CS342: Operating System

Project 4: File System

Submitted By:

Pradeep Bansal

10010143

ASSIGNMENT 1: File System Implementation.

CHANGED FILES

1. **~/pintos/src/filesys/inode.h**
2. **~pintos/src/filesys/inode.c**
   Changed Functions
   a. bool inode_create (disk_sector_t sector, off_t length).
   b. static disk_sector_t byte_to_sector (const struct inode *inode, off_t pos)
   c. off_t inode_write_at (struct inode *inode, const void *buffer_, off_t size, off_t offset)

   Added Functions
   a. void map_sector(struct inode_disk *disk_inode,disk_sector_t sector_of_inode,size_t logical_sector,disk_sector_t physical_sector)

3. **~/pintos/src/threads/thread.c**
   Changed Functions
   a. void thread_exit (void).

DATA STRUCTURES

1.

```
struct inode_disk
  {
    disk_sector_t start;              /* First data sector. */
    off_t length;                     /* File size in bytes. */
    unsigned magic;                   /* Magic number. */
    /*
          Added Data Structures.
    */
    disk_sector_t index[15];          /* Mapping Table in the Inode.*/
    uint32_t last_index;              /* First Invalid Logical Sector of the File.*/
    bool sector_allocated[128];       /* To know whether the index has been allocated a sector or
not(Useful for double Indirection).*/
    /*
          Added Data Structures Ends.
    */
    uint32_t unused[77];              /* Not used. */
  };
```

2. bool write_beyond_eof=false; **/* To track whether the write is strictly beyond the EOF*/**

Explanation of Data Structures.

disk_sector_t index[15]
This is mapping table for the inode same as UNIX. First 13 pointers (0-12) are direct pointers they point directly to the data blocks of the file. Next pointer (13) is a single indirect pointer and the last one is (14) is a double indirect pointer. (So a total of (512/4 = 128) no of mapping hence a total (128*128 + 128 +12)*Disk_Sector_Size = 8MB (approx.) file can be created on this FileSystem which is precisely the size of the filesystem itself.)

ALGORITHMS

Scheme for File Allocation.

The concept of combined scheme is used for allocating data blocks for the file. Combined Scheme means that the original concept of contiguous allocation is retained but only to the size which is specified at the creation time after that we have used the concept of indexed allocation as in UNIX.

Mapping Function

The map_sector() function in source code is responsible for correctly mapping logical sectors of a file to physical sectors of the file.

First Logical sector number is examined so that it can be anywhere between the three following ranges

1. 0-12 // just map it directly to the physical sector

2. 13-140 // First allocate a sector for the 14(13th index in the array) entry and then update the sector entry by entry as physical sectors are allocated to the file.

3. Greater than 140 // First allocate a sector for the 15(14th index in the array) entry and then use the sector_allocated data structure to know whether the entry has been allocated any sector which will store the actual physical sector for the file.

140 is calculated using DISK_SECTOR_SIZE. But code will handle if DISK_SECTOR_SIZE is changed.

Changing byte_to_sector (const struct inode *inode, off_t pos)

This function serves as the converter between logical sectors to physical sectors. Since the allocation method has changed therefore the converter has to change. This algorithm reads from the index table. If the offset is less than the file's length then the byte_to_sector returns correct physical sector otherwise returns -1.

File Growth

For this the function inode_write_at is changed. If byte_to_sector returns positive number. First normal write is performed till the point the file's EOF is reached. Then we have to see whether the last physical sector is fully filler or not. If it is we fill the sector till size of data sector is exhausted.

Otherwise, we allocate a new sector and write data into it either the sector or the data to be written is exhausted. We continue the above process until the data is fully written.

Special attention is paid when the offset is strictly beyond the EOF of the file. In that case we fill the space between the offset and current EOF with zeros. After that the normal process of writing is restored as mentioned above.

ASSIGNMENT 2: Subdirectory Implementation.

CHANGED FILES

1. ~/pintos/src/filesys/filesys.c
   Changed Functions
   a. bool filesys_create (const char *name, off_t initial_size)
   b. struct file * filesys_open (const char *name)
   c. bool filesys_remove (const char *name)

2. ~/pintos/src/userprog/syscall.c
   Changed Functions
   a. static void syscall_handler (struct intr_frame *f UNUSED)
   Added new Syscalls(mkdir, chdir, isdir, readdir, inumber)

3. ~/pintos/src/filesys/directory.c
   Changed Functions
   a. bool dir_create (disk_sector_t sector, size_t entry_cnt)
      For adding entries "." and ".." for **ROOT** directory.
   b. bool lookup (const struct dir *dir, const char *name, struct dir_entry *ep, off_t *ofsp)
      Added Support for traversing file system tree(by tokenizing for path).
   c. bool dir_add (struct dir *dir, const char *name, disk_sector_t inode_sector,bool is_dir)

4. ~/pintos/src/threads/thread.h
   Changed thread structure to have inode number of the current working directory of this process.
   Changed open_file structure to include support for open directories.

