# Project: File System

**Lubomir Bic**
**University of California, Irvine**

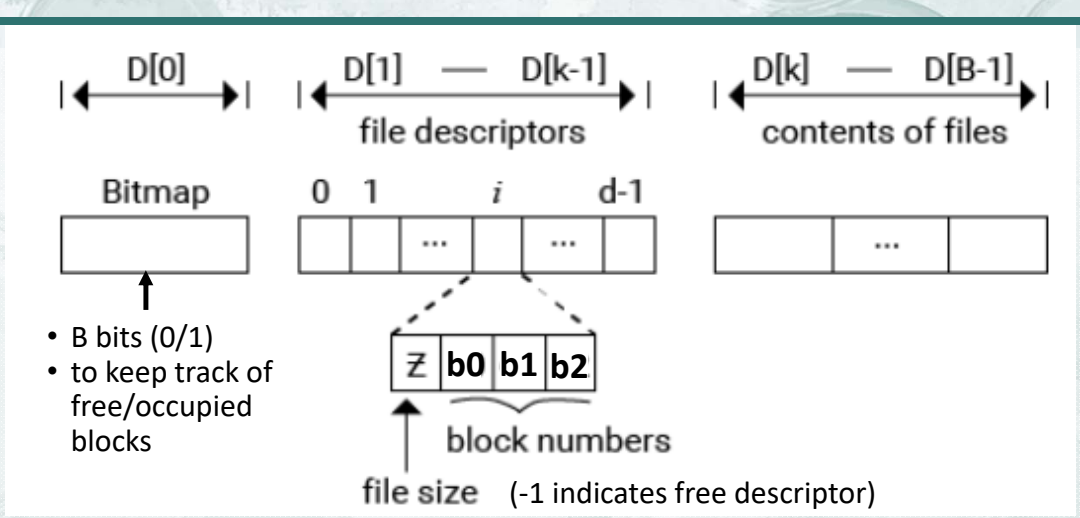
## Project Overview

- Implement a simple **file system** (FS) using an emulated disk
- FS supports commands to:
  - create and destroy files
  - open and close files
  - sequentially access files using buffered read and write operations
  - a single directory and a function to list the directory contents
- Files are mapped onto the disk using fixed index structures in file descriptors
- **Extended** version (not required for this course)
  - hierarchical directory structure
  - expanding indices in file descriptors to support larger file sizes

# The Emulated Disk

- A physical disk consists of one or more rotating magnetic surfaces
  - each surface consists of concentric tracks
  - each track is subdivided into sectors
  - each sector consists of a fixed number of bytes
- Modern disks hide internal complexity:
  - disk is a linear sequence of fixed-size blocks accessed using block numbers
- We **emulate** the disk as a two-dimensional integer array, D[B][512]
  - B: number of blocks
  - 512: block size
- The disk may only be accessed **one block at a time**:
  - *read_block(b)* copies block D[b] into an input buffer, a byte array I[512]
  - *write_block(b)* copies output buffer, a byte array O[512], to disk block D[b]
- All file system blocks are kept on the disk

# Disk Organization



# The User Interface

- The FS supports the following functions:
  - *create()*: create a new file
  - *destroy()*: destroy a file
  - *open()*: open a file for reading and writing
  - *close()*: close a file
  - *read()*: copy a number of bytes from an open file to a main memory area
  - *write()*: copy a number of bytes from a main memory area to an open file
  - *seek()*: change the current position within an open file
  - *directory()*: list all files and their sizes

# The Directory

- A **single** directory organized as a sequence of entries:
- **Symbolic file name**
  - 4-byte field to hold file name (maximum 3 characters plus end of string)
  - 0 indicates that the entry is free
- **Index of the file descriptor**
  - integer in range [1 : d−1]; selects a file descriptor on disk blocks D[1] - D[k−1]
  - descriptor 0 is reserved for directory, can never occur in any directory entry
- Directory: **ordinary** file
  - accessed using same *read*, *write*, *seek* functions as other files
  - described by file descriptor 0
  - created and opened automatically at system initialization
  - must not be closed or destroyed

# Create File

**create(name):**

- check if file already exists
  - using *seek* function, move current position within the directory to 0
  - repeat until end of file is reached
    - read the next directory entry
    - if name field matches parameter *name*, exit with error: duplicate name
- search for a free file descriptor, i
  - starting with disk block D[1], check each descriptor until:
    - −1 is found in the size field of descriptor i, or
    - end of block D[k−1] is reached /* no free descriptor; exit with error: too many files */

