File Systems

File

fcb_t

file_t

path

dblock_t

cmpsc473-p3 <fs> <cmd>

pathname: /a/b/c

link: a/ → /etc/passwd/c

case: /a/B/c

DIR

ddir_t

↑

disk

dir_t

DENTRY

ddentry_t

dentry_t

FILESYS

dfilesys_t

filesys

What is a file?

linear array of bytes.

存放处

- A repository for data
- Is long lasting (until explicitly deleted).

Why not just an address space?

- You may want data to persist longer than a process
- You may want data that is larger than a virtual address space – your program or process store data for later use. when done, you may not need them.
- Easier to share the data across processes.

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?

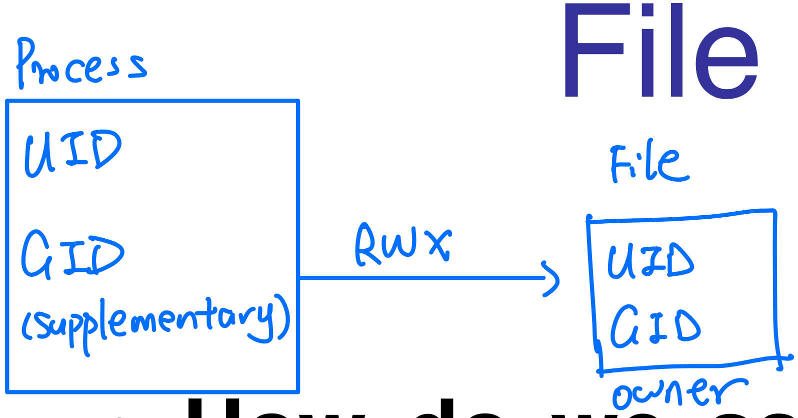
- What permissions do we want to allow to other processes?

File Permissions

- **What permissions do we want to allow to other processes?**
  - **Read, write, execute**

- **How do we assign permissions to a process for each file?**
  - **Could be a lot of work**

File Permissions



user application

- How do we assign permissions to a process for each file?
  - Processes are associated with a single user or application ID

- So, each file has a single owner ID and group ID (ID for a set of users/apps)

File Permissions

- To simplify access, file permissions are assigned to **owner ID**, **group ID**, and then everyone else, called "**others**"

  owner group other

  - rw-r--r-- means owner read/write, group and others can read

owner	group	other
rw-	r--	r--
(read, write no exe)	(read only)	(read only)

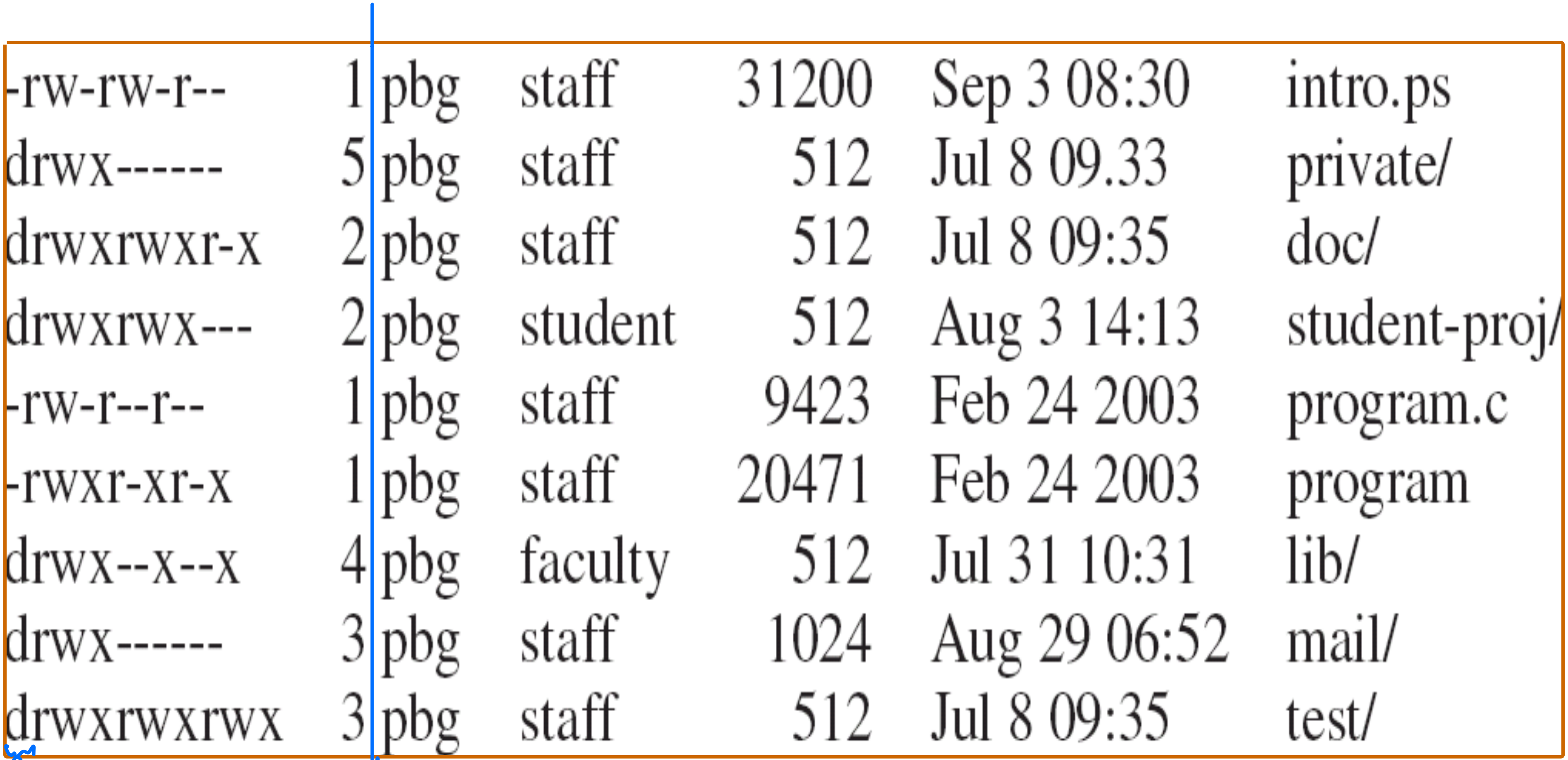
Directory

- 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

-l
-a



```
-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/
```

d: directory

user id of owner

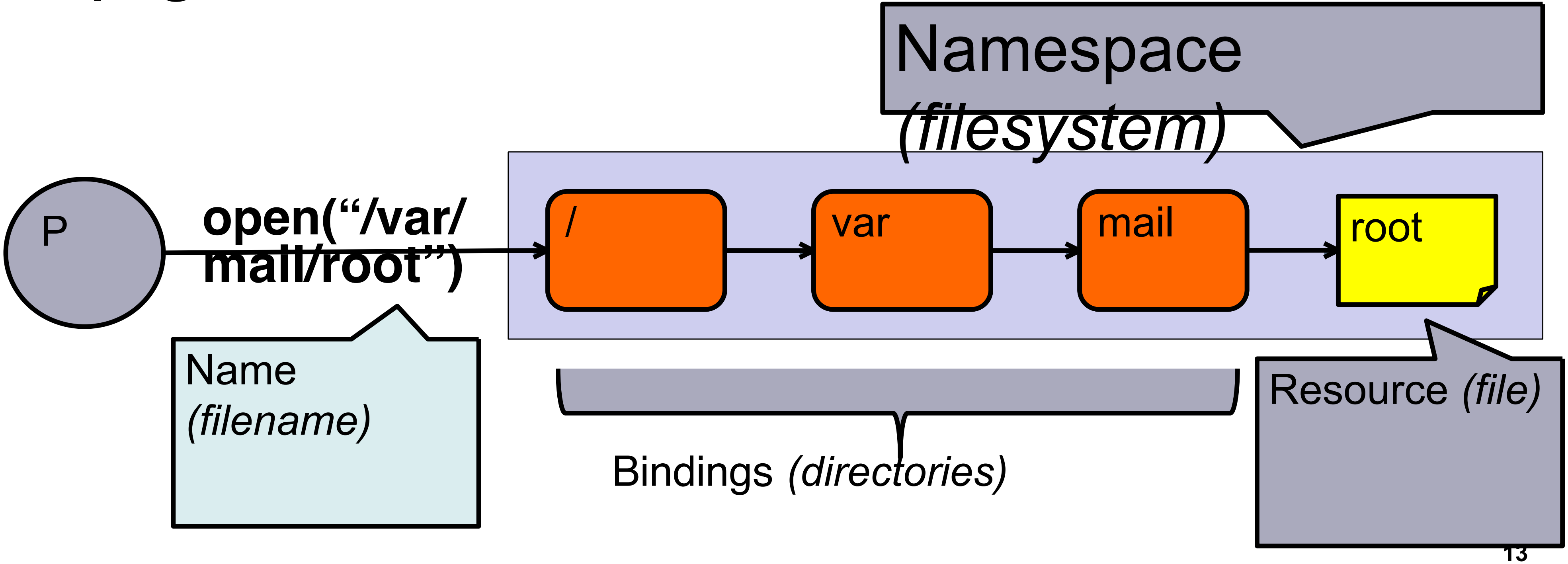
Pathnames

- **Files are accessed (opened) using pathnames**

- **Sequence of directories followed by a file**
  - **E.g., /var/mail/root**
  - **What are the directories?**
    - root directory → /, ← root directory entry /var, /var/mail → var directory entry
  - **What is the file?**
    - **root**

