P6 DESIGN

Operating Systems – CSCE 613

ABSTRACT

File Systems

Tharun Battula

82400197

This is design doc Disk Scheduling and File System Management with bonus options. In addition to implementing the required tasks in project 5 (blocking drive and file system), Bonus option 2 has been completed. Bonus Option 3 is partially completed.

DESIGN

The objective of this project is to implement a blocking drive and simple file system. When any thread makes a request for read or write operation from the disk drive, it should give up the CPU until the disk operation is complete. It should not keep CPU occupied doing nothing, waiting for I/O operation to complete. Also, this project includes file system in while files support sequential accesses.

Blocking Drive

For each disk drive, a queue using array is instantiated which stores the thread waiting for the disk I/O in first come first serve basis. The queue holds information of thread, operation code and block number corresponding block number to be written.

Read/ Write Operation

Whenever operation is requested,

- If there is no operation running in the disk or there is no queue, then current operation is issued
    - If the current operation is in progress, enter waiting state where you yield CPU to other threads until the result is ready.
- If there is queue for the disk, your process is enqueued after others and you wait in the queue by yielding to others.
    - In this waiting state, if you observe no operating running but queue is not empty, you will dequeue out the first in thread and move the queue. For this dequeuing the stored information of data structure and thread, block number, READ/WRITE code are used. Any thread can dequeue appropriately.
    - If the disk is doing your operation, enter into wait for ready state and wait until results are ready

- Thread safe implementation is done using lock mechanism, i.e., All crucial operation requiring atomicity are done by maintaining a single lock for single drive. More details to follow.
- If the disk is still busy, above process of yielding and checking back repeats until it is ready and the correct thread has given up using the disk.

Scheduler class's resume and yield functionalities from P5 implementation are used to call dispatch during waiting status (either during the operation in progress or in queue). The usual thread dispatch happens in Round Robin for each interrupts and also FIFO is embedded.

This is a polling method since we are checking in regular timings whenever thread returns back in scheduling we check the status of operation and its corresponding progress.

I am using an array of data type to maintain as the ready queue. Using array a ready queue is implemented by rotating array. i.e., initially head is at the 0 and enqueuer is done by adding new thread pointer at the tail and head is moved on. When the tail/head meets the tail the loop is rolled back by doing modulo operations. This way Queue is implemented and it creates the restriction on the maximum number of threads for the kernel but it is impractical to maintain lot of threads beyond certain point.

Class for this queue structure can be found in the files submitted exta → **queue.H** and **queue.C**. It creates queue for any data type. I am working with one drive, as I am not implementing mirror disk.

#define MAX_QUEUE_SIZE 100

IssueOperation()

Has the low level port selection and data parsing for doing read and write operations, this is inferred from simpleDisk class

write ()/read()

This above mentioned logic for scheduling and queuing structure are maintained and polling method is implemented. The explanation on Lock Mechanism can be found in the Bonus3 part description.

Additional functions created are

File Systems

The disk consists of blocks, each block being 512B. I have used iNodes made my design to save the files and file system on the hard drive instead of RAM. This design is persistent on non-volatile so the file system can be accessed later with multiple boots or across systems. This is main advantage that data is persistent. This comes with overhead of performance since every memory access might need to be accessed from the disk or even understanding file system or manipulating a disk access write may be needed.

File system attributes

static unsigned int size; → size of memory from formatted time

static BOOLEAN is_mounted; → flag for mount property

*static unsigned long * free_blk_bit_map; → Bit map for finding blocks unused*

static unsigned long no_of_blks; → number of total blocks

static unsigned long no_of_inodes; → number of files

static unsigned long inode_info_blocks; → number of blocks used for file management

*static BlockingDisk * disk; → Disk with concurrent access, read/write thread scheduling*

(Part1)

Memory Management

A file can have maximum of 10 blocks = 10*512 . Size of the file system is decided from the moment of formatted size. Proper rounding off has been taken care in the actual implementation.

