Project 4: File System

1. PROJECT OVERVIEW

This project develops a simple file system (FS) using an emulated disk.

The FS supports commands to create and destroy files, to open and close files, and to sequentially access files using buffered read and write operations. In the most basic form, the FS provides only a single directory and a function to list the directory contents.

All files are mapped onto the emulated disk using fixed index structures in file descriptors.

The extended version of the FS implements a hierarchical directory structure and expanding indices in file descriptors to support larger file sizes.

2. THE EMULATED DISK

2.1 Implementation

A physical disk consists of one or more rotating magnetic surfaces. Each surface consists of concentric tracks, each track is subdivided into sectors, and each sector consists of a fixed number of bytes. Modern disks typically hide the internal

complexity by presenting the disk as a linear sequence of fixed-size blocks, which can be accessed one at a time using a block index, b.

In this project, the disk is emulated as a two-dimensional byte array, D[B][512], where B is the number of blocks and 512 is the block size.

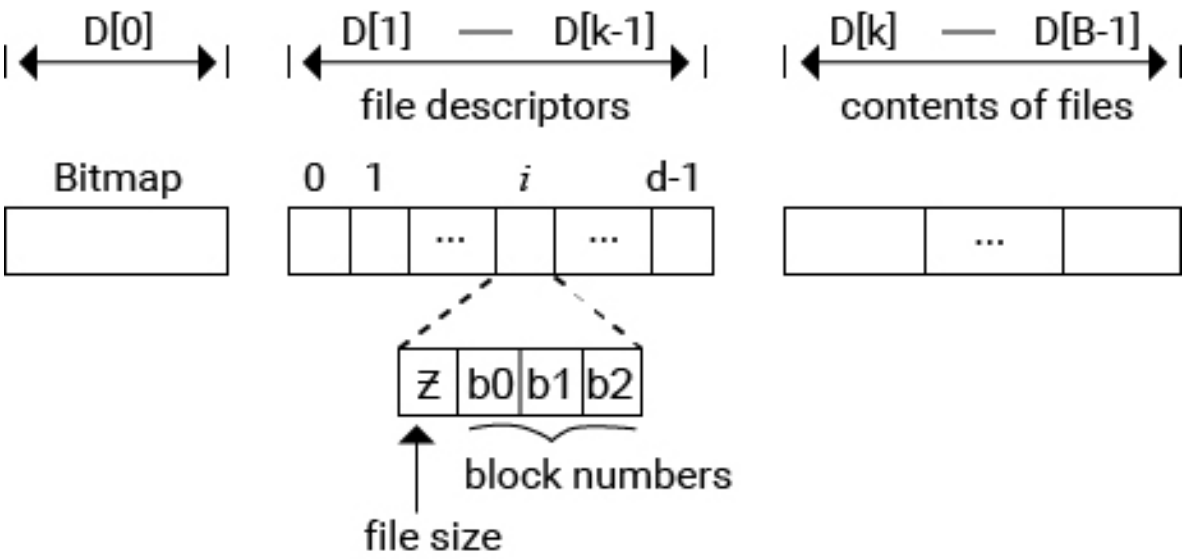
The disk may only be accessed one block at a time:

- The function *read_block(b)* copies the entire block D[b] into an input buffer, implemented as a byte array I[512].
- Similarly, function *write_block(b)* copies the contents of the output buffer, implemented as a byte array O[512], to disk block D[b].

All blocks of the FS are kept on the emulated disk.

2.2 Organization of the Disk

The following diagram shows the layout of the disk:



The first k blocks are reserved as a systems area:

- Block D[0] contains a bitmap, which is an array of B bits, each corresponding to one disk block. The bitmap describes which blocks are free and which are

occupied. (A 0 represents a free block and a 1 represents an occupied block.) The bitmap is consulted by the FS to find a free block whenever a file grows, or to mark blocks as free when a file is destroyed.

- Blocks D[1] through D[k−1] contain an array of d *file descriptors*.
- In the basic version of the FS, a descriptor consists of 4 fields, each holding an integer value:
    - The first field holds the file size (number of bytes); −1 in the size field indicates that the descriptor is free.
    - The remaining 3 fields hold up to 3 disk block numbers allocated to the file. The number of blocks depends on the file size. That is, a newly created file has a size 0 and thus no block is allocated. When the file size is between 1 and 512, the first field, b0, contains a valid number. When the file size exceeds 1024 bytes, all 3 block numbers are needed.
- The number of blocks, k, depends on the number of files that need to be represented. Each file descriptor needs 4 integers (16 bytes) and thus each disk block can accommodate 512/16 = 32 descriptors. To be able to represent d files, the number of blocks needed is k = $\lceil d/32 \rceil$.

The remaining blocks, D[k] through D[B−1], are initially free and can be used to hold the contents of files.

Knowledge Check 1

Availability Start Date:

Jan 19, 2020 @ 10:00 PM PST

Instructions:

NOTE: Please be sure to click the Save Answer button after typing/selecting your answer(s) then "I am Fisnished" to submit for grade.

Your Status:

BEGIN YOUR ATTEMPT (HTTPS://UCI.GRLCONTENT.COM/LAUNCH-ASSESSMENT/4925862?PAGE-ID=WEBCOM-VIEW-

PAGE%2F5%2F11988%2F11985%3FPAGE-ID%3D683217).