Name Resolution

- Processes use ***pathnames*** to obtain access to ***file system resources*** – files and directories
- A ***nameserver*** (e.g., OS) performs ***name resolution*** to convert a ***pathname*** into a system ***resource*** (e.g., *file*)



Symbolic Links

absolute path: start with "/" root

relevent path: start with current directory, not start with "/"

- **UNIX supports special files whose values are pathnames to files or directories** say you have path: /a/b/c/d/e/f/g/h

  you can use symbolic links to simplify: ln -s /a/b/c/d/e/f/g/h /a/b/h

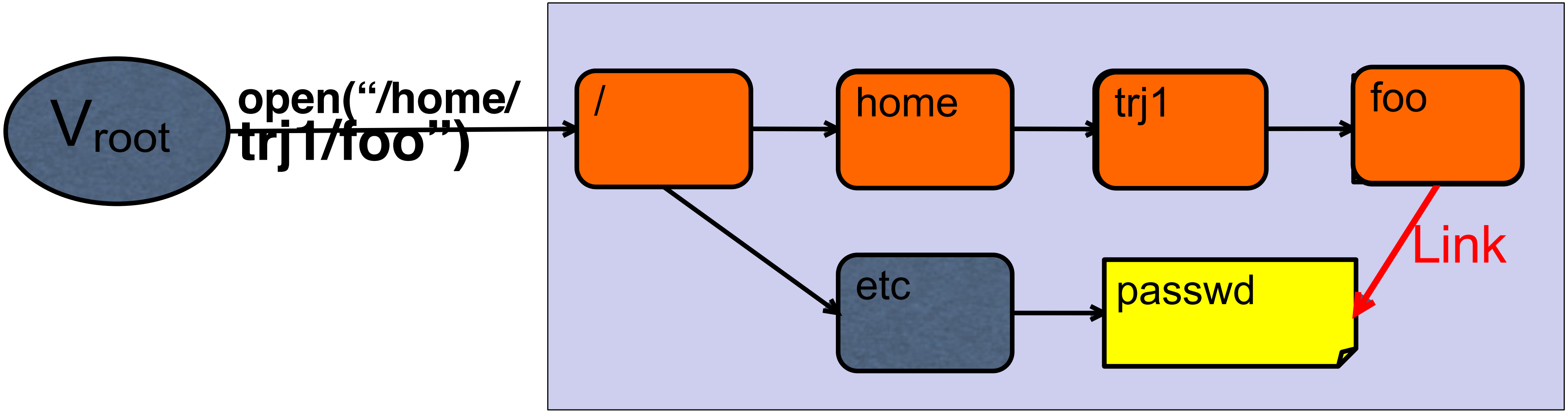
  - **ln –s <target-pathname> <link-filename>**

- **Suppose you create a symbolic link "foo" in the directory "/home/trj1" with a target pathname of "/etc/passwd"**
  - **What will happen?**

ln -s /etc/password /home/trj1/foo

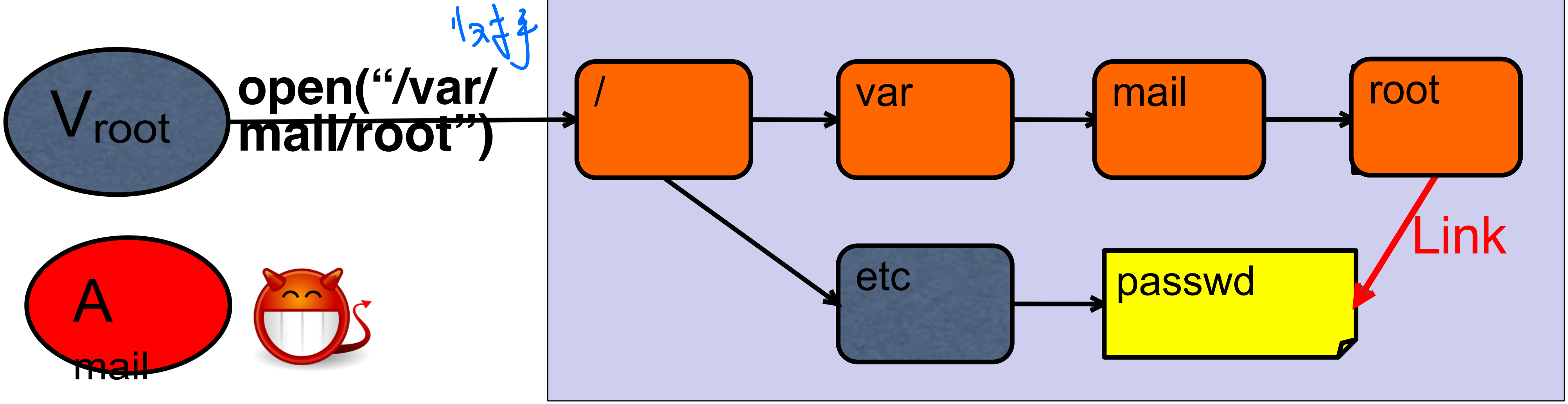
Name Resolution w/ Links

- A symbolic link provides further input to name resolution – link continues resolution
- In the example, once “foo” is found to be a symbolic link, the OS resolves its value “/etc/passwd” as part of the pathname



Link Traversal Attack

- Symbolic links can be exploited in attacks
- Victim **expects** "root" to be a file, but the adversary can provide a symbolic link instead
  - Victim is routed to a file the victim can access, but the adversary cannot



- **When are link traversal attacks possible?**

- **If an adversary of your program has write-permission to any directory used in pathname resolution by your program, the adversary 对手 can add a symbolic link to redirect your program to any resource**
  - **That your program can access**
  - **Time-of-check-to-time-of-use (TOCTTOU)**

Case Sensitivity

- Names have different meanings in different file systems
  - Case-sensitive: “FOO” and “foo” refer to two different files
  - Case-preserving: “FOO” and “foo” refer the same file – first one created wins
- Linux is case-sensitive (mostly) and Windows and Mac OS X are case-preserving

Name Collisions

- Suppose you copy the file “FOO” to a directory containing the file “foo”
  - What happens?
- Case-sensitive
  - Two files “FOO” and “foo”
- Case-preserving
  - Overwrite “foo” with ”FOO” data and metadata (permissions)
  - No solution yet!

