50004 - Operating Systems - Lecture 14

Oliver Killane

18/12/21

Lecture Recording

Lecture recording is available here

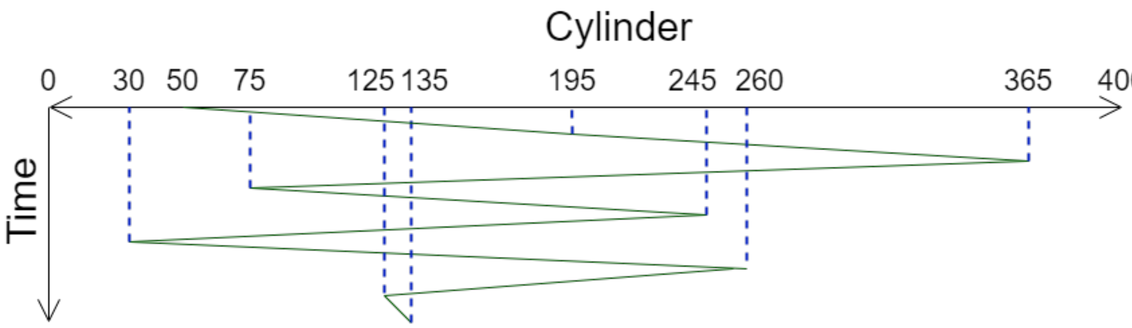
Disk Scheduling

(FCFS) First Come First Served

Requests completed in the order they were received.

- For light loads this is fine (time between requests is much larger than time taken to fulfil any request).
- Low performance for heavy loads (ends up traversing the disk more than optimal to get each request).
- Fair (no bais towards any process).

For example:
Head at 50. Queue: $195, 365, 75, 245, 30, 260, 125, 135$
Operations: $50 \to 195 \to 365 \to 75 \to 245 \to 30 \to 260 \to 125 \to 135$

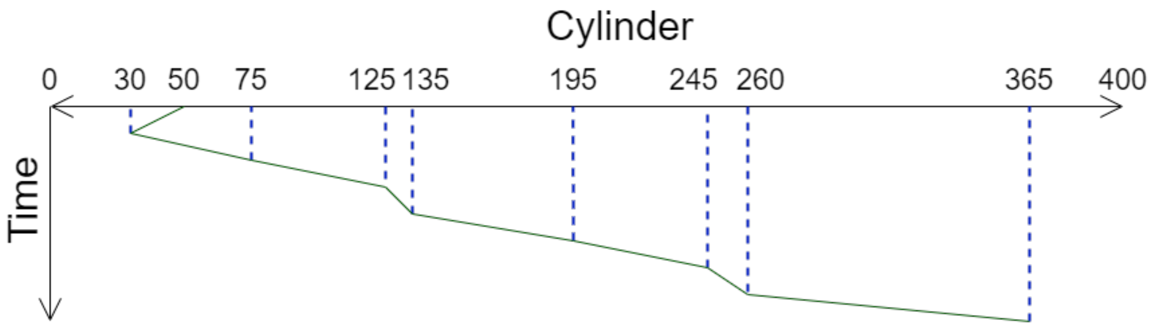


(SSTF) Shortest Seek Time First

Order requests according to shortest seek distance from the current head position

- Baised against innermost and outermost tracks (middle tracks on average closer, even with random head positions)
- Unpredictable performance (Requests may face a long delay if several very close requests are serviced)
- Processes can use dummy requests to keep control of head and have their requests prioritised.
- Can delay requests indefinitely (e.g many requests come in very close to eachother, trapping the head).

Head at 50. Queue: $195, 365, 75, 245, 30, 260, 125, 135$
Operations: $50 \to 30 \to 75 \to 125 \to 135 \to 195 \to 245 \to 260 \to 365$

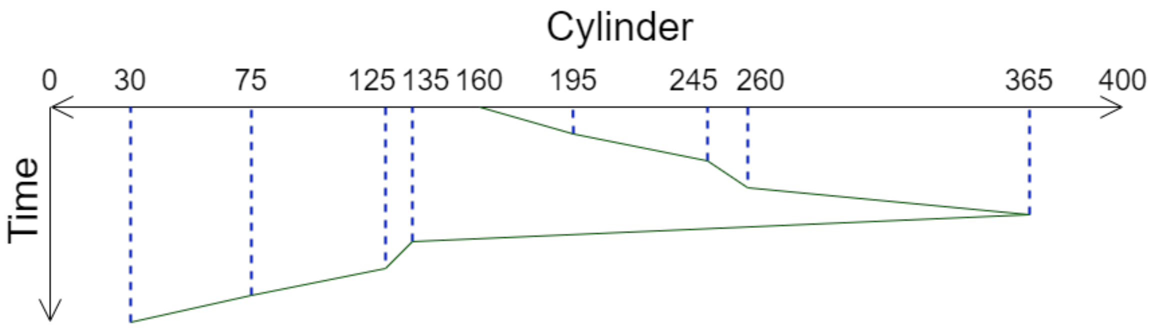


SCAN Scheduling

Also called elevator scheduling. Select requests with the shortest seek time in the preferred direction.

- Only change direction when at the outermost/innermost cylinder (no more requests in the direction)
- Long delays for requests that are not in the path of the algorithm (e.g come in just behind the head) and for the extremes.
- Base for the most common algorithms used.
- Suffers from same delay issue as **SSTF**, though reduced (only in one direction)

Head at 160. Queue: $195, 365, 75, 245, 30, 260, 125, 135$
Operations: $160 \rightarrow 195 \rightarrow 240 \rightarrow 260 \rightarrow 365 \rightarrow 135 \rightarrow 125 \rightarrow 75 \rightarrow 30$

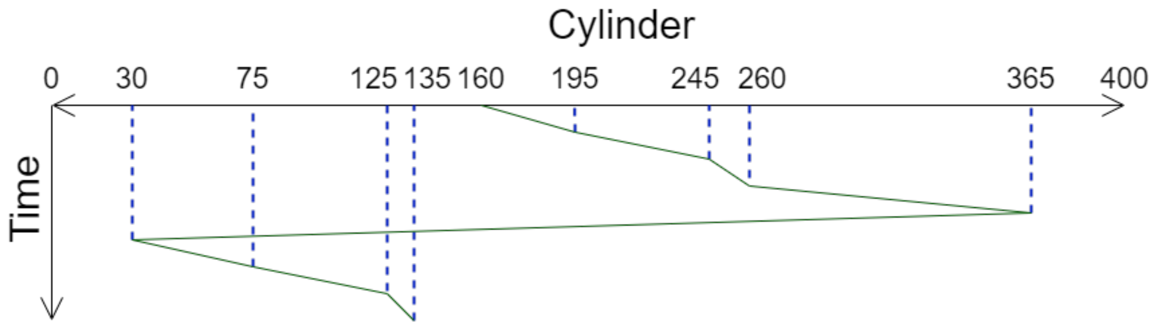


C-SCAN Scheduling

Much like **SCAN** but scanning in one direction only, jumping to the start (e.g innermost) when at the end (e.g outermost)

- Lower variance of requests on extreme tracks
- Largely reduces issue of indefinite wait from **SSTF**.

