Week 8 – Recap

Pamela Delgado

April 10, 2019

File System Interface

File

- Un-interpreted un-typed sequence of bytes
- Identified by a globally unique *uid*

Open File

- File instance accessed by a process
- Identified by a per-process unique *tid* or *fd*

Directory

- Set of mappings (string $\rightarrow$ uid)

File System Primitives

- Access: Create(), Delete(), Read(), Write()
- Random vs. Sequential and Seek()
- Concurrency: Open(), Close()
- Naming:
  - Insert(), Lookup(), Remove()
  - CreateDirectory(), DeleteDirectory(), List()

Hierarchical Directory Structures

- Tree
- (Acyclic) Graph
  - Allows sharing of two *uids* under different names
- Two additional primitives
  - HardLink() and SoftLink()

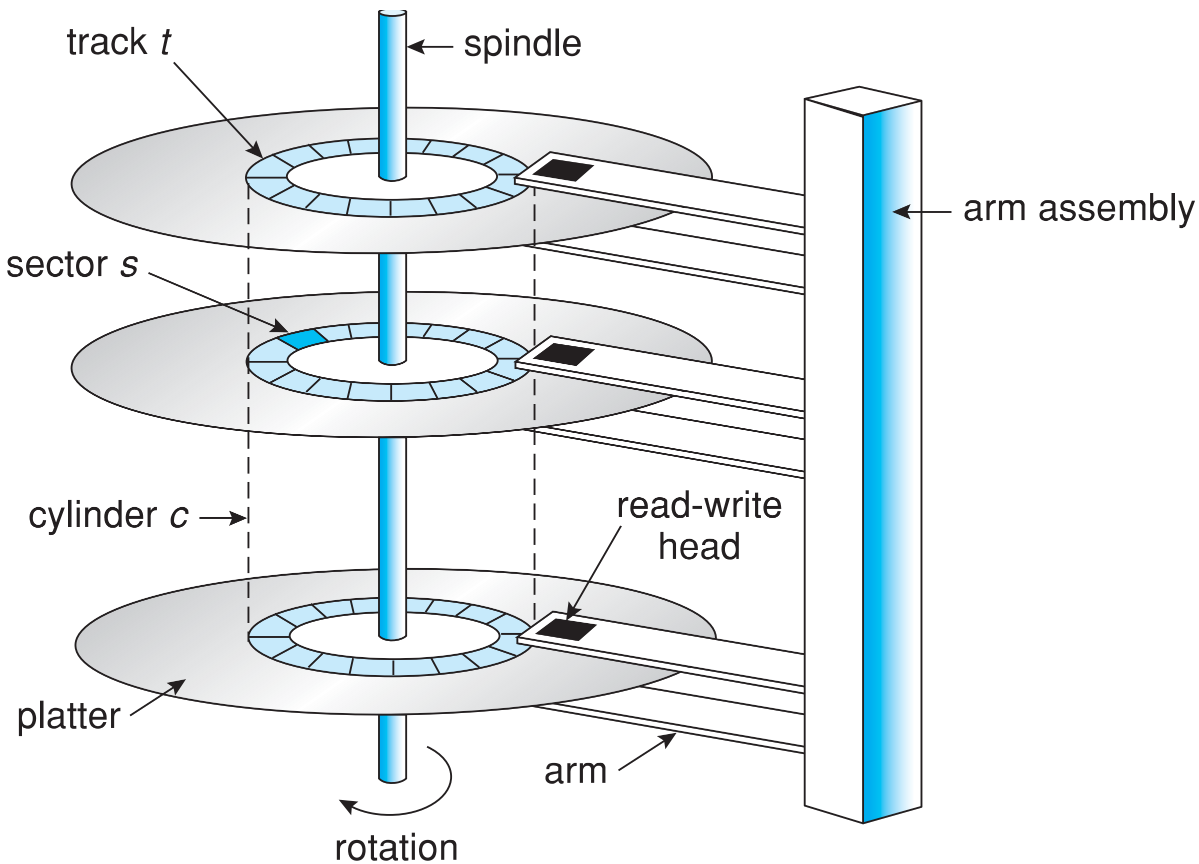
Linux: Collapsed Interface for Storage and Naming

- Creat(string)*
- fd = Open(string, [optional_args])
- Read(fd, buffer, bytes)
- …
- Close(fd)

* Open can also create a file, Creat use not so favored

Disks and Disk Optimization

Disk


track t
spindle
arm assembly
sector s
cylinder c
read-write
head
platter
arm
rotation

Disk API

- ReadSector(logical_sec_no, buffer)
- WriteSector(logical_sec_no, buffer)

Disk Access Times

- Disk access >> memory access
- Seek > Rotational Latency > Transfer

Disk Access Optimizations

Optimization	Goal
Caching	Avoid disk access
Read-ahead	Avoid waiting for disk
Disk allocation	Avoid seek and rotational latency
Disk scheduling	Avoid seek

Another Way to Think about Disk Access

- Disks are random access devices
- But, sequential access >> random access
  - Bandwidth 100x greater for sequential
- Applications should aim for sequential access
- Systems should aim for sequential access

Week 9 –
Basic File System Implementation

Pamela Delgado

May 1, 2018

based on:
- W. Zwaenepoel slides
- Arpaci-Dusseau book
- Silbershatz book

Aside: learning about systems

- Learn about principles/concepts
- Tradeoffs
- Develop a mental model

- Not (so much) about an specific way of implementing some particular OS/System

Two Main Functionalities

- Naming/Directory
  - Use file to store directory
- Storage
  - On-disk data structures
  - In-memory data structures

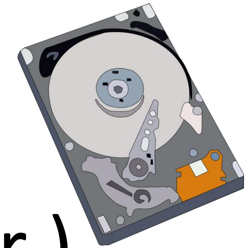
- Focus on storage

File System Implementation

- The main task of the file system is to translate

- From user interface methods
- Read(uid, buffer, bytes)

- To disk interface methods
- ReadSector(logical_sector_number, buffer)



Two Small Simplifications - 1

- User Read() allows arbitrary number of bytes
- Simplify to only allowing Read() of a block
  - Read(uid, block_number)
- A block is fixed-size

Two Small Simplifications - 2

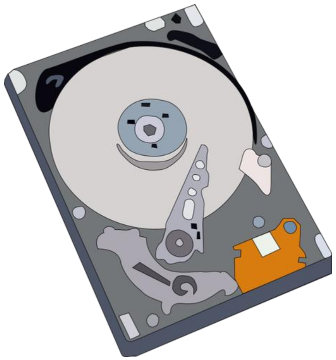
- Typically
  - Block size = 2^n * sector size
- For instance
  - Block size = 4,096 bytes
  - Sector size = 512 bytes
- For simplicity of presentation in class
  - Block size = sector size

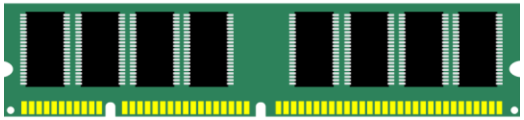
Terminology/Presentation Note

- When talking about disks
- The word “pointer” is often used
- It means
  - A *disk* address, i.e., a logical_sector_number
  - Not a memory address
- Pictures often contain “arrays”
- They mean here regions of sectors on disk

Disk vs. In-Memory

- Simple but golden rule
- If it is not on disk and you crash, it is gone!
- If you need it after a crash, it must be on disk





