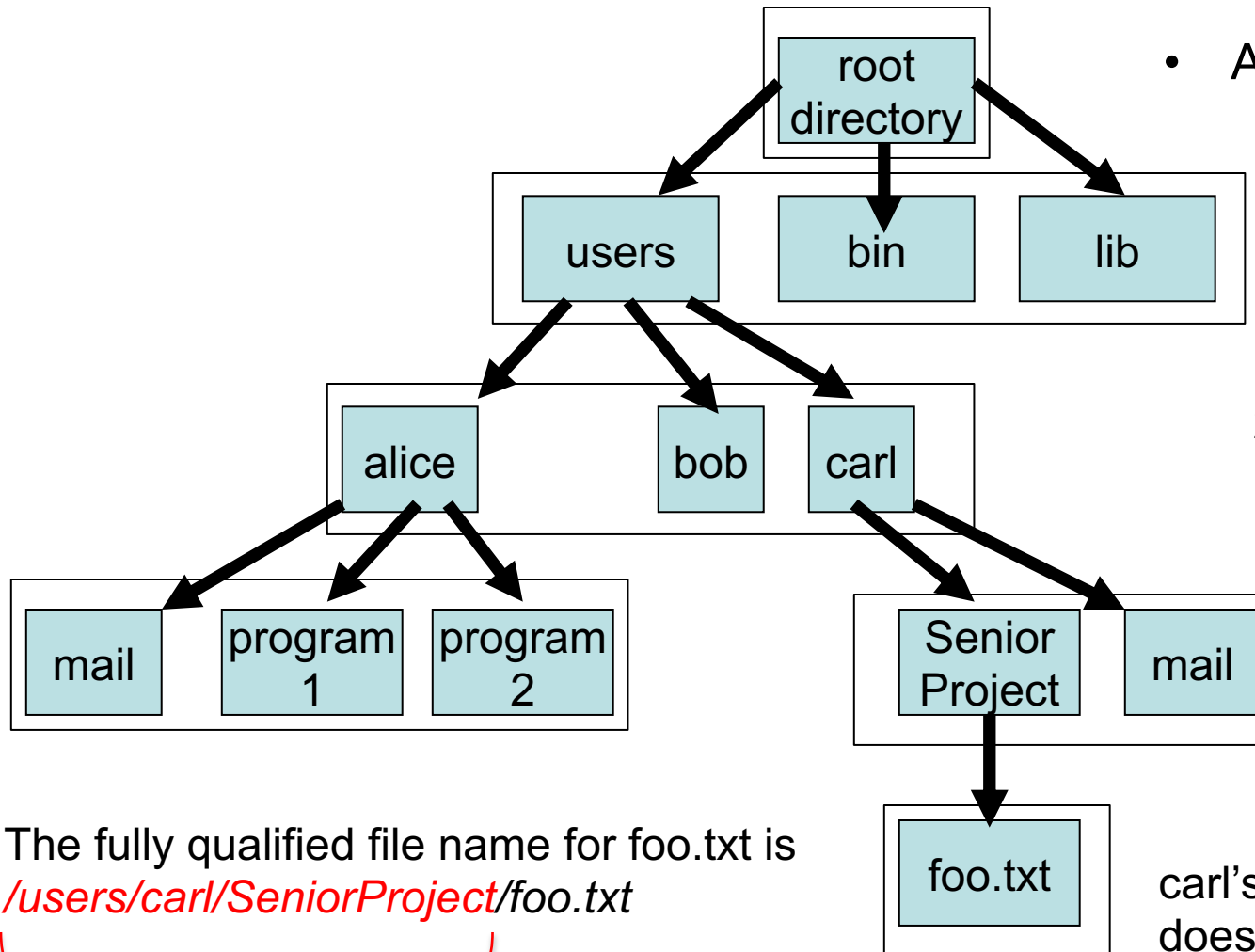




# **Lecture 26**

## **File System Implementation**

# Tree-structured Directory



- Advantages:
  1. hierarchical & customized organization of files by each user
  2. Unique naming
    - no name conflicts
  3. users can share & access files in other directories


The fully qualified file name for foo.txt is

*/users/carl/SeniorProject/foo.txt*

*“path” or directory name*

carl's file name “mail”  
does not conflict  
with alice's file name “mail”

# File System Implementation

- File system elements are stored on *both*:
  - Disk/flash – persistent storage
  - Main memory/RAM – volatile storage
- On *disk/flash*, **the entire file system is stored**, including 5 main elements:
  1. its entire directory tree structure
  2. each file's attributes are in a File Control Block 
  3. each file's data
  4. a *boot block*, typically the first block of a volume, that contains info needed to boot an operating system from this volume. Empty if no OS to boot.
  5. a *volume control block* that contains volume or partition details,  
e.g. tracks free blocks on disk,  
the number of blocks in a partition,  
size of a block, etc.

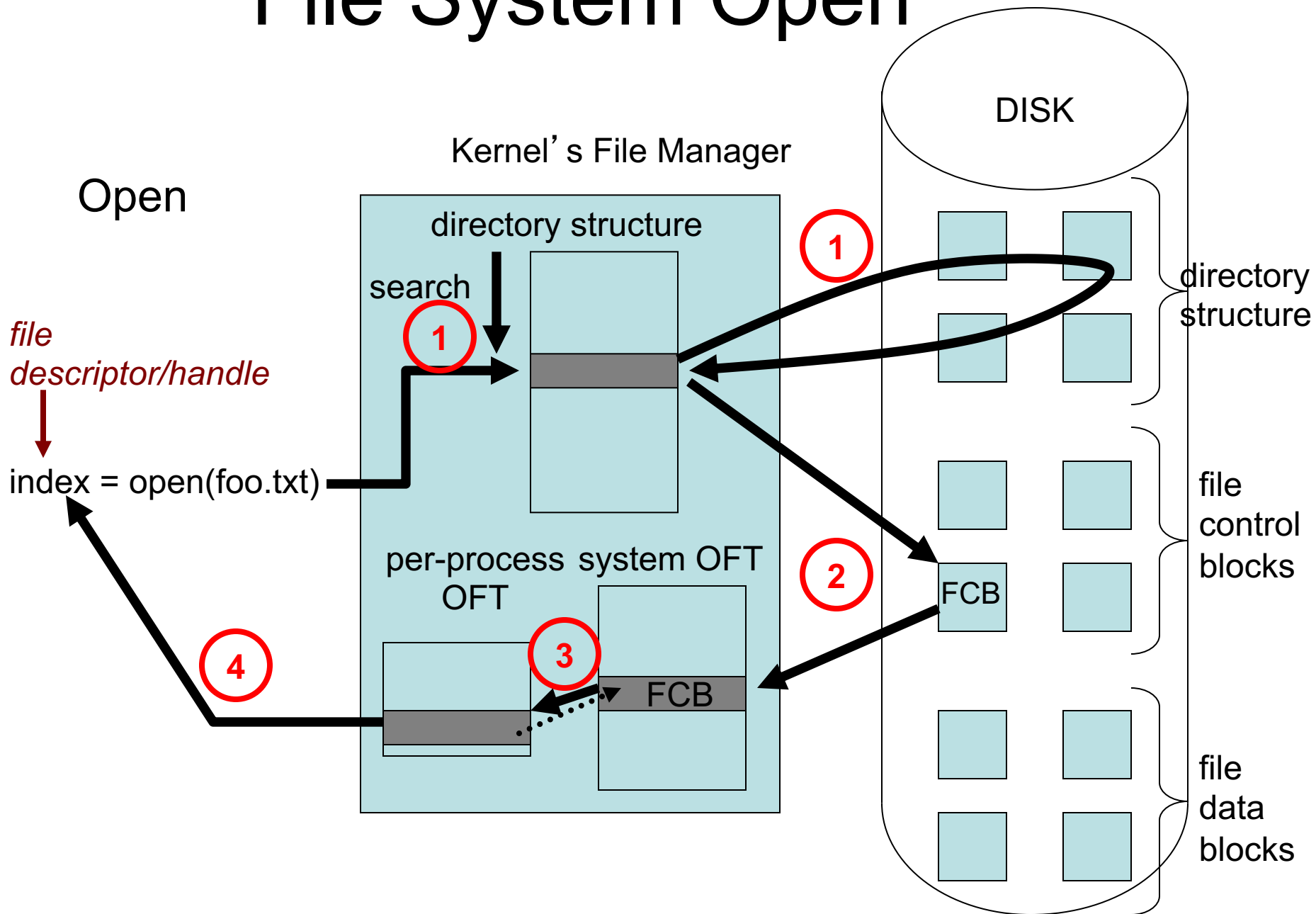
example FCB

name
unique ID
file permissions
dates (created,...)
size
location on disk

# The four main file system components in *memory* are:

1. **Recently accessed parts of the directory structure tree** are stored in memory – for faster look up
2. OS also maintains a ***system-wide open file table (OFT)*** that tracks process-independent info of open files
  - the file header containing attributes about the open file is stored here
  - an open count of the number of processes that have files open is stored here
3. OS also maintains a ***per-process OFT*** - tracks all files that have been opened by a particular process, may store access rights, etc.
  - Also keeps a *current-file-position pointer*, i.e. where in the file the process is currently reading/writing
4. OS keeps a ***mount table of devices with file systems*** that have been mounted as volumes

