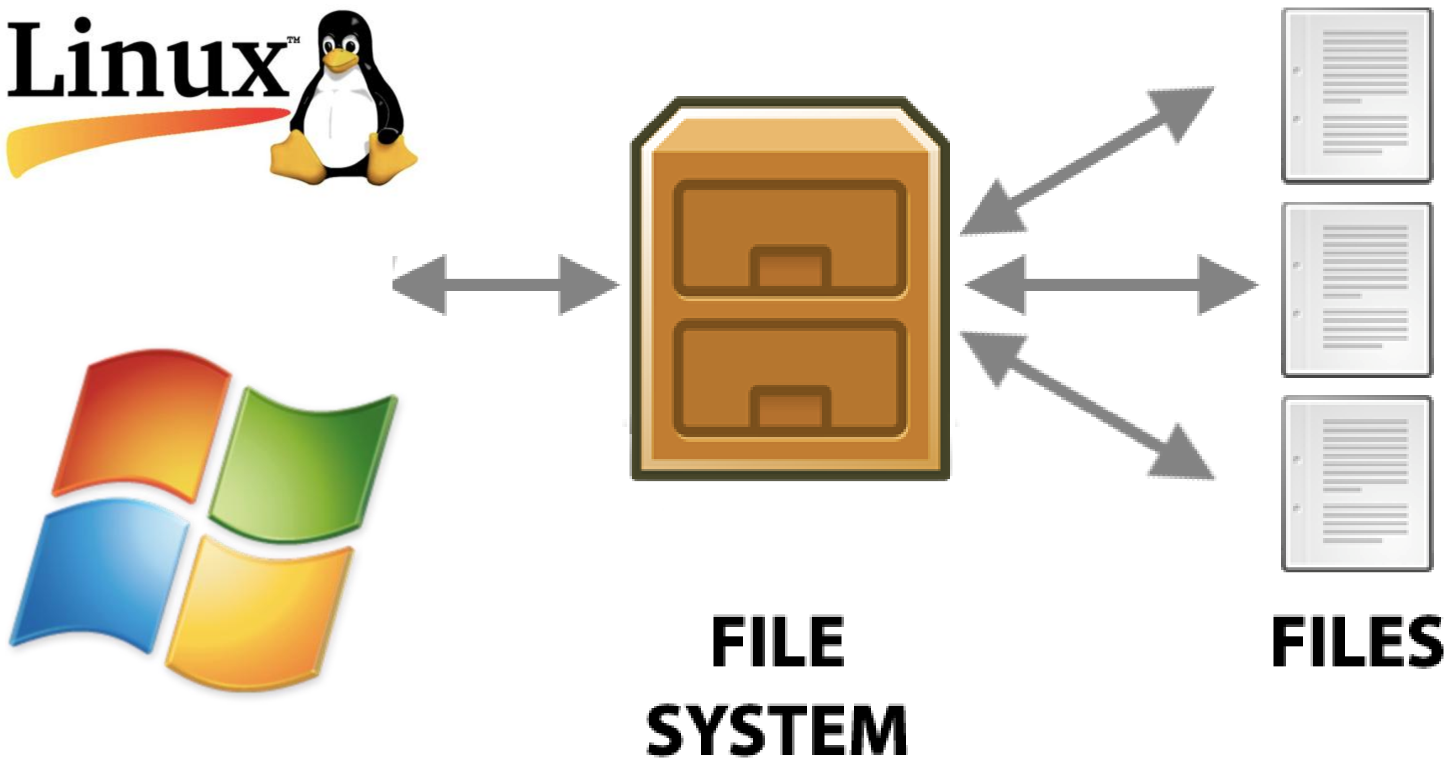
File Systems



File Allocation Example - (Linked) Block Chaining

The File Allocation Table

Topics

1. FAT
2. Defragmentation
3. File Descriptor
4. Buffers

Recall from last lecture

…We looked at **3 different types** of file allocation methods which were

1. Contiguous
2. Non-contiguous Linked – Block Chaining
3. Non-contiguous Indexed – Block Mapping

Example Block Approach in Practice

- The **File Allocation Table** (FAT)
  - Is an old Linked - Block Chaining approach (superceded by NTFS)
  - **Recall that** links from one block to another are saved in a separate ‘map’ table
  - **F**ile **A**llocation **T**able is the mechanism used
    - e.g. FAT16, FAT32, ExFAT

