File System Implementation

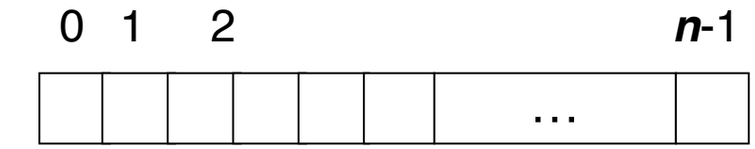
- **Ch14.5: Free-Space Management**
    - Bitmap
    - Linked List
    - Grouping
    - Counting
    - Cylinder groups
- Ch14.2: In-Memory File System Structure
    - C.7.4 Mapping a File Descriptor to an Inode

Free Space Management

- Whenever a new block is requested for a file we need to check to see if there is an empty block. How do we keep track of all the empty blocks?
- File system maintains free-space list to track available blocks/ clusters
- There are lots of them – my 6TB disk with 4KB blocks has about 1.6 billion blocks.

Bitmap

- We can maintain a bitmap (or bit vector).
  - Each bit represents a used or free block (1 represents free)
  - e.g., n blocks

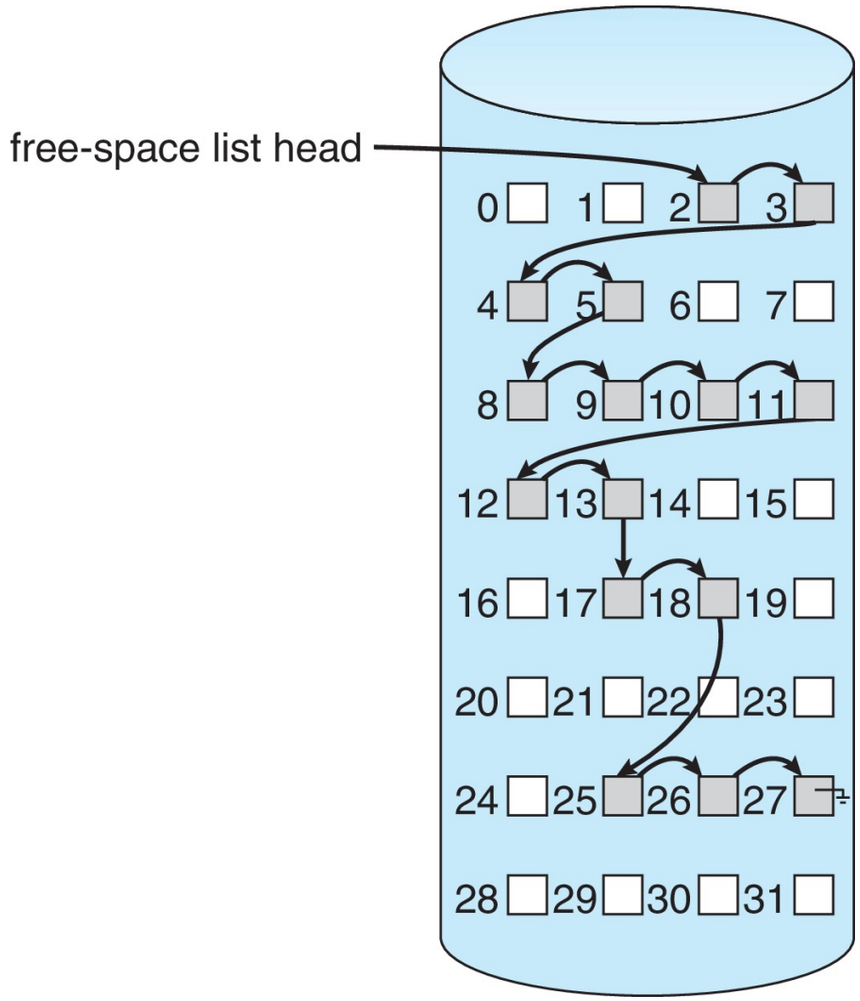


$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{block}[i] \text{ free} \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

- Disadvantages
  - Space: For my disk this takes up about 200 Megabytes of space.

Linked Free Space List on Disk

- Linked list (free list): link the free ones together like one large empty file using the linked allocation method.
- With a linked list it is trivial to find the first free block.



