Chapter IX
File Systems

Jehan-François Pâris
jfparis@uh.edu

Chapter overview

- General organization
- Protection
- UNIX Implementation
  - FFS
  - Journaling file systems
- Recent file systems
- Mapped files

General Organization

The file system

- Provides ***long term storage*** of information.
- Will store data in ***stable storage*** (disk)
- Cannot be RAM because:
  - ***Dynamic RAM*** loses its contents when powered off
  - ***Static RAM*** is too expensive
  - System crashes can corrupt contents of the main memory

A file system

File and file names

- Data managed by the file system are grouped in ***user-defined*** data sets called ***files***
- The file system must provide a mechanism for ***naming*** these data
  - Each file system has its own set of conventions
  - All modern operating systems use a ***hierarchical directory structure***

Windows solution (I)

- Each device and each disk partition is identified by a letter
  - A: and B: were used by the floppy drives
  - C: is the first ***disk partition o***f the hard drive
  - If hard drive has no other disk partition,
    D: denotes the DVD drive
- Each device and each disk partition has its ***own hierarchy of folders***

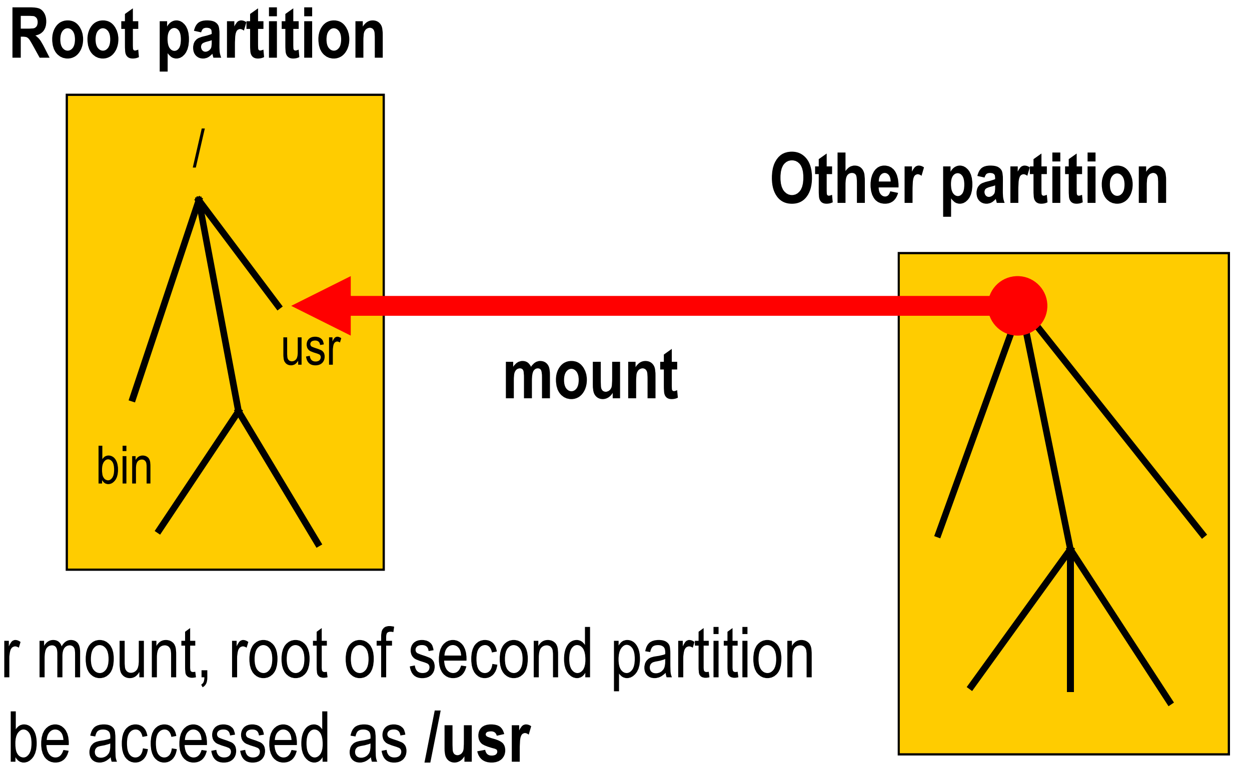
Windows solution (II)

- In a hierarchical file system files are grouped in ***directories*** and ***subdirectories***
  - The ***folders*** and ***subfolders*** of Windows
- These directories and subdirectories form ***one tree*** in each disk partition

UNIX solution

- Each device and disk partition has its own directory tree
  - Disk partitions are glued together through the ***mount*** operation to form a single tree
    - Typical user does not know where her files are stored
  - Devices form a separate device hierarchy
    - Can also be ***automounted***

“Mounting” a file system



After mount, root of second partition can be accessed as **/usr**

File organizations (I)

- Earlier file systems organized files into ***user-specified records***
  - *They were read and written atomically*
- Starting with UNIX modern file systems organize files as sequence of bytes
  - Can be read or written to in an arbitrary fashion

File organizations (II)

- Files are stored on disk using fixed-size records called ***blocks***
  - All files stored on a given device or disk partition have the ***same block size***
- Block sizes are ***transparent to the users***
  - They rarely know them

The case for fixed-size blocks (I)

- Programmer defined records were often too small
  - A grade file would have had one record per student
    - Around 100 bytes
  - Can pack around 40 student records in a single 4-kilobyte block.
    - One single read replaces 40 reads

The case for fixed-size blocks (II)

- Could not read a file without knowing its record format
  - Hindered the development of utility programs

Selecting the block size

Selecting the block size

- Much more important issue than selecting the page size of a VM system because
  - Many very small files
    - Small UNIX test files, …
  - Some very large files
    - Music, video, …

The 80-20 rule

- We can roughly say that
  - 80 percent of the files occupy 20 percent of the disk space
  - Remaining 20 percent occupy the remaining 80 percent

The dilemma

- Small block sizes
  - Minimize internal fragmentation
    - Best for storing small files
  - Provide poor data transfer rates for large files
    - Too many small data transfers

- There is no single optimum block size
  - Depends too much on file sizes

Protection

Objective

- To provide controlled access to information
- Both Windows and UNIX let file owners decide who can access their files and what they can do
  - Not true for more secure file systems
    - They enforce ***security restrictions***

Enforcing controlled access

- ***Two basic solutions***
  - Access control lists
  - Tickets
- Each of them has its advantages and disadvantages

Access control lists (I)

- Table specifying what each user can do with the file

User	***Permissions***
Alice	read, write
Bob	read
Donna	read, write

Access control lists (II)

Access control lists (III)

- ***Main advantage:***
  - ***Very flexible:*** can easily add new users or change/revoke permissions of existing users
- ***Two main disadvantages:***
  - ***Very slow:*** must authenticate user at each access
  - Can take more space than the file itself

Tickets (I)

- Also known as ***capabilities***
- Specify what the ticket holder can do
- Must prevent users from forging tickets
  - Use ***encryption***
    - *Similar to using patterns that are hard to forge on bills*
  - Let kernel maintain them
    - *Similar to bank doing all the bookkeeping for our accounts*

Tickets (II)

Tickets (III)

- ***<u>Main advantage:</u>***
  - ***Very fast:*** must only check that the ticket is valid
- ***<u>Two main disadvantages:</u>***
  - ***Less flexible than access control lists:*** cannot revoke individual tickets
  - ***<u>Less control:</u>*** ticket holders can make copies of tickets and distribute them to other users

Conclusion

- Best solution is to ***combine both approaches***
  - Use access control lists for long-term management of permissions
    - Once a user has been authenticated, give him or her a ticket
    - Limit ticket lifetimes to force users to be authenticated from time to time

UNIX solution

- UNIX
  - Checks access control list of file whenever a file is opened
  - Lets file descriptor act as a ticket until the file is closed

UNIX access control lists (I)