- When a file is requested
  - The file manager must know the starting point of where the file is on the disk.
  - This information is maintained by the File Allocation Table

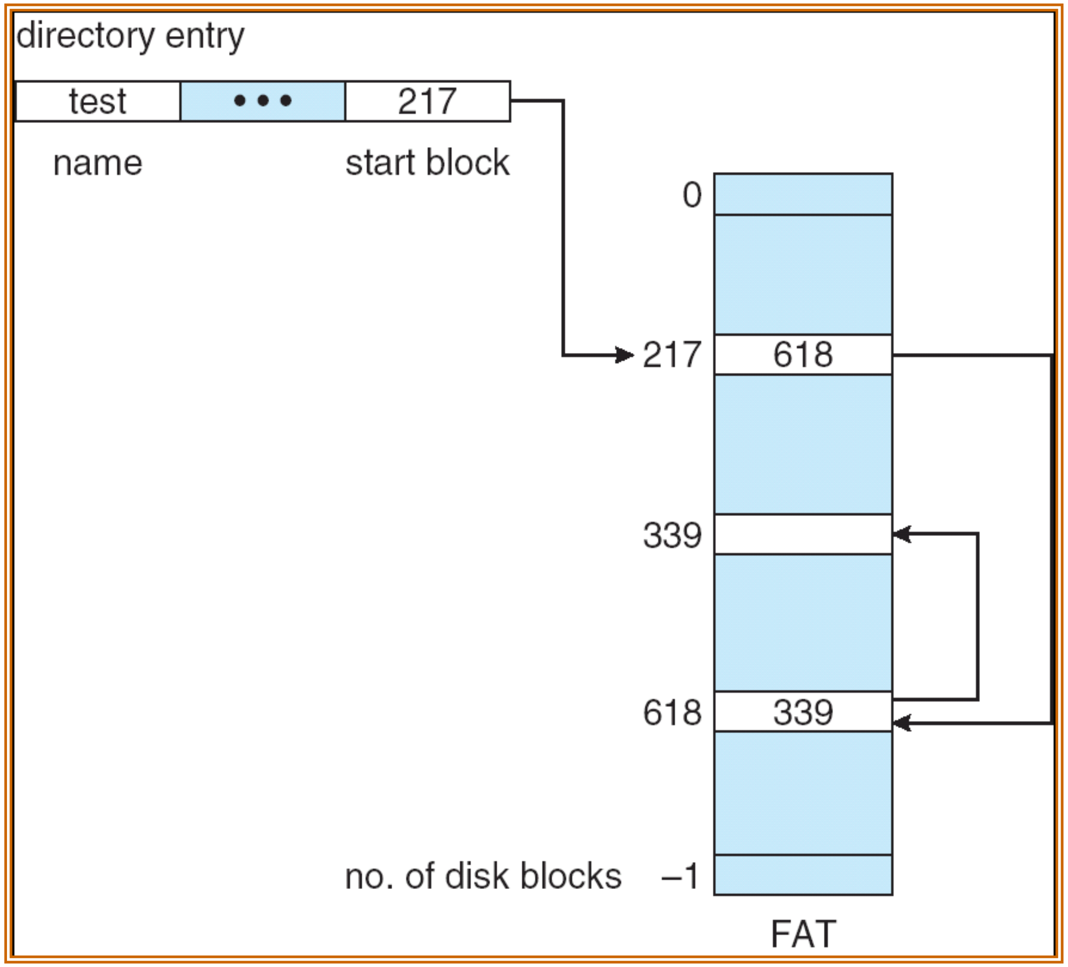
File Allocation Table (FAT), continued

- No other file is allowed to reside at the location where the FAT is

- The file reference table is copied from secondary memory (Disk) to primary memory (DRAM) to avoid excessive secondary storage I/O
  - This is done at boot time

- Damage to the FAT can result in lost, missing or inconsistent file information throughout the file system

File-Allocation Table, example



File Allocation table, continued

- **File Allocation Table** (**FAT**) is widely used on many computer systems

- FAT file systems are commonly found on flash memory cards, digital cameras, and many other portable devices because of their relative simplicity.

