[Home][Classes][Research][Links][Biography]

Up

CS Dept.

CSC 342 Operating Systems

Assignment 4 - File System

Due: 4/11/2017

Description

Overview

Read section 1.9 (pp. 38-43) of the OSP text and any other sections you think might be relevant.

In this assignment, you will implement the file system of OSP simulator. There are two disk which you will use to store the contents of the files. You will implement index block allocation, storing the physical block numbers in the inode associated with a particular file. You will also implement the bitmap (or vector) method of keeping track of the free blocks.

The file system will have a single directory in which all files will be located. Files will also be *non-persistent*. In other words, you will create a file the first time it is opened, and you will remove it when it is no longer open by any processes. Files will have an initial size of zero bytes when they are opened for the first time. They will grow only when written to. Also, we will restrict the number of open files to 10, so that we can use arrays instead of linked lists.

There are 6 functions you need to implement. The functions that are required are **files_init(), openf(), closef(), readf(), writef(),** and **notify_files()**. In addition to these 6 functions, I suggest that you create the following functions to assist with the implementation of the required functions: **allocate_blocks()**, **search_file()**, **new_file()**, and **delete_file()**.

Data

There are five structures that you will primary work with: a directory entry (**file_dir_entry_node**), an inode (**INODE**), an open-file descriptor (**OFILE**), the Device Table (**Dev_Tbl[MAX_DEV]**, which consists of device entries **DEV_ENTRY**), and an I/O request block (**IORB**). These definitions are:

```
#define MAX_PAGE       16                 /* max size of page tables        */
#define PAGE_SIZE      512                /* size of a page in bytes        */
#define MAX_TRACK      60                 /* num of tracks on a devices     */
#define TRACK_SIZE     1024               /* track size on each device      */
#define MAX_BLOCK      MAX_TRACK * TRACK_SIZE / PAGE_SIZE
                                          /* num of blocks on a device      */
#define MAX_DEV        2                  /* size of the device table       */

typedef struct file_dir_entry_node {
    BOOL   free;
    char   *filename;
    INODE  *inode;
    int    *hook;
} FILE_DIR_ENTRY;

typedef struct inode_node {
    int    inode_id;    /* inode id                                         */
    int    dev_id;      /* device id:  its index in the Dev_Tbl            */
    int    filesize;    /* file size in bytes                              */
    int    count;       /* num of open files associated with the i-node    */
    int    allocated_blocks[MAX_BLOCK];
                        /* user-designed; contains info on how the file is  */
                        /* stored                                           */
    int    *hook;       /* can hook up anything here                        */
} INODE;

typedef struct ofile_node {           /* an entry in the table of open files */
    int    ofile_id;                 /* can be used for easy identification */
    int    dev_id;                   /* device allocated to the file       */
    int    iorb_count;               /* number of pending IORBs            */
    INODE  *inode;                   /* pointer to the i-node of the file  */
    int    *hook;                    /* can hook up anything here          */
} OFILE;

typedef struct iorb_node {
    int    iorb_id;     /* iorb id                                          */
    int    dev_id;      /* associated device; index into the device table   */
    IO_ACTION action;   /* read/write                                       */
    int    block_id;    /* block involved in the I/O                        */
    int    page_id;     /* buffer page in the main memory                   */
    PCB    *pcb;        /* PCB of the process that issued the request       */
    EVENT  *event;      /* event used to synchronize processes with I/O     */
    OFILE  *file;       /* associated entry in the open files table         */
    IORB   *next;       /* next iorb in the device queue                    */
    IORB   *prev;       /* previous iorb in the device queue                */
    int    *hook;       /* can hook up anything here                        */
} IORB;

typedef struct dev_entry_node {
    int    dev_id;      /* device id - index into Dev_Tbl                   */
    BOOL   busy;        /* the busy flag ("true", if busy)                  */
    BOOL   free_blocks[MAX_BLOCK];
                        /* block i is free: free_blocks[i] = true;          */
                        /* else: = false                                    */
    int    num_of_free_blocks;
    IORB   *iorb;       /* the iorb currently being processed by the device */
    int    *dev_queue;
    int    *hook;       /* can hook up anything here                        */
} DEV_ENTRY;



extern DEV_ENTRY Dev_Tbl[MAX_DEV];                   /* device table             */
extern FILE_DIR_ENTRY theDirectory[MAX_OPENFILE];
extern INODE inodesTbl[MAX_OPENFILE];
```

The **`FILE_DIR_ENTRY`** stores the filename and inode of each file in the directory. The directory itself will be an array of these.