3. THE FILE SYSTEM

3.1 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()*: sequentially copy a number of bytes from an open file to a main memory area
- *write()*: sequentially 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

3.2 The Directory

In the basic version of the FS, only one directory exists to keep track of all files. The directory is organized as a sequence of entries, each containing the following information:

- Symbolic file name
    - This is a 4-byte field that holds the file name; a file name is limited to a maximum of 4 characters
    - A 0 in this field is used to indicate that the entry is free; thus all entries are initially free by default
- Index of the file descriptor

- This is an integer in the range 1 through d−1 and is used as an index into the array of file descriptors kept in disk blocks D[1] through D[k−1]
- Descriptor 0 is reserved for the directory itself and thus can never occur in any directory entry

The directory is implemented as an ordinary file, requiring no special operations. The same *read*, *write*, and *seek* functions implemented for ordinary files are used to access and manipulate the directory.

What distinguished the directory from other files are the following features:

- The directory is always described by file descriptor 0 on the disk
- The directory is created and opened automatically at the time of system initialization
- The directory must not be closed or destroyed
- The directory has an initial size of 0 and expands in fixed increments of 8 bytes since each new entry consists of 4 characters followed by an integer

Knowledge Check 2

Availability Start Date:

Jan 19, 2020 @ 10:00 PM PST

Instructions:

NOTE: Please be sure to click the Save Answer button after typing/selecting your answer(s) then "I am Fisnished" to submit for grade.

Your Status:

BEGIN YOUR ATTEMPT (HTTPS://UCI.GRLCONTENT.COM/LAUNCH-ASSESSMENT/4925864?PAGE-ID=WEBCOM-VIEW-PAGE%2F5%2F11988%2F11985%3FPAGE-ID%3D683217).

3.3 Creating and Destroying a File

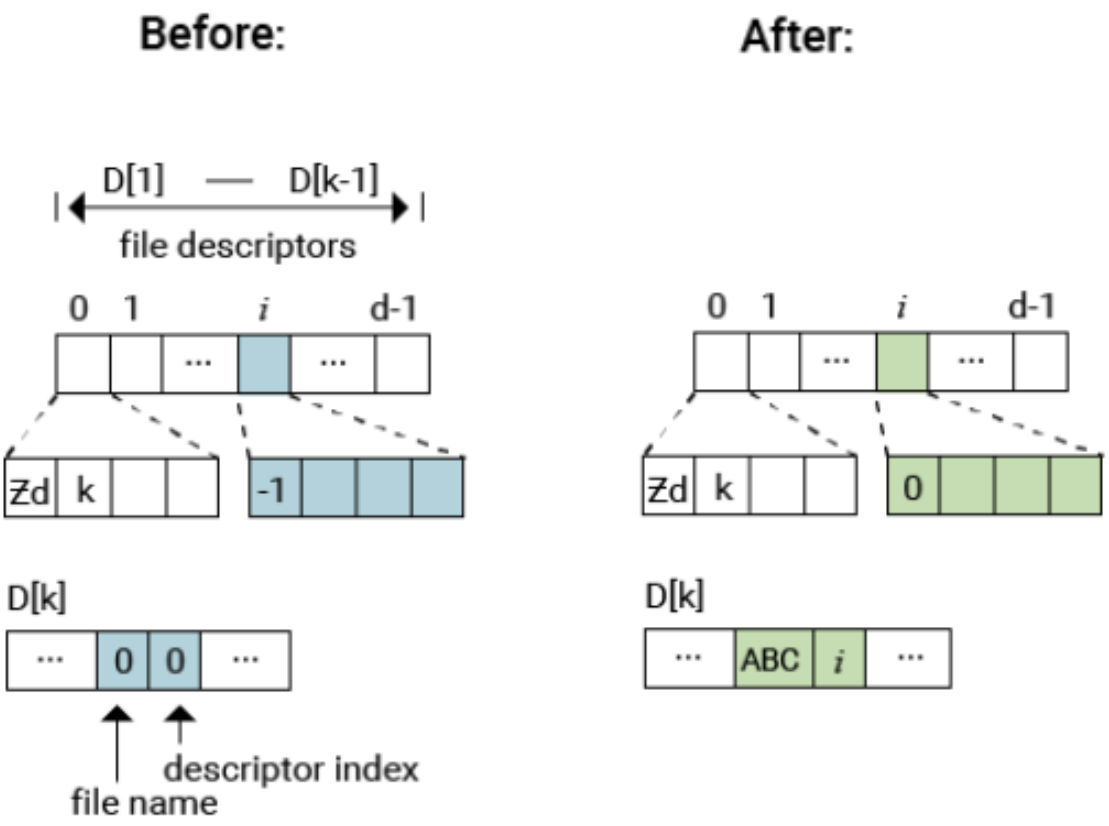
The function *create(name)*, where *name* is a symbolic file name, performs the following tasks:

- Check if file already exists
    - Using the *seek* function (described later), move current position within the directory to 0
    - Repeat until end of file is reached
        - Read the next directory entry
        - If the name field matches the parameter *name*, exit with error: duplicate file 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
        - The end of block D[k−1] is reached, indicating that no descriptor is free; exit with error: too many files
- If a free descriptor i is found, assign the descriptor to the new file
    - Change the size field from −1 to 0, indicating that the new file is empty
- Search for a free entry in the directory
    - Using the *seek* function, move current position within the directory to 0
    - Repeat
        - Read the next directory entry
        - If the name field is 0 then:
            - Overwrite the name field with *name*
            - Overwrite the index field with i
            - Exit with success: file *name* created
        - If current position = 1536, exit with error: no free directory entry found

