50004 - Operating Systems - Lecture 15

Oliver Killane

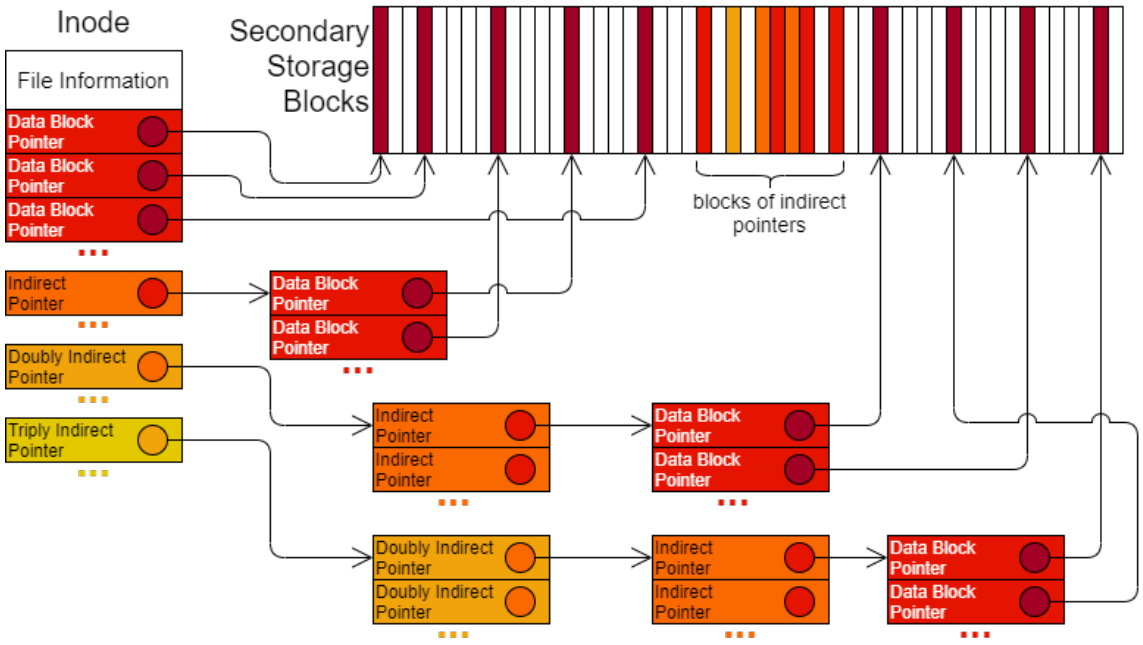
20/12/21

Lecture Recording

Lecture recording is available here

Linux Inodes Continued...

Indirect pointers are used to increase the number of data blocks that can be associated with a file. Indirect pointers can point to blocks, filled with either more indirect pointers or pointers to data blocks.



For example:
Take an OS with inodes containing:

- *6 direct pointers*
- *1 indirect pointer*
- *1 doubly indirect pointer*

Each pointer requires 8 *bytes, each block is* 1024 *bytes and each indirect block fills a whole block on the disk.*

What is the maximum file size?

Each indirect block can hold $\frac{1024}{8} = 128$ pointers to other blocks. Hence we can calculate the maximum number of blocks:

$$6 + 128 \times 1 + 128 \times 128 \times 1 = 16,518 \; blocks = 16,914,432 \; bytes \approx 16.13 \; MB$$

By adding a single triply indirect pointer we can increase this:

$$6 + 128 + 128^2 + 128^3 = 2113670 \; blocks = 2,164,398,080 \; bytes \approx 2.02 \; GB$$

We can also determine the number of disk reads required for accessing a block.
Given an OS using inodes with 6 direct pointers, 1 indirect and 1 doubly indirect, with pointers being 4 bytes long and blocks 1024 bytes large.

Calculate the disk block reads required to get the 1020^{th} *data byte, and then* $510,100^{th}$ *data byte, assuming nothing is cached in memory.*

