

# CS202 – Computer Security

Filesystems

February 26, 2025

# What is a file?



- A repository for data
- Is long lasting (until explicitly deleted)
- Long than the lifetime of a process

# Two aspects to consider ...



- User's view
  - ▣ Abstractions, Operations, Naming, Permissions, Security, ...
- File System implementation

# File Operations



- Create, Delete, Open, Close, Read, Write, Append, Seek, Get attributes, Set attributes, Rename
- Exercise: Get acclimatized to UNIX file system calls

# File Permissions



- How to specify the access of all user processes to any file?
  - ▣ Without making this super tedious and super complicated?
- What permissions do we want to allow to other processes?

# File Permissions



- To simplify access, file permissions are assigned to **owner ID**, **group ID**, and then everyone else, called **“others”**
  - ▣ `rw-r--r--` means owner can read/write, group and others can read

# Directory



- A way of organizing files hierarchically
- Each directory entry has:
  - ▣ Directory name
  - ▣ A list of directory entries (e.g., subdirectories and files)
  - ▣ Permissions to operate on directory entries
    - Read, write, execute

# UNIX File & Dir Permissions

-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx-----	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx-----	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/



# File System Implementation

- View the disk as a logical sequence of blocks
- A block is the smallest unit of allocation.
- Issues:
  - ▣ How do you assign the blocks to files?
  - ▣ Given a file, how do you find its blocks?



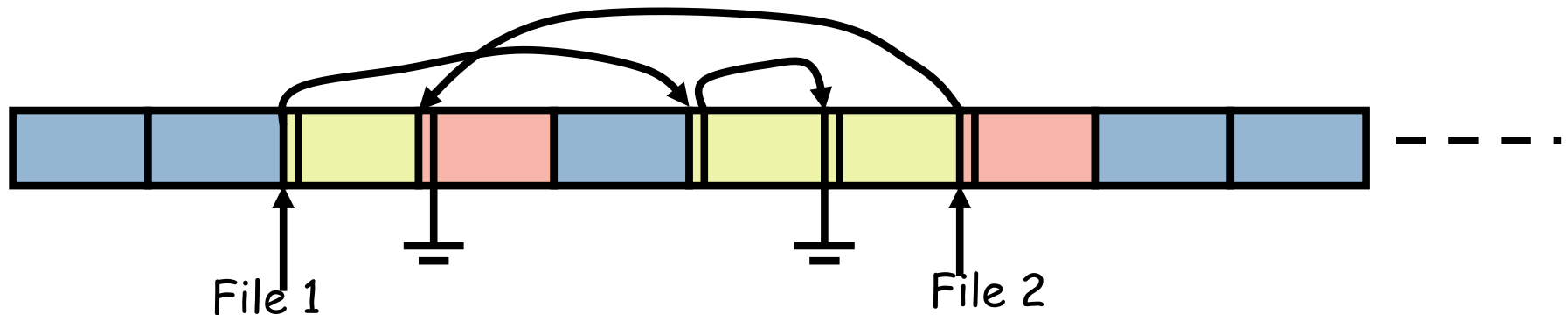
# Contiguous Allocation



- Allocate a sequence of contiguous blocks to a file.
- Advantages:
  - ▣ Need to remember only starting location to access any block
  - ▣ Good performance when reading successive blocks on disk
- Disadvantages:
  - ▣ File size has to be known a priori.
  - ▣ External fragmentation

# Linked List Allocation

- Keep a pointer to first block of a file.
- The first few bytes of each block point to the next block of this file.
- Advantages: No external fragmentation
- Disadvantages: Random access is slow!



# Linked List Allocn. Using an Index (e.g. DOS)

- ❑ In the prev. scheme, we needed to go to disk to chase pointers since memory cannot hold all the blocks.
- ❑ Why not remove the pointers from the blocks, and maintain the pointers separately?
- ❑ Perhaps, then all (or most) of the pointers can fit in memory.
- ❑ Allocation is still done using linked list.
- ❑ However, pointer chasing can be done “entirely” in memory.