- The FAT file system is relatively straightforward technically and is supported by virtually all existing operating systems for personal computers

http://en.wikipedia.org/wiki/File_Allocation_Table

Disk Defragmentation

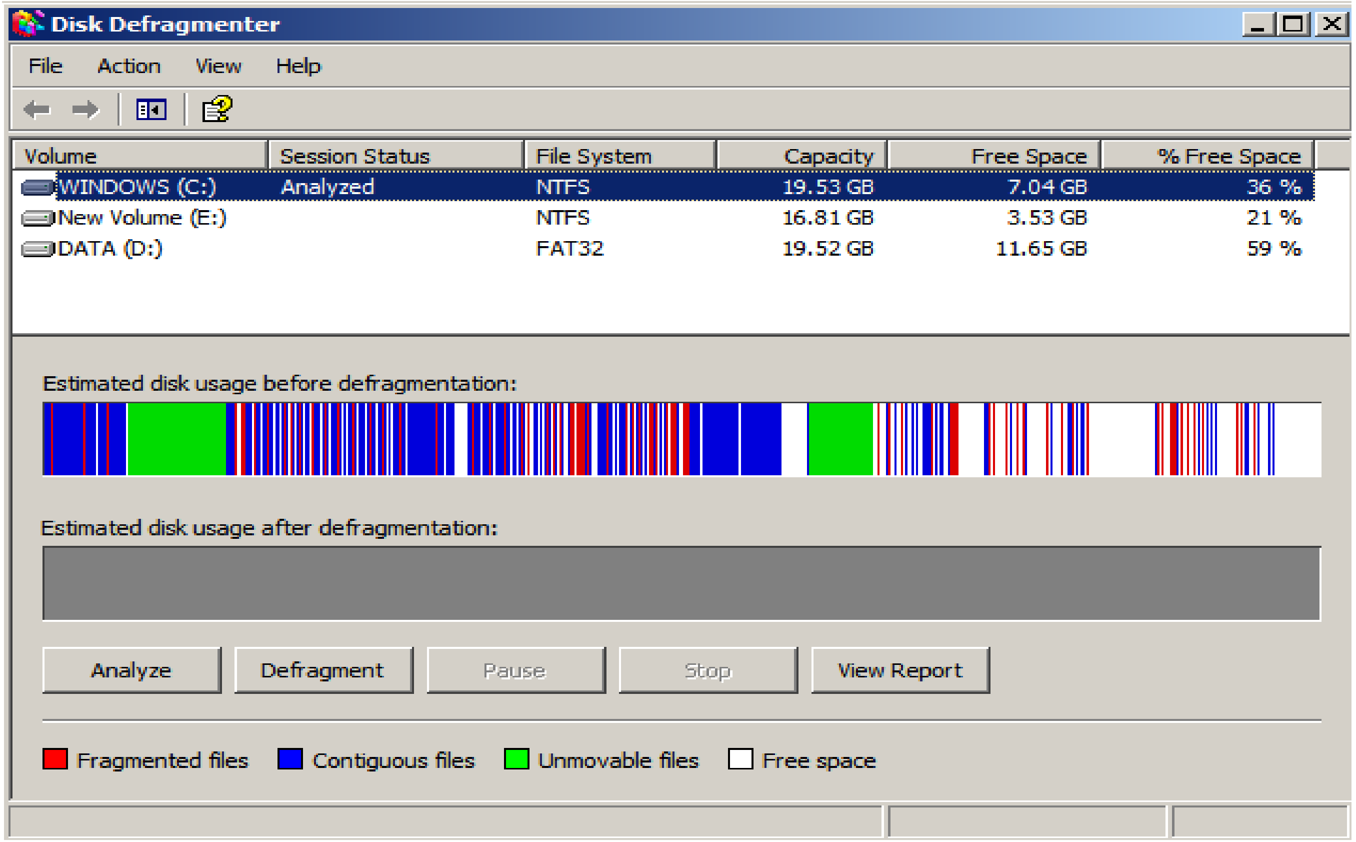
Disk Defragmentation

- Operating system tries to place programs on the disk in continuous sectors
  - As files are deleted, there are gaps between sectors
  - Files end up scattered all over the disk

