UNIVERSITY OF PORTSMOUTH

# Operating Systems and Internetworking

## OSI – M30233 (OS Theme)

Week 7- File Systems


*Dr Tamer Elboghdadly*

# Plan

- Files and file operations
- Directories and directory hierarchies
- File systems and their implementation, including case studies:
  - FAT
  - Ext4
  - NTFS

# Files

- The *file* is a central abstraction in most OSs.
- Logically, a file is a *named* unit of storage that exists *persistently*, from the time it is *created* to the time it is *destroyed*.
- Generally, *file content* may be *written*, *read* or *updated*.
- *File size* varies over lifetime of file, as data is added (or removed).
- Files are manipulated, ultimately, by a set of *primitive operations*, usually implemented as OS *system calls*.

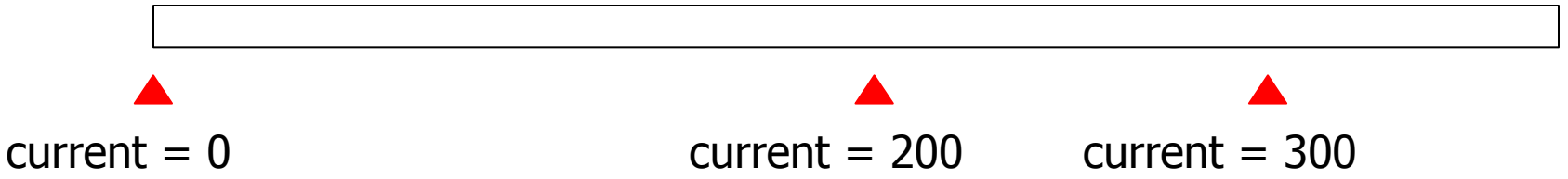
# Primitive Operations on Files (1)

- *Create* named file
  - Write various data *describing* new file to disk. Usually file is empty (no content) when created.
- *Delete* named file
  - Logically, delete content and all data describing file from disk (physically, *data may remain intact*).
- *Open* named file
  - Fetch data describing file from disk to memory, prior to reading or writing.
- *Close*
  - Purge those structures from memory.

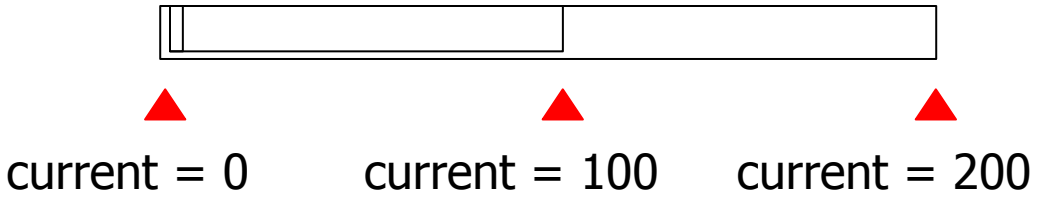
# Primitive Operations on Files (2)

- *Read* from open file
  - Read some bytes of data – usually from *current position* in file.
  - After open, current position is start of file. Read increments this by number of bytes read.
- *Write* to open file
  - Write some bytes of data – usually starting at “current position” (incremented as above).
  - If current position is end of file (commonly it is), file grows accordingly.
- *Seek* in open file
  - Move “current position” to specified location in file.

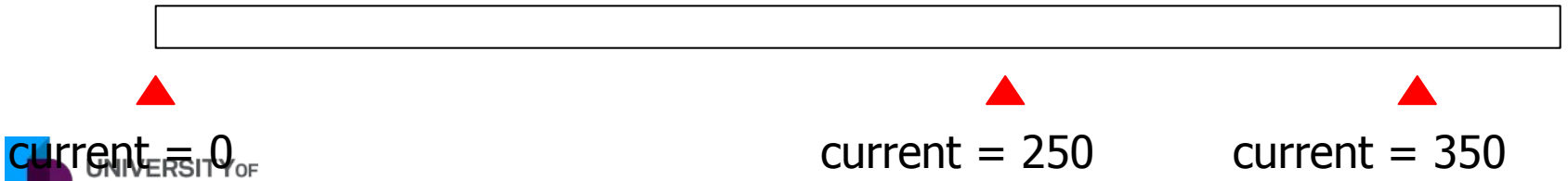
# “Current File Position”




1. **open**
2. **read 200 bytes**
3. **read 100 bytes**




1. **create**
2. **write 100 bytes**
3. **write 100 bytes**




1. **open**
2. **seek 250**
3. **write 100 bytes**

# Primitive Operations on Files (3)

- *Get attributes* of file
  - Various kinds of “metadata”, such as last modification data
- *Set attributes* of file
- *Rename* a file