- Advantage
  - No waste of space
  - No need to traverse the entire list (if # free blocks recorded)
- Disadvantages
  - Cannot get contiguous space easily
  - Inefficient if we want several blocks at a time

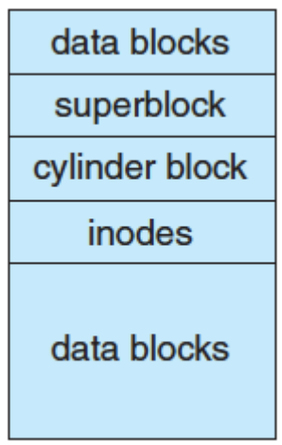
Free Space Management

- Grouping
  - Modify linked list to store address of next n-1 free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)
    - The first n–1 of these blocks are actually free.
    - The last block contains the addresses of another n free blocks

- Counting
  - Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering
  - Keep address of first free block and count of following free blocks, i.e., length
  - Free space list then has entries containing addresses and counts

E.g., UNIX 4.3BSD
(C.7.7 Layout and Allocation Policies)

- Early versions of UNIX maintained a device wide free list of blocks in a stack, whereby the next allocated block is the most recently returned one.
  - What are the consequences of this?
- Also all inodes were stored in one place. What are the consequences of this?
- **Cylinder groups** were introduced in order to improve file access and reliability.
- Cylinder groups - one or more consecutive cylinders (disk head seek time is minimised so access to all blocks in the same group is fast).
  Inodes and free lists stored with each cylinder group.
  - Disk accesses within the cylinder group require minimal disk head movement.
  - Each cylinder group has a copy of the superblock for the "*file system*" (UNIX terminology for a partition or disk device.)
  - Tries to keep blocks from the same file within the same cylinder group.
  - But larger files are split over different cylinder groups.



In-Memory File System Structure

- Now that we know how the information can be represented on the disk we need to know about the data structures the OS maintains when we use a file.

System wide open file table: contains a copy of the FCB of each file and other info

- The system must keep track of all open files.
- Information from the on-disk file control block (these must be kept consistent).
- Which processes are accessing the file?
- How is the file being accessed?

Process open file table: contains pointers to appropriate entries in system-wide open-file table as well as other info

- A pointer to the system open file table.
- Current file position (for sequential reading or writing). May be in system wide table.
- A pointer to the buffer being used for this file by the process.

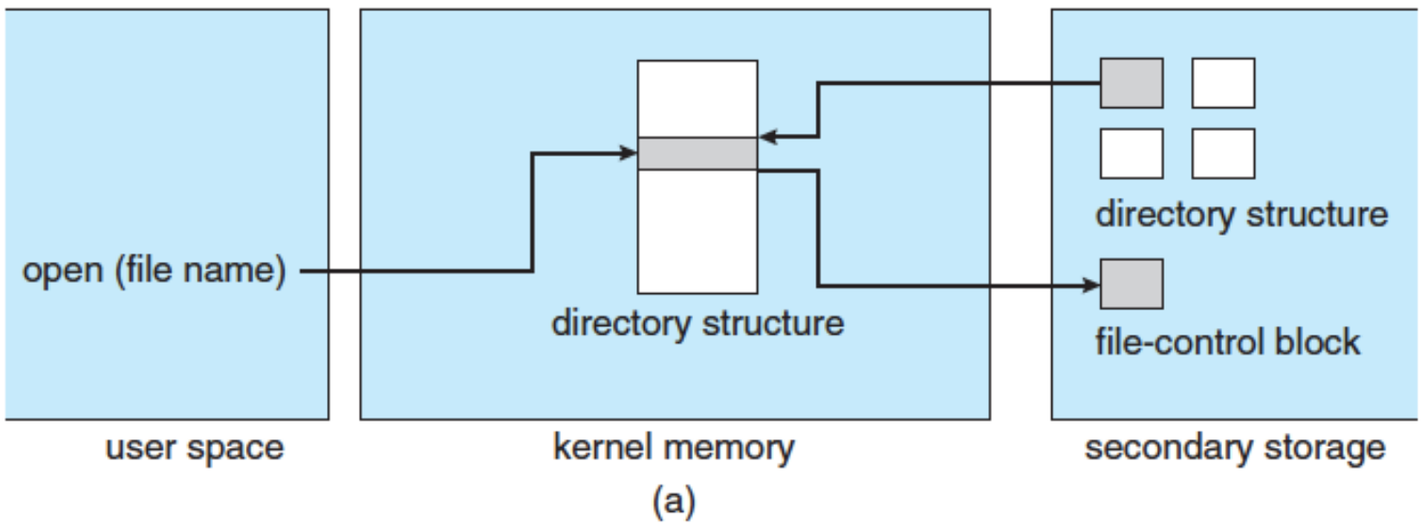
The file buffer: hold data blocks from secondary storage

- Data is read in block (or cluster) amounts.