DATA STRUCTURES

1. struct dir_entry

```
 {
   disk_sector_t inode_sector;        /* Sector number of header. */
   char name[NAME_MAX + 1];           /* Null terminated file name. */
   bool in_use;                       /* In use or free? (Useful for deletion).*/
   bool is_dir;                       /* Whether this directory entry is directory or a file.(1-->directory
0-->file).*/
 };
```

2. struct open_file

```
{
        int fd;
        struct file *file;
        struct list_elem elem;
        bool is_dir;            /* Classifies the entry in the open file table of the process.(Whether
it is a file or a directory).*/
        struct dir *dir;        /* It could well be a directory that the process opens.*/
};
```

Explanation of Data Structures.

bool is_dir
Purpose: Whether the directory entry or the open file is directory or a normal file.

struct dir *dir
Purpose: If the entry is a directory in the open file table then this pointer is kept to take us to the directory or else file pointer is kept to take us to file.

disk_sector_t inode_current_directory
Purpose: Keeps the inode number of the current working directory of the process.

ALGORITHMS

Changing lookup() function.

To add the support for traversing through directories we first tokenize the absolute or relative path with "/" as delimiter.
For all tokens we find the entry of the current token in the directory entries of the previous token this we do until we have gone through all the entries when we find the token we change the directory of the current token and take the next token.

Depending on whether whole path exists or not lookup either returns true or false.

Updating Open, Remove and Close System Calls.

Open ()
1. First we check whether the name (argument) is a file or a directory which is checked by checking the is_dir of the dir_entry we recieve by calling lookup.

2. We also check whether the entry is in use or not(implicitly checked by lookup()).

3 .Call dir_open() and filesys_open() when the name to be opened is a directory or a file respectively.

4. Update the Per Process Open File Table accordingly.

5. filesys_open() is changed because the name it receives now can be absolute or relative path and has to be properly tokenized in order to find the file.

Remove ()
1. First lookup is called to actually find the file or directory to be removed
2. Then is_dir field of directory entry is examined.
3. If it is a directory then we perform the following checks in order
   a. 1.It is not the Root directory.
   b. It is not opened by any process.
   c. It contains only those directory entries corresponding to the "." and ".." .
4. If all the above checks are passed then we open the parent directory of this directory by finding entry of ".." in this directory and remove the entry of this directory from the parent directory.
5. Remove all sectors of the directory associated with the inode and other entries (there are just two of them).
6. Otherwise it is a file and we call filesys_remove(), which also has been upgraded to handle pathname(Absolute or relative).

Close ()

1. After finding the appropriate file descriptor we just check the is_dir field and call appropriately the functions dir_close and file_close in case of directory and file respectively.

New System Calls (mkdir, chdir, isdir, readdir, inumber)

mkdir ()

First the usual checks on the user memory accesses are done and then it checked whether the path including the whole but last directory exists or not.
For Example mkdir("/a/b/c") is called then we check whether path /a/b exists or not.

If it does not then we indicate failure and return, Otherwise we perform a lookup and update the parent directory of the new directory being made by making a new entry.

We then make the new directory and then add two entries for "." and ".." respectively in the new directory.

chdir ()

Again the check for violation of user memory accesses and then lookup the path where the process will change to.

If the path is non-existent then we return immediately indicating failure. Otherwise we check if it is a directory or not .If not we return signaling failure (f->eax = 0) Otherwise we change the value of the **inode_current_directory(mentioned in data structures)** to new directory**.**

Isdir ()

Scan through the open file list of the process and find the file descriptor required.
Check the is_dir field if yes return true else false.

Inumber ()

Scan through the open file list of the process and find the file descriptor required.
Check the is_dir field if yes return inode number using the file pointer else return inode number using the directory pointer.

Readdir ()

Scan through the open file list of the process and find the file descriptor required.
Then check whether it is a directory or not .If not we return signaling failure.

Else go through all the directory entries until we find the first directory entry which is not "." or ".." if found **(tracked by entry_found variable in the source code)** we store the NULL terminated filename in the name array which is checked against any invalide user memory accesses and the length (**It should have atleast NAME_MAX+1**) failing which we return signaling failure.
If no entry is present except "." and ".." then we return signaling failure.

SYNCHRONIZATION

There is a single lock for the entire file system. So, no two processes can extend the file at the same time. This is called external synchronization.

RATIONALE

The proposed design has the following advantages and disadvantages.

Advantages

1. **The Unix Like Inode structure allows files or directories to be as large as the filesystem itself.**

Disadvantages

1. **File name lengths have been constrained to be at max 14 characters in length. (Although absolute and relative pathname can be of any length)**
2. **Since we use a combined scheme it could be possible to demand such a large space at the creation time such that the space is not available.**

The rationale behind for persisting with the above disadvantages is to keep the design simple.