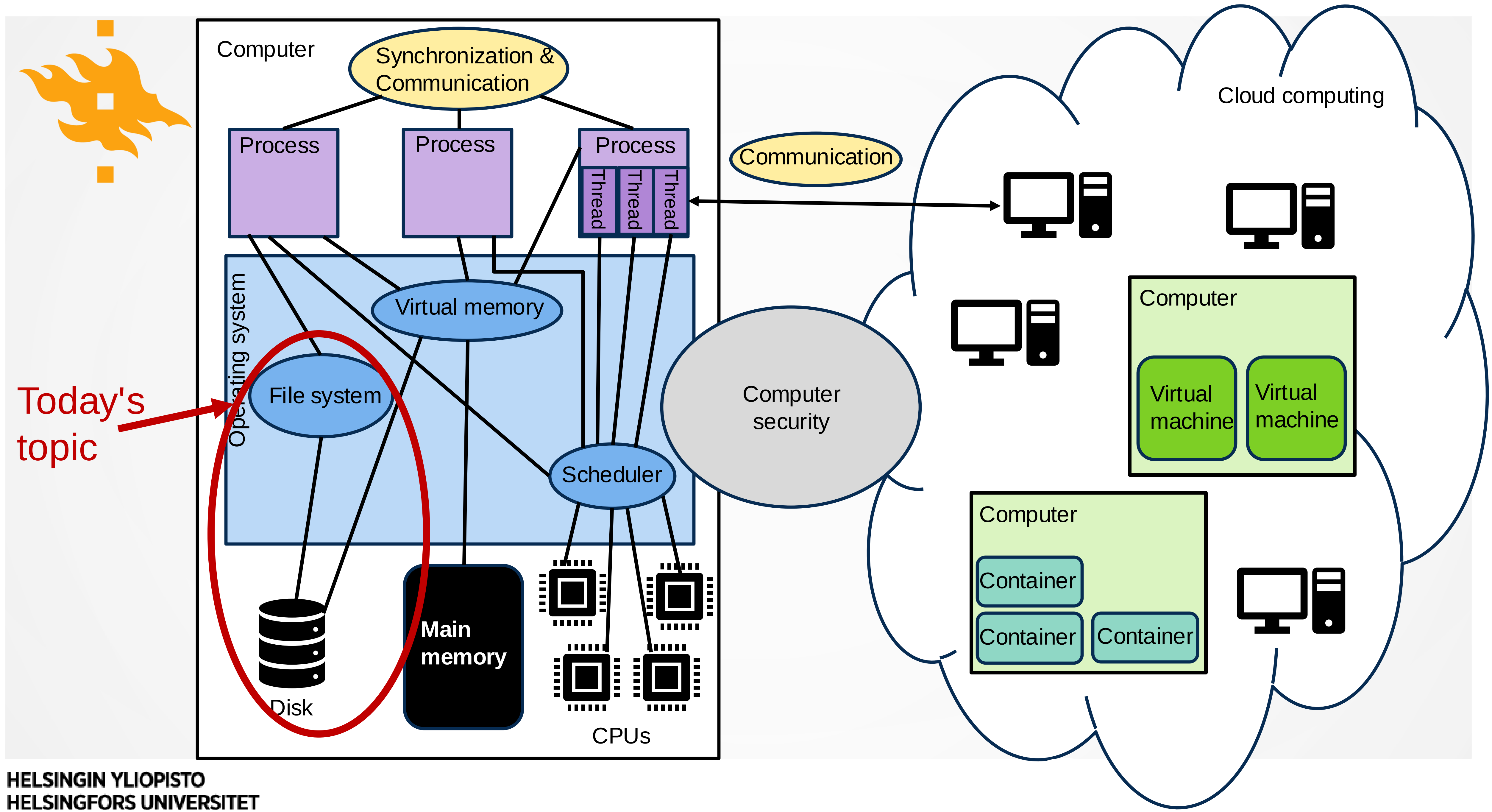
COMPUTING PLATFORMS

I/O and file systems


Computer
Synchronization & Communication
Process
Process
Process
Thread
Thread
Thread
Communication
Operating system
Virtual memory
File system
Scheduler
Computer security
Disk
Main memory
CPUs
Today's topic
Cloud computing
Computer
Virtual machine
Virtual machine
Computer
Container
Container
Container

LEARNING OUTCOMES

- After today's lecture you
  - Know key categories of I/O devices
  - Can describe and compare hard disk drives and SSDs
  - Are able to describe basic concepts of files and file systems
  - Are able to explain the data structures and access methods of a file system

I/O DEVICES

- **External devices** that engage in I/O with computer systems
  - **Human readable**: suitable for communicating with a **human user**, e.g., printers, terminals, video display, keyboard, mouse
  - **Machine readable**: suitable for communicating with **electronic equipment**, e.g., disk drives, USB sticks, sensors, controllers
  - **Communication**: suitable for communicating with **remote devices**, e.g., network interface cards, modems

DIFFERENCES IN I/O DEVICES

- **Data rate:** differences of several orders of magnitude (keyboard vs network card)
- **Application:** use to which a device is put, has an influence on the software and policies in the OS and supporting utilities
- **Complexity of control:** printer requires a relatively simple control interface while disk is much more complex
- **Unit of transfer:** Data may be transferred as a **stream** of bytes or characters (terminal I/O) or in larger **blocks** (disks)
- **Data representation:** Different data encoding schemes are used by different devices, including differences in character encoding and parity code
- **Error conditions:** type of errors, the way errors are reported, consequences of errors, available range of responses differ widely

SYSTEM ARCHITECTURE

- Several buses to connect different I/O devices
  - CPU attached to main memory with **memory bus**
  - High-performance I/O devices (e.g., graphics) connected to **general I/O bus** (e.g., PCI)
  - Slow I/O devices (disk, mouse, keyboard) are connected to **peripheral I/O bus**
- Faster buses are more expensive and can support only limited number of devices

