# The UNIX Time-Sharing System

Dennis M. Ritchie and Ken Thompson, Bell Laboratories, 1974

Full Paper

## Abstract

UNIX is a general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP-11/40 and 11/45 computers. It offers a number of features seldom found even in larger operating systems, including: (1) a hierarchical file system incorporating demountable volumes; (2) compatible file, device, and inter-process I/O; (3) the ability to initiate asynchronous processes; (4) system command language selectable on a per-user basis; and (5) over 100 subsystems including a dozen languages. This paper discusses the nature and implementation of the file system and of the user command interface.

## Historical Context - Hardware

- Mainframe was king
- First "minicomputers" e.g. PDP/11 series
- Cost < $25,000 !!!!!! (=$165K in 2019 $)
- PDP 11/45
  - 16 bit word, 144K RAM (UNIX – 42K)
  - 1M fixed hard disk, + disk cartridges, tapes
- Popular for academics and small research labs
- Went out of style when single chip CPU's made PC's possible, starting in 1971

## Historical Impact

- These days, seems like "No big deal"
  - We expect all modern operating systems to contain this stuff
- OS Historical Context – Late 1960's
  - OS/360 MVS – most ambitious OS of its time – large, late, and buggy
  - MULTICS – Collaboration between MIT, GE, Bell Labs
    - Lots of new exciting features, but also late and buggy
    - First OS written in a high-level language
    - Ritchie and Thompson both worked on MULTICS
- UNIX family is now the single most popular OS in the world

"Perhaps paradoxically, the success of UNIX is largely due to the fact that it was not designed to meet any predefined objectives."

"...the users of UNIX will find that the most important characteristics of the system are its simplicity, elegance, and ease of use."

elegance - the quality of being pleasingly ingenious and simple; neatness.

## Digression: Really Good Software

From a presentation by Melinda Varian, 1991 on VM

1. can be written only by very small groups of very skilled programmers
2. a labor of love
3. the author himself needs it
4. never finished… if it doesn't grow, it decays
5. must be "hacked at"
6. intelligent, adventurous user community with influence

"All of us loyal VMers know that UNIX is ugly and cryptic and hard to use compared to CMS, and the lack of security in UNIX systems is appalling."

## Processes and Images

## User/Application view of Processes

- pid = fork(label)
- filep = pipe()
- execute(file, $arg_0$, $arg_1$, … , $arg_n$)
- pid = wait()
- exit(status)

## Shell

- > command $arg_1$ $arg_2$ …
- Standard input/output
  - Input and output redirection using > and <
- Pipes and Filters using |
- Multi-processing – Background using &
  - > gcc –g source > compilemsgs.txt & ls | pr -2 > files.txt&
- Grouping using parenthesis ()
  - ( date; ls ) | pr -2 > files.txt &
- Running a shell in a shell: e.g. > bash <commands.txt

## Starting Up

- UNIX initialization ends with starting the "init" process
- "init"
  - When a tele-type user dials in, prompt for userid/password
  - If verified, chdir to the user's home directory, and start a shell
    - Password directory may specify a different program to start
  - wait for completion… re-prompt for userid/password

## UNIX File System

"The most important role of UNIX is to provide a file system. "

## What is a File System?

- OS component that organizes data on raw storage devices

- Data by itself is just a meaningless sequence of bits and bytes
- Metadata (attributes) - the information that describes the data

- A File system:
  - Defines the structure imposed on top of the data
  - Defines the structure and meaning of the meta-data
    - file name, permissions, file size, etc.
  - Manages mapping from disk data blocks to data and metadata
  - Manages free space on a disk

## Types of Files

- Ordinary Files
  - Text files – streams of characters with newlines
  - Binary files – streams of binary words
  - Structure of the file is defined by the programs that use them
- Directories
  - Mapping between file-names and files (or directories)
  - Written only by the kernel
  - Read like an ordinary file
- Special files
  - I/O devices
  - From user point of view, act as ordinary files, but commands like "read" invoke device driver functions which don't have to read from a disk drive!

# UNIX File System Metadata Conventions

- Root directory, specified by "/"
- Path names "/alpha/beta/gamma" – slash is subdirectory
- Current directory – relative path names
- "." represents current directory
- ".." (almost always) represents parent directory

- Elegance: These conventions hold no matter where we are in the file system hierarchy, or whether it's an actual disk file or a connection to some other device