- File owner can specify three access rights
  - ***read***
  - ***write***
  - ***execute***

  for
  - herself (***user***)
  - a group in /etc/group (***group***)
  - all other users (***other***)

UNIX access control lists (II)

- Three groups of three access rights
  - Nine bits
    - Can be tuned on and off

User
(owner) **Group** **Other**
rwxrwxrwx

UNIX access control lists (III)

- `rwx------`
  Owner can do everything she wants with her file and nobody else can access it
- `rw-r--r--`
  Owner can read from and write to the file, everybody else can read the file
- `rw-rw----`
  Owner and any member of group can read from and write to the file

UNIX access control lists (IV)

- ***Main advantage:***
  - ***Takes very little space:***
    9 bits plus 32 bits for group-ID
- ***Main disadvantage***
  - ***Less flexible than full access control lists:***
    Groups are managed by system administrator
    - Works fairly well as long as groups remain stable

Unix File Semantics

File types

- Three types of files
  - ***ordinary files****:*
    uninterpreted sequences of bytes
  - ***directories****:*
    accessed through special system calls
  - ***special files****:*
    allow access to hardware devices

Ordinary files (I)

- Five basic file operations are implemented:

`open()`	returns a file descriptor
`read()`	reads so many bytes
`write()`	writes so many bytes
`lseek()`	changes position of current byte
`close()`	destroys the file descriptor

Ordinary files (II)

- All reading and writing are sequential.

  The effect of direct access is achieved by manipulating the offset through **lseek()**
- Files are stored into fixed-size *blocks*
- Block boundaries are hidden from the users

  Same as in MS-DOS/Windows

The file metadata

- Include file size, file owner, access rights, last time the file was modified, …
  but not the ***file name***
- Stored in the file ***i-node***
- Accessed through special system calls:
  chmod(), chown(), ...

I/O buffering

- UNIX caches in main memory
  - I-nodes of opened files
  - Recently accessed file blocks
- Delayed write policy
  - Increases the I/O throughput
  - Will result in lost writes whenever a process or the system crashes.
- Terminal I/O are buffered one line at a time.

Directories (I)

- Map file names with i-node addresses

Name	I-node
vi	203
edit	203
pico	426
emacs	173
…	…

- ***Do not contain any other information!***

Directories (II)

- Two directory entries can point to the same i-node
- Directory subtrees cannot cross file system boundaries unless a new *file system* is ***mounted*** somewhere in the subtree
- To avoid loops in directory structure, ***directory files*** cannot have more than ***one pathname***

Special files (I)

- Not files but devices:
  - `/dev/tty` is your current terminal
  - `/dev/sdb0` your flash drive
  - …
- ***Advantage:***
  - Allows to access devices such as flash drives, tape drive, … as if they were regular files

Special files (II)

- ***Disadvantage:***
  - We want to see flash drives as file systems integrated in our file system hierarchy not as single files
- A better solution is to mount them automatically when they get inserted **`(automount`**)
  - Windows solution
  - **`media/usb[0-7]`** on Ubuntu

Unix File System Internals

Version 7 Implementation

- Each disk partition contains:
  - A ***superblock*** containing the parameters of the file system disk partition
  - An ***i-list*** with one ***i-node*** for each file or directory in the disk partition and a ***free list***.
  - ***Data blocks*** (512 bytes)

A disk partition (“filesystem”)

Superblock

I-nodes

Data Blocks

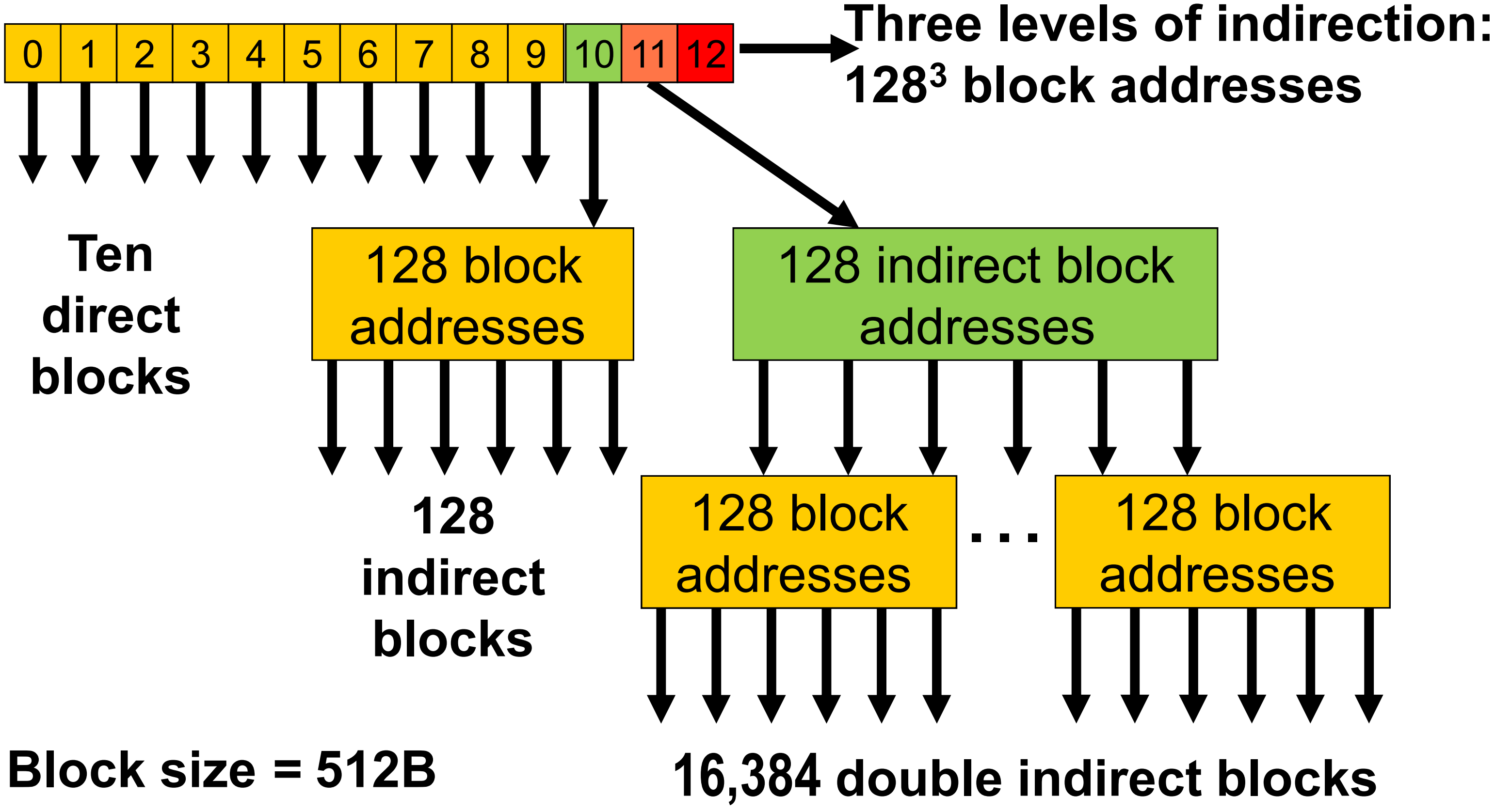
The i-node (I)

- Each ***i-node*** contains:
  - The ***user-id*** and the ***group-id*** of the file owner
  - The file protection bits,
  - The file size,
  - The times of file creation, last usage and last modification,

The i-node (II)

  - The *number* of directory entries pointing to the file, and
  - A flag indicating if the file is a directory, an ordinary file, or a special file.
  - Thirteen block addresses
- The file name(s) can be found in the directory entries pointing to the i-node

Storing block addresses



How it works (I)