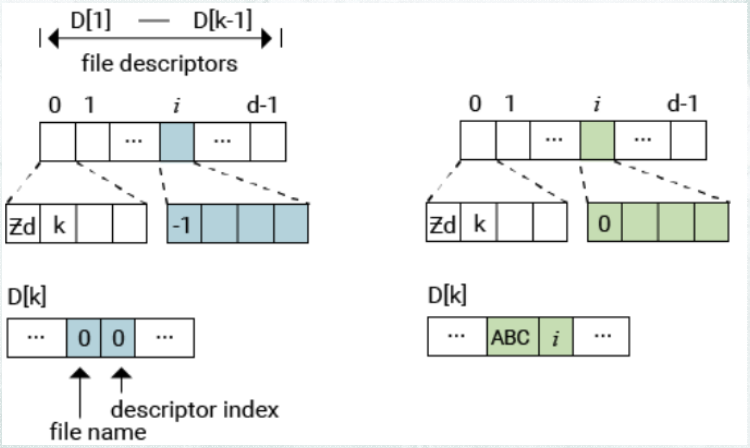
# Create File (cont.)

- if a free descriptor i is found, assign i to new file
  - change size field from −1 to 0 /* the new file is empty */
- search for a free directory entry
  - using *seek* function, move current position within directory to 0
  - repeat
    - read the next directory entry
    - if name field is 0 then:
      - overwrite name field with *name*
      - overwrite index field with i
      - exit with success: file *name* created
    - if end of file is reached, exit with error: no free directory entry found

# Create File - Example

**create(ABC)**

**Before** **After**




- index i is free:
  -1 → 0
- search directory:
- k is first directory block
- contains free entry 0, 0
  0, 0 → ABC, i

# Destroy File

**destroy(name):**

- search directory for file *name*
  - using *seek* function, move current position within directory to 0
  - repeat
    - read next directory entry (name and index i)
    - if name field matches *name*, then
      - mark descriptor i as free by setting size field to −1
      - for each nonzero block in descriptor, update bitmap to free block
      - set all block numbers to 0
      - mark directory entry as free by setting name field to 0
      - exit with success: file *name* destroyed
    - if end of file is reached, exit with error: file does not exist

# The Open File Table

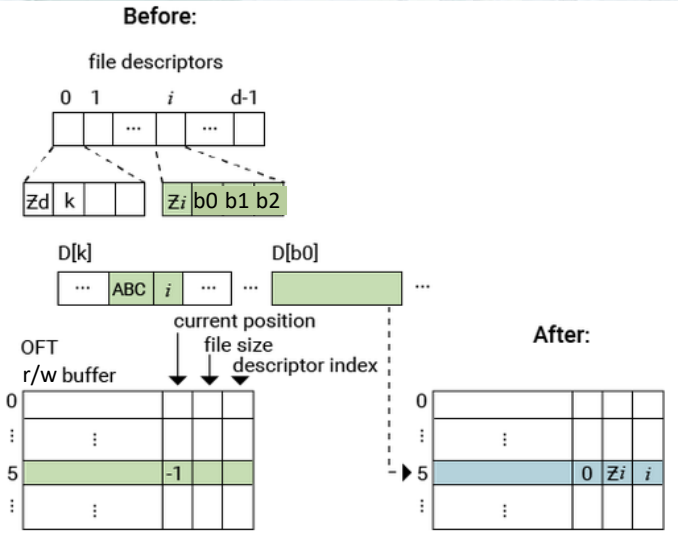
- Before a file can be accessed, it must be opened:
  - directory is searched only once using symbolic name subsequent operations use index
  - uses buffer to avoid repeated accesses to the same block
- **Open file table** *(OFT):* fixed size array, each entry has 4 fields
  - Read/write **buffer**: 512 byte array to hold currently accessed block
  - Current **position**
    - sequential read and write start at current position
    - −1 marks a free OFT entry
  - File **size**: current size of file in bytes
  - File **descriptor** index: index on one of the disk blocks D[1] - D[k−1]

# Open File

**open(name):**

- search directory for a match on file *name*
  - if no match is found, exit with error: file does not exist
- search for free OFT entry, j /* current position = −1 */
  - if no entry found, exit with error: too many files open
- enter 0 into current position of entry j
- copy file size from file descriptor i into entry j
- enter i into file descriptor field of entry j
- if file size = 0, use bitmap to find free block, record block number in descriptor
- otherwise, copy first block of file into the buffer of entry j
- exit with success: file *name* opened at index j

# Open File - Example




**open(ABC):**

- search directory: block k has (ABC, i)
- OFT entry 5 is free
- enter i into index field
- set current position to 0
- copy file size Zi to size field
- copy block b0 to buffer
- return index 5

# Close File

**close(i):**

- write buffer to disk
  - determine (using current position) which block is currently in buffer
  - copy buffer contents to disk block
- update file size in descriptor
  - copy file size from OFT to descriptor
- mark OFT entry as free
  - set current position to −1
- exit with success: file *i* closed

- Assume that i is a valid OFT index of an open file. No error checking.