Disk Data Structures

- Boot block
- Device directory
- User data
- Free space

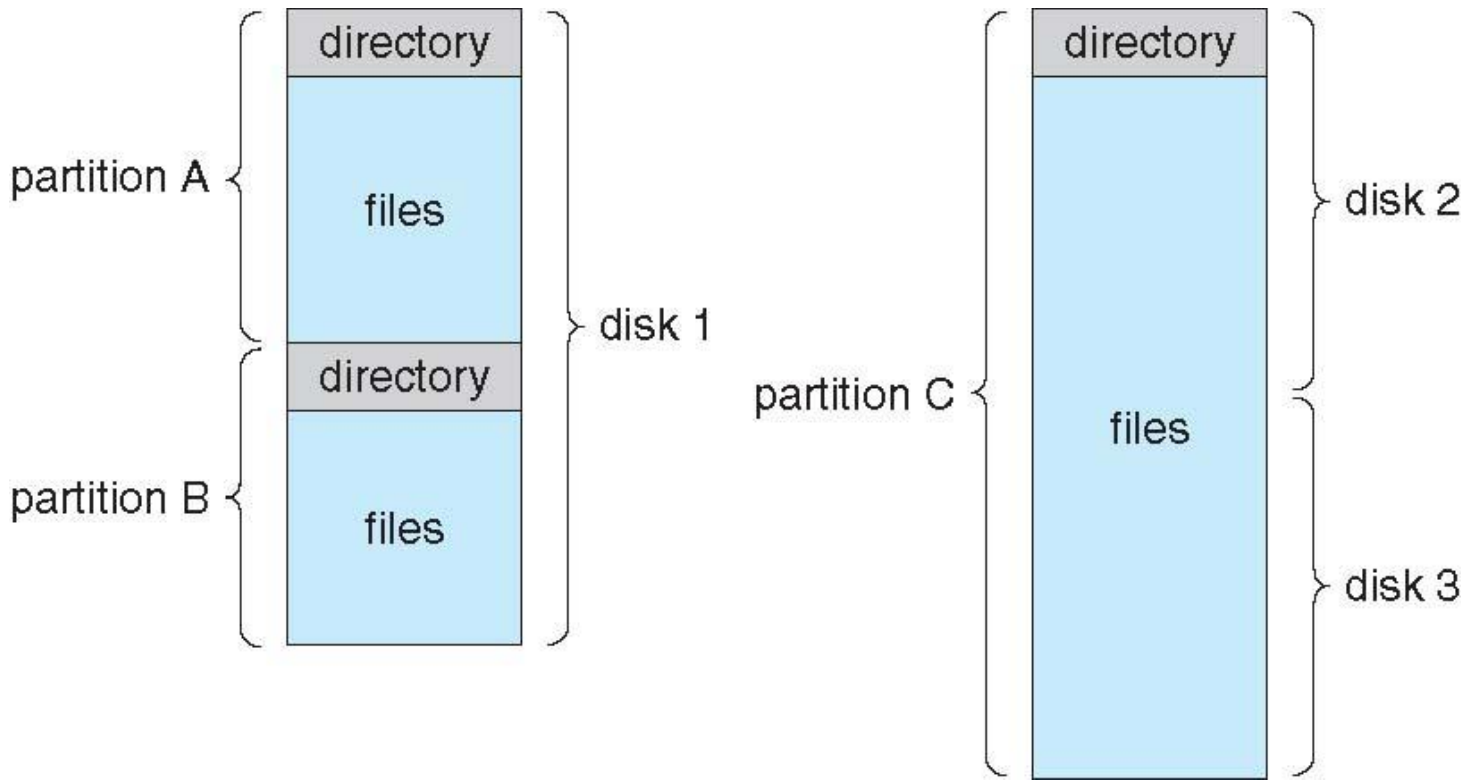
Boot Block

- At fixed location on disk (usually sector 0)
  - Own format
  - Boot operating system
  - Otherwise, empty
- Contains boot loader
  - Information about file system
  - To load and execute kernel
- Read on machine boot

Disk Data Structures

- Boot block
- Device directory
- User data
- Free space

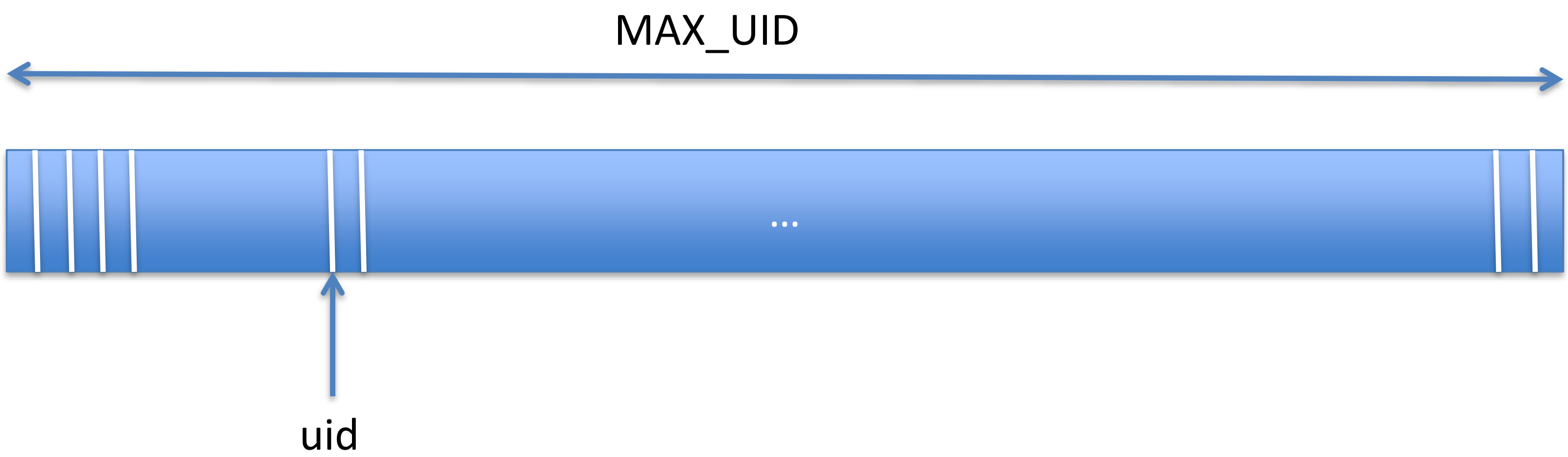
Device Directory



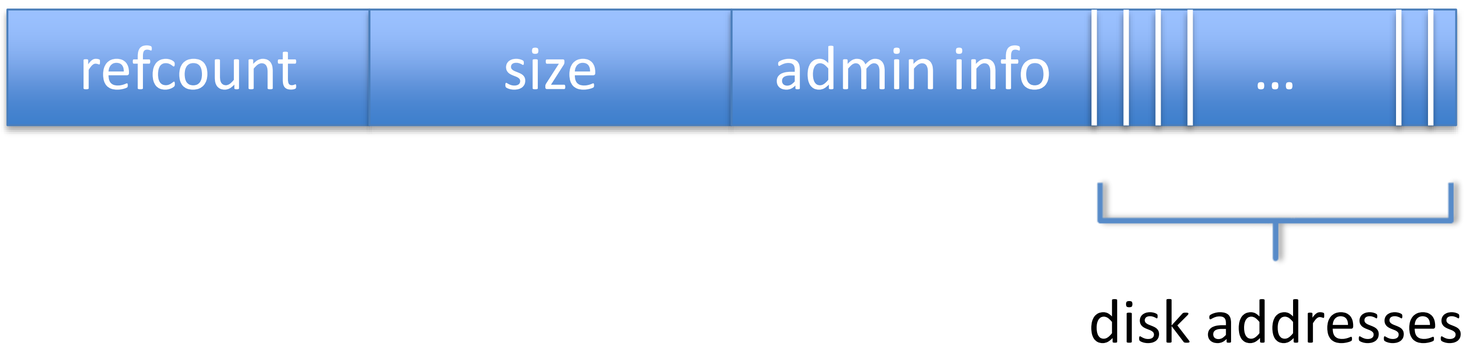
Device Directory

- Fixed, reserved area on disk
- Array of records (device directory entry or DDE)
- Indexed by *uid*
- Record contains
  - In-use bit (more generally, reference count)
  - Size
  - Other info: access rights, etc.
  - Disk address(es) pointing to file data

Device Directory



Device Directory Entry



Disk Data Structures

- Boot block
- Device directory
- User data
- Free space

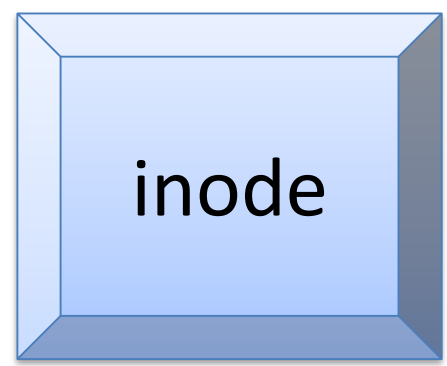
Disk building blocks

- Divide disk into blocks
- Asume one size for now
- Fill in those blocks → with user data ! mostly

  → the rest?

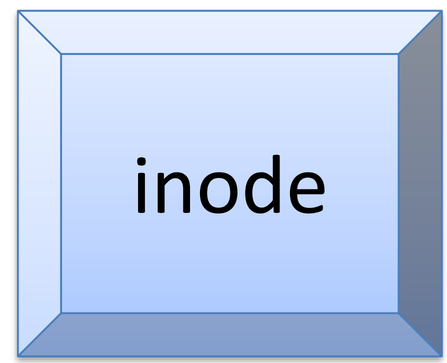
Disk building blocks

- Divide disk into blocks
- Asume one size for now
- Fill in those blocks → with user data ! Mostly
- The rest? → information for file system
  - Which data blocks make a file
  - Size
  - Owner, access rights
  - Access/modify time



Disk building blocks

- Divide disk into blocks
- Fill in those blocks → with user data ! Mostly
- The rest? → information for file system
  - Which data blocks make a file
  - Size
  - Owner, access rights
  - Access/modify time
- Something missing?