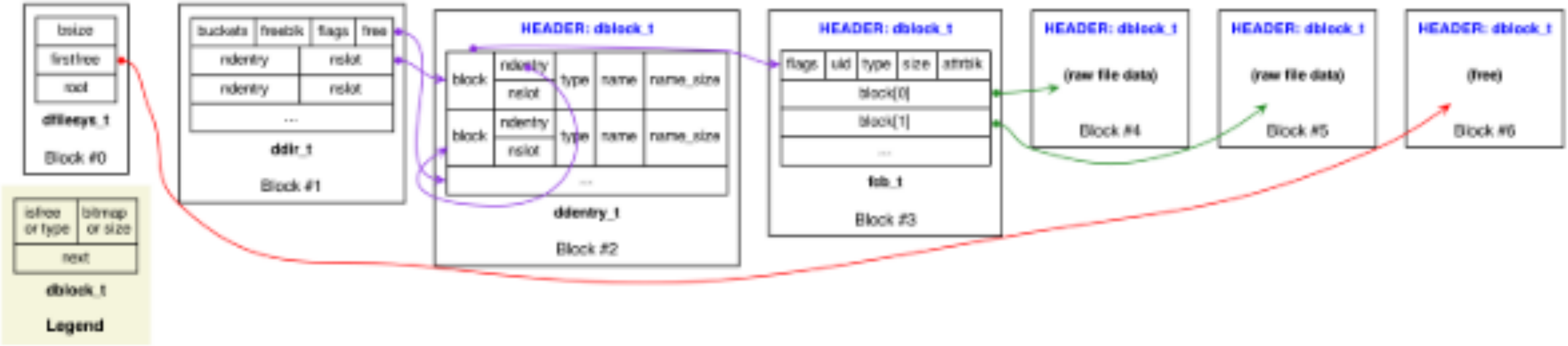
Handle with Care

- Need to be careful when accessing files
  - Open: may be redirected by a symlink
  - Copy/move: may overwrite other files unexpectedly
- Still many bugs from such mistakes, so be careful
  - Explore in Project P3

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?

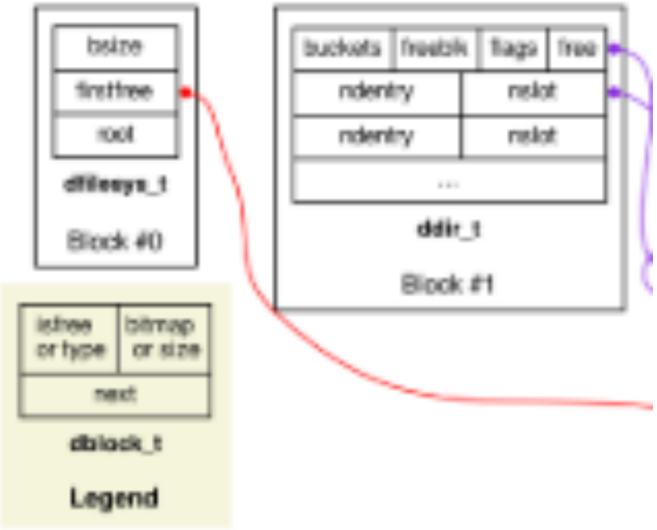
P3 File System



On-disk Layout

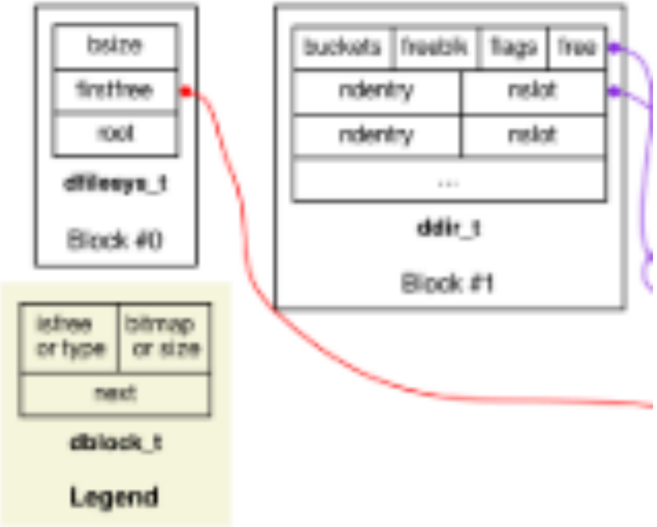
- File system is stored as a sequence of blocks for
  - File system (dfilesys_t) Directory (ddir_t)
  - File (fcb_t)
  - And their data – e.g., Dir entry (ddentry_t)

P3 File System



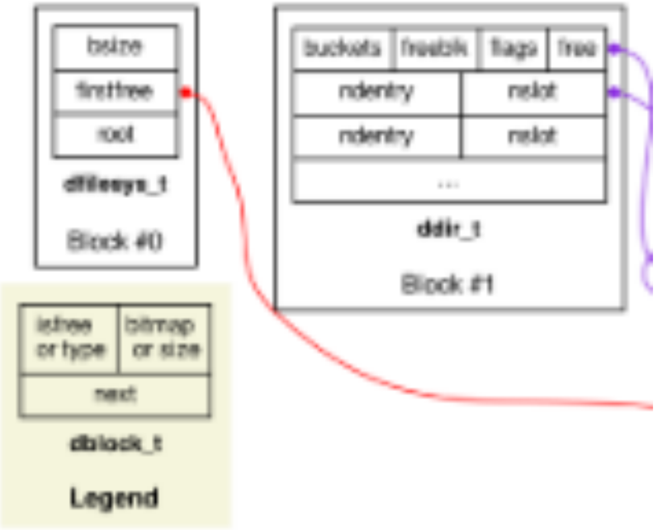
- File system has blocks (dblock_t)
  - Free (is free) and next in free list
  - Or block type and bitmap/size

P3 File System



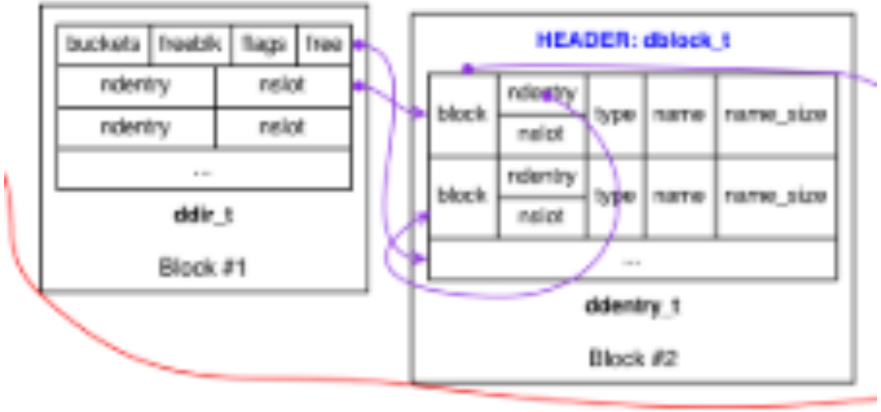
- 0^{th} block for the overall file system (dfilesys_t)
    - – References the root ‘/’ directory
    - – And the first free block in file system
        - More on allocation later in these slides

P3 Directory



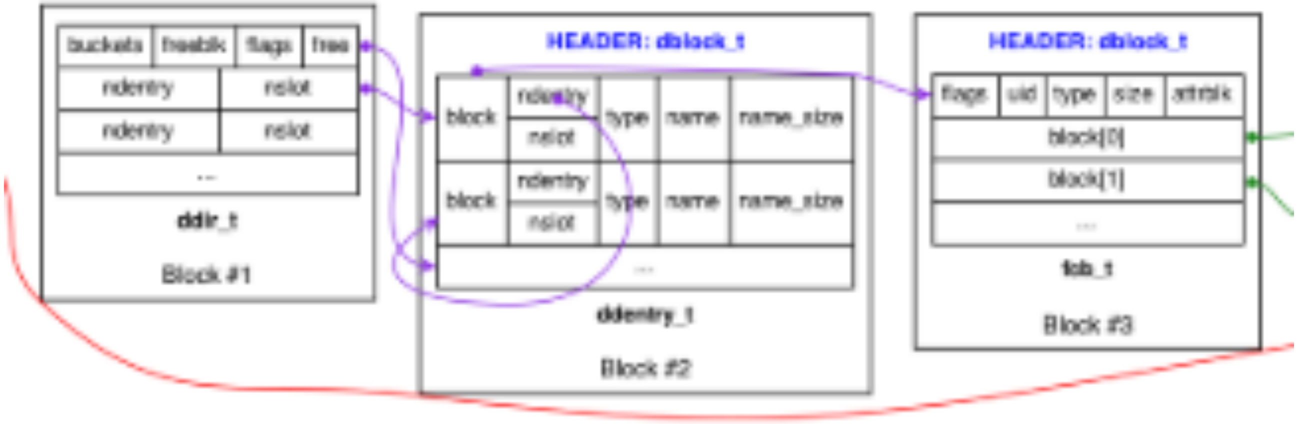
- Root directory '/' is a directory block (ddir_t)
  - Directories have entries (files and subdirs)
  - You will create subdirectories (Task 2)