The **`INODE`** is a structure that keeps track of the important information about a file: on which device is the file stored (**`dev_id`**), how big is it (**`filesize`**), which physical blocks of the device does it use (**`allocated_blocks [MAX_BLOCK]`**), and how many processes have it open (**`count`**). **The `allocated_blocks` in an array integers that map logical blocks to physical blocks.** In other words, **`allocated_blocks[logical block #] = physical block #`**.

The **`OFILE`** has information about open files. This may seem redundant, but if we were implementing a persistent files system, then files would have inodes and not necessarily be open. Furthermore, multiple processes may open the same file. There is only one inode per file, but there may be many `OFILE`'s. In our case, all files will be opened by at least one process (otherwise they would have been destroyed). In the `OFILE` is: the device on which the file is stored (`dev_id`), the inode (`inode`), the number of pending and not yet completed I/O requests to read and write (`iorb_count`). **(Note that the `iorb_count` in the `OFILE` is not the same as the `count` in the `INODE` . The inode's `count` is the number of times this file is open. The `iorb_count` is the number of pending I/O requests.)**

The `IORB` contains the information pertaining to I/O requests of the device. In particular, a request to read or write to a file will result in an `IORB`, and the process is usually blocked. The device will service the `IORB`'s as soon as it is able to (which may take a long time) and, when it completes them, will signal an event that indicates that the request has completed and the process can continue. Most of this is performed by the Devices module, but you will need to fill in some of the information in the `IORB` structure.

The `Dev_Tbl` is an array of information for each device. The information that is relevant to this assignment is the bitmap of free blocks (`free_blocks[MAX_BLOCK]`). You need to maintain this bitmap as you allocate and deallocate blocks to files. **The free_blocks is also an array of integers, but this time the subscript is the physical block #.** In other words, `free_blocks[physical block#] = true` or `false`.

Functions

files_init()

The `files_init()` function will be called at the beginning of the simulation. You should put in this function any initializations you need to do, such as:

1. initialize directory `free` to all `true` and filenames to `NULL`
2. initialize the directory inodes to point to the corresponding inodes in the inodes array
3. initialize the array of inodes so that their ids match their index within the array
4. initialize the total number of free blocks in the Device Table to `MAX_BLOCK`
5. initialize the free blocks of the Device Table to all `true`

void openf(char *filename, OFILE *file)

The `openf()` function is used to open a file. You are given the filename and a template `OFILE`, which you will need to fill in. The steps you will do in this function are:

1. Search through the directory to see if the file exists. (I would suggest creating a separate function for this. See below.) If it does exist, then use the inode stored in the directory for this file.
2. Otherwise, if it is not in the directory, then create it. (I would suggest creating a separate function for this. See below.)
3. Once you have the inode (either from either step 1 or step 2 above) then:
    a. Set the `inode` field (of the `OFILE`) to point to this inode
    b. Copy the `dev_id` from the inode to the `OFILE`
    c. Set the `iorb_count` (of the `OFILE`) to zero
    d. Increment the inode's `count`

EXIT_CODE closef(OFILE *file)

The `closef()` function is called whenever a process wants to close a file. The steps of this function are:

1. If the `iorb_count` of the `OFILE` is greater than zero, return `fail` and do nothing else. (There are still pending I/O requests so we can't close the file yet.)

2. Decrement the inode count (which is the number of opens)
3. If the inode count is zero, then delete the file (I would suggest creating a function to do this. See below.)
4. return `ok`

EXIT_CODE readf(OFILE *file, int position, int page_id, IORB *iorb)

The `readf()` function performs a read request on behalf of the process. The process wants to read a byte at offset `position` bytes from the beginning of the file. The `IORB` is a template that you need to fill in (but not the `iorb_id`). The step of this function are:

1. Save the current process (`PTBR->pcb`)
2. Make sure position is within a proper range (`0 <= position < filesize`). If not, then set the `iorb->dev_id` to -1 and return `fail`
3. Convert the position to a logical block number. Since the devices use a block size equal to one page or frame, then the logical block number is `position` DIV `PAGE_SIZE`.
4. Determine which physical block number this corresponds by consulting the `allocated_blocks` array of the inode.
5. Increment the `iorb_count` of the `OFILE`
6. Fill in `IORB` template (except the `iorb_id` and the `event`):
    1. Copy the `dev_id` from the inode
    2. Set the `block_id` to be the physical block number
    3. Set the `action` to be `read`
    4. Set the `page_id` (from the parameter `page_id`)
    5. Set the `pcb` to the current process
    6. Set the `file` to be the parameter `OFILE`
7. Perform an I/O Request Interrupt by:
    1. `iorb->event->happened = false;`
    2. `Int_Vector.event = iorb->event;`
    3. `Int_Vector.iorb = iorb;`
    4. `Int_Vector.cause = iosvc;`
    5. `gen_int_handler();`
8. return `ok`

EXIT_CODE writef(OFILE *file, int position, int page_id, IORB *iorb)

The `writef()` function performs a write request on behalf of the process. The process wants to write a byte at offset `position` bytes from the beginning of the file. The `IORB` is a template that you need to fill in (except for the `iorb_id`). If the position is beyond the filesize, then the filesize should be increased and new blocks allocated to the file. The step of this function are:

1. Save the current process (`PTBR->pcb`)
2. Make sure position is within a proper range (`0 <= position`). If not, then set the `iorb->dev_id` to -1 and return `fail`
3. Convert the position to a logical block number. Since the devices use a block size equal to on page or frame, then the logical block number is `position` DIV `PAGE_SIZE`.
4. Determine the last block, which is `(filesize - 1)` DIV `PAGE_SIZE` if the file is not empty or -1 if the file is empty
5. If the logical block is greater than the last block, then allocate enough blocks so that the file will fit given the desired new filesize. (I would suggest creating a separate function for this. See below.) If the number of blocks needed is greater than the total free blocks for the device, then set `iorb->dev_id` to -1 and return `fail.`
6. If the `filesize` is less than or equal to `position` then set `filesize` to be `position+1`
7. Determine which physical block number this corresponds by consulting the `allocated_blocks` array of the inode.
8. Increment the `iorb_count` of the `OFILE`
9. Fill in `IORB` template (except the `iorb_id` and the `event`):

   a. Copy the **dev_id** from the inode
   b. Set the **block_id** to be the physical block number
   c. Set the **action** to be **write**
   d. Set the **page_id** (from the parameter **page_id**)
   e. Set the **pcb** to the current process
   f. Set the **file** to be the parameter **OFILE**

10. Perform an I/O Request Interrupt by:
    a. **iorb->event->happened = false;**
    b. **Int_Vector.event = iorb->event;**
    c. **Int_Vector.iorb = iorb;**
    d. **Int_Vector.cause = iosvc;**
    e. **gen_int_handler();**
11. return **ok**

void notify_files(IORB *iorb)

The **notify_files()** function is called after the device as finished performing the request I/O. You simply need to decrement the **iorb_count** of the **OFILE** stored in the iorb.

Suggested Functions

These are the functions that I recommend you create.

EXIT_CODE allocate_blocks(INODE *inode, int numBlocksNeeded)

This function should search the the device's free blocks and allocate **numBlocksNeeded** to the file, given the inode (of an already existing file) and the number of blocks required. It needs to do the following:

a. If the number of blocks needed is greater than the total free blocks for the device, then return **fail**
b. Compute the first logical block number needed. This will be **(filesize - 1)** DIV **PAGE_SIZE + 1** if **filesize** is greater than zero or 0 if **filesize** is zero.
c. Search through the blocks from the device table for free blocks until you have the number of needed blocks. For each free block that you find, allocate it by:
   i. Set the flag in the **free_blocks** to **false**
   ii. Decrement the total number of free blocks
   iii. Store this new physical block number in the **allocated_blocks** for the current logical block number
   iv. Increment the logical block number
d. return **ok**.

int search_file(char *filename)

This function should search through the directory for the given filename. If the directory entry is ***not free*** and the filename matches (use **strcmp** not **==** on strings), it should return the index within the array. If it does not exists it should return a -1.

int new_file (char *filename)

This function should search the directory for a free directory entry, create the file, and return the index in the array where the new file. It should do the following:

a. Find a free directory entry. If there are no free directory entries or inodes, then print an error and return a -1.

