

File System Implementation

Operating Systems

Hebrew University

Spring 2010

File

- Sequence of bytes, with no structure as far as the operating system is concerned. The only operations are to read and write bytes.
- Interpretation of the data is left to the application using it.
- File descriptor – a handle to a file that the OS provides the user so that it can use the file

File Metadata

- Owner: the user who owns this file.
- Permissions: who is allowed to access this file.
- Modification time: when this file was last modified.
- Size: how many bytes of data are there.
- Data location: where on the disk the file's data is stored.

File System Implementation

- We discuss the classical Unix File system
- All file systems need to solve similar issues.

Hardware background: Direct Memory Access

- When a process needs a block from the disk, the cpu needs to copy the requested block from the disk to the main memory.
- This is a waste of cpu time.
- If we could exempt the cpu from this job, it will be available to run other 'ready' processes.
- This is the reason that the operating system uses the DMA feature of the disk controller.
- Disk access is **always in full blocks**.

Direct Memory Access – Cont.

- Sequence of actions:
 1. OS passes the needed parameters to the disk controller (address on disk, address on main memory, amount of data to copy)
 2. The running process is transferred to the blocked queue and a new process from the ready queue is selected to run.
 3. The controller transfers the requested data from the disk to the main memory using DMA.
 4. The controller sends an interrupt to the cpu, indicating the IO operation has been finished.
 5. The waiting process is transferred to the ready queue.

Disk Layout

Boot Block – for loading the OS (optional)

Swap area (optional)

Super Block – File system management

i-nodes – File metadata

Data blocks – Actual file data

Super Block

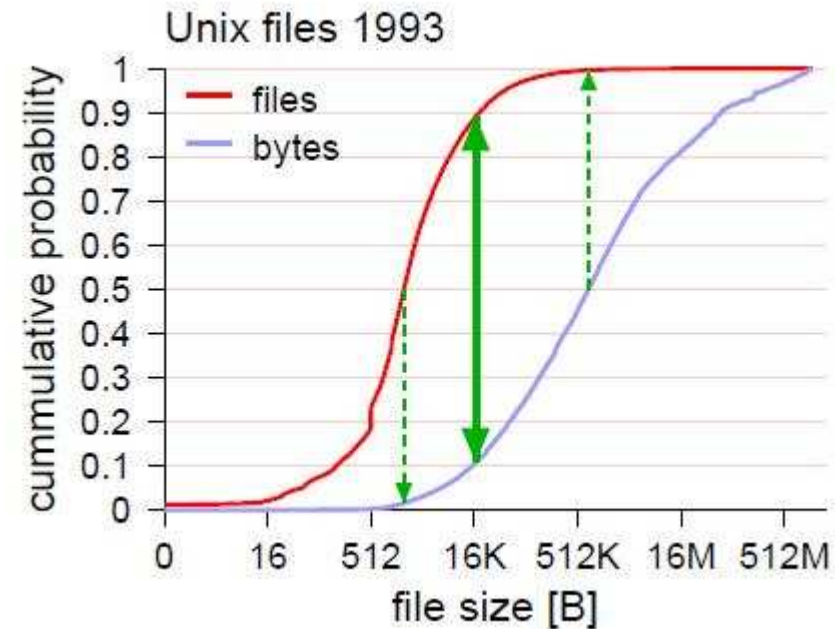
- Manages the allocation of blocks on the file system area
- This block contains:
 - The size of the file system
 - A list of free blocks available on the fs
 - A list of free i-nodes in the fs
 - And more...
- Using this information it is possible to allocate disk block for saving file data or file metadata

Mapping File Blocks

- It is inefficient to save each file as a consecutive data block.
 - Why?
- How do we find the blocks that together constitute the file?
- How do we find the right block if we want to access the file at a particular offset?
- How do we make sure not to spend too much space on management data?
- We need an efficient way to save files of varying sizes.

Typical Distribution of File Sizes

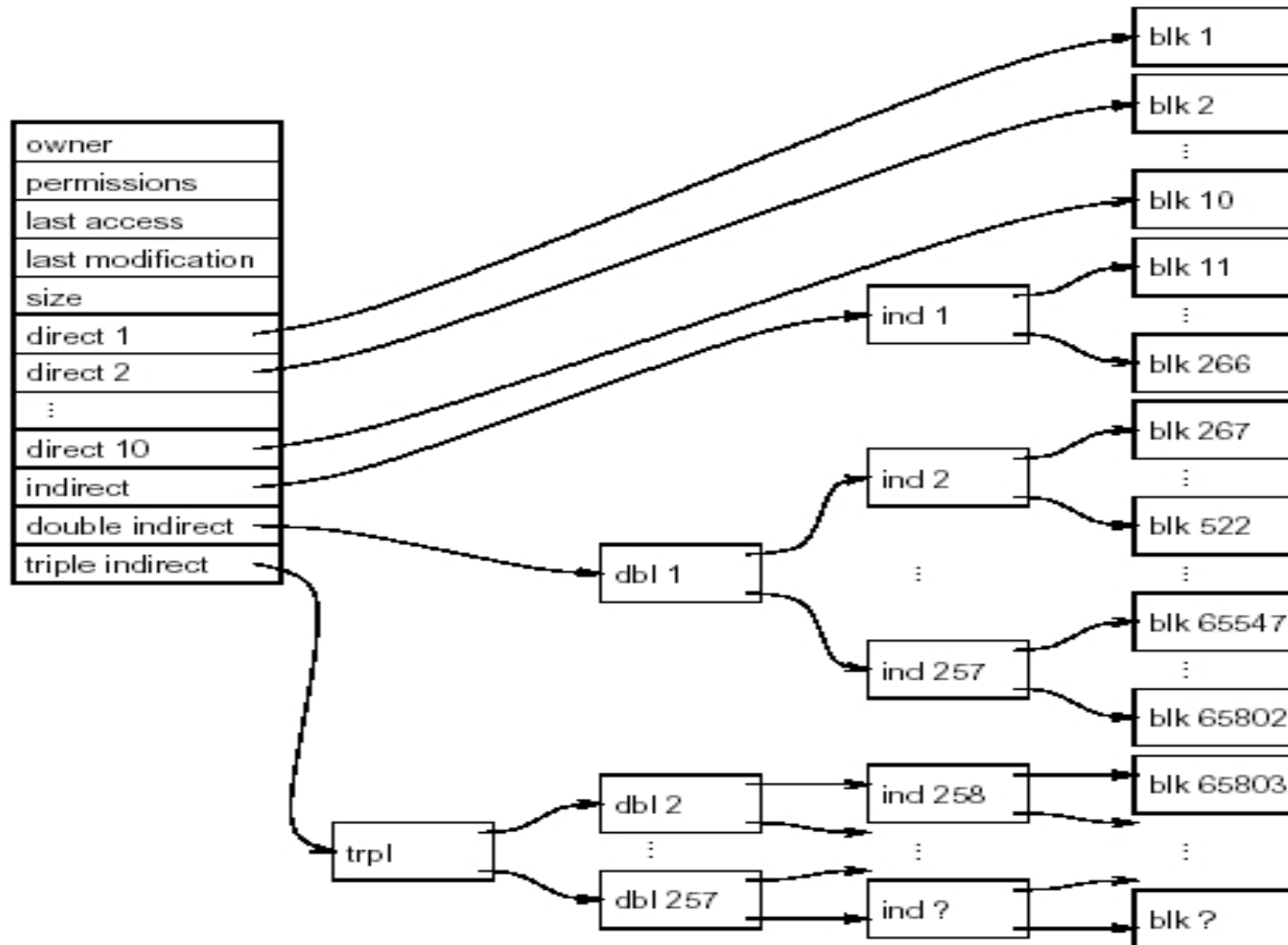
- Many very small files that use little disk space
- Some intermediate files
- Very few large files that use a large part of the disk space



The Unix i-node

- In Unix, files are represented internally by a structure known as an inode, which includes an index of disk blocks.
- The index is arranged in a hierarchical manner:
 - Few *direct* pointers, which list the first blocks of the file (Good for small files)
 - One single *indirect* pointer - points to a whole block of additional direct pointers (Good for intermediate files)
 - One *double* indirect pointer - points to a block of indirect pointers. (Good for large files)
 - One *triple* indirect pointer - points to a block of double indirect pointers. (Good for very large files)

i-node Structure



i-node - Example

- Blocks are 1024 bytes.
- Each pointer is 4 bytes.
- The 10 direct pointers then provide access to a maximum of 10 KB.
- The indirect block contains 256 additional pointers, for a total of 266 blocks (266 KB).
- The double indirect block has 256 pointers to indirect blocks, so it represents a total of 65536 blocks.
- File sizes can grow to a bit over 64 MB.

i-node – a more up-to-date example

- Blocks are 4096 bytes.
- Each pointer is 4 bytes.
- The 12 direct pointers then provide access to a maximum of 48 KB.
- The indirect block contains 1024 additional pointers, for a data of size (4 MB).
- The double indirect block has 1024 pointers to indirect blocks, so it points to 4GB of data
- The triple indirect block allow files of 4TB
- This is a 64bit implementation!

i-node Allocation

- The super blocks caches a short list of free inodes. When a process needs a new inode, the kernel can use this list to allocate one.
- When an inode is freed, its location is written in the super block, but only if there is room in the list
- If the super block list of free inodes is empty, the kernel searches the disk and adds other free inodes to its list.
- To improve performance, the super block contains the **number** of free inodes in the file system.