- Disk defragmentor re-organizes the disk so that files are again consecutive for faster access

Disk Defragmentation

- The Windows defragmentation utility program



Disk Defragmenter

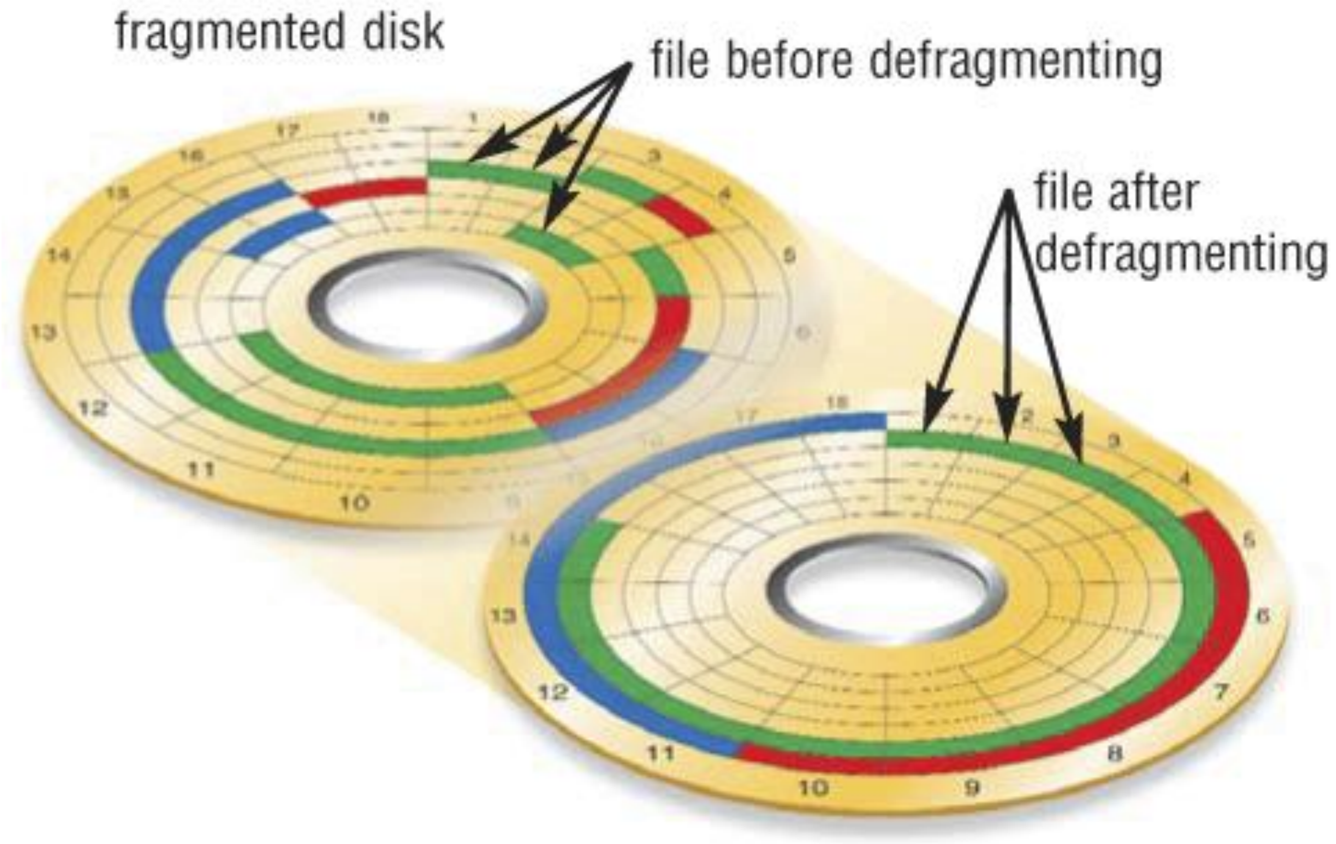
Volume	Session Status	File System	Capacity	Free Space	% Free Space
WINDOWS (C:)	Analyzed	NTFS	19.53 GB	7.04 GB	36 %
New Volume (E:)		NTFS	16.81 GB	3.53 GB	21 %
DATA (D:)		FAT32	19.52 GB	11.65 GB	59 %

Estimated disk usage before defragmentation:

Estimated disk usage after defragmentation:

🟥 Fragmented files 🟦 Contiguous files 🟩 Unmovable files ⬜ Free space

Disk Defragmentation



Modern File Systems do this automatically now

The File Descriptor/Handle

Introduction – File Descriptor

- Files need to be accessed both by the Operating System to run programs and by applications that want to read/write data to them.
  - **File Descriptors** are used to provide this functionality

- Many processes may be accessing the same file so need a means to support this

The File Descriptor

- Every time a process wishes to access a file,
  it must do so using a *file descriptor* (FD).