- Otherwise write a new entry with name and i; Exit with success: file name created

Example

The following diagram shows the changes resulting from creating a new file named ABC: *create(ABC)*



Before

- File descriptor 0, which corresponds to the directory, shows that the first block of the directory is block k
- Block k contains a free entry (file name and descriptor are both 0)
- File descriptor i is free (the size field contains −1)

After

- Descriptor i changes from free to occupied by changing the size field from −1 to 0
- The first available directory entry in block k (containing 0, 0) is allocated. The first field changes from 0 (free) to the file name, ABC. The second field changes from 0 to i, which is the index of the corresponding file descriptor

The function *destroy(name),* where *name* is a symbolic file name, reverses the effect of *create*. Assuming the file ABC is not open, the function performs the following steps:

- Search the directory for a match on the file *name.* If found, mark the entry as free.
    - Using the *seek* function, move the current position within the directory to 0
    - Repeat
        - Read the next directory entry (name field and index field, i)
        - If the name field matches *name*, then
            - Mark descriptor i as free by setting the size field to −1
            - For each nonzero block number in the descriptor, update bitmap to reflect the freed block
            - Set all nonzero block numbers to 0
            - Mark the directory entry as free by setting the name field to 0
            - Exit with success: file *name* destroyed
        - If end of file is reached, exit with error: file does not exist

Knowledge Check 3

Availability Start Date:

Jan 19, 2020 @ 10:00 PM PST

Instructions:

NOTE: Please be sure to click the Save Answer button after typing/selecting your answer(s) then "I am Fisnished" to submit for grade.

Your Status:

BEGIN YOUR ATTEMPT (HTTPS://UCI.GRLCONTENT.COM/LAUNCH-ASSESSMENT/4925866?PAGE-ID=WEBCOM-VIEW-PAGE%2F5%2F11988%2F11985%3FPAGE-ID%3D683217).

3.4 Opening and Closing a File

Before a file can be accessed, it must be opened. Opening a file serves 2 main purposes:

- The directory needs to be searched only once using the symbolic file name. If found, an integer index is returned to the program, which is then used for subsequent read, write, and other operations.
- A disk can only be accessed one block at a time. To avoid repeated accesses to the same block, a buffer is used to hold a copy of the current block in memory.

Opening a file involves making a new entry in an *open file table (OFT)* and returning the index of the OFT entry to the calling program for future use. The OFT is a fixed size array, where each entry has the following form:

- Read/write (r/w) buffer
    - This is an area of 512 bytes, used to hold the currently accessed block
- Current position
    - This maintains the current position within the file and is used by sequential read and write operations; with a maximum file size of 3 blocks, the current position ranges from 0 to 1536
    - −1 is used to mark the corresponding OFT entry as free

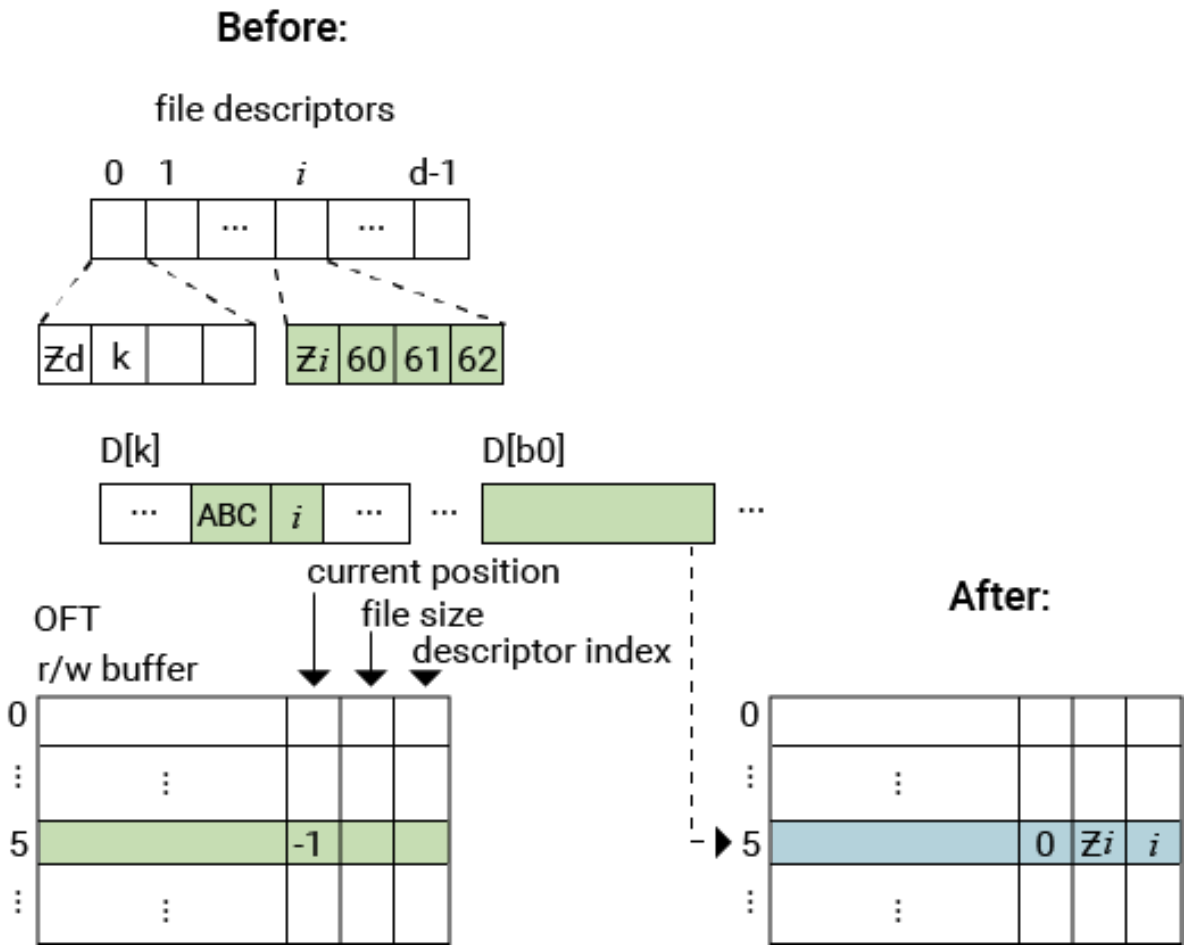
- File size
    - This is the current size of the file, in bytes; initially, the value is copied from the file descriptor but may increase with each write operation
- File descriptor index
    - This is the index of the file descriptor located on one of the disk blocks D[1] through D[k−1]