# File System Open



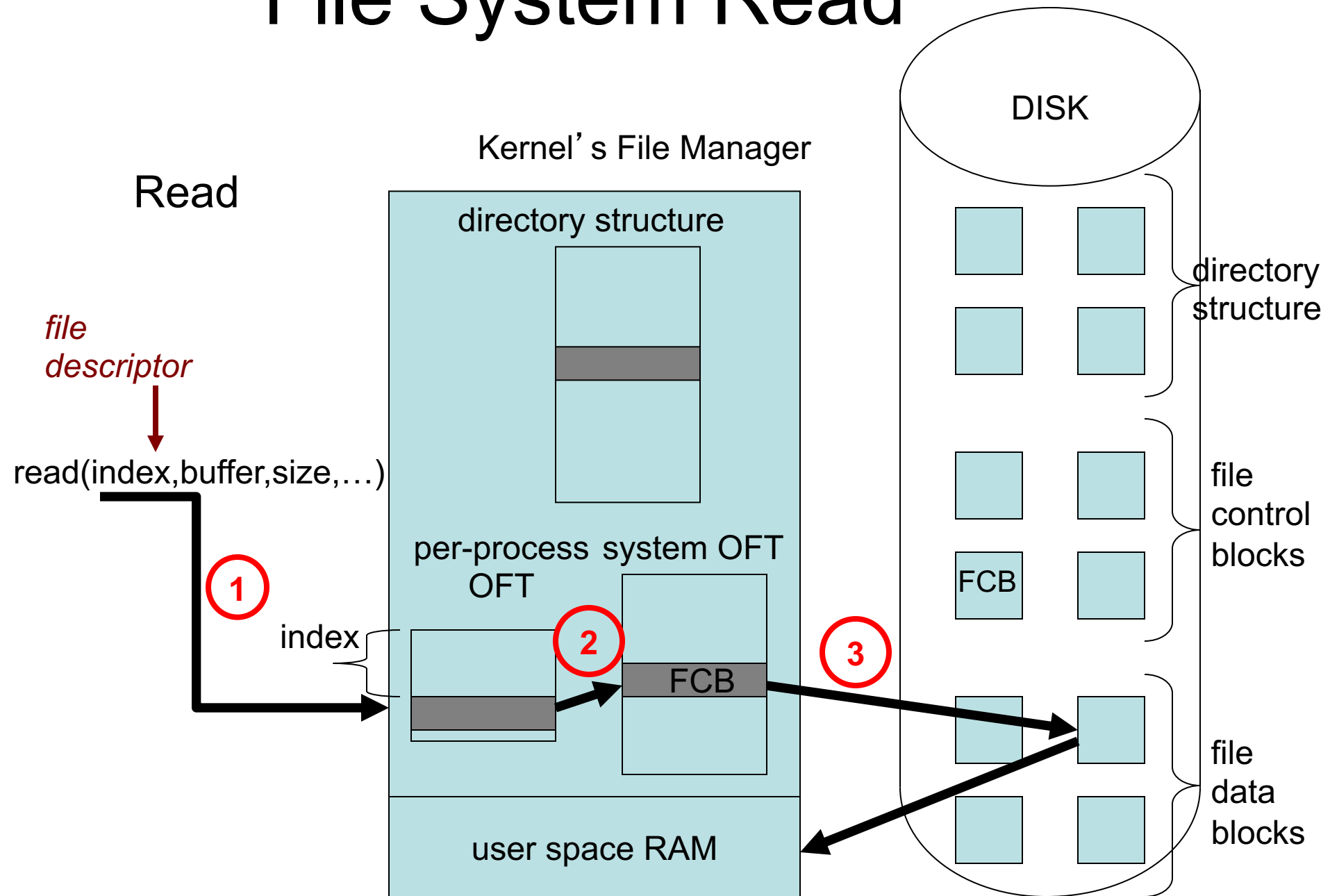
# File System Open (steps)

- When a process calls `open(foo.txt)` to set up access to a file, the following procedural steps are followed:
  1. the directory structure is searched for the file name `foo.txt`
    - if the directory entries are in memory, then the search is fast
    - otherwise, directories and directory entries have to be retrieved from disk and cached for later accesses
  2. once the file name is found, the directory entry contains a pointer to the FCB on disk
    - retrieve the FCB from disk
    - copy the FCB into the system OFT. This acts as a cache for future file opens.
    - Increment the open file counter for this file in the system OFT
  3. add an entry to the per-process OFT that points to the file's FCB in the system OFT
  4. return a file descriptor or handle to the process that called `open()`

# File System Open

- Some OS's employ a mandatory lock on an open file
  - Only one process at a time can use an open file
  - Windows policy
- Other OS's allow optional or advisory locks
  - UNIX policy
  - It's up to users to synchronize access to files

# File System Read





# File System Close

- on a close(),
  1. remove the entry from the per-process OFT
  2. decrement the open file counter for this file in the system OFT
  3. if counter = 0, then write back to disk any metadata changes to the FCB, e.g. its modification date
    - Note: there may be a temporary inconsistency between the FCB stored in memory and the FCB on disk – designers of file systems need to be aware of this. A similar inconsistency occurred for modified memory-mapped file data in RAM that had not yet been written to disk.



# **File Allocation**

# File Allocation

## Approaches:

### 1. Contiguous file allocation

- a file is laid out contiguously, i.e. if a file is  $n$  blocks long, then a starting address  $b$  is selected and the file is allocated blocks  $b, b+1, b+2, \dots, b+n-1$

### 2. Linked Allocation

- each file is a linked list of disk blocks

### 3. File Allocation Table (FAT) is an important variation of linked lists

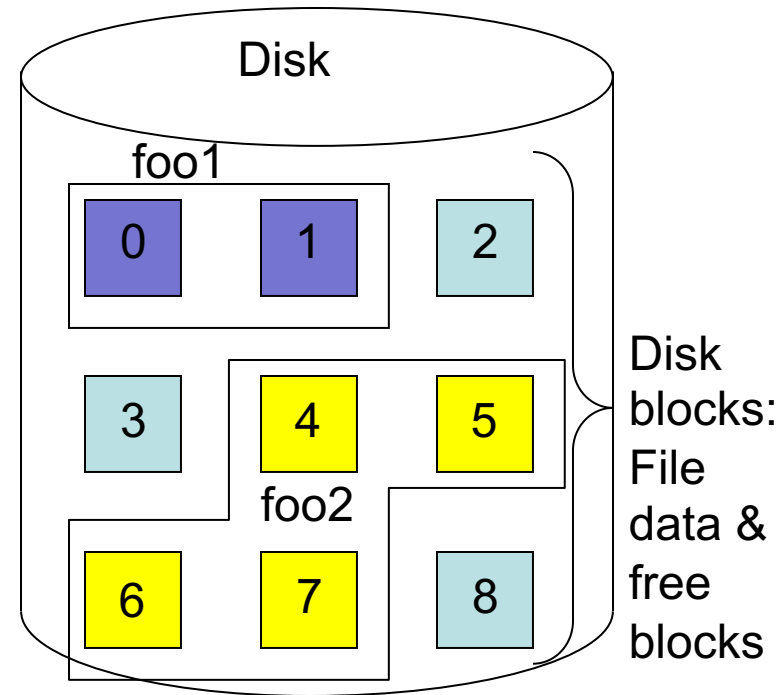
- Don't embed the pointers of the linked list with the file data blocks themselves
- Instead, separate the pointers out and put them in a special table – the file allocation table (FAT)
- The FAT is located at a section of disk at the beginning of a volume

### 4. Indexed Allocation

- collect all pointers into a list or table called an *index block*
- the index  $j$  into the list or index block retrieves a pointer to the  $j$ 'th block on disk

# Approach #1: Contiguous File Allocation

File headers		
file	start	length
foo1	0	2
foo2	4	4



- Advantage: fast performance (low seek times because the blocks are all allocated near each other on disk)

# Approach #1: Contiguous File Allocation

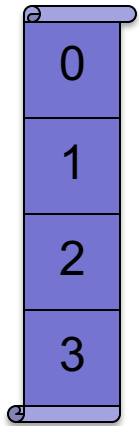
- Disadvantages:
  - Problem 1: external fragmentation (same problem as trying to contiguously fit processes into RAM)
    - same solutions apply: first fit, best fit, etc.
    - also compact memory/defragment disk
    - can be performed in the background late at night, etc.
  - Problem 2: May not know size of file in advance
    - allocate a larger size than estimated
    - if file exceeds allocation, have to copy file to a larger free “hole” between allocated files
  - Problem 3: Over-allocation of a “slow growth” file
    - A file may eventually need 1 million bytes of space
    - But initially, the file doesn’t need much, and it may be growing at a very slow rate, e.g. 1 byte/sec
    - So for much of the lifetime of the file, allocating 1 MB wastes allocation

# General File Allocation

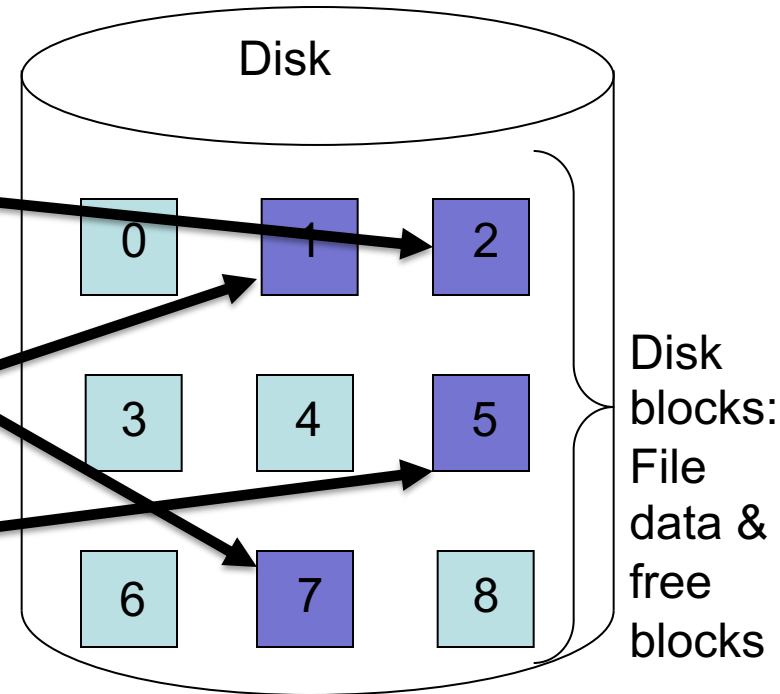
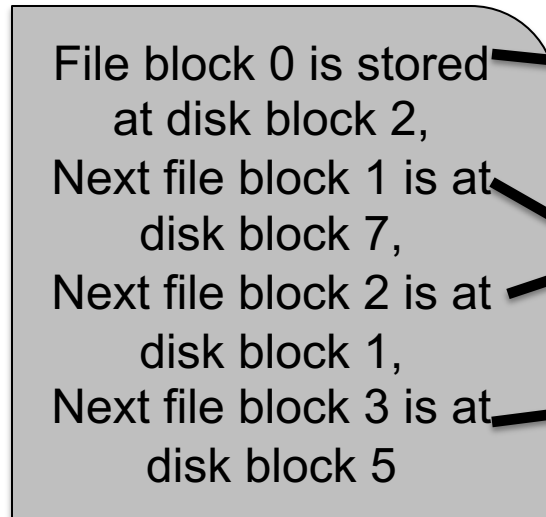
- Page table solved external fragmentation problem for process allocation
- Apply a similar concept to file allocation
  - Divide disk into fixed-sized blocks, just as main memory was divided into fixed-sized physical frames
  - Allow a file's data blocks to be spread across any collection of disk blocks, not necessarily contiguous
  - *Need a data structure to keep track of what block of a file is stored on which block in disk*

