CS 342 Design Document
Project 4 – File System

Rajat Kateja (r.kateja@iitg.ernet.in)
Maths and Computing
10012338

TA: Lalatendu Behera
Chetti Prasad

------ DATA STRUCTURES ------

Modified the inoed_disk data structure to

```
struct inode_disk
  {
    disk_sector_t start;              /* First data sector. */
    off_t length;                   /* File size in bytes. */
    unsigned magic;                   /* Magic number. */
    disk_sector_t file_index[15];                 /* For indexed allocation of file */
    uint32_t is_dir;                              /* 0->not an dir 1->is a dir */
    uint32_t unused[109];             /* Not used. */
  };
```

The two new data fields are:

1.) file_index[15] which is used as an indexed file table to map the logical file blocks to actual sector numbers on the disk.
2.) is_dir, this stores whether the inode is that of a directory.

Added the following to struct thread

uint32_t cur_dir, this stored the sector number of the inode corresponding to the current working directory of the thread.

Added the following to struct fd (the structure to maintain open file)

bool is_dir, this identifies whether the open file is a directory.

------ ALGORITHMS ------

File Growth

The original implementation of PintOS does not support growth of file beyond the size specified at the time of creation of the file. This needed to be modified. It also required the implementation of a indexed file table to map the logical file blocks to sector numbers on a disk. This was done using the combined multi level indexing scheme as used in Linux (although allowing a maximum of two level indirection only, which is sufficient for the disk size considered in PintOS.). When a file is created the initial size of the file is passed as a parameter. In the new implementation the first 13 blocks required by the file are allocated contiguously, and after that each block is allocated independently, as per the combined scheme. While allocating the sectors on the disk, the mapping from the logical file block number to the actual sector number is done. All this is done in the inode_create function, which is called during the creation of files as well as directories thus avoiding duplication of code.

Next, the possibility of extending the file had to handled. The file length had to be increased when it was written. This was also implemented in the inode_write_at funciton to handle the case of both files and directories. In the new implementation of this function, after the already present code, it is checked whether the bytes_written (number of bytes actually written in the file) is same as the size (number of bytes that had to be written). If the check fails, (which implies that the end of file has been reached without completing the entire data that had to be written), the remaining data had to be written. For this first it is checked whether the sector which contains the end of file is completely filled or not. If not, that sector is filled with the data from the buffer. After that, if more data is remaining to be written, new sectors from the disk are allocated to this file, and data is written onto them until all the data from the buffer has been written. As and when a new sector is added to the file, it is mapped from to its corresponding logical block number in the file_index in the inode of the file on disk.

Along with all this, to complete the implementation, the function byte_to_sector had to be changed. This function returns the sector number on the disk where the data corresponding to the offset (input parameter) is present. This was required because now the file block sectors are not continuous, and hence the lookup according to the original function would return the wrong sector number. The new implementation, traverses the file_index of the inode, and returns the correct sector number.

A special case to handle was when the write is attempted at a location other than the end of file, which creates a hole in the file. This was done by filling in the hole with zeroes, adding appropriate sectors if required and mapping the sectors to the logical file block numbers.

Subdirectory Structure

The original implementation of PintOS supports only a single root directory and all files are inside this directory only. The new implementation required handling of subdirectories. This required many minor changes. Subdirectories are nothing but files but with some unique data and some unique system calls associated with them. Firstly a variable is_dir had to be kept with all inodes identifying whether the inode corresponds to that of a file or of a subdirectory. The same was also kept in the (per process) open file table list, again to identify whether the open file is a file indeed or a subdirectory, because not all system calls can be implemented on subdirectories. Next, all the function concerned with files such as file_read, file_write, file_open, file_close had to be modified to take in as argument the entire path of the files (either absolute or relative). The path was then parsed breaking it into the directories and the final file. Then the appropriate calls to the built in functions were made. Also appropriate modifications were made to handle the cases of opening/reading/writing/closing of directories using these system calls. The is_dir variable was checked and the calls for directory were made instead of that of a file if is_dir was set to be true.

Next some extra system call were required for the subdirectories. They are as described below:

CHDIR

This changes the current working directory of the process to the one received in the arguments. Firstly some sanity checks are performed on the received argument, and if everything is correct, the cur_dir field of the thread is modified and made to store the inode number of the new current directory.

MKDIR

This system call is used to create new directories. It is the counterpart of the CREATE system call for files. In this function, firstly the path to make the directory is parsed along with some sanity

checks, and then the function dir_create is called. Now in the dir_create function, modifications were required. The earlier implementation was used to create only the root direcory (since it was the only possible directory), but now in the creation of subdirectories, an entry had to be made in the parent directory for this subdirectory. Also two entries are made in this particular directory. These correspond to the current directory and the parent directory ('.' and '..' respectively).

ISDIR

This system call is used to check whether an open file is a file or a directory. This has been easily handled using the is_dir filed in the open files table.

INUMBER

This call is used to return the sector number corresponding to the inode of the direcory. This also has been easily handled using the already present data fields in the inode data sturcture.

READDIR

This system call is used the read the next file/directory name present in the directory passed as argument. This was implemented using the already present function dir_readdir after performing some simple sanity checks.

------ SYNCHRONIZATION ------

There were no special race conditions to consider in this project and hence no extra synchronization efforts were required.

------ RATIONALE ------

1.) The current implementation does not have buffer cache to speed up the disk read and write. This is a bottle neck in the system.
2.) The implementation follows the exact structure of Linux file systems, supporting a combined allocation scheme and indexed allocation table. Also files can be as large as the file system size itself.
3.) The file name size has been limited to 14 characters, which might be a drawback, but still the path names can be arbitrarily long. Also relative as well as absolute path names are supported.