- First ten blocks of file can be accessed directly from i-node
  - 10x512= 5,120 bytes
- Indirect block contains 512/4 = 128 addresses
  - 128x512= 64 kilobytes
- With two levels of indirection we can access 128x128 = 16K blocks
  - 16Kx512 = 8 megabytes

How it works (II)

- With three levels of indirection we can access 128x128X128 = 2M blocks
  - 2Mx512 = 1 gigabyte
- Maximum file size is
  1 GB + 8 MB + 64KB + 5KB

Explanation

- File sizes can vary from a few hundred bytes to a few gigabytes with a hard limit of 4 gigabytes
- The designers of UNIX selected an i-node organization that
  - Wasted little space for small files
  - Allowed very large files

Discussion

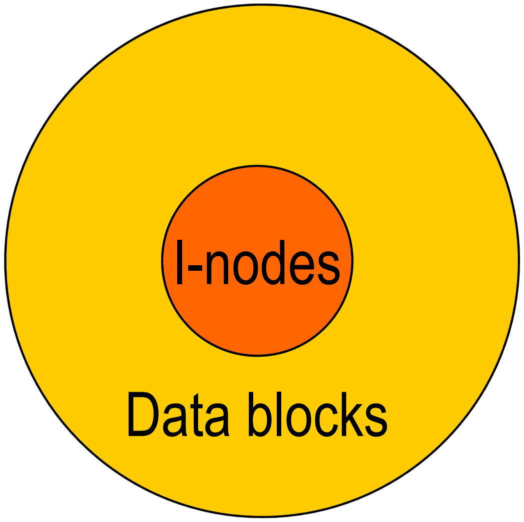
- What is the true cost of accessing large files?
  - UNIX caches i-nodes and data blocks
  - When we access sequentially a very large file we fetch only once each block of pointers
    - Very small overhead
  - Random access will result in more overhead

FFS Modifications

- BSD introduced the “fast file system” (***FFS***)
  - ***Superblock*** is replicated on different cylinders of disk
  - Disk is divided into ***cylinder groups***
  - Each cylinder group has its own i-node table
    - It minimizes disk arm motions
  - Free list replaced by ***bit maps***

Cylinder groups (I)

- In the old UNIX file system i-nodes were stored apart from the data blocks

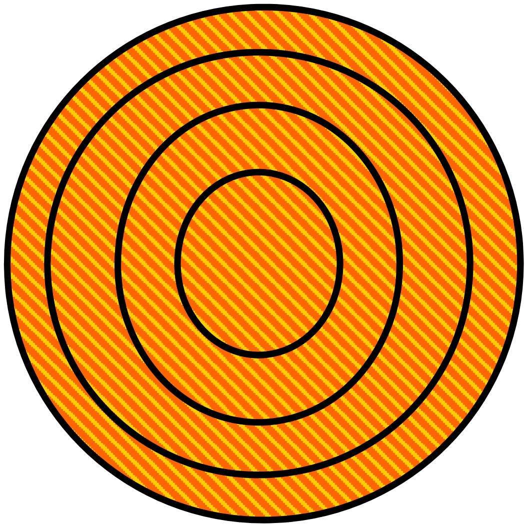


Too many ***long seeks***

➢ ***Poor disk throughput***

Cylinder groups (II)

- FFS partitions the disk into cylinder groups containing both i-nodes and data blocks



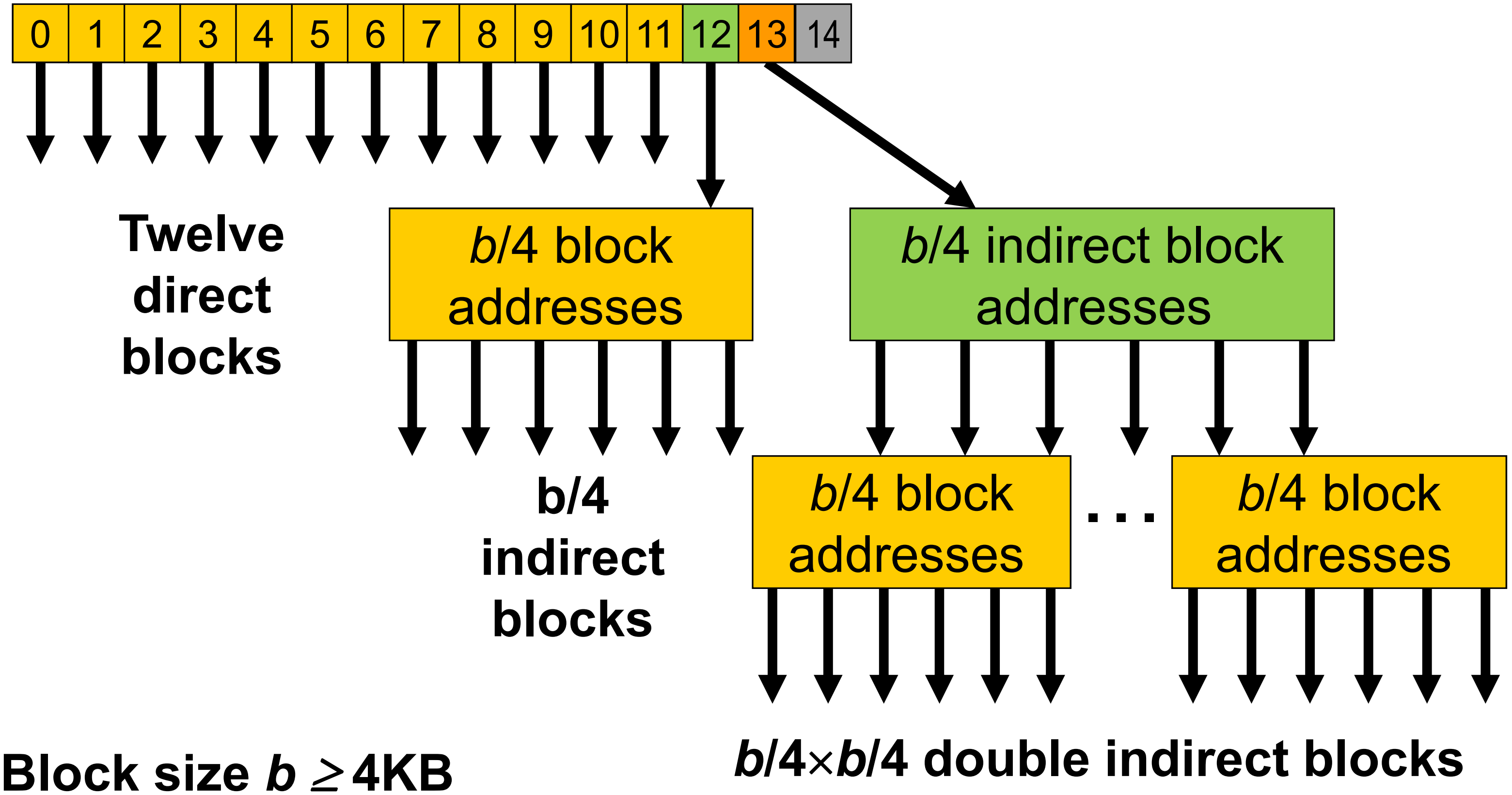
Most files have their data blocks in the ***same cylinder group*** as their i-node

➢ ***Problem solved***

The FFS i-node

- I-node has now 15 block addresses
- Minimum block size is 4K
  - 15th block address is never used

FFS organization (I)