Allocation of data blocks

- When a process writes data to a file, the kernel must allocate disk blocks from the file system for a direct or indirect block.
- Super block contains the number of free disk blocks in the file system.
- When the kernel wants to allocate a block from the file system, it allocates the next available block in the super block list.

Storing and Accessing File Data

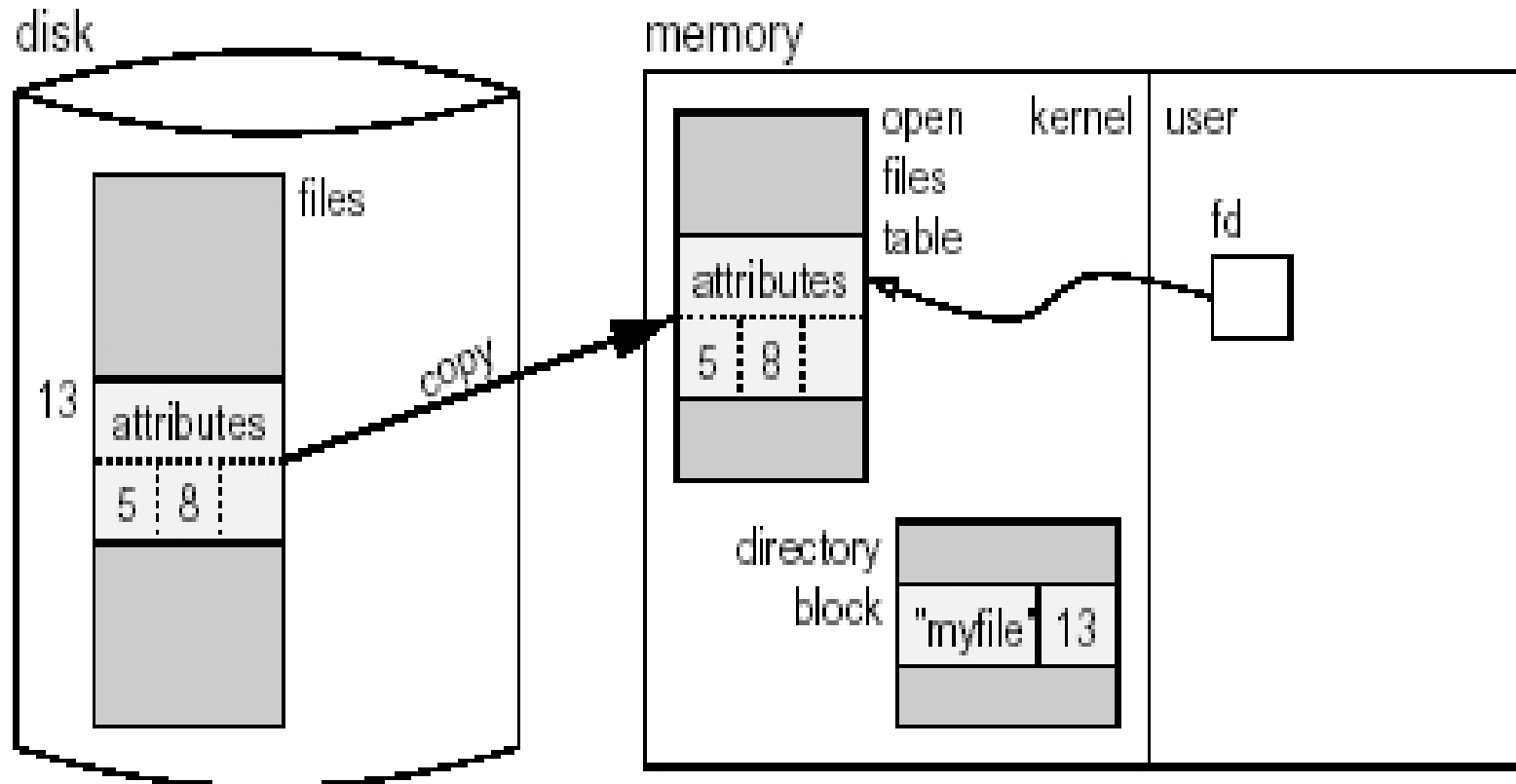
- Storing data in a file involves:
 - Allocation of disk blocks to the file.
 - Read and write operations
 - Optimization - avoiding disk access by caching data in memory.

File Operations

- **Open:** gain access to a file.
- **Close:** relinquish access to a file.
- **Read:** read data from a file, usually from the current position.
- **Write:** write data to a file, usually at the current position.
- **Append:** add data at the end of a file.
- **Seek:** move to a specific position in a file.
- **Rewind:** return to the beginning of the file.
- **Set attributes:** e.g. to change the access permissions.

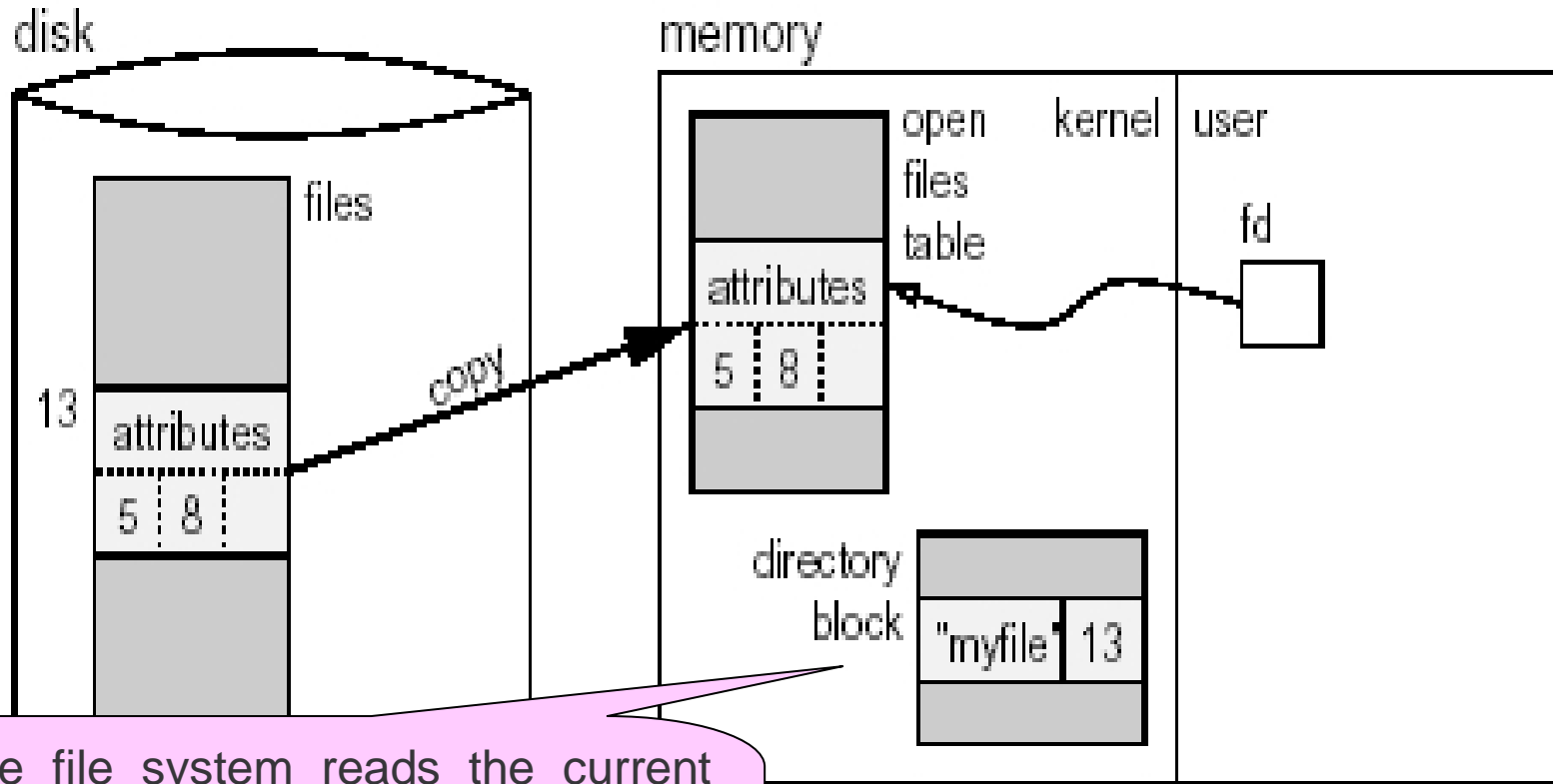
Opening a File

```
fd=open("myfile",R)
```