# General File Allocation

File “foo1.txt”



Generic Data Structure

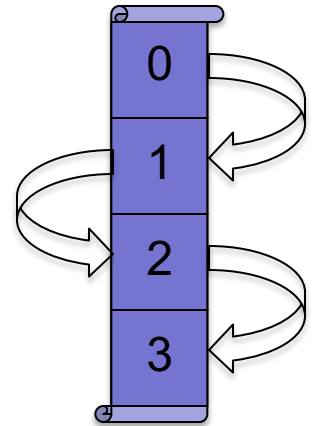


- Generic data structure can be:
  - A Linked list and variants
  - Indexed allocation (somewhat resembles a page table) and variants

# Approach #2: Linked File Allocation

- Linked Allocation
  - each file is a linked list of disk blocks
  - to add to a file, just modify the linked list either in the middle or at the tail, depending on where you wish to add a block
  - to read from a file, traverse linked list until reaching the desired data block

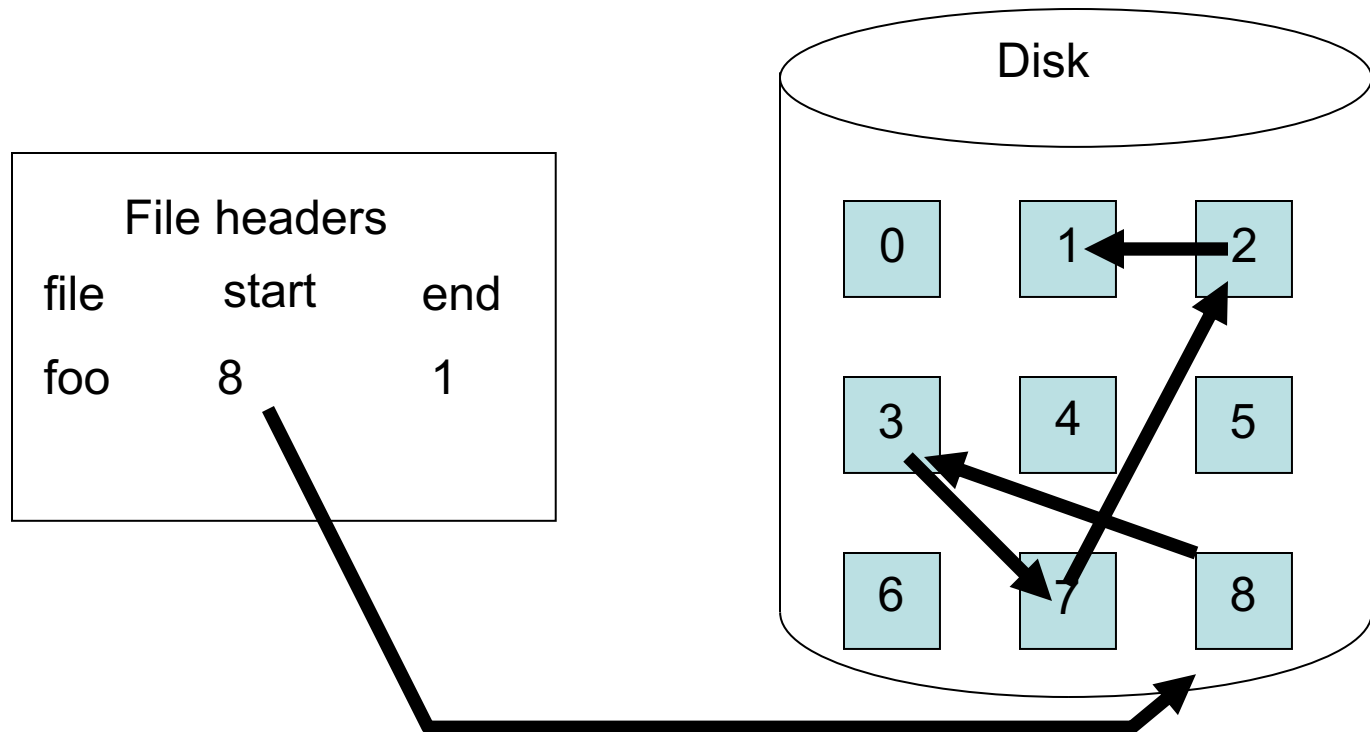
File “foo1.txt”





# Linked File Allocation

- Linked Allocation
  - each file is a linked list of disk blocks



# Linked File Allocation

- Advantages:
  - solves problems of contiguous allocation
    - no external fragmentation
    - don't need to know size of a file a priori
  - Minimal bookkeeping overhead in file header – just a pointer to start of file on disk
    - (-) Compromise is that all the pointer overhead is stored in each disk block
  - Good for sequential read/write data access
  - Easy to insert data into middle of linked list

# Linked File Allocation

- Disadvantages:
  - performance of random (direct) data access is extremely slow for reads/writes
    - because you have to traverse the linked list until indexing into the correct disk block
  - Space is required for pointers on disk in every disk block
  - reliability is fragile
    - if one pointer is in error or corrupted, then lose the rest of the file after that pointer

# File Allocation

## Approaches:

### 1. Contiguous file allocation

- a file is laid out contiguously, i.e. if a file is  $n$  blocks long, then a starting address  $b$  is selected and the file is allocated blocks  $b, b+1, b+2, \dots, b+n-1$

### 2. Linked Allocation

- each file is a linked list of disk blocks

### 3. File Allocation Table (FAT) is an important variation of linked lists

- Don't embed the pointers of the linked list with the file data blocks themselves
- Instead, separate the pointers out and put them in a special table – the file allocation table (FAT)
- The FAT is located at a section of disk at the beginning of a volume

### 4. Indexed Allocation

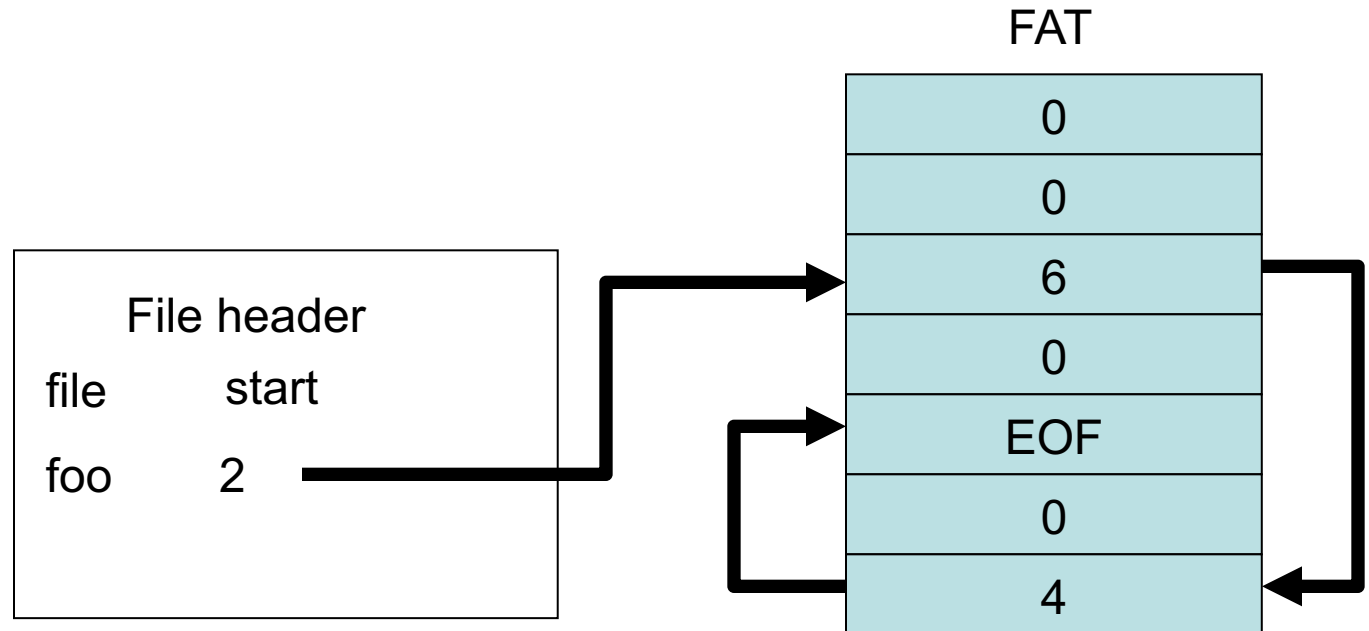
- collect all pointers into a list or table called an *index block*
- the index  $j$  into the list or index block retrieves a pointer to the  $j$ 'th block on disk

# Approach #3: File Allocation Table (FAT)

- the File Allocation Table (FAT) is an important variation of linked lists
  - Don't embed the pointers of the linked list within the file data blocks themselves
  - Instead, separate the pointers out and put them in a special table – the file allocation table (FAT)
  - The FAT is located at a section of disk at the beginning of a volume

# File Allocation Table

- entries in the FAT point to other entries in the FAT as a linked list, but their values are interpreted as the disk block number
- unused blocks in FAT initialized to 0



# File Allocation Table

- FAT file systems used in MS-DOS and Win95/98
  - Bill Gates designed/coded original FAT file system
  - replaced by NTFS (basis of Windows file systems from WinNT through Windows Vista/7)
  - Variants include FAT16, FAT32, etc. FAT16 and FAT32 refer to the size of the address used in the FAT.