Twelve direct blocks

b/4 block addresses

b/4 indirect blocks

b/4 indirect block addresses

b/4 block addresses

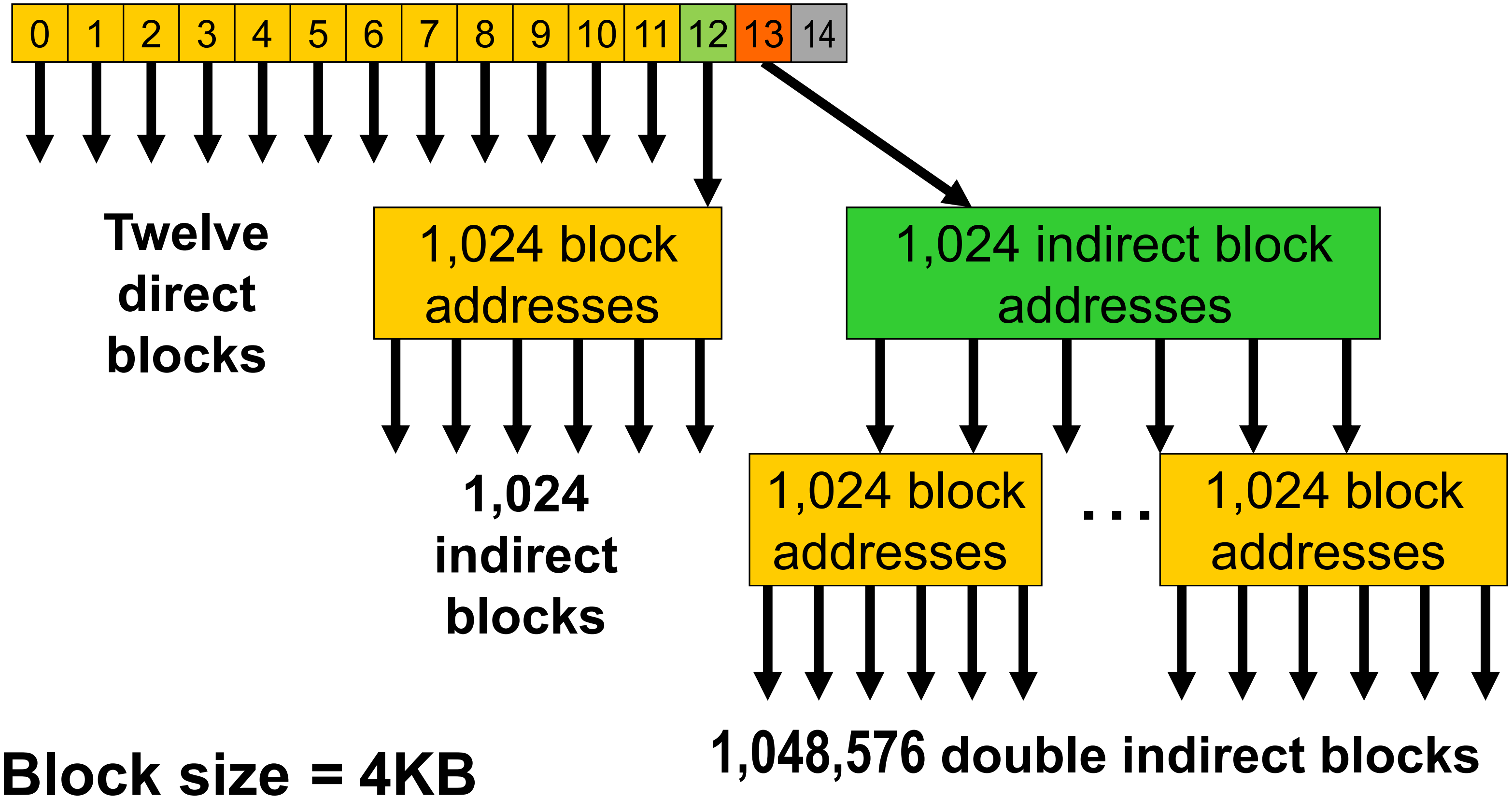
. . .

b/4 block addresses

***b*/4×*b*/4 double indirect blocks**

Block size *b* ≥ 4KB

FFS organization (II)



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Twelve direct blocks

1,024 block addresses

1,024 indirect blocks

1,024 indirect block addresses

1,024 block addresses . . . 1,024 block addresses

1,048,576 double indirect blocks

Block size = 4KB

How it works

- In a 32 bit architecture, file size is limited to 2^{32} bytes, that is, 4GB
- When block size is 4KB, we can access
  - **12 ×4KB = 48KB *directly*** from i-node
  - **1,024 ×4KB = 4MB** with ***one level*** of indirection
  - **4GB – 48KB – 4MB** with ***two levels*** of indirection

The bit maps

- Each cylinder group contains a bit map of all available blocks in the cylinder group

  The file system will attempt to keep consecutive blocks of the same file on the same cylinder group

Block sizes

- FFS uses larger blocks allows the division of a single file system block into 2, 4, or 8 fragments that can be used to store
  - Small files
  - Tails of larger files

Explanations (I)

- Increasing the block size to 4KB eliminates the third level of indirection
- Keeping consecutive blocks of the same file on the same cylinder group reduces disk arm motions

Explanations (II)

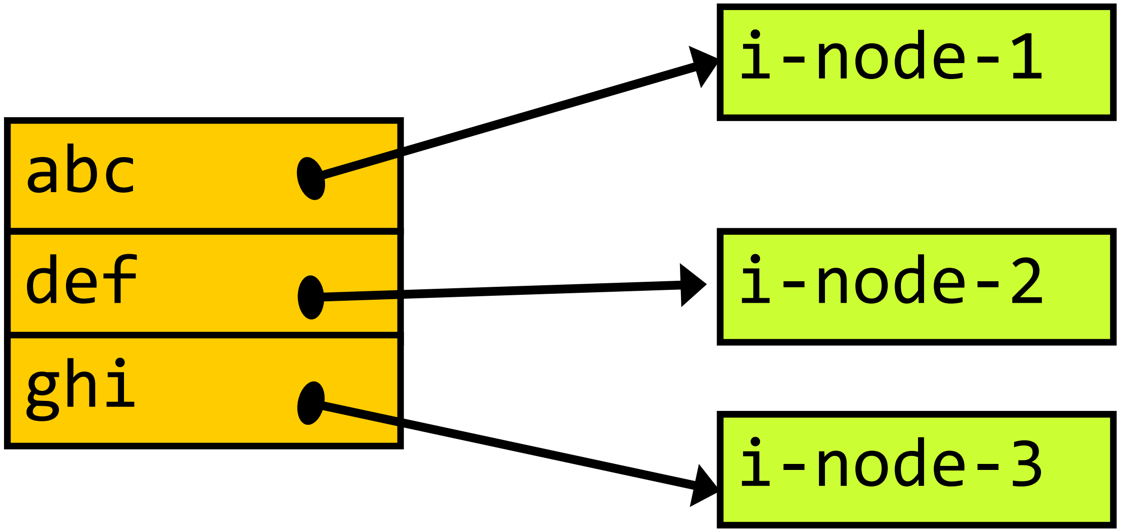
- Allocating full blocks and block fragments
  - allows efficient sequential access to large files
  - minimizes disk fragmentation
- Using 4K blocks without allowing 1K fragment would have wasted **45.6%** of the disk space
  - This would not true today