Opening a File

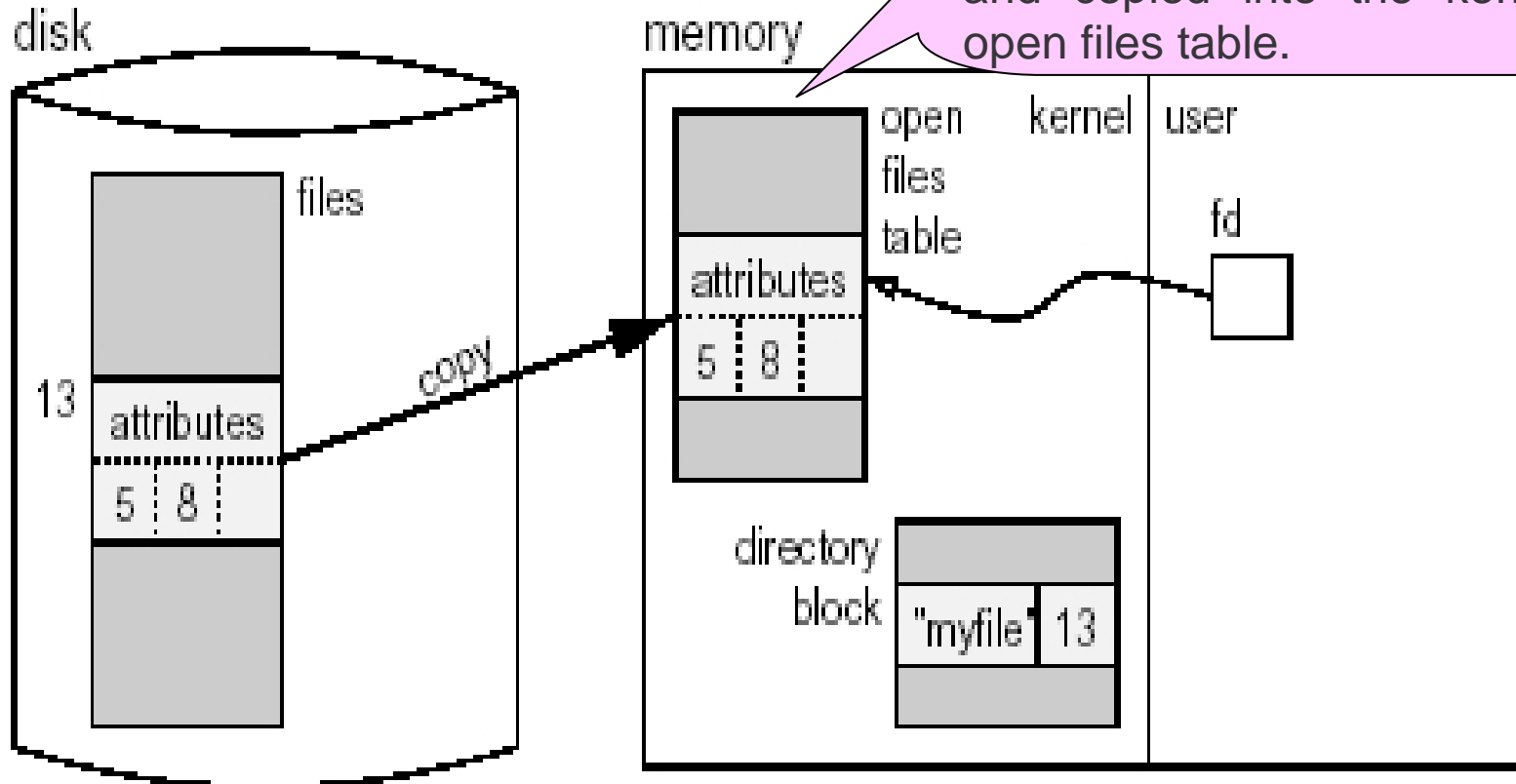
```
fd=open("myfile",R)
```



1. The file system reads the current directory, and finds that "myfile" is represented internally by entry 13 in the list of files maintained on the disk.

