Operating Systems (CSL 3030)

Readme file

Lab Assignment	09
Roll No.	B20CS018
Name	Harshita Gupta

Ques1

Command lines: gcc DiskScheduling.c -o scheduling

./scheduling

Approach:

1. The solution to this problem includes 2 approaches. One is a File Allocation system or FAT and the second one is an indexed implementation using i-node. The main disadvantage of linked list allocation is that Random access to a particular block is not provided. To access a block, we need to access all its previous blocks.
2. The File Allocation Table overcomes this drawback of linked list allocation. In this scheme, a file allocation table is maintained, which gathers all the disk block links. The table has one entry for each disk block and is indexed by block number.
3. The file allocation table needs to be cached to reduce the number of head seeks. Now the head can only traverse some of the disk blocks to access one successive block.
4. It simply accesses the file allocation table, reads the desired block entry from there, and accesses that block. This is the way by which random access is accomplished by using FAT. MS-DOS and pre-NT Windows versions use it.
5. Instead of maintaining a file allocation table of all the disk pointers, the Indexed allocation scheme stores all the disk pointers in one of the blocks called an indexed block.
6. The indexed block doesn't hold the file data, but it holds the pointers to all the disk blocks allocated to that particular file. Directory entry will only contain the index block address.
7. We have submitted multiple files for the given approach. They can simply be compiled and executed like a normal c file.

Ques2

Alternative-1 Linked List Implementation using FAT

Command lines: First go to the FAT directory and create the executable using :

Run: make

Run executable: ./a.out

Approach:

1. By invoking the method 'init,' the user function must enter the file size, block size, and initialize the file system.

2. The file system has a fixed number of blocks, which is determined by the file and block sizes. The first block is the superblock, which contains important file system information as well as a bit vector indicating the free blocks.

3. The FAT table is in the first block, the directory information is in the second block, and the other blocks are data blocks.

4. We also keep a file descriptor database that keeps track of the valid file descriptors and pointers for reading and writing to the file.

5. The my open() function creates a file descriptor and opens it. It initializes the file descriptor's read and write pointers. If the data isn't already in the directory, it creates a new file and places it there.

6. The file descriptor, a character buffer containing the material to be read, and the number of bytes to be read are all passed to my read() function. It reads from the file after first going to the current block in the file descriptor table.

7. The file descriptor, a character buffer to be written into the file, and the number of bytes of data to be written are all input parameters to the my write() method. The operation is comparable to the function of my read().

8. The my cat() function prints the file's contents.

9. my copy() accepts a file descriptor (1st argument), a Linux file descriptor (2nd argument), and a flag to indicate the copied file's source and destination. Copy from Linux fd to FAT fd if the flag is 0. If the flag is 1, do the inverse.

10. The file descriptor is invalidated by my close() method.

11. The functions can be called from a user program via an API call. The functions will be contained in a library called "alt.h," which will be provided. To run the application, the user must use Makefile.

12. The header file is called 'alt1.h,' and the function definitions are written in 'alt1.cpp.'

Alternative-2 Indexed implementation using i-node

Command lines: fat: 1. make secfile.o : 2. make inodeexe.a : 3. make a.out

a.out:
- file_submitted.cpp alt2.h libalt2.a
- g++ -g file_submitted.cpp -L. -l alt2

libalt2.a:
- alt2.o
- ar -rcs libalt2.a alt2.o

alt2.o:
- alt2.cpp alt2.h
- g++ -c alt2.cpp

clean:
- rm alt2.o libalt1.a a.out

Approach:

1. The header file defines all of the structures and the API function prototype. The file system is represented as a block structure, with the first block being the superblock, which contains all of the essential information, a list of free inodes, and a pointer to the first free block. The inodes are stored in the following two blocks, and the data is stored in the remaining blocks.

2. We start by executing init(), which fills in the proper values in the fields and allocates RAM to the file system.

3. The input to my mkdir() is a string. It then looks for a match among the blocks pointed to by the inode. If a match is found, the result is -1. Otherwise, a new entry to the block is made by entering the directory's name in the relevant block pointed by DP, SIP, or DIP. In addition, an inode is chosen to hold the information for the

newly created directory.

4. my rmdir() takes a text as input and first searches for the desired directory's inode number. If it is detected, the inode is cleared, as are all subsequent directories and files in the selected directory. The free inode list and free blocks are updated as needed.

5. my chdir() takes a string as an argument and determines whether or not it is the parent directory. It makes the necessary adjustments. If the directory is found, the current directory pointer is changed to reflect the change.

6. my open() accepts a string filename as input and checks to see if the file already exists in the current directory. If this is the case, add a new entry to the FD list to open a new stream. Create the FD table from scratch. If the file cannot be located, create an entry in the current directory's inode. The new file's inode is picked, and then the inode is initialized. A new item is created and initialized in the FD table to correspond to the specific file.

7. With a file descriptor, a buffer, and a count as input, my read() reads the count number of bytes from the file and writes them to the buffer.

8. After receiving bytes from the buffer, my write() takes a file descriptor, a buffer, and a count as input and writes a count number of bytes to the file.

9. my close() invalidates the file descriptor by closing it.