdev;        /* device ID (if special file) */
            off_t st_size;        /* total size, in bytes */
            blksize_t st_blksize; /* blocksize for filesystem I/O */
            blkcnt_t st_blocks;   /* num of 512B blocks allocated */
            time_t st_atime;      /* time of last access */
            time_t st_mtime;      /* time of last modification */
            time_t st_ctime;      /* time of last status change */
      };
```

```
$ ls -i
-rwx------ 1 ssilvestro faculty  514 Feb 21 09:52 findexes.bash
-rwx------ 1 ssilvestro faculty  705 Feb 21 09:52 killexes.bash
-rwx------ 2 ssilvestro faculty  423 Feb 21 09:52 linecount.bash
-rw------- 1 ssilvestro faculty 1.2k Feb 21 09:52 safegcc.bash
```

The '2' means there's another directory entry on the file system that points to the directory.

```
$ ls -i linecount.bash #finding inode num
52560050 -rwx------ 2 ssilvestro faculty  423 Feb 21 09:52 linecount.bash
$ find . - inum 52560050 -ls
52560050 -rwx------ 2 ssilvestro faculty  423 Feb 21 09:52 ./working/linedup.bash
52560050 -rwx------ 2 ssilvestro faculty  423 Feb 21 09:52 linecount.bash
#This will find all the files under the current directory that has that
#same inode number
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char ** argv) {
      printf("size of struct stat = %zu\n", sizeof(struct stat));
      return EXIT_SUCCESS;
}
```

The white data blocks are also not to scale. They're the smallest allocated unit of storage on the storage device, so you can't allocate anything smaller than a block.

A block is a block is a block. It doesn't matter what is in the block. If we're using a block of size 4kb, that means all the blocks are 4kb.

inode doesn't include the name of the file.