The function *open(name)*, where *name* is a symbolic file name, performs the following tasks:

- Search the directory for a match on the file *name*
    - If no match is found, exit with error: file does not exist
- Search for a free OFT entry, j (current position = −1)
    - If no such entry is found, exit with error: too many files open
- Enter 0 into the current position of entry j
- Copy the file size from the file descriptor i into entry j
- Enter i into the file descriptor field of entry j
- If file size = 0, then use the bitmap to find a free block and record the block number in the file descriptor
- Otherwise, copy the first block of the file into the r/w buffer of entry j
- Exit with success: file *name* opened at index j

Example

The following diagram shows the changes resulting from opening a file named ABC: *open(ABC)*



Before

- The directory contains an entry (ABC, i) in the first block, k.
- File i occupies 3 blocks: b0, b1, and b2.
- OFT entry 5 is free (current position = −1)

After

- OFT entry 5 is changed as follows:
    - Current position is set to 0
    - The file size Zi is copied from the descriptor into the size field
    - The index i is entered into the index field
    - Block b0 is copied from the disk into the r/w buffer
    - Index 5 is returned to the calling program

The function *close(i)*, where *i* is an OFT index, reverses the effect of *open*. The function performs the following steps:

- Write the r/w buffer to disk
    - Determine which block is currently held in the r/w buffer and copy the current buffer contents to the block
- Update file size in the descriptor
    - Copy the file size from the OFT to the descriptor
- Mark the OFT entry as free
    - Set the current position field to −1
- Exit with success: file *i* closed

3.5 Reading, Writing, and Seeking in a File

When a file is open, it can be *read* or *written* starting from the current position. The *seek* function allows the current position to be changed. The functions operate on a main memory area, implemented as a byte array M[512].

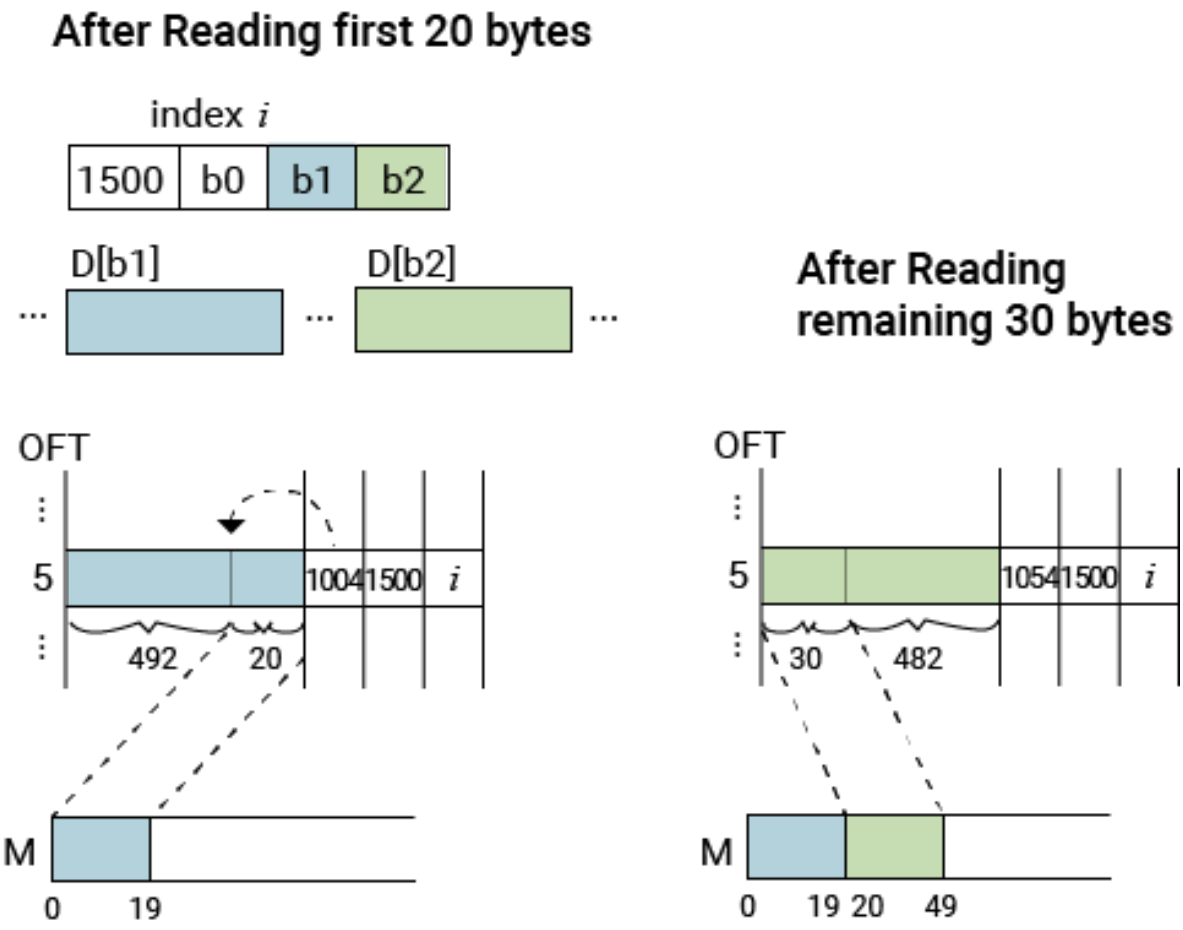
The function *read(i, m, n)* copies n bytes, starting from the current position of the open file i, to memory M starting at location m. The function performs the following tasks:

- Compute the position within the r/w buffer that corresponds to the current position within the file
    - The current position ranges over all blocks of the file; taking the file size modulo 512 maps the file position onto the block currently in the r/w buffer
- Copy bytes from the r/w buffer, starting at the current position, into memory, starting at M[m], until one of the following occurs:

- The desired count n is reached or the end of the file is reached:
    - Update current position in the OFT
    - Exit with success: display all bytes read
- The end of the r/w buffer is reached:
    - Copy the r/w buffer into the appropriate block on disk
    - Copy the sequentially next block from the disk into the r/w buffer
    - Continue copying bytes from the r/w buffer into memory until again one of the above events occurs

Example

Assume the current position within the open file at OFT index 5 is 1004. The following diagram shows the changes resulting from reading the next 50 bytes from the file: *read(5, 0, 50)*



- The file at index i consists of 3 blocks: b1, b2, and b3
- The current position within the file is 1004, which means
    - Block b1 is currently in the r/w buffer (1004/512 = 1)
    - The offset within the r/w buffer is (1004 mod 512) = 492
- The read function copies the remaining 512 − 492 = 20 bytes from the r/w buffer into M[0] through M[19]
- Since the end of the buffer was reached before copying all 50 bytes, b1 is copied back to the disk, and the next block, b2, is copied into the r/w buffer
- The function copies the remaining 30 bytes into memory M[20] through M[49]
- The current position is incremented by 50 to 1054

The function *write(i, m, n)* copies n bytes from memory M starting at location m to the open file i, starting at the current position. The function performs the following tasks:

- Compute the position within the r/w buffer that corresponds to the current position within the file
- Copy bytes from memory, starting at M[m], to the r/w buffer, starting at the current position, until one of the following occurs:
    - The desired count n is reached or the maximum file size is reached:
        - Update current position in OFT
        - If current position > size, then update size in the OFT and in the descriptor
        - Exit with success: display number of bytes written
    - The end of the buffer is reached:
        - Copy the r/w buffer into the appropriate block on disk
        - If the sequentially next block exists, then copy it from the disk into the r/w buffer
        - Otherwise, use the bitmap to find a free block and record it in the file descriptor
        - Update the bitmap accordingly
        - Continue copying bytes from the memory into the r/w buffer until again one of the above events occurs

Knowledge Check 4

Availability Start Date:

Jan 19, 2020 @ 10:00 PM PST

Instructions:

NOTE: Please be sure to click the Save Answer button after typing/selecting your answer(s) then "I am Fisnished" to submit for grade.

Your Status:

BEGIN YOUR ATTEMPT (HTTPS://UCI.GRLCONTENT.COM/LAUNCH-ASSESSMENT/4925868?PAGE-ID=WEBCOM-VIEW-PAGE%2F5%2F11988%2F11985%3FPAGE-ID%3D683217).

Knowledge Check 4b

Availability Start Date:

Jan 19, 2020 @ 10:00 PM PST

Instructions:

NOTE: Please be sure to click the Save Answer button after typing/selecting your answer(s) then "I am Fisnished" to submit for grade.

Your Status:

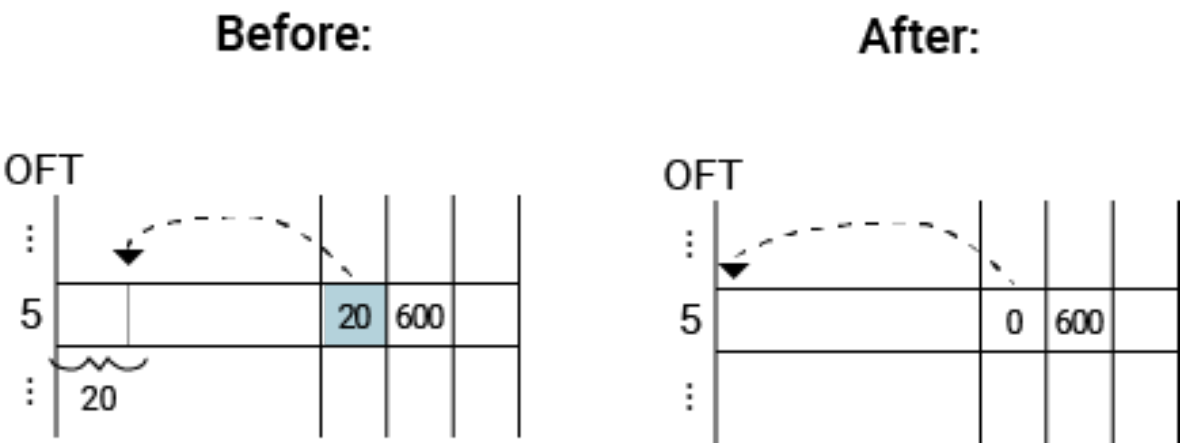
BEGIN YOUR ATTEMPT (HTTPS://UCI.GRLCONTENT.COM/LAUNCH-ASSESSMENT/4925869?PAGE-ID=WEBCOM-VIEW-PAGE%2F5%2F11988%2F11985%3FPAGE-ID%3D683217).