Disk Blocks

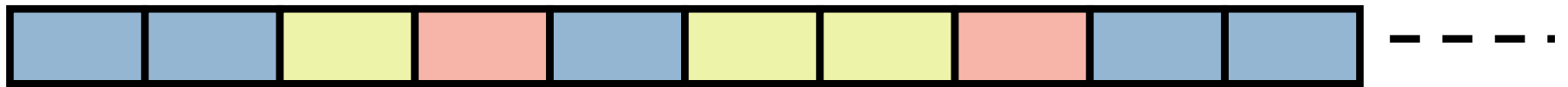
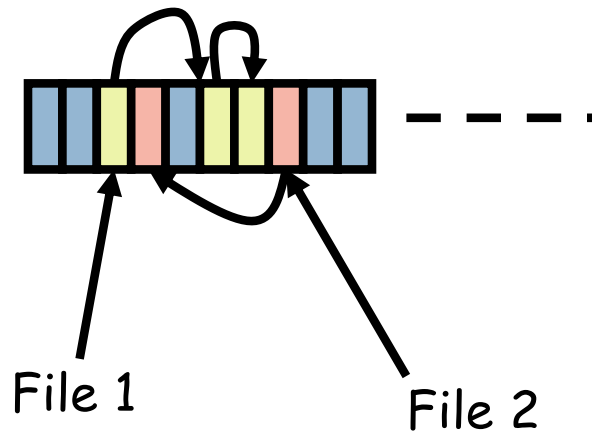


Table of  
Pointers  
(in memory?)