- The FD is a data structure created by the file system used to identify a file which has been opened for I/O

- A *new* FD is created each time a file is opened and retired when the file closes

- If the file \exam-results.html" is
  opened by 5 different web browsers:
  - There will be 5 File Descriptors
  - One for the private use of each process.

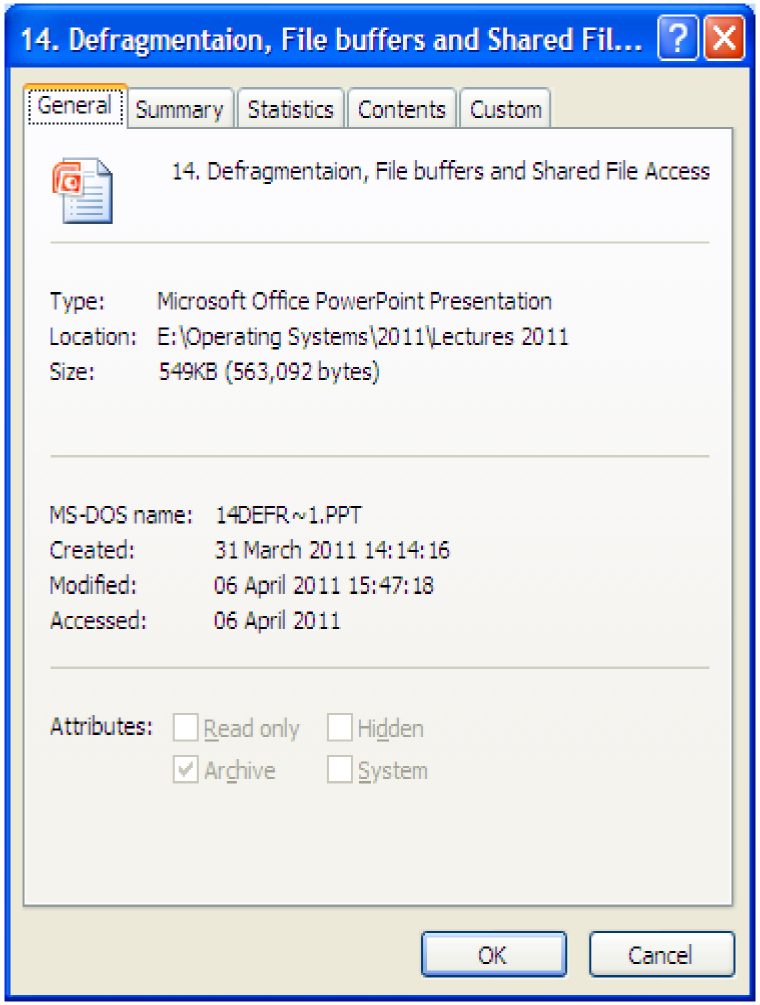
The File Descriptor, continued

- The **File Descriptor** (**FD**) typically contains the following fields:
  - **File Pointer (FP) position**:
    - Position in the file that this descriptor currently pointing
  - **Read/Write Buffers**
    - memory arrays that are temporarily used to hold data while it is being transferred from an input device (e.g. keyboard), or just before it is sent to an output device

The Master File Descriptor (MFD), in Windows

- There is exactly one MFD for each file in the file system

- The MFD maintains the following:
  - **External name**: character string, used at the Command Line Interface
  - **Current state**: open, closed etc
  - **Sharable:** multiple process access rules
  - **Owner:** User name and ID of user who created the file
  - **User**: List of users currently with the file open (using File Descriptors)