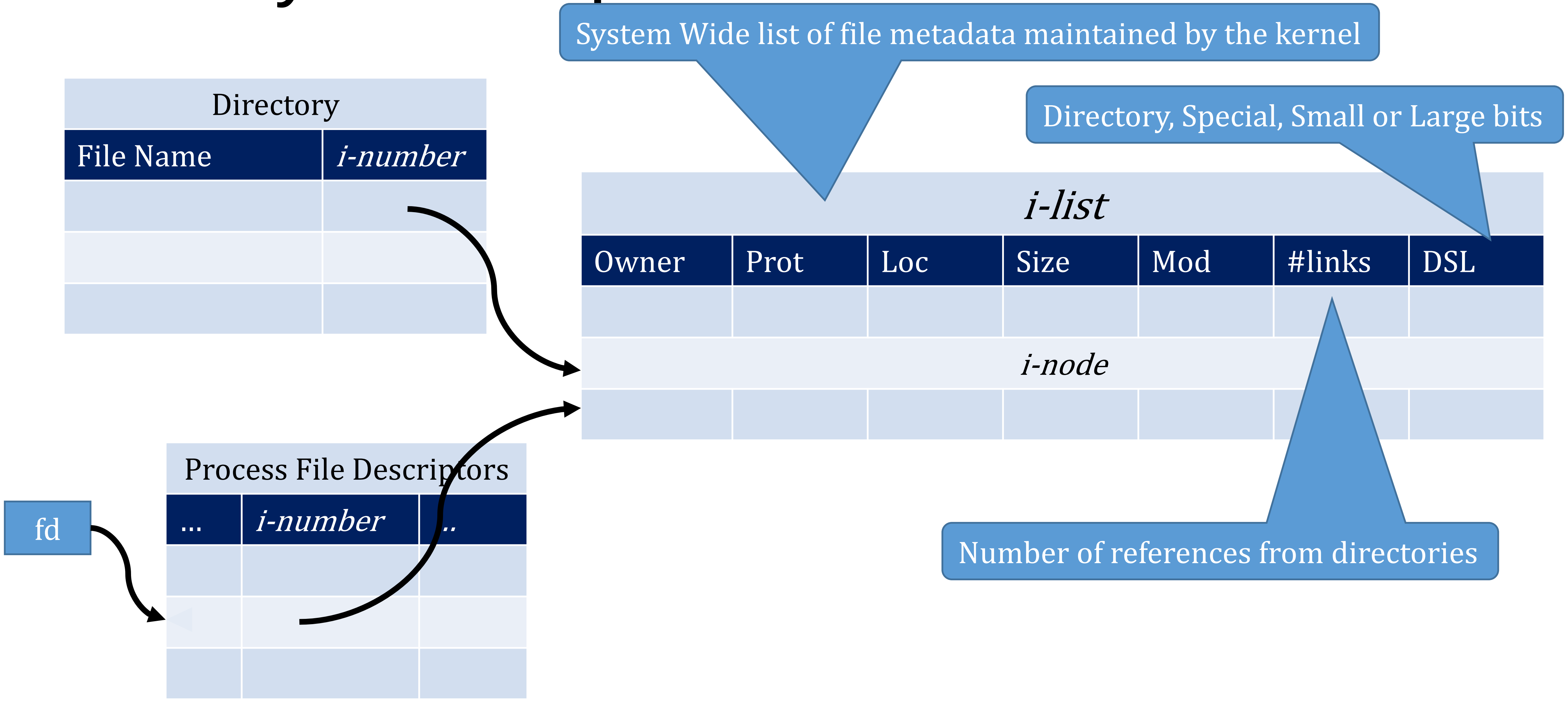
## Protection

- Each user assigned a unique userid (number)
- File metadata includes "owner" – userid of the creator
- File metadata includes 7 "protection bits"
  - RWX for owner, RWX for others and a "set-uid" flag (group added later)
- If set-uid bit is 1, while file is executing, change userid of the current user to the userid of the owner of the file!
  - Enables safe use of kernel programs that require access to system files
  - Original userid (userid of the invoker) available for credential checks
- One "superuser" exempt from protection checking (usually "root")

# Application view of I/O

- fd = open(path,flags)
  - fd – file descriptor, small integer; path either file or device (/dev/xyz)
  - Implicit file offset = 0 (or last byte in file if append)
- n = read(fd,buffer,count)
- n = write(fd,buffer,count)
  - copy up to count items between buffer and file starting at offset
  - update offset
  - n is number of bytes transferred; if reading n=0 indicates EOF
- loc=seek(fd,base,delta)
  - updates implicit offset
- close(fd)

# File System Implementation


System Wide list of file metadata maintained by the kernel
Directory, Special, Small or Large bits
Directory
File Name
i-number
i-list
Owner
Prot
Loc
Size
Mod
#links
DSL
i-node
Process File Descriptors
…
i-number
..
fd
Number of references from directories

## linking files

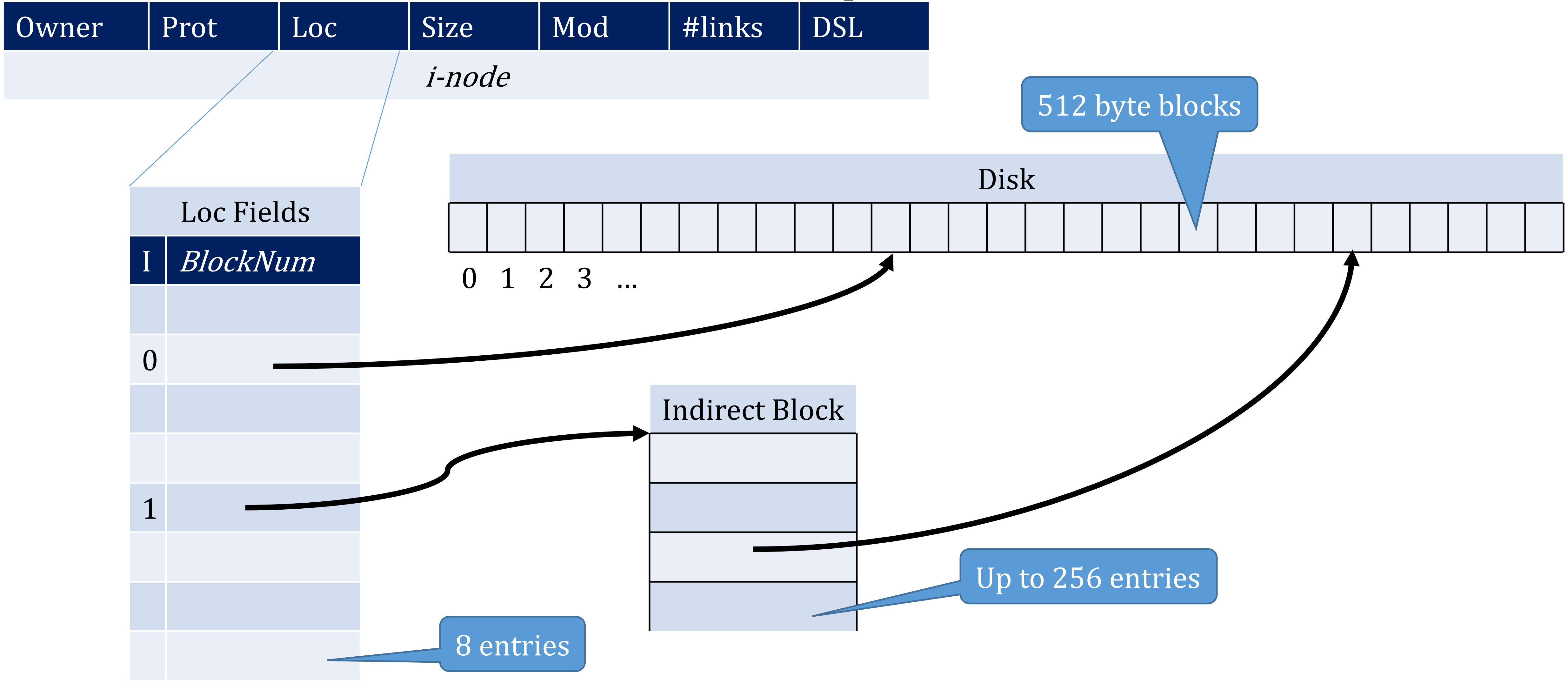
- The link command allows you to specify
  - The original file
  - A new file name that points to the original file