Head at 160. Queue: $195, 365, 75, 245, 30, 260, 125, 135$
Operations: $160 \rightarrow 195 \rightarrow 240 \rightarrow 260 \rightarrow 365 \rightarrow 30 \rightarrow 75 \rightarrow 125 \rightarrow 135$

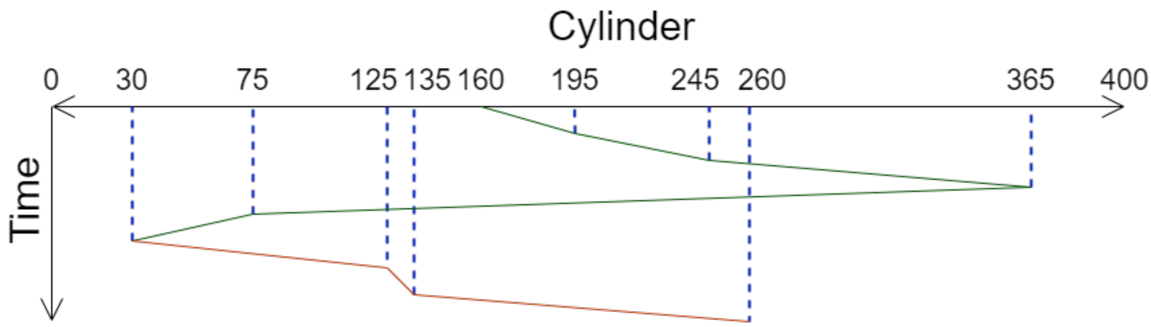


N-Step SCAN Scheduling

Same as **SCAN**, however only services requests waiting when the sweep began (for each sweep)

- Requests arriving during sweep are serviced before end of the sweep (no long waits)
- No indefinite waits possible.

Head at 160. Queue: $195, 365, 75, 245, 30$ Then $260, 125, 135$
Operations: $160 \to 195 \to 245 \to 365 \to 75 \to 30 \to$ (*new sweep*) $125 \to 135 \to 260$



Linux Disk Scheduling

struct bio

Can be found here in the linux kernel.

```c
/* main unit of I/O for the block layer and lower layers (ie drivers and
 * stacking drivers)
 */
struct bio {
        struct bio          *bi_next;   /* request queue link */
        struct block_device     *bi_bdev;
        unsigned int         bi_opf;    /* bottom bits req flags, top bits REQ_OP.
                                         * Use accessors.
                                         */
        unsigned short       bi_flags;  /* BIO_* below */
        unsigned short       bi_ioprio;
        unsigned short       bi_write_hint;
        blk_status_t         bi_status;
        atomic_t             __bi_remaining;
        struct bvec_iter     bi_iter;
        bio_end_io_t        *bi_end_io;
        void                *bi_private;
#ifdef CONFIG_BLK_CGROUP

        /* Represents the association of the css and request_queue for the bio.
         * If a bio goes direct to device, it will not have a blkg as it will
         * not have a request_queue associated with it.  The reference is put
         * on release of the bio.
         */
        struct blkcg_gq         *bi_blkg;
        struct bio_issue        bi_issue;
#ifdef CONFIG_BLK_CGROUP_IOCOST
        u64                     bi_iocost_cost;
#endif
#endif

#ifdef CONFIG_BLK_INLINE_ENCRYPTION
        struct bio_crypt_ctx    *bi_crypt_context;
#endif

        union {
#if defined(CONFIG_BLK_DEV_INTEGRITY)
                struct bio_integrity_payload *bi_integrity; /* data integrity */
#endif
        };

        unsigned short bi_vcnt; /* how many bio_vec's */

        /* Everything starting with bi_max_vecs will be preserved by bio_reset()
         */
        unsigned short  bi_max_vecs;    /* max bvl_vecs we can hold */
    atomic_t        __bi_cnt;   /* pin count */
        struct bio_vec *bi_io_vec;      /* the actual vec list */
        struct bio_set *bi_pool;

        /* We can inline a number of vecs at the end of the bio, to avoid
         * double allocations for a small number of bio_vecs. This member
         * MUST obviously be kept at the very end of the bio.
         */
        struct bio_vec          bi_inline_vecs[];
};
```

- **I/O Requests added to request list**
  There is one request list per device, a **bio** that keeps track of the pages associated with the request.

- **Block Device drivers define a request operation for the kernel to use**
  Interface provided by function pointers.

  - Kernel passes ordered request list to driver.
  - Driver completes all requests in the list.
  - Drivers use the read/write operations defined by the kernel (uniform interface fro kernel to send uniform request types to many different drivers).

- **Driver Based ordering**
  Some drivers bypass kernel ordering and do it themselves (e.g **RAID**). This is done for more complex disk setups where assumptions made by the kernel ordering algorithm do not apply.

- **Default Algorithm: SCAN Scheduling**
  Kernel attempts to merge requests to adjacent blocks (grouping adjacent requests results in less seek time, higher request throughput)

  However read requests may starve during very large writes (from merging), if these are done synchronously, a program may hang.

- **Deadline Scheduler**
  Ensures that reads are performed before a set deadline to prevent long or indefinite waits. (Elimintates read request starvation)

- **Anticipatory Scheduler**
  Delay after read requests completed. If a process sends a second synchronous read request, it will be attended to quickly (due to delay after completing the first).

  - Reduces Excessive seeking (Second request will likely be very close to the first, so avoid seeking away, and then back)
  - Can reduce throughput (if no more read requests are sent by the process during the delay, or if the request sent requires a large seek)

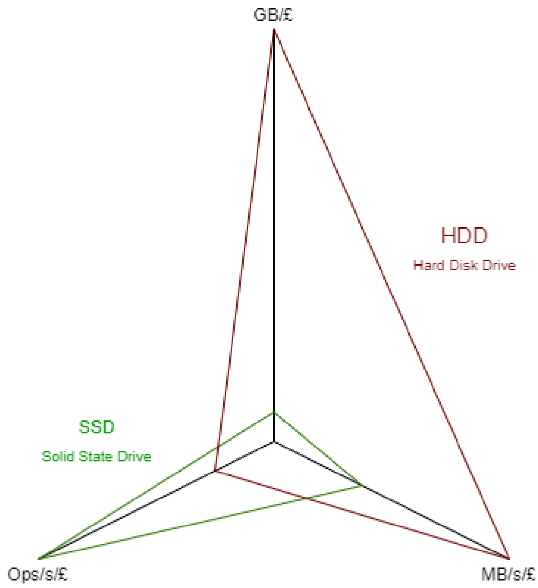
Solid State Drives

SSD Scheduling

Scheduling for SSDs do not require scheduling algorithms as many memory modules can be read/written in parallel and write/read speed is approximately constant.

However drivers need to overcome issues with limited writes, tracking virtual to physical blocks & assignment of free blocks.