Speeding up metadata updates

Metadata issues

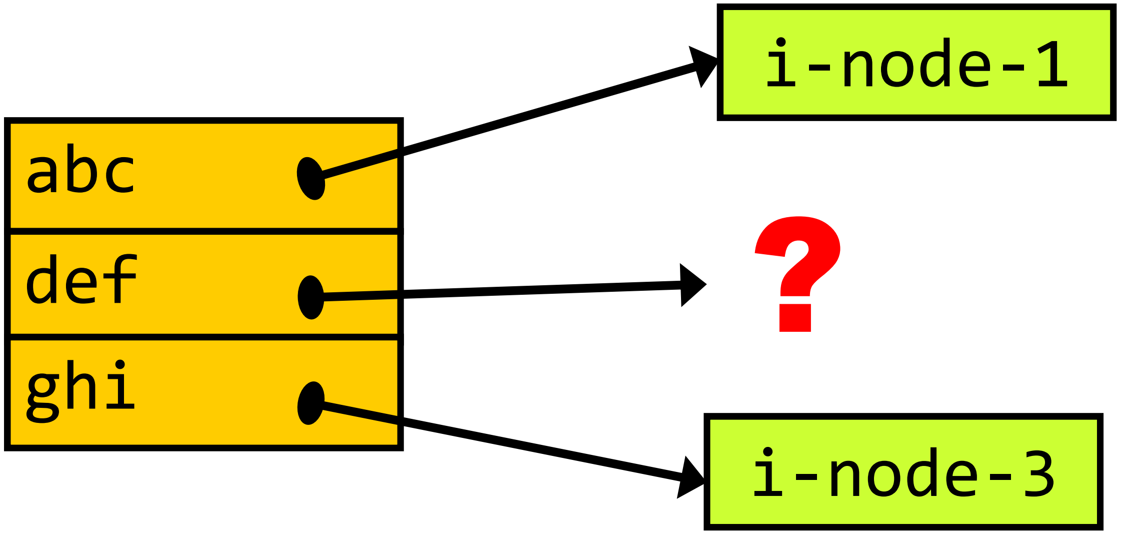
- Most of the good performance of FFS is due to its extensive use of **I/O buffering**
  - **Physical** writes are totally **asynchronous**
- ***Metadata updates*** must follow a ***strict order***
  - FFS uses ***blocking writes*** for all metadata updates
  - More recent file systems use better solutions

Deleting a file (I)



Assume we want to delete file "`def`"

Deleting a file (II)

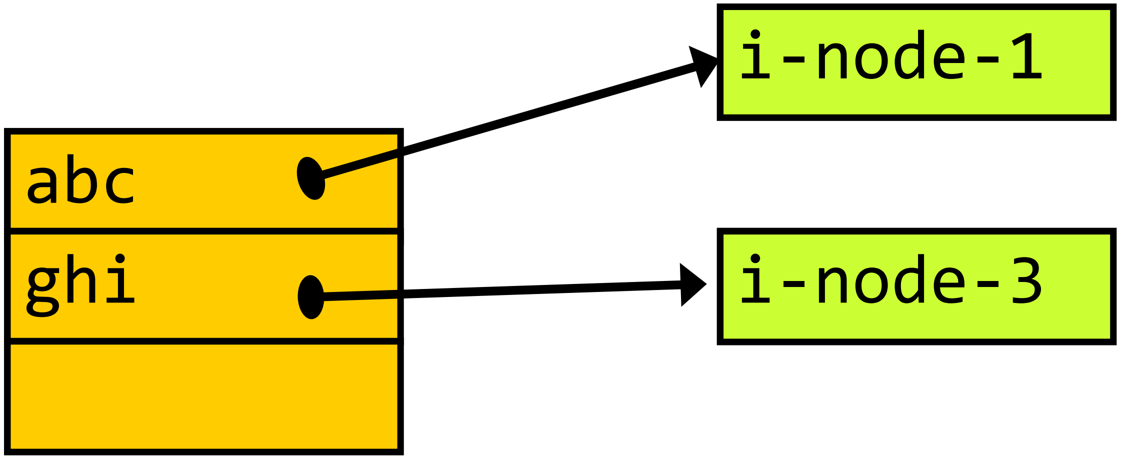


Cannot delete i-node before deleting directory entry “`def`”

Deleting a file (III)

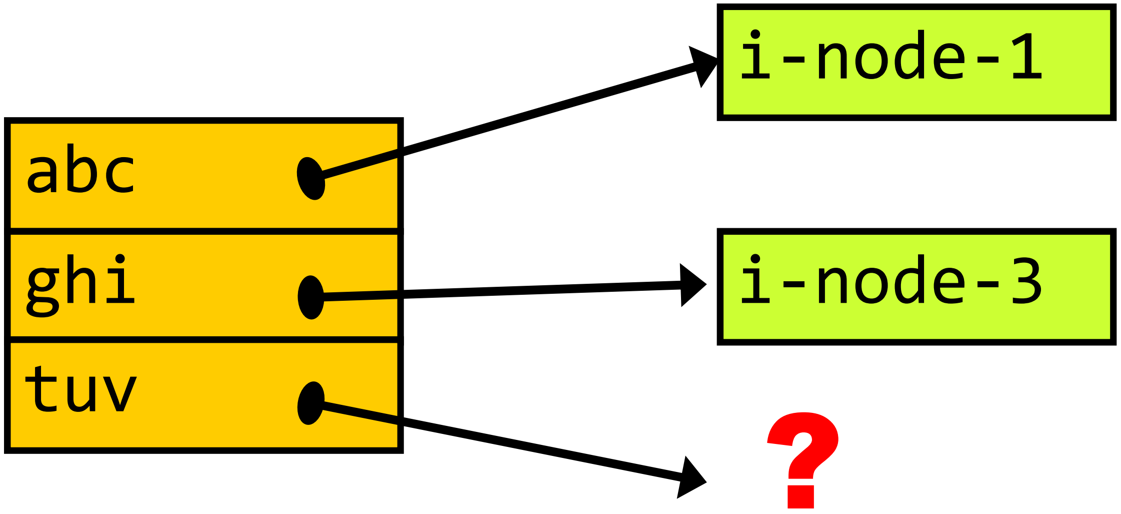
- Correct sequence is
  1. Write to disk directory block containing deleted directory entry “def”
  2. Write to disk i-node block containing deleted i-node
- Leaves the file system in a consistent state

Creating a file (I)



Assume we want to create new file "`tuv`"

Creating a file (II)



Cannot write add entry “`tuv`” to before creating the corresponding new i-node

Creating a file (III)

- Correct sequence is
  1. Write to disk i-node block containing new i-node
  2. Write to disk directory block containing new directory entry
- Leaves the file system in a consistent state

Handling metadata updates

- Out-of-order metadata updates can leave the file system in ***temporary inconsistent state***
  - Not a problem as long as the system does not crash between the two updates
  - Systems are known to crash

FFS Solution

- FFS performs ***synchronous updates*** of ***directories*** and ***i-nodes***
  - Requires ***many more seeks***
  - Causes a serious ***performance bottleneck***