# File Allocation Table

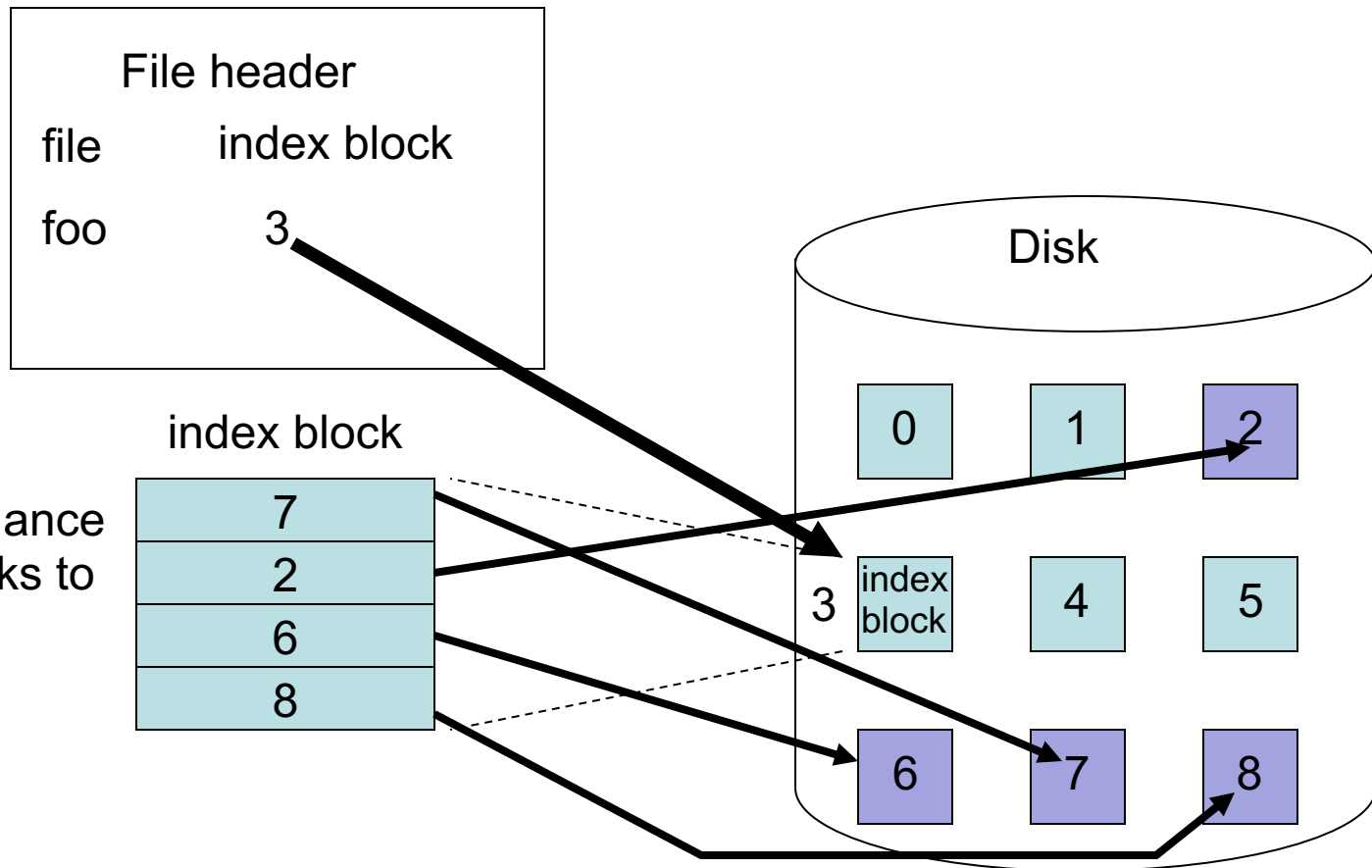
- Linked list for a file is terminated by a special end-of-file EOF value
- (+) Allocating a new block is simple - find the first 0-valued block
- (+) Random Reads/Writes faster than pure linked list
  - the pointers are all colocated in the FAT near each other at the beginning of disk volume - low disk seek time
- (-) still have to traverse the linked list though to find location of data – this is still a slow operation



# Approach #4: Indexed Allocation

- Conceptually, collect all pointers into a list or table called an *index block*
  - the index  $j$  into the list or index block retrieves a pointer to the  $j$ 'th block on disk
  - Looks kind of like a page table, except it's extensible
- Unlike the FAT, the index block can be stored in any block on disk, not just in a special section at the beginning of disk
- Unlike the FAT, the index is just a linear list of pointers

# Indexed Allocation

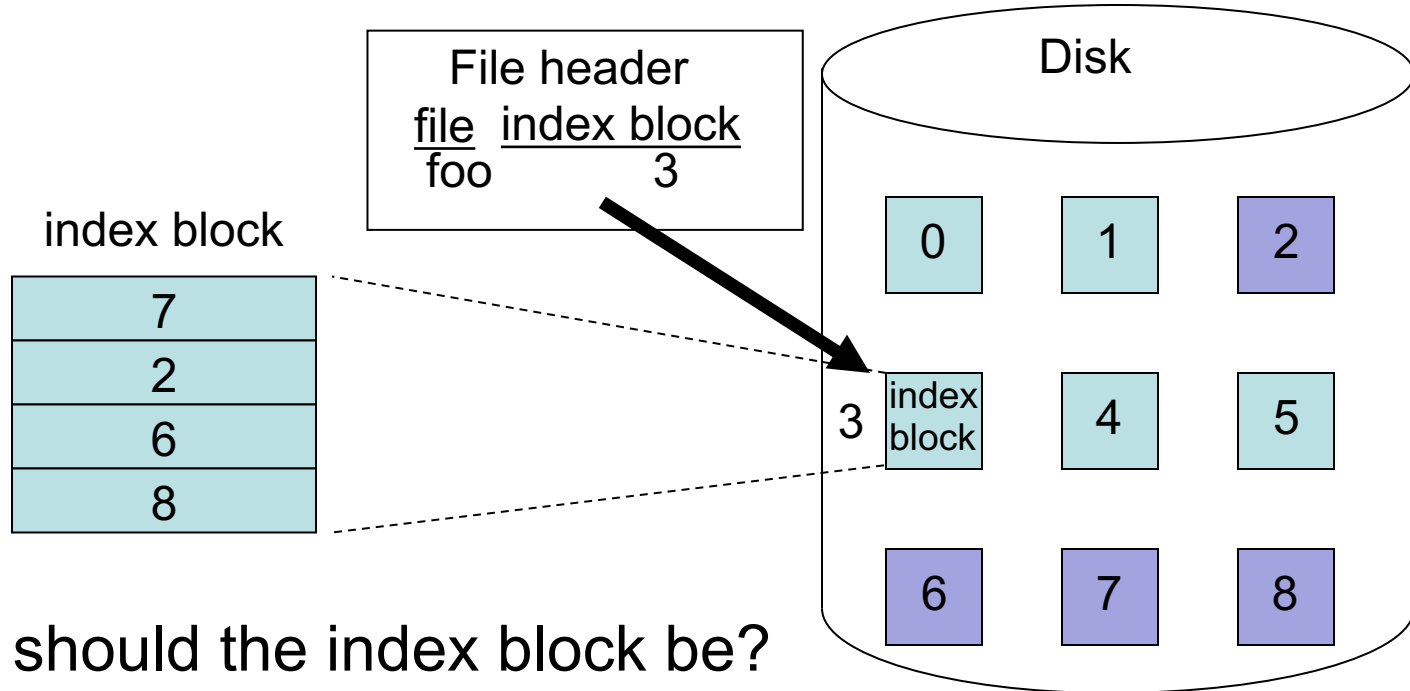


# Indexed Allocation

- Solves many problems of contiguous and linked list allocation:
  - no external fragmentation
  - size of file not required a priori
  - don't have to traverse linked list for random/direct reads/writes
    - just index quickly into the index block

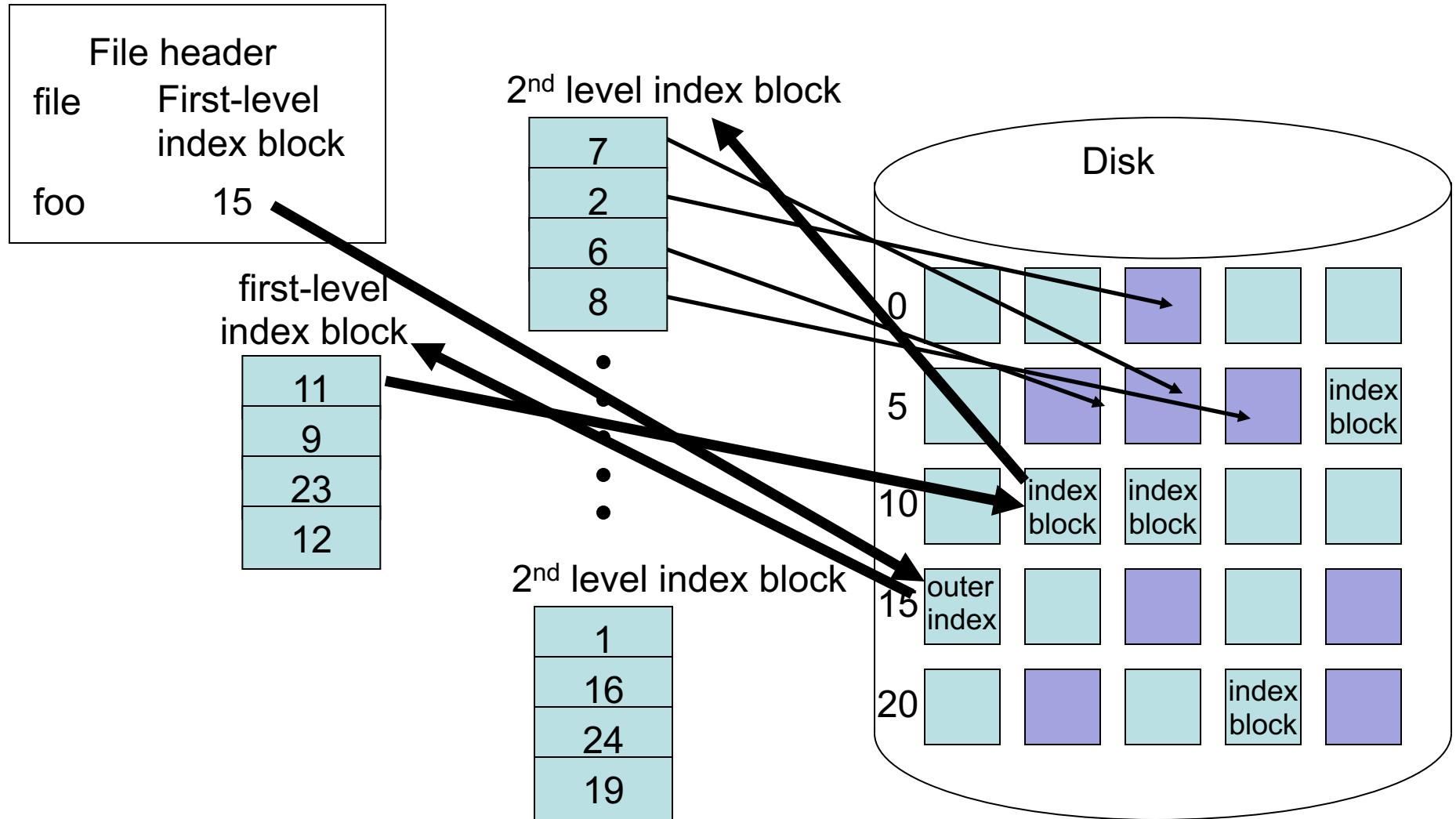
# Indexed Allocation

- Solutions:
  1. link together index blocks –
    - each index block has link to next index block
  2. *multilevel index* (like hierarchical page tables!)
    - First level is list of all index blocks for file
    - Second level is list of all data blocks in that section of the file



How big should the index block be?