int struct stat{} - there's no file name.
hard links - a filename exists only in a directory.

The OS uses inodes and inode numbers to uniquely identify files. An inode represents a whole file. You can't take the double indirect pointer and have it point to a data block, only the direct pointer can do that. Data blocks contain the actual file data, we're tracking it via the inode itself and these traditional blocks of pointers.

A block address is another name for a pointer. It's the width of the address space for blocks.
Assuming 4 byte block addresses and 4096 byte blocks, how big of a file can we possibly reference with this inode approach?
This means - how large of a file can we support?

inode blocks populate from the top -> down.
In order to use the single indirect pointer you've already populated: stat info, direct tr p1, 2, etc. So if we want to determine what is the largest file we can support; we can break it into pieces.

What do direct pointers do? They point directly to data blocks. I know if I have *n* data blocks, I'll have *n* direct pointers. If I have *n* direct pointers, I'll have *n* data blocks.

Again:

A block is a block is a block.
- A block is the smallest allocatable unit of storage, and the OS/device/controller doesn't care what the block is being used for.

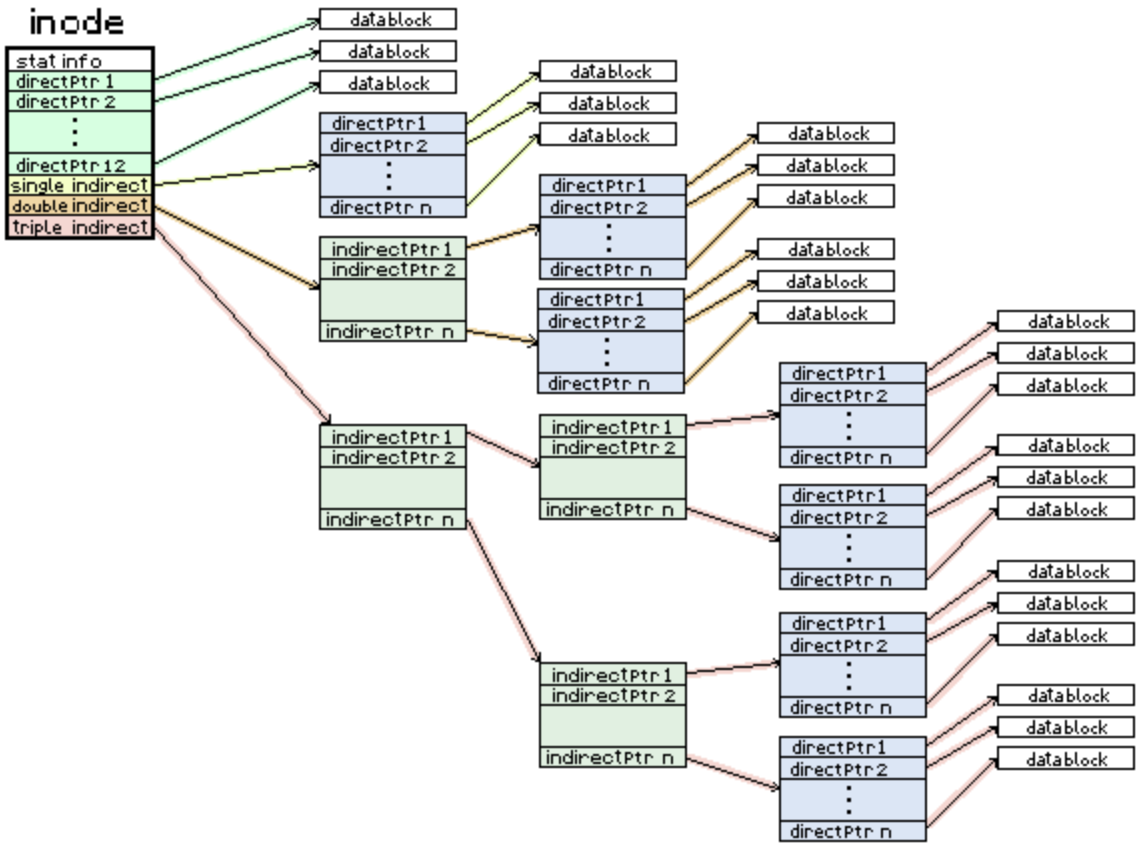
- A pointer of indirection level *x* points to a block of pointers of indirection level *x-1*.
  Similar to an int * in C would point to an int. int * -> int

indirect pointer -> block of direct ptrs
double pointer -> single indirect ptrs
triple pointer -> double indirect ptrs

All pointers are the same size.
sizeof(int *) == sizeof(int **) == sizeof(int ***)

Pointers per block = blk_sz/ptr_sz