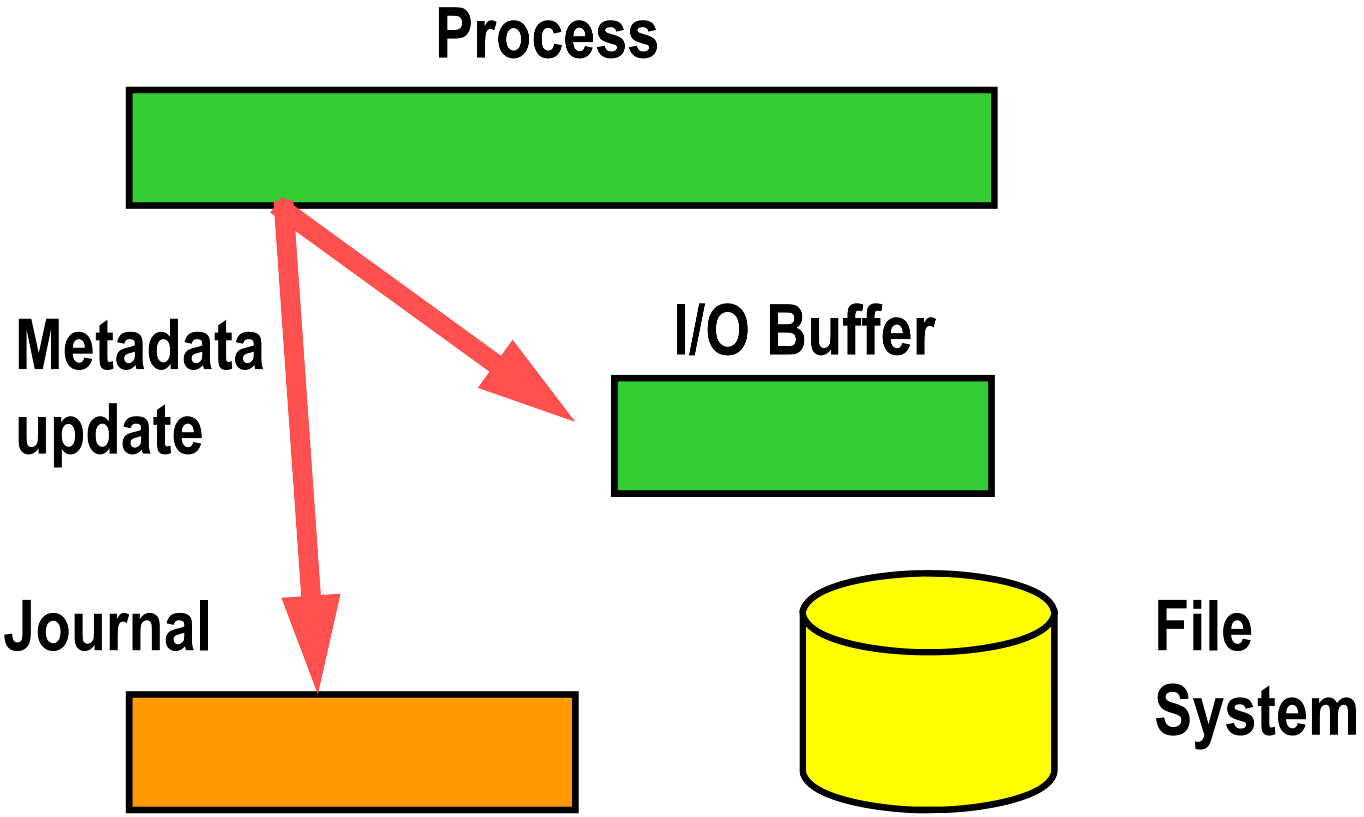
Better solutions

- Log-structured file systems
    - BSD-LFS
- Soft updates
- ***Journaling file systems***
    - Most popular approach

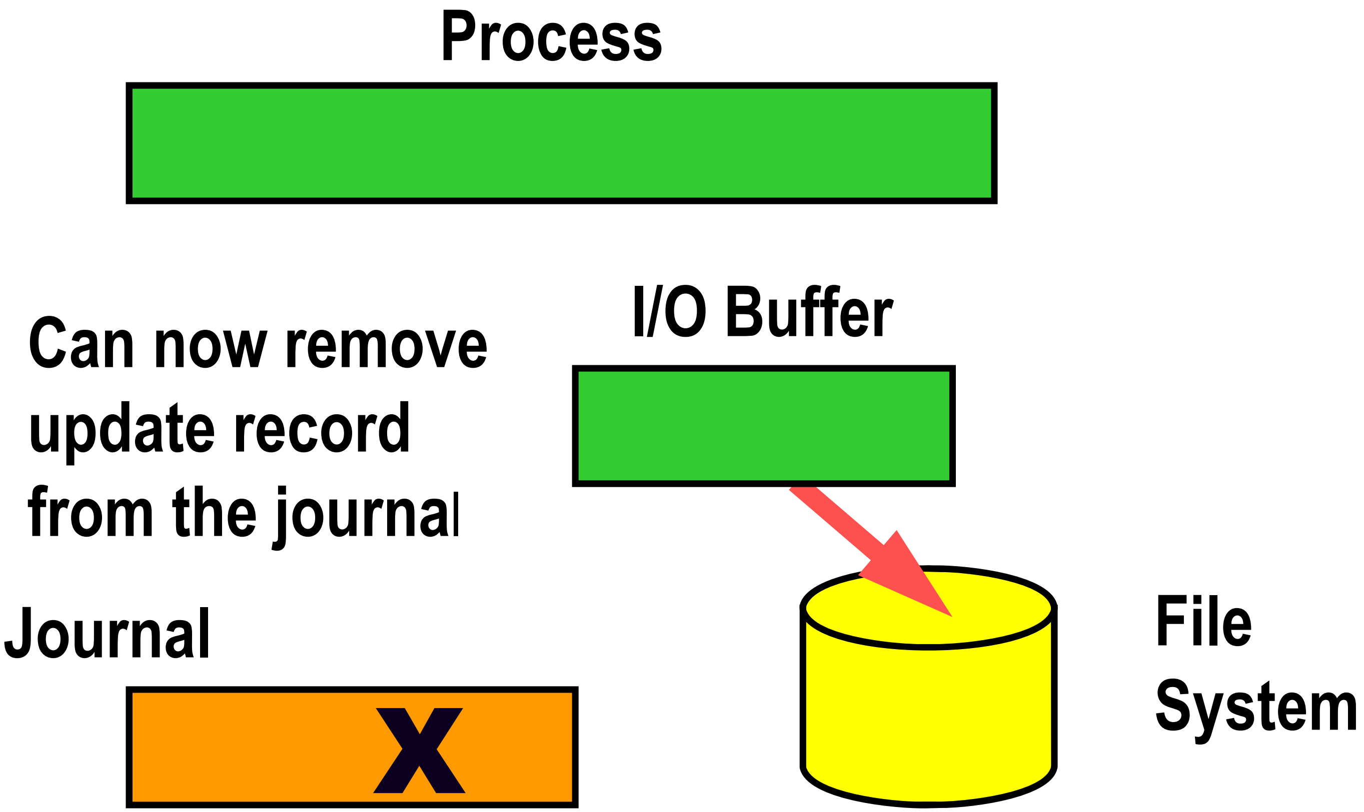
Journaling file systems

- ***Key Idea:***
  - Record metadata updates
    - First on a log (the ***journal***)
    - Later at their proper location
  - When recovering from a crash, use the journal to finalize all incomplete metadata updates

Step 1: update buffer and journal



Step 2: update the file system



Process

I/O Buffer

Can now remove
update record
from the journal

Journal

File
System

Explanations

- Metadata updates are ***written twice*** on disk
    - ***First*** in the ***journal***
    - ***Then***, and only then, at the proper place in the file system

- All other updates remain ***asynchronous***

Advantage

- Writing metadata updates twice is still cheaper than using a single blocking write because
    - Journal is organized as a log and all writes are sequential
    - Second update is ***non-blocking***

Implementation rules

- Journaling file system must ensure that
  - Every update is written first in the journal ***before*** the file system is updated
  - Journal entries cannot be removed until the corresponding updates have been propagated to the file system
- ***Complicates I/O buffer design***

Synchronous JFSes

- Write all metadata updates ***one by one*** in the journal without any delay
- Guarantee file system will always recover to a consistent state
- Guarantee that metadata updates will ***never be lost***
  - *All updates are **durable***

Asynchronous JFSes

- Writes to the journal are buffered until an entire buffer is full
- Guarantee file system will always recover to a ***consistent state***
- Do not guarantee that metadata updates will never be lost
- Are ***much faster*** than synchronous JFS

Recent File Systems

Linux file systems

- First Linux file system was a port of Minix file system
  - Essentially a "toy" file system
  - Maximum file size was 64MB
- Many more recent file systems
  - Ext1, ext2, ext3, ext4, …
  - Others

Ext2

- Was essentially analogous to the UNIX fast file system we have discussed
  - Fifteen block addresses per i-node
  - Cylinder groups are called ***block groups***
- Major differences include
  - Larger maximum file size: 16 GB - 2 TB
  - Various extensions
    - Online compression, full ACLs, …

Ext3fs

- Offers three levels of journaling
  - ***Journal:*** journals metadata and data updates
  - ***Ordered:*** guarantees that data updates will be written to disk before associated metadata are marked as committed
  - ***Writeback***: makes no such guarantees

Ext4fs (I)