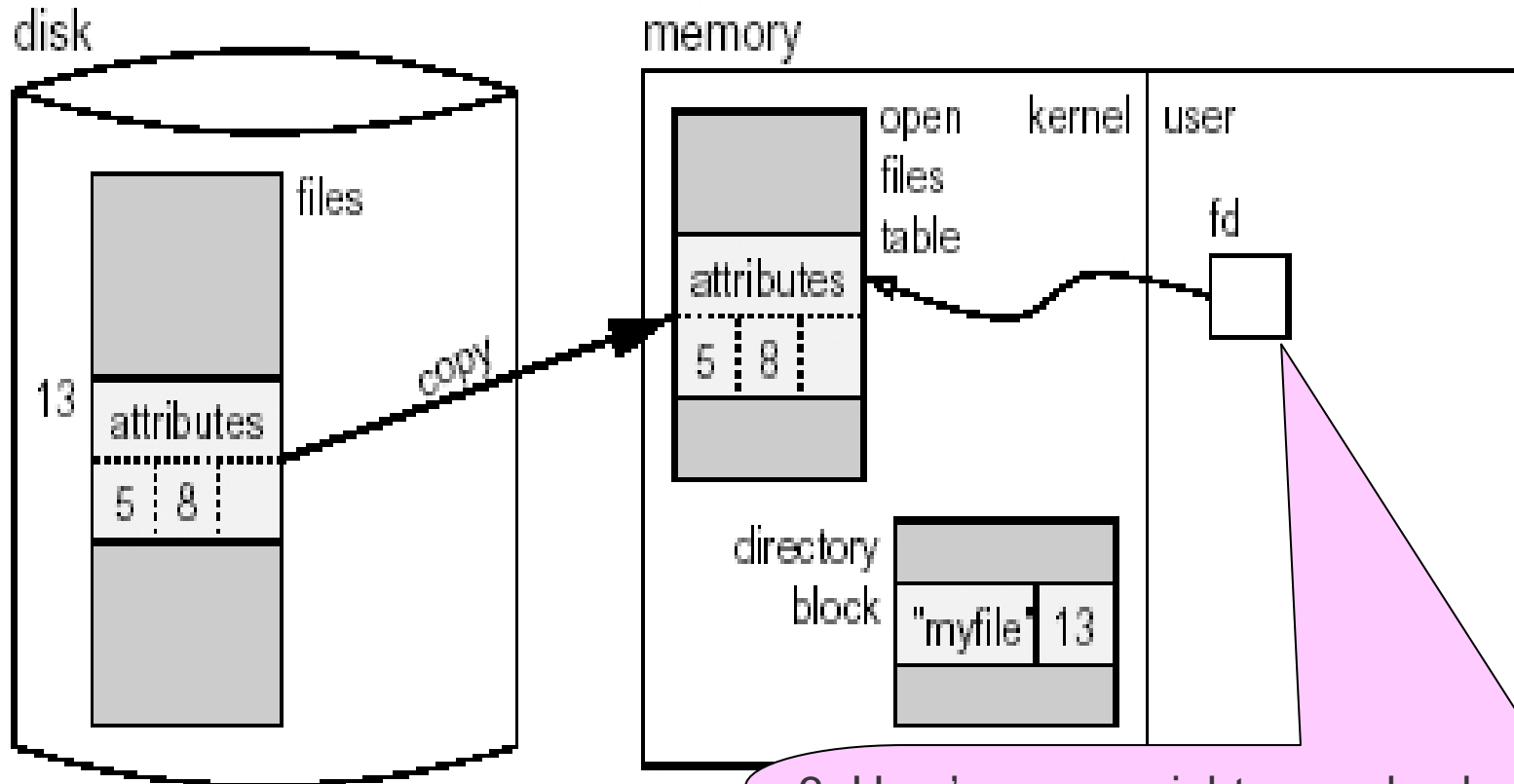
Opening a File

`fd=open("myfile",R)`



Opening a File

```
fd=open("myfile",R)
```



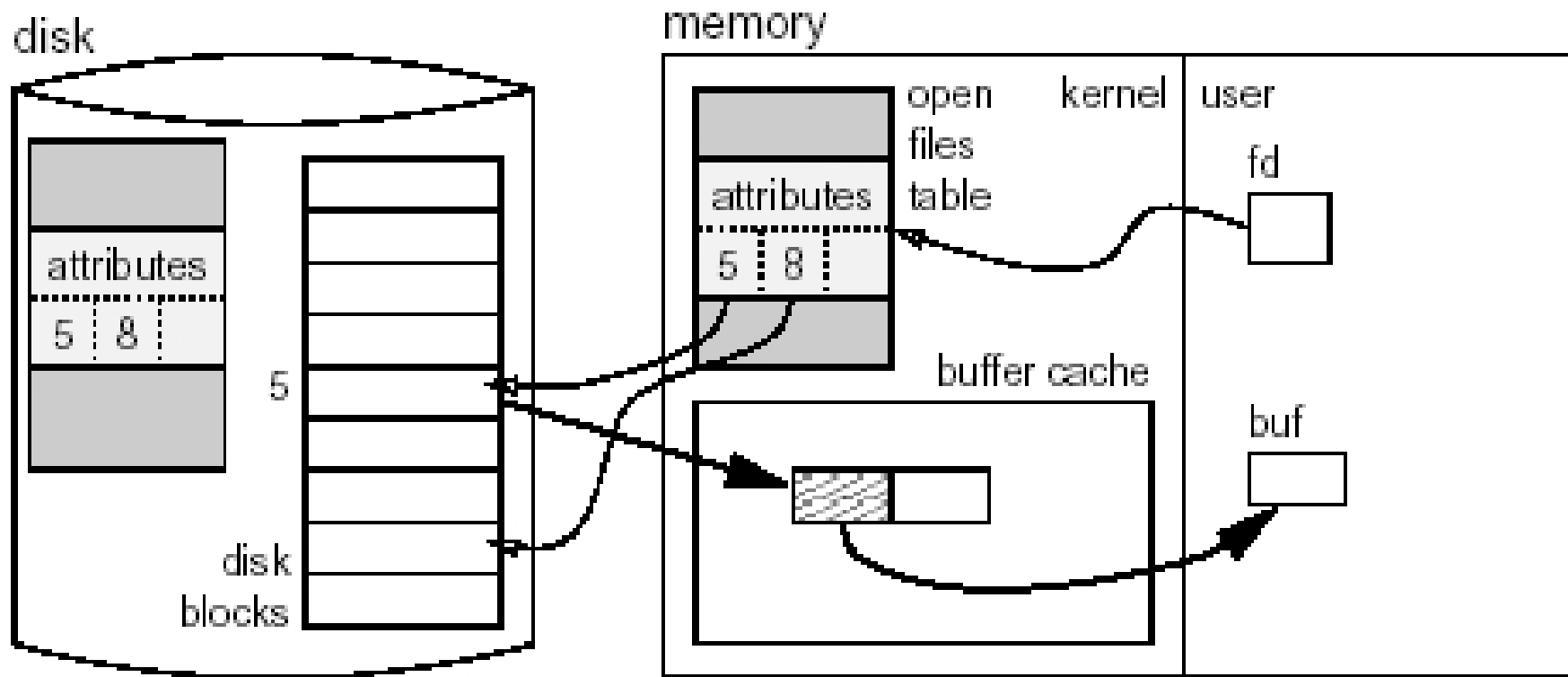
3. User's access rights are checked and the user's variable **fd** is made to point to the allocated entry in the open files table

Opening a File

- Why do we need the open file table?
 - Temporal Locality principle.
 - Saving disk calls.
- All the operations on the file are performed through the file descriptor.

Reading from a File

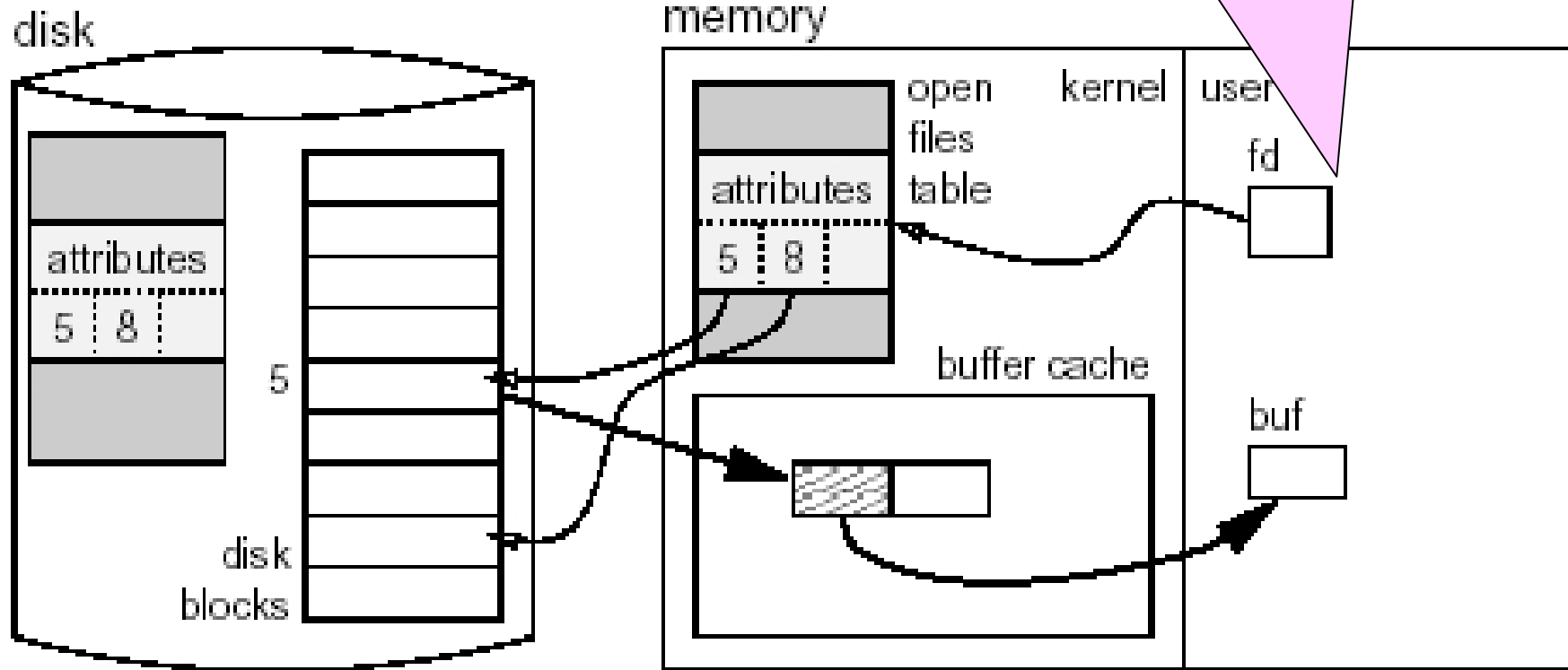
`read(fd,buf,100)`



Reading from a File

`read(fd,buf,100)`

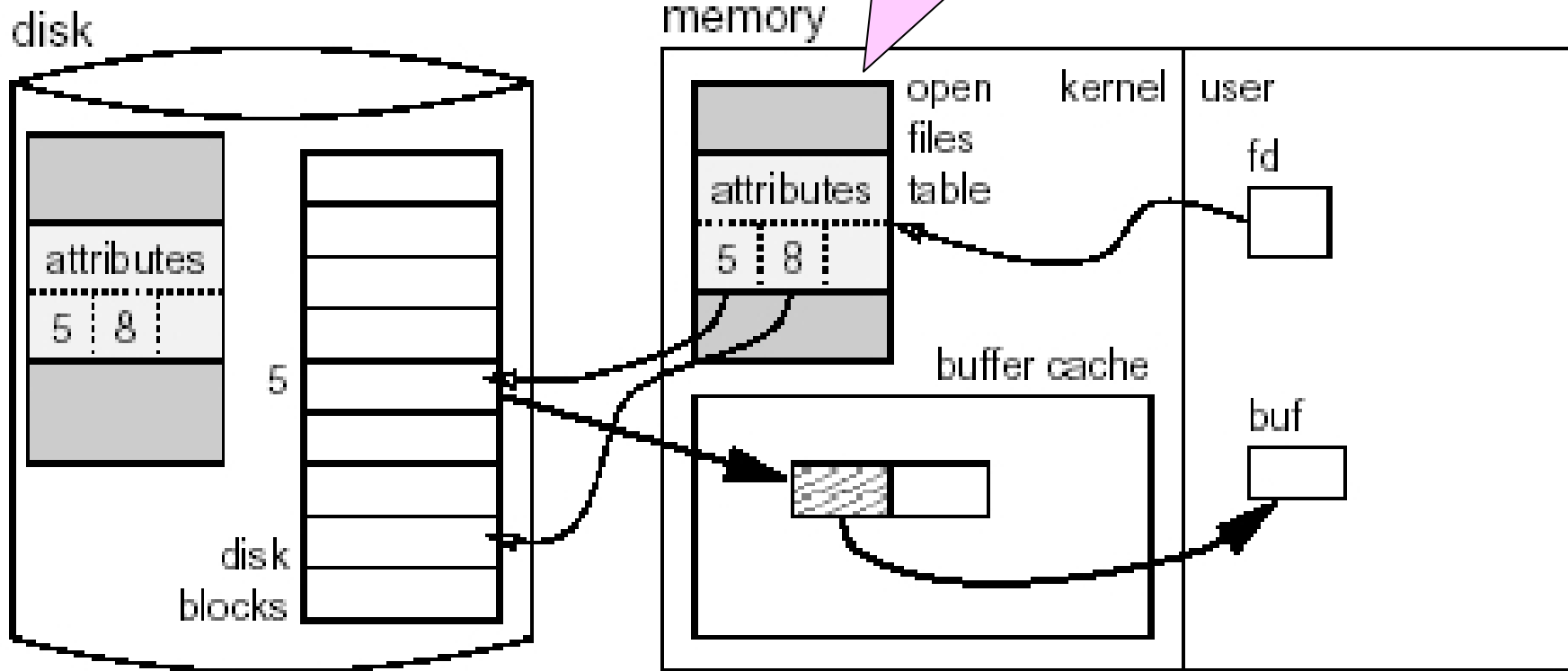
1. The argument `fd` identifies the open file by pointing into the kernel's open files table.