Called FAT  
in DOS



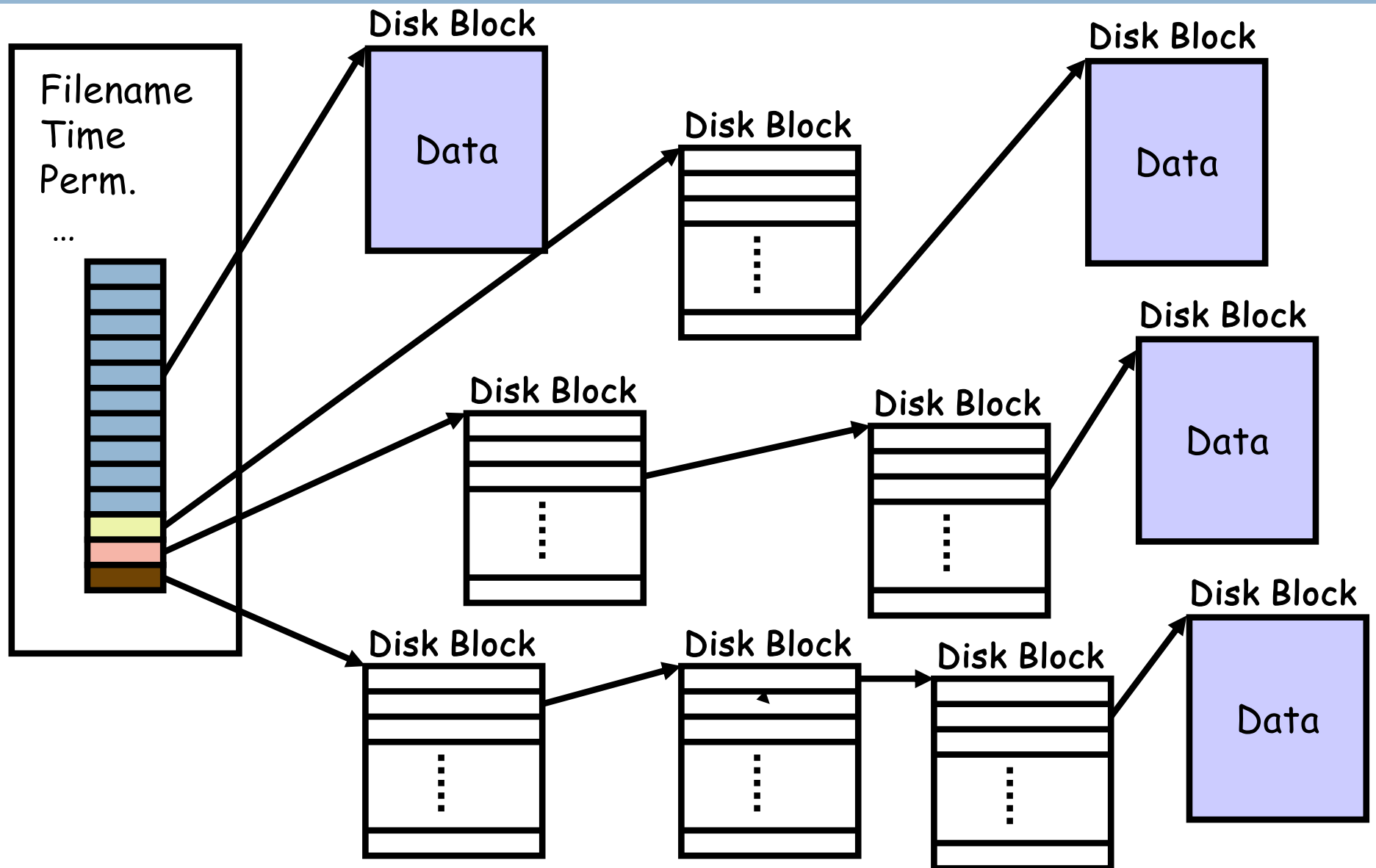
# Indexed Allocation


## (e.g., UNIX)



- For each file, you directly have pointers to all its blocks.
- However, the number of pointers for a file can itself become large.
- UNIX uses i-nodes.
- An i-node contains:
  - ▣ File attributes (time of creation, permissions, ....)
  - ▣ 10 direct pointers (logical disk block ids)
  - ▣ 1 one-level indirect pointer (points to a disk block which in turn contains pointers)
  - ▣ 1 two-level indirect pointer (points to a disk block of pointers to disk blocks of pointers)
  - ▣ 1 three-level indirect pointer (points to a disk block of pointers to disk blocks of pointers to pointers of disk blocks)

# inode




- 
- Exercise: Given that the FAT is in memory, find out how many disk accesses are needed to retrieve block “x” of a file from disk. (in DOS)
  - Exercise: Given that the i-node for a file is in memory, find out how many disk access are needed to retrieve block “x” of this file from disk. (in UNIX)



# Tracking free blocks



- List of free blocks
  - ▣ bit map: used when you can store the entire bit map in memory.
  - ▣ linked list of free blocks
    - each block contains ptrs to free blocks, and last ptr points to another block of ptrs. (in UNIX).
    - Pointer to a free FAT entry, which in turn points to another free entry, etc. (in DOS)

- 
- Now we know how to retrieve the blocks of a file once we know:
    - ▣ The FAT entry for DOS
    - ▣ The i-node of the file in UNIX
  
  - But how do we find these in the first place?
    - ▣ The directory where this file resides should contain this information

# Directory




- Contains a sequence (table) of entries for each file.
- In DOS, each entry has
  - ▣ [Fname , Extension , Attributes , Time , Date , Size , First Block #]
- In UNIX, each entry has
  - ▣ [Fname, i-node #]