- The link command:
  - Finds the i-number of the original file
  - Creates a new directory entry for the new file name
    - Points to the same i-number as the original file
  - Increments #links in the i-node

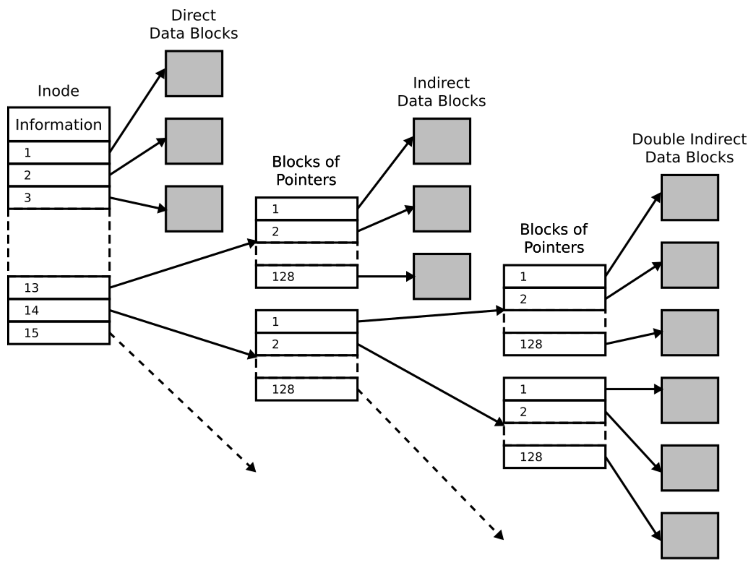
## File Creation / Deletion

- Creating a new file
    - Create a new i-node with #links=1
    - Add the i-node to the i-list to get an i-number
    - Add the file name and i-number to the directory
- Deleting a file
    - Find i-number
    - decrement #links
    - if #links==0, remove physical blocks and remove i-node
    - Remove directory entry

# File Location : Ordinary Files



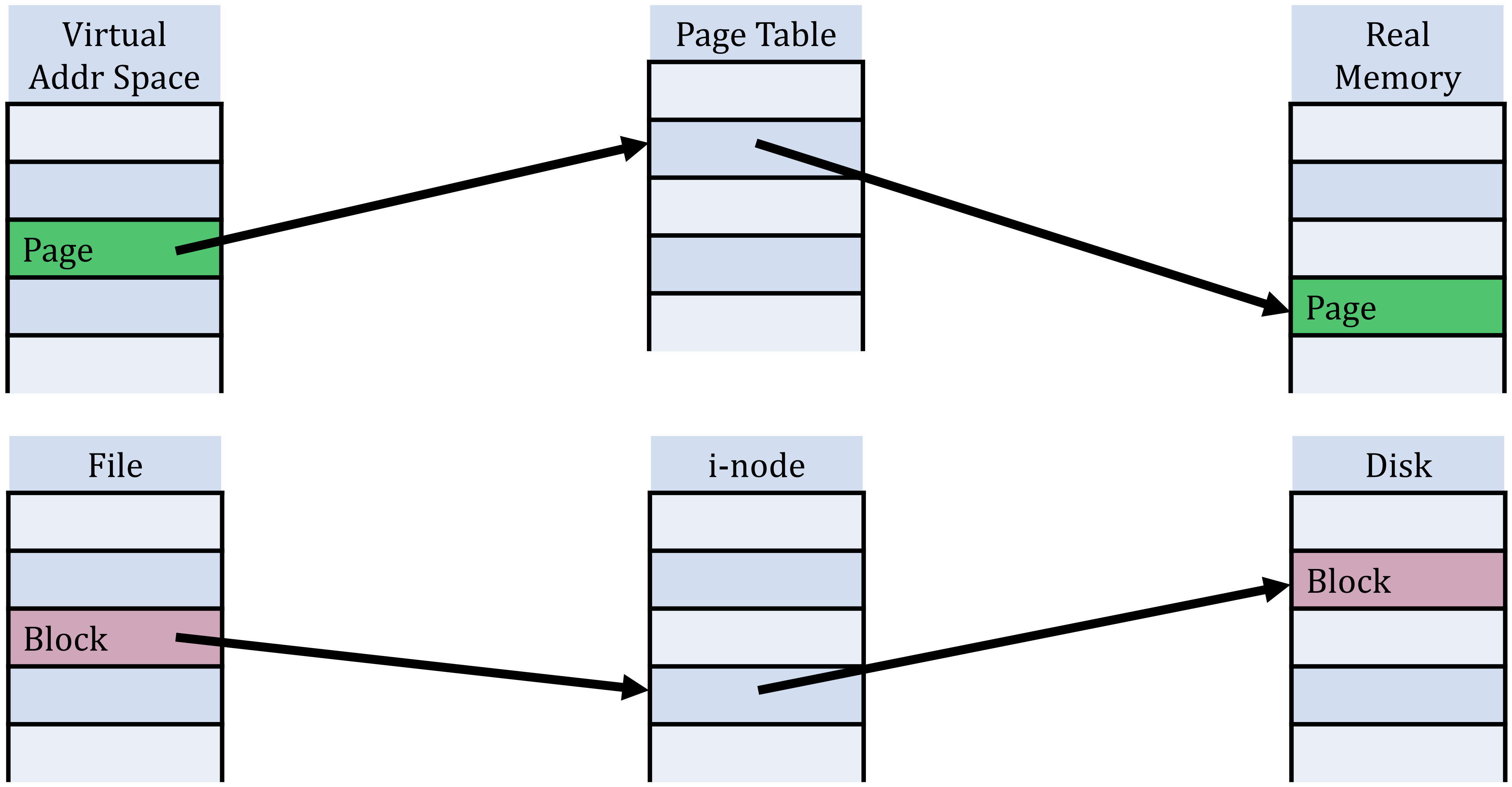
## File Location Details



- Original Specifications
  - Small files (S=1) < 8 x 512 = 4K (no indirection)
  - "Large files" (S=0) < 8 * 256 * 512 = 1M