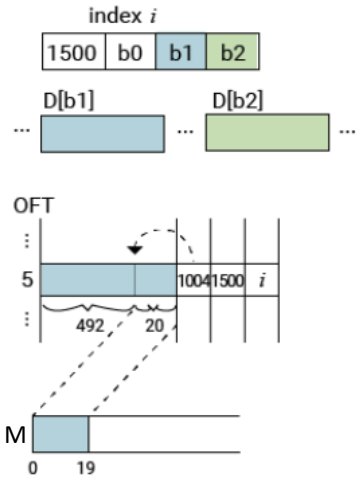
# Read File

**read(i, m, n):** copy n bytes from open file i (starting from current position) to memory M (starting at location M[m])

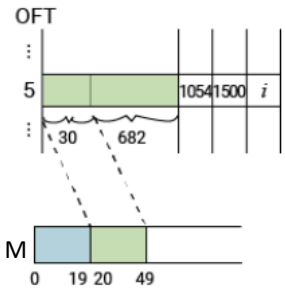
- compute position within buffer corresponding to current position within file
  - file position ranges over all blocks; position in buffer = file position mod 512
- copy bytes from buffer to memory until one of the following occurs:
  - desired count n is reached or end of the file is reached:
    - update current position in OFT
    - exit with success: display all bytes read
  - end of buffer is reached:
    - copy buffer into appropriate block on disk
    - copy next block from disk to buffer
    - continue copying until again one of the above events occurs

# Read File -Example

**After Reading first 20 bytes**




**After Reading remaining 30 bytes**




**read(5, 0, 50):**

- position = 1004:
  - b1 is in buffer (1004/512 = 1)
  - offset = (1004 mod 512) = 492
- copy 20 bytes to M[0] - [19]
- end of buffer reached:
  - copy b1 back to disk
  - copy next block b2 to buffer
  - copy remaining 30 bytes to M[20] - M[49]
- current position = 1004+50 = 1054

# Write File

**write(i, m, n):** copy n bytes starting from M[m] to file i starting at current position

- compute position within buffer corresponding to current file position
- copy bytes from memory to buffer until one of the following occurs:
  - desired count n is reached or maximum file size is reached:
    - if current position > size, update size in OFT and in descriptor
    - exit with success: display number of bytes written
  - end of buffer is reached:
    - copy buffer to appropriate block on disk
    - if next block exists, copy it from disk to buffer
    - otherwise, use bitmap to find free block and record it in file descriptor
    - update bitmap accordingly
    - continue copying until again one of the above events occurs

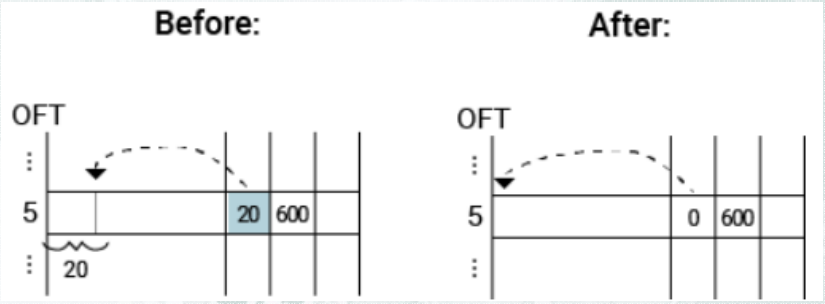
# Seek in File

**seek(i, p):** move current position within open file i to new position p

- if p > file size, exit with error: current position is past the end of file
- determine block, b, containing new position p
- if buffer does not currently contain block b:
  - copy buffer into appropriate block on disk
  - copy block b from disk to buffer
- set current position to p
- exit with success: current position is p

# Seek - Example

**seek(5, 0)**




- new position 0 is less than 600 and thus valid
- current position = 20: buffer contains file block 0 (20/512 = 0)
- new position 0 is within the same block 0: only update current position field