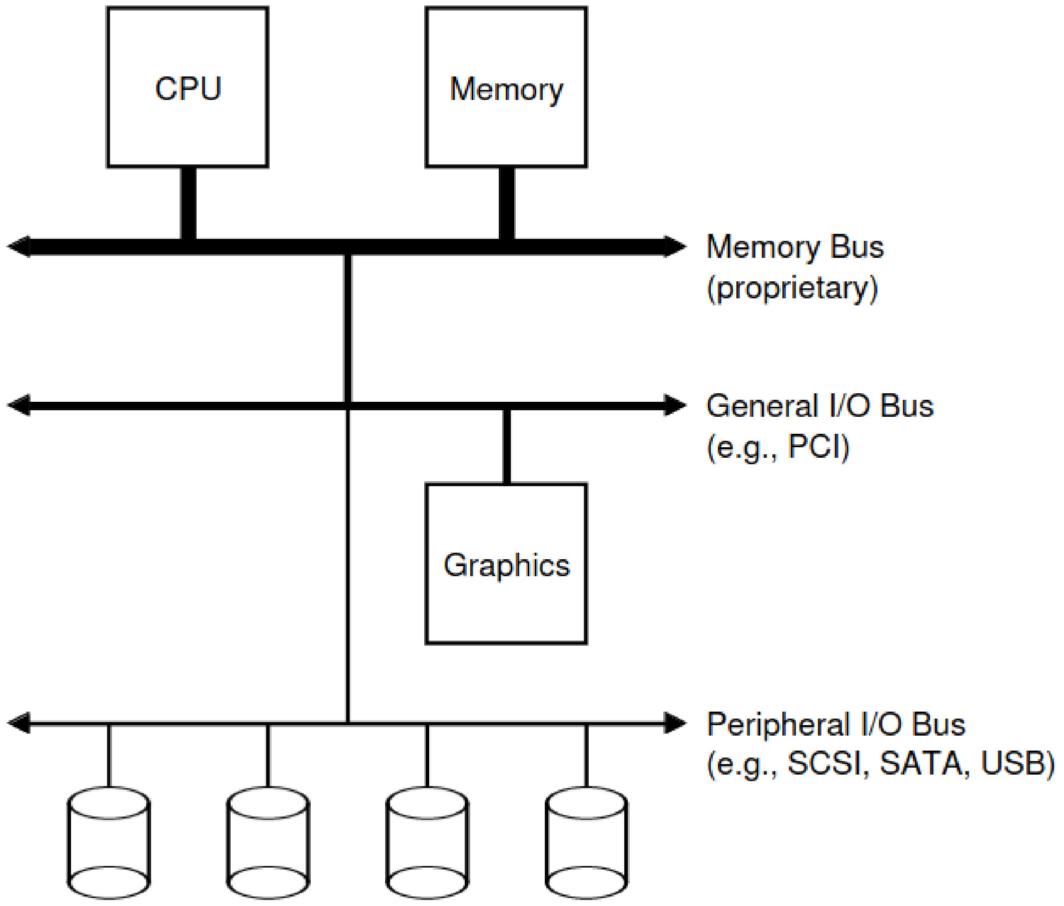


Figure 36.1: **Prototypical System Architecture**

Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

TECHNIQUES FOR I/O

- **Programmed I/O:** *Processor issues an I/O command on behalf of a process to an I/O module*; process then busy waits for the operation to be completed before proceeding
- **Interrupt-driven I/O**: *Processor issues an I/O command on behalf of a process*. If I/O instruction is **nonblocking**, then processor continues to execute instructions from the process that issued I/O command. If I/O command is **blocking**, then next instruction executed is from OS, which puts the current process to blocked state and schedules another process. *Device raises an interrupt when the I/O command is ready*.
- **Direct memory access (DMA):** *DMA module controls the exchange of data between main memory and an I/O module*. Processor sends request for the transfer of a block of data to the DMA module and is interrupted only after the entire block has been transferred.

METHODS OF DEVICE INTERACTION

- **I/O instructions:**
  - The processor has specific I/O instructions which the OS can use to interact with devices
  - I/O instructions are **privileged** (cannot be used by user land programs)
- **Memory-mapped I/O:**
  - Hardware makes device registers available as if they were memory locations
  - OS uses normal load and store instructions to access the memory location

DEVICE DRIVERS

- OS should be largely device-neutral, i.e. most OS subsystems should not be concerned with specifics of a certain device
- **Abstraction**: Isolate the device specific code to a **device driver**

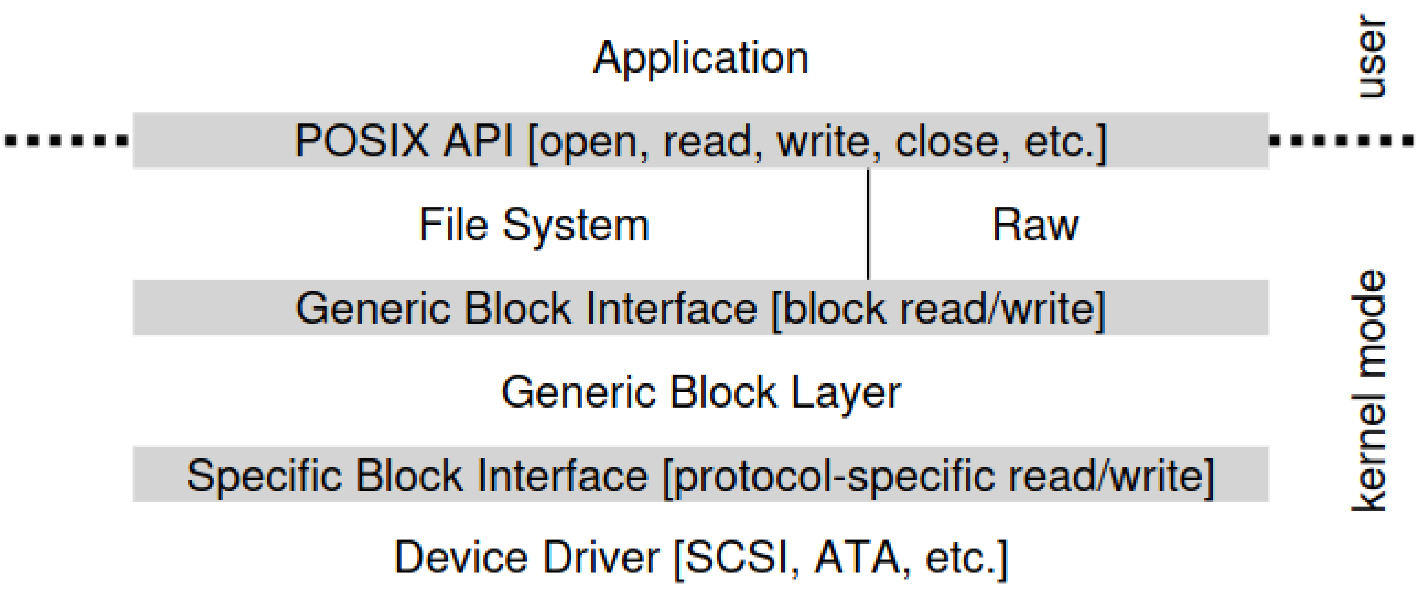
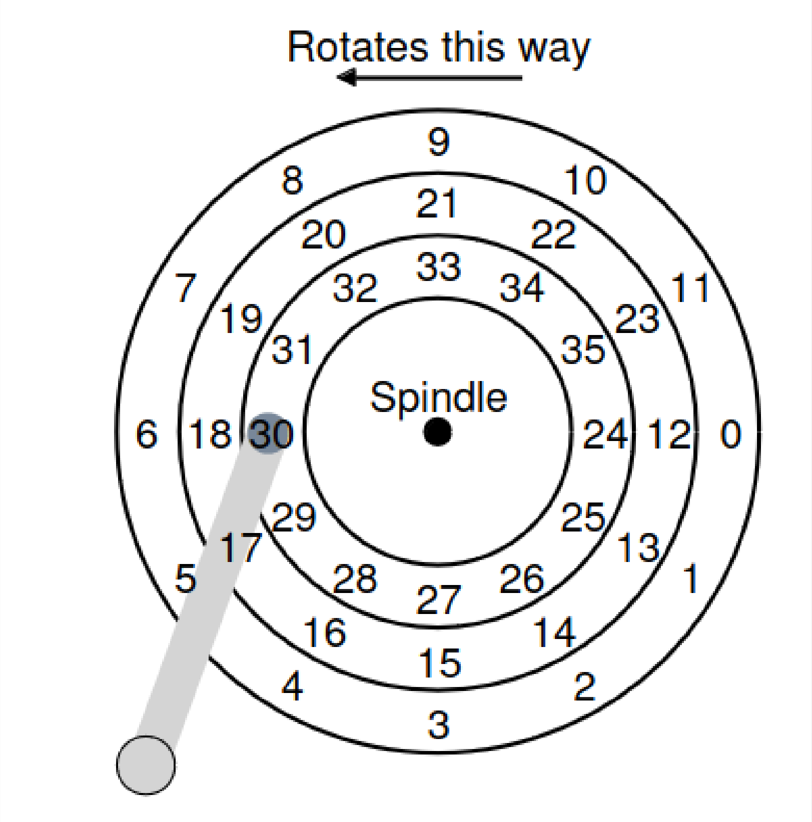


Figure 36.4: **The File System Stack**

Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

EXAMPLE DEVICE: HARD DISK DRIVE

- When a disk drive is operating, **disk rotates at constant speed**
- Disk drive is organized into **tracks** which contain **sectors**
- To read or write the **head** must be **positioned** at the desired **track** and at the beginning of the desired **sector** on that track
- Data is transferred in **blocks** (normally 512 bytes)



A disk with three tracks (Fig. 37.3)

Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

EXAMPLE DEVICE: HARD DISK DRIVE

- Disk performance parameters:
  - **Seek time**: time needed to position the head at the track
  - **Rotational delay**: time it takes for the beginning of the sector to reach the head
  - **Access time** = Seek time + Rotational delay