128/256 bytes

inode

inode (short for index node) = structure that holds metadata of a given file such as its length, permissions and the location of its constituent blocks

source: R. and A. Arpacci-Dusseau, Operating systems Three easy pieces

User Data Allocation

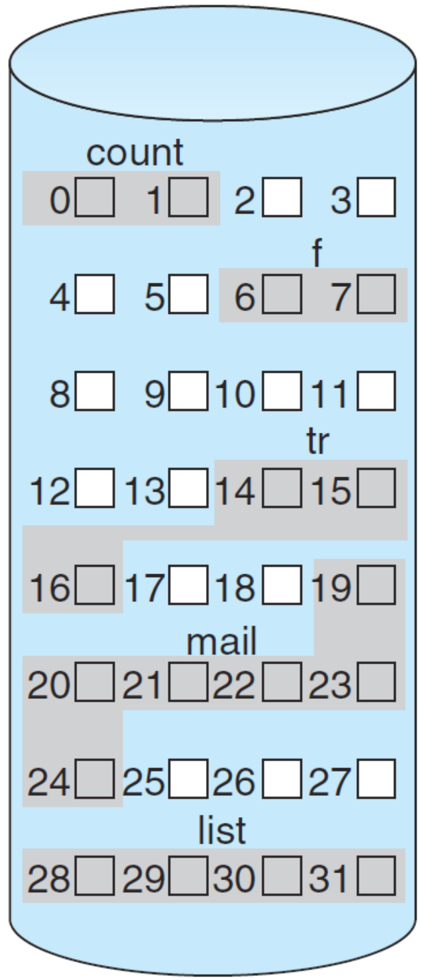
- Contiguous
- Linked list
- Indexed
- Indexed with indirect blocks
- Extent-based

Goals:

1. Use disk space effectively!
2. Quick file access

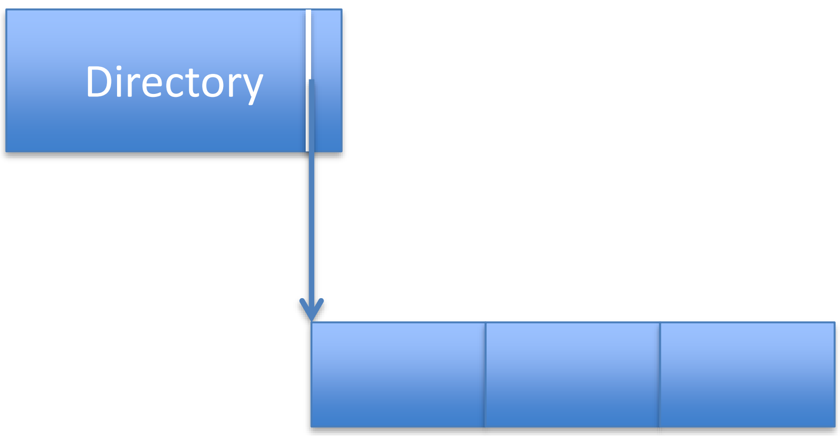
Contiguous Allocation

- Disk data blocks contiguous on disk



Contiguous Allocation

✓ Need only 1 pointer in device directory entry

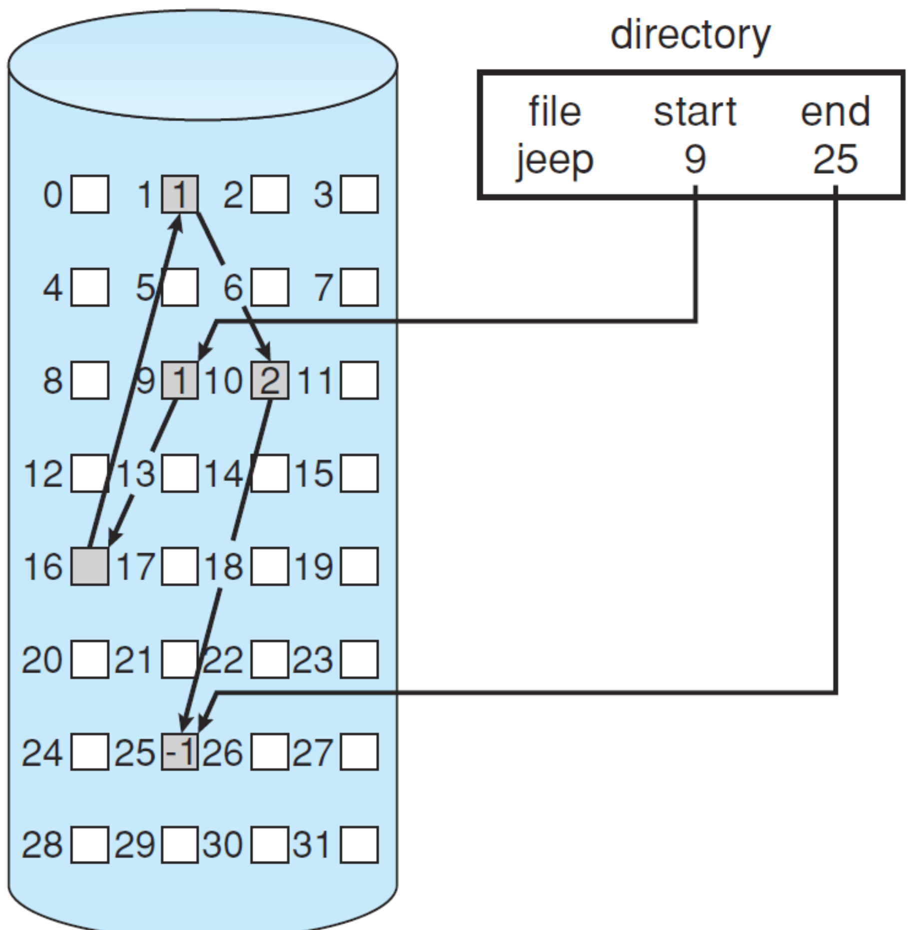


Contiguous Allocation

- ❌ Creates disk fragmentation
  - Many un-usable holes
- Impractical

Linked List Allocation

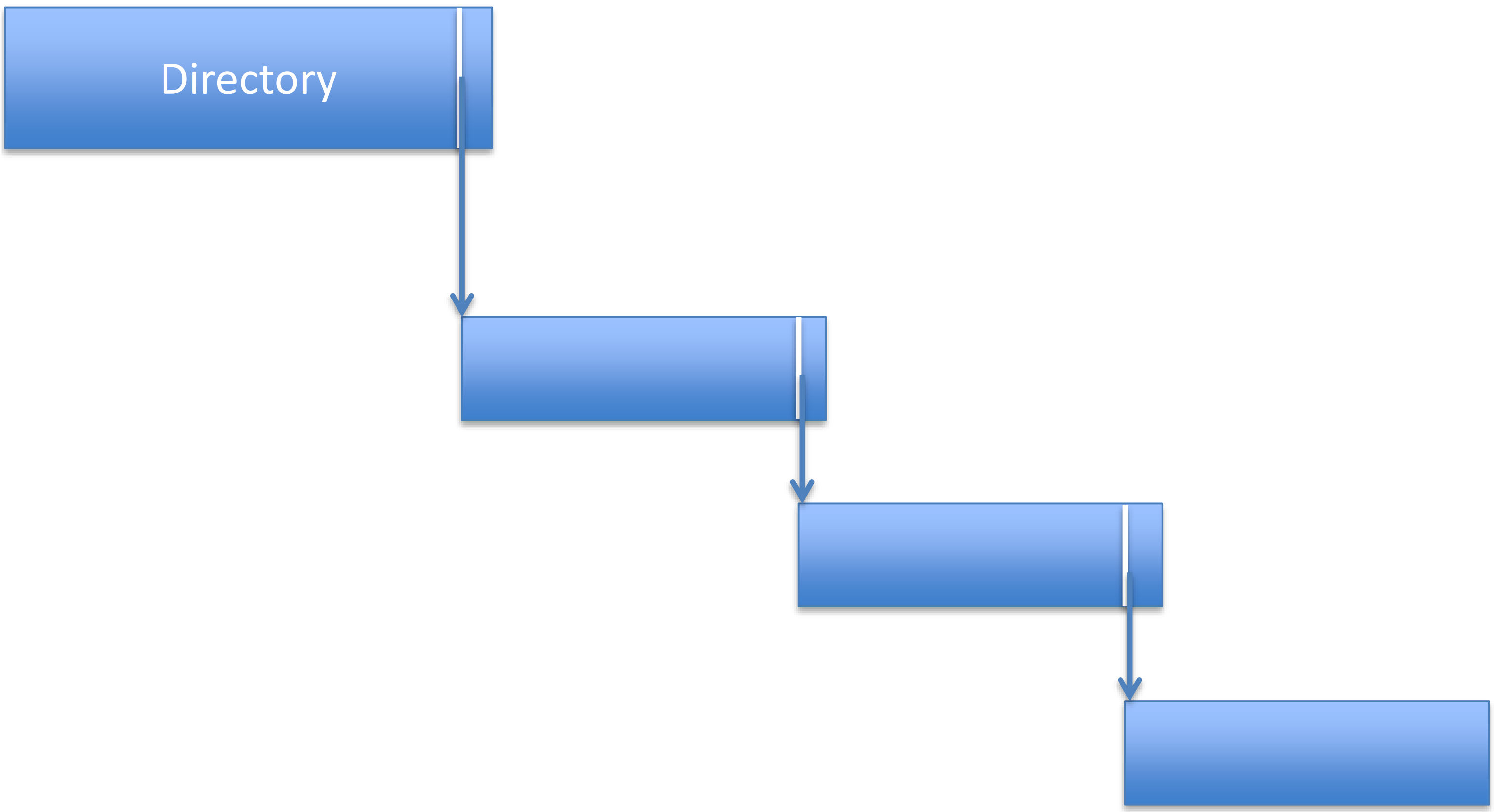
- Each data block contains pointer to next