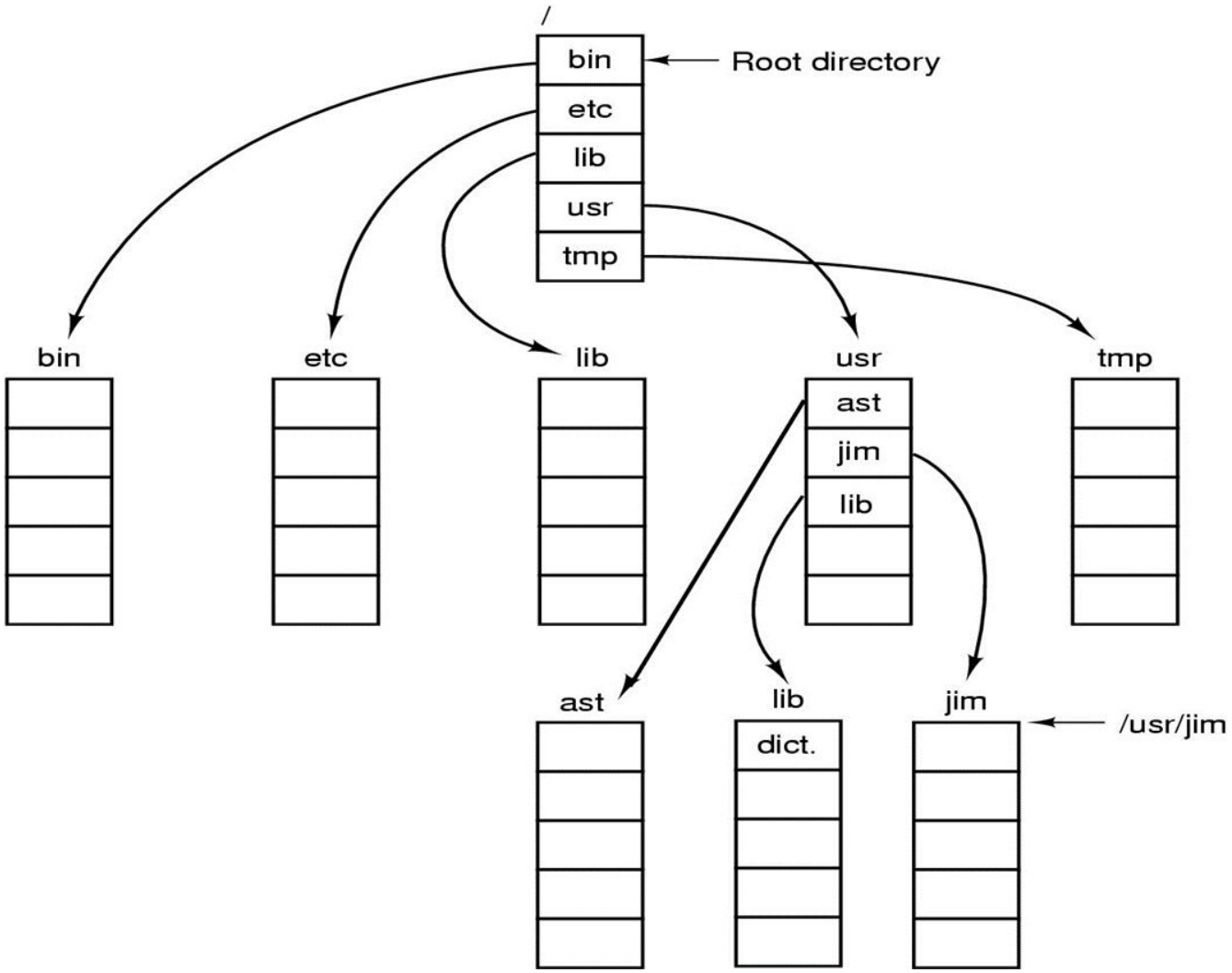
# File Attributes - Metadata

- Examples of file attributes:
  - *Type* - needed for systems that support different file types.
  - *Size* - current file size.
  - *Protection* - controls who can do reading, writing, executing.
  - *Time*, *date*, and *user identification* - data for protection, security, and usage monitoring.
  - *Location* - pointers to file *content* location.
- Exactly where this *metadata* is stored depends on the type of file system.
- It is *not* considered part of file *content*.

# Directories

- A *directory* is a special type of file that contains a list of names of some other files, together with “references” to those files.
  - Entries in directories are references to ordinary files, or to *other directories*
  - Referenced files or directories are considered to be “*contained in*” the directory.
  - Directory entries are considered *child directories*.

# Tree Structured Directories†



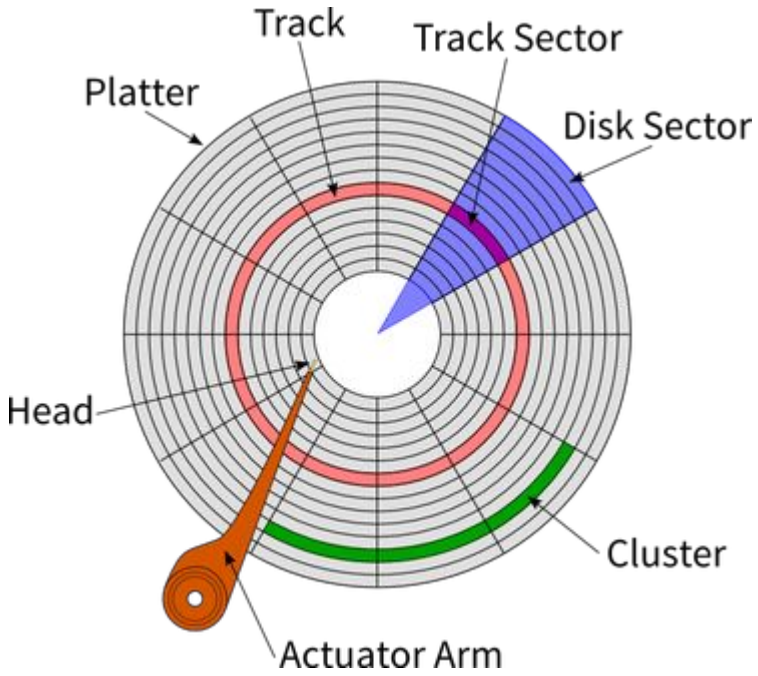

†Tannenbaum, MOS, Fig 4-8

# The Root Directory

- Any file system has a *root directory*, called just "**/**" in UNIX-like OSs, and "\" in Windows.
- Note that neither files nor directories contain *absolute path names* – not even for themselves.
- To locate **/usr/jim/a.txt**:
  - first go to the root directory, /, and look for location of **usr**;
  - hence go to directory **usr** and look for location of **jim**;
  - hence go to directory **jim** look for location of **a.txt**.