In-Memory File System Structure

Open()

- searches for the file in the in-memory **directory structure** with that name
- verifies that the **process** has access rights to use the file in the way specified
- records the fact that the file is open (in the **system-wide open file table**) and which process is using it
- constructs an entry in the **process open file table**
- allocates a buffer for file data
- returns a **pointer** (file handle or file descriptor) to be used for future access



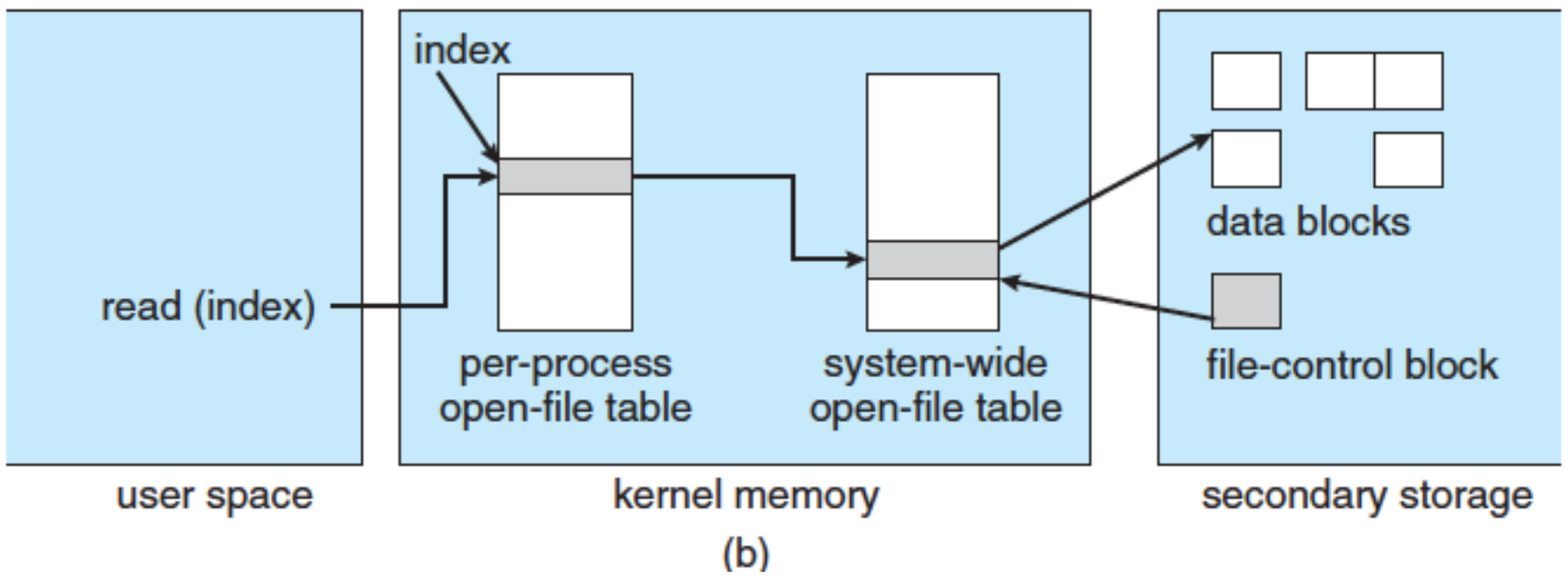
(a)

Most systems require some open call to make the connection between a **process** and a **file**.

In-Memory File System Structure

Open() and Read()