- Current UNIX: (See inode pointer structure)
  - Block size no longer fixed at 512 bytes… can be much larger
  - Each indirect table is one block – bigger blocks contain more entries
  - i-node contains 15 location entries (with indirection, gets to >4TB for 4K blocks)
    - 0-11 are direct entries – contain block indexes
    - 12 is a single indirect block pointer
    - 13 is double indirect block pointer indirect block -> indirect block -> block
    - 14 is a triple indirect block pointer indirect block -> indirect block -> indirect block -> block

# Page tables Compared to i-node

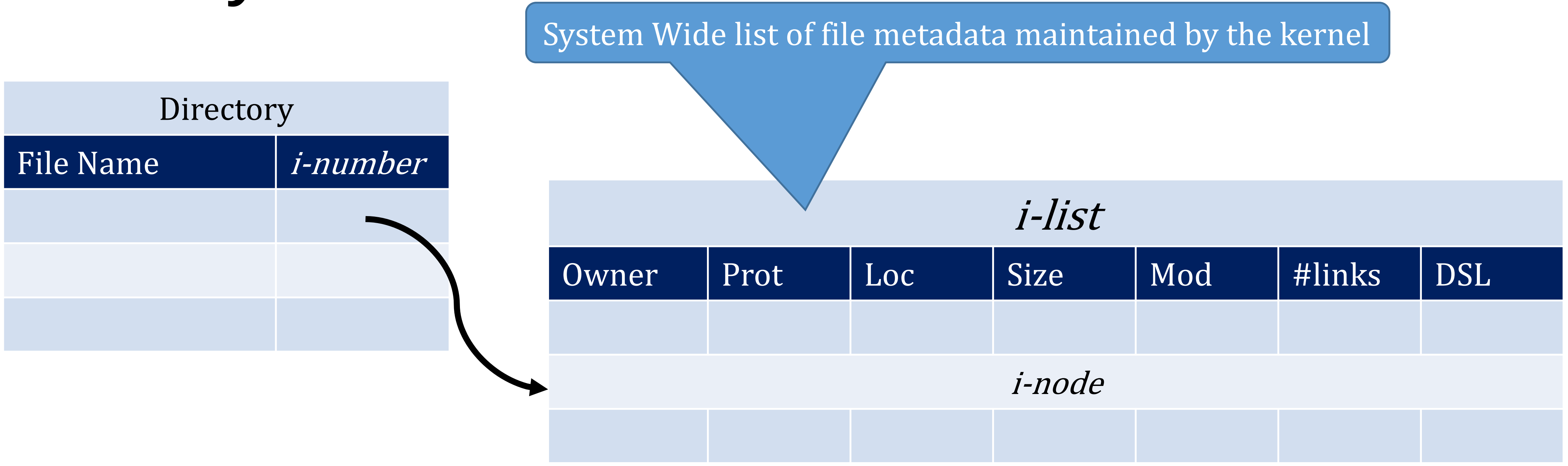


## i-node for special files

- First loc field divided into 2 bytes
  - Major device number – Device type
  - Minor device number – sub-device number

- Major device number points to device driver
- Minor device number enables variations for a single device

## File System Persistence



System Wide list of file metadata maintained by the kernel

| Directory | |
|---|---|
| File Name | *i-number* |
| | |
| | |
| | |

| *i-list* | | | | | | |
|---|---|---|---|---|---|---|
| Owner | Prot | Loc | Size | Mod | #links | DSL |
| | | | | | | |
| *i-node* | | | | | | |
| | | | | | | |

**What happens when the system shuts down?**

## File System Persistence

- i-node metadata stored on disk with each file
- When i-list is updated, requires write to disk
- Special "Superblock" points to all i-nodes on disk
- When system is started, fixed disk i-nodes added to i-list, and directories are constructed

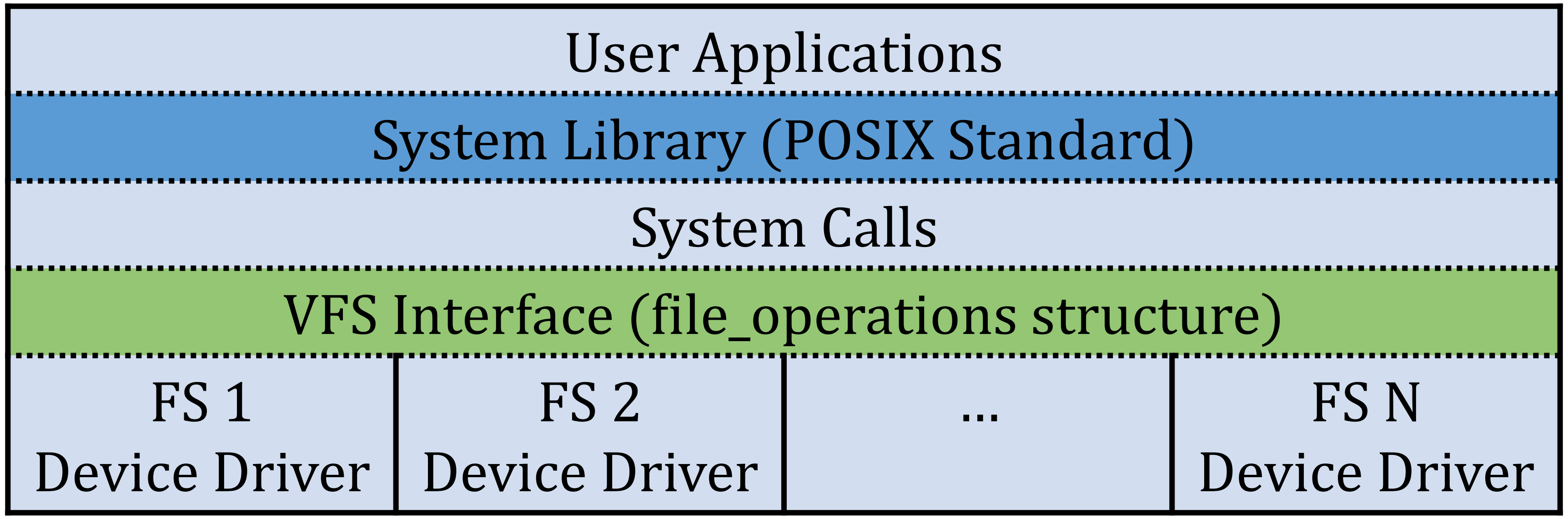
## Mounting a detachable device

- Read the device superblock and add devices i-nodes to i-list
- Create directories and directory tree for mounted device
- Connect the mounted devices directory tree to the existing directory tree