# Accessing a file block in DOS: \a\b\c

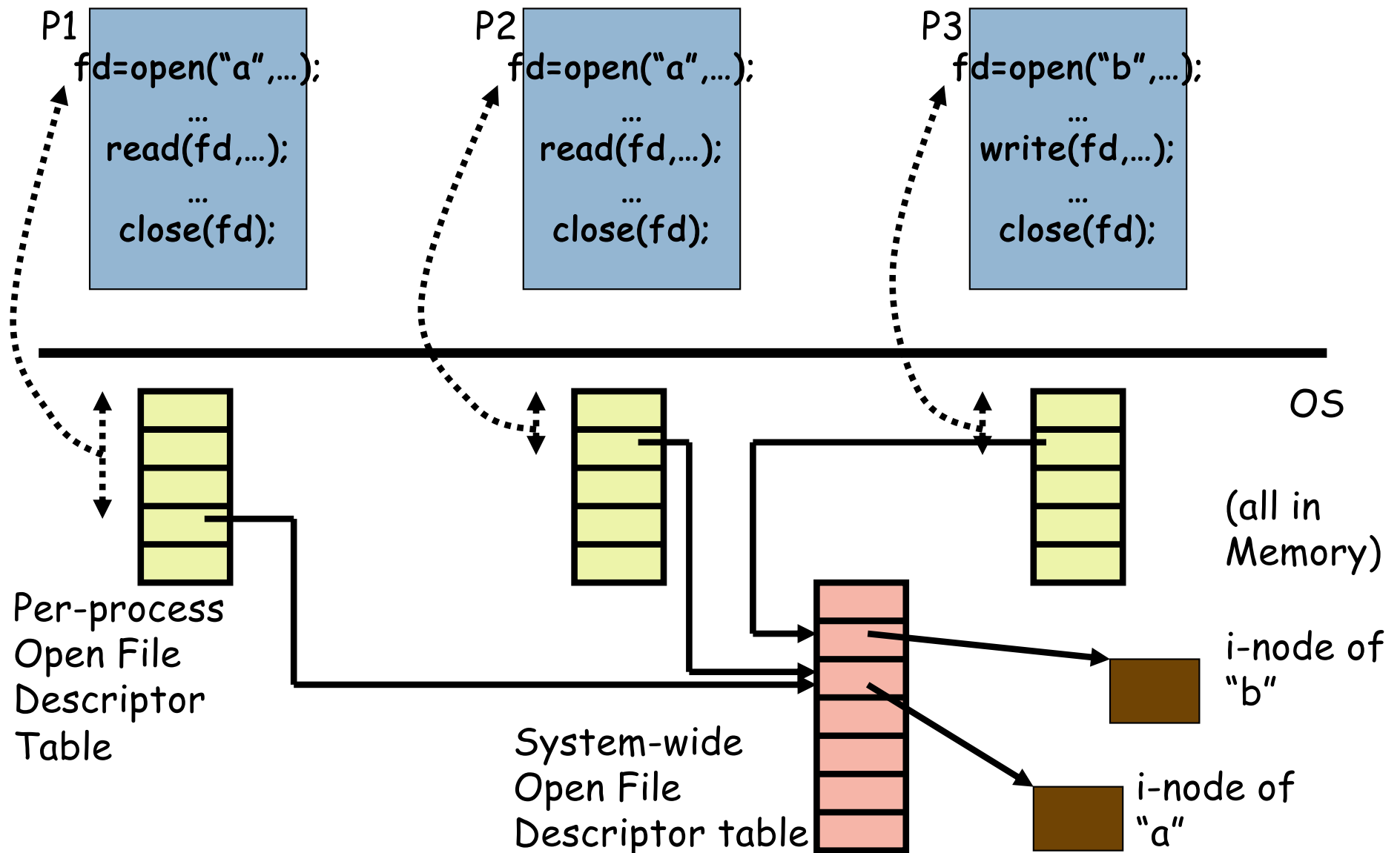
- Go to “\” FAT entry (in memory)
- Go to corresponding data block(s) of “\” to find entry for “a”
- Read 1<sup>st</sup> data block of “a” to check if “b” present. Else, use the FAT entry to find the next block of “a” and search again for “b”, and so on. Eventually you will find entry for “b”.
- Read 1<sup>st</sup> data block of “b” to check if “c” present. ....
- Read the relevant block of “c”, by chasing the FAT entries in memory.

# Accessing a file block in UNIX: /a/b/c

- Get “/” i-node from disk (usually fixed, e.g., #2)
- Get block after block of “/” using its i-node till entry for “a” is found (gives its i-node #).
- Get i-node of “a” from disk
- Get block after block of “a” till entry for “b” is found (gives its i-node #)
- Get i-node of “b” from disk
- Get block after block of “b” till entry for “c” is found (gives its i-node #)
- Get i-node of “c” from disk
- Find out whether block you are searching for is in 1<sup>st</sup> 10 ptrs, or 1-level or 2-level or 3-level indirect.
- Based on this you can either directly get the block, or retrieve it after going through the levels of indirection.

- 
- Imagine doing this each time you do a read() or write() on a file
  - Too much overhead!
  - However, once you have the i-node of the file (or a FAT entry in DOS), then it is fairly efficient!
  - You want to cache the i-node block (or the block with the FAT entry) for a file in memory and keep re-using it.

# This is the purpose of the open() syscall




# Exercise




- See how you can implement create file, remove file, open, close, read, write, etc. for the UNIX file system.