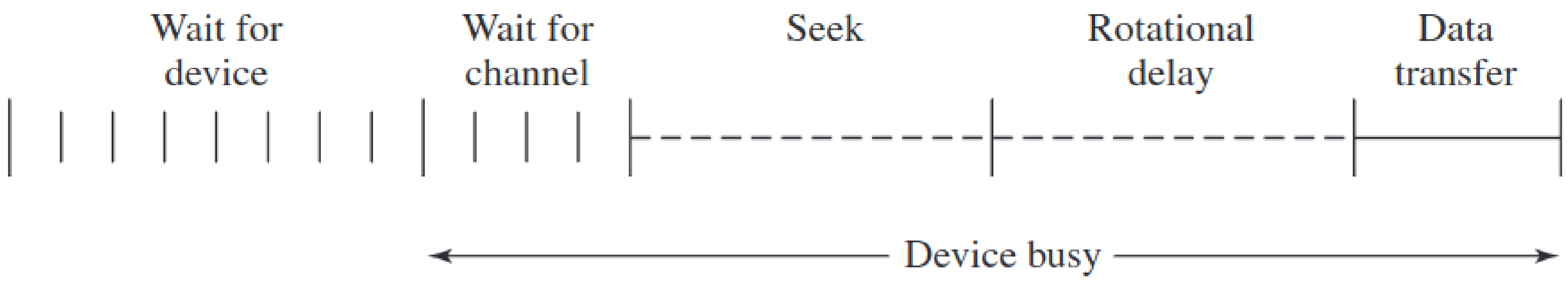


Figure 11.6 Timing of a Disk I/O Transfer

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

EXAMPLE DEVICE: SOLID STATE DRIVE

- No mechanical or moving parts (unlike in hard disks)
- Built out of transistors
- Persistent storage
- We will focus on flash based SSDs
- Data is organized into **flash pages** (512 – 4096 bytes) which are further grouped into **flash blocks** (32-128 pages)

EXAMPLE DEVICE: SOLID STATE DRIVE

- Fast to read
- Slow to write
  - First **erase** a **flash block** (consisting of several flash pages) and then **program** a **flash page**
- Each erase/program cycle **wears out** the flash page: Need to organize writes so that they are distributed over the whole drive

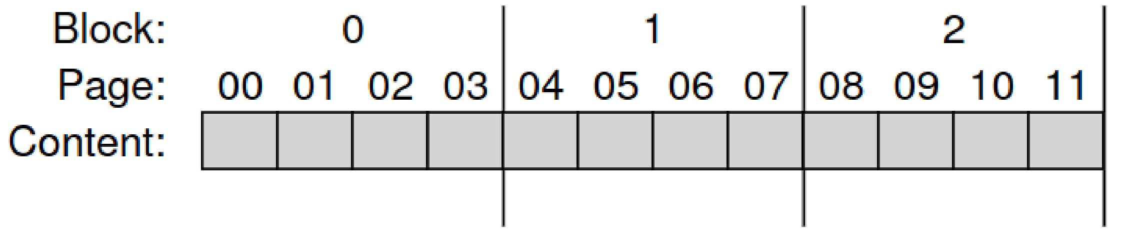


Figure 44.1: **A Simple Flash Chip: Pages Within Blocks**

Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

FLASH TRANSLATION LAYER

- Convert read/write requests on **logical flash pages** to read/erase/program operations on **physical flash pages** and **physical flash blocks**
- To reach good performance
  - Utilize multiple flash chips **in parallel**
  - Reduce **write amplification**: Because writing one flash page requires erasing a whole block, bad organization can result in a lot of extra program operations
- To reach high reliability
  - **Wear leveling:** Attempt to distribute writes evenly across the flash blocks
  - Avoid **program disturbance**: Reading or writing a block can cause bits to flip in neighboring flash pages

FLASH TRANSLATION LAYER: DIRECT MAPPING

- Logical flash page *i* directly corresponds to physical flash page *i*
- Bad approach. Why?

FLASH TRANSLATION LAYER: DIRECT MAPPING

- Logical flash page *i* directly corresponds to physical flash page *i*
- Bad approach. Why?
  - Write amplification: When writing a flash page, need to first read the whole block, erase it, and write all the flash pages back
  - Wear out: writes are not distributed over the whole device

FLASH TRANSLATION LAYER: LOG-STRUCTURED FTL

- Keep a mapping table that maps logical flash pages to physical flash pages
- When writing, append the flash page to the next available flash page in the currently-being-written-to block
- Effective **wear leveling** distributed writes over the drive
- **Garbage collection** needed to manage free space
- Mapping table size is an important consideration

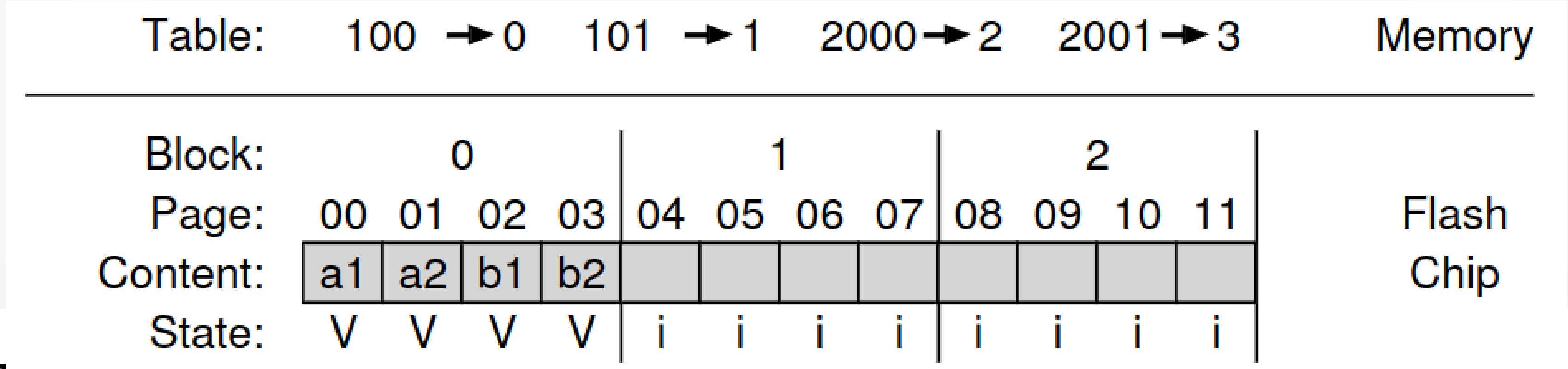


Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

HOW IS DATA STORED ON DISKS?

- Disks provide **persistent** storage
- Data in memory is lost when there is power loss, data on disks remains intact
- How should the OS manage devices providing persistent storage?
- What kind of API does the OS provide for these devices?
- How is the data arranged on such devices?