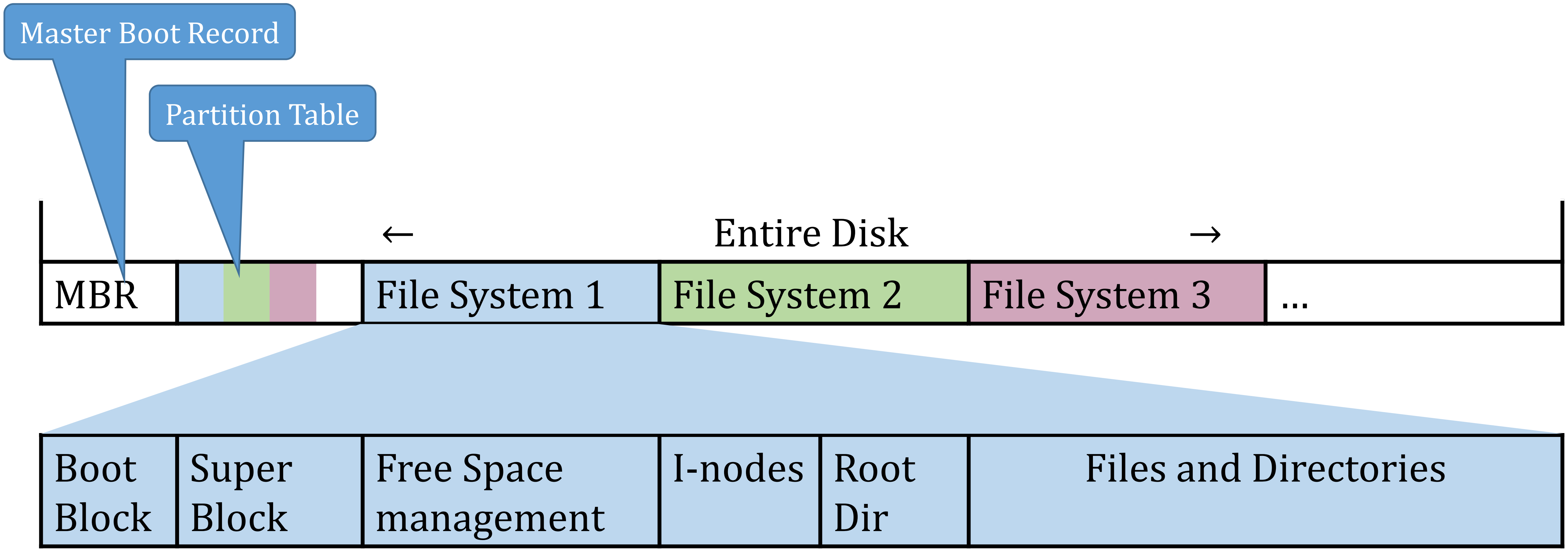
- Unmount detaches mounted devices directory tree from the existing directory tree
- Marks i-nodes in i-list as invalid/free

## Virtual File System

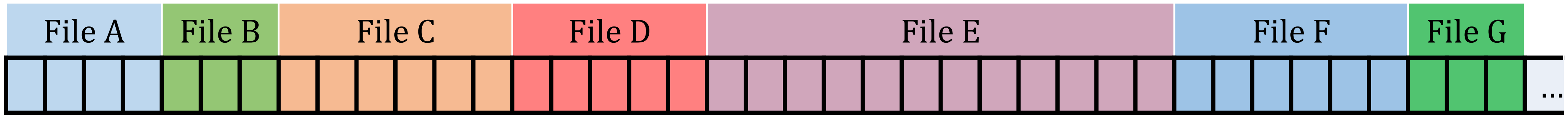
- Operating system provides a common call interface (API) to access a file system… e.g. "open(), read(), write(), lseek(), fctl(), close(), etc.
- Operating system provides a mechanism for any file system that supports the common call interface to "plug in" to the file system