The function *seek(i, p)* moves the current position within an open file at index i to a new position, p. The function performs the following tasks:

- If p > file size, exit with error: current position is past the end of file
- Determine the block, b, containing the new position p
- If the r/w buffer does not currently contain block b:
    - Copy the r/w buffer into the appropriate block on disk
    - Copy block b from the disk into the r/w buffer
- Set current position to p
- Exit with success: current position is p

Example

The following diagram shows the changes resulting from rewinding the current position within a file to 0: *seek(5, 0)*



- File size is 600 bytes; the new position, 0, is less than 600 and thus valid
- The current position is 20 and thus the r/w buffer contains block 0 of the file (20/512 = 0)
- The new position, 0, is within the same block 0 and thus only current position field needs to be updated.

3.6 Listing the Directory

The function *directory()* displays a list of all files and their sizes. The function performs the following tasks:

- Seek to position 0 in the directory
- Repeat until the end of file is reached
    - Read the next 2 fields
    - If the name field is not 0
        - Display file name
        - Using the index field, find file descriptor and display the file size

3.7 System Initialization

When the system starts, the following data structures must be created and initialized:

- The disk D[B]:
    - D[0] contains the bitmap
        - Blocks 0 through k−1 are permanently allocated to the bitmap and descriptors
        - Block k is allocated as the first block of the directory when the directory is opened as part of the initialization
        - The remaining blocks k+1 through B−1 are initially free
    - D[1] through D[6] contain the d = 192 file descriptors
        - Descriptor 0 corresponds to the directory and its size field contains 0
        - The remaining descriptors are all free (all size fields contain −1)
    - All remaining blocks D[7] through D[B−1] contain all zeros
- Memory buffers
    - Each of the memory buffers I[512], O[512], M[512] is a byte array of size 512, initialized to all zeros
- Open file table
    - The OFT[N] is an array of structures
    - The r/w buffer within each entry is an array of 512 bytes
    - The remaining fields (current position, file size, and descriptor index) are all integers
    - OFT[0] always corresponds to the open directory. Initially, all fields are 0
    - The remaining OFT entries are all free (marked by -1 in the current position field)

In addition to the FS functions described so far, several auxiliary functions must be implemented to manage the system.

- *init()* restores the system to the original initial state. The purpose of *init* is to allow continuous testing of the system without having to repeatedly terminate and restart the program

The following 2 functions are used to examine and to change the contents of the memory area M.

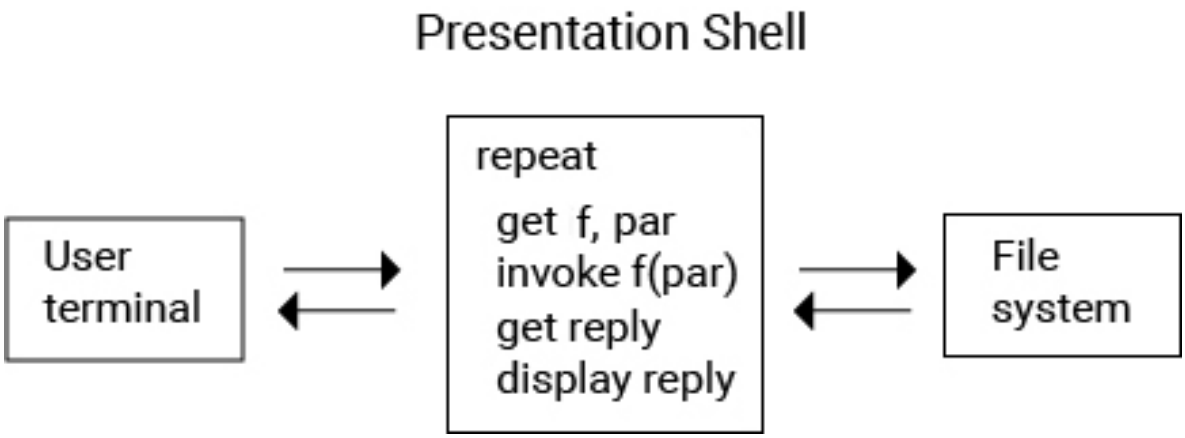
- *read_memory(m, n)*
    - m is an index into the array M and n is the number of bytes
    - The function displays the contents of memory M[m] through M[m + n − 1]
- *write_memory(m, s)*
    - m is an index into the array M and s is a string of n characters
    - The function copies the string s into memory locations M[m] through M[m + n − 1]

The following 2 functions jointly allow the emulated disk to retain its content between login sessions:

- *save(f)*
    - f is a text file; if f does not exist, it is created by the function
    - The function copies the contents of the entire array D[] to the file f, thus saving the contents of the emulated disk
- *restore(f)*
    - f is a text file created by a previous *save* function
    - The function copies the contents of file f into the array D[], thus restoring the state of a previously saved disk

4. THE PRESENTATION SHELL

To test and demonstrate the functionality of the manager, a presentation shell must be developed, which repeatedly accepts commands from the user terminal, invokes the corresponding function of the FS, and displays feedback messages on the screen to reflect the behavior of the system. The following diagram illustrates the basic structure:



Using the above organization, the user terminal represents the currently running process. That means, whenever a command is entered, it is interpreted as if the currently running process issued the command. The presentation shell invokes the corresponding function, f, of the FS, and passes to it any parameters (par) supplied from the terminal. The function invocation results in some changes to the disk D, the memory buffers I, O, M, and/or the OFT. Each function then returns a message to the shell, indicating the success or failure or the last operation. The shell displays the message on the screen.