FILES

- Data collections created by users
- Desirable properties of files
  - **Long-term existence** (storage on persistent devices)
  - **Sharable between processes** (files have names and can have associated access permissions that permit controlled sharing)
  - **Structure** (files can be organized into hierarchical or more complex structure to reflect relationships between files)

DIRECTORIES

- **Basic information** about files in that directory: file name, file type
- **Address information**: where is the file stored, volume, starting address, size used, size allocated
- **Access control information**: owner, other users allowed to access the file, permitted actions
- **Usage information**: date created, identity of creator, date last read access, identity of last reader, date last modified,…
- Directories can contain other directories creating a directory tree

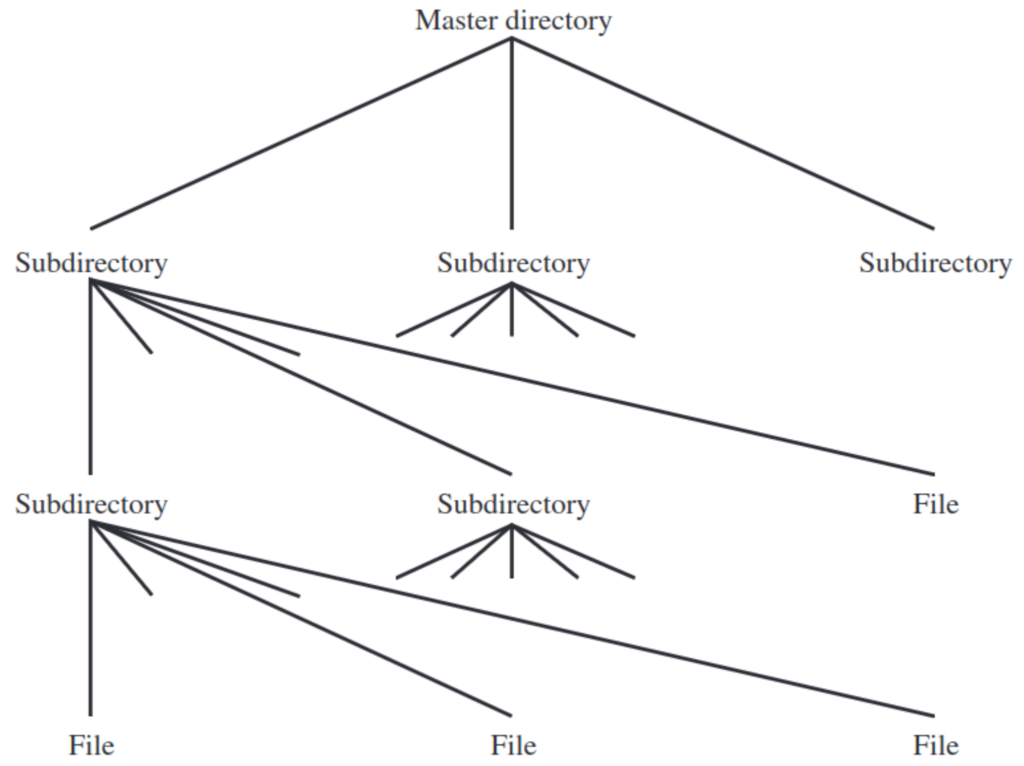


Figure 12.6 **Tree-Structured Directory**

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

OPERATIONS ON FILES & DIRECTORIES

Files:

- Create
- Open
- Read
- Write
- Get information
- Rename
- Link
- Remove

Directories:

- Create
- Read
- Get information
- Rename
- Link
- Remove

FILE SYSTEM IMPLEMENTATION

Data structures

- How is the data and the metadata organized on the disk?
- What structures do we need on the disk?
- What information is tracked by the data structures?
- How are the structures accessed?

Access methods:

- How do we map the requests a process makes onto the data structures?
- Which structures are read/written when processing a particular request (open file, read file, write file, …)?
- How efficiently is this done?

ORGANIZATION OF FILE SYSTEM ON DISK

- Data stored in **blocks** (typical size 4KB) on the disk.
  Note: File system block size need not be the same as disk block size.
- User data stored in data region
- For each file, we store an **inode** in **inode table**
  - Data blocks containing the data stored in the file
  - Other metadata (file size, file owner, creation time, modification time,…)

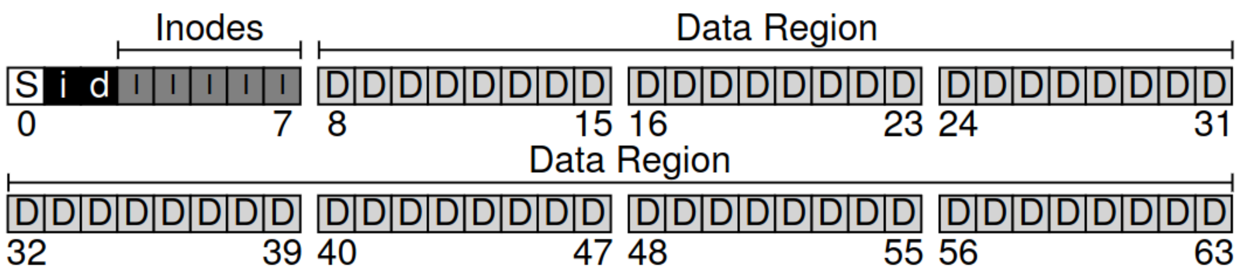


Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

ORGANIZATION OF FILE SYSTEM ON DISK

- **Allocation structures** keep track of free and reserved space (data blocks and inodes)
- Different implementations possible
  - **Free list**: keep a pointer to first free block/inode, which points to the second free block/inode,…
  - **Bitmap**: each bit indicates whether a block is free or not
- **Superblock**: stores the file system parameters (number of inodes, number of data blocks, magic number to identify the file system,…)

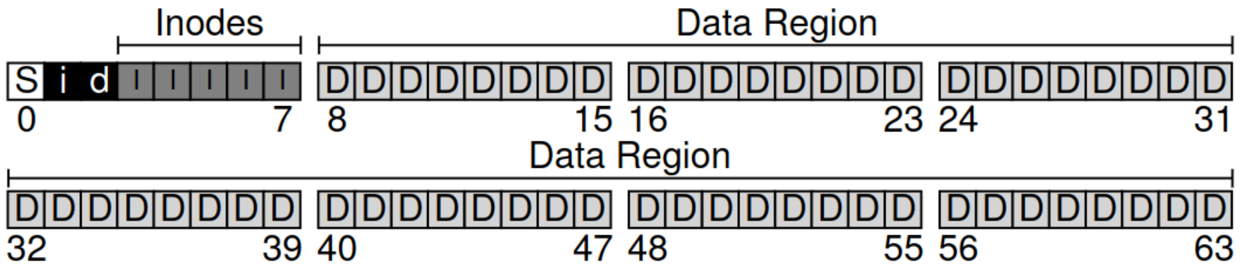


Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

WHAT ABOUT DIRECTORIES?

- In many systems, a directory is just a special type of file
- Predefined structure for the file. E.g.:

```
inum | reclen | strlen | name
  5       12        2      .
  2       12        3      ..
 12       12        4      foo
 13       12        4      bar
 24       36       28      foobar_is_a_pretty_longname
```

- Organization on the disk can thus remain unchanged

Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

EXAMPLE: FILE SYSTEM STRUCTURES

- File system consisting of 64 blocks, 4 KB each
  - Block 0 is superblock
  - Master directory is in inode 0
  - Block 1 contains a bitmap tracking free inodes
  - Block 2 contains a bitmap tracking free data blocks
  - Blocks 3-7 contain inodes
  - Blocks 8-63 contain data blocks

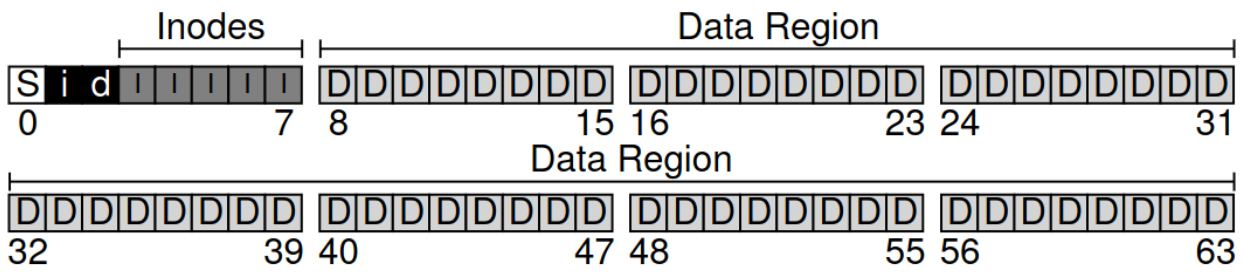
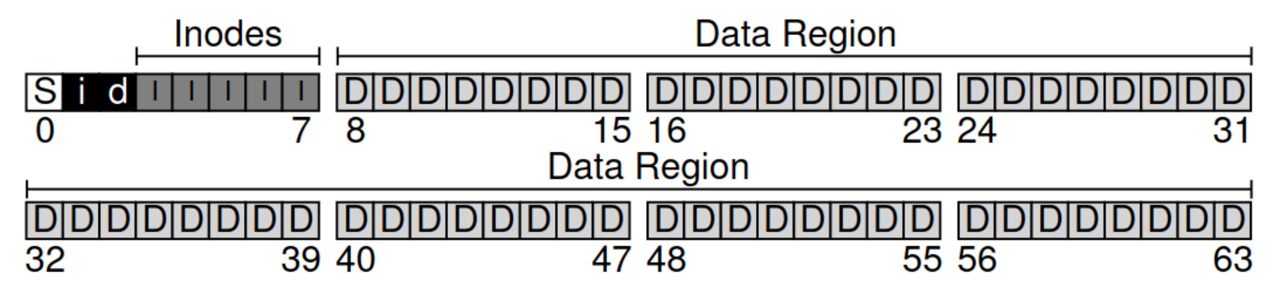


Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

EXAMPLE: FILE SYSTEM STRUCTURES

- File system consisting of 64 blocks, 4 KB each
  - Block 0 is superblock
  - Master directory is in inode 0
  - Block 1 contains a bitmap tracking free inodes
  - Block 2 contains a bitmap tracking free data blocks
  - Blocks 3-7 contain inodes
  - Blocks 8-63 contain data blocks



- How is the file system contents shown below stored in this system?
  - Inodes and their contents?
  - Inode and data block bitmaps?
  - Where are the file contents stored?

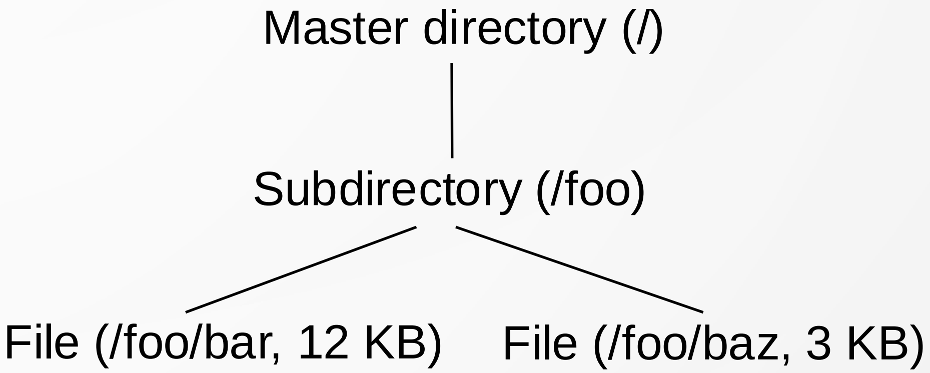
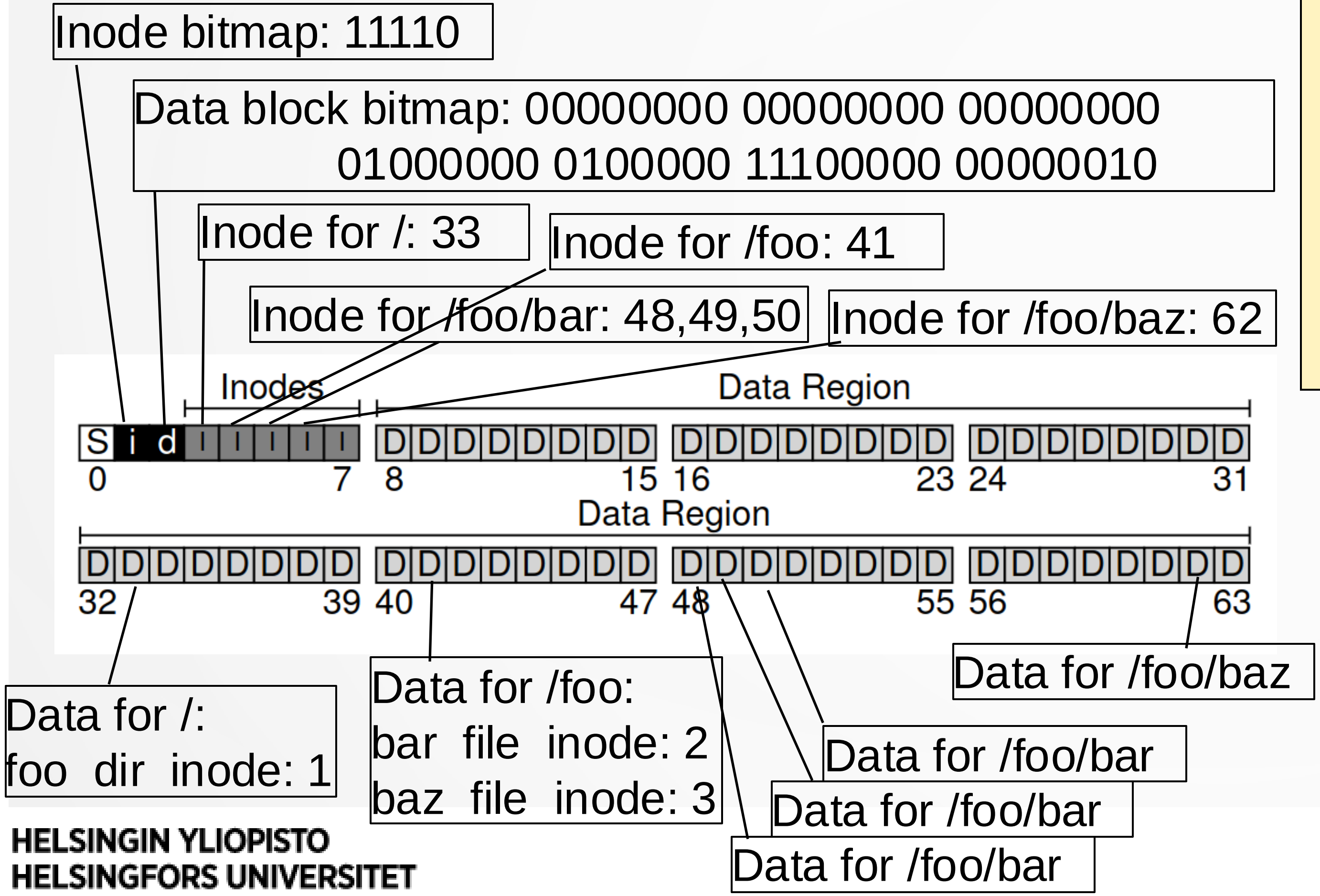


Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

EXAMPLE: FILE SYSTEM STRUCTURES

- File system consisting of 64 blocks, 4 KB each



- How is the file system contents shown below stored in this system?
  - Inodes and their contents?
  - Inode and data block bitmaps?
  - Where are the file contents stored?

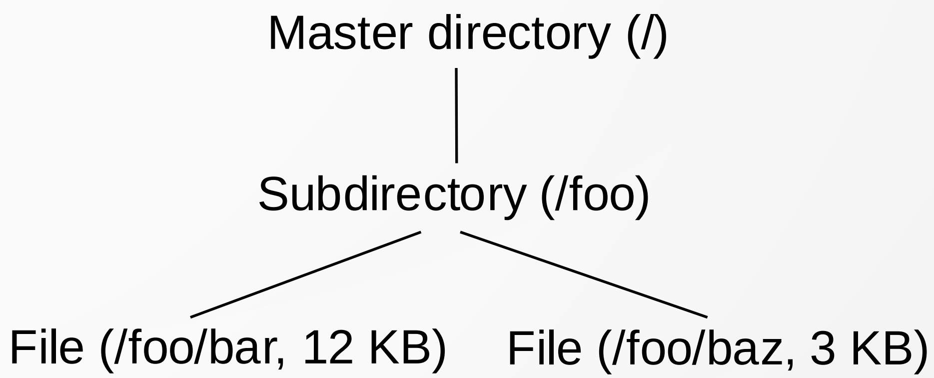


Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

INODES: HOW TO STORE POINTERS TO DATA BLOCKS?