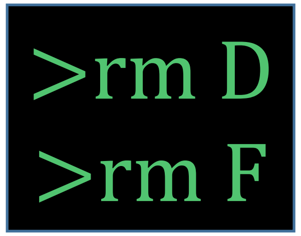


# Physical Disk Layout

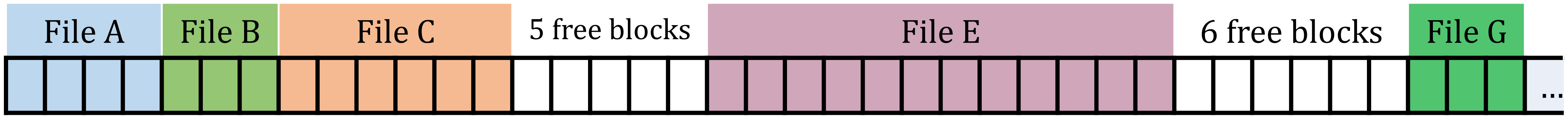


## Disk Management: Contiguous Blocks



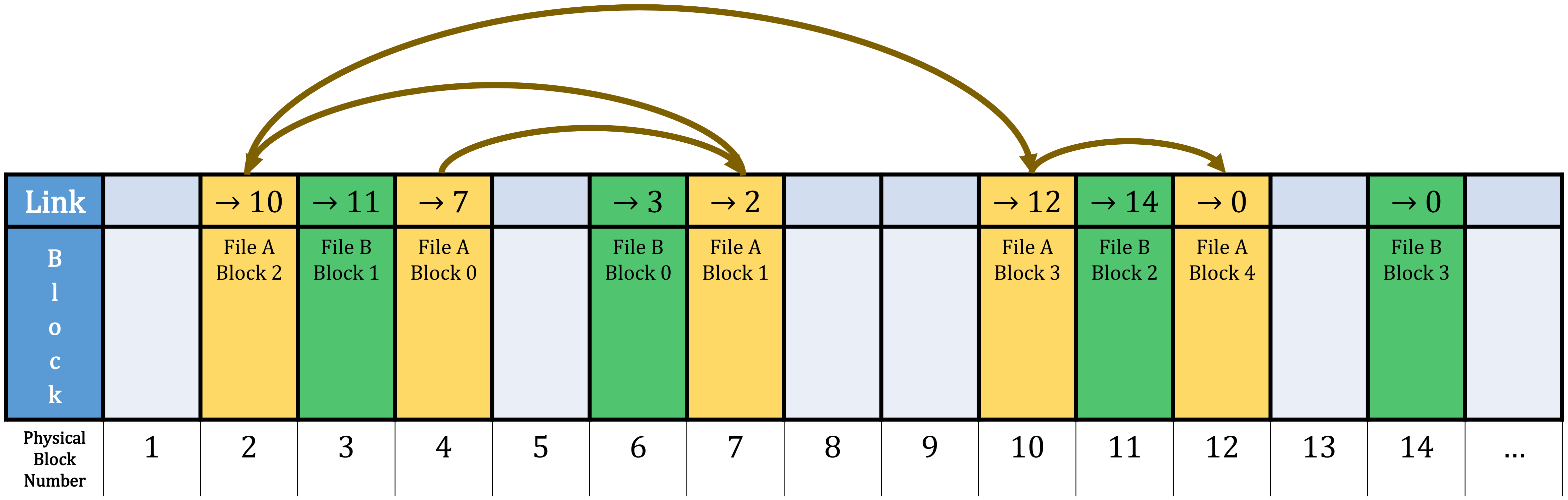


```
>rm D
>rm F
```



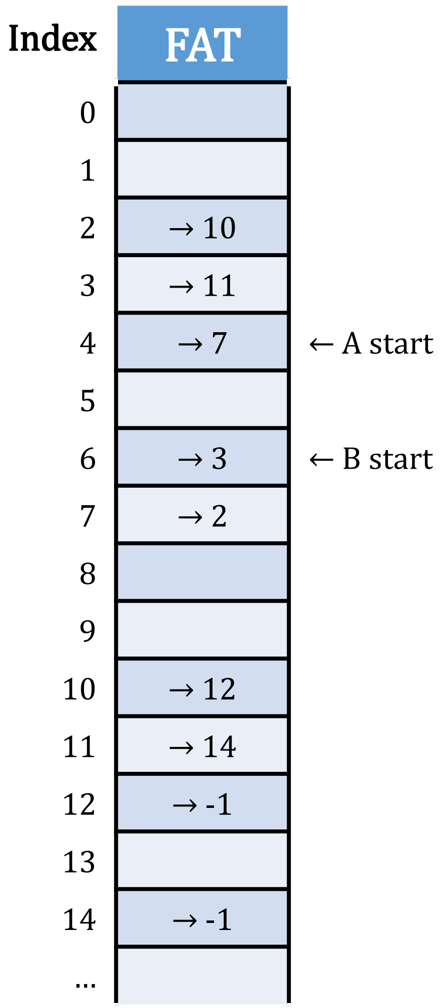
**External Fragmentation!!!**

# Disk Management: Singly Linked List

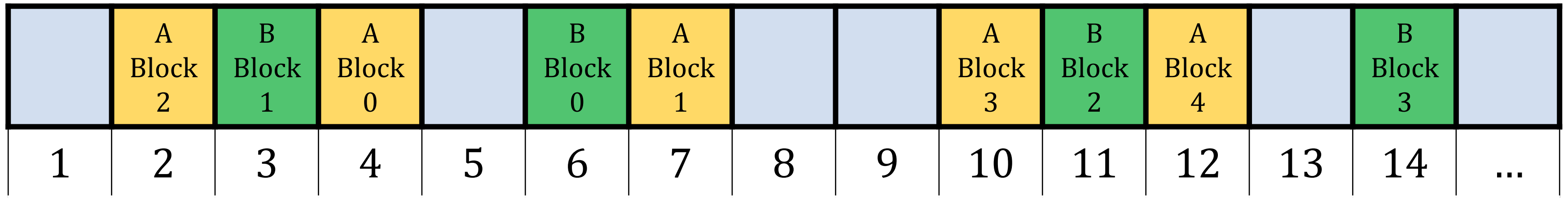


- Advantage: Logically contiguous blocks can be discontiguous on disk
- Disadvantage: Random seeks are expensive. Requires traversal from start

## Disk Mgmt: File Allocation Table (FAT)



| Index | FAT | |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | → 10 | |
| 3 | → 11 | |
| 4 | → 7 | ← A start |
| 5 | | |
| 6 | → 3 | ← B start |
| 7 | → 2 | |
| 8 | | |
| 9 | | |
| 10 | → 12 | |
| 11 | → 14 | |
| 12 | → -1 | |
| 13 | | |
| 14 | → -1 | |
| ... | | |



| | A Block 2 | B Block 1 | A Block 0 | | B Block 0 | A Block 1 | | | A Block 3 | B Block 2 | A Block 4 | | B Block 3 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | ... |

- Linked list allocation using FAT in RAM
- No need for "next" pointer for each block on disk
- Random seeks are still expensive
- FAT saved on disk for persistence