For a given size = S

Number of blocks = S/512

Number of inodes = S/(512*10)

Number of info blocks=(S/(512*10) * sizeof(inode data structure))/512

Number of files that can be created is ~= Number of blocks - Number of info blocks.

The last information blocks hold the key file system data and used to manage bitmaps and allocated information about each individual file or inode. The number of information blocks is very less only 1 or 2 blocks for smaller size of file system.

File Manipulations:

Since the blocks and data is residing in the drive, for manipulation of any file or system first the required data is fetched from disk and is observed for details and update is made and then it is written back. In between new blocks may be created or rewrite. For simple call of files reading and writing data is called multiple times in multiple calls. There is lot of overheads observed in doing operations repeatedly. But this makes the design more atomic and latest information can be read and be written any time.

File Class

File is implemented through as inode, its information is saved in bitmaps and corresponding block indices. Information is in blocks (chunks of data) blocks may not be sequential in actual memory. This design is chosen to avoid external fragmentation and blocks can be shifted or used anywhere. The internal fragmentation problem still persists. Though the maximum blocks for file is fixed to be 10, all blocks are not allocated once. Files are allocated block during the write if a user needs to write extra data.

Block Allocation Strategy: First Fit Algorithm

In this project, we use continuous memory allocation. Thus when blocks need to be allocated, continuous group of free blocks is searched on first-fit basis. First fit means that whichever block satisfies the condition first in the iteration, that is taken and allocated. This information is maintained iNode bitmaps.

Attributes:

unsigned long cur_blk_for_read; → *The current block number pointer. It changes only in read*

unsigned int ind_of_cur_block; → *The index of current block in the serial blocks of memory*

unsigned int cur_pos_in_block; → *The current position pointer, it changes only in read*

unsigned long first_blk; → *The first block*

Implementation

1. File() : Just initializes the variables and sets default values

   When a file is created first a single block is allocated but no values are initialized. CreateFile of File systems will have more details on this.

2. Read Operation

   For read operating on current position in file a cursor pointers are maintained and these pointers are updated only for read. First data is fetched and and it is read to the buffer and based on the file size next data is fetched based on inode blocks information and in a loop this happened. The position of pointer is updated. The data need not be written back but if there is any change iNode information it needs to be written back.

   If current cursor is pointing to end of file or close to EOF then for a read request of large size will happen only till EOF. i.e., only the possible information is read. To read from the starting of the file one must reset the cursor before read.

3. Write Operation

   Unlike read this does not change the current cursor based on writing data. But this changes the blocks size based on the amount of data. There are 3 cases in write operation. Write overwrites data from the current position in the file.

   Case 1) File is new or no blocks are allocated to the file (length of file is 0): In this case, it is simply written into the document one by one using the blocking disk API, file size is the new file size

   Case 2) Blocks are allocated to file and there is sufficient space to write the data then we do not allocate any new block but overwrite contents of data, file size will not change

   Case 3) When the new size is more than, Blocks are allocated and new content needs to be written. Blocks allocated are NOT sufficient, file size will change to the newly added extra movement.

4. Reset Operation – Reset means what cursor position of file is set to first character in the file. Thus, current cursor position of position and block are set to default initial values

5. Rewrite Operation – Rewrite means that all content of the files are cleared (note that file is not deleted). In this all the data is written zero in hard disk for cleaner implementation or corruption purposes. And blocks allocated per file are released. In

file management attributes start block and length of file are set to 0. First block allocated remains as is holding -1 to represent EOF of the file.

6. End of File – It is important to know if EoF has reached during read operation and stop reading if it is. This can be simple implementation of checking cursor with the file size. Additionally I have made one more condition to check file if file character as -1. End of file returns high if cursor position matches the length of the file – that is it is situated at end of the file. For a deleted or newly created (unwritten) file EOF is true immediately.

FileSystem Class

File system is responsible for managing the files in the disk. This class implements the main functionalities as per the given homework and also it accommodated some helper functions to improve the algorithm and results.