- **Direct pointers**: inode contains pointers directly to data blocks
  - Efficient
  - Limits the size of files
- **Indirect pointers**: inode contains pointers to blocks that contain more pointers
  - Less efficient
  - Larger files can be supported

Most files are small	˜2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of space
File systems contain lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ˜50% full
Directories are typically small	Many have few entries; most have 20 or fewer

Figure 40.2: **File System Measurement Summary**

Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

INODES: MULTILEVEL INDEXES

- Direct and indirect pointers can be mixed. E.g.
  - 13 direct pointers
  - 1 block of single indirect pointers
  - 1 block of double indirect pointers
  - 1 block of triple indirect pointers

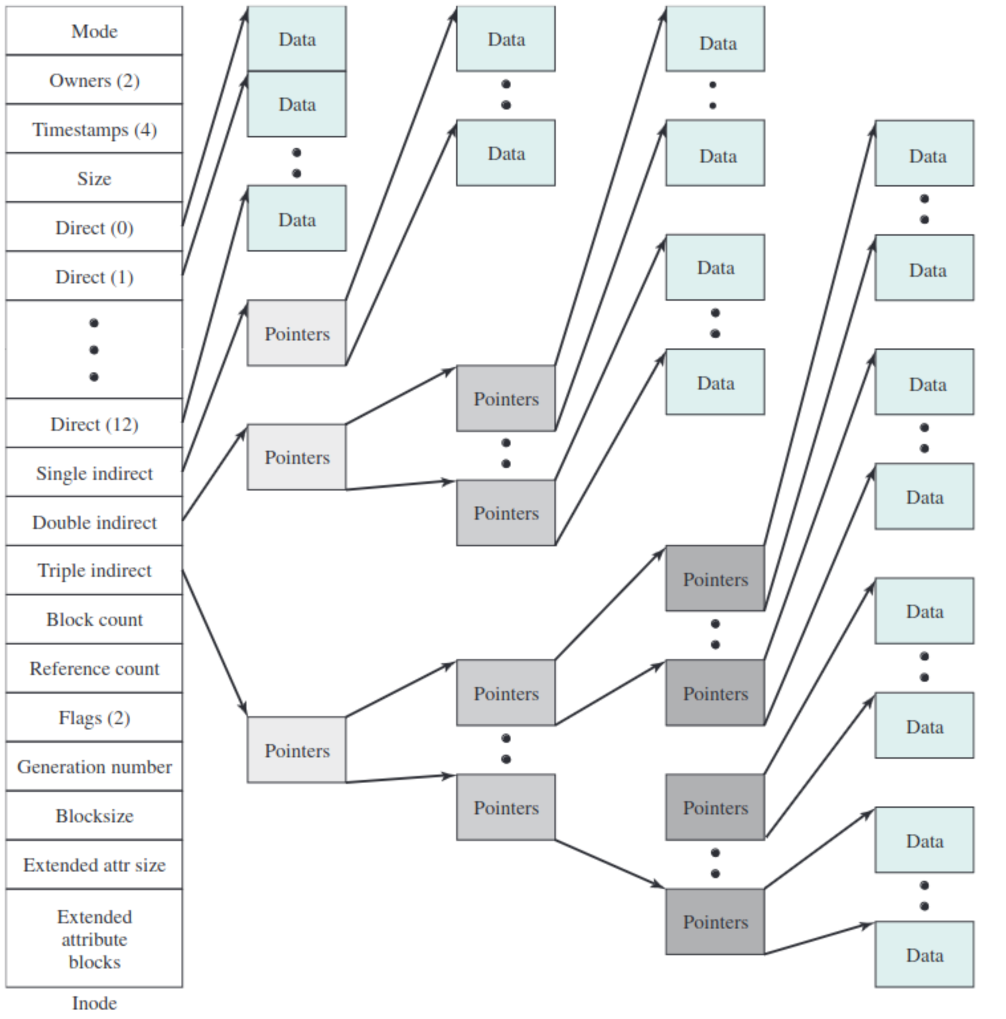


Figure 12.15 Structure of FreeBSD Inode and File

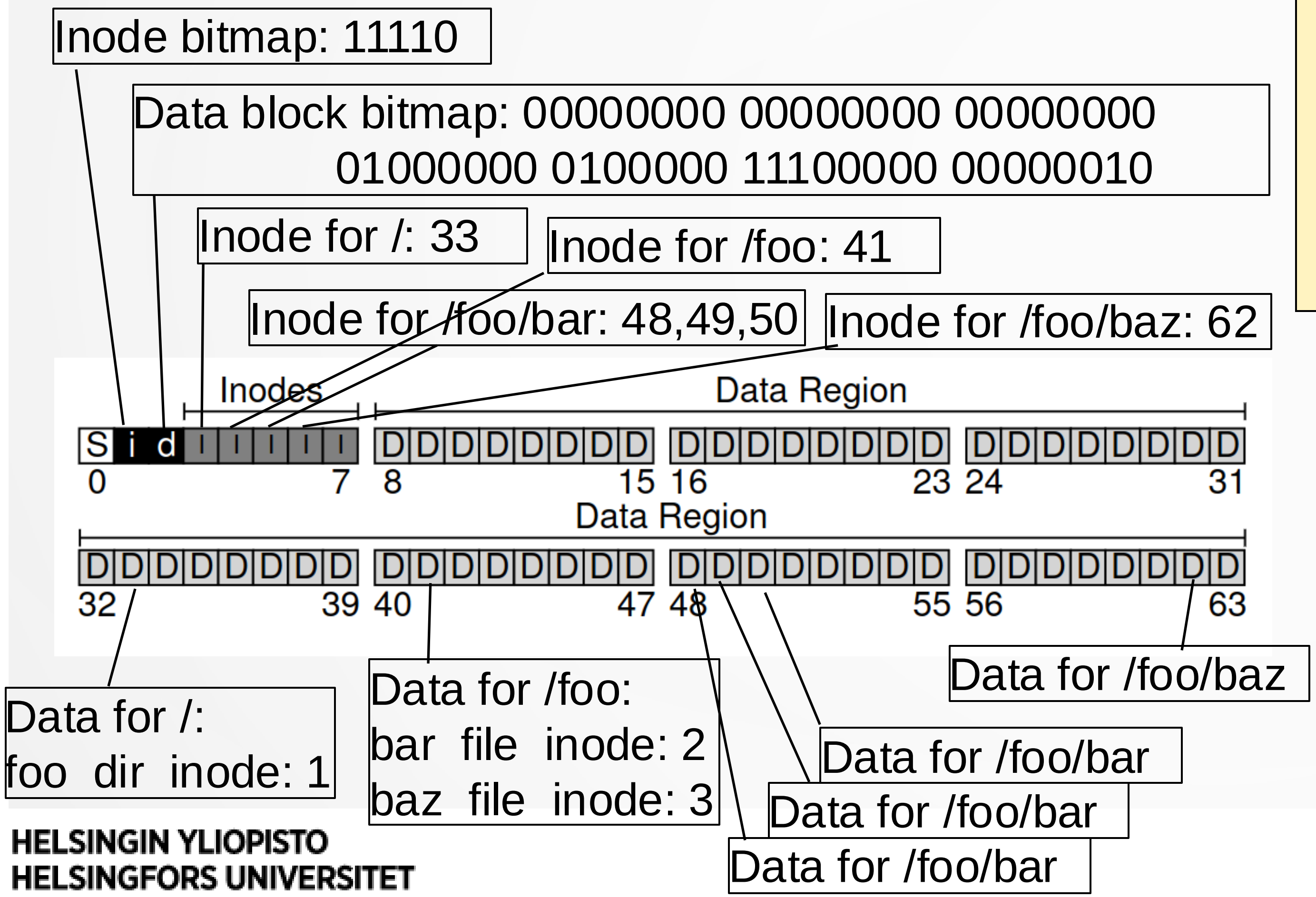
Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

ACCESS METHODS

- How do we map the requests a process makes onto the data structures?
- Which structures are read/written when processing a particular request (open file, read file, write file, …)?
- How efficiently is this done?