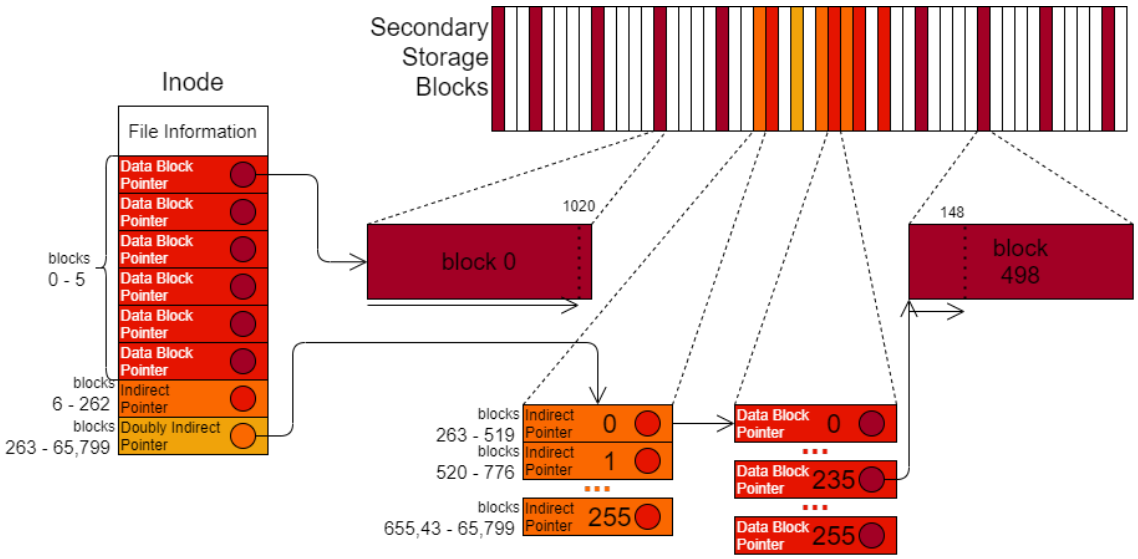
$1020^{th} byte$

$\lceil \frac{1020}{1024} \rceil = 1 \rightarrow 1^{st}$ *block*. Hence 1 read required to get the inode, and 1 required to get the data from the block pointer & the offset of 1020 bytes.
2 *reads*

$510,100^{th} byte$

$\lceil \frac{510,100}{1024} \rceil = 499 \rightarrow 499^{th}$ *block*. Each block of pointers contains $\frac{1024}{4} = 256$ *pointers*. Hence this block is pointed to indirectly by the doubly indirect pointer.

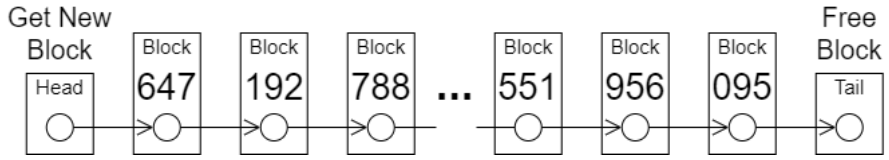
Therefore we require 1 read to get the inode, then 2 reads to get the data block, then 1 more to read from the data block.
4 *reads*



Free Space Management

When allocating a new block for a file, we need to quickly determine which is available.

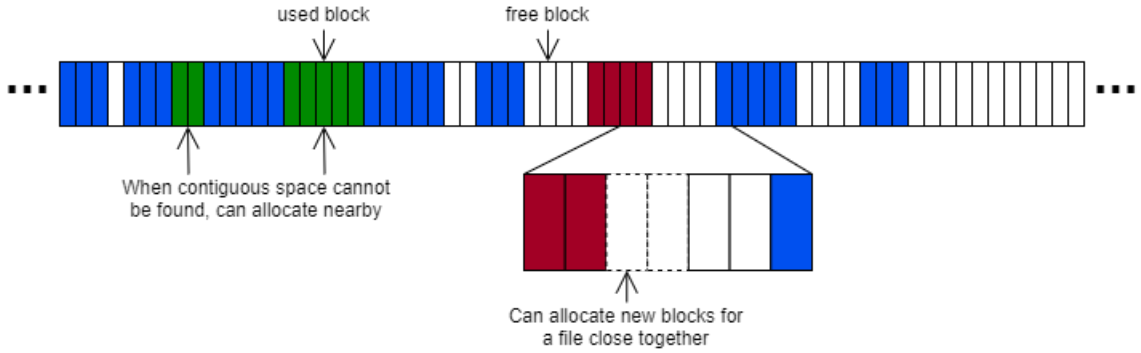
Free List



Linked list of blocks that are free. To allocate, take free blocks from the start of the list. To free, place block at the end of the list.

- Freeing and allocating blocks is fast($O(1)$), simply go to head/tail of the list.
- Files unlikely to be contiguously allocated. The location of an allocated block is effectively random, so when reading files, we must seek across many random locations on the disk which is slow.

Bitmap



Each bit represents a block at the same index.