1. Mount Operation - Mounting the Blocking Disk pointer and mounted flag written.
2. Format Operation

   This is the main initial operation setting up the variables and memory sizes for blocks, inodes and their corresponding info. Once the variables are initialized all the blocks are formatted by writing 0 to the blocks in the hard disk. The format is needed since the previous disk might be having different file system. For this file system anyway mount will wipe all the data and start the disk fresh 0's. The wiping of starting few blocks is definitely necessary since it has file management information.

   The overhead for this is it takes lot of times to format all the information by writing '0's For this reason grader it is preferable that grader wait until format happens, to lower the time choose disk size as small.
3. Creating a File

   First it is checked if the file already exists. If exists, then function returns false.

   Then, a free location in file management is found and first attribute (file id) is set. Initial start block, length and cursor position is set to 0. Then, file management blocks from memory is copied back to the disk.

When a new file is created, no blocks are allocated to the file. So its start block and length attribute are 0. Blocks are sequentially allocated when something is written to the file.

4. Deleting a File

   File information in inodes is cleared and file pointers are set to null. The blocks used up for file are released and all the updated information is updated by writing back the inode information.

5. Searching the file (lookup file)

   First the information corresponding to the file management infor is obtained. In the inode information File ID is matched with the appropriate block running in a loop. Then the file attributes are loaded stored in order

   Other Helper Functions created for the design

   a) *get_next_free_block_num();*

      Bit map information for free blocks is maintained and this function searches first fit algorithm and find the first empty free block

   b) *GetFileBlocks(int _file_id);*

      This returns information array about the file blocks mapping for all the 10 blocks to real order of blocks in memory.

   c) *release_blk(unsigned long block_no);*

      To free block by setting bit map information for the block as null and adding in free pool of memory.

   *d) inode_increase_size(File *_file, int file_id, int file_size_increase);*

      Whenever there is a file size happening due to write, the corresponding info is modified in inodes by doing size increase.

   e) inode_add_new_block_num(int file_id, int block_no);

      This is similar to the file size increase, here whenever we add new block to a file as part of writing, we update information about blocks number in bit map infor using this function.

Code Modifications

File Name	Reason for Modification
Scheduler.C, Scheduler.H	**P5 Modifications**, Code for FIFO, add, resume, yield, terminate .H modified declaration of data structure private variables
thread.C and thread.H	**P5 Modifications**, Implemented functions: thread shutdown, start function
simple_timer.C and simple_timer.H	**P5 Modifications**, To handle interrupt for Round Robin timer, using the existing code and frequency to utilize it. New variable is created to count the timer for desired 50 ms time period for RR quantum.
interrupt.C and interrupt.H	**P5 Modifications**, This was needed to create a segregated function for handling EOI
blocking_disk.C, blocking_disk.H	**Part1 of P6,** In kernel need to modify simpleDisk declaration to blocking_disk declaration
file_system.C, file_system.H	File System Implemenation of **P6**
Kenel.C	Test case in exercise file system, Stack increase for fun2 and fun3
queue.C, queue.H	A generic array based template implementation for queue of any data type No need to add these in the makefile

The other files modified are (which are not required to submit)

1. console.c → to print the bochs emulator output to a text file stream to view the output and analysis.
2. Makefile.linux64 added scheduler.c and its corresponding make commands for compilation (Add all the above files and corresponding makefilles., no need to add queue.C queue.H)

Assumptions

- The same assumptions from P5 over FIFO and Round Robin are assumed. Scheduler queue has some thread running.

- In kernel.C **BlockingDisk** is used instead of simple disk, in initialization and usage of disk for the file systems
- USES_FILESYSTEM will be enabled only when _USES_DISK is enabled.
- Assumption of **Stack size** for thread2 **or** thread3exercisiing file system to be >= **4096.** I have tested with 8192 for thread3
- Proper initializations are assumed, File system is called, first formatted then mounted and then created a file or so on
- File size limited to 10*512 B, file system of large data can take lot of time suggest to limit the file system size to very few for faster level of results printing in the console.
- The order is maintained create, lookup, write, read, write, read, rewrite.
- All corner cases may not be covered such as accessing file pointer after deleting the file.
- I used a macro #define DEBUG_ENABLE 0 to enable the control over prints.
- All the files are compiled properly in the make file, the new ones.
- I am not using rand() for the testing.