# Blocks, Clusters

- Most file systems store *file content* in "*units of storage";* on an HDD a unit may consist of *several consecutive disk sectors*.
- In UNIX-derived file systems, these units are usually called *blocks*; in Windows file systems they are usually called *clusters*.
- In any case, *block size* is typically some multiple of physical sector size (e.g. 1KB, 2KB, 4KB, …)
- Each file is allocated a *whole number* of blocks to store its content.



# Types of File System

- File systems of modern operating systems look quite similar at the user level, but each OS usually supports several different *types* of file system
- Different file system implementations, according to:
  - Characteristics of storage device,
  - Which operating system *wrote* the file system,
  - Legacy file systems from earlier versions of an operating system,
  - etc

# Example File Systems

- *FAT* family from Microsoft
  - Legacy DOS/Windows, still widely used on smaller devices.
- *New Technology File System* (NTFS)
  - Default in modern Windows – much more flexible and reliable than FAT
- *ExtX* family
  - Default in Linux based systems.
- *HFS+* (Hierarchical File System Plus)
  - Used in MAC OS

# FAT

# Real File Systems 1: FAT

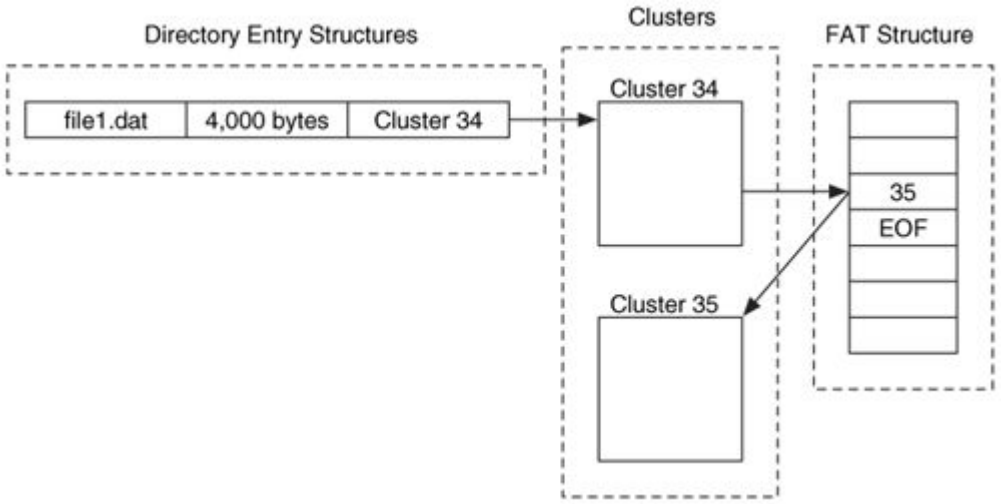
- *File Allocation Table* (FAT) was the file system of MS-DOS, circa 1980
  - May have originally been designed by Bill Gates, circa 1976 - used in Microsoft Basic.†
- Versions of FAT were primary file system of MS Windows through *Millennium Edition*.
  - Replaced by *New Technology File System* (NTFS) on NT branch of Windows, which became mainstream with *Windows 2000*.
- Still widely used on small storage devices, and recognized by essentially all modern OSs.



†Source: Wikipedia

# Directories

- As in most file systems, directories are “just” files consisting of a series of *directory entries*.
- In FAT, a directory entry is just 32 bytes long and contains *file name*, file *metadata* (see slide 8), and the id of the *first cluster only* of file content.