Linked List Allocation

✓ Only 1 pointer in device directory entry (head)

- 2 if you want to store the tail



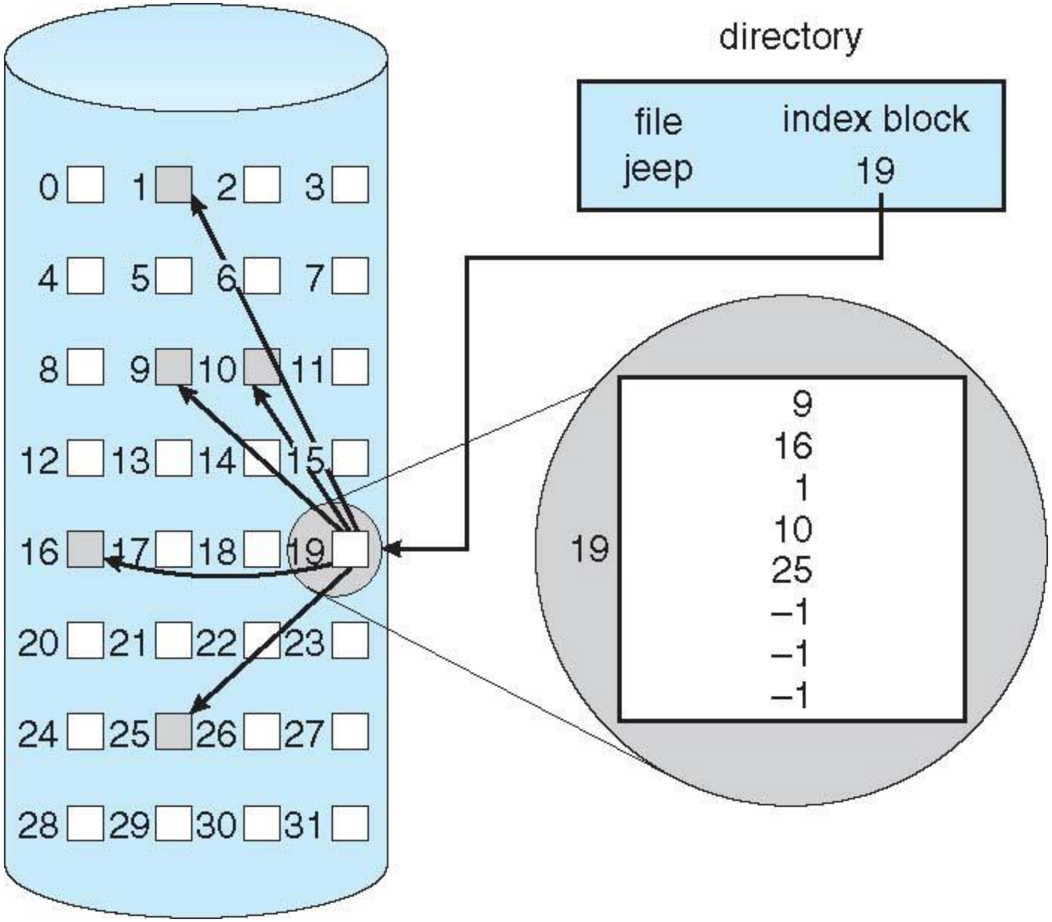
Linked List Allocation

- ❌ Inefficient access (esp. random access)
- ❌ Pointer space in data block

How would
you make
this better?

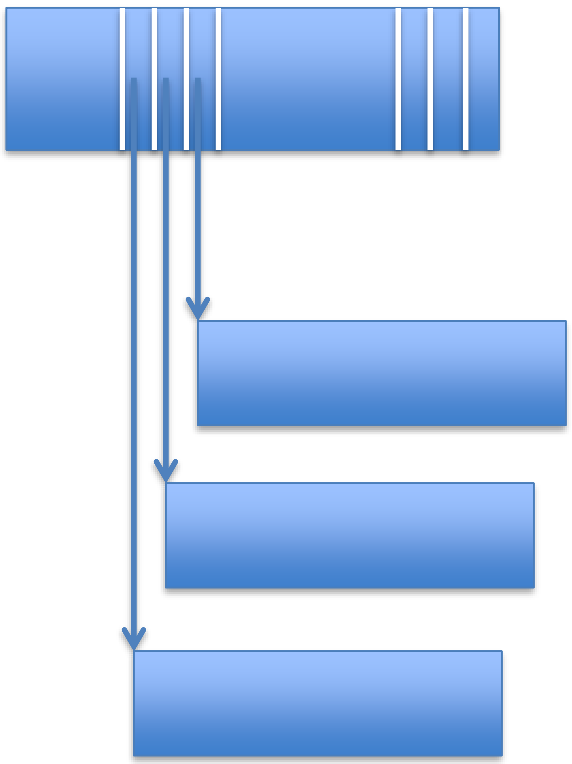
Indexed Allocation

- Pointers to blocks in an *index block* (in order)



Indexed Allocation

- N pointers in device directory entry
- Point to index block



Indexed Allocation

✓ Efficient direct access

✓ No external fragmentation

- What if file only needs two blocks?
- What if file size > N * size of data block?