# Approach #5: Multilevel Indexed Allocation

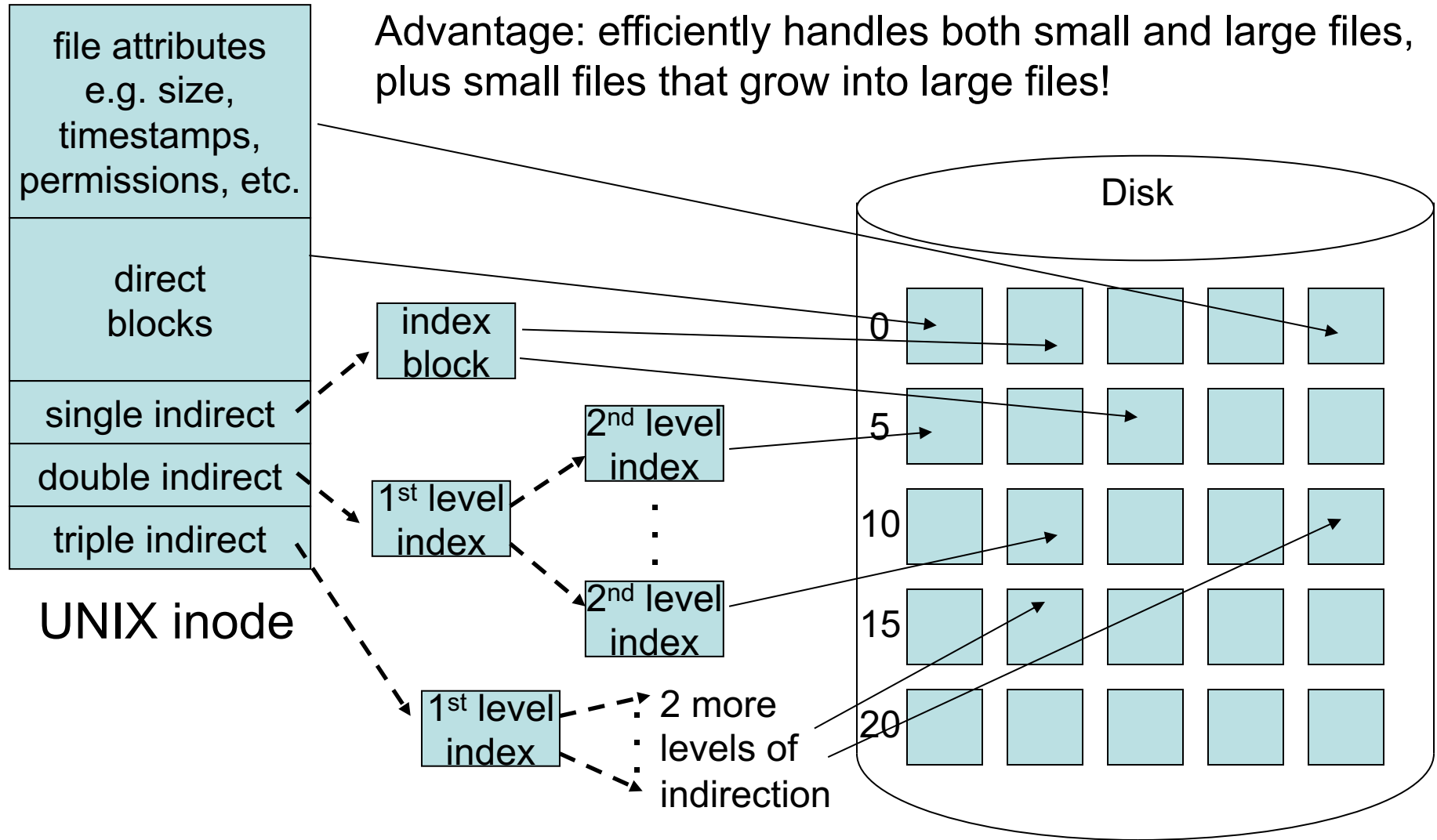


# Multilevel Indexed Allocation

- Problem with multi-level indexing:
  - accessing small files might take just as long as large files
  - have to go through the same # of levels of indexing, hence same # of disk operations
  - accessing the data of a 100 byte file requires at least 4 block reads

# UNIX Multilevel Indexed Allocation

Advantage: efficiently handles both small and large files, plus small files that grow into large files!



# UNIX Multilevel Indexed Allocation

- for small files
  - only uses a small index block of 15 entries, so there is very little wasted memory
- for large files
  - the indirect pointers allow expansion of the index block to span a large number of disk blocks

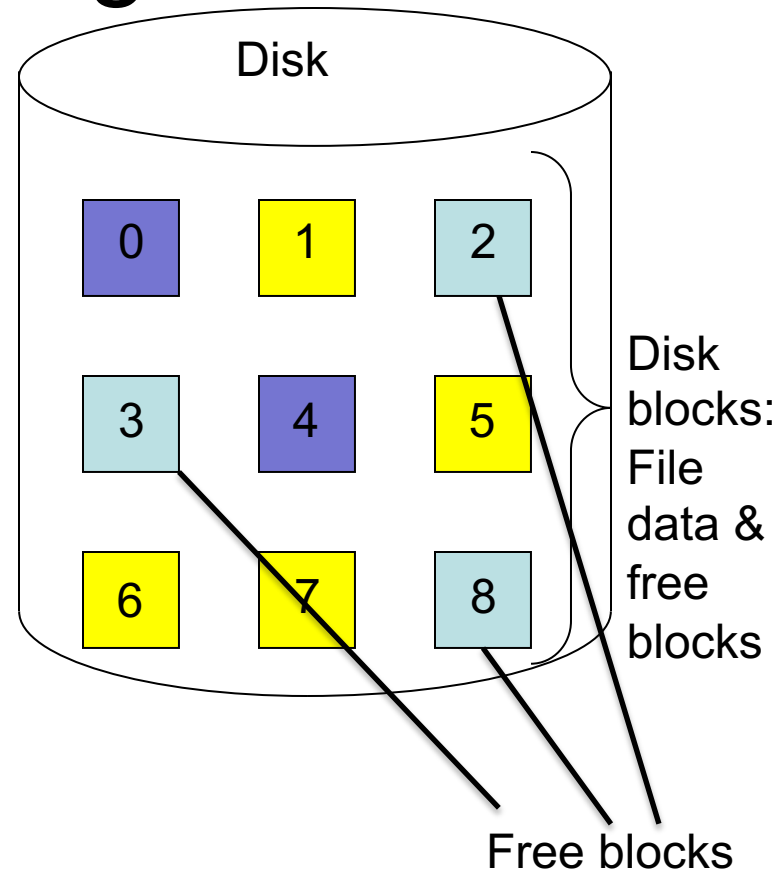


# Comparing File Allocation with Process Allocation

- In both cases, mapping an entity to storage
  - Process address space allocated frames in RAM via page tables
  - File data is allocated to disk/flash
- Differences:
  - Address spaces are fixed in size and known in advance,
  - Files grow/contract over time – files need a mapping/allocation system that is more flexible than page tables, which can't grow
  - Address spaces can be sparse and mostly unused, while file data is all “used”

# Free Space Management

- Another aspect of managing a file system is managing free space
  - the file system needs to keep track of what blocks of disk are free/unallocated
  - keeps a free-space “list”
  - In this example, need to keep track that disk blocks 2, 3 and 8 are free/unallocated



# Free Space Management Approaches

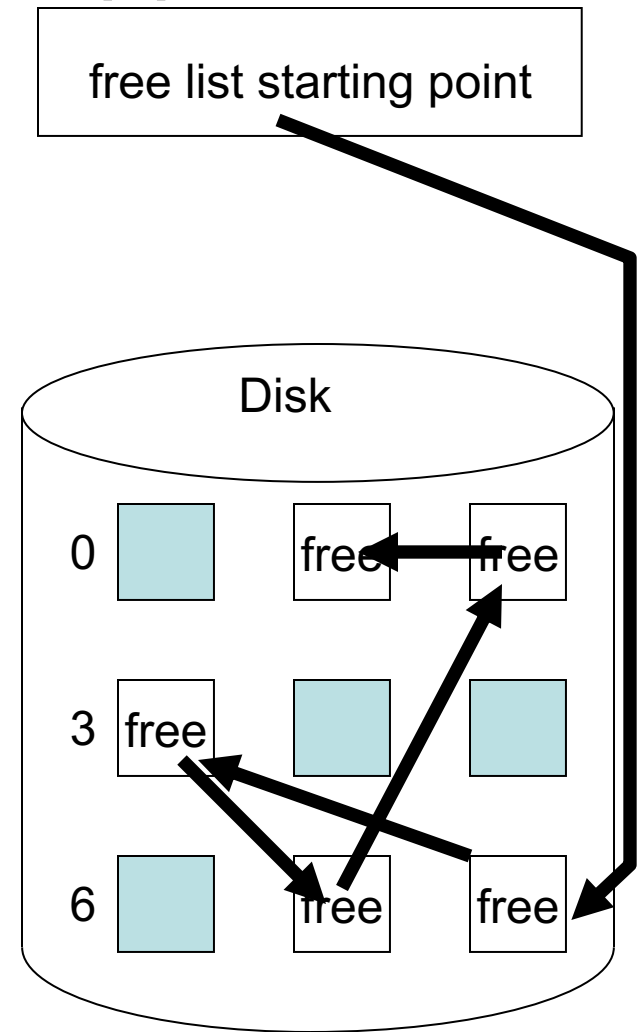
## 1. Bit Vector or Bit Map

- each block is represented by a bit.
- Concatenate all such bits into an array of bits, namely a bit vector.
  - The  $j$ 'th bit indicates whether the  $j$ 'th block has been allocated.
  - if bit = 1, then a block is free, else if bit = 0, then block is allocated