- Evolution from ext3fs
  - Can mount an ext4fs partition as ext3fs or an ext3fs partition as ext4fs
- 64-bit file system
  - 48-*bit block addresses*
- Can support very large volumes
  - One exabyte, that is, 2^{30} gigabytes!
  - Very large files (16 terabytes)

Ext4fs (II)

- Can support ***extents***
  - Becomes then incompatible with ext3fs
- Uses ***delayed extent allocation***
  - ***Reduces file fragmentation***
    - *Especially when file grows*
- ***Checksums*** contents of journal
  - More reliable

Windows file system (NTFS)

- Another journaling file system
- Each file is an object composed of one or more ***data streams***
  - ***"Only the main stream of a file is preserved when it is copied to a FAT-formatted USB drive, attached to an e-mail, or uploaded to a website."***

Wikipedia

NTFS data structures

- ***Master File Table*** (MFT)
  - Contains most metadata
  - Equivalent to UNIX i-node table
- Each file can have one or more MFT records depending on file size and attribute complexity
- MFT records contain
  - Pointers to data blocks for most files
  - Contents of very small files

NTFS block allocation policy

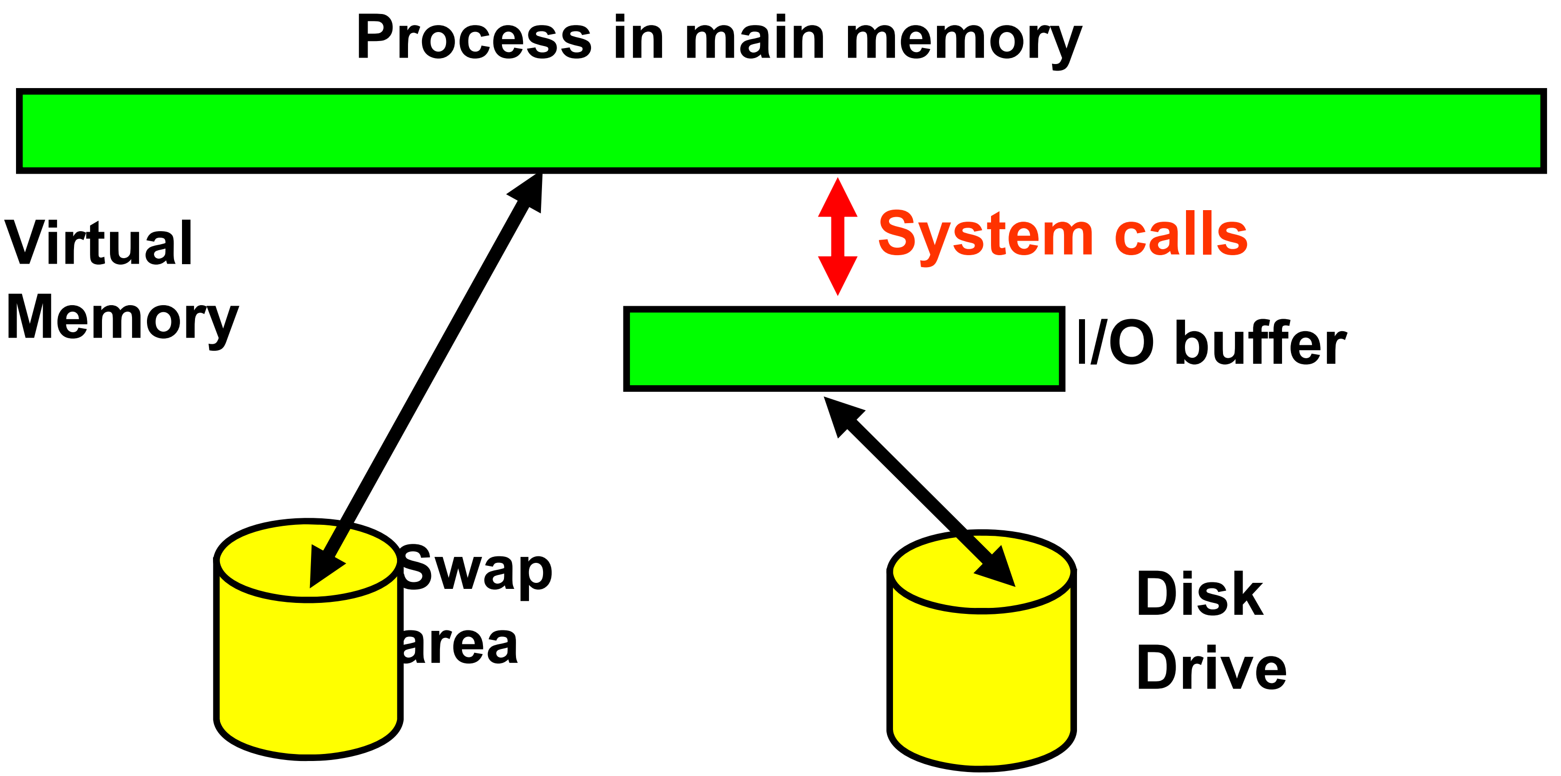
- Allocates block clusters instead of individual blocks.
    - Each cluster has space for several contiguous blocks
    - Cluster size is defined when the disk drive is formatted
    - Improves performances but increases internal fragmentation

*As disk capacities are now measured in terabytes,
we are more willing to sacrifice a few megabytes of disk
space to internal fragmentation in order to obtain a better
overall performance of the file system.*

Mapped Files

Virtual memory and I/O buffering (I)

- **Now:**



Process in main memory

Virtual Memory

System calls

I/O buffer

Swap area

Disk Drive

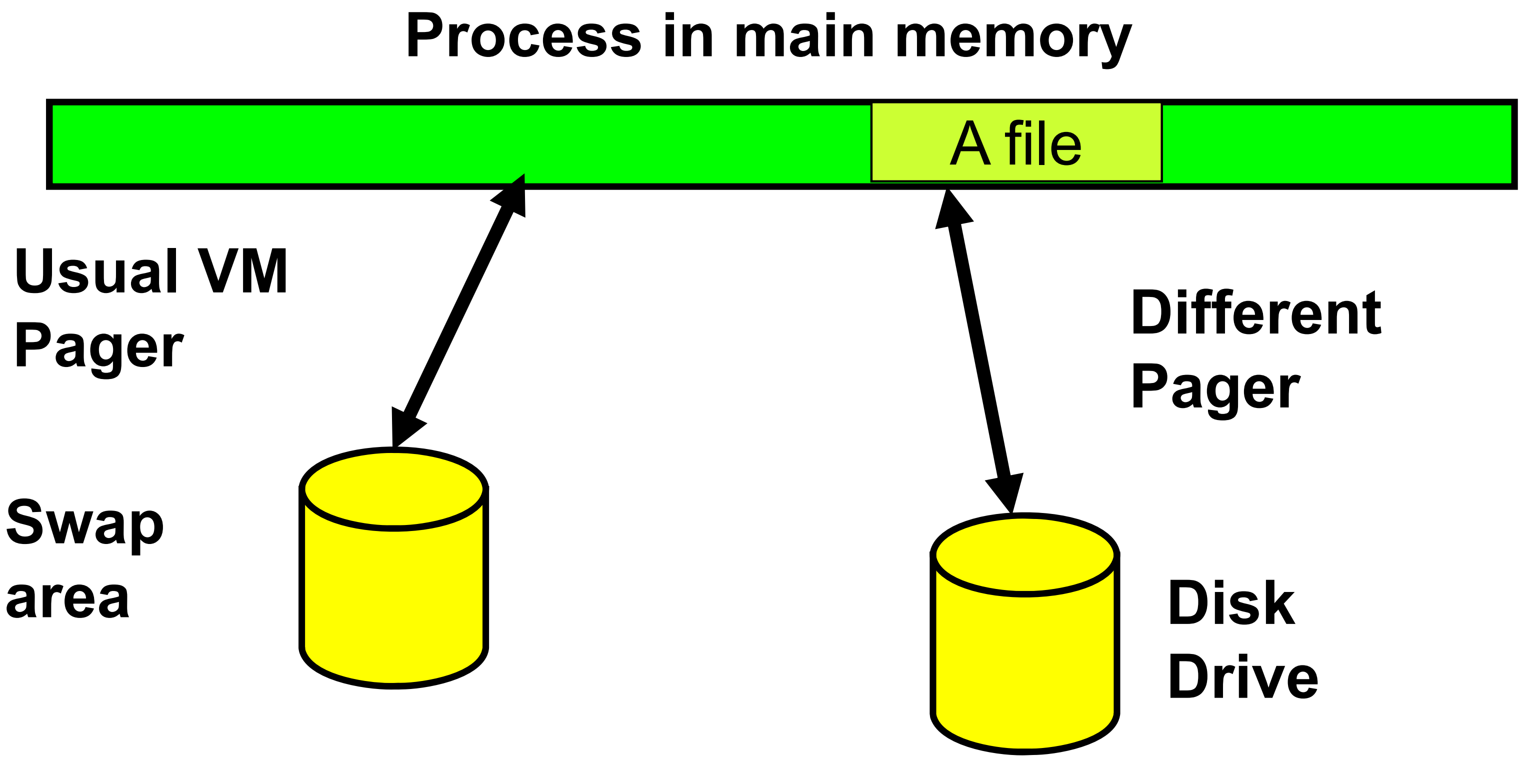
Virtual memory and I/O buffering (II)