EXAMPLE: READING A FILE

- File system consisting of 64 blocks, 4 KB each



- Reading the file /foo/bar
  - Open(/foo/bar)
  - Read the data (12 KB)
  - Close the file handle

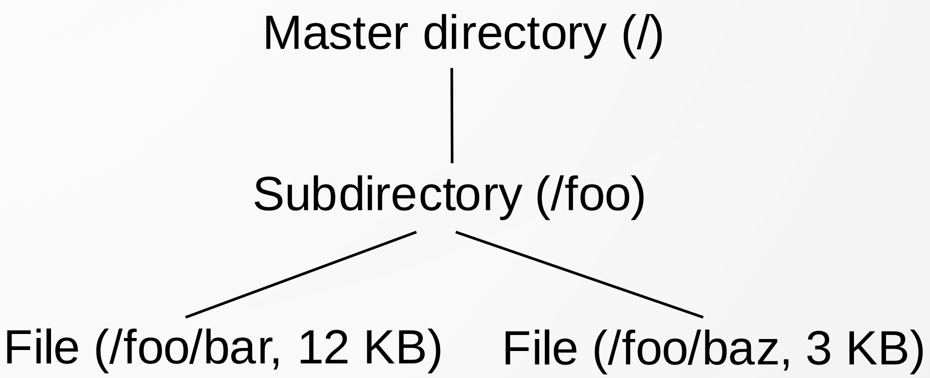
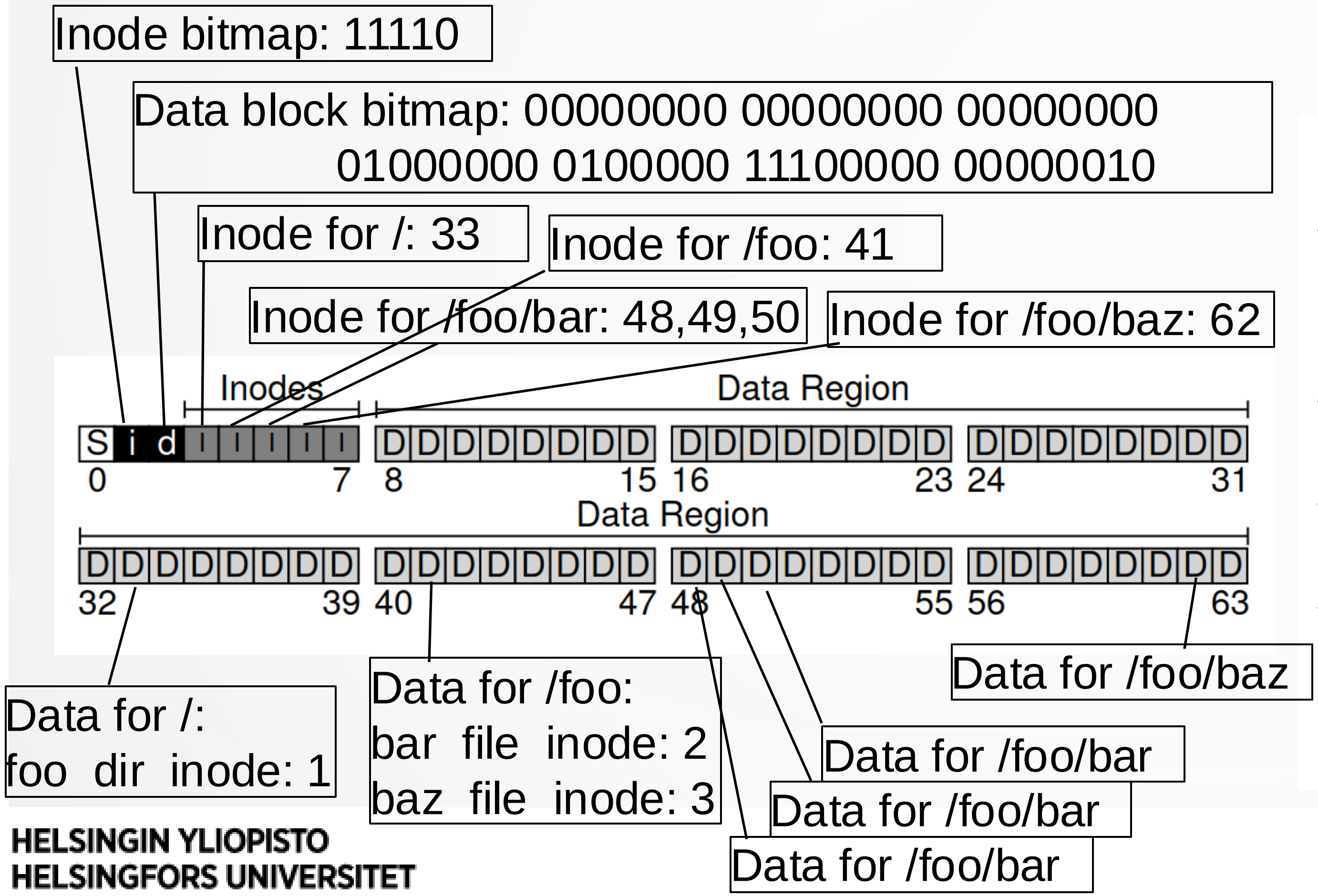


Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

EXAMPLE: READING A FILE

- File system consisting of 64 blocks, 4 KB each

Reading file /foo/bar



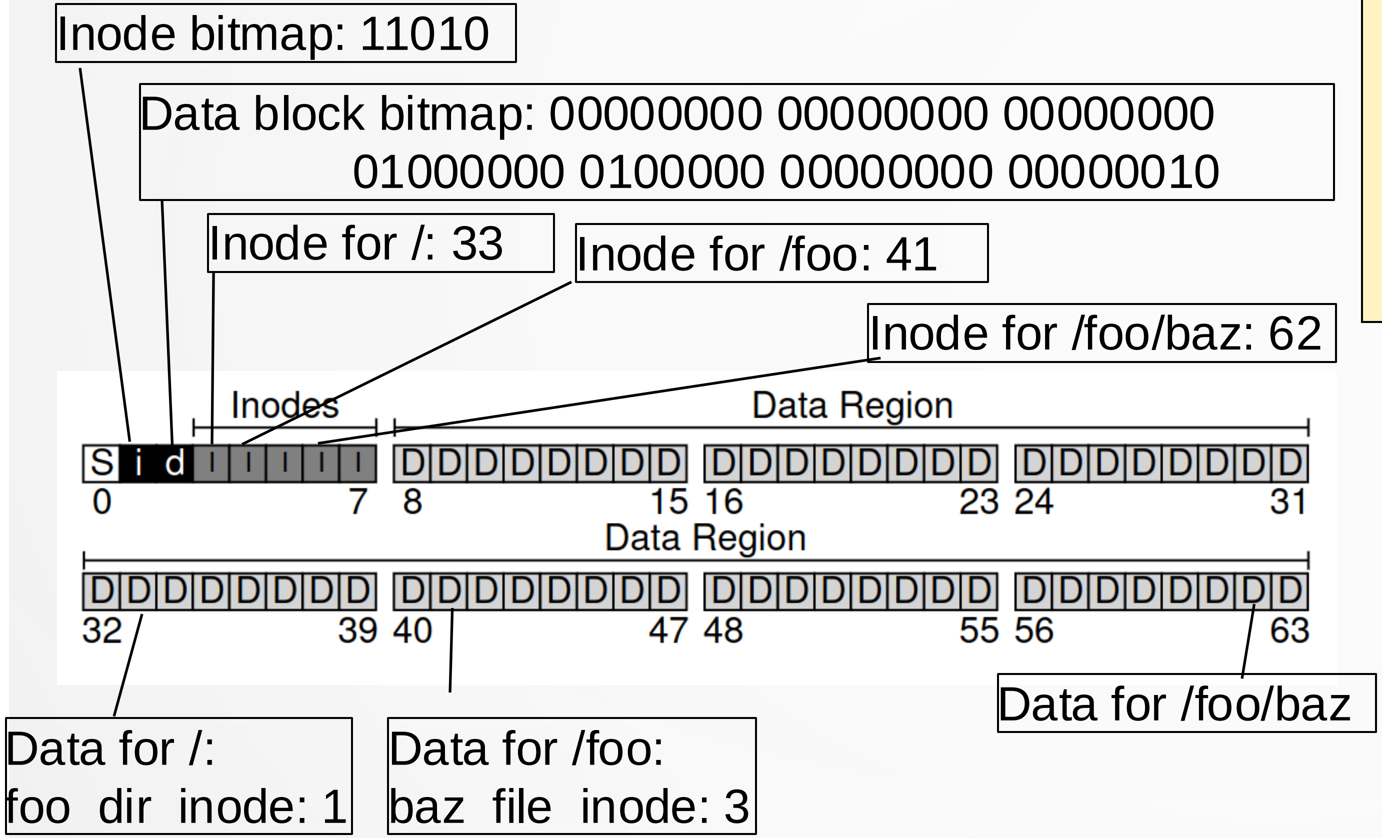
	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			read	read	read	read	read			
read()					read write			read		
read()					read write				read	
read()					read write					read