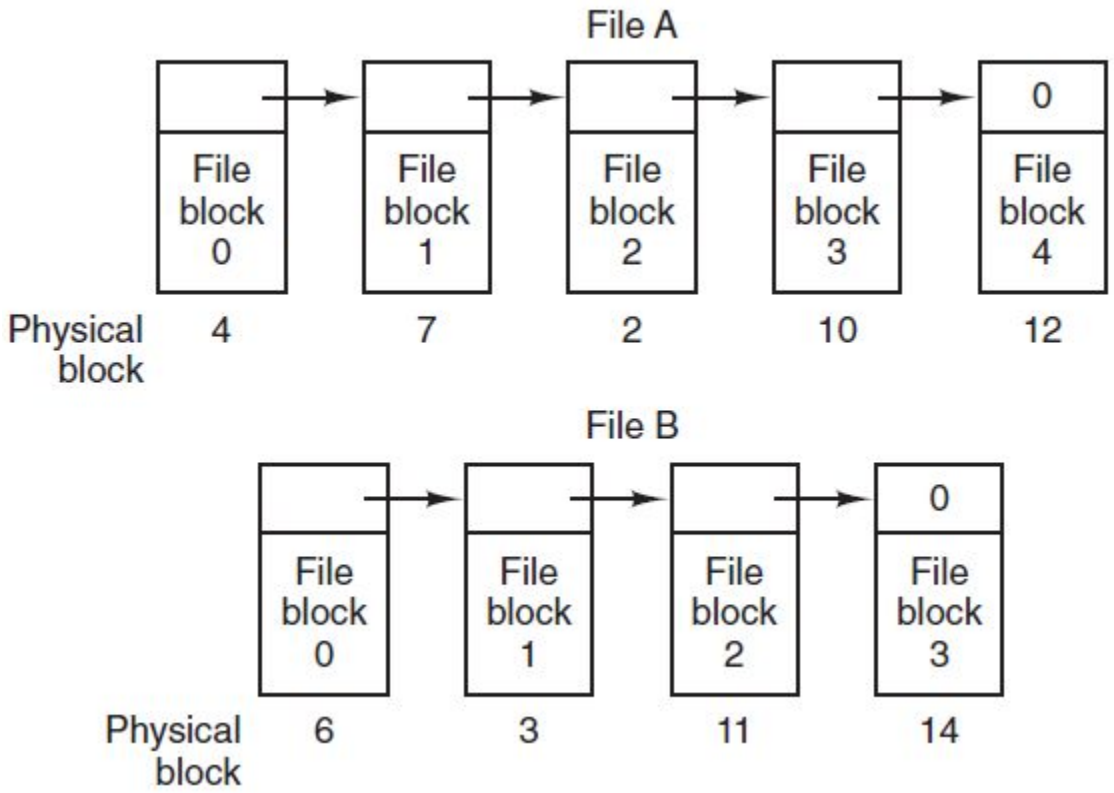



†Carrier, FSFA, Fig 9.1: Relationship between the directory entry structures, clusters, and FAT structure.

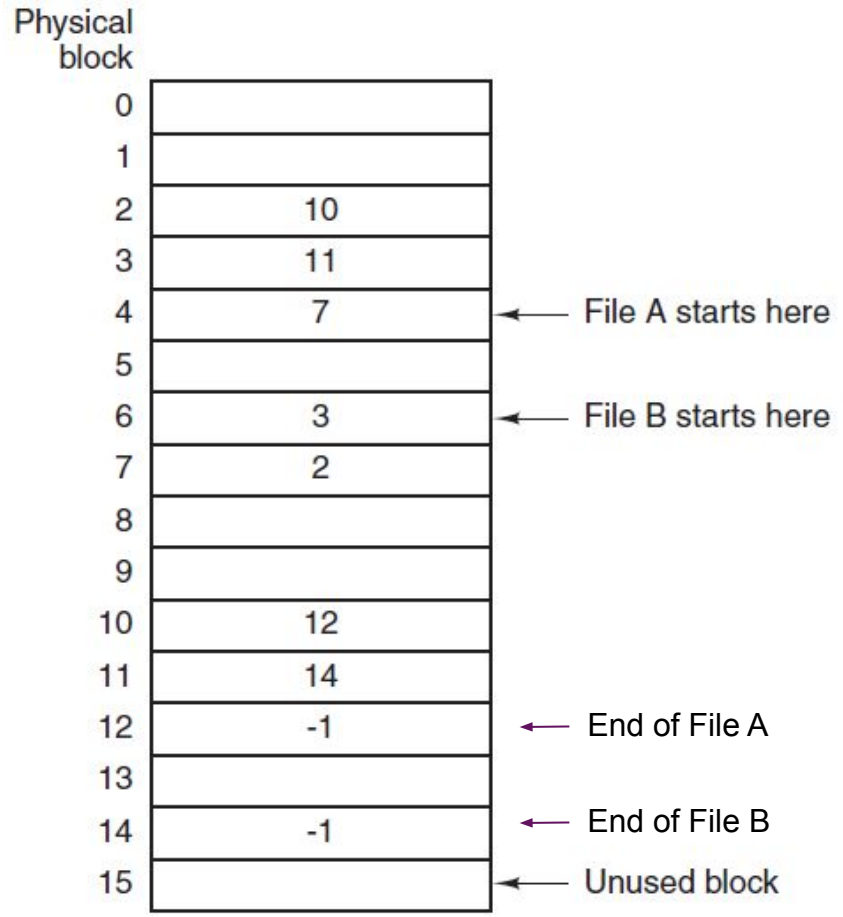
# The File Allocation Table

- As the Directory entry contains id of the first cluster only, all remaining clusters of a file are located through a separate data structure – the eponymous *File Allocation Table*.
- This is an implementation of the *linked list allocation scheme* †, where the links are stored by themselves in a dedicated area of the disk.
  - One FAT entry for every cluster in data area of disk.

# The File Allocation Table




Storing a file as a linked list of disk blocks.