To simplify the interactions with the shell, we devise a syntax similar to the UNIX shell, where each line starts with a short command name, followed by optional parameters separated by blanks. The shell interprets each line and invokes the corresponding FS function. The following is a list of commands and the corresponding functions to be invoked:

Shell command	Function
cr <name>	create(name)
de <name>	destroy(name)
op <name>	open(name)

Shell command	Function
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)
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)

5. THE EXTENDED FS

5.1 A Hierarchical Tree-Structured Directory

Instead of a single flat directory, most systems support a hierarchical structure, where each directory entry can point to a file or to another directory. Thus, all directories form a tree with the original top directory as the root.

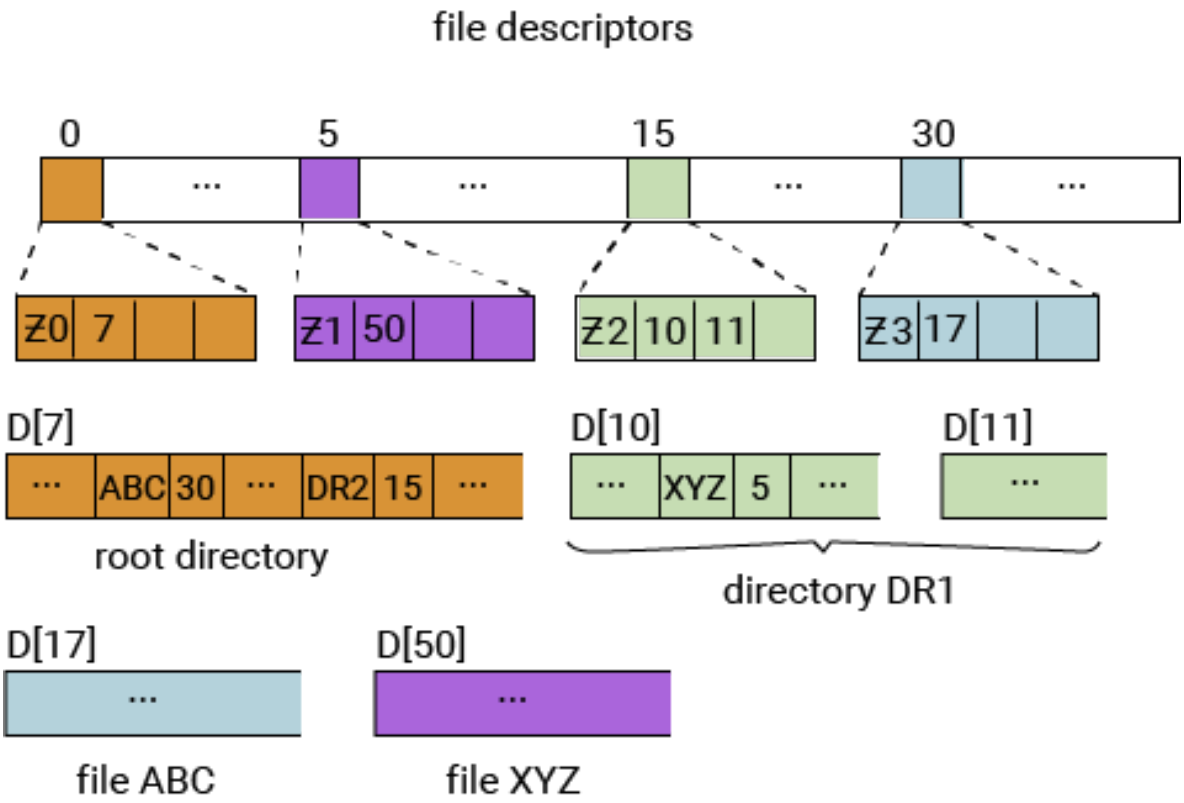
Files and directories are referred to using path names. A path name is the concatenation of simple file names, separated by the delimiter “/.”

We extend the current implementation by allowing any file to serve as a lower-level directory. Thus, no special operations are needed to create, search, or otherwise manipulate directories.

Example

Assume that the root directory contains entries for files ABC and DR1. ABC is treated as an ordinary file, but DR1 is interpreted as a directory. DR1 contains an entry for another file, XYZ.

The following diagram illustrates the representation of the above hierarchy:



- Disk blocks D[1] through D[6] contain 4 file descriptors.
- Descriptor 0 corresponds to the root directory, with size z0 and a single allocated block 7.
- Block 7 shows 2 file entries.
    - File ABC is defined by descriptor 30, which points to block 17, where the file contents are stored.
    - Path name /ABC refers to the file.
    - File DR1 is defined by descriptor 15, which points to blocks 10 and 11. The file DR1 is interpreted as a directory: Block 10 contains an entry for a file XYZ, defined by the descriptor 5. This descriptor shows that file XYZ has size z1 and consists of one block, 50.
    - Path name /DR1/XYZ refers to the file.