- Very high bandwidth (e.g 1GB/s SSD vs 100MB/s HDD)
- Lower latency
- High parallelism



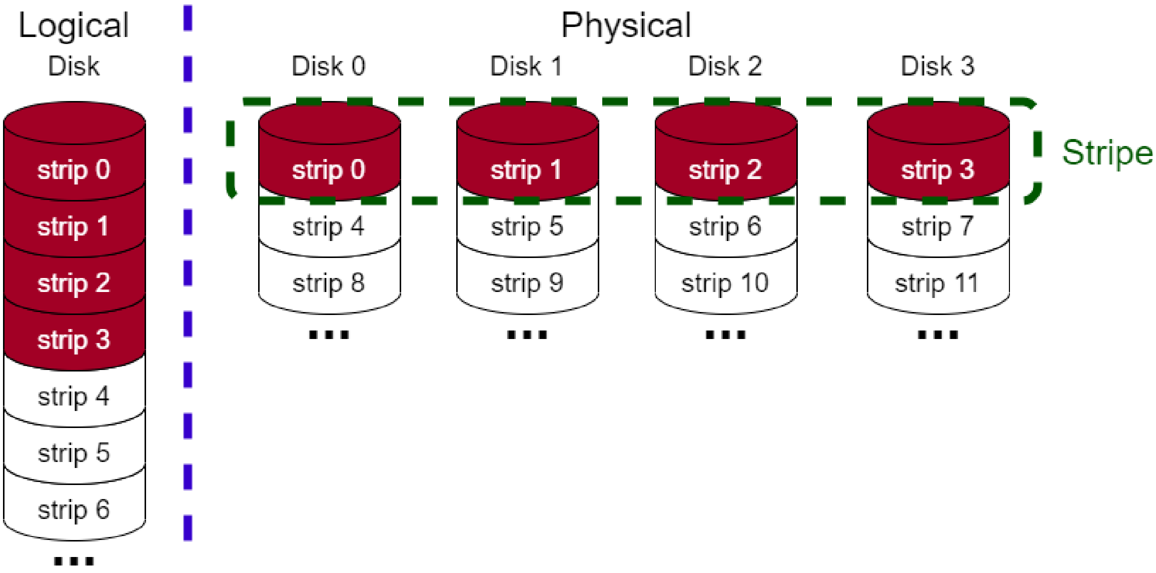
RAID

Disk performance has not kept pace with CPU performance, creating a bottleneck. Redundant Array of Inexpensive Disks (**RAID**) increases disk based system performance by using many disks in parallel.

- An array fo physical drives appears as a single virtual drive.
- Distributes stored data over physical disks to allow parallel operation, improving performance.
- Use redundant disk capacity to respond to disk failure (more disks means lower mean-time-to-failure, more disks, higher chance a single disk in the group fails).

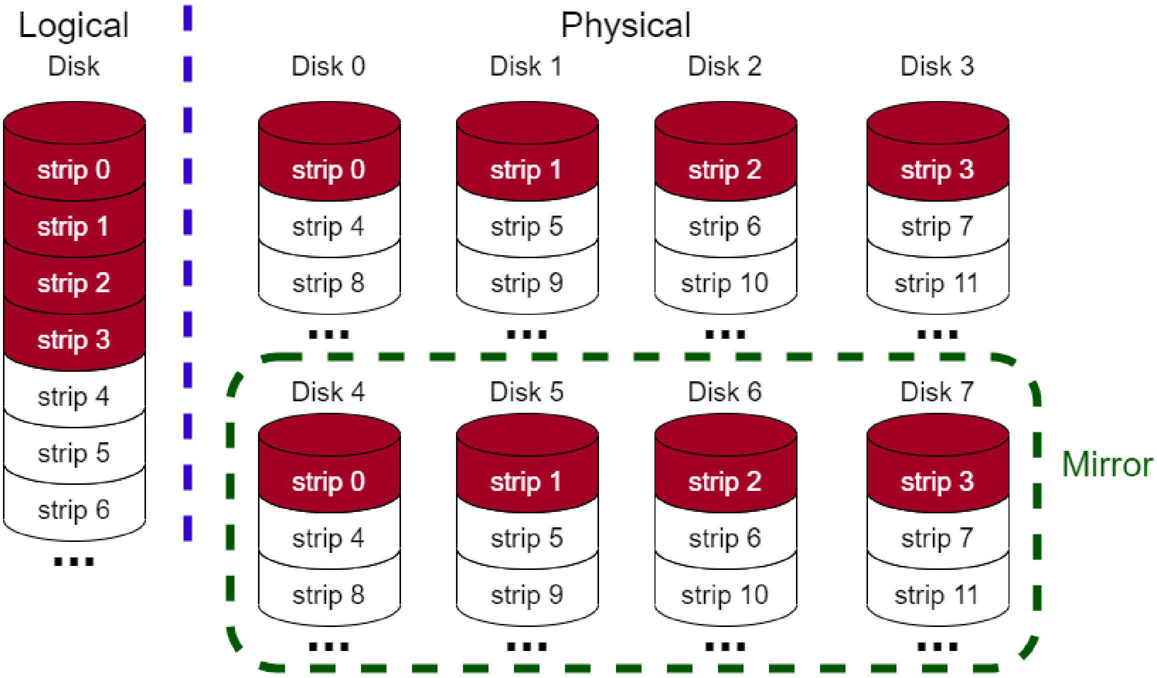
There are several levels of **RAID**, each with differing performance, redundancy and space-efficiency/cost.

RAID 0 - Striping



- Spread blocks in round robin fashion across disks.
- Can concurrently seek/transfer data (provided blocks on different physical disks).
- Can balance load across disks (sometimes).
- No redundancy (any disk failure will result in data being lost).

RAID 1 - Mirroring



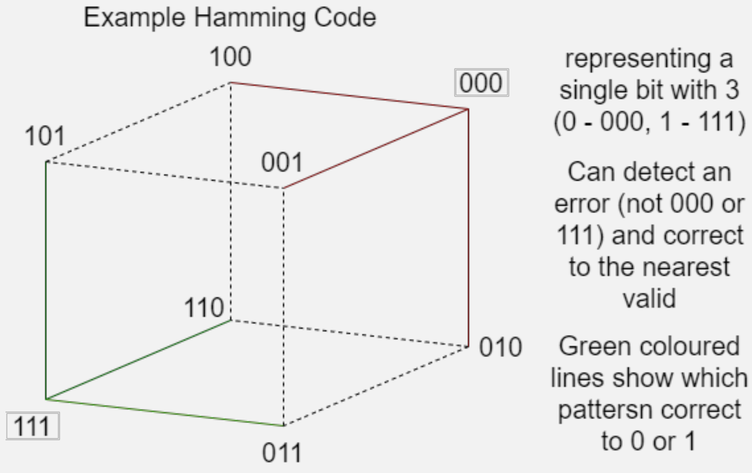
Mirror Data across disks to increase redundancy.

- Reads can be serviced by either disk (e.g can read $0, 1, 2, 3, 4, 5, 6, 7$ at the same time).
- Writes must update both disks in parallel (e.g can effectively only write to 4 disks at a time). (Slower)
- Failure recovery is easy (when a disk fails, use its mirror).
- Low space efficiency and hence high cost (store everything twice).