size of container/size of object I wish to store inside the container



In order to use the single indirect pointer, the 12 direct pointers have already had to be populated.

Back to the question:
Assuming 4 byte block addresses and 4096 byte blocks, how big of a file can we possibly reference with this inode approach?

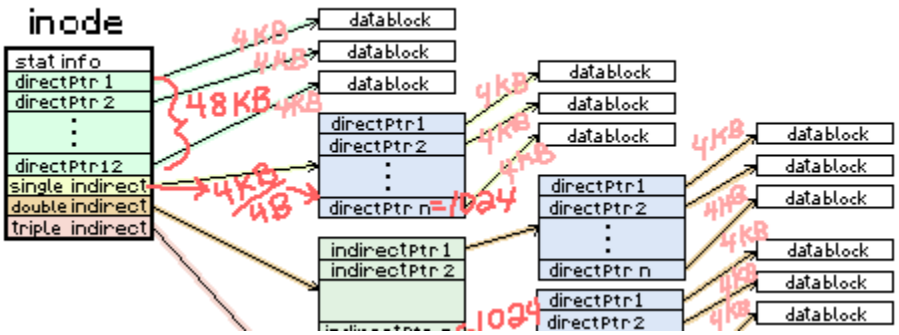
1. How much data can be referenced by the total from the 12 direct pointers?

    12 direct ptrs * 4KB = 48KB (direct ptr 1 - 12)
    12 direct ptrs * 4096 bytes = 49,152 data bytes

2. How much data can be referenced by the 12 direct pointers and the additional indirect pointer?

    Pointers ber block = block_size / pointer_size
    $4KB/4B = 2^{12}B/2^{2}B = 2^{10} = 1{,}024$ pointers per block = n
    All the other blocks will be 1,024 because all pointers are the same size (4KB).

If one direct pointer gives me access to a 4KB data block; then all of a sudden I enter an extra 1,024 of those into the system. I've added (1,024 * 4KB) because this single indirect pointer points to a block of direct pointers, and that block of direct pointers points to data blocks.



1,024 * 4KB = 4MB
The single indirect pointer by itself adds 4MB
So at this point the answer is 4MB + 48KB
(12 direct pointers + single indirect pointer)
Total number of direct pointers = 12 + 1024 = 1,036
1,036 * 4,096 = 4,243,456 data bytes = 4MB + 4KB

3. How much data can be referenced by the 12 direct pointers, one indirect pointer, and the one double indirect pointer?

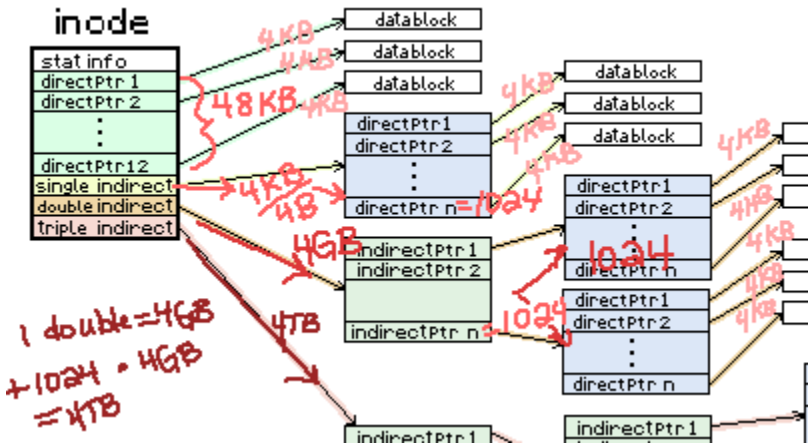
If one single indirect pointer gave me an additional 4MB of space, I'm just adding another 4MB

So
12 direct pointers -> 48KB
(12 * 4096 = 49,152 = 48KB)

12 direct pointers + single indirect -> 4MB
pointers per block = 1024