Required extensions

To support a tree-structured directory, the following extensions are necessary:

- File names
    - The functions *create*, *destroy*, and *open* must accept pathnames of the form /name1/name2/ . . . , where name1 is a file name in the root directory and each subsequent name is a file name in the next lower directory
    - Only absolute path names are supported (starting with the root directory) and thus all names start with “/”
    - The functions must be extended to follow the path name to the corresponding file before performing the specific operation
- The *destroy* function must be extended further to recursively destroy the entire subtree designated by the path name. For example, destroying DR1 above would also destroy all files listed in DR1, including XYZ
- The *directory* function must be extended to recursively display the files in all subdirectories
- The command language of the shell must be extended to accept path names for the above functions

Knowledge Check 5

Availability Start Date:

Jan 19, 2020 @ 10:00 PM PST

Instructions:

NOTE: Please be sure to click the Save Answer button after typing/selecting your answer(s) then "I am Fisnished" to submit for grade.

Your Status:

BEGIN YOUR ATTEMPT (HTTPS://UCI.GRLCONTENT.COM/LAUNCH-ASSESSMENT/4925871?PAGE-ID=WEBCOM-VIEW-PAGE%2F5%2F11988%2F11985%3FPAGE-ID%3D683217).

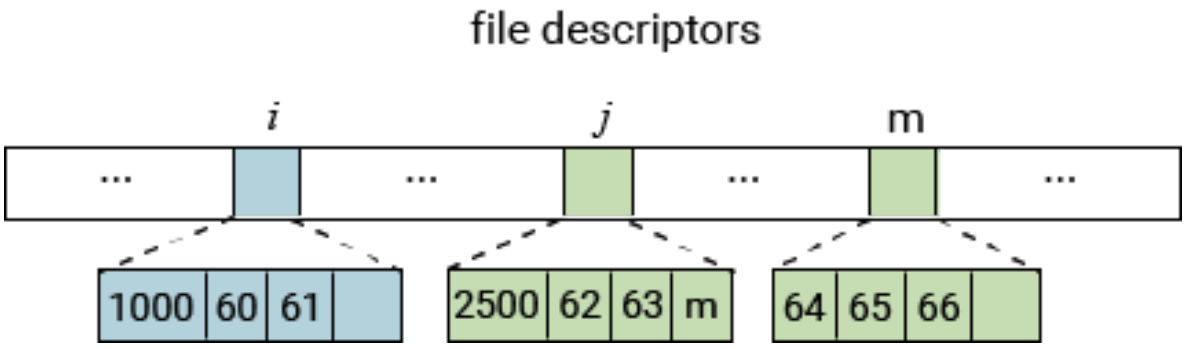
5.2 Expanding Index Structure

The disk allocation scheme implemented so far uses a flat list of up to 3 blocks for each file. The block numbers are recorded directly in the file descriptor. A variation of this simple index structure is an *expanding index*. The first n blocks of each file are recorded directly in the descriptor as before. The last entry points to a secondary index, which contains additional disk blocks. As a result, a short file can be accessed efficiently using the direct block numbers. Longer files can be supported but require an indirect access via the secondary index.

Example

Assume that each file descriptor consists of 4 fields: the file size, 2 direct block numbers, and a secondary index. The secondary index points to another descriptor, which contains up to 4 additional block numbers.

The following diagram illustrates the concept:



- The file at index i contains 1000 bytes, which fit into the 2 direct blocks: b0 and b1. Thus, all accesses to that file require only the index i.

- The file at index j contains 2500 bytes, which require 5 blocks. The first 1024 bytes are accessed directly in blocks b2 and b3. Accessing the remaining bytes requires indirect access via the expanded descriptor at index m. This contains the remaining 3 block numbers: b4, b5, and b6.

Required extensions

To support the expanding index structure, the *read*, *write*, *seek*, and *close* functions must be extended to understand the 2-level index hierarchy:

- When copying a block between the OFT and memory, the function must consider the file size.
- If the size exceeds 2 blocks, then the relevant block must be located via the secondary index.

Knowledge Check 6

Availability Start Date:

Jan 19, 2020 @ 10:00 PM PST

Instructions:

NOTE: Please be sure to click the Save Answer button after typing/selecting your answer(s) then "I am Fisnished" to submit for grade.

Your Status:

BEGIN YOUR ATTEMPT (HTTPS://UCI.GRLCONTENT.COM/LAUNCH-ASSESSMENT/4925873?PAGE-ID=WEBCOM-VIEW-PAGE%2F5%2F11988%2F11985%3FPAGE-ID%3D683217).

5. SUMMARY OF SPECIFIC TASKS

1. Design and implement the emulated disk, which includes the following:
    - The data structure D[] to represent the emulated disk
    - The 2 disk access functions, *read_block()* and *write_block()*, used by the FS
2. Design and implement the FS, which includes the following:
    - The data structures I[], O[], M[], OFT[]
    - The FS functions *create(), destroy(), open(), close(), read(), write(), seek(), directory()*
    - The auxiliary functions *init(), read_memory(), write_memory(), save(), load()*
3. Design and implement the shell to be able to test and demonstrate the functionality of the project.
4. Initialize the system at startup as described in Section 3.7.
5. Test the FS using a variety of command sequences to explore all aspects of its behavior, including the detection of errors.