# List Directory

**directory():** display a list of all files and their sizes

- seek to position 0 in directory
- repeat until end of file is reached
  - read next 2 fields
  - if name field is not 0
    - display file name
    - using index field
      - find file descriptor
      - display file size

# System Initialization

At system startup, the following data structures must be created and initialized:

- **Disk D[B]:**
  - D[0] contains bitmap
    - blocks 0 through k−1 are permanently allocated to bitmap and descriptors
    - block k is allocated as the first block of the directory when the directory is opened as part of the initialization
    - remaining blocks k+1 through B−1 are initially free
  - D[1] through D[6] contain d = 192 file descriptors
    - directory = 512*3=1536 Bytes = 284 integers = 192 entries (need 6 blocks)
    - descriptor 0 corresponds to directory and its size field contains 0
    - remaining descriptors are all free (size fields contain −1)
  - all remaining blocks D[7] through D[B−1] contain all zeros

# System Initialization (cont)

- **Memory buffers:**
  - each memory buffer I[512], O[512], M[512] is a byte array of size 512
  - initialized to all zeros
- **Open file table:**
  - OFT[N] is an array of structures
  - read/write buffer within each entry is an array of 512 bytes
  - remaining fields are all integers
  - OFT[0] always corresponds to the open directory; initially all fields are 0
  - remaining OFT entries are all free (marked by −1 in the current position field)

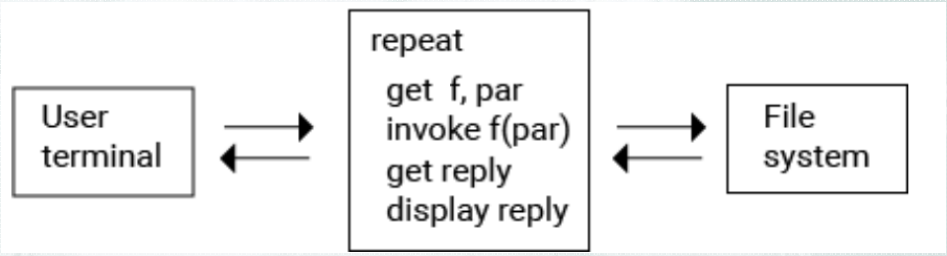
# System Initialization (cont)

Additional functions needed to test the system:

- **init():** restores system to original initial state to allow continuous testing
- **read_memory(m, n):** display contents of memory M[m] - M[m+n−1]
- **write_memory(m, s):** copies string s into memory locations M[m] - M[m+n−1]
- ~~**save(f):** save contents of emulated disk~~
  - ~~f is a text file; if f does not exist, it is created by the function~~
  - ~~function copies the contents of the entire array D[] to file f~~
- ~~**restore(f):** restore state of a previously saved disk~~
  - ~~f is a text file created by a previous *save* function~~
  - ~~function copies contents of file f into array D[]~~

# The Presentation Shell

- **Presentation shell:** allows testing and demonstration of FS
  - repeatedly accepts commands from user terminal (or a file)
  - invokes corresponding FS function
  - displays feedback messages on the screen




- user terminal represents the currently running process

# Shell syntax

| Shell command | Function |
|---|---|
| cr <name> | create(name) |
| de <name> | destroy(name) |
| op <name> | open(name) |
| cl <i> | close(i) |
| rd <i> <m> <n> | read(i, m, n) |
| wr <i> <m> <n> | write(i, m, n) |
| sk <i> <p> | seek(i, p) |

| Shell command | Function |
|---|---|
| dr | directory() |
| in | init() |
| rm <m> <n> | read_memory(m, n) |
| wm <m> <s> | write_memory(m, s) |
| ~~sv <f>~~ | ~~save(f)~~ |
| ~~ld <f>~~ | ~~load(f)~~ |

# Summary of Specific Tasks

- Design and implement emulated disk:
  - data structure D[] to represent the emulated disk
  - disk access functions *read_block()* and *write_block()* used by FS
- Design and implement FS:
  - data structures I[], O[], M[], OFT[]
  - functions *create(), destroy(), open(), close(), read(), write(), seek(), directory()*
  - auxiliary functions *init(), read_memory(), write_memory(),* ~~*save(), load()*~~
- Design and implement shell to test and demonstrate project
- Initialize system at startup as described in Section 3.7
- Test FS using a variety of command sequences to explore all aspects, including detection of errors