b. Save the filename in the directory entry
c. Set the boolean associated with this file entry to false to indicate that the directory entry is used
d. Set the `filesize` and `count` (number of opens) of the inode to zeros.
e. Choose a device (use the device with the largest number of total free blocks), and set the `dev_id` of the inode to this
f. Set the `allocated_blocks` in the inode to all empty (such as -1's)
g. return the directory index

void delete_file(int dirNum)

This function should delete a file and return the directory entry to the pool of free entries. It should do the following:

1. Go through the list of blocks in the `allocated_blocks` array from the inode, and, for each valid block:
   i. Set the block to free in the Device Table
   ii. Increment the total number of blocks for that device
   iii. Mark this block as unallocated (such as -1)
2. Set the boolean to `true` and the filename to `NULL`

Provided Functions

void print_dir()

This function will print the entries in the directory. This is not the same as the list of open files that SIMCORE prints when you call the `print_open_files_tbl()` function. The table produced shows the filename, inode number, and the physical blocks that have been assigned to each file.

void print_disk_map()

This function will print the free blocks of the device. This is a slightly different view of the disk map than is produced when you call the `print_sim_disk_map()` function. For each block, the total number of free blocks is given as well as the bitmap of free blocks. In the bitmap, a dot ('.') indicates that the block is free, whereas an 'X' indicates that it has been assigned to a file.

Setup

Copy the files from the assignment directory (`~osp/a4.linux`) to your working directory. These files are:

- `dialog.c`
- `hand_in`
- `Makefile`
- `files.c`
- `files.h`
- `OSP.demo`
- `osp.o`
- `parameters.high`
- `parameters.low.trace`
- `parameters.med`

The `files.c` file is where you need to implement the above functions.

The `dialog.c` file contains functions that are called at certain times, such as at each snapshot, an error, or a warning. You may add code to these functions as you see fit.

Helpful Notes

To compile your program, just type **make**. To run it, type **./OSP**. This will run interactively, prompting you first for parameter information. Much of this information will not be relevant to this assignment. You will also be given an opportunity to change the parameters at snapshots. The parameters that you provide will be saved in a file called **simulation.parameters**. If you want to rerun your program with the same parameters, then type or **./OSP simulation.parameters**. You will eventually need to run your program on the three parameter files **parameters.low.trace** , **parameters.med**, **parameters.high**.

The program **OSP.demo** is a working implementation of the CPU scheduling module. You can run this program the same way that you run your **OSP**, to see how your program might work. You do not have to have the exact same output as **OSP.demo** , but it is simply there for comparison purposes.

The **parameters.low.trace** file has tracing turned on. To run your program on this file, type **OSP parameters.low.trace** . This means that there will be debug print statements throughout the output. You can add debug statements within your functions by doing something like this:

```
if (______trace_switch)
        printf("CLOCK> %6d# FILES:openf(filename==\"%s\",file=%d) \n",
           get_clock(), filename, file->ofile_id);
```

The simulator will reseed its random number generator every time you run it. This means that the results will differ with each run. If you find a problem with your program and you want to track down a particular bug, it is helpful to run with the exact same input each time. If you run the **OSP** with the **-d** option (i.e. **./OSP -d parameters.low.trace**), then it will use the same random numbers it used in the last run.

If you have a segmentation fault, then it is best to turn on interactive mode. Without interactive turned on, the results are sent to the file **simulation.run** . The output will not necessarily be flushed at the time of the segmentation fault. This means that you may not necessarily see the last print statements before the segmentation fault. To deal with this, turn on interactive by changing the second-to-last field in the parameters file from the "n" to a "y".

Turning in the Assignment

- Run the **hand_in** program, please provide at least the following parameter files: **parameters.low.trace, parameters.med** and **parameters.high**.
- Print a hardcopy (one copy per group) of the source files that you modified (e.g. **files.c** and/or **dialog.c**) and hand it in at the beginning of class. Only one copy per group is necessary
- Print the project assessment document (PDF), fill it out, and *hand it in separately from your source code (one for each person). Please fold the assessment so that your responses will be seen only by me.*

This page was last updated: **March 30, 2017**

Email: cferner@uncw.edu

[Home][Classes][Research][Links][Biography]