- 
- Even if after all this (i.e., bringing the pointers to blocks of a file into memory), may not suffice since we still need to go to disk to get the blocks themselves.
  - How do we address this problem?
    - ▣ Cache disk (data) blocks in main memory – called **file caching**

# File Caching/Buffering

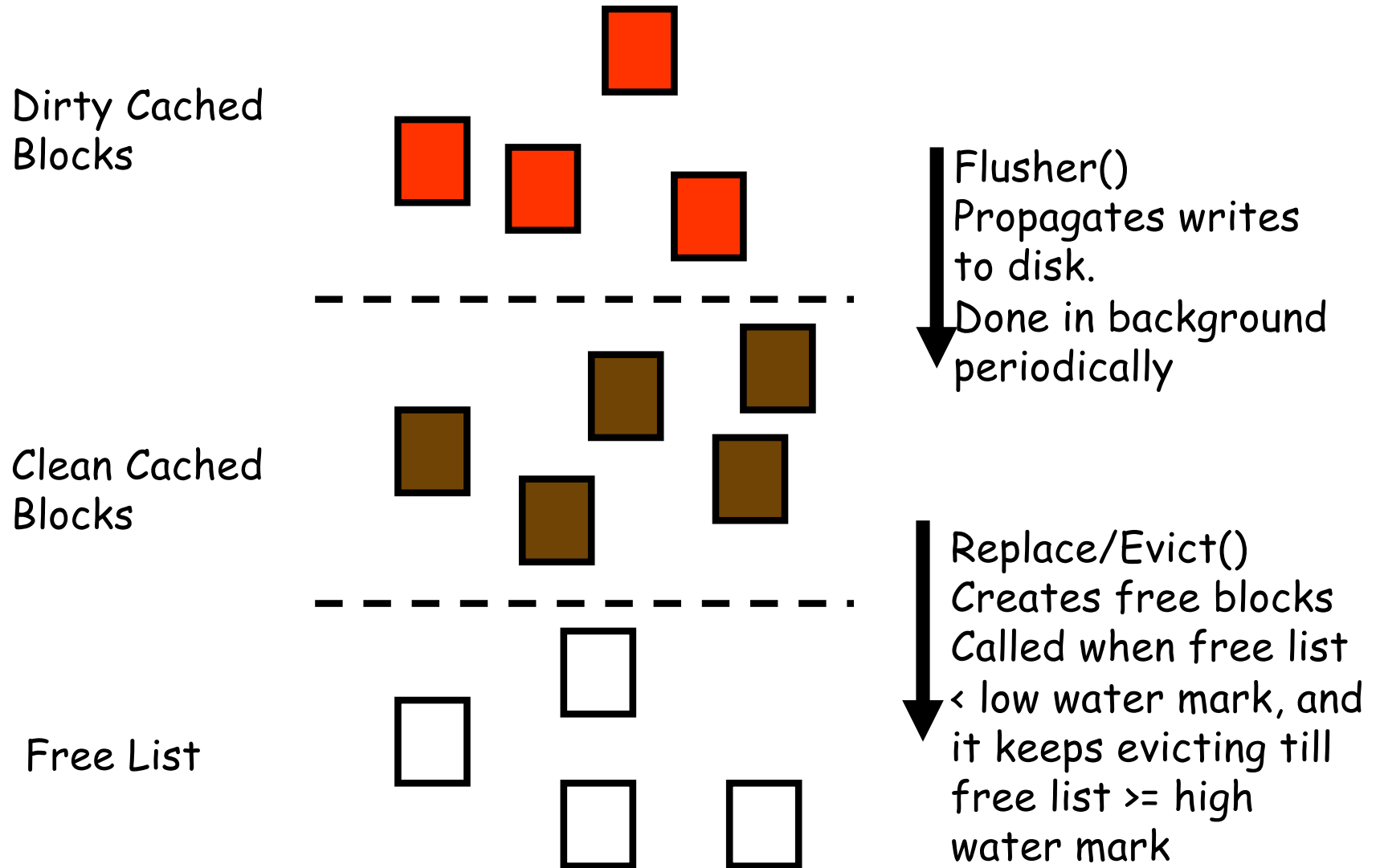
- Cache disk blocks that are in use in physical memory.
- On a read() system call, first look up this cache to check if block is present.
  - ▣ This is done in software
  - ▣ Look up is done based on logical block id.
  - ▣ Typically perform some kind of “hashing”
- If present, copy this from OS cache/buffer into the data structure passed by user in the read() call.
- Else, read block from disk, put in OS cache and then copy to user data structure.