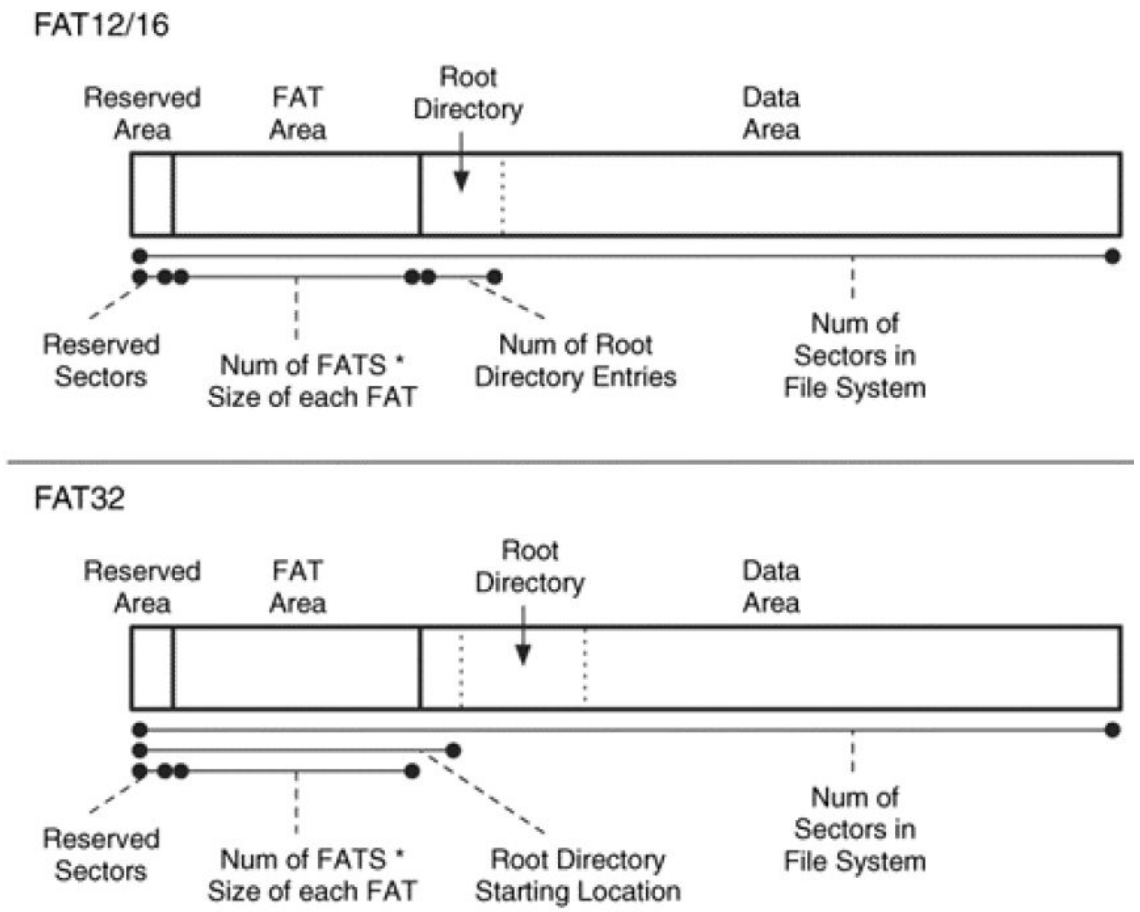



Linked-list allocation using a file-allocation table in main memory.

UNIVERSITY OF PORTSMOUTH

†Tannenbaum, MOS, Fig 4-11 and 4-12

# Layout of FAT File System†

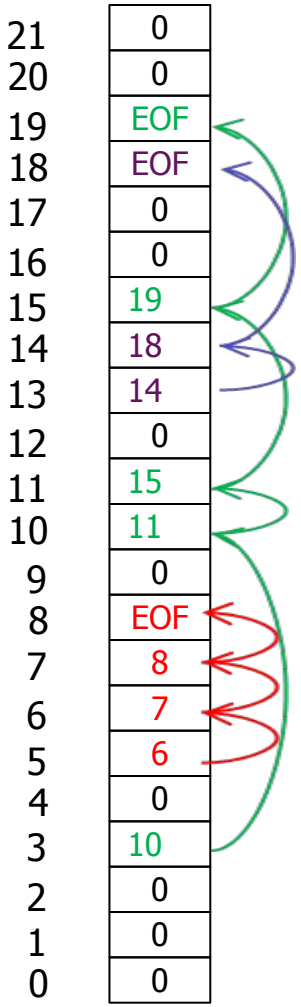



†Carrier, FSFA, Fig 9.3

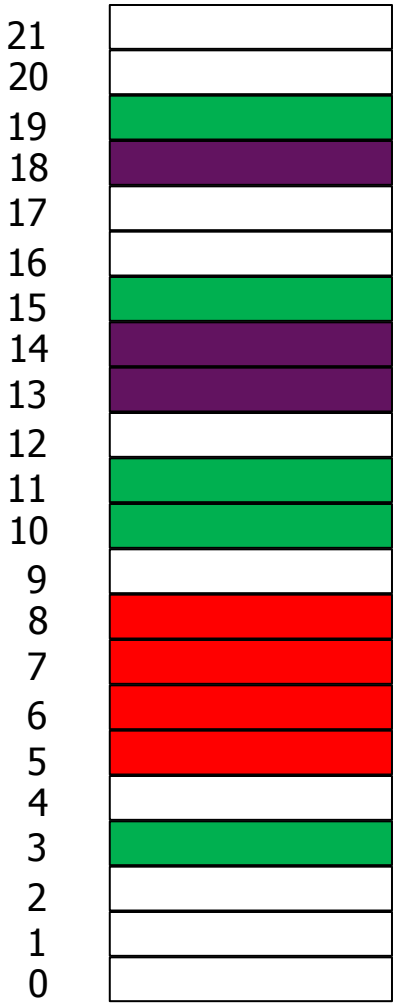
# FAT entries

- The FAT area contains one FAT entry - consisting of just a link - for each cluster in the data area.
- The FAT entries (plus associated clusters) implement data structures called *cluster chains* (really *linked lists*), that represent file content.
- First cluster of file contained in its directory entry - in the following example, file 1 starts at cluster 3, file 2 at cluster 5, file 3 at cluster 13.

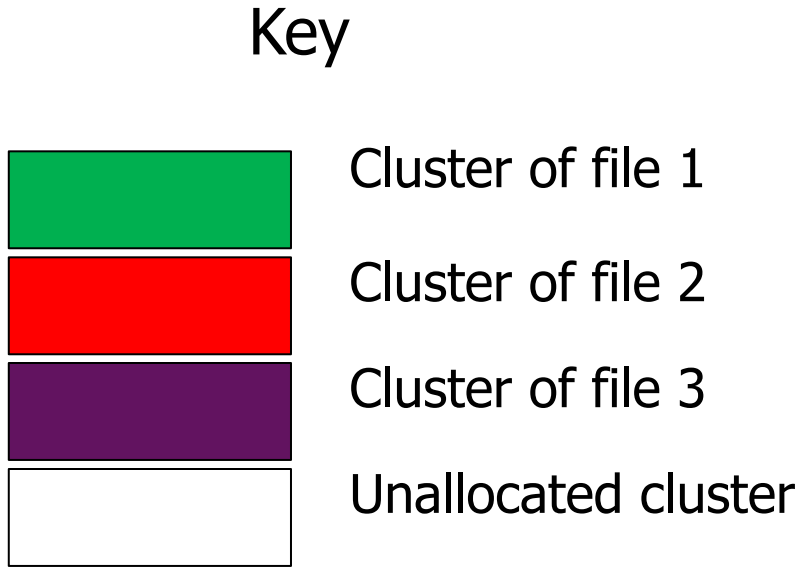
# Relation of FAT and Cluster Table




FAT




Cluster Table (Data Area)