# Free Space Management Approaches

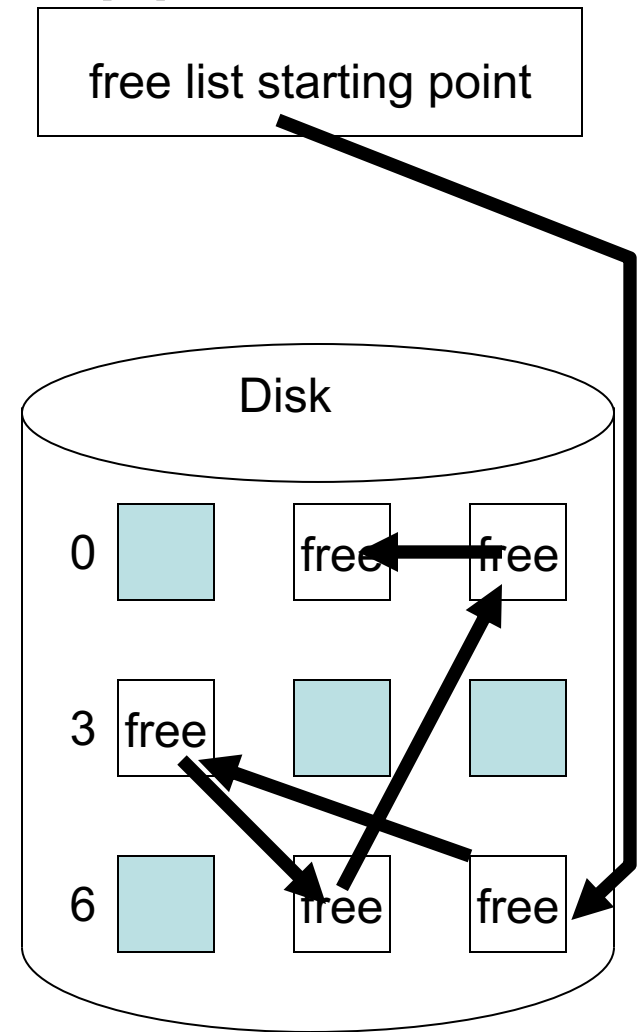
## 2. Linked List

- link together all free blocks
- efficient - keeps track of only the free blocks.
  - bitmap has the overhead of tracking both free and allocated blocks - this is wasteful if memory is mostly allocated
- Faster than bitmap – find 1<sup>st</sup> free block immediately



# Free Space Management Approaches

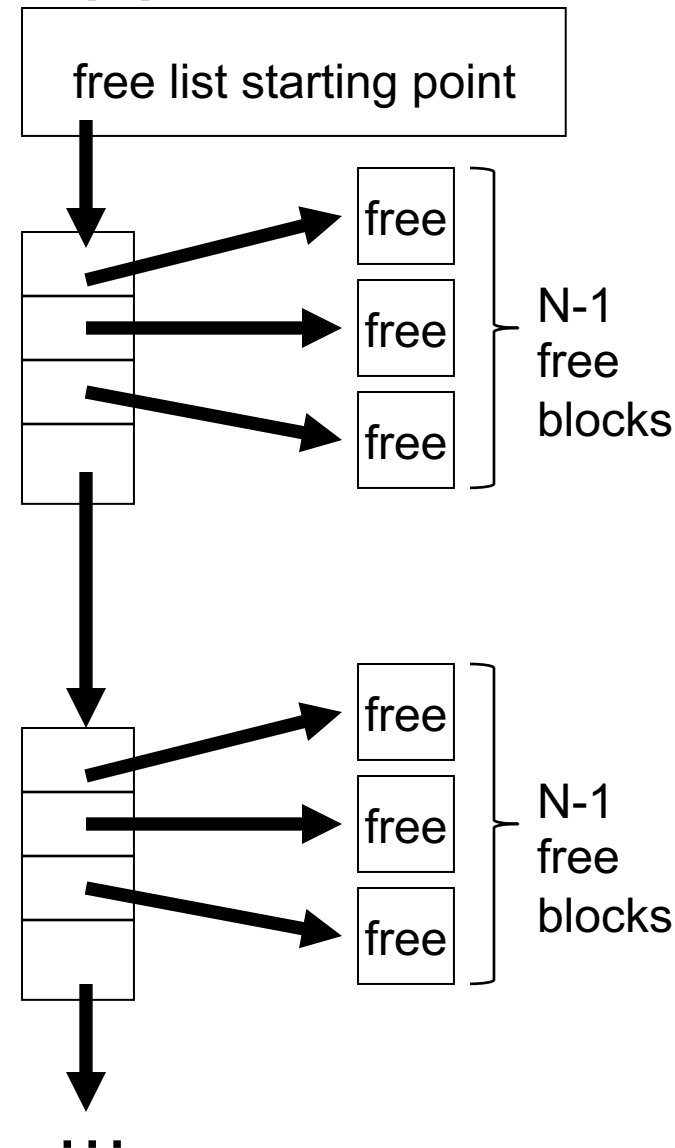
- Problem with Linked List free space management:
  - traversing the free list is slow if you want to allocate a large number of free blocks all at once
    - hopefully this occurs infrequently



# Free Space Management Approaches

## 3. Grouping

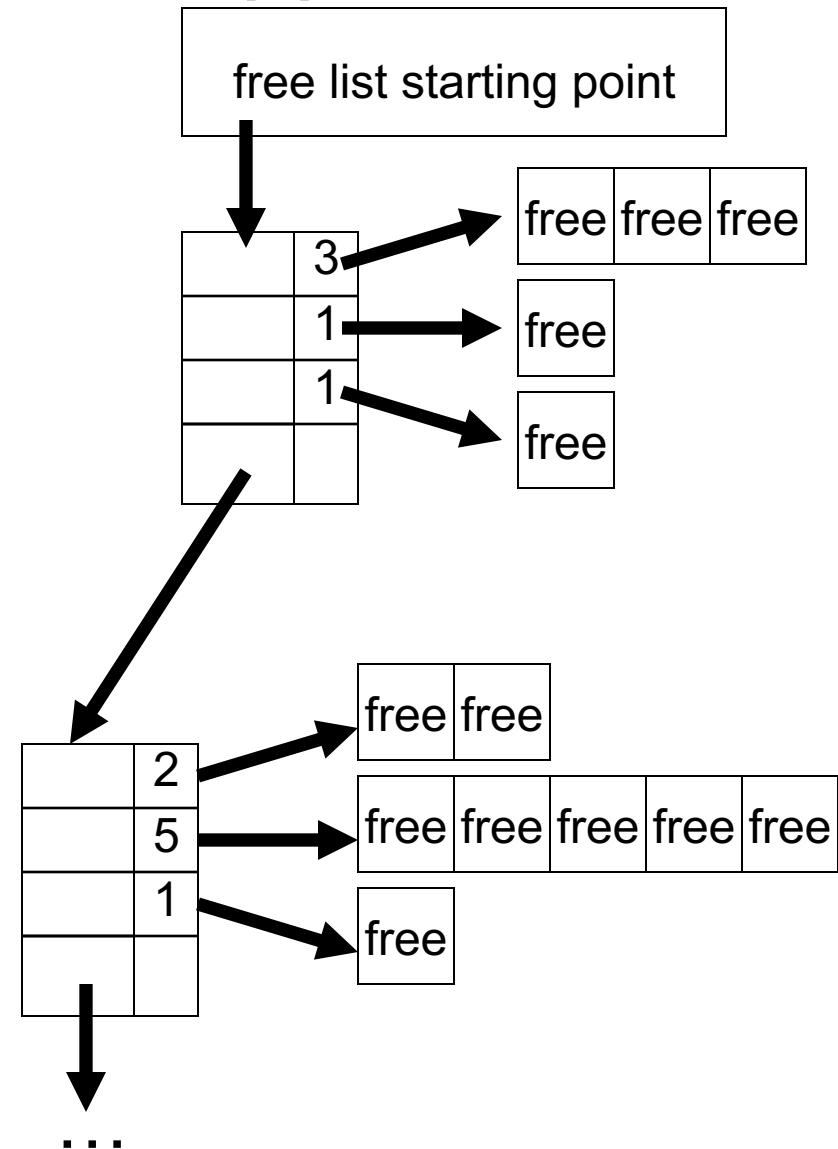
- linked list, except store  $n-1$  pointers to free blocks in each list block
- the last block points to the next list block containing more free pointers
- allows faster allocation of larger numbers of free blocks all at once



# Free Space Management Approaches

## 4. Counting -

- grouped linked list, but also add a field to each pointer entry that indicates the number of free blocks immediately after the block pointed to
- even faster allocation of large #'s of free blocks





# **File System Performance, Reliability, and Fault Recovery**



# File System Performance

- Approaches to improve performance in a file system:
  - In memory:
    - file header: caching FCB information about open files in memory improves performance (faster access)
  - directory:
    - caching directory entries in memory improves access speed.
    - And hash the directory tree to quickly find an entry and see if it's in memory.