14. Defragmentaion, File buffers and Shared Fil...

General | Summary | Statistics | Contents | Custom

14. Defragmentaion, File buffers and Shared File Access

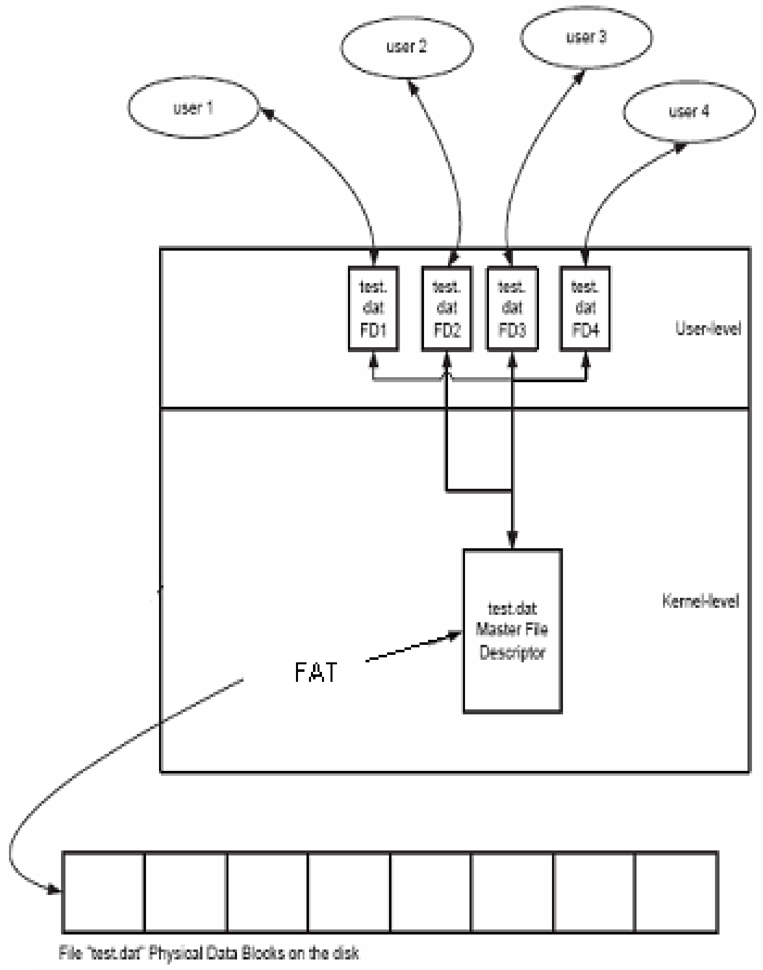
Type:	Microsoft Office PowerPoint Presentation
Location:	E:\Operating Systems\2011\Lectures 2011
Size:	549KB (563,092 bytes)
MS-DOS name:	14DEFR~1.PPT
Created:	31 March 2011 14:14:16
Modified:	06 April 2011 15:47:18
Accessed:	06 April 2011

Attributes: ☐ Read only ☐ Hidden ✓ Archive ☐ System

EXAMPLE

More than 1 User opens the Same File

- F4Four different users (1 -4) are accessing the same file: **\test.dat**
- As can be seen, each user process is allocated a private File Descriptor (FD 1 - 4).
- There is only one master file descriptor for \test.dat which acts as a central structure managing access to the file
- Finally, the FAT acts as the structure to point to the physical on-disk properties of the file.



How can 4 different users be reading from 4 different places in the file and not interfere with each other?

Parallel access

- The FD’s all have separate pointers for the read and write buffers
- Therefore simple for all 4 processes to read at same time

- Is it so simple for writing?
- Remember race conditions and Mutexes

A closer look at File Buffers

A buffer is a region of memory that is temporarily used to hold data

while it is being transferred from an input device (e.g. keyboard),

or just before it is sent to an output device (e.g. printer)

File Buffers

- When data is read or written to a file, it is not written directly to the file, but is written to a memory structure first

- This memory structure is called a ***buffer***

- There are three kinds of buffers:
  - **Read Buffer**
  - **Write Buffer**
  - **Read-Write Buffer**

File Buffers, continued

- All input/output (I/O) operations on files access buffers.
  - Because buffers are in-memory, access to them is significantly faster than directly accessing the underlying disk file.