- Uses little memory (single bit per entry).
- Can quickly determine contiguous blocks at locations (useful for placing file's blocks next to each other to reduce seek time when accessing).
-
- Highly optimised bit operations exist as instructions for most CPUs (allows for fast operations on the bitmap).
- May need to search the entire bitmap to find a free spot ($O(n)$ complexity) which is slow.

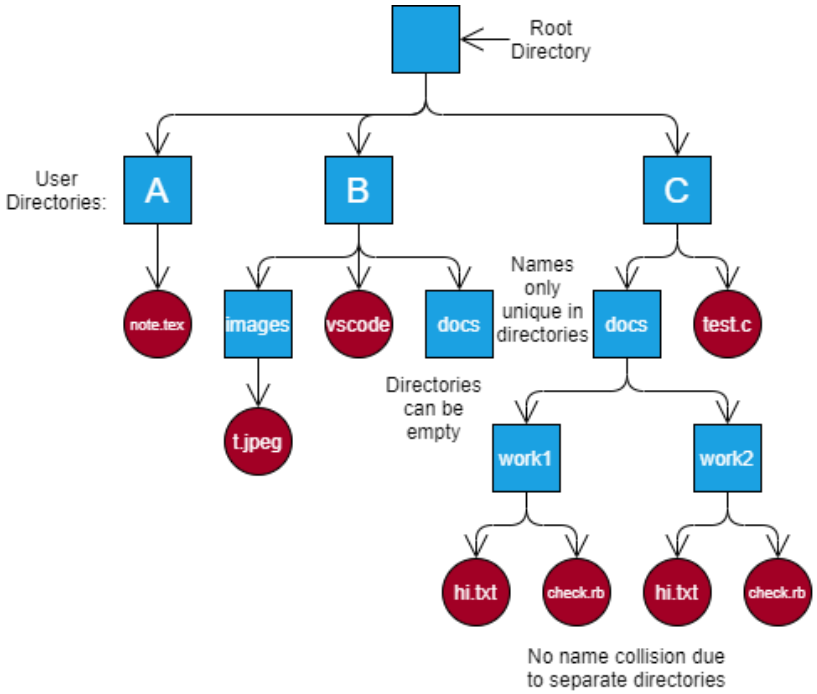
File System Layout

- **Boot Block** Used to store data to be loaded to start the OS.

- **Superblock** Contains file system information, including:
  - Number of inodes.
  - Number of data blocks.
  - Start of Inode & free space bitmaps.
  - Location of first data block.
  - Block size.
  - Maximum file size.

Filesystem Directory Organisation

- Map symbolic names (e.g "Lecture15.tex") to logical disk locations (e.g Disk 0, block 2354 (Logical Block Addressing)).
- Helps with file organisation.
- Prevents naming collisions (e.g names only unique inside a directory).



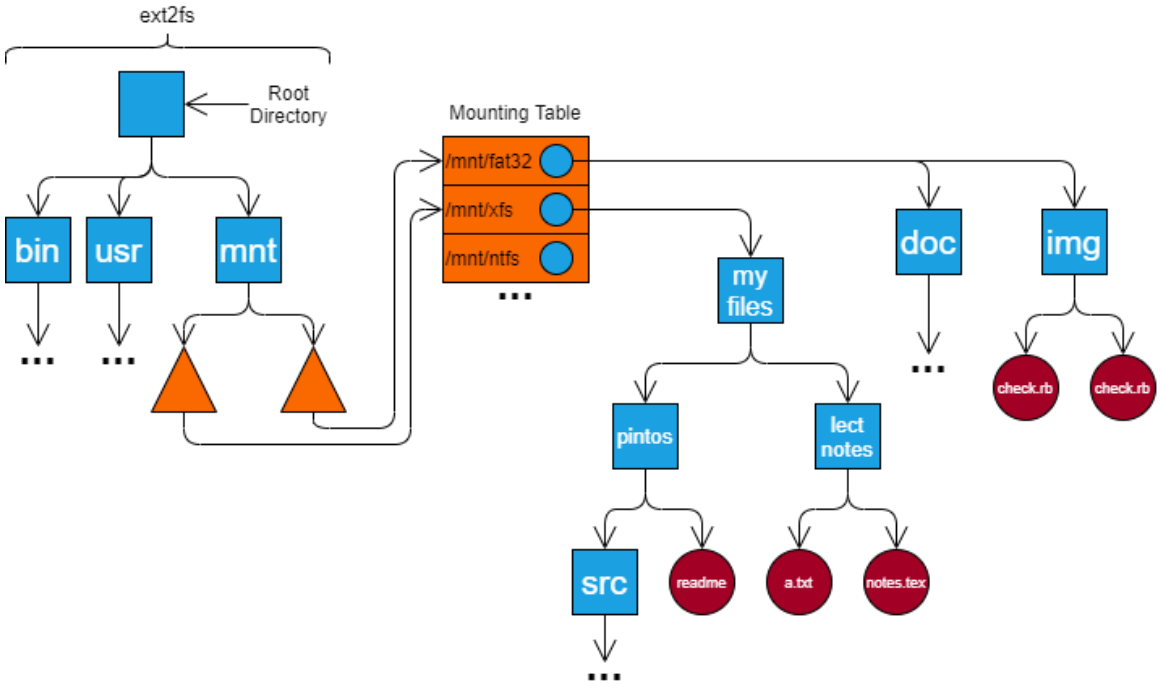
- **Pathnames**
  A path from the root directory to a file. Can be absolute (from root directory) or relative (from current/working directory). Delimited by "\" on Windows, and "/" on Linux/Unix.
- **Search**
  Can match pathnames with wildcards (e.g "*.pdf" (match all pdfs in current directory) or "lecture[0-9]/images" match all image directories from single digit lecture folders.)
- **Mount**
  Create a link to another file system such as a remote server or another disk (e.g usb drive).