((1024+12) * 4096 = 4,243,456 = 4MB + 48KB)

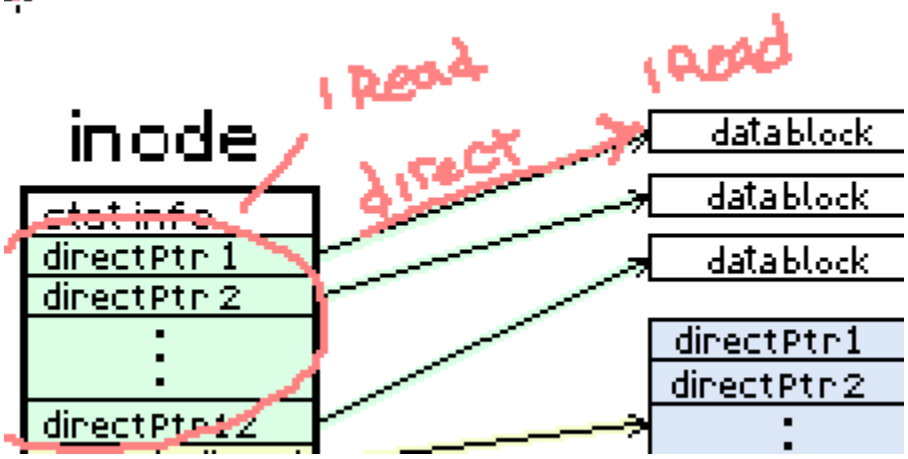
12 direct pointers + single indirect + double indirect -> 4GB + 4MB + 48KB
((1024*1024) = 1,048,576
(1,048,576 + 1024 + 12) = 1,049,612
1,049,612 * 4096= 4,299,210,752 = 4GB + 4B + 48KB

4. Adding the triple indirect pointer
4TB + 4GB + 4MB + 48KB = ~4.004TB

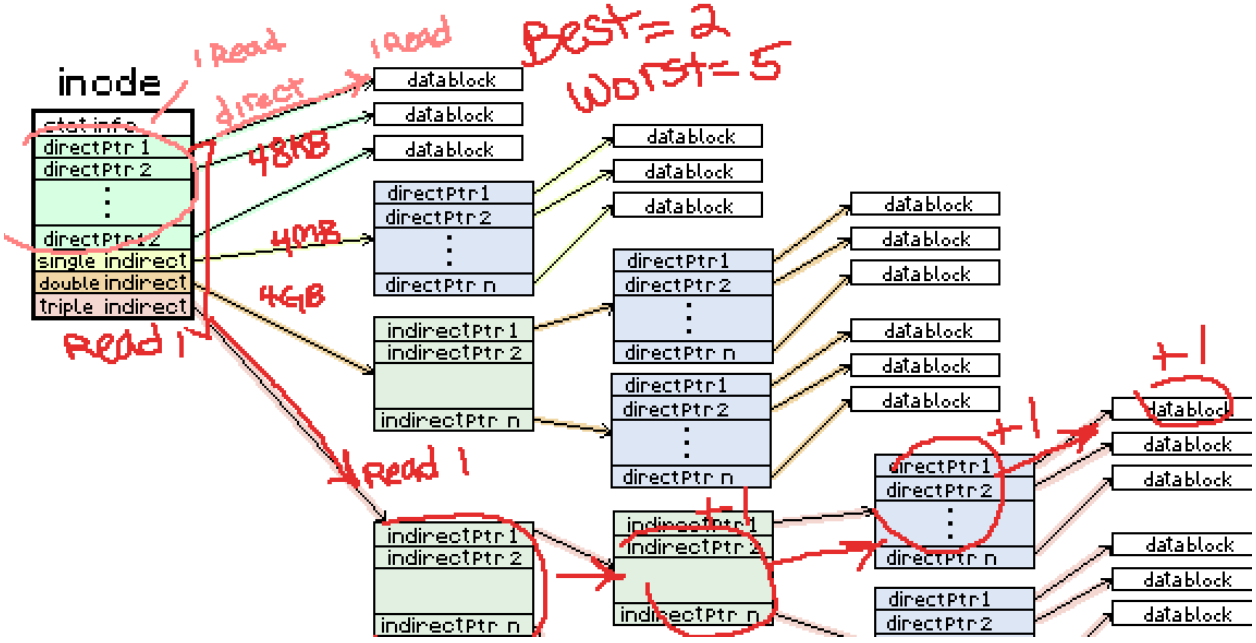


To access a particular data block (just one), if our file is 1 TB, what is the maximum number of reads necessary?
We would need the triple indirect pointer due to the size. Worst case is with a triple indirect.
1 - for inode (which is frequently in memory if heavily used)
3 - index node reads since triple indirect
1 - data read from data block
Worst case is 5 reads.

Best case: If we're lucky and the block I'm looking for is pointed by one of the 12 direct pointers built into the inode. That means:
1 - for inode
1 - data read from data block
Best case is 2 reads.



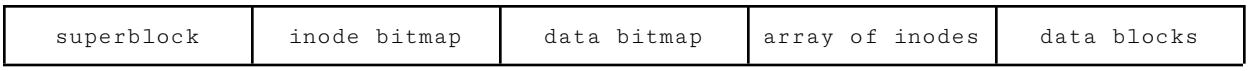
Worse case



File Systems

Physical disks are organized into file systems to provide logical access to the actual bytes of a file. The file system controls how data is stored and retrieved. To logically access data in a file, we specify a filename and a relative byte offset. Using the file system, Linux translates those into actual physical location.

Formatted file system:



The superblock and the bitmaps are usually kept in memory during execution.

superblock - size of the file system, number of inodes, number of data blocks, block size

allocation bitmaps - free and used inodes, free and used data blocks

inode blocks - array of inode blocks

data blocks - used for data blocks

Example for specifying block size via 'mkfs' command:

```
mkfs [options] [-t type] [fs-options] device [size]
```

-t -> type of filesystem to be built (default is ext2 if not specified)
fs-options -> filesystem-specific options to be passed to the real file system builder

```
$ mkfs.ext4 -L my_volume /dev/sda1 #assigns a volume label to file sys.
$ mkfs.ext4 -n /dev/sda1 #dry run, no file system created
```

Most common uses for 'mkfs.ext4' is formatting a new partition to prepare it for data storage:

```
$ mkfs.ext4 /dev/sdb1
```

```
$ ls /sbin/mkfs*
lrwxrwxrwx 1 root root 6 Dec 18 2013 /sbin/mkfs.ntfs -> mkntfs
lrwxrwxrwx 1 root root 6 Sep  9 2015 /sbin/mkfs.ext4dev -> mke2fs
etc . .
```

array of inodes & data blocks:

"You can sort of think of an array as well. Especially if we start counting data blocks at some first number, like 0. And you went all the way up to 'n'. Well you can think of a data block number then as an index into this contiguous array of data blocks which makes up the vast majority of the file system.

The array of inodes is similar; it's a contiguous section that contains just inodes, and of course the inode number is treated as an index into this array.

The bitmap stuff- A bitmap is a space and speed efficient technique/data structure. It's a means by which you can easily keep track of the binary statis of things. Whether a certain thing is in state A or B. Those things (with OS) are gonna be either blocks of memory or blocks of storage and the statis is gonna be allocated vs un-allocated.

You've got one bit that is gonna represent the statis of each item that you have, e.g 1,024 data blocks for example; then my bitmap would have 1,024 bits. The idea is the OS can scan this:
0101001001000101001001000001000001000000010001
very quickly to determine whether there is an available block or inode.
There are some instructions in the x86-64 that will scan double or quadword 64-bit chunks for certain things like "does it have a 0 in it, or 1, or how many leading zeros, etc" those instructions are useful for scanning bitmaps (amongst other useful things).
So we'd scan it, maybe we're looking for a 0, and once we find that 0 we allocate it, flip the bit to 1 in the bitmap to indicate it's now in use:
1101001001000101001001000001000001000000010001
It's space efficient and performance efficient which is why bitmaps are widely used for allocators and tracking the allocation status of all sorts of things.

Directories

There can be many directories on a file system. A directory is a single file which contains a list of directory entries:

filename - name of a file
inode - index into an array of inodes

Effectively, a directory maps file names to inode values. Once we have a file's inode, we can then reference any of its data. To allow the internal implementation of a directory file to be implementation independent, Linux provides functions for creating, reading, and removing directories.

A directory is a special type of file, not uncommon to hear them referred to as a "directory file".
Everything in Linux/Unix is a file, including devices. A directory just has mappings between filenames and inode numbers.

"If it's another directory we're trying to access, let's say I'm in a directory
and I'm looking for something in a subdirectory named:
$ **# cat Pear/one**
I'm in this fruit directory so I can read my inode in my current directory and see the mappings, I know I'm looking for a subdirectory named pear. Because we have the inode number we can get all the pointer data for that subdirectory so we can read its mapping and then we look inside its mapping to see if there's a subdirectory named 'one' and there is so we have that entry's inode number, and then you can see how this just cascades down and then we read the inode for that file (assuming its a data file) and then you read the file data directly from utilizing the inode pointer.

Where does Linux know the type of a file (e.g. directory, regular file, link, pipe, socket)? Is it in the directory?
No it's not. The file mode is called the "mode" because it contains more than just the permissions. The least significant 9 - 12 bits represent the permissions of the file. On the other hand, the most significant represent the type of file. It's there in the node, which makes it a part of the inode stat structure.

S_ISREG(m) -> is it a regular file?
S_IFIFO 0010000
"If I wanted to see if something was FIFO. You can see it's an octal number because it begins with a leading zero. What you do with this is you take this value, and I do a bitwise AND with mode, and if that results in true/nonzero value, then I know this was a FIFO. Same with directory (S_IFDIR). You can do the same thing with the user class here:
S_IRUSR 00400
S_IWUSR 00200
S_IXUSR 00100
These are octal digits, so each of these can be represented by 3-bits.
If we wrote this in binary it'd be 001 000 000
If there was a bit in that 0 position, indicating execution permissions, then 1 AND 1 is true. If it was 0, 1 AND 0 is false."

Links

Files can be referenced in multiple places through the use of links. A symbolic link/symlink is a file with its own inode which points directly to another file (the contents of the symlink's file is a path to the other file) while a hard link shares the file's inode (a symlink shares nothing).

If the original file to which a link is pointing is deleted, the symlink will remain, but no longer work since it points to nothing. The hardlink will still point to the file even though the original name/inode mapping no longer exists.
You can delete a softlink without deleting the actual file since the inode of the file is different than the inode of the link. The softlink is simply a file which contains a path to the target file.
Creating a hard link is like creating a new file which points to the same data as another file.

$ **ls ulc ulcs ulcs3423**

```
lrwxrwxrwx 1 ssilvestro faculty 30 Sep 18 2019 ulcs -> /usr/local/courses/ssilvestro
lrwxrwxrwx 1 ssilvestro faculty 19 Sep 20 2019 ulc -> /usr/local/courses
lrwxrwxrwx 1 ssilvestro faculty 37 Aug 27 2024 ulcs3423 ->
/usr/local/courses/ssilvestro/cs3423
```

These are symbolic links/symlinks/softlinks. The link itself is a separate file.