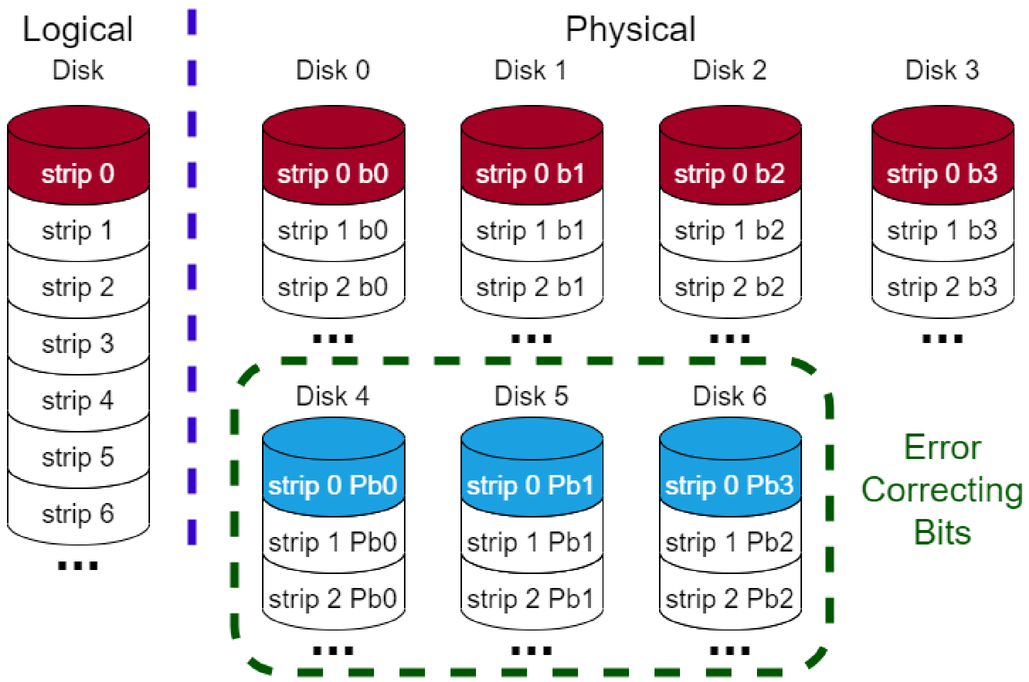
RAID 2 - Bit-Level Hamming

Hamming Codes

A family of error correcting codes that make use of hamming distance (number of bits different between two patterns) to correct single bit errors.



More complex codes can be used to check larger bit patterns with very few bits. The general algorithms can be seen here

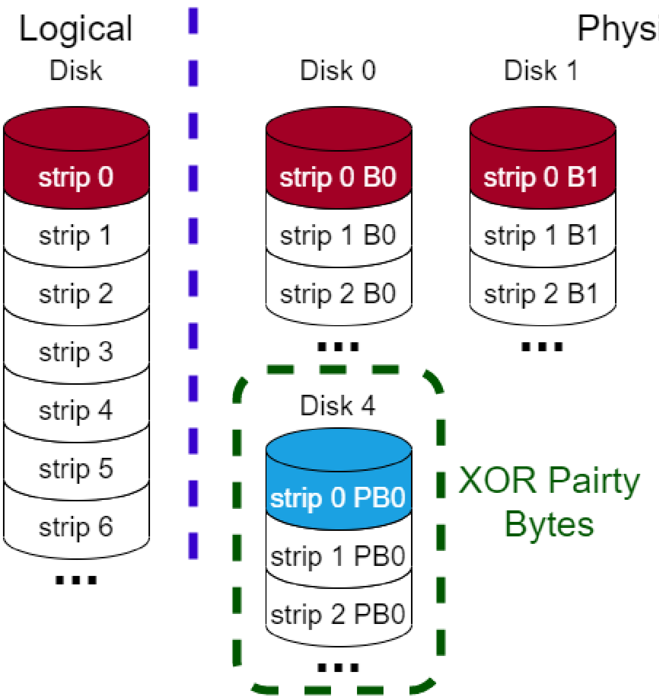


Parallel access by striping at the bit level.

- Consecutive (in this case 4) bits are read in parallel, hence very high throughput (always reading/writing in parallel).
- Hamming error-correcting codes used to detect and correct single bit errors (can detect but not correct double bit errors).
- Cannot process requests in parallel (each request requires all disks, no concurrency).
- High storage overhead (less space efficient, increasing cost).
- Large number of writes as the error correcting codes must be updated (can become a bottleneck) & every disk must write for every write operation.

RAID 3 - Byte-Level XOR

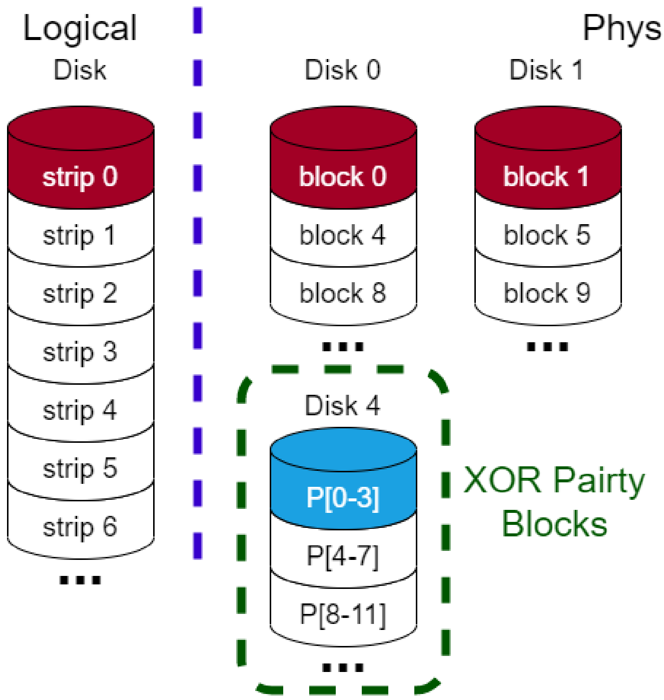
A single parity strip used (data on other disks XORed together).



Can reconstruct missing data from parity and remaining data when a disk fails (disk reports).

- Easy to reconstruct data when a disk fails
- More space efficient than **RAID 2**
- Only one I/O request at a time (but each request can be read in parallel from all disks)

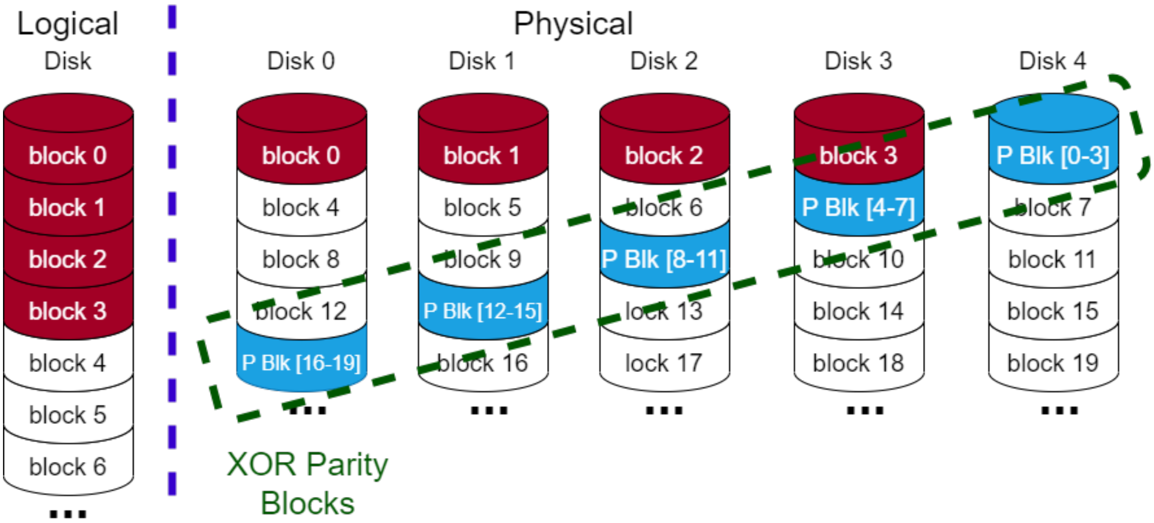
RAID 4 - Block Level XOR



Parity Strip is XOR over blocks, allowing entire blocks to be accessed independently on different disks.