  When mounting:

- Combine multiple filesystems into a single namespace (so can traverse through multiple).
- Allow reference from a single root directory (filesystem accessed as a directory from the root).
- Mount point is the folder from which the mounted filesystem can be accessed from the native one.
- Support soft links (not hard links as they are dependent of filesystem implementation).

This is managed using a mounting table, which records location of mount points and devices.

When a mount point is encountered the native filesystem consults the mounting table to determine which file system to go to.



- **Link** A reference to a directory/file in another part of the filesystem.
  - **Hard Link** Reference the address of the file (only allowed for files, not directories, in unix). A counter is used to check for when a file should be deleted (when no links).
  - **Soft/Symbolic Link** Reference the address using a directory entry (when deleting, leave links, raise an exception when the link is used).

  Must keep track of links to prevent looping when traversing the file system as a tree.

  Note that hard links are not allowed for directories as they may result in an island (where a hard link points to itself, or two mutually point to eachother) meaning they can never be deleted.

Unix/Linux Directory System

System Calls

```c
/* Create a new directory. */
#include <sys/stat.h>
int mkdir(const char *pathname, mode_t mode);

#include <fcntl.h> /* Definition of AT_* constants. */
#include <sys/stat.h>
int mkdirat(int dirfd, const char *pathname, mode_t mode);

/* Remove a directory (must be empty already). */
#include <unistd.h>
int rmdir(const char *pathname);

/* Create a new hard link to an existing file. */
#include <unistd.h>
int link(const char *oldpath, const char *newpath);

#include <fcntl.h> /* Definition of AT_* constants. */
#include <unistd.h>
int linkat(int olddirfd, const char *oldpath, int newdirfd, const char *newpath, int
    ↪ flags);

/* Remove a filename from the file system, if this is the last link, remove the
 * file.
 */
#include <unistd.h>
int unlink(const char *pathname);

#include <fcntl.h> /* Definition of AT_* constants. */
#include <unistd.h>
int unlinkat(int dirfd, const char *pathname, int flags);

/* Change working directory. */
#include <unistd.h>
int chdir(const char *path); /* Change using path. */
int fchdir(int fd);          /* Change based on file descriptor. */

/* Open a directory. */
#include <sys/types.h>
#include <dirent.h>

struct DIR {
    struct dirent ent;
    struct _WDIR *wdirp;
};

struct DIR *opendir(const char *name);
struct DIR *fdopendir(int fd);

/* Close a directory. */
#include <sys/types.h>
#include <dirent.h>

int closedir(struct DIR *dirp);

/* Read a directory. */
```

```c
#include <dirent.h>

/* Directory entry structure.  */
struct dirent {
    ino_t          d_ino;       /* Inode number. */
    off_t          d_off;       /* Offset to the next dirent. */
    unsigned short d_reclen;    /* Length of this record. */
    unsigned char  d_type;      /* Type of file; not supported by all file system
        ↪ types. */
    char           d_name[256]; /* Null-terminated filename. */
};

struct dirent *readdir(struct DIR *dirp);

/* Reset the position of directory stream DIRP to the beginning of the directory. */
#include <sys/types.h>
#include <dirent.h>

void rewinddir(struct DIR *dirp);
```

Directory Representation

Directories map symbolic names to inodes, these inodes can be other directories (typically only a single block) or files.