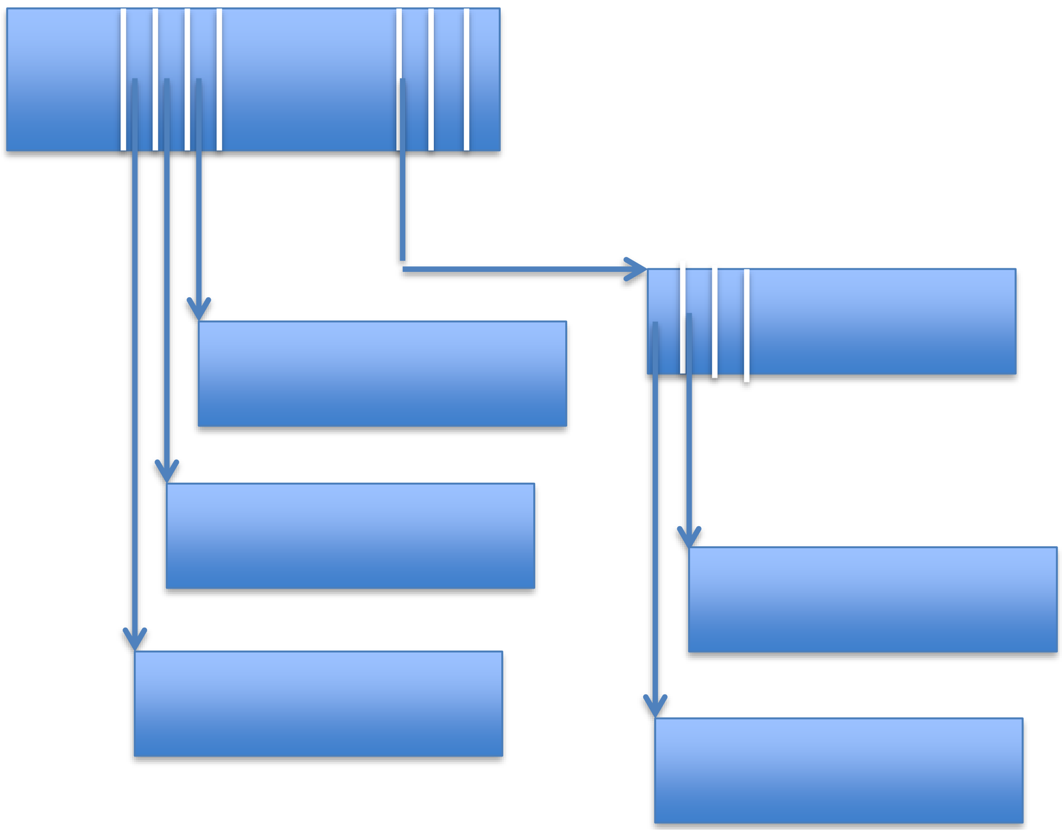
✗ No space-efficient

Indexed Allocation with Indirect Blocks

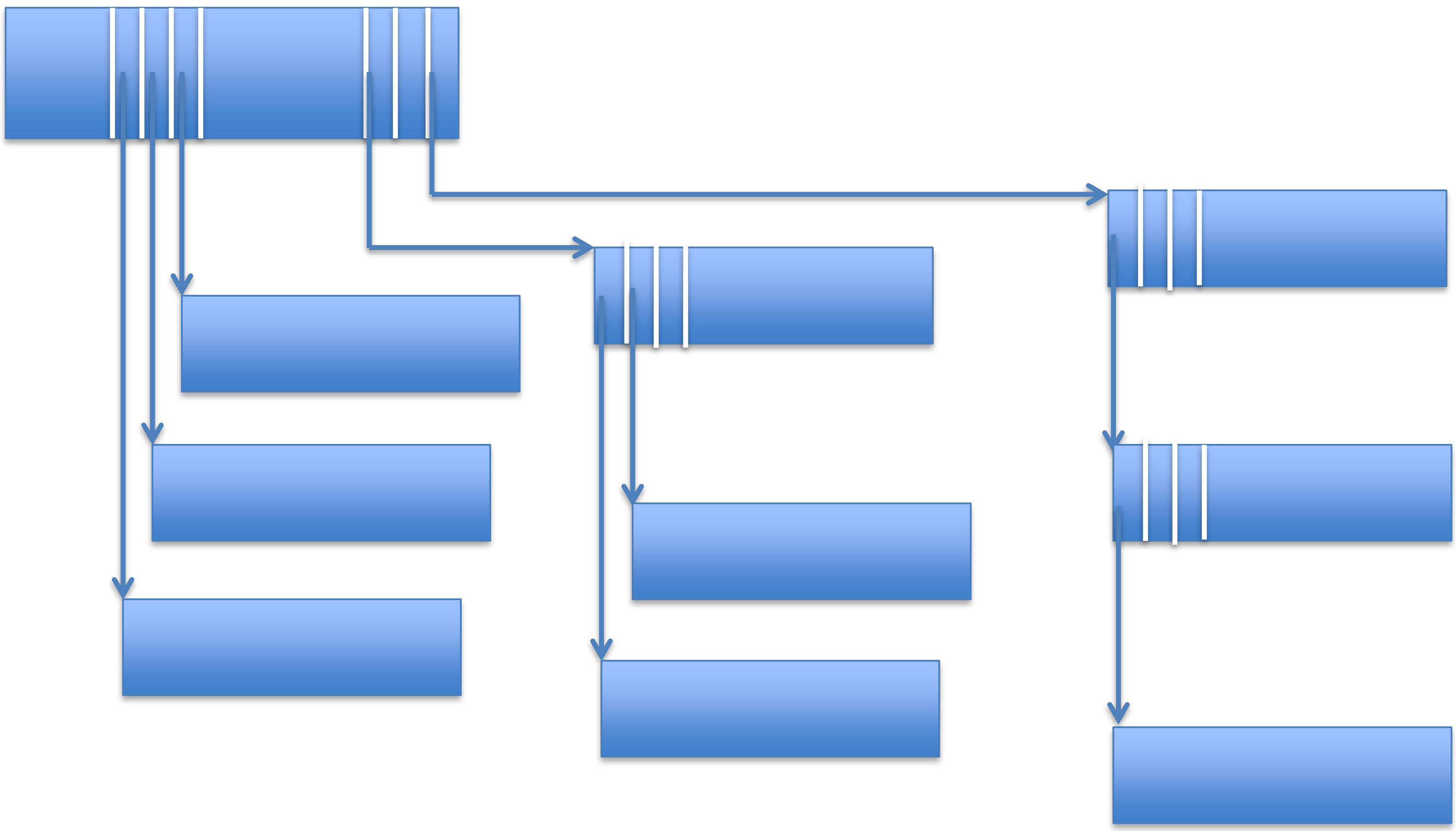
- N pointers in device directory entry
- First M (< N) point to first M data blocks
- Blocks M+1 to N point to *indirect blocks*

- Indirect blocks
  - Do not contain data
  - But pointers to subsequent data blocks
- Double-indirect blocks also possible

Indexed Allocation with Indirect Blocks



Indexed Allocation with Indirect and Double-Indirect Blocks



Indexed Allocation with Indirect Blocks

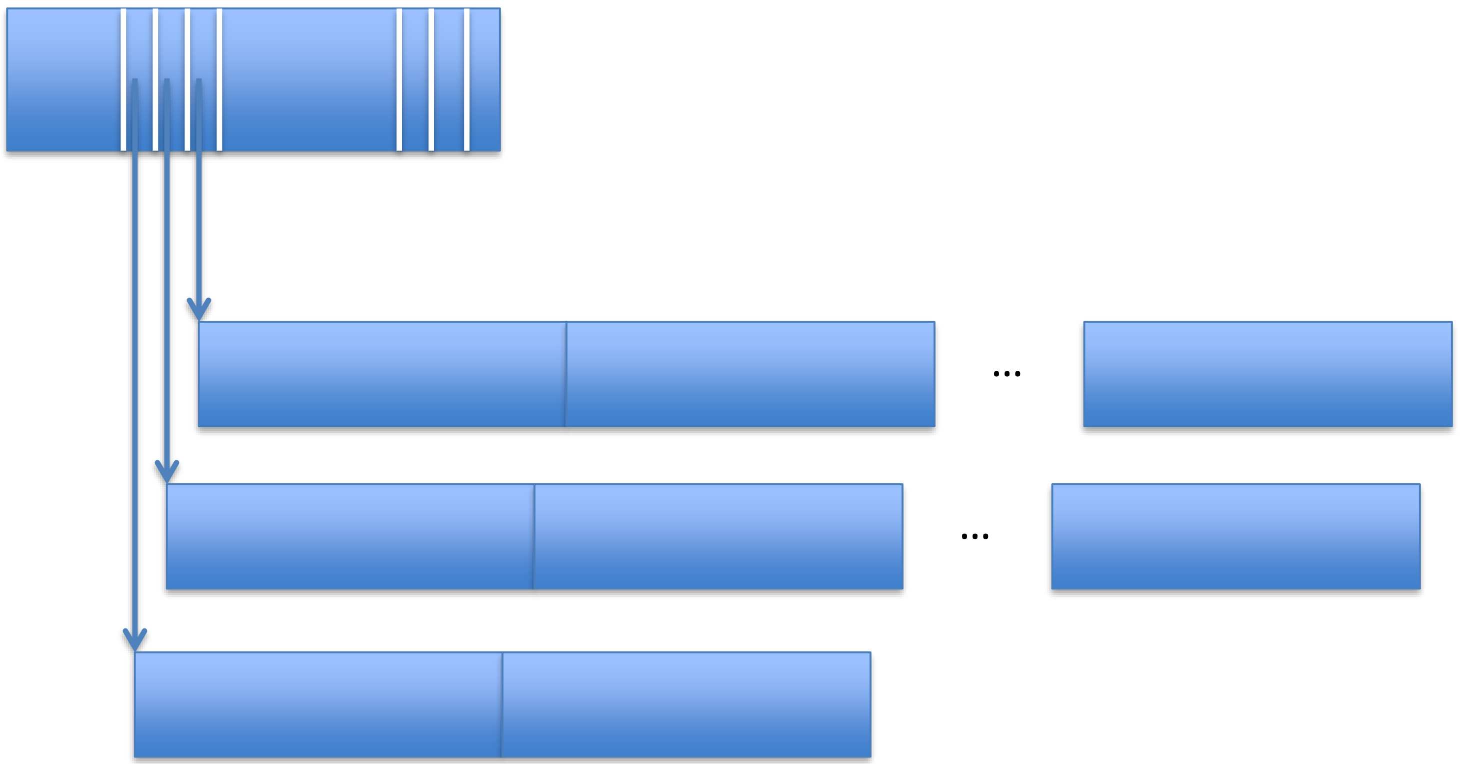
- ✓ Efficient access for small files
- ✓ Possible to extend to very large files

Pre allocation?