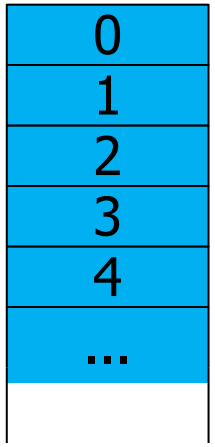
# UNIX FILE SYSTEMS

# Real File Systems 2: Ext Family

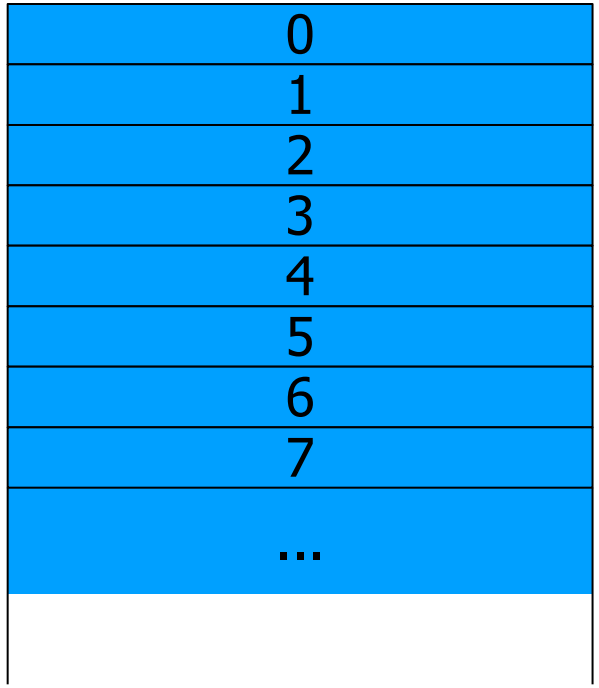
- Extended file system (Ext), Used by various UNIX-like operating systems (e.g Linux).
- For example the current Linux default file system is *Ext4*.
- Everything in Unix is considered to be a file, including physical devices such as DVD-ROMs, USB devices, and floppy drives.
- Allocation in these systems follow an *inode* approach.
  - Any block or inode can be in an *allocated* or *unallocated* state.