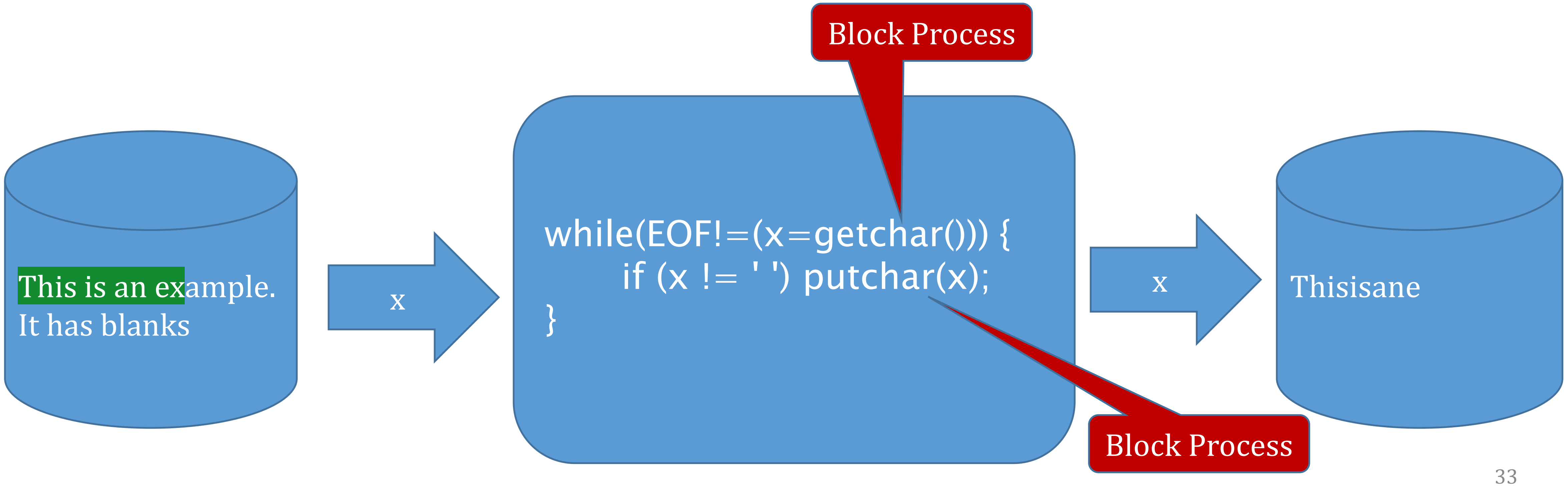
## Consider the following "stripBlanks" pgm

```c
#include <stdio.h>
int main(int argc, char **argv) {
  char x;
  while(EOF!=(x=getchar())) {
    if (x != ' ') putchar(x);
  }
  return 0;
}
```

```
>cat xmp.txt
This is an example.
It has blanks
>./stripBlanks <xmp.txt
Thisisanexample.
Ithasblanks
```

# Read/Write Buffering

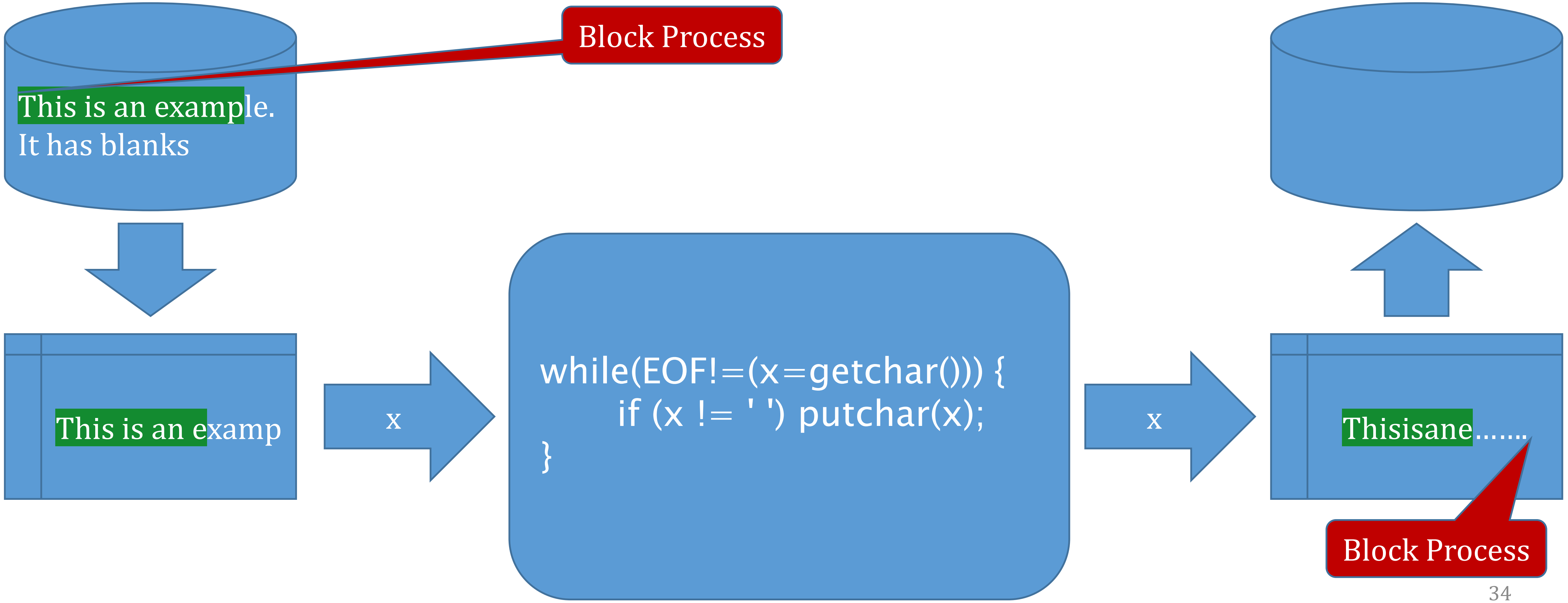
- Disk I/O is about 100 times slower than RAM access
- Disk I/O often causes process to get blocked – cold start penalty



```
while(EOF!=(x=getchar())) {
      if (x != ' ') putchar(x);
}
```