- Allows read requests to be serviced concurrently.
- Low redundancy overhead.
- Parity must be updated after every single write, Parity disk becomes a bottlenecks

RAID 5 - Block Level Distributed XOR



The most commonly used **RAID** level, by distributing parity there is potential for concurrency.

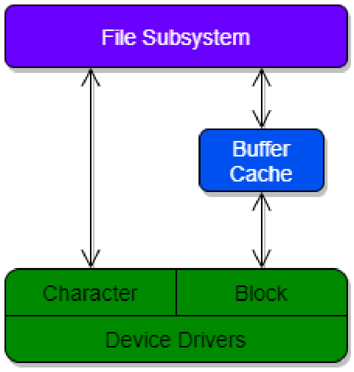
- Some potential for write concurrency as parities are on different disks.
- Good storage/efficiency tradeoff.
- Reconstruction of disk is non-trivial (and slow).

RAID Level summary

Speeds compared with using a single disk.

Category	Level	Description	I/O Data Transfer		I/O Request Rate	
			Read	Write	Read	Write
Striping	0	Non-Redundant	↑	↑	↑	↑
Mirroring	1	Mirrored	↑	=	↑	=
Parallel Access	2	Redundant (Hamming ECC)	↑↑	↑↑	=	=
	3	Redundant (Bit Interleaved Parity)	↑↑	↑↑	=	=
Independent Access	4	Block interleaved Parity	↑	↓	↑	↓
	5	Block interleaved distributed parity	↑	↓	↑	= or ↓

Disk Caching



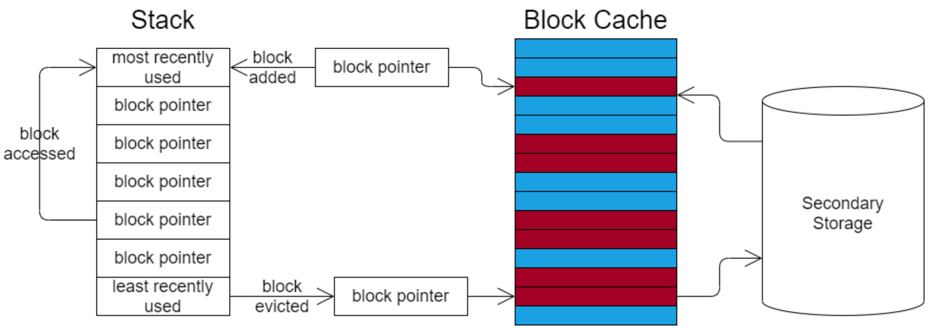
We can cache sectors of disk in main memory to reduce access times.

- Buffer contains copies of disk sectors.
- OS manages disk in terms of **blocks** which are likely much larger than sectors (so loads multiple sectors).
- Must ensure contents are saved in case of failure (e.g lazy writing is very complex).
- Cahce has finite space, so a replacement policy must be implemented.

(LRU) Least Recently Used

Replace the block that was in cache longest with no references.

Cache is a stack of pointers to blocks in memory, when a block is referenced, it is pushed to the top of the stack. Replacement evicts the block at the bottom of the stack.

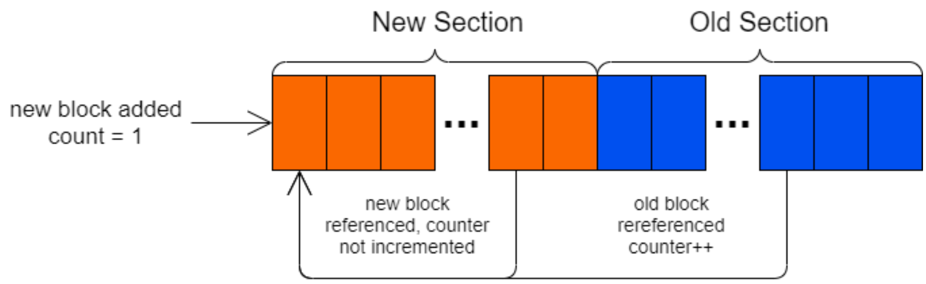


This replacement policy does not track how many times a block is accessed, only the relative time of the last access (through stack position).

(LFU) Least Frequently Used

- Replace the block with the fewest references.
- Each block has a counter (incremented when referenced), the lowest counter block is evicted during replacement.
- To avoid this, can use frequency based replacement.

Frequency-Based Replacement



To prevent items that get a sudden burst of accesses from lingering for a long time in the cache, we only increment the count when a block is shuffled out of the **new** section.

File Systems

File systems organise information. Their main objectives:

- **Non-volatile, long term storage**
- **Sharing information** e.g compilers, applications, text data etc.
- **Concurrent Access** Many programs can access many files pesudo-simultaneously
- **Convenient Organisation** symbolic names, directories
- **Easy management of data** automatic backup, snapshots etc.
- **Security** Permissions, read/write, hidden files

File Naming

A **file** is an arbitrary size collection of data, extensions allow for easy identification of type from an identifier:

Type	Extension	Function
executable	exe,com,bin,no extension	read & load to run machine code program.
object	obj, o	compiled machine language, not linked.
source code	c, cpp, java, rs, py, s	Source code, extension identifies language.
batch/bash	bat (windows), sh (unix)	Script of commands to be interpreted by the terminal.
text	txt	Text data (usually **ascii**).
library	lib, a, so, dll	Libraries that programs can link to.
archive	arc, zip, tar	Compressed archives (for transmission or storage).

There are also several types of file.

Type	Description
Hard Links	A file that aliases the data of another file (refers to the specific location of the data).
Soft Links	A soft/symbolic link aliases the path to a file by another (e.g file points to another file, which in turn points to its data). The **ln** command can be used to soft link files in unix based systems.
Regular	Normal file as discussed in the table above.
Directory	A collection of files, that can itself be added to directories.
Character Special	A file that provides access to a character I/O device
Block Special	Same as a character special file, but for block devices.