- In a VM system, we have
  - **Implicit transfers** of data between main memory and swap area (page faults, etc.)
  - **Implicit transfers** of information between the disk drive and the system I/O buffer
  - **Explicit transfers** of information between the I/O buffer and the process address space controlled by the programmer

Virtual memory and I/O buffering (III)

- I/O buffering greatly reduces number of disk accesses
- Each I/O request must still be serviced by the OS:
  - Two context switches per I/O request
- Why could we not ***map files directly into*** the process ***virtual address space***?

Mapped files (I)



Process in main memory

A file

Usual VM Pager

Different Pager

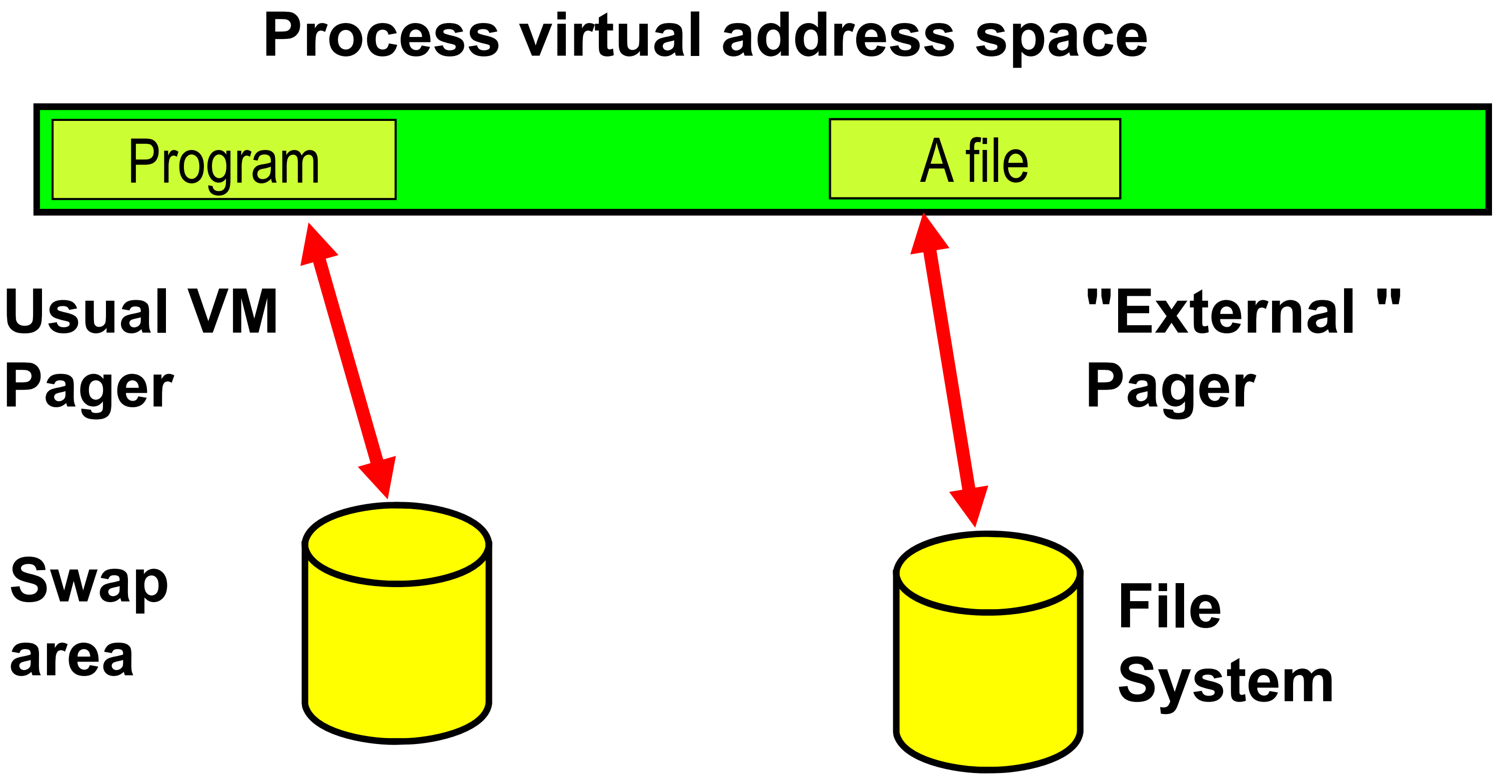
Swap area

Disk Drive

Mapped files (II)

- When a process opens a file, the whole file is mapped into the process virtual address space
  - **No data transfer takes place**
- File blocks are brought in memory ***on demand***
- File contents are accessed using regular program instructions (or library functions)
- Shared files are in ***shared memory segments***

Mach implementation (I)



Process virtual address space

Program

A file

Usual VM Pager

"External " Pager

Swap area

File System

Mach implementation (II)

- Mach organizes active parts of virtual address space of each process into ***address ranges***

- Each address range can have a different pager
  - Executable in file system for code segment
  - Swap area for data segment
  - Files themselves for mapped files

Linux implementation (I)

- **mmap(…)**
  - Maps files or devices into memory
  - Implements demand paging
    - File blocks are brought ***on demand***
      - ***Lazy*** approach
  - Can map a portion of a file (offset + number of bytes)

Syntax

-
```
#include <sys/mman.h>
void *mmap(void *addr,
           size_t length,
           int prot,
           int flags,
           int fd,
           off_t offset);
```

Protection(rwx)

File descriptor

Start offset

- Must ***first open*** the file!

A few options and flags

- Setting ***addr*** to **NULL** lets the system choose the start address of the mapped file
- Flag **MAP_SHARED** makes updates to the mapping visible to all processes that map the file
- Flag **MAP_PRIVATE** keeps these updates private
- Flag **MAP_ANONYMOUS** along with flag **MAP_SHARED** creates a ***shared memory segment***

#include <sys/mman.h>

Linux implementation (II)

```
#include <sys/mman.h>
int msync(void *addr,
         size_t length,
         int flags);
```

- Flushes back to disk all changes made in main memory from address *addr* to address
  addr + *length* – 1
- Many flag options

Discussion

- Solution requires very large address spaces
- Most programs will continue to access files through calls to read() and write()
  - Function calls instead of system calls
  - ***NO context switches!***

A major problem

- Much harder to emulate the UNIX consistency model in a **distributed file system**
  - ***How can we have atomic writes?***
  - Not a problem for laxer consistency model (close-to-open consistency)