P3 Directory Entries



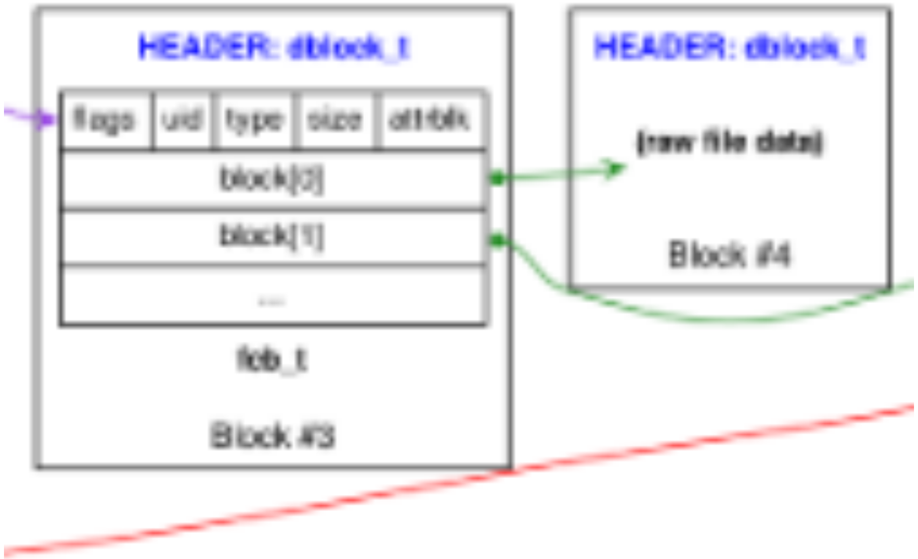
- A directory (ddir_t) and its entries (ddentry_t)
  - A ddentry_t block is a data block for directories containing a list of its entries
  - Type, name (string), and name size (length)

P3 Directory Entries



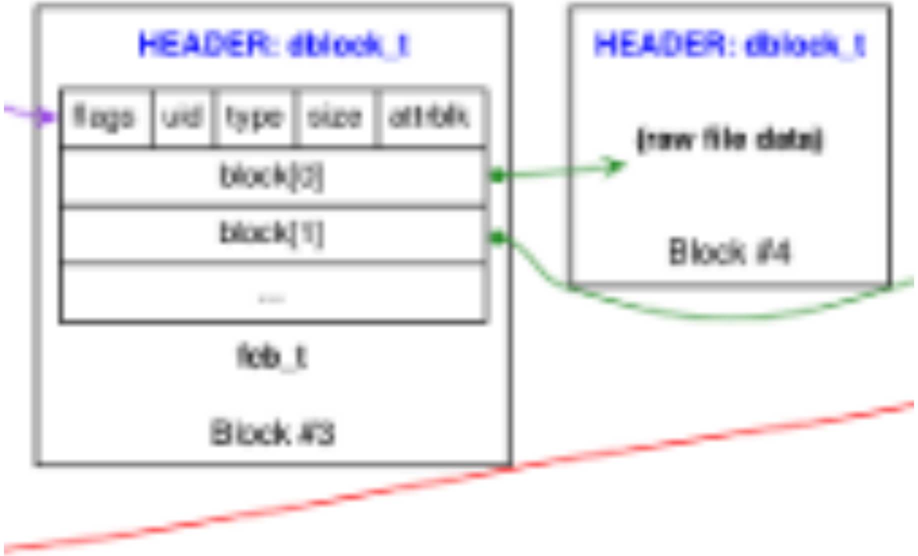
- **A directory entry (ddentry_t)**
  - **Block – ddir_t (directory) or fcb_t (file)**
  - **Type – directory or file**
  - **Name – file/dir name**

P3 Files



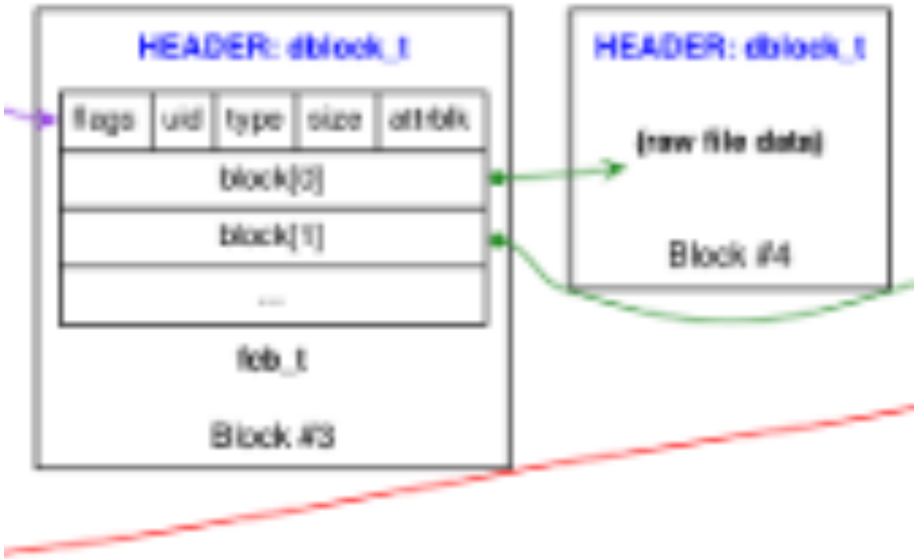
- A file control block (fcb_t)
  - UID – creator/owner
  - Type – Regular file or symbolic link
  - Size – amount of data
  - Attrblk – ignore for this project

P3 Files



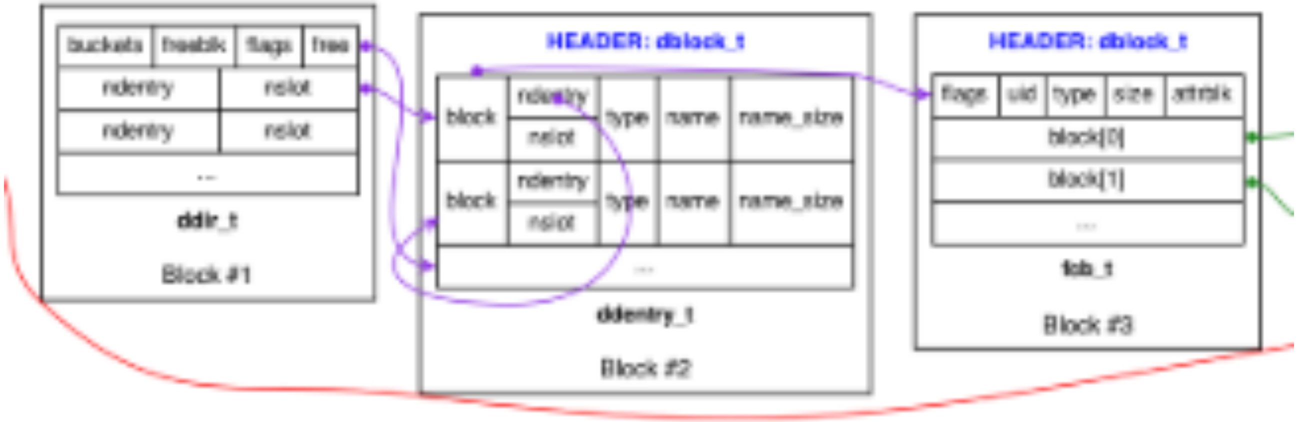
- A file control block (fcb_t) is associated with data blocks that store its data
  - Block[i] – file data
  - Regular file – file data being stored
  - Symbolic link – target file path

P3 Tasks



- (T1) Create a symbolic link (file)
  - Like creating a new regular file (fcb_t)
  - UID owner
  - Type – for a symbolic link file
  - Block[i] – symbolic link data - target file path