Filesystem support functions

- **Name Translation** Converting paths to disks & blocks (logical) for use by the driver.
- **Management of Disk Space** Allocating and deallocating storage for files.
- **File locking for exclusive access** Important when many programs may access a fiule concurrently (can fuse the **fcntl** syscall with the **F_SETLK**,**F_SETLKW**, and **F_GETLK**)
- **Performance Optimisation** Caching/Buffering
- **Protection against system failure** Backup/Restore in case of a crash/power failure/unexpected shutdown
- **Security** Enforcing file permissions

File Attributes

- **Basic**
  - **File Name** Symbolic name, unique in directory
  - **File Type** e.g text, binary, executable, directory
  - **File Organisation** placement of contents in blocks (sequential, random)
  - **File Creator** program that created the file
- **Address information**
  - **Volume** Disk drive, partition
  - **Start Address** cylinder, head, sector (logical block addressing information)
  - **Size Used**
  - **Size Allocated**
- **Access Control Information**
  - **Owner** User that controls the file
  - **Authentication** Locked files may need a password/key
  - **permitted actions** e.g read/write, dlete permissions (owner/others)
- **Usage Information** Metadata for paper-trail.
  - **Creation Timestamp** Date & Time
  - **Last Modified** (can include the user that made the modification)
  - **Last Read**
  - **Last Archived** For keeping track of backups
  - **Expiry Date** For automatic deletion (e.g recycle bin contents)
  - **Access activity** Metadata for reads/writes etc. (can be used in improving performance)

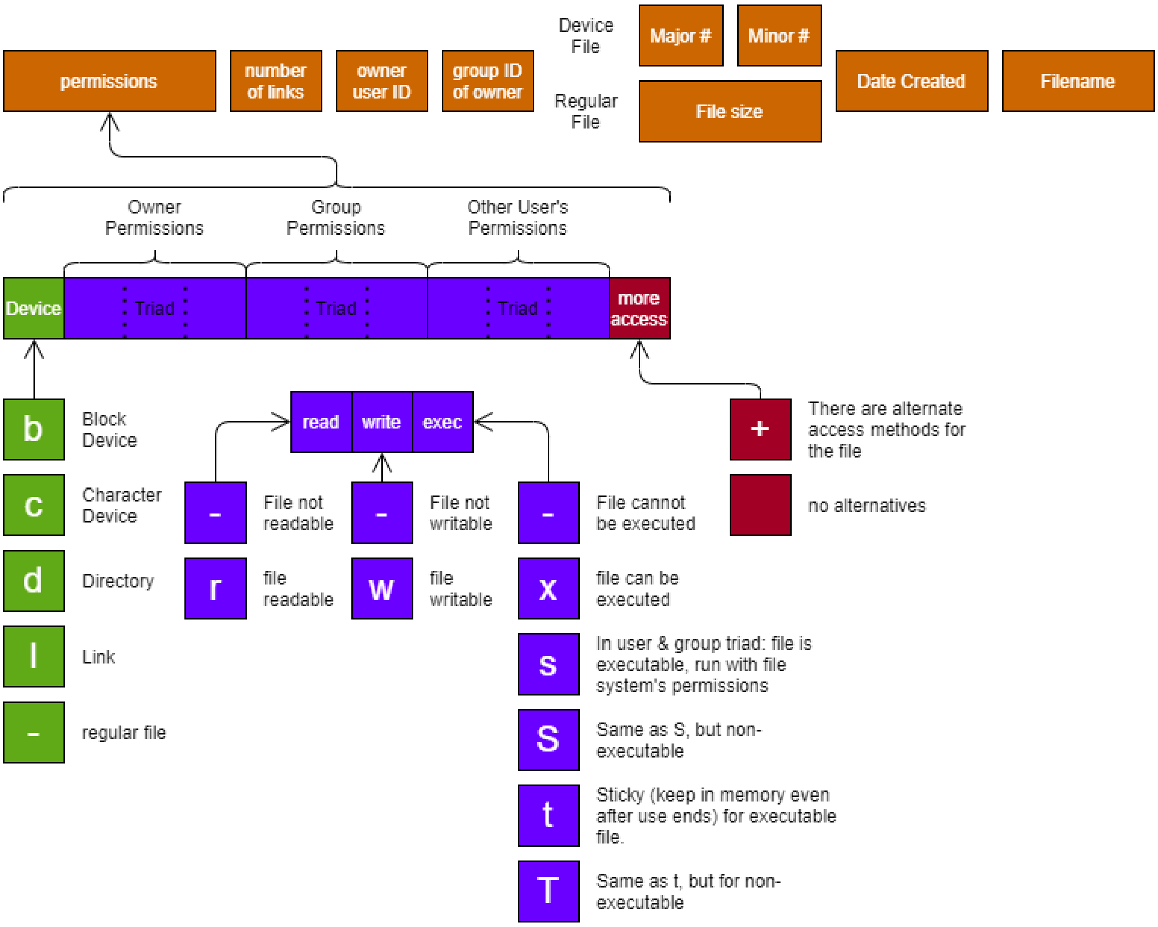
Unix/Linux Files

Common Filesystem Syscalls

```c
/* UNISTD functions (header for POSIX api) */
#include <unistd.h>
/* Close a file */
int close(int fd);

/* Read COUNT bytes from file FD into buffer BUF. Return number of bytes read. */
ssize_t read(int fd, void *buf, size_t count);

/* Write to file FD, COUNT bytes from BUF. Return number of bytes written. */
ssize_t write(int fd, const void *buf, size_t count);

```

```c
/* Seek to the given OFFSET in file FD, options in WHENCE. "lseek" stands for
 * long seek (offset is 64 bits for large files).
 */
off_t lseek(int fd, off_t offset, int whence);

/* Return file information for file at PATHNAME into buffer STATBUF. */
int stat(const char *restrict pathname, struct stat *restrict statbuf);

struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number 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;     /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
     *  precision for the following timestamp fields.
     *  For the details before Linux 2.6, see NOTES.
     */

    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */

    #define st_atime st_atim.tv_sec      /* Backward compatibility */
    #define st_mtime st_mtim.tv_sec
    #define st_ctime st_ctim.tv_sec
};


/* File Control functions */
#include <fcntl.h>

/* Note that interestingly, despite close being unistd, open is not. This is as
 * many of the flags required for open are declared in FCNTL, and the developers
 * wanted to avoid polluting UNISTD with these.
 */

/* Open a file at PATHNAME with FLAGS to get a file descriptor. */
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

/* File controls (e.g lock, set read status). */
int fcntl(int fd, int cmd, ... /* arg */ );
```

File Attributes




```
nolinks# For a device:
# <permission attributes> <no. links> <owner> <group> <major> <minor> <creation date> <filename>

crw------- 1 root root   5,  1 Dec 20 14:59 console
# Character device, readable and writeable, but not executable by the owner

brw------- 1 root root   1, 15 Dec 20 14:59 ram15
# Block device, readable and writeable, but not executable by the owner

# For a file:
# <permission attributes> <no. links> <owner> <group> <size> <creation date> <filename>

-rwxrwxrwx 1 oliverkillane oliverkillane  21775 Dec 20 16:31 'Lecture 14.tex'
# File, readable, writable and can be executed by any user
```

File System Organisation

Space Allocation

File size is variable (can increase or decrease) and is allocated on the disk in blocks (usually 512 → 8192 bytes). The block size is determined by the file system.