Extent-Based Allocation

- Device directory entry
  - Contains disk address and length of extent
  - Instead of just disk address
  - In other words, point to a sequence of disk blocks

Extent-Based Allocation



Extent-Based Allocation

✓ Good sequential and random access
✓ Can be combined with indirect blocks

- Common practice in Linux

Free Space

- Linked list
- Bitmap
  - Array[#numsectors]
  - Free / In-use

In-Memory Data Structures

- Cache
- Cache directory
- Queue of pending disk requests
- Queue of pending user requests
- Active file table
- Open file tables

Cache

- Fixed contiguous area of kernel memory
- Size = max number of cache blocks x block size
- A large chunk of memory of the machine

Cache Directory

- Usually a hash table
- index = hash(disk address)
- With an overflow list in case of collision
- Usually has a “dirty” bit

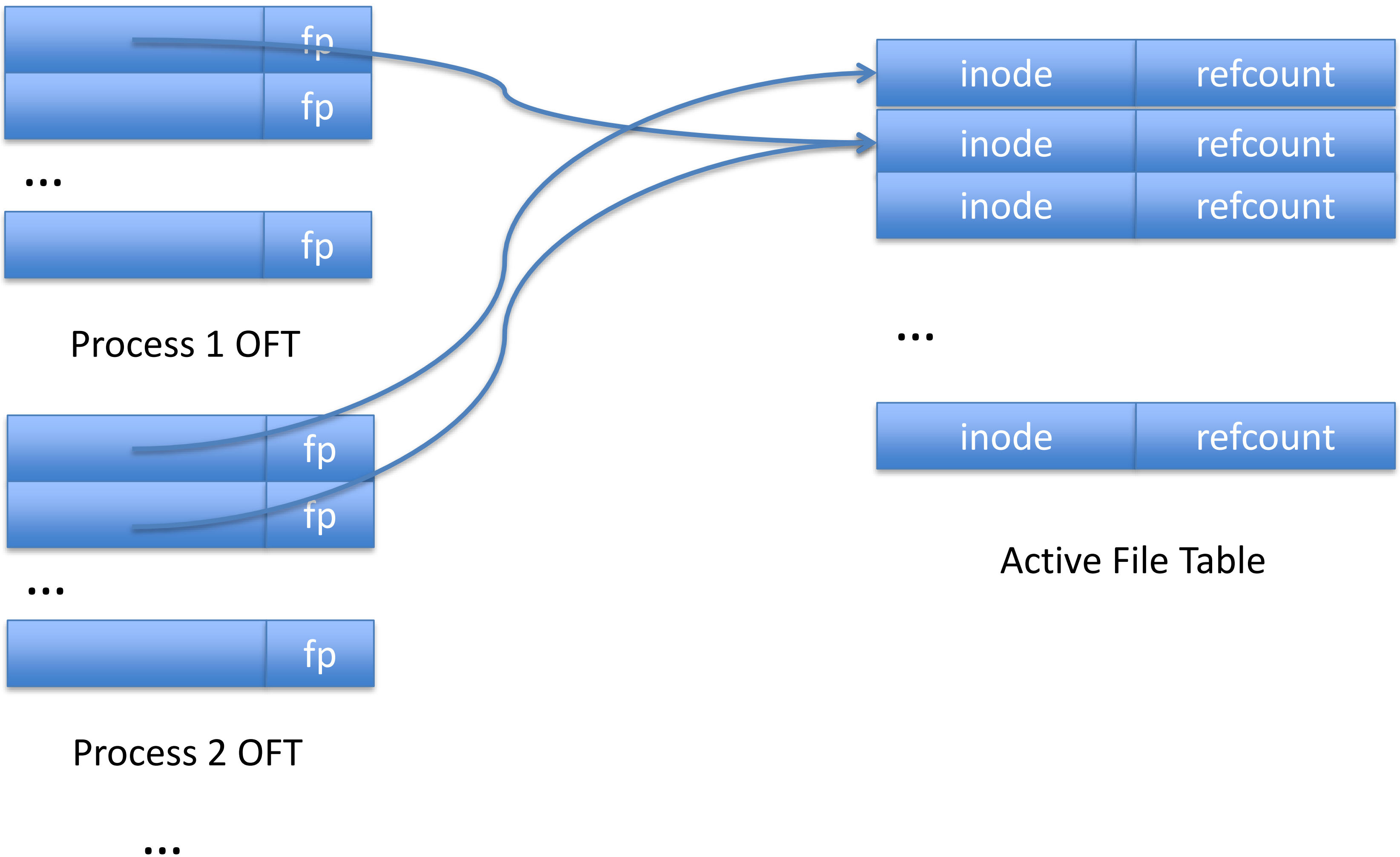
(System-Wide) Active File Table

- One array for the entire system
- One entry per *open file*
- Each entry contains
  - Device directory entry of file
  - Additional info
    - Refcount of number of file opens

(Per-Process) Open File Tables

- One array per process
- One entry per *file open* of that process
- Indexed by file descriptor *fd*
- Each entry contains
  - Pointer to entry in active file table
  - File pointer *fp*
  - Additional info

Picture of Open File Tables



Putting it All Together

- File systems main methods
  - Create(), Delete()
  - Open(), Close()
  - Read(), Write(), Lseek()
  - Cache flush and replacement
- With some major simplifications
  - No access permission checks
  - No return value checks
  - Etc.

uid = Create()

- Find a free uid (refcount = 0)
- Set refcount to 1
- Fill in additional info
- Write back to cache (and to disk)

- Device directory is cached in memory
- Usually easy to find free uid

Delete(uid)

- Find inode
- Decrement refcount
- If refcount == 0
  - Free all data blocks and indirect blocks
  - Set entries in free space bitmap to 0
- Write back to cache (and to disk)

Note

- In general, write-behind is used
- For user data ok
- For metadata
  - Written to disk more aggressively
  - Affects integrity of file system

tid = Open(uid)

- Check in Active File Table if uid already open
- If so,
  - Refcount in Active File Table ++
  - Allocate entry in Open File Table
  - Point to entry in Active File Table
  - Set fp = 0

tid = Open(uid)

- Check in Active File Table if uid already open
- If not,
  - Find free entry in Active File Table
  - Read inode and copy in Active File Table entry
  - Refcount = 1
  - Allocate entry in Open File Table
  - Point to entry in Active File Table
  - Set fp = 0

Close(tid)

- Find entry in Active File Table
- Refcount --
- If refcount == 0 remove entry Active File Table
- Remove entry from Open File Table

Read()

- Find fp in Open File Table and increment
- Compute block number to be read
- Find disk address in inode in Active File Table
- Look up in Cache Directory (disk address)
- If present, return
- If not, find free entry in cache
- ReadSector(disk address, free cache block)

Write()

- Find fp in Open File Table and increment
- Compute block number to be written
- Find/allocate disk address in Active File Table
- Look up in cache directory (disk address)
- If present, overwrite and return
- If not, find free cache entry and overwrite

Lseek(tid, new_fp)

- In Open File Table set fp = new_fp

Cache Replacement

- Keep LRU list
  - Unlike memory management, here easy to do
  - Accesses are far fewer (file vs memory access)
- If no more free entries in the cache
  - Replace “clean” block according to LRU
  - Replace “dirty” block according to LRU

Cache Flush

- Find “dirty” entries in cache
- WriteSector(…)

- Periodically (30 seconds)
- When disk is idle

What About Directories?

- Directories stored as files

Typical Operation

- fd = Open(string)
- Read(fd, …)

Typical Operation

- fd = Open(string)
  - Directory lookup (disk reads)
  - Inode lookups (disk reads)
- Read(fd, …)
  - Data (disk reads)

Disk Behavior

- Head moves between
  - Directories
  - Inodes
  - Data

Advanced Disk Layout

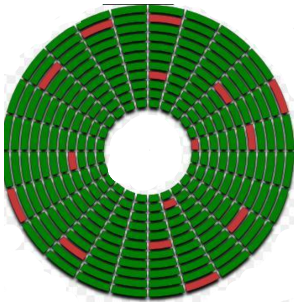
- Co-locate related
  - Directories
  - Inodes
  - Data
- In same “cylinder group”
  - Set of cylinders next to each other

Some Loose Ends

- File system checking
- Sector replication
- Defragmentation
- Memory-mapped files (mmap)

File System Startup

- Normally, nothing would be necessary
- Sometimes things are not normal
  - Disk sector goes bad
  - File system software has bugs
  - …
- Common to “check” the file system (fsck)



File System Check

- No sectors are allocated twice
- No sectors are allocated and on free list
- Reconstruct free list

Replication

- Some key sectors are replicated
  - Boot blocks
  - Sometimes also device directory

Disk Fragmentation

- Free space consists
  - Small “holes” (of 1 or 2 sectors)
  - Spread all over the disk
- Happens even if good disk allocation
- No longer possible to do good disk allocation