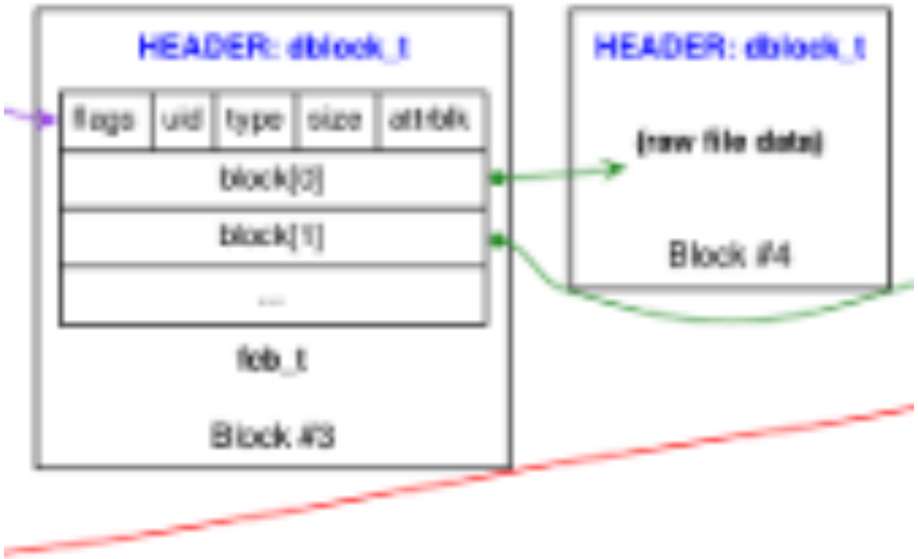
P3 Tasks



hash table

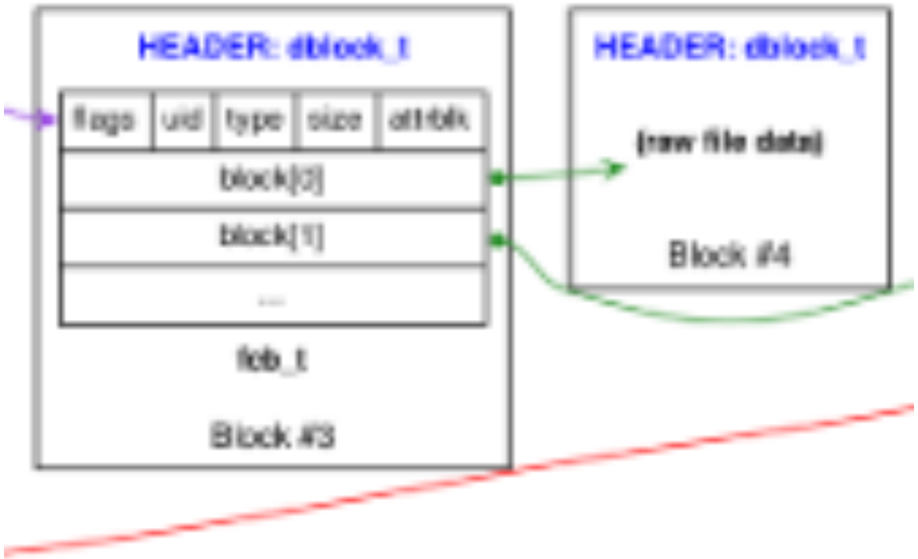
- **(T2) Create a subdirectory (of ddir_t)**
  - **Type of ddentry – directory**
  - **Name of ddentry - directory name**
  - **Block of ddentry – reference a new ddir_t**

P3 Tasks



- **(T3) Resolve pathname that includes a link**
  - **Function: fsResolveName**
  - **UID owner**
  - **Type – for a symbolic link file**
  - **Block[i] – symbolic link data - target file**

P3 Tasks



- (T4) Prevent unsafe actions in resolution
  - – Function: fsResolveName
  - - FLAG_NOFOLLOW – no links in resolution
  - - ”UNTRUSTED – link owner == target owner
  - - FLAG_SAVEDNAME – name == fcb name

from beginning to end →

Contiguous Allocation

- Allocate a sequence of contiguous blocks to a file.
- Advantages:

|A|A|A|A|B|B|B|B|B|C|C

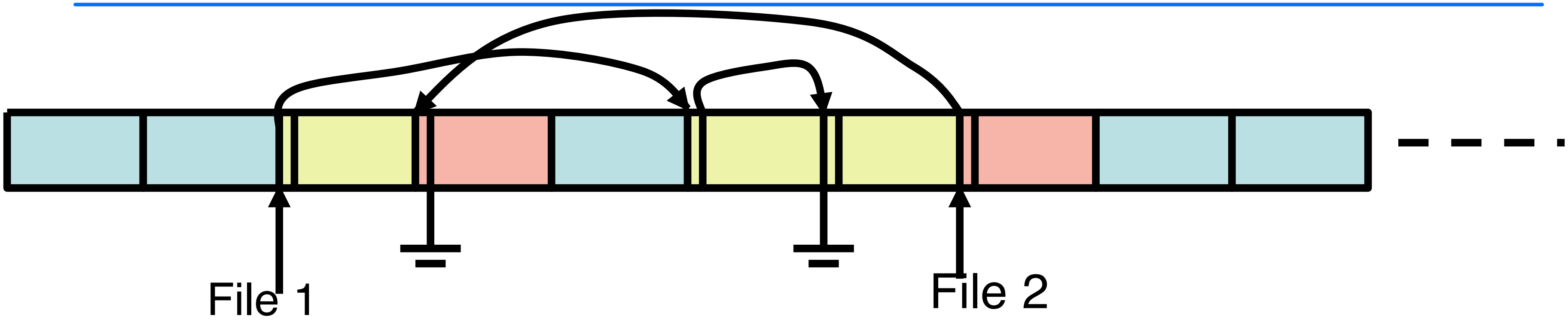
  - Need to remember only starting location to access any block
  - Good performance when reading successive blocks on disk
- Disadvantages:
  - File size has to be known a priori.
  - External fragmentation

"." current directory

".." parent directory

Linked List Allocation

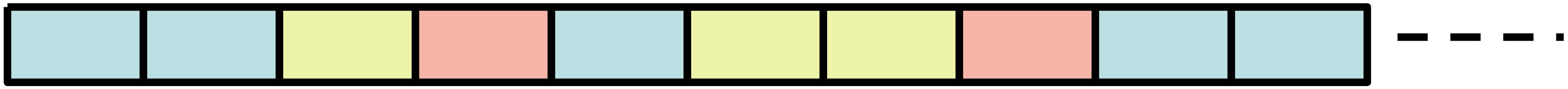
- **Keep a pointer to first block of a file.**
- **The first few bytes of each block point to the next block of this file.**
- **Advantages: No external fragmentation**
- **Disadvantages: Random access is slow!**