```
lrwxrwxrwx 1 ssilvestro faculty 19 Sep 20 2019 ulc -> /usr/local/courses
```

1. The link
2. The file that is being pointed to
3. The file sizes of the links (in bytes).

The contents of the files are just the filenames and paths of whatever you're wishing to refer to.
Similar to windows shortcuts (.lnk files). This is basically the exact same thing. You create a file, and its contents is simply the path to some other file.
So instead of having to type /usr/local/courses/ssilvestro/cs3423 every single time, you can just type 'ls ulcs3423/'

To display the inode number for a symlink:

$ **ls -i ulc ulcs ulcs3423**

```
52559972 lrwxrwxrwx 1 ssilvestro faculty 30 etc..
52559965 lrwxrwxrwx 1 ssilvestro faculty 19 etc..
52559953 lrwxrwxrwx 1 ssilvestro faculty 37 etc..
//file type is 'l' for link
```

The original file's inode number:

$ **ls -i /usr/local/courses/ssilvestro/**

```
925704 drwxr-xr-x 5 1002 faculty 4.0k Feb 17 15:27 usr/local/courses/ssilvestro/
```

Hard links are just directory entries.

$ **ls**

```
-rw------ 2 ssilvestro faculty 423 Mar 24 13:28 linecount.bash
```

1. Represents the number of hardlinks. This is stored in the stat structure of the inode: nlink_t or st_nlink;

Lets say I have a home directory that contains "blah.txt" that maps to this inode number:

```
~:
blah.txt   3938050284
```

And then maybe in a different directory, like under '~/working/' I have something like 'readme.md' and it points to the same inode number. These are the same file, it's just one file with two directory entries.

```
~/working/:
readme.md  3938050284
```

So going back to:

```
-rw------ 2 ssilvestro faculty 423 Mar 24 13:28 linecount.bash
//How do I know the other file that points to this inode?
```

```
$ ls -i linecount.bash
52560050 -rw------ 2 ssilvestro faculty 423 Mar 24 13:28 linecount.bash
$ find . -inum 52560050 -ls
52560050 4 -rwx------ 2 ssilvestro faculty 423 Mar 24 13:28 ./working/linedup.bash
52560050 4 -rwx------ 2 ssilvestro faculty 423 Mar 24 13:28 ./linecount.bash
```

You can look up the inode number, then use find to look up that specific number which will show you all the files under said number in various directories.
The metadata will always be the same because they point to the same inode, and the metadata is stored in the stat structure of the inode. So the permissions are the same, the link counts, the owners, owner group, filesize, timestamps, etc. The only things that are different are the filenames. But they're part of the directory, they're not part of the inode.

```
//making a new hard link
$ ln linecount.bash newdirent.bash
$ ls -i newdirent.bash
52560050 -rwx------ 3 ssilvestro faculty 423 Mar 31 09:52 newdirent.bash
$ ls -i linecount.bash newdirent.bash working/linedup.bash
52560050 -rwx------ 3 ssilvestro faculty 423 Mar 31 10:35 ./working/linedup.bash
52560050 -rwx------ 3 ssilvestro faculty 423 Mar 31 10:35 ./linecount.bash
52560050 -rwx------ 3 ssilvestro faculty 423 Mar 31 10:35 newdirent.bash
```

You can't tell which came first since they all have the exact same metadata.

```
$ rm newdirent.bash //removing the hard link
$ ln -s linecount.bash newsymlink.bash //new symlink
$ ls -i linecount.bash newsymlink.bash
52560050 -rwx------ 2 ssilvestro faculty 423 Mar 31 10:35 linecount.bash
52561077 lrwxrwxrwx 1 ssilvestro faculty  14 Mar 31 10:38 newsymlink.bash -> linecount.bash
```

```
//Didn't increase the hardlink count, the inode counts are completely different.
$ rm newsymlink.bash
//removes symlink

$ ln /usr/local/courses/ssilvestro/loadme ~/loadme
ln: failed to create hard link '/home/ssilvestro/loadme' =>
'/usr/local/courses/ssilvestro/loadme': Invalid cross-device link
```

//Why did it fail to create the hard link?
Remember, hard links are directory entries. An inode number is only unique within that particular file system. I can't create a hard link from one file system to another because the inode numbers are not unique across filesystems, only within a filesystem. Basically for the fox machines that means any hard links you have only exist somewhere within or under your home directory.