Disk Defragmentation Utility

- Takes the file system offline
- Moves files into contiguous locations

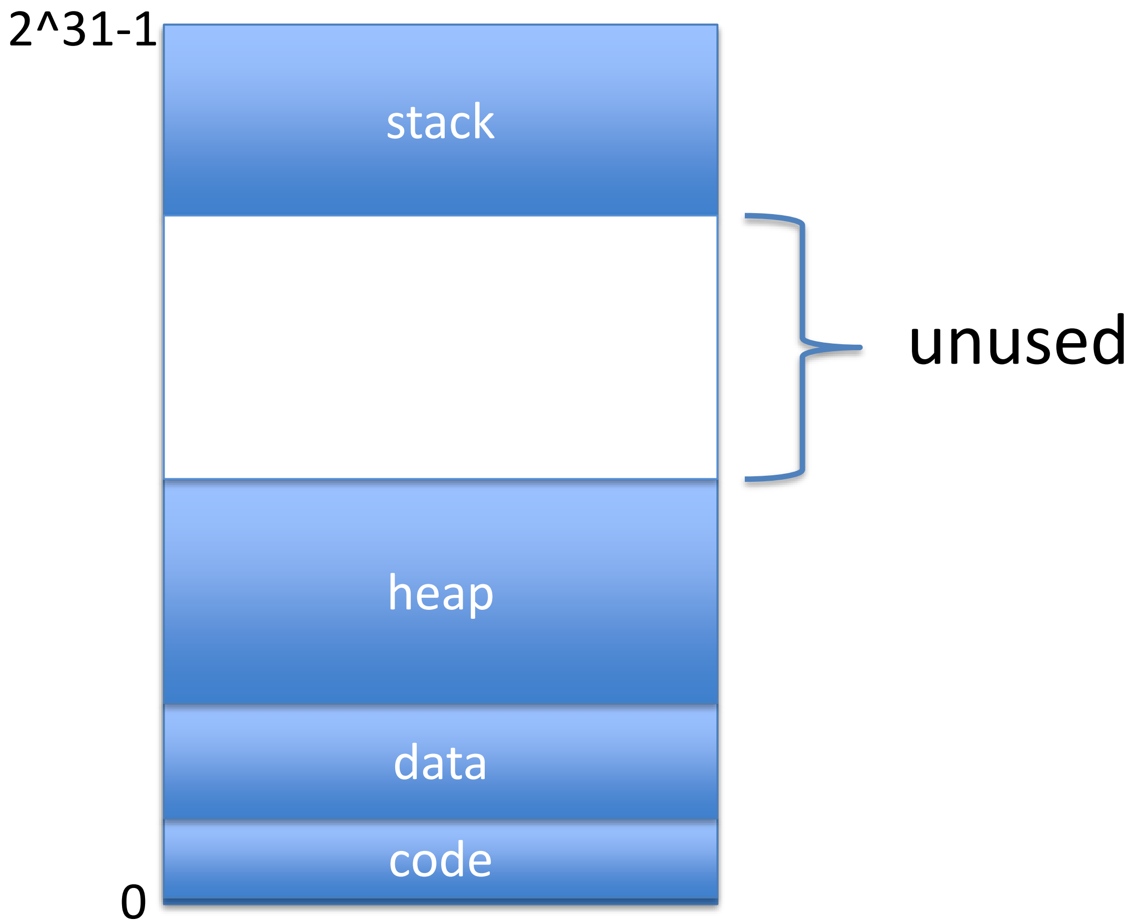
- Better performance
- More room for good disk allocation

- Can be done online, but tricky

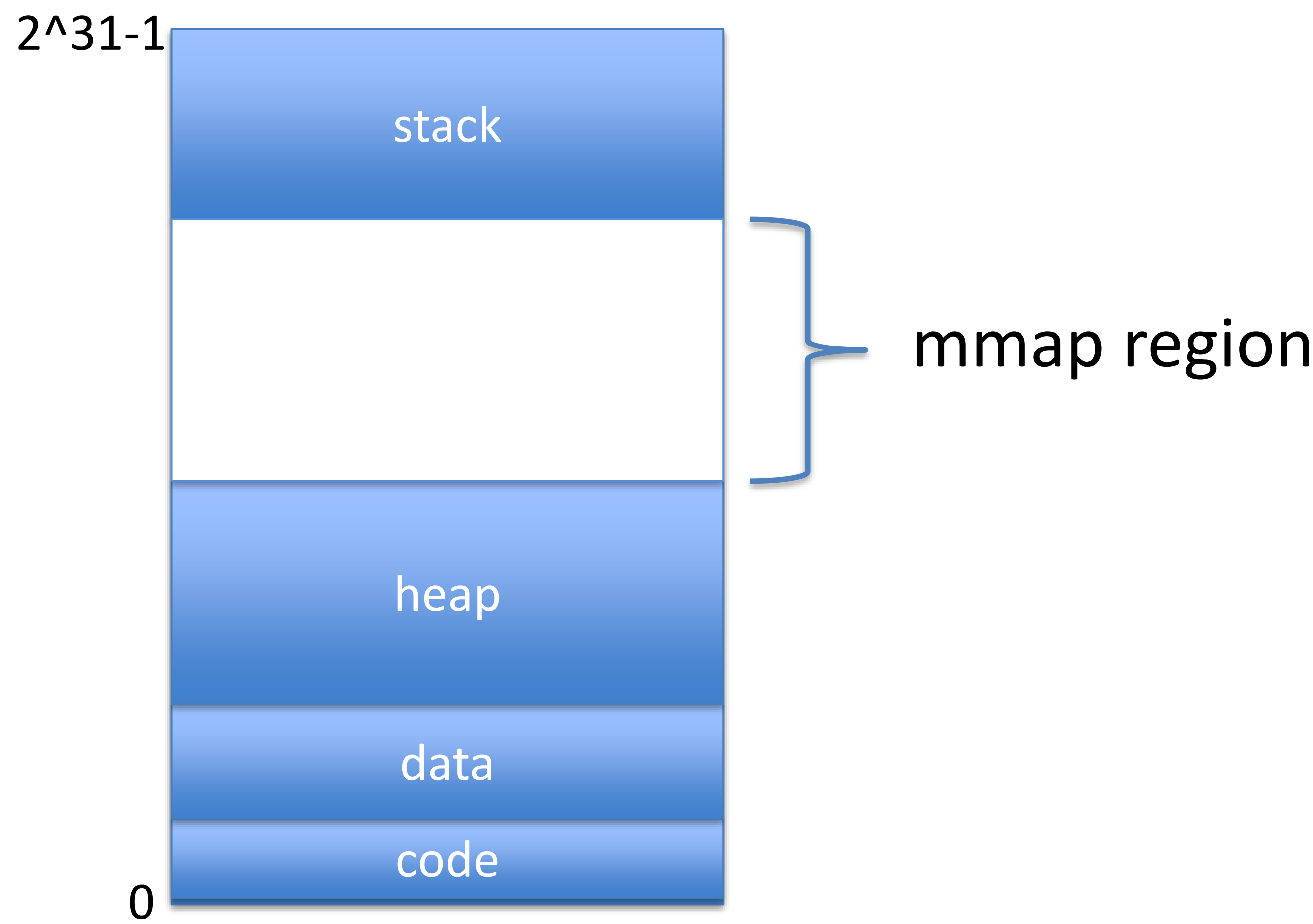
Alternative File Access Method: Memory Mapping

- mmap()
  - Map the contents of a file in memory
- munmap()
  - Remove the mapping