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



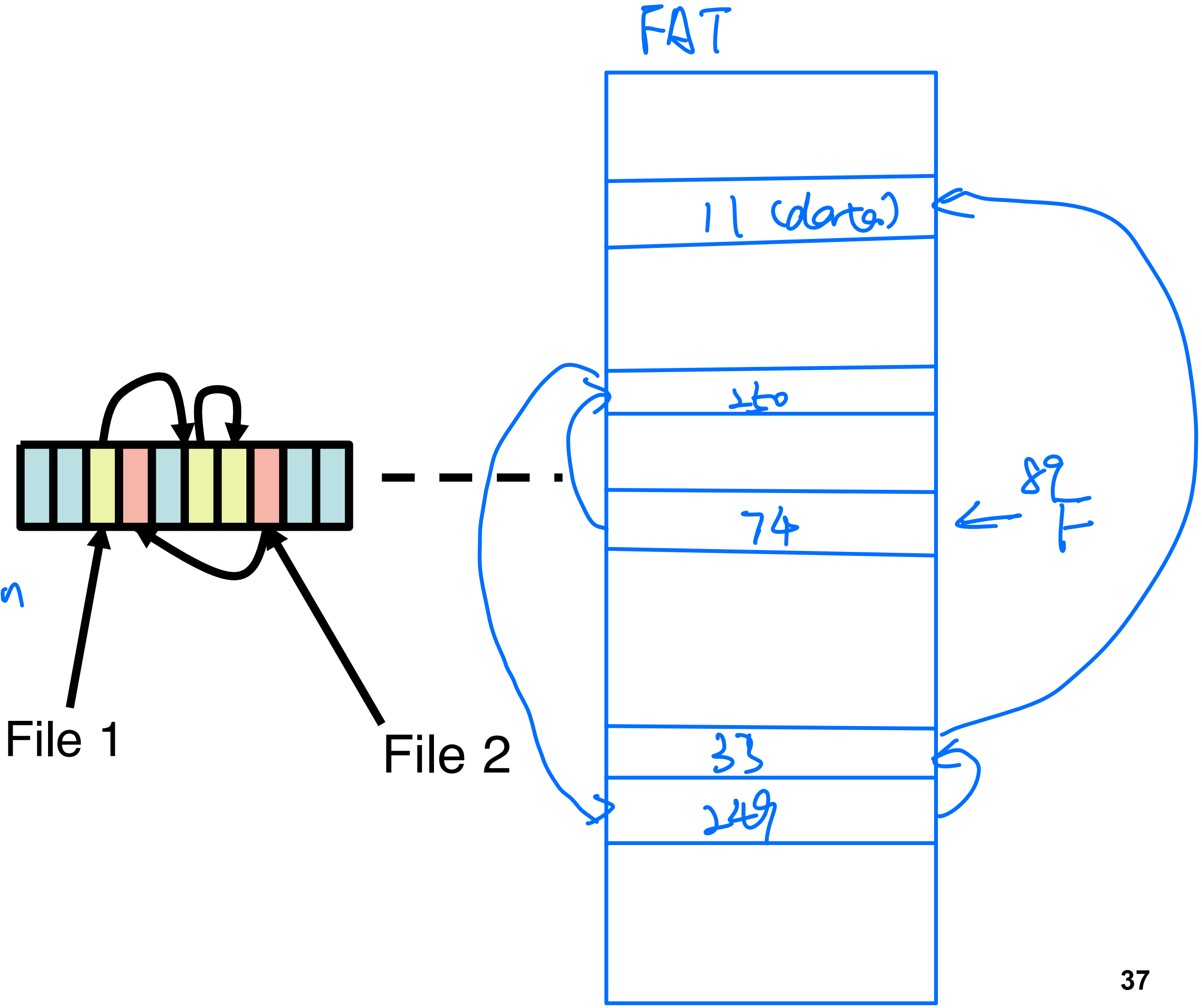


Table of Pointers (in memory?)

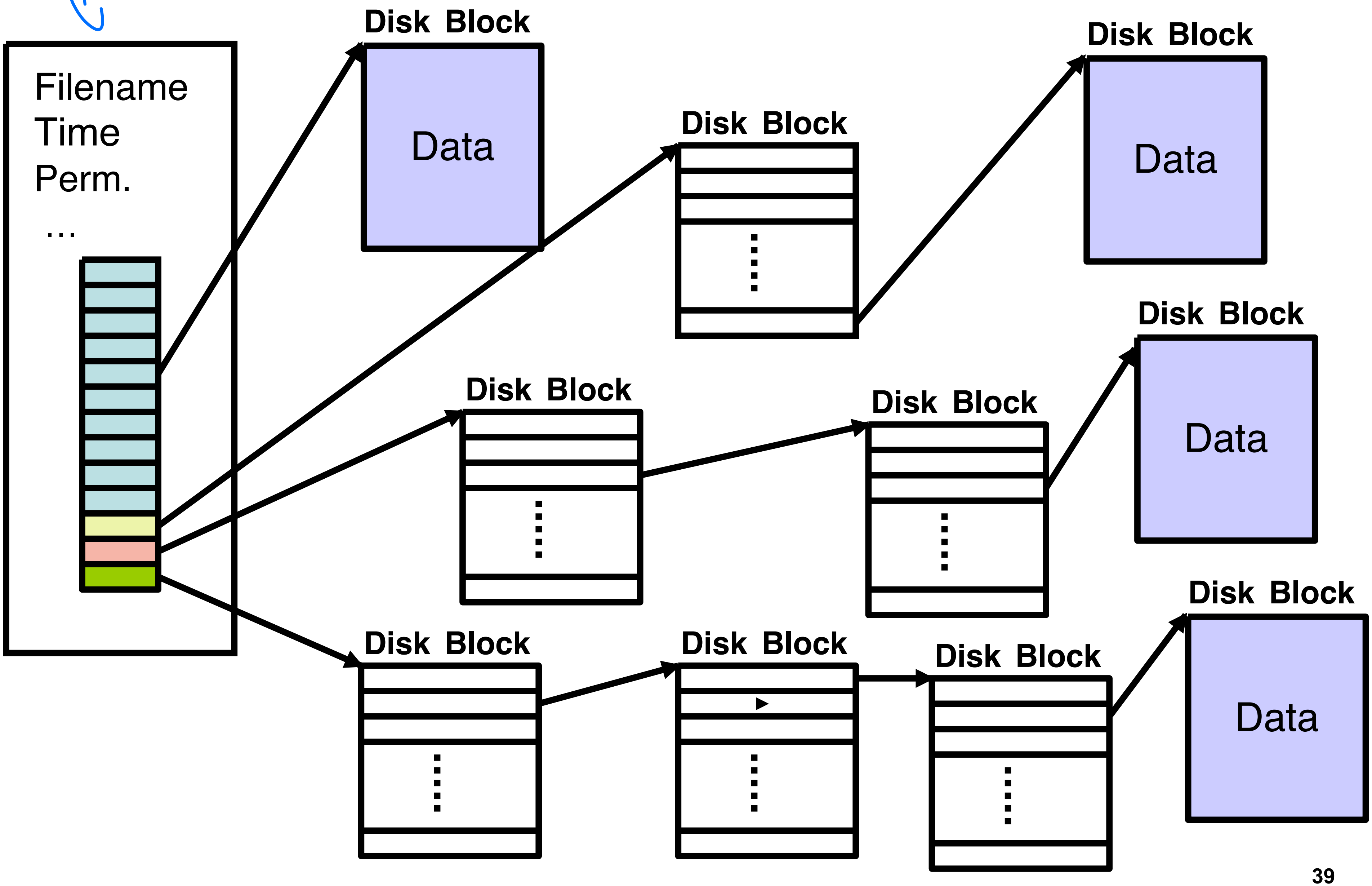
Called FAT in DOS - File allocation table

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) → first 10 blocks of data
  - 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)

i-node

fopen


Filename
Time
Perm.
...
Disk Block
Data
Disk Block
Disk Block
Data
Disk Block
Disk Block
Disk Block
Data
Disk Block
Disk Block
Disk Block
Disk Block
Data

- **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)** = 1 ”

- **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)

bitmap

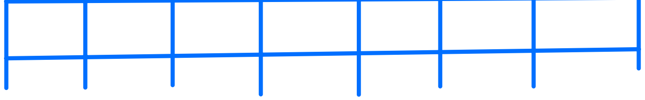
0	1	0	0	1	1	0

1 TB

1 KB/Block

1 Billion Bytes

8 bit/Byte



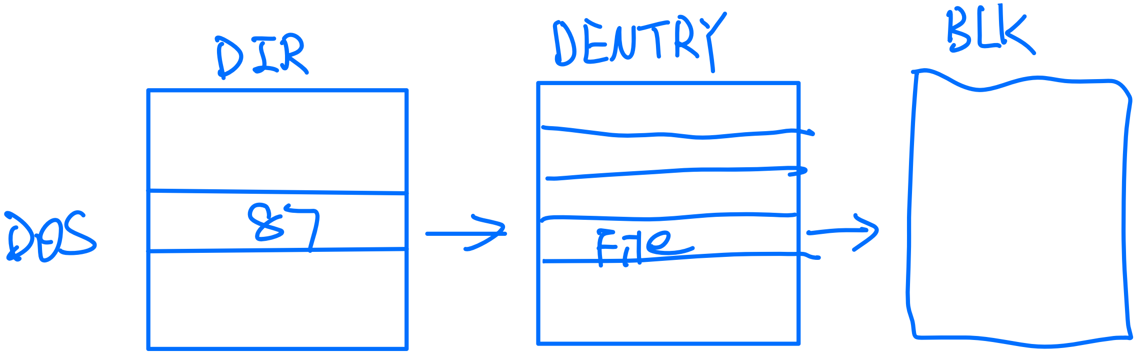
125 MB

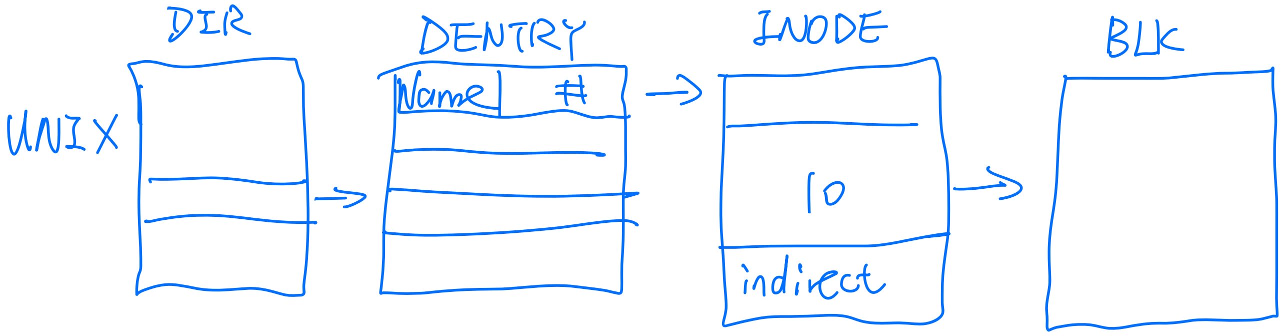
- Now we know how to retrieve the blocks of a file once we know:
  - The FAT entry for DOS
  - The i-node of the file in UNIX