Figure 40.3: **File Read Timeline (Time Increasing Downward)**

Figures from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

EXAMPLE: WRITING A FILE

- File system consisting of 64 blocks, 4 KB each



- Writing the file /foo/bar
  - Create(/foo/bar)
  - Write the data (12 KB)
  - Close the file handle

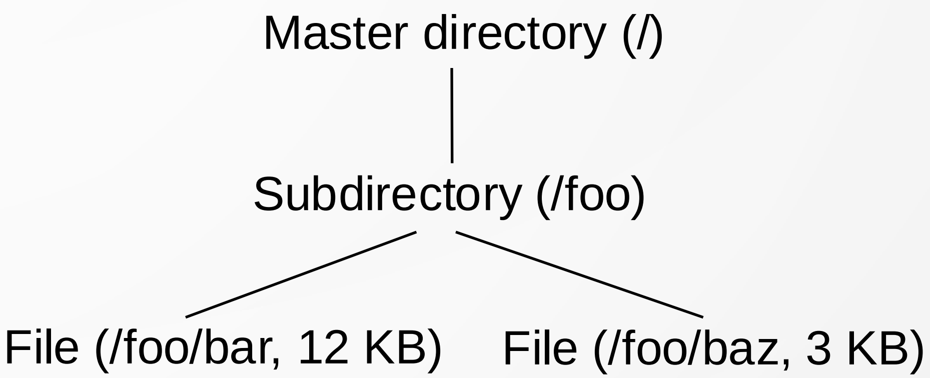
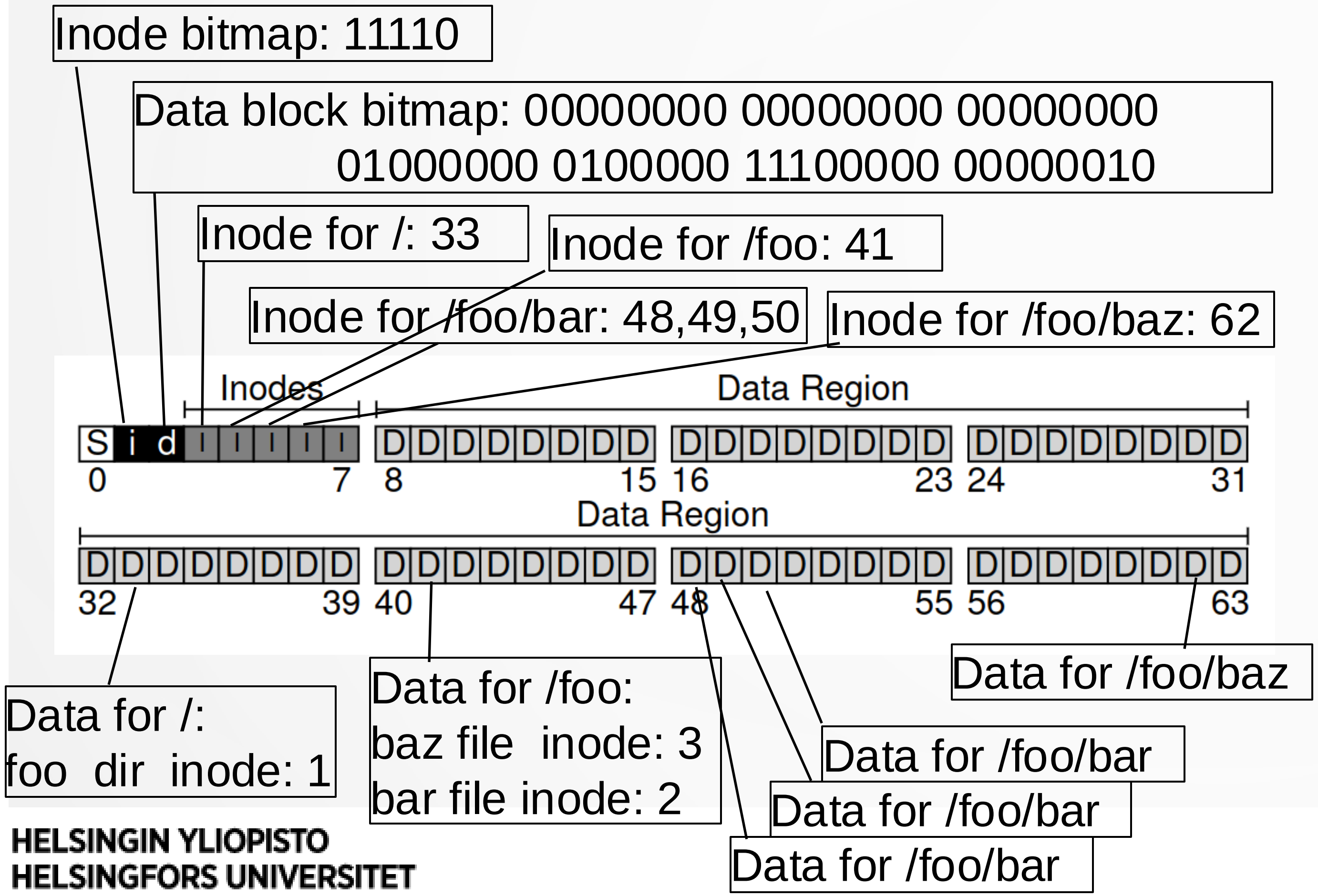


Figure from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

EXAMPLE: WRITING A FILE

- File system consisting of 64 blocks, 4 KB each



Writing file /foo/bar

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)		read write	read	read write	read write	read	read write			
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write

Figure 40.4: **File Creation Timeline (Time Increasing Downward)**

Figures from [Arpaci-Dusseau&Arpaci-Dusseau, Operating Systems: Three Easy Pieces]

CACHING AND BUFFERING

- Reading/writing a single file can cause many slow I/O operations.
  How to reduce the high costs of so many I/O operations?
- **Caching**: keep recently accessed blocks in memory
  - **Static partitioning** allocates a **fixed-size cache** for the file system
  - **Dynamic partitioning** has a common pool of frames for file system pages and virtual memory pages
- **Write buffering** allows
  - **Batching** updates to a smaller set of updates
  - **Scheduling** the writes which can increase performance
  - Sometimes **avoiding** the whole write operation
    (e.g., an application creates a file and deletes it soon after)
- Write buffering can cause data loss in case of system crash

SUMMARY

- **I/O architecture** allows the system to communicate with the outside world
- **Hard disk drives** and **SSDs** are two options for persistent data storage
- **File system** provides services to users / applications
  to store data **persistently on disk**
  - **Data structures**
  - **Access methods**