# Simplified UNIX File System

Superblock




Inode Table

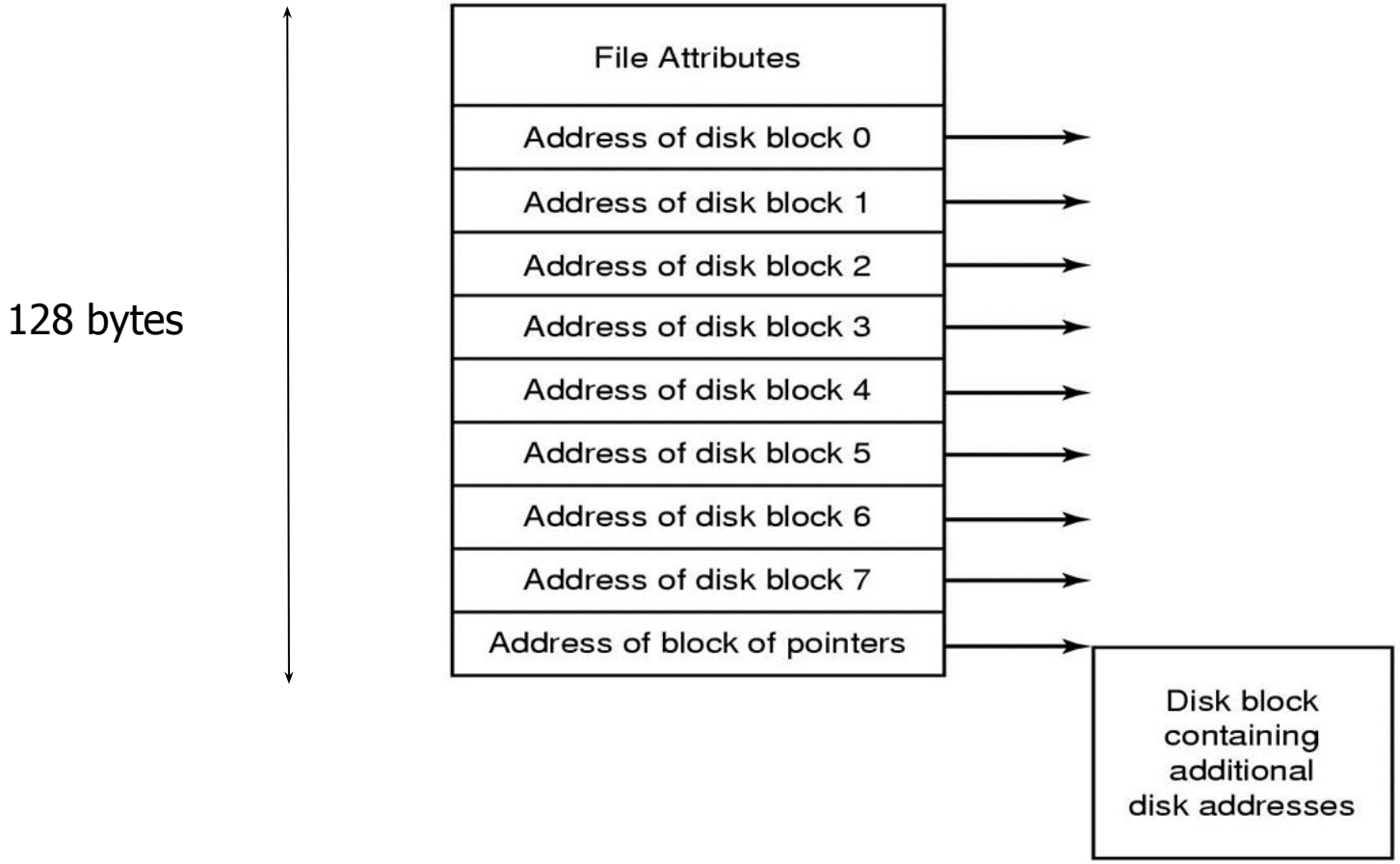



Block Table

# Inode

- Inode or I-node refer to (index node).
- Every file or directory in the system has precisely one *inode*.
- The inode is a small (e.g. 128 byte) data structure containing *metadata* (see slide 8) plus block pointers for content of file.
- Within the implementation of the file system, the *inode number* is the principal means of referring to a file or directory.

# Logical View of Inode†



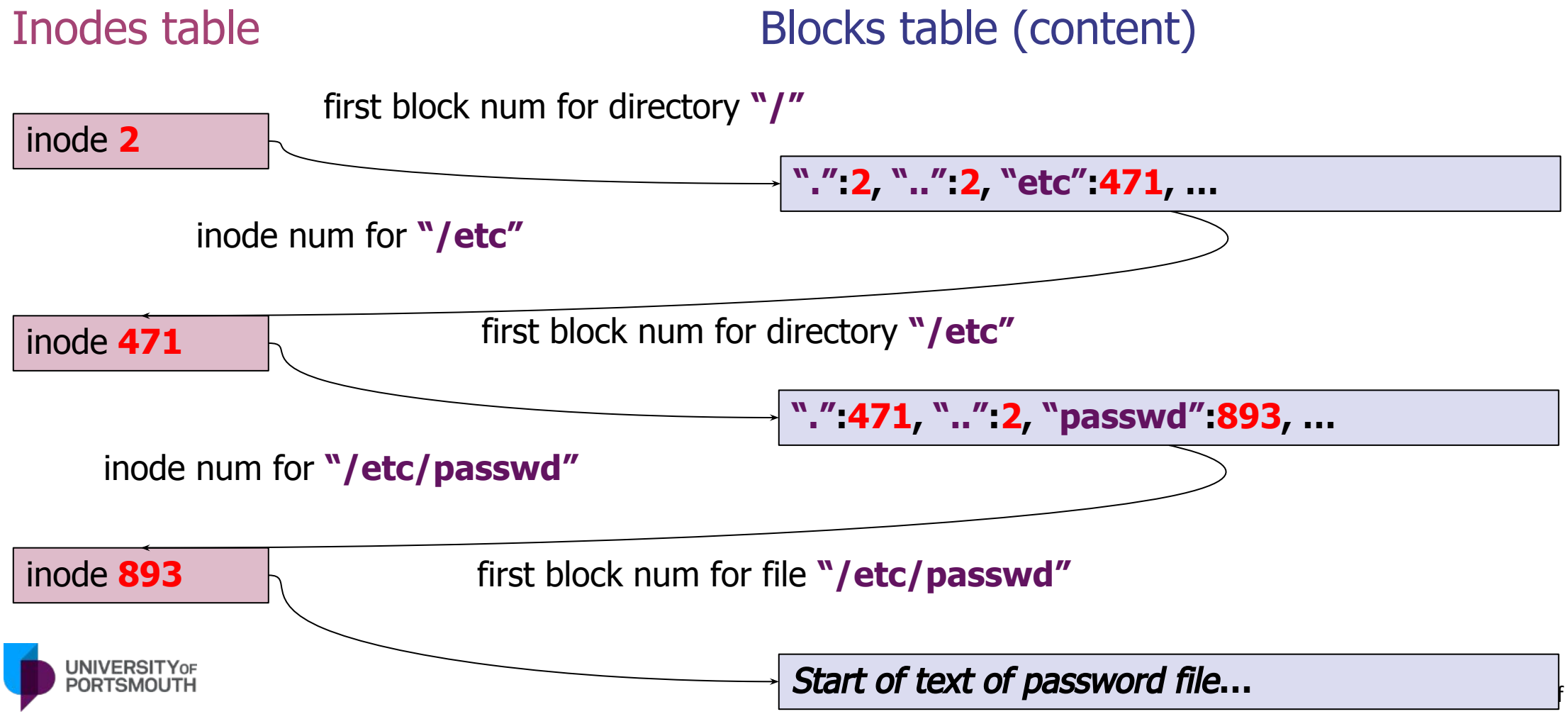

†Tannenbaum, MOS, Fig 4-13

# Directories

- A directory is a file, and thus has its own inode (metadata).
- As usual, this inode references blocks holding the *content* of the directory.
- In the case of a directory, the *content* follows a strict format – it contains a list of names and inode numbers for the files “in” the directory (and essentially nothing else).

# Inodes, Blocks and Directories

c.f. Slide 11 – here look up "/etc/password"



# NTFS

# Real File Systems 3: NTFS

- The *New Technology File System* (NTFS) was introduced by Microsoft for *Windows NT* and successors
  - which include XP, Vista, Windows 7, Windows 8, Windows 10…
- Much more complex file system than FAT, that natively supports long, Unicode file names, security descriptors, encryption, journaling, etc.