Block Size too Small

- High overhead for managing large files (large number of blocks to keep track of)
- High file transfer time (large file → lots of blocks across the disk → lots of seeking back and forth)

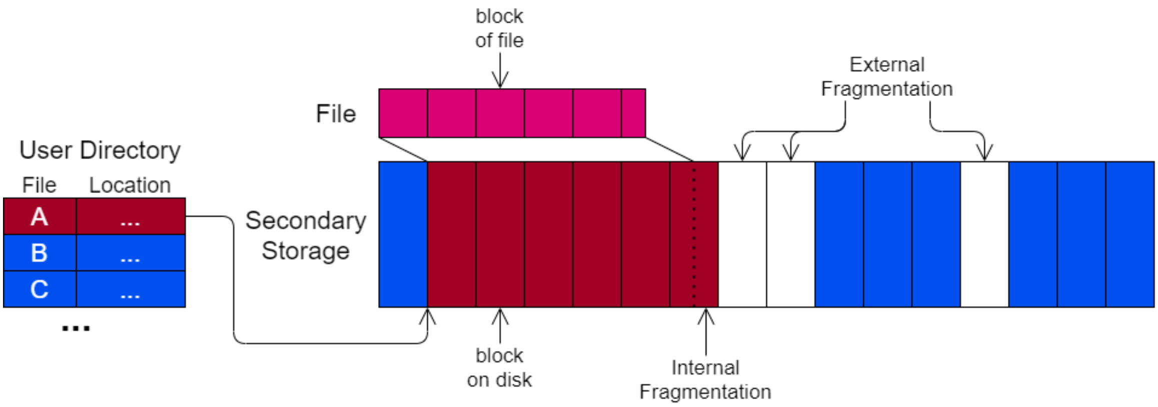
Block Size too Large

- Internal fragmentation (Files leave parts of large blocks unused - wasteful)
- Small files waste lots of space.
- Caching is based on blocks, so end up having a very large cache space, or unable to cache many blocks.

Contiguous File Allocation

Place file data on a contiguous stretch of addresses on storage device. Similar to segment based memory management.

- Successive logical records are usually physically adjacent, reducing seek time when reading the file (seek to start, then just traverse file).
- External Fragmentation can occur (unused blocks between files).
- If unable to allocate new blocks as a file grows, must transfer to new large free section (expensive, requires lots of I/O).

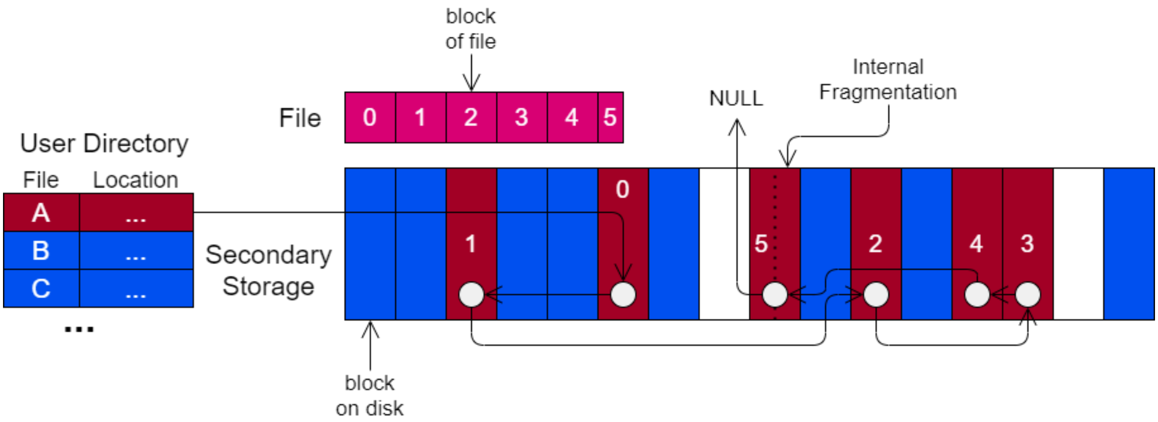


Block Linkage/Chaining

Each file contains a linked list of blocks. When locating a block, traverse the linked list to by seeking to block, reading pointer.

- Can grow files without expensive re-allocation (just set pointer of end block to point to a newly allocated one).
- No external fragmentation (as all blocks can be used in any order, by any file).
- Space used for pointer in each block.
- Traversing files requires lots of seeking from block to block.

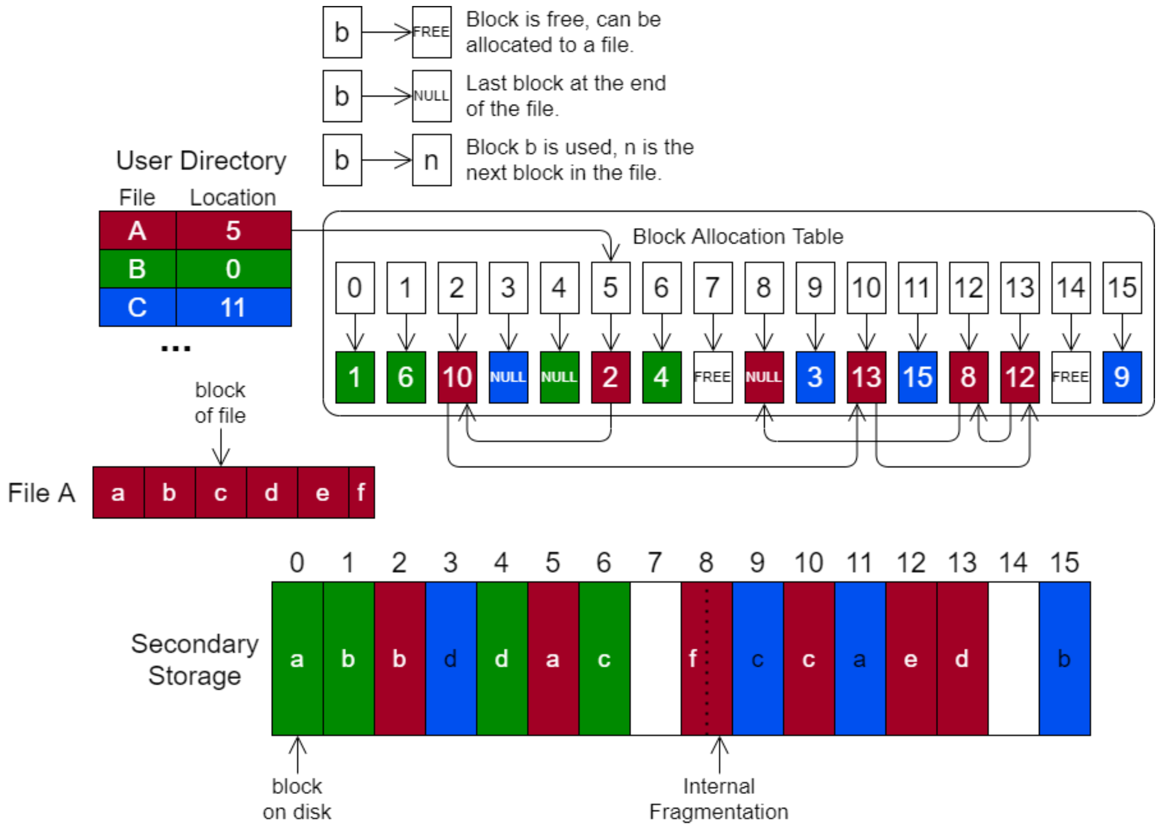
- For small block sizes the number of seeks to traverse a file increases.



Block Allocation Table

Uses a directory mapping files to first block. Table maps blocks to the next block in the file, indicating free spots (**FREE**) and block with no next (end of file - **NULL**).