- 
- On a write, should we do write-back or a write-through?
    - ▣ With write-back, you may lose data that is written if machine goes down before write-back
    - ▣ With write-through, you may be losing performance
      - Loss in opportunity to perform several writes at a time
      - Perhaps the write may not even be needed!
  - DOS uses write-through
  - In UNIX,
    - ▣ writes are buffered, and they are propagated in the background after a delay, i.e., every 30 secs there is a sync() call which propagates dirty blocks to disk.
    - ▣ This is usually done in the background.
    - ▣ Metadata (directories/i-nodes) writes are propagated immediately.

# Cache space is limited!

- We need a replacement algorithm.
- Here we can use LRU, since the OS gets called on each reference to a block and the management is done in software.
- However, you typically do not do this on demand!
- Use High and Low water marks:
  - ▣ When the # of free blocks falls below **Low water mark**, evict blocks from memory till it reaches **High water mark**.

# Buffer/Cache management



# Block Sizes



- Larger block sizes
  - higher internal fragmentation.
  - unnecessary data brought in
  - + higher disk transfer rates
- Median file size in UNIX environments ~ 1K
- Typical block sizes are of the order of 512, 1K or 2K.

# File System Reliability



- **Availability** of data and **integrity** of this data are both equally important.
- Need to allow for different scenarios:
  - ▣ Disks (or disk blocks) can go bad
  - ▣ Machine can crash
  - ▣ Users can make mistakes

# Disks (or disk blocks) can go bad

- Typically provide some kind of redundancy, e.g., Redundant Arrays of Inexpensive Disks (RAID)
  - ▣ Parity
  - ▣ Complete Mirroring
- When the data from the replicas/parity do not match, you employ some kind of voting to figure out which is correct.
- Once bad blocks/sectors are detected, you mark them, and do not allocate on them.



# Machine crashes

- Note that data loss due to writes not being flushed immediately to disk is handled separately by setting frequency of `flusher()`.
- When the machine comes back up, we want to make sure the file system comes back up in a consistent state, e.g., a block does not appear in a file and free list at same time.
- This is done by a routine called `fsck()`.

# Fsck – File System Consistency Check

- Blocks:
  - ▣ for every block, keep 2 counters:
    - a) # occurrences in files/directories
    - b) # occurrences in free list.
  - ▣ For every inode, increment all the (a)s for the blocks that the file covers.
  - ▣ For the free list, increment (b) for all blocks in the free list.
  - ▣ Ideally  $(a) + (b) = 1$  for every block.
  - ▣ However,
    - If  $(a) = (b) = 0$ ,  
missing block, add to free list.
    - If  $(a) = (b) = 1$ ,  
remove the block from free list
    - If  $(b) > 1$ ,  
remove duplicates from free list.
    - If  $(a) > 1$ ,  
make copies of this block and insert into each of the other files.

# Fchk- File System Consistency Check

- Inodes (Files):
  - ▣ Maintain a counter for each inode.
  - ▣ Recursively traverse the directory hierarchy.
  - ▣ For each file, (a) increment the counter for the inode.
  - ▣ At the end compare this (a) counter with the (b) link count in inode.
  - ▣ Ideally, both should be equal.
  - ▣ However
    - if  $(b) > (a)$ ,  
just set  $(b) = (a)$
    - if  $(a) > (b)$ ,  
again set  $(b) = (a)$

# Conclusions

63

- Today we reviewed the basics of **filesystems**
  - ▣ **Abstraction** of filesystem resources provided to the user programs (e.g., operations and permissions)
  - ▣ **Implementation** of that abstraction that enables efficient and reliable access to storage
- Several considerations in implementation
  - ▣ For each file, find its data
  - ▣ Find files in a hierarchical namespace
  - ▣ Cache filesystem metadata and data
  - ▣ Recover from crashes in the middle of an operation

# Questions

64