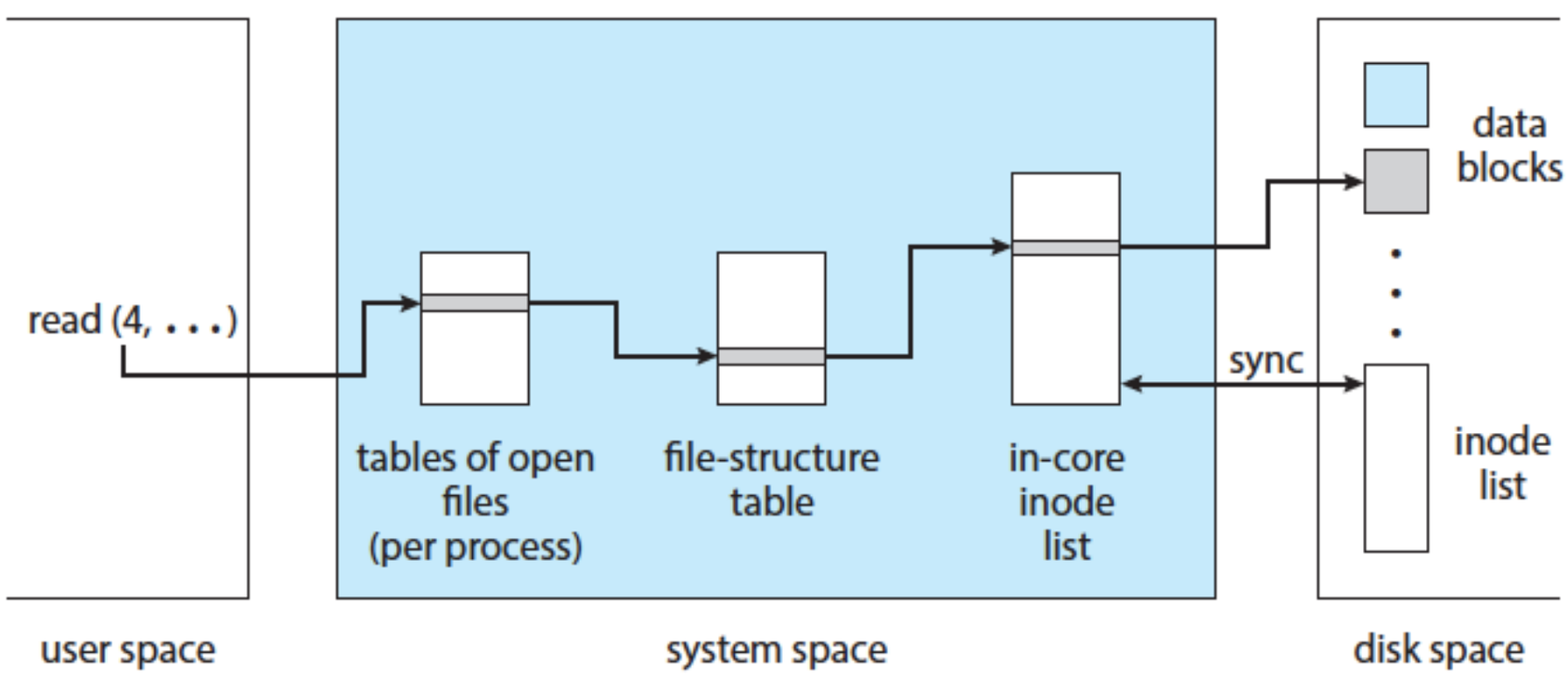
- the on-disk **file control block** (FCB) and **data blocks**
- the memory copy of the FCB, this includes the reference count
- the **system wide open file table** (file-structure table), this actually stores one entry for every time the file was opened
- the **process open file table**, just an array of pointers to the file-structure table
- data from read eventually copied to specified user process memory address



(b)

E.g, UNIX Runtime File Structure

- read() -> process -> file -> inode -> data
- File structures are inherited by the child process after a fork(), so several processes may share the same offset location for a file



E.g., UNIX Fork Interaction

```
# twice.py
import os

file = open("temp", "w")
file.write("before the fork") # 15
#file.flush()

if os.fork() == 0: # in the child
    print("child: {}".format(file.tell()))
    print("child: {}".format(file.tell()))
else: # in the parent
    print("parent: {}".format(file.tell()))
    print("parent: {}".format(file.tell()))
file.close()
```

Produces the following output (without flushing):

```
$ python3 twice.py
parent: 15
parent: 30
child: 30
child: 30

$ python3 twice.py
parent: 15
child: 30
parent: 30
child: 30

$ python3 twice.py
parent: 15
parent: 15
child: 30
child: 30
```

Takeaway: Be careful with open file when doing a fork

E.g., Opening a File in UNIX

open(filename, type of open)

```
fd = open("OS/test/answers", O_RDWR);
```

- convert filename to an inode – this also copies the on-disk inode into memory (if not already there) and **locks the inode** for exclusive access
- if file does not exist or not permitted access
  return error (ensure inode is not locked)
- allocate system-wide file table entry, points to incore inode, increment the count of open references to the file
- fill per-process file table entry with pointer to system-wide file table entry
- **unlock the inode**
- return the index into the per-process file table entry (known as the file descriptor)

E.g., UNIX Write System Call

data write from

```
write(fd, buffer, count)
```

get file table entry from fd
check accessibility
lock inode <—
while not all written
    if a block doesn't exist for the current position **//request a new block**
        allocate one - updates the inode
    if not writing a complete block **//write may require read**
        read the block in
    put the data in the block's buffer
    delay write the block (some later time) **//previous example, more on next slide**
    update file offset, amount written
update file size
unlock the inode <—

Delay Write

- Buffers are shared by the system

  The write doesn't occur until another process is to use the buffer for a different block (LRU replacement) or a daemon process flushes it.

- Advantage

  if a process wants to access this information it is already/still in memory

  e.g. process writes some more and it fits in the same block

- Disadvantage

  information is not written immediately

  usually a daemon process writes data buffers after 30 secs, metadata buffers after 5 secs

  sync command forces buffers to write

E.g., UNIX Append

- If the file has been opened in append mode `O_APPEND` then there is a possible race condition.
- Before each write the file position pointer is moved to the length of the file.
- What if another write changes the length of the file before this write completes?
- The file system must guarantee atomicity for the append write operation.
- That is why there is an append mode for opening a file.

Before Next Time

- Read from the textbook
  - Recovery: Version Control System
  - ZFS: https://en.wikipedia.org/wiki/ZFS