# File System Performance

## – On disk:

- file data: indexed allocation is generally faster than traversing linked list allocation
- free block list: counting, grouped, linked list allows fast allocation of large # of files

# File System Performance

- Other potential optimizations:
  - the disk controller can also have its own cache that stores file data/FCBs/etc. for fast access
  - Cache file data in memory
  - Smarter layout on disk: keep an inode/FCB near file data to reduce disk seeks, and/or file data blocks near each other
  - *read ahead*:  
if the OS knows this is sequential access, then read the requested page and several subsequent pages into main memory cache in anticipation of future reads

# File System Performance

- Some other potential optimizations:
  - *asynchronous writes*: delay writing of file data until sometime later.
  - Advantages:
    - removes disk I/O wait time from the critical path of execution, e.g. a write(X) to a file can return quickly rather than waiting for completion of disk I/O, thereby allowing the program to move forward in its execution
    - This allows a disk to schedule writes efficiently, grouping nearby writes together
    - May avoid a disk write if the data has been changed again soon
    - note that in certain cases, you may prefer to enforce synchronous writes, e.g. when modifying file metadata in the FCB on an open() call

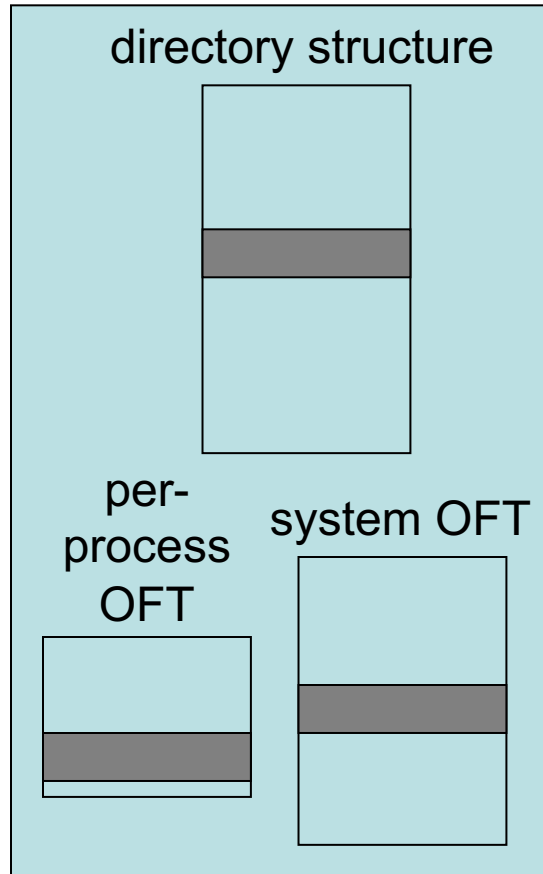
# File System Reliability and Fault Recovery

# File System Reliability/ Fault Recovery

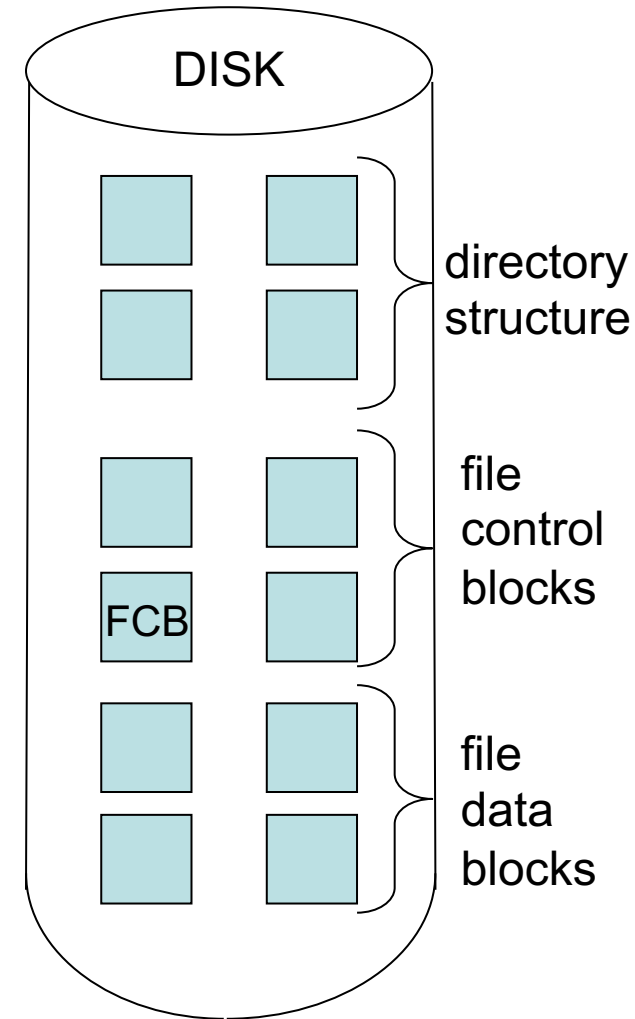
- **In general, OS should gracefully recover from hardware or software failure**
  - The file system needs to be engineered to ensure reliability/fault recovery
- **Problem: File system is quite fragile to system crashes**
  - There is a portion of the file system that is cached in memory
  - This portion may be inconsistent with the complete file system stored on disk

# File System Fragility

## In-Memory OS File Manager



- All in-memory file system data are lost on a **power loss**:
  - Directories, file metadata, and file data may all be cached in memory
  - They may all be modified
  - These modifications are lost if they weren't saved to disk



# File System Reliability/ Fault Recovery

- Problem #1: **asynchronous writes produce inconsistency between in-memory and on-disk file system**
- Example: promised writes of filed data that were delayed by asynchrony may be lost
- Example: asynchronous writes of directory metadata can create inconsistency between the file system on disk and the writes cached in RAM
  - Directory information in RAM can be more up to date than disk
  - if there is a system failure, e.g. power loss, then the cached writes may be lost
  - in this case, the promised writes will not be executed



# File System Reliability & Fault Recovery

- **Solution: To address asynchronous write inconsistency,**
  - UNIX caches directory entries for reads
  - But UNIX does not cache any data write that changes metadata or free space allocation

*These changes to critical metadata are written synchronously (immediately) to disk, before the data blocks are written*

- **Problem #2: Even if all writes are synchronous, there is still a consistency problem:**
  - any of the individual synchronous/asynchronous writes to disk can fail halfway through the operation, leaving a half-written directory entry, FCB, or file data block.

# File System Reliability & Fault Recovery

- **Problem #3: Complex operations can create inconsistency while waiting for them to complete**
  - e.g. a file create() involves many operations, and may be interrupted at any time in mid-execution
  - file create() updates the directory, FCB, file data blocks, and free space management
  - if there is a failure after creating the FCB, then the file system is in an inconsistent state because the file data has not yet been saved on disk,
    - i.e. the directory says there is a file and points to the FCB, but the FCB is incomplete because its index block hasn't been fully allocated

# Reliability/Fault Recovery Solutions

- **Approach:** file systems can run a consistency checker like fsck in UNIX or chkdsk in MSDOS
  - in linked allocation, would check each linked list and all FCB's to see if they are consistent with the directory structure.
    - similar checks for indexed allocation
  - Check each allocated file data block to see that its checksum is valid
- Disadvantages:
  - This is computational intensive and takes a long time to check the entire file system.
  - This can detect an error, but doesn't ensure recovery or correction

# Reliability/Fault Recovery Solutions

- **Approach:** *log-based recovery* is a solution that helps OS *recover* from file system failures:
  - OS maintains a log or journal on disk of each operation on the file system
  - called log-based or journaling file systems,
- The **log on disk is consulted after a failure** to reconstruct the file system
  - In a **journaling file system**, the **log is seen as a separate entity** from the file data.
  - In a **log-structured system**, the **log *is* the file system**, and there are no separate structures for storing file data and metadata – it's all in the log.

# Log-Based Recovery

- Each operation on the file system is written as a record to the log on disk *before* the operation is actually performed on data on disk
  - this is called *write-ahead logging*
- The file system has a sequence of records of operations in the log about what was intended in case of a crash.
  - The log contains a sequence of statements like **“I’m about to write this directory entry/file header/file data block”**,
  - and **“I just finished writing this directory/FH/data”**.

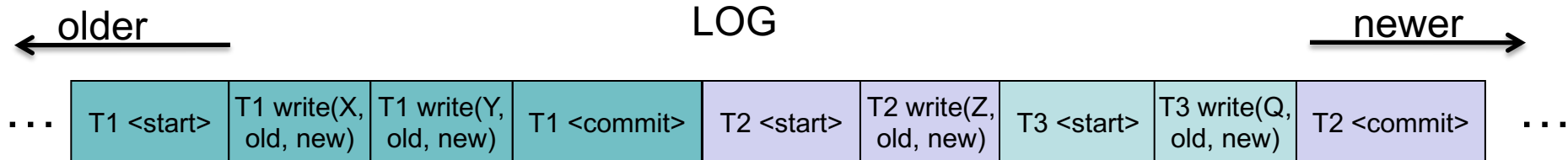
# Log-Based Recovery

- operations are grouped in sets called ***transactions***
  - e.g. a file create() has many steps.
    - You either want the entire file created, or not at all if it fails at any step along the way.
  - Need the set of steps into a single logical unit
    - It is performed in its entirety or not at all.
- Transactions are viewed as atomic
  - Either succeed in their entirety or not at all

# Log-Based Recovery

- A transaction  $T_i$  looks like the following:
  - begins with  $\langle T_i \text{ starts} \rangle$
  - followed by a sequence of records like  $\text{write}(X)$ ,  $\text{read}(Y)$ , ... needed to complete the transaction, e.g. a file  $\text{create}()$
  - ends with  $\langle T_i \text{ commits} \rangle$
- Write each of these operations to the log

# Log-Based Recovery



- Each log record of an operation within a transaction consists of:
  - transaction name  $T_i$
  - data item name, e.g. X
  - old value
  - new value
- Both the old and new values must be saved in order for the system to recover from crashes in mid-transaction