- File allocation table can be cached in memory for fast lookup.
- Does not require lengthy seeks to traverse the block numbers for the file.
- No external fragmentation.
- However files can become fragmented (spread across disk) which reduces read/write speed (e.g must do lots of seeks to read the whole file), so should be periodically defragmented (disk reorganised to place files blocks next to eachother, very expensive operation).
- Table can become impractically large, using up lots of memory, and more than one block of storage.

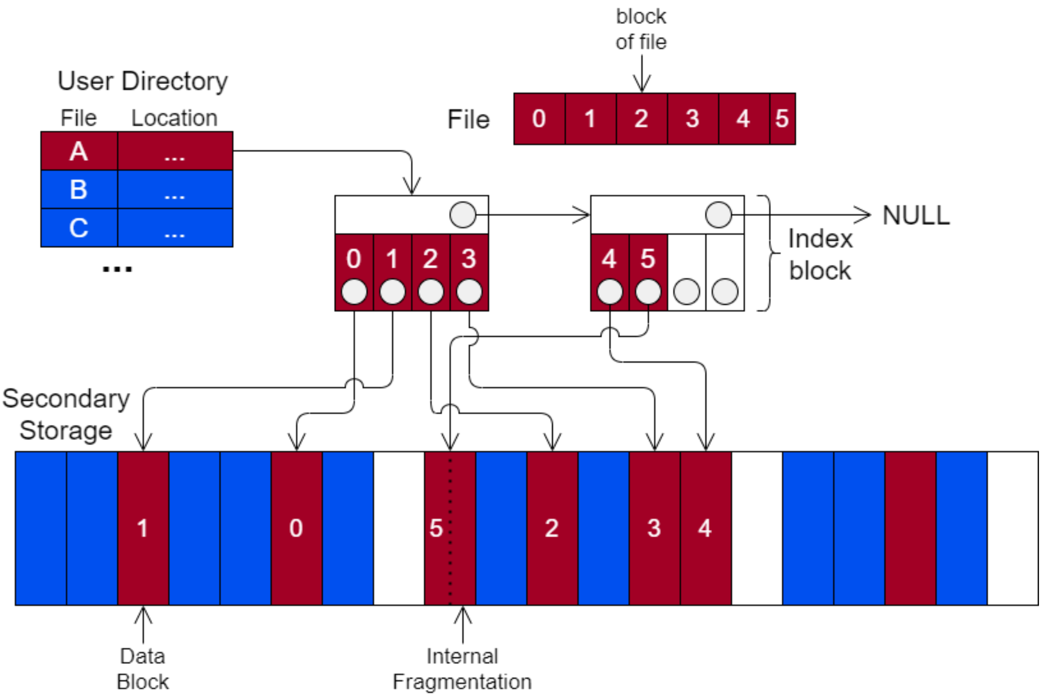


This system is used for **microsoft**'s **FAT16/32** file system (with the table cached in memory).

Index Blocks

Each table has one or more index blocks. Index blocks contain an array of pointers to data blocks, and pointers to subsequent index blocks. Basically page tables for file systems. The example below has miniscule index blocks for illustration purposes.

- Can search through index blocks to find location fo file blocks easily.
- No external fragmentation & can extend files easily.
- Index blocks can be cached in memory just like data blocks.
- Index blocks are per-file, hence can load file's table, rather than have an enormous global table as with **FAT**



Linux Inodes

Linux/Unix uses the **Index Blocks** strategy through **inodes** (index nodes).

Inode contain:

- Type & Access control.
- Number of links to that inode.
- User & Group ID.
- Access & Modification time.
- Inode change time (e.g when permissions, data stored in inode were changed).
- Direct, Indirect, Double Indirect & Triple Indirect pointers to data blocks.

The **inode** struct can be found here in the linux kernel.

```
struct inode {
        umode_t         i_mode;
        unsigned short  i_opflags;
        kuid_t          i_uid;
        kgid_t          i_gid;
        unsigned int    i_flags;

#ifdef CONFIG_FS_POSIX_ACL
        struct posix_acl        *i_acl;
        struct posix_acl        *i_default_acl;
#endif

        const struct inode_operations   *i_op;
        struct super_block              *i_sb;
        struct address_space            *i_mapping;
```

```c

#ifdef CONFIG_SECURITY
	void			*i_security;
#endif

	/* Stat data, not accessed from path walking */
	unsigned long		i_ino;
	/*
	 * Filesystems may only read i_nlink directly.  They shall use the
	 * following functions for modification:
	 *
	 *    (set|clear|inc|drop)_nlink
	 *    inode_(inc|dec)_link_count
	 */
	union {
		const unsigned int i_nlink;
		unsigned int __i_nlink;
	};
	dev_t			i_rdev;
	loff_t			i_size;
	struct timespec64	i_atime;
	struct timespec64	i_mtime;
	struct timespec64	i_ctime;
	spinlock_t		i_lock;	/* i_blocks, i_bytes, maybe i_size */
	unsigned short          i_bytes;
	u8			i_blkbits;
	u8			i_write_hint;
	blkcnt_t		i_blocks;

#ifdef __NEED_I_SIZE_ORDERED
	seqcount_t		i_size_seqcount;
#endif

	/* Misc */
	unsigned long		i_state;
	struct rw_semaphore	i_rwsem;

	unsigned long		dirtied_when;	/* jiffies of first dirtying */
	unsigned long		dirtied_time_when;

	struct hlist_node	i_hash;
	struct list_head	i_io_list;	/* backing dev IO list */
#ifdef CONFIG_CGROUP_WRITEBACK
	struct bdi_writeback	*i_wb;		/* the associated cgroup wb */

	/* foreign inode detection, see wbc_detach_inode() */
	int			i_wb_frn_winner;
	u16			i_wb_frn_avg_time;
	u16			i_wb_frn_history;
#endif
	struct list_head	i_lru;		/* inode LRU list */
	struct list_head	i_sb_list;
	struct list_head	i_wb_list;	/* backing dev writeback list */
	union {
		struct hlist_head	i_dentry;
		struct rcu_head		i_rcu;
	};
	atomic64_t		i_version;
	atomic64_t		i_sequence; /* see futex */
	atomic_t		i_count;
```

```
	atomic_t	i_dio_count;
	atomic_t	i_writecount;
#if defined(CONFIG_IMA) || defined(CONFIG_FILE_LOCKING)
	atomic_t		i_readcount; /* struct files open RO */
#endif
	union {
		const struct file_operations	*i_fop;	/* former ->i_op->default_file_ops */
		void (*free_inode)(struct inode *);
	};
	struct file_lock_context	*i_flctx;
	struct address_space	i_data;
	struct list_head	i_devices;
	union {
		struct pipe_inode_info	*i_pipe;
		struct cdev		*i_cdev;
		char			*i_link;
		unsigned		i_dir_seq;
	};

	__u32			i_generation;

#ifdef CONFIG_FSNOTIFY
	__u32			i_fsnotify_mask; /* all events this inode cares about */
	struct fsnotify_mark_connector __rcu	*i_fsnotify_marks;
#endif

#ifdef CONFIG_FS_ENCRYPTION
	struct fscrypt_info	*i_crypt_info;
#endif

#ifdef CONFIG_FS_VERITY
	struct fsverity_info	*i_verity_info;
#endif

	void			*i_private; /* fs or device private pointer */
} __randomize_layout;
```