# Attributes and Streams

- An NTFS file has an associated *set of attributes*, and value of *each* attribute is a sequence (or *stream*) of bytes.
- Most notably, the value of the *$DATA* attribute holds what we would previously have thought of as the *content* of the file.

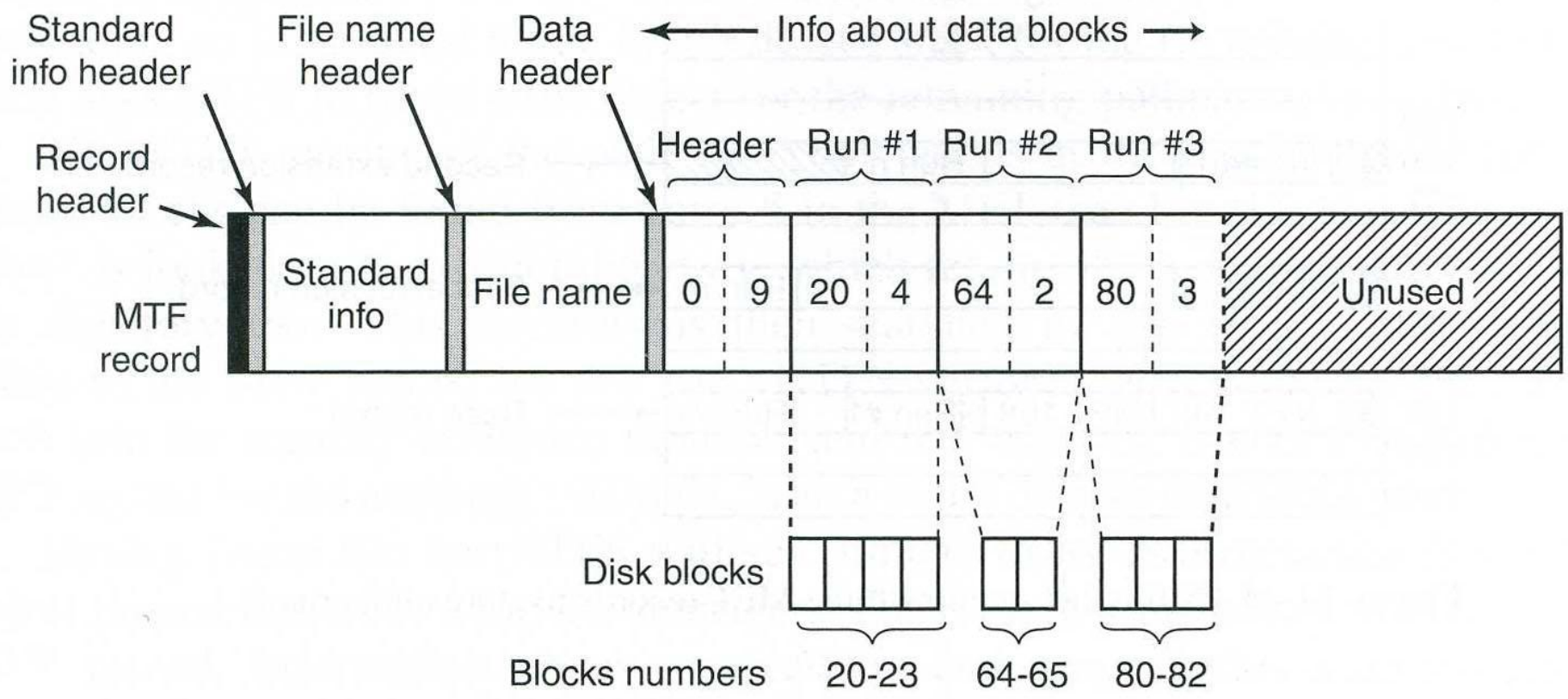
# The Master File Table

- The primary storage of file *metadata* in NTFS is in the *Master File Table* (MFT).
- It contains (at least) one entry (*file record*) describing every file and directory.
  - Roughly, MFT entries are analogous to UNIX inodes.
- Every entry (record) in the MFT has a fixed  size.
  - In principle this is configurable in the boot sector, but normally it is 1KB.

# Resident vs Non-Resident Attributes

- The value of any attribute can be *resident* or *non-resident*.
- Attributes with short, fixed length values will normally be “resident”
  - i.e. the value is stored *in the file record* in the MFT.
- Attributes with large values (including *$DATA* attributes that hold large file content) will normally be non-resident.
  - i.e. value is stored *outside* the MFT, with just *storage locations (*cluster ranges) in the file record.

# File Record with Non-Resident Data†




†Tannenbaum, MOS, Fig 11-41

# Storage of the MFT

- In UNIX, *inodes* were stored in a dedicated table, separate from the *blocks* storage.
- NTFS is different - the *whole* of an NTFS file system is dedicated to *clusters*.
- So where is the metadata actually stored in NTFS?
- Intriguing and slightly paradoxical answer is that the *Master File Table is itself a file*, stored in the file system, like any other file!
  - It is not even stored physically at any specially distinguished location in the file system.

# Summary

- Reprised the basic properties of files and directories in modern file systems.
- Went on to discuss implementation of several real file systems.
- *Next Lecture – Virtual Memory*

# Further Reading

- Andrew S. Tanenbaum, "*Modern Operating Systems*", 4th Edition, Pearson, 2014 (MOS)
  - See chapter 4 for general discussion of file systems, section 10.6 for discussion of the Linux file systems, section 11.8 for discussion of NTFS.
- Brian Carrier, "File System Forensic Analysis", Addison Wesley, 2005 (FSFA)
  - For exhaustive description of FAT and other file systems.

UNIVERSITY OF PORTSMOUTH

# Questions?