Reading from a File

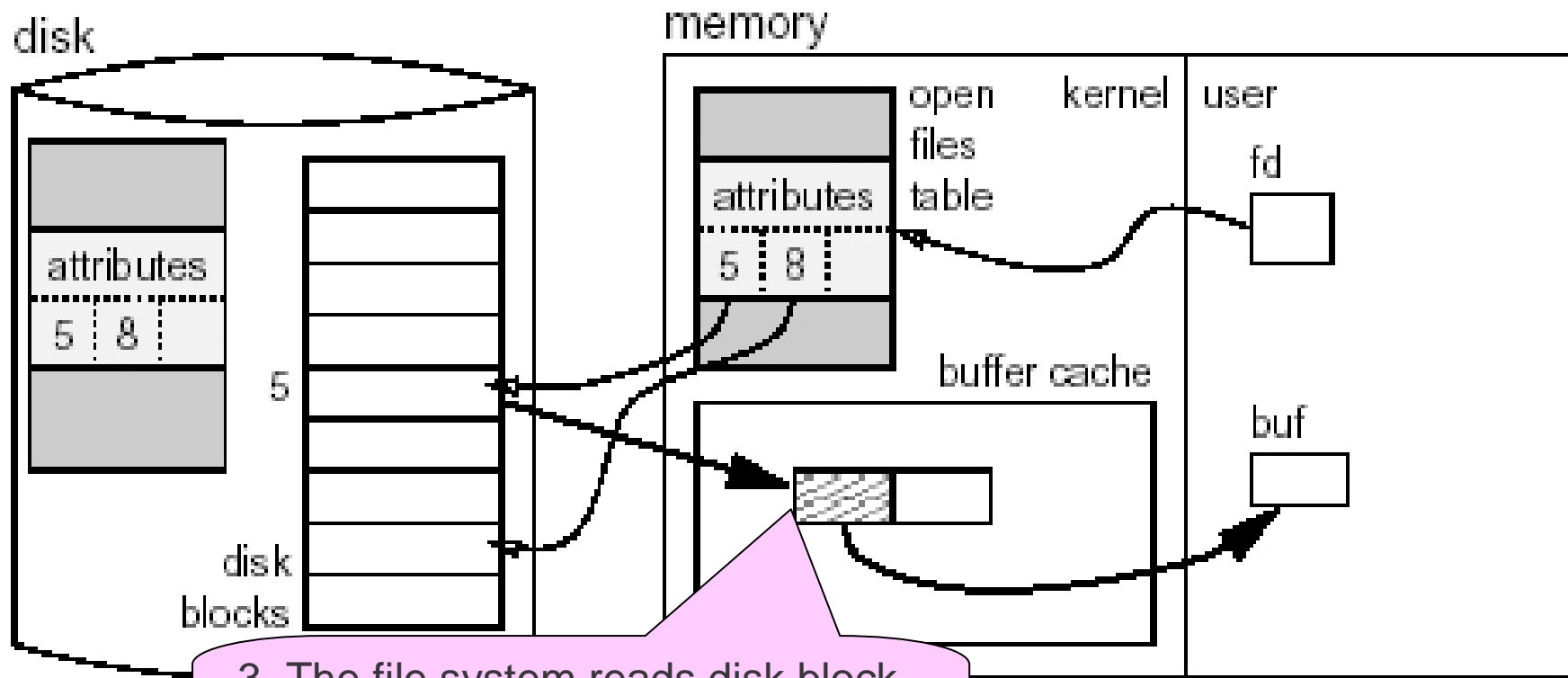
`read(fd,buf,100)`

2. The system gains access to the list of blocks that contain the file's data.



Reading from a File

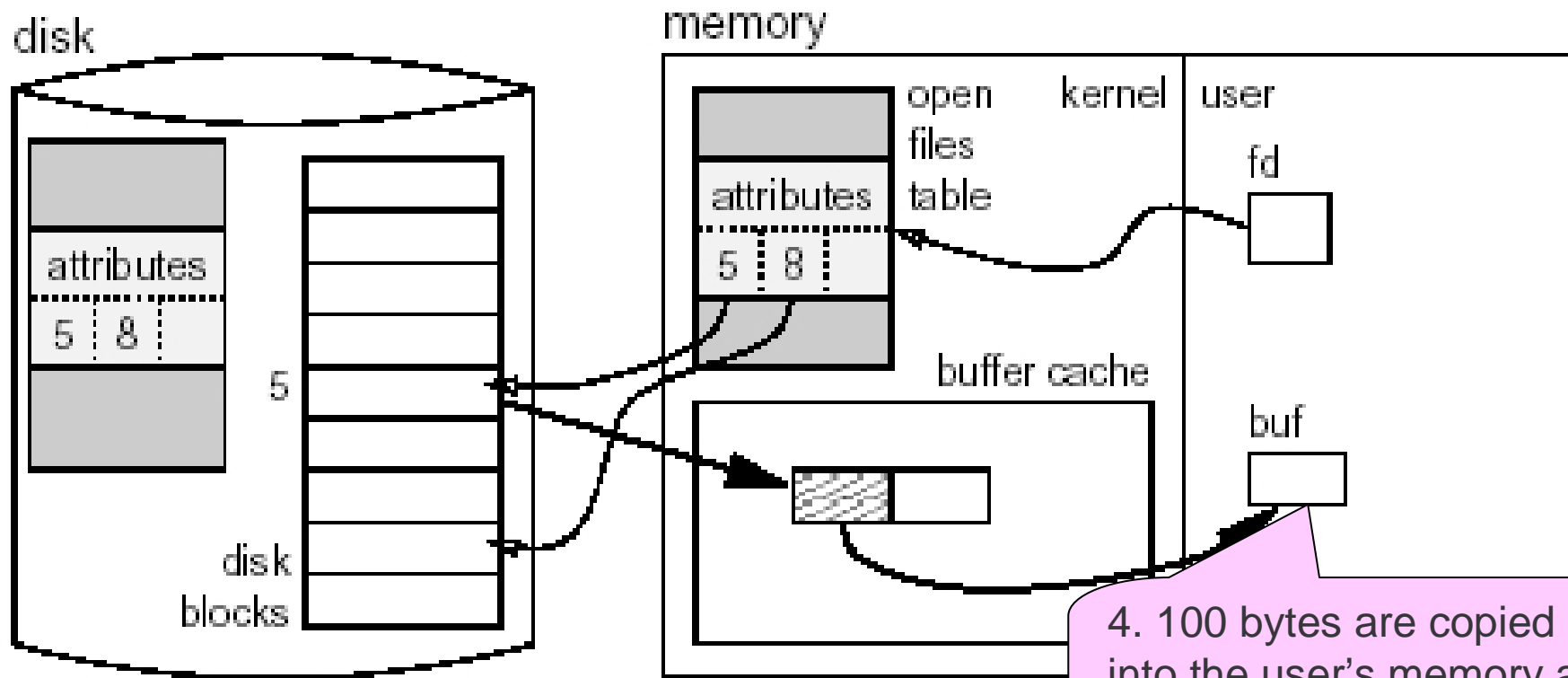
`read(fd,buf,100)`



3. The file system reads disk block number 5 into its *buffer cache*. (full block = Spatial Locality)

Reading from a File

`read(fd,buf,100)`

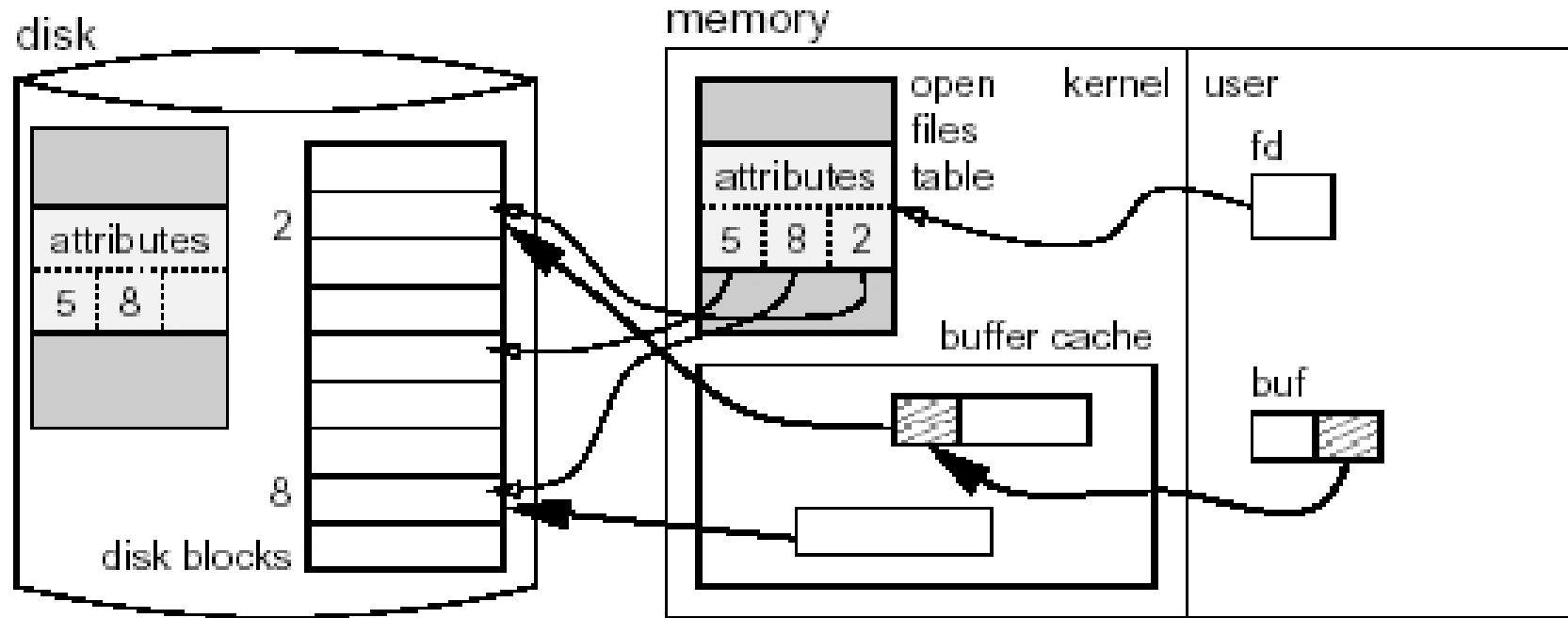


4. 100 bytes are copied into the user's memory at the address indicated by buf.