- 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 #]



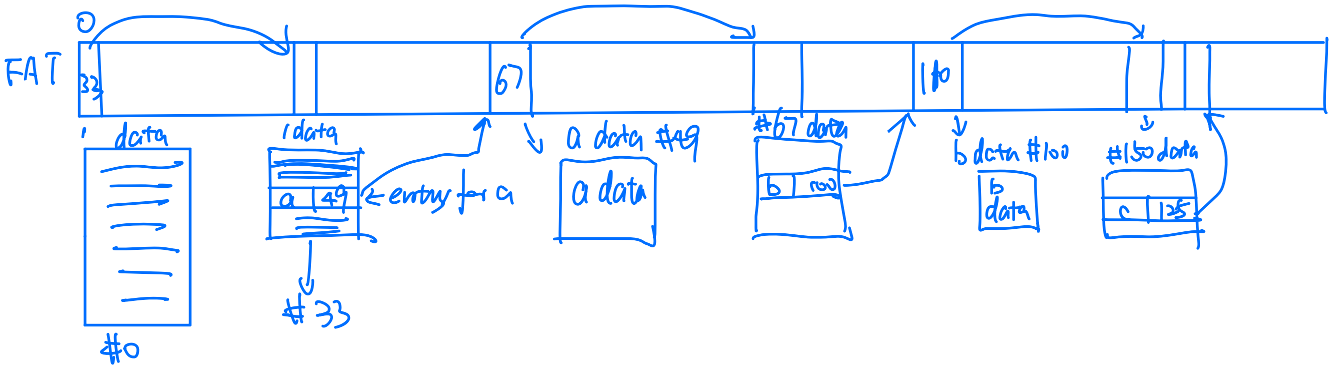

UNIX
DIR
DENTRY
Name
#
INODE
10
indirect
BLK

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 1st 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 1st data block of "b" to check if "c" present.**

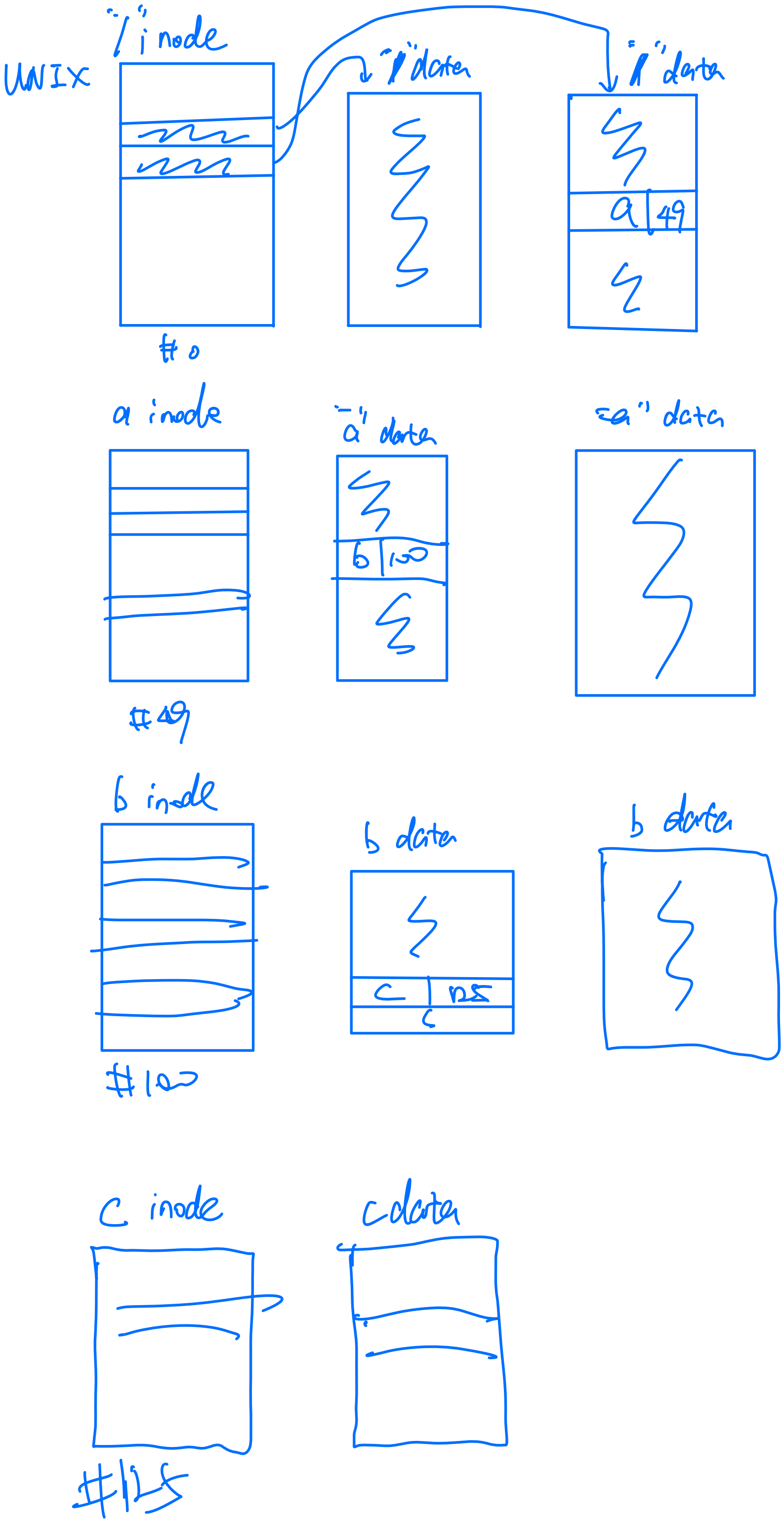
……

- **Read the relevant block of "c", by chasing the FAT entries in memory.**


FAT
0
32
data
#0
data
a | 49
← entry for a
#33
67
a data #49
a data
#67 data
b | 100
150
b data #100
b
data
#150 data
c | 125