# Read/Write Buffering

- Instead OS Reads/Writes Buffers of Data

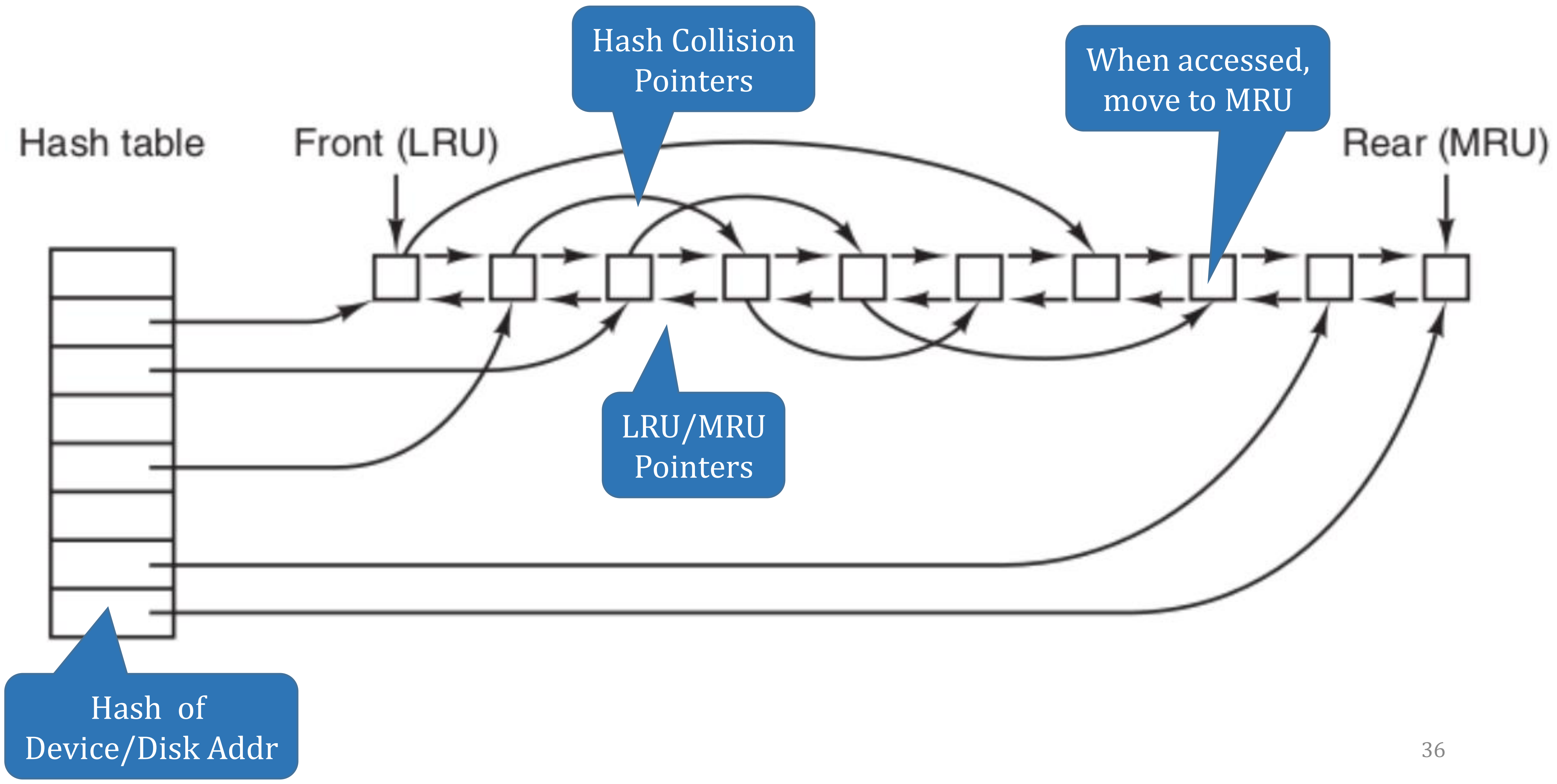


```
while(EOF!=(x=getchar())) {
        if (x != ' ') putchar(x);
}
```

## File System Cache

- Small Area in user main memory to store frequently accessed data
- Before reading from the disk, check the cache
  - If the data block in question is in the cache, no need to go to disk
  - If the data block in question is NOT in the cache, evict LRU block, read from disk to available cache slot

# File System Cache Access / LRU



## File System Cache: Data Integrity

- What happens if the system crashes?
  - Data Blocks in cache may include data, directories, indirect blocks, i-nodes
  - All "dirty" pages have been updated in RAM, but not yet on disk!
  - Can leave File System in an inconsistent state… crash the file system!

- Solutions:
  - Windows: Make the Disk cache a "write-through" cache
  - UNIX: Support "sync" command to write all dirty blocks
  - UNIX: Make File System blocks write-through
  - UNIX: Background deamon "update" – run sync every 30 seconds

## File System Cache Warning

- Don't forget to run sync before unmounting a file system!
- Why?

## Unified Virtual Memory and File System $

- Use virtual memory cache for file system blocks as well as virtual memory pages
- Take advantage of L1, L2, L3 cache for file data as well as memory
- Enable file access at CPU speeds
- Need extra handling for data integrity issues!

## Log-Structured File Systems

- Disk caches are getting larger
- Increasing % of read requests come from file system cache
- Most disk accesses will be writes!

- Log File System (LFS) treats the entire disk as a circular log
  - All writes are initially buffered in memory
  - When the disk block becomes full, write it to head of log
    - May need to update i-nodes … hopefully in cache
  - On open, search through log for all i-nodes, keep xref on disk and in cache
    - inodes point to only the LATEST versions of each block of the file
  - Periodically, go through log to move used segments to head (clean)

# Ungraded Quiz on i-nodes

- Assume:
  - all blocks in a disk are 4096 (4K) large
  - A block can store either data, or metadata, but not both
  - Each block address (block's location on disk) is 8 bytes long
  - all file attributes take up negligible (0) space in the top-level i-node block
  - Last three entries of the top-level block of an i-node contain single, double, and triple indirect entries
  - The rest of the space in the top-level i-node block (between the end of attributes and single-indirect block address) is used for direct block addresses
- Question 1: What is the largest size file that can be accessed through:
  - Only direct block entries
  - direct and single indirect block entries
  - direct and single and double indirect block entries
  - direct and single and double and triple indirect block entries
- Question 2: What is the size (in bytes) required to hold i-node data for a 32G file?