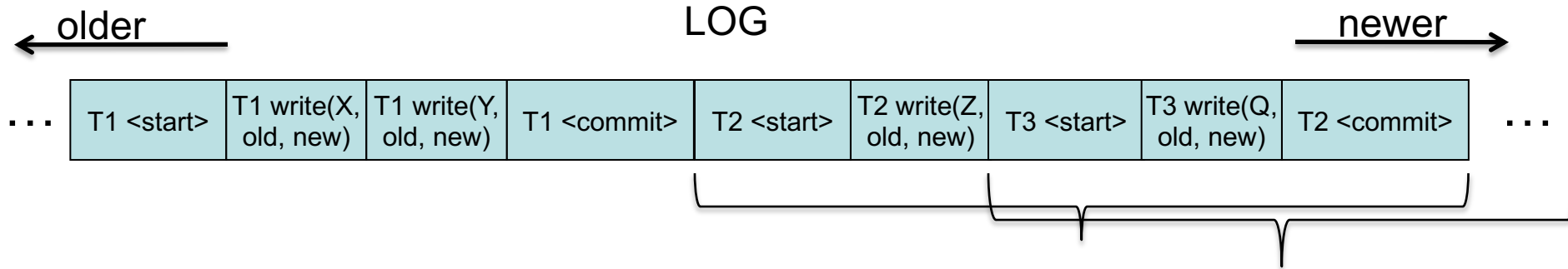


# Log-Based Recovery



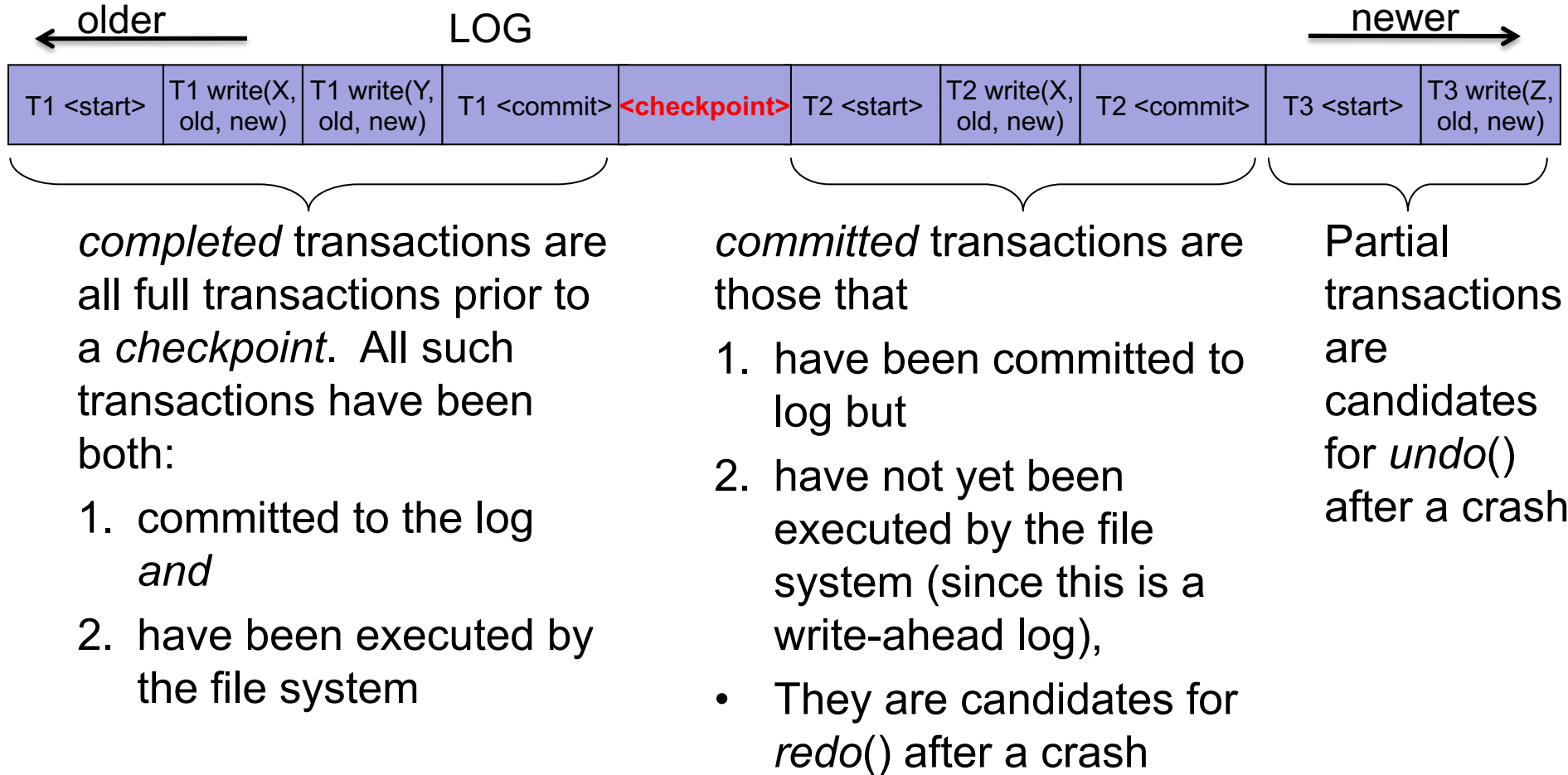
- **A transaction is not considered complete until it is committed in the log**
  - once the <commit> appears in the log, then even if the system crashes after this point, there is enough information in the log to fully execute the transaction upon recovery
  - therefore, once the <commit> appears in the log, it is OK to return from the system call that called file create() or file write()

# Log-Based Recovery



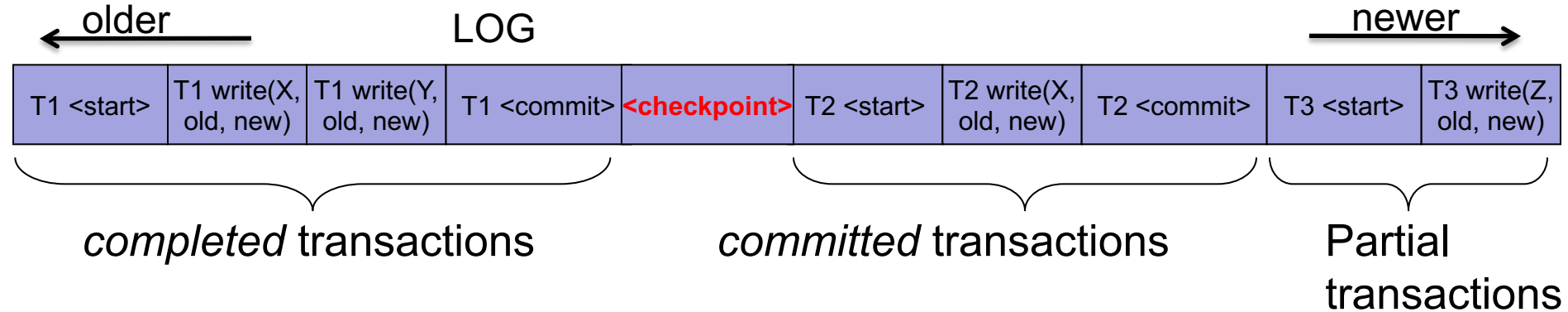
- Operations in different transactions can overlap in the log
- For asynchronous writes, the actual write(X) to disk may occur much later than the entry written to the log

# Log-Based Recovery



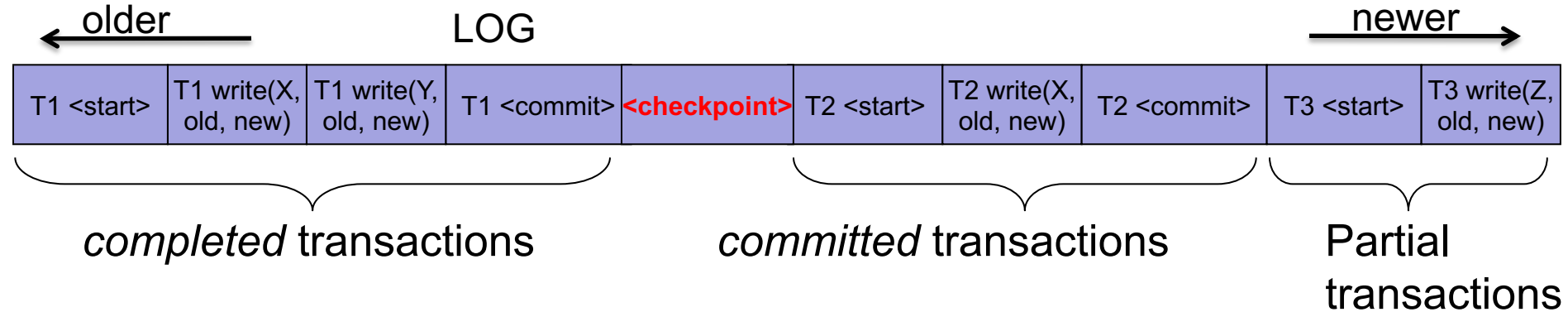
**Checkpoint:** confirm the state of the system: All trans are committed & executed

# Log-Based Recovery



- The checkpoint indicates that all full transactions (those with a <start> and <commit>) prior to the checkpoint (to the left) have been written to *both* the disk *and* log
- Committed transactions are full transactions in the log that are to the right of the most recent checkpoint, and thus have not been written to disk yet

# Log-Based Recovery



- In normal operation, the file system will
  - periodically *replay* committed transactions in the log onto disk,
  - Then add a new checkpoint to the log,
  - thus all committed transactions to the left of the newly added checkpoint are converted into completed transactions
  - completed transactions can be removed from the log, or just written over if it's a circular log

# Log-Based Recovery

- On a failure, the OS looks for the latest checkpoint in the log, and redo()'s committed transactions and undo()'s partial transactions from that point on
  - redo() transaction  $T_i$  if the log contains both  $\langle T_i \text{ starts} \rangle$  and  $\langle T_i \text{ commits} \rangle$  and these transactions appear after a checkpoint
  - undo() transaction  $T_k$  if the log contains  $\langle T_k \text{ starts} \rangle$  but not  $\langle T_k \text{ commits} \rangle$ 
    - this is called an aborted transaction
    - during recovery, such a transaction is rolled back to its former state

# Log-structured File Systems

- Assumption:
  - Write is more expensive than read since read can be served very quickly from cache
- Treat the storage as a circular log – Write sequentially to the head of the log
- Advantages:
  - Improve write performance through low seek time (batched write)
  - Allow time-travel or snapshotting
  - Recovery from crash is simpler based on the previous checkpoint.
- Disadvantage:
  - Might make read to be much slower since it fragments files (Optic and Magnetic disk)

# Journaling File Systems

- Some file systems like NTFS only write changes to the metadata of a filesystem to the log
  - file headers and directory entries only
  - NOT any changes to file data
  - The journal is separate from the main file system
- Copy-on-write file systems (such as ZFS and Btrfs)
  - Avoid in-place changes to file data by writing out the data in newly allocated blocks
  - Followed by updated metadata that would point to the new data and disown the old
  - Followed by metadata pointing to updated metadata repeatedly to the root of the file system hierarchy
  - Has the same correctness-preserving properties as a journal, without the write-twice overhead
  - Add metadata (checksums) to insure data integrity



# Journaling File Systems

- Linux's ext4fs can be parameterized to operate in 3 modes (cont):
  1. *Journal mode*: both metadata and file data are logged. This is the safest mode, but there is the latency cost of two disk writes for every write.
  2. *Ordered mode*: only metadata is logged, not file data, and it's guaranteed that file contents are written to disk before associated metadata is marked as committed in the journal.
    - This is the default on many Linux distributions.
  3. *Write-back mode*: only metadata is logged, not file data, and no guarantee file data written before metadata, so files can become corrupted.
    - This is riskiest mode/least reliable but fastest.