Suggestions for testing:

Tips for checking the file system functionality in a faster way

- Select Smaller file system around 10*512 and restrict number of files
- Make Function-1 or Function-4 vanish after some time, as they keep switching and make difficult to notice the output
- Fun2 Fun3 tests the main files
- In function -2 you can remove random block read.

To check the proper functionality of File Systems, either disable concurrent file access from fun2 or make fun2 access far block.

Testing

According to piazza discussion.

I have created and test with multiple test files,

1) Format - small size is preferable for faste output
2) mount
3) createfile (tried multiple types)
4) lookupfile
5) write- 600
6) read- 600 → compared data
7) reset-
8) write-300
9) read-300 → compared data
10) reset-
11) write-1024
12) read-1024→ compared data
13) write- 500
14) read-500 → compared data
15) rewrite- checked
16) eof- checked
17) deletefile- checked
18) Lookup after delete

Bonus 2 - Design of thread safe disk and file system *(design, implemented partially)*

In P5, we assumed that the disk can be accessed by at most one thread at a time. If multiple threads can access the disk concurrently, then there can be race condition. The following section describes how to handle concurrent operation on the disk and file system in a safe fashion.

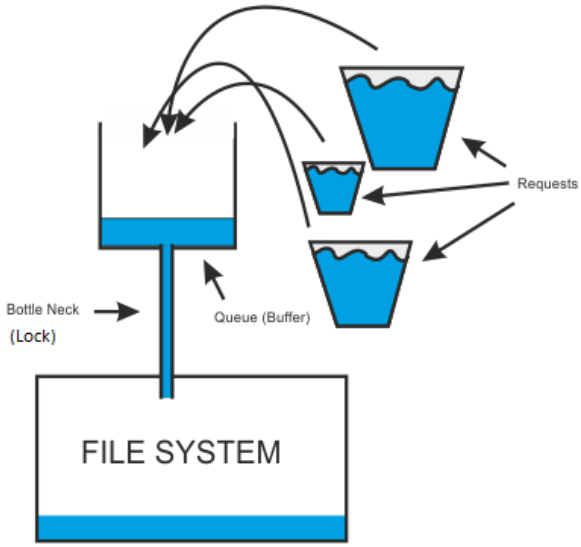
For handling concurrent access of file system, concept of "locks" can be used. Example of such file system is NTFS in windows.
Whenever a thread tries to access a file system, it can be inserted into a **waiting queue**.
For critical section, locking concept needs to be used to guard concurrent access to the same file.
There are 2 ways to implement locks – with busy wait OR without busy wait. No busy waiting is more efficient, but difficult to design and implement.

Busy Wait Implementation:

Mutual exclusive access to the critical section can be used.

Concept - When a thread tries to access a disk or a file system, it needs to get access to the lock. The first process to get access to the lock can execute its critical section. The threads trying to gain access to the lock later on are inserted into a waiting queue.

Image reference: http://stackoverflow.com/questions/7421797/how-to-handle-large-numbers-of-concurrent-disk-write-requests-as-efficiently-as



Sample code for implementing lock using MUTEX:

Using Petersons algorithm

https://en.wikipedia.org/wiki/Peterson%27s_algorithm

```
bool flag[2]   = {false, false};
int turn;

flag[0] = true;
turn = 1;
while (flag[1] && turn == 1)
{
// busy wait
}
// critical section
...
// end of critical section
flag[0] = false;
```

There can be 2 queues (similar to blocking disk queue)

One: Ready queue which is similar to scheduler queue.

Second: Waiting queue in which threads are waiting for locks/ resources to get available.