Writing to a File

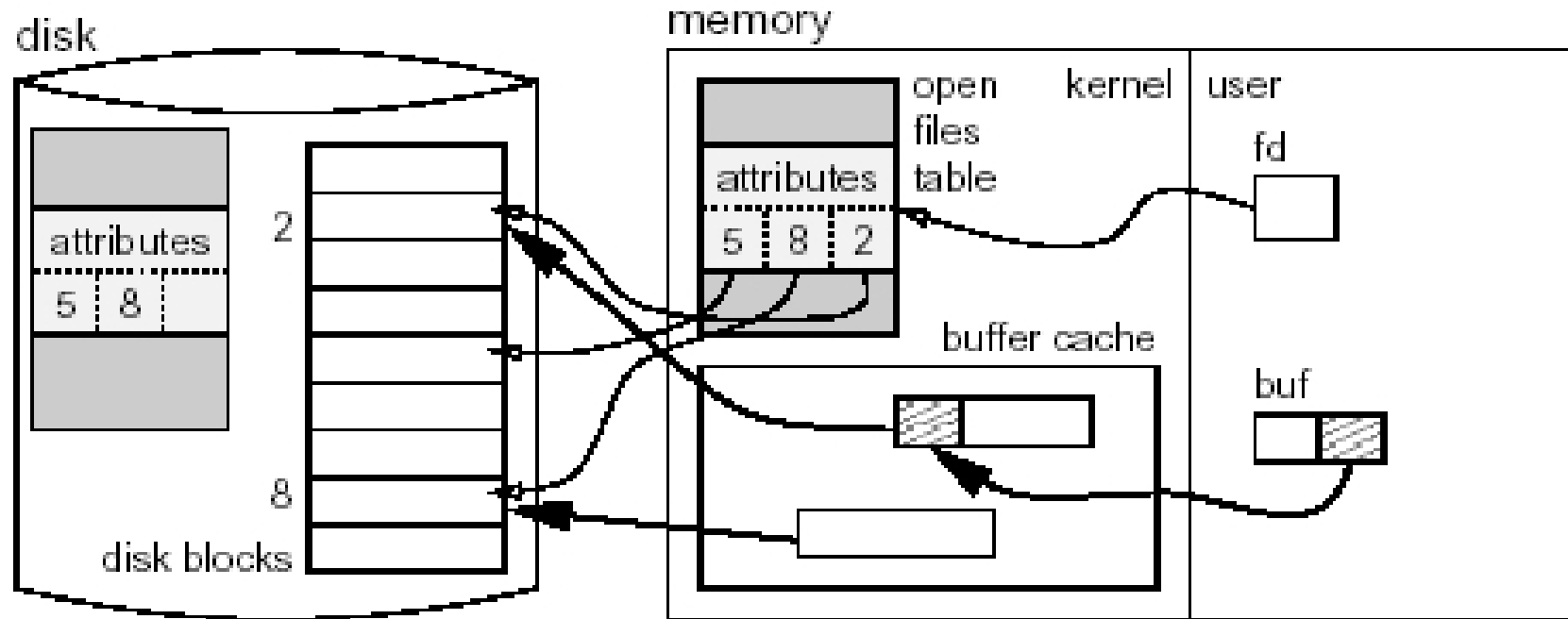
- Assume the following scenario:
 1. We want to write 100 bytes, starting with byte 2000 in the file.
 2. Each disk block is 1024 bytes.
- Therefore, the data we want to write spans the end of the second block to the beginning of the third block.
- The full block must first be read into the buffer cache. Then the part being written is modified by overwriting it with the new data.

Writing to a File – Cont.



- Write 48 bytes at the end of block number 8.
- The rest of the data should go into the third block.
- The third block is allocated from the pool of free blocks.

Writing to a File – Cont.



- We do not need to read the new block from disk.
- Instead
 - allocate a new block in the buffer cache
 - prescribe that it now represents block number 2
 - copy the requested data to it (52 bytes).
- Finally, the modified blocks are written back to the disk.

A note on the buffer cache

- The buffer cache is important to improve performance.
- But it can cause reliability issues:
 - It delays the writeback to disk
 - Therefore if we are unlucky the data may be lost.

The Location in the File

- The **read** system-call provides a buffer address for placing the data in the user's memory, but does not indicate the **offset in the file** from which the data should be taken.
- The operating system maintains the **current offset** into the file, and updates after each operation.
- If random access is required, the process can set the file pointer to any desired value by using the **seek** system call.

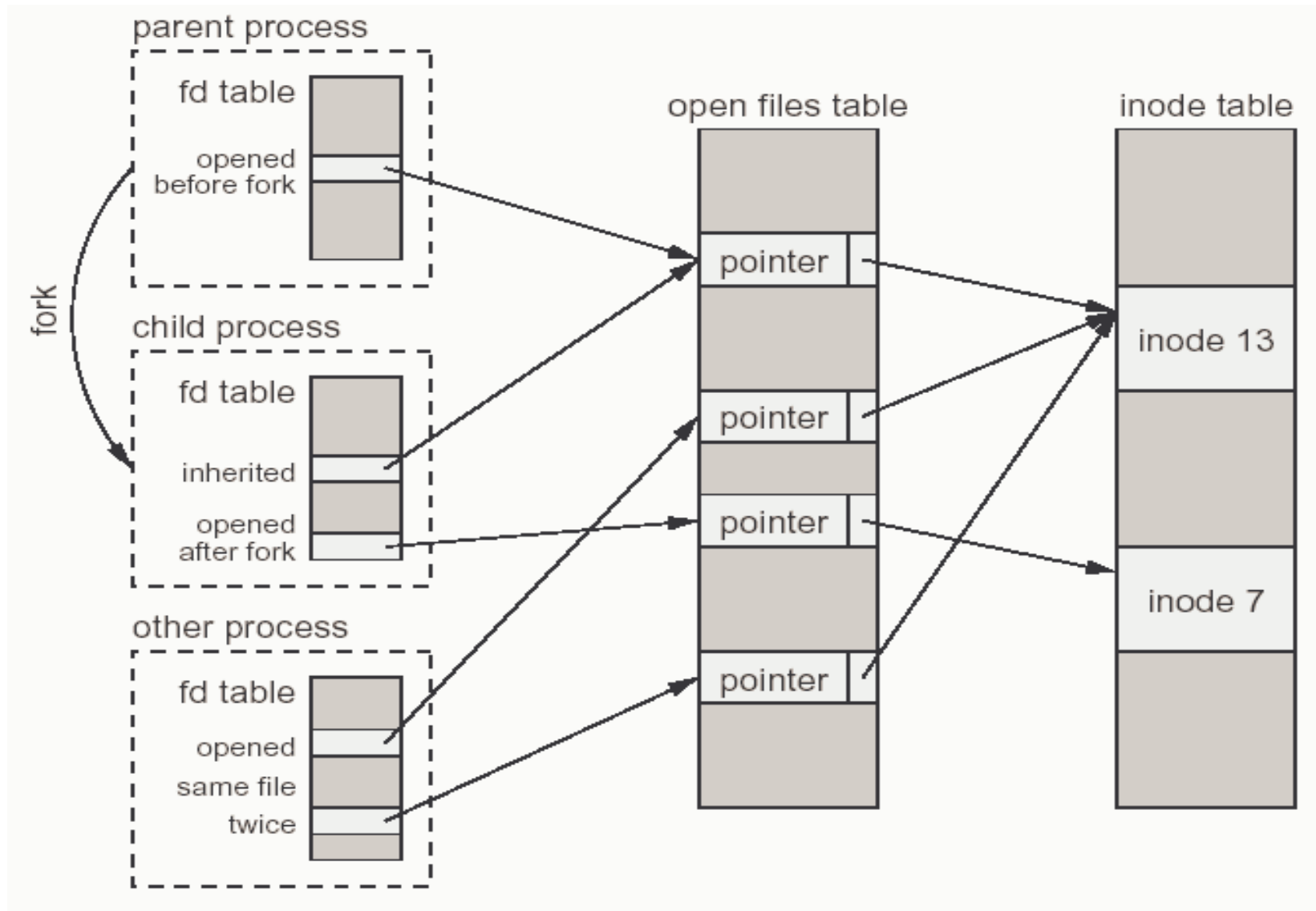
The main OS file tables

- **The i-node table** - each file may appear at most once in this table.
- **The open files table** – an entry in this table is allocated every time a file is opened. Each entry contains a pointer to the inode table and a position within the file. There can be multiple open file entries pointing to the same i-node.
- **The file descriptor table** – separate for each process. Each entry points to an entry in the open files table. The index of this slot is the fd that was returned by **open**.