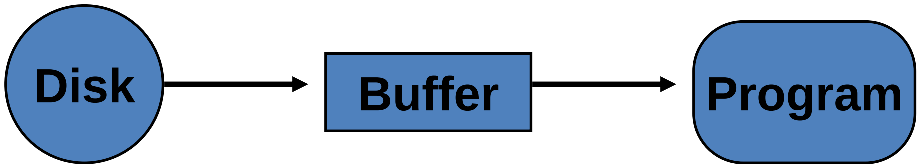
- Buffers are synchronised to disk contents by the kernel, therefore the user-level process does not need to worry about complex low-level disk access
  - I/O operations occur when the OS can schedule it. This means it will not always happen when you press save, for example
  - One of the reasons you can not pull a USB thumb drive out without requesting its removal
  - Note: one can setup the File System to sync immediately if your design requires that.

File Buffer Behaviour

- When the RO buffer is full:
  - The I/O subsystem is blocked from writing to the buffer until the user process has read (and consumed) data from the buffer
  - This is known as the ***producer-consumer* problem**
- Buffers are finite in size:
  - When a write buffer nears full, the I/O subsystem must *push* it to the disk, and then the user process can start writing to the buffer once more.
  - This is also known as **buffer *synchronisation*.**
- Buffers are *circular* :
  - when the end of the buffer is reached, the user process just continues to write to the front of the buffer
  - The I/O subsystem must make sure the contents of the buffer are pushed to disk.

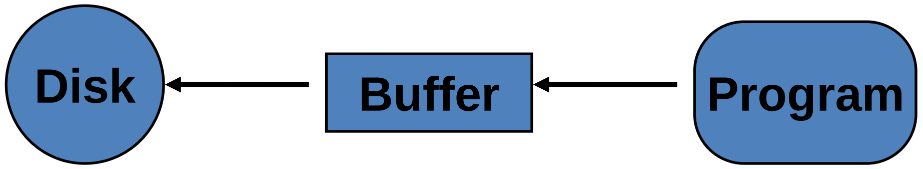
Operation of File Buffers

1. **Reading**
   - File block containing file pointer is copied into file buffer from secondary storage.
   - User may only have asked to read 10 bytes but the whole physical secondary storage memory block e.g. 4k is transferred. Why is this is a good thing?
   - Data read from buffer until pointer goes beyond the end
   - Next block is copied into buffer

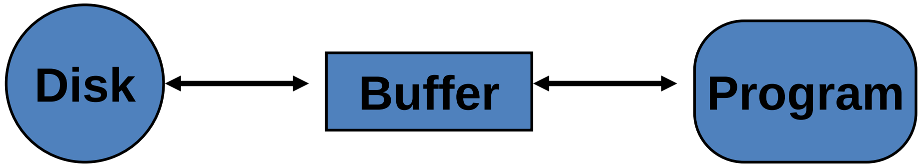


Operation of File Buffers, continued

2. **Writing (Append)**
   - Data is transferred to file buffer until full
   - Data from buffer is copied to secondary storage



3. **Modifying (Read/Write)**
   - File block containing file pointer is copied into file buffer from secondary storage
   - Data is copied into the file buffer, replacing some or all of the original contents
   - When the file pointer is moved outside the range of the file buffer, the contents of the buffer are copied back to secondary storage



Why are file buffers important

- Improves I/O Efficiency

  File buffering allows the operating system to group multiple small I/O operations into a single larger one. This reduces the number of direct disk accesses, which are typically slow and resource intensive. Improves overall system performance and responsiveness, especially during frequent read/write operations.

- Reduces CPU Waiting Time

  Without buffering, processes must wait for each I/O operation to complete. With buffering, a process can continue while the OS handles the actual disk operation in the background. Thus, enhances CPU utilization and multitasking by minimizing process idle time.

Why are file buffers important

- Manages Bursty I/O

  Buffering helps handle situations where data is generated or consumed in sudden bursts. The buffer absorbs this burst, preventing data loss or system slowdown. This helps maintaining system stability and avoiding crashes or delays during heavy I/O.

- Supports Asynchronous I/O

  File buffering is essential for asynchronous input/output, where processes do not block while waiting for I/O to finish. This enables better user experience in applications like video streaming or downloading files.