(a) Disk Access

Disk contains blocks of data, file systems and files. Certain portion of disk needs to be reserved for this management information.

i. Each file system can be allocated particular number of blocks. So, each file system would have limit on maximum number of files and total of file sizes.

ii. File System Management information (similar to file management information) can be kept at a particular location of the disk.

iii. To access management information of the disk (which can have possibly multiple file systems), **disk_lock** needs to be accessed.

iv. Disk Lock can be implemented similar to Busy Wait Implementation described in the section above.

(b) File System Access

A particular file system can be mounted by only multiple threads for concurrency. If mounted by multiple threads, then we need to design distributed file system.

Some file system file NTFS allows concurrent reads, but not writes. Following is the design for concurrent file system.

i. Multiple threads can read from file system concurrently without contention

ii. If any thread is trying to write into a file system, then it needs to acquire **file_system lock** before it can make any changes to the FS. Further, this lock can be made available only when no other thread is reading from it.

(c) File Access

In a particular file system, if multiple threads try to access same file, then concepts similar to file system access can be used.

Further, while opening a file, new attributes like read only OR read-write can be associated with the file. Only if read-write access is high, program allows making changes in the file. Else it returns FALSE.

Further point to note is that in above section busy implementation was described. There can also be a non-busy implementation by putting the thread into the **waiting queue**, and waking up (putting back into **ready queue**) when the resources become available.

However, in non-blocking case we would need synchronizers to sync the operations. For example: read after write at same location.

Bonus 3 – Thread safe mechanism for Disk Access

I implemented the busy wait implementation explained in the above Locking Mechanism and existing code developed. We have already a queue for disk scheduler and another ready queue for System Scheduler. Using the existing queue of disk scheduler I have embedded the queue for the lock. I am using thread as the lock value in below attributes.

Lock == 0 → the resource is free,

Lock! =0 → the resource is operating for a thread ← (Thread *) lock

```
unsigned int lock;    // disk is current operating if not 0

disk_op current_status;
Queue<disk_op*> disk_ready_Que; // Queue Holding each disk info
```

I use the current_status variable to denote the information of current resource and functionality happeneing on it. I.e, whenever I issue operation I store the information in current status. I use this information for processing the type of queue for managing the queue and critical section.

Below is my implementation for the queue of the data type. It has information of Threads.

```
typedef struct disk_operation{
      unsigned long block_no;
      DISK_OPERATION operation;
      Thread * thread;
}disk_op;
```

Sample intuitive pseudo code is the design

First issue time:

If you see a queue or resource is busy
    Enqueue your self with thread number
Else
    Issue operation and update
    Lock = current_thread ;
    Wait for the operating

After issuing and once waiting in disk queue

While()
    If you are the lock and the operation and block number are correct
        go wait for the result to come out

If lock is 0

Dequeue and Issue next operation (update corresponding current status)

Lock = dequeue thread;

If lock is not you

Wait and yield CPU

Once Data is ready , and read by you

lock =0; //reset the lock

dequeue and issue next operation

update the lock to corresponding dequeued thread.

This design works since every request by same thread is sequential and all other threads will not touch the resource until they get the lock i.e., lock ==0 and getting dequeued. This required atomicity maintained at the critical sections of manipulation of management section.

Atomicity

From the interrupts and simpletimer implementations from P5, using machine_enable_interrupts() and machine_disable_interrupts() can be used for shielding the critical sections in the code by disabling interrupts during critical execution.

Previously for P5, I have disabled interrupts in starting of resume () and enabled them back in the last part of yield(). This is to avoid interrupts during manipulation system scheduler queue.

Similarly in this project for above mentioned bonus 3 part lock mechanism I have disabled interrupts during

Critical sections covered:

1) Enqueue of Block Disk Waiting queue
2) Dequeue of Block Disk waiting queue
3) Issue Interrupts + setting the lock
   this has information is set in low level assembly, interrupts would confuse the operating if there are multiple requests at once per block
4) Reading of the buffer from the output of disk controller
5) Writing of the buffer to the input of disk controller