Why three tables?

- Assume there is a **log file** that needs to be written by more than one process.
- If each process has **its own file pointer**, there is a danger that one process will overwrite log entries of another process.
- If the file pointer is **shared**, each new log entry will indeed be added to the end of the file.
- The **three OS file tables** allow us to control sharing and permissions.
- The file descriptor table adds a **level of indirection**, so the user cannot “guess” open files to bypass access permissions.

Sharing a File - Cont.

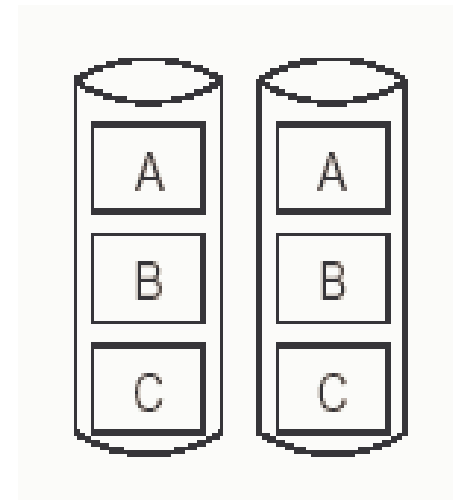


RAID Structure

- **Problems with disks**
 - Data transfer rate is limited by serial access.
 - Reliability
- **Solution to both problems:
Redundant arrays of inexpensive disks (RAID)**
- In the past, RAID (combination of cheap disks) was an alternative for large and expensive disks.
- Today: RAID is used for higher reliability and higher data-transfer rate.
- So the ‘I’ in RAID stands for “independent” instead of ‘inexpensive’.
 - So RAID now stands for **Redundant Arrays of Independent Disks.**

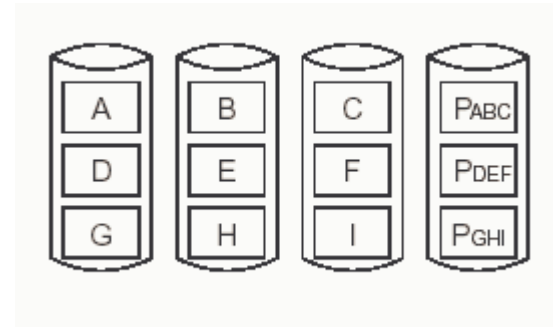
RAID Approaches

- *RAID 1*: mirroring —there are two copies of each block on distinct disks.
- This allows fast reading (you can access the less loaded copy), but wastes disk space and delays writing.



RAID Approaches – Cont.

- *RAID 3*: parity disk — data blocks are distributed among the disks in a round-robin manner.
- For each set of blocks, a parity block is computed and stored on a separate disk.
- If one of the original data blocks is lost due to a disk failure, its data can be reconstructed from the other blocks and the parity.

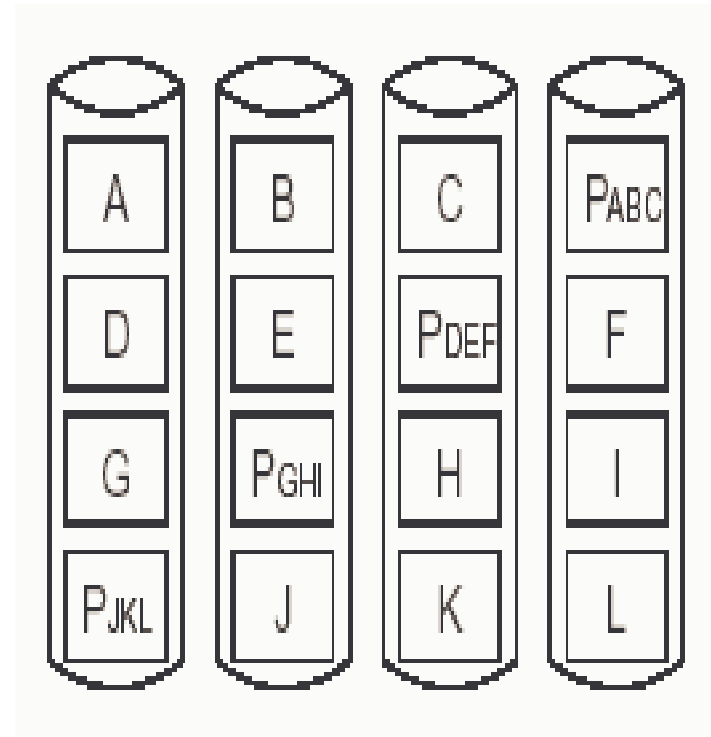


Example:

disk 1	disk 2	disk 3	parity
0110	1000	0010	1100

RAID Approaches - Cont

- *RAID 5*: distributed parity — in RAID 3, the parity disk participates in every write operation (because this involves updating some block and the parity), and becomes a bottleneck.
- The solution is to store the parity blocks on different disks.

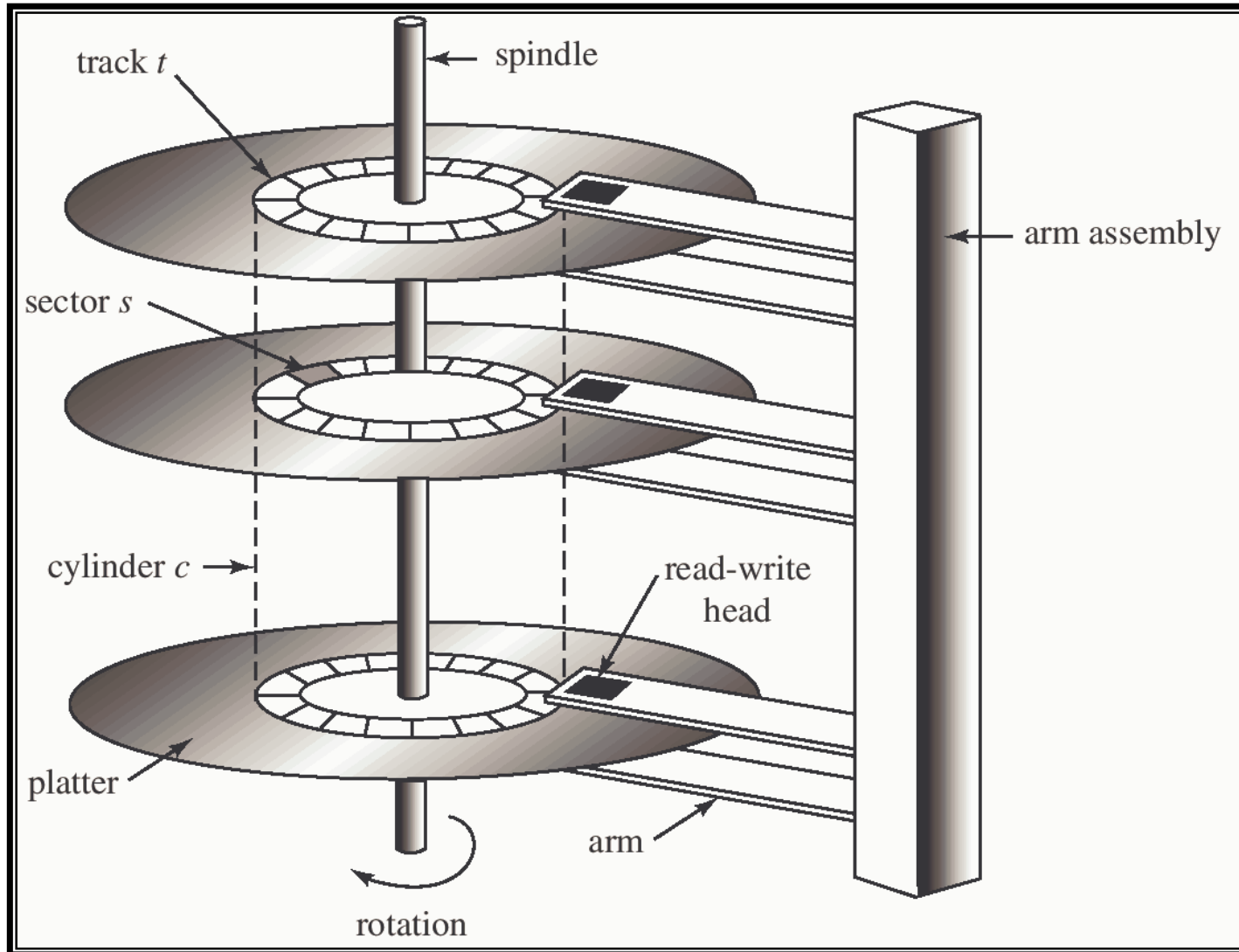


I/O Management and Disk Scheduling

Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the logical block is the smallest unit of transfer.
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially.
 - Sector 0 is the first sector of the first track on the outermost cylinder.
 - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

Moving-Head Disk Mechanism



Disk Scheduling

- The operating system is responsible for using hardware efficiently
 - having a fast access time and high disk bandwidth.
- Access time has two major components:
 - *Seek time* is the time for the disk to move the heads to the cylinder containing the desired sector.
 - *Rotational latency* is the additional time waiting for the disk to rotate the desired sector to the disk head.
- Goal: Minimize seek time and maximize disk bandwidth
- Seek time \approx seek distance
- **Disk bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

Disk Scheduling - Cont.