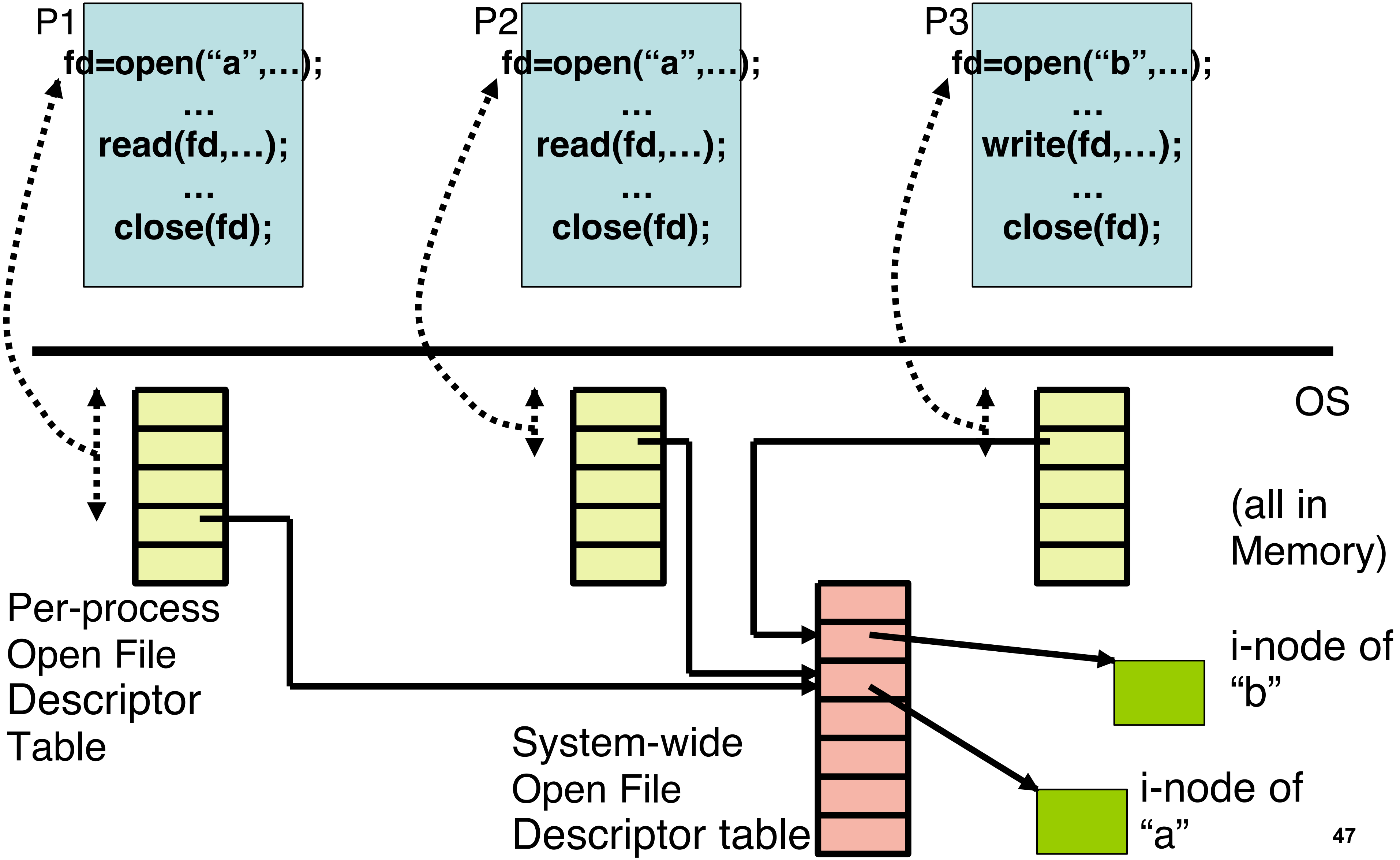
Accessing a file block in UNIX /a/b/c

- 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 1st 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.


UNIX
"/" inode
"/" data
"/" data
a | 49
#0
a inode
"a" data
b | 100
"a" data
#49
b inode
b data
c | 125
b data
#100
c inode
c data
#125

- 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 (or the id of 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

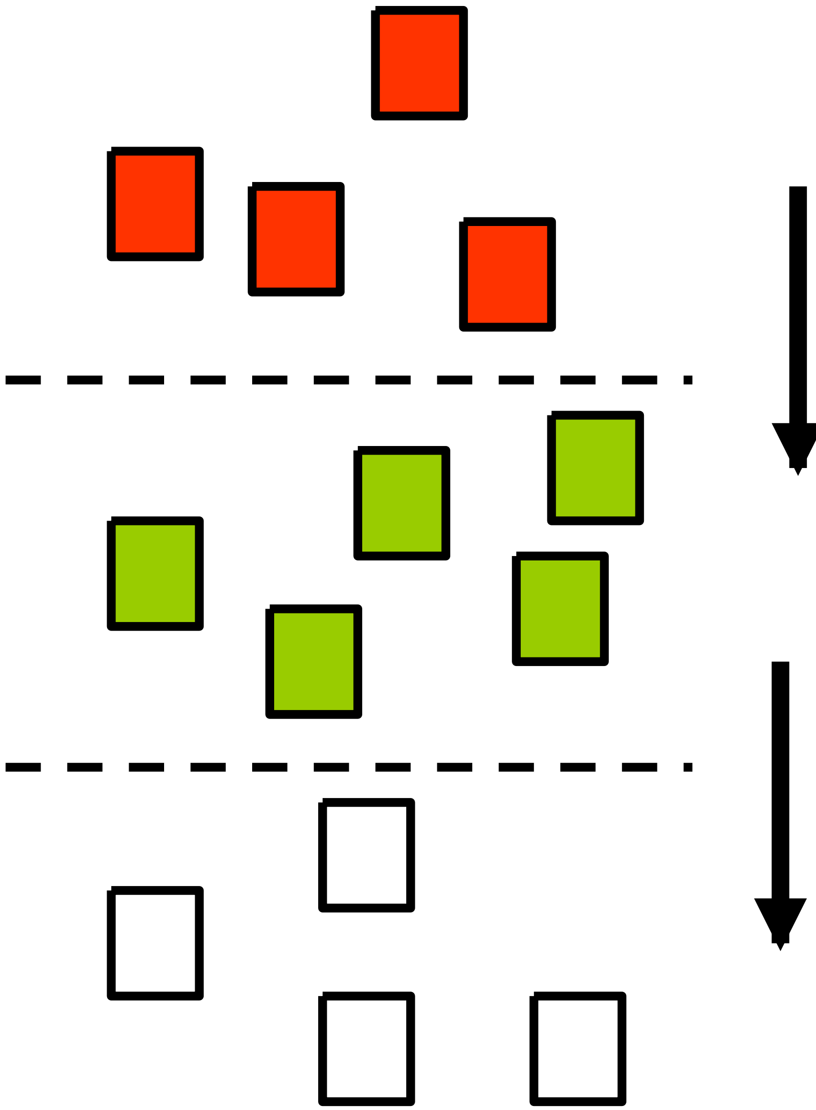
- Cache disk blocks that are in need in physical memory.
- On a read() system call, first look up this cache to check if block is present.
  - This is done in software
  - Look up is done based on logical block id.
  - Typically perform some kind of “hashing”
- If present, copy this from OS cache/buffer 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 loose data that is written if machine goes down before write-back
  - With write-through, you may be loosing 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



Dirty Cached Blocks

Flusher()
Propagates writes to disk.
Done in background periodically

Clean Cached Blocks

Replace/Evict()
Creates free blocks
Called when free list < low water mark, and it keeps evicting till free list >= high water mark

Free List

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.

Links in UNIX

- **Makes a file appear in more than 1 directory.**
- **Is a convenience in several situations.**
- **2 types of links:**
  - **Soft links**
  - **Hard links**

- Soft links:
  - Create a file which contains the name of the other file.
  - Use this path name to get to the actual file when it is accessed.
  - Problem: Extra overhead of parsing and following components till the file is reached.

- Hard links:
  - Create a directory entry and make the inode pointer same as the other.
  - Problem: What if the creater wants to delete the file?
    - We cannot free up the inode.
    - Instead we have to still keep the file.
    - Done by keeping a counter in the inode that is incremented for each hard link. On removing a link, decrement counter. Only if counter is 0, remove i-node.

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().

↑
file system consistency check

Fsck – File System Consistency Check

- **Blocks:**
  - for every block keep 2 counters:
    - a) # occurrences in files
    - 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.

Fcsk- File System Consistency Check

- **Files:**
  - Maintain a counter for each inode.
  - Recursively traverse the directory hierarchy.
  - For each file, 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)

[illegible]

[illegible]

① FCFS

- service order : 1, 36, 16, 34, 9, 12
- seek cost = $10 + 35 + 20 + 18 + 25 + 3 = 111$ cylinders

  $(11 \to 1)\ (1 \to 36)\ (36 \to 16)\ (16 \to 34)\ (34 \to 9)\ (9 \to 12)$

② SSTF (shortest Seek Time first) — may lead to starvatation
violate bounded wait

- service order : 12, 9, 16, 1, 34, 36
- seek cost = $1 + 3 + 7 + 15 + 33 + 2 = 61$

  $(11 \to 12)\ (12 \to 9)\ (9 \to 16)\ (16 \to 1)\ (1 \to 34)\ (34 \to 36)$

③ SCAN Algorithm

service order : 12 16, 34, 36, 9, 1

④ C-SCAN : $1 + 4 + 18 + 2 + 63 + \overset{(99 \to 9)}{\underline{90}} + 1 + 8 = 186$

service order : 12, 16, 34, 36, 1, 9

cost : $1 + 4 + 18 + 2 + 63 + \underset{(99 \to 0)}{\underbrace{99}} + 1 + 8 = 197$

⑤ Look (after reach max go back

service order : 12, 16, 34, 36, 9, 1

seek cost = $1 + 4 + 18 + 2 + \underset{(36 \to 9)}{27} + 8 = 60$ cylinder

⑥ c-look (after reach max, go to the lowest request)

service order : 12, 16, 34, 36, 1, 9.

seek cost = 1+4+18+2+35+8=68 cylinders