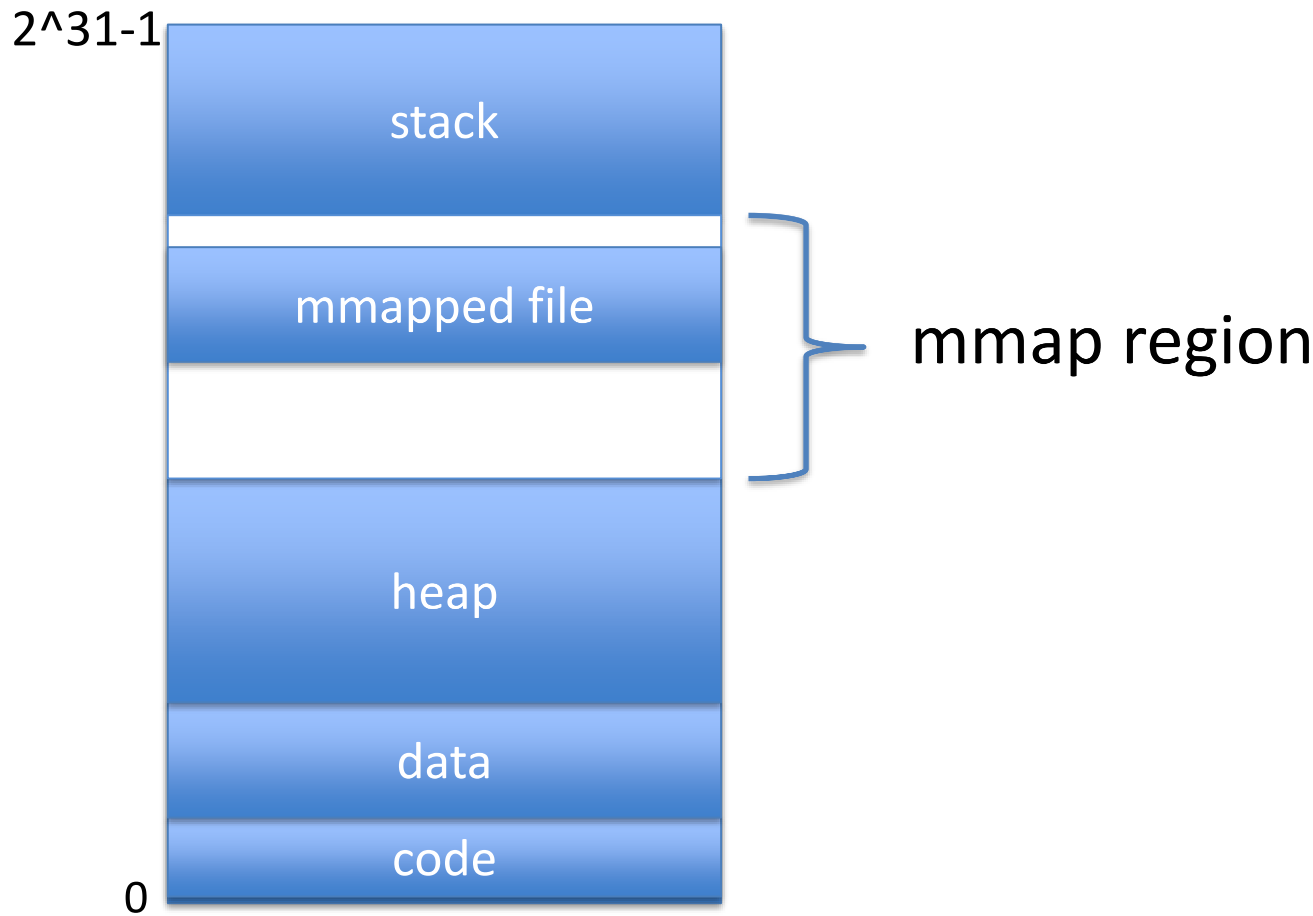
Remember this Picture? Typical Virtual Address Space



Remember this Picture? Typical Virtual Address Space



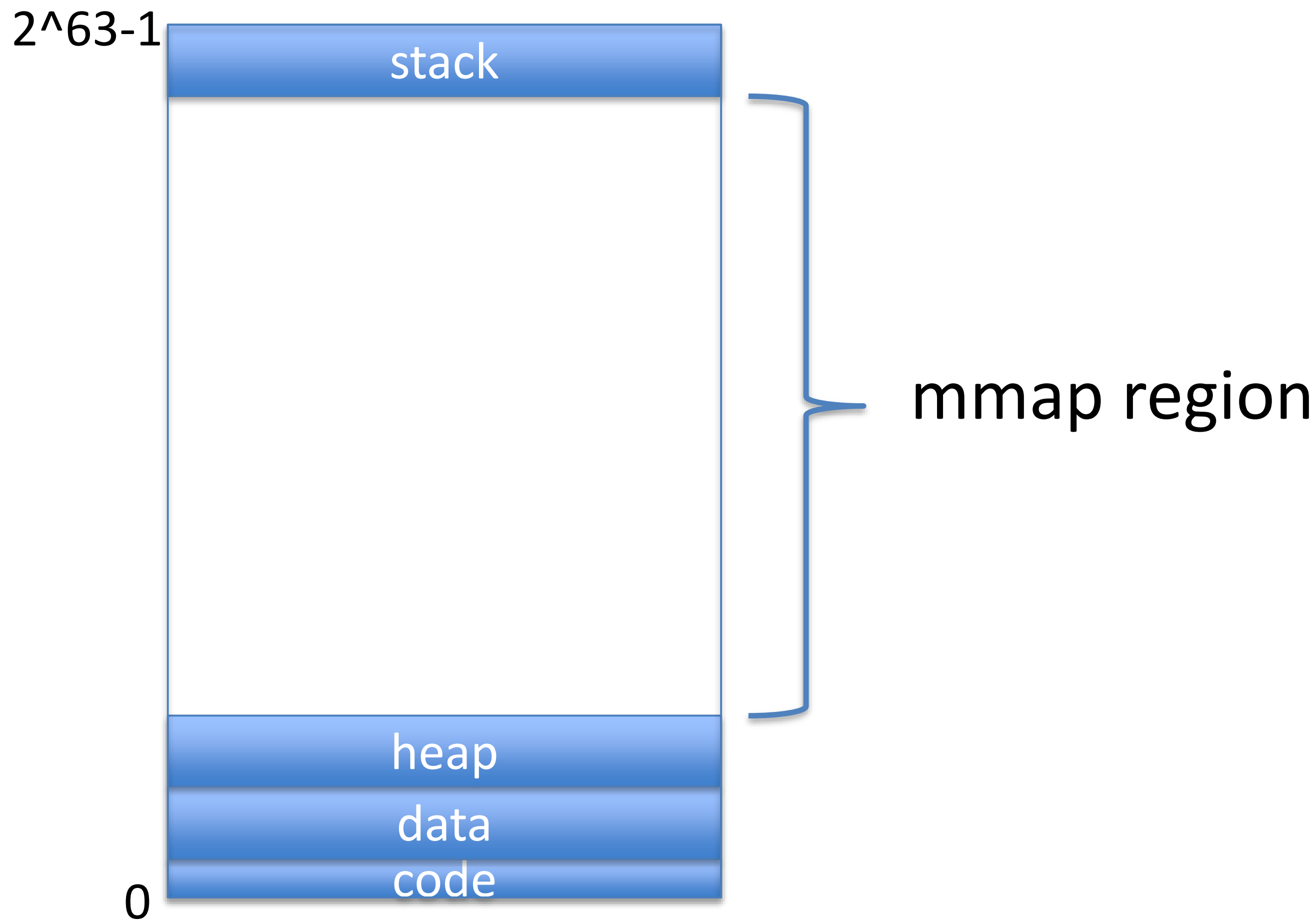
Remember this Picture? Typical Virtual Address Space



Remember Large Address Spaces?

- 64 bit address space
- Do you know now understand why desirable?
- 32 bits → 4 GBytes
- A few big files mmap()-ed
- You are out of virtual address space!

64-bit Address Space: Huge mmap() Region



Example with 3 (Large) Files Mapped

Access to mmap()-ed Files

- Access to memory region mmap()-ed
- Causes page fault
- Causes page/block of file to be brought in

mmap() Implementation - 1

- On mmap()
  – Allocate page table entries
  – Set valid bit to “invalid”

mmap() Implementation - 2

- On mmap()
  - Allocate page table entries
  - Set valid bit to “invalid”
- On access,
  - Page fault
  - File = backing store for mapped region of memory
  - Just like in demand paging
  - Except paged from mapped file

mmap() Implementation - 3

- On mmap()
  – Allocate page table entries
  – Set valid bit to “invalid”
- On access,
  – Page fault
  – File = backing store for mapped region of memory
  – Just like in demand paging
  – Except paged from mapped file
- After page fault handling
  – Set valid bit to true

How to get data to disk for mmap?

- Through normal page replacement
- Or through an explicit call *msync()*

What is mmap() good for?

- Random access to large file

Random Access with mmap()

- addr = mmap()
- Use memory addresses in [addr, addr+len-1]

Random Access with Read() Interface

- Open
- Read entire file into memory buffer
- Then use memory address in buffer

Advantage with mmap()

- Only accessed portions brought in memory

- Huge advantage
  - For large files
  - Sparsely accessed

Random Access with LSeek()

- Open
- LSeek
- Read into Buffer
- Lseek
- Read into Buffer

Advantage with mmap()

- Much easier programming model
  - Follow pointer in memory
  - As opposed to (Lseek, Read) every time
- Easier if reuse
  - VM system keeps page for you
  - Otherwise, have to do your own replacement

mmap() Advantages for Random Access

- Easy to write
- Only bring in memory what you read
- Easy reuse

Issues with mmap()

- Alignment on page boundary
- Not easy to extend a file
- For small files
  - Read() more efficient than mmap() + page fault

Another Use of mmap()

- Sharing memory between processes
- A form of interprocess communication
- Use shared and anonymous map flags

File System/memory Management Implementation

- File system has buffer cache
  - File data on disk, recently used data in memory
- Memory management has page replacement
  - Data in memory, not recently used data on disk
- Same thing, but from an opposite angle

Integrated Buffer Cache

- One region of memory
- Used both as
  - File system buffer cache
  - Demand paged in-memory data

- Advantage:
  - One piece of code instead of two
  - Avoids “double caching”

Summary

- Device directory
- File data allocation methods
  - Indexed, indirect, extent-based methods
- Free bitmap

- Cache (and cache directory)
- Queues of pending requests
- Active file table and open file table(s)

- Memory-mapped files