- Whenever a process needs I/O from the disk, it issues a system call to the OS with the following information.
 - Whether the operation is input or output
 - What is the disk address for the transfer
 - What is the memory address for the transfer
 - What is the number of bytes to be transferred.
- Several algorithms exist to schedule the servicing of disk I/O requests.
- We illustrate them with a request queue (0-199).

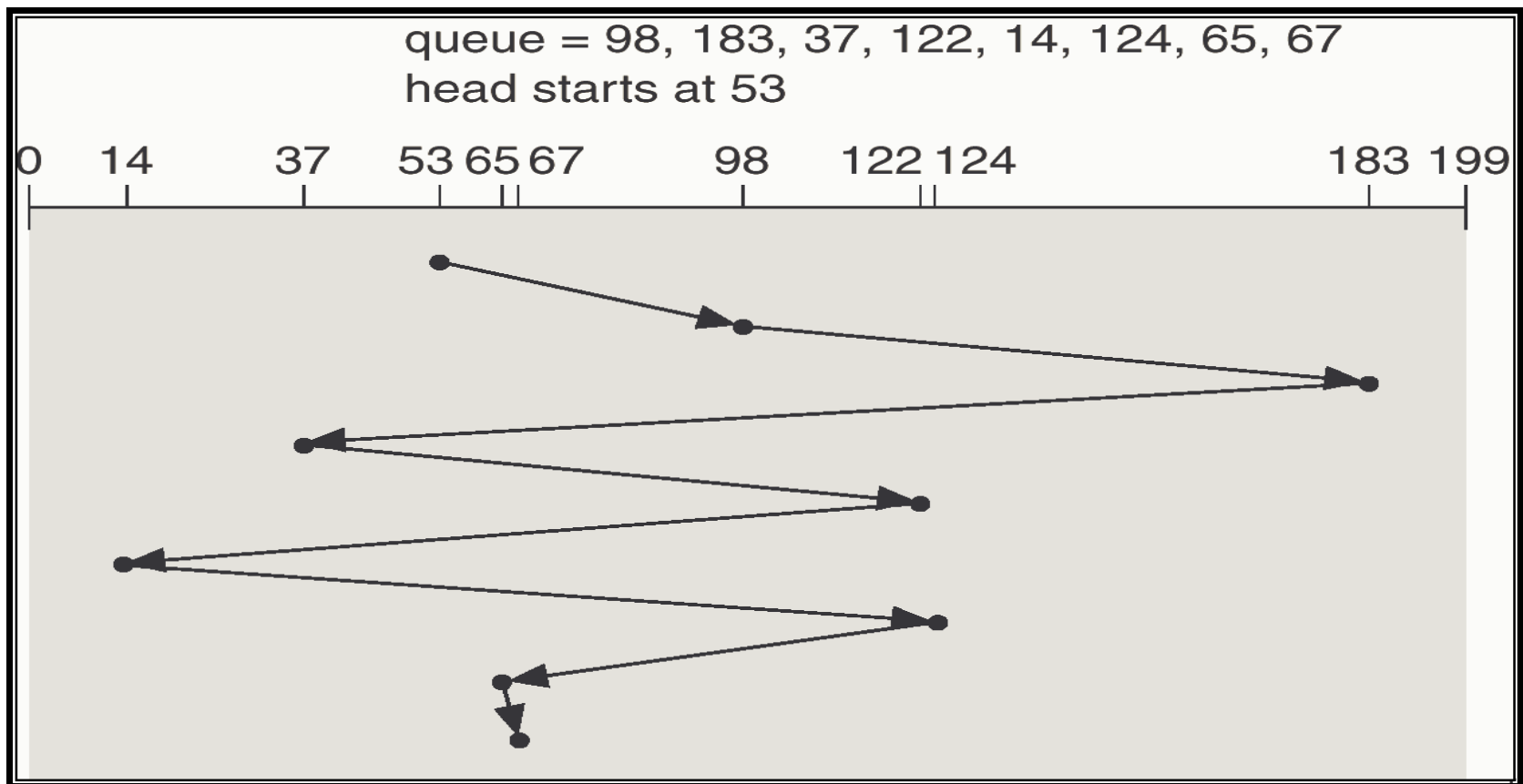
98, 183, 37, 122, 14, 124, 65, 67

Head pointer 53

- First Come First Serve

FCFS

- Advantage: fair
- Disadvantage: slow
- Illustration shows total head movement of 640 cylinders.

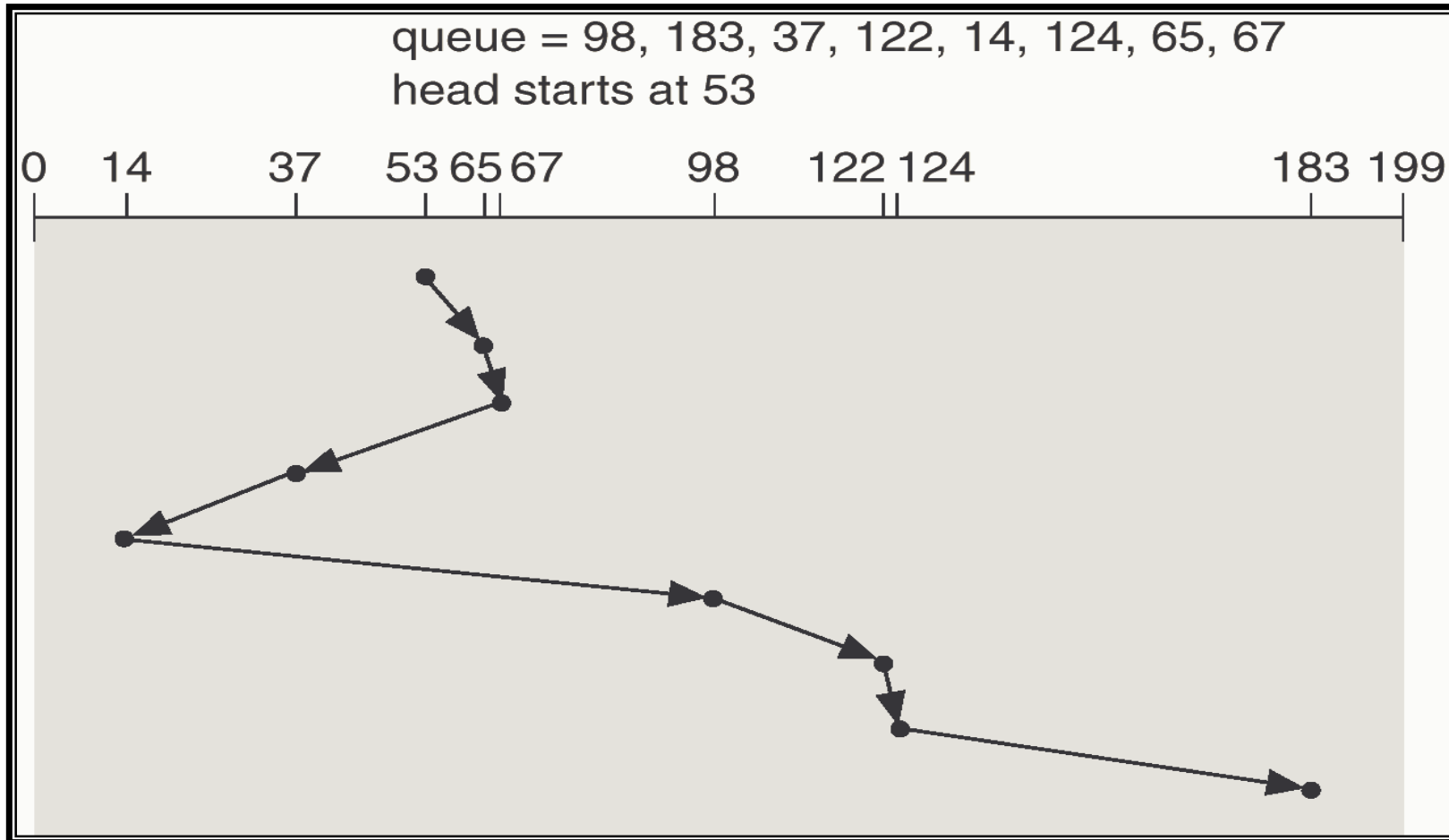


Shortest-Seek-Time-First (SSTF)

- Selects the request with the minimum seek time from the current head position.
- SSTF scheduling is a form of SJF scheduling
- May cause starvation of some requests.
- Illustration shows total head movement of 236 cylinders.

SSTF - Cont.

Total head movement=236 cylinders



Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural advantage over FCFS
- There are other algorithms that perform better for systems that place a heavy load on the disk (reduce chance to starvation).
- Performance depends on the number and types of requests.
- Requests for disk service can be influenced by the file-allocation method.

Summary

- How files are stored on disk
 - The i-node data structure
- OS File operations and their implementation
- The tables used by the OS file system
- RAID
- Disk scheduling algorithms