CSE 542/498J HWP 1: File System

Assigned: Thurs, Aug 29

Due: Tues, Sep 10, 11:00AM

Late submissions will not be accepted

For this homework, we will implement a simple file system built on top of our own simulated disk storage system. Our file system will support files stored in a single directory (no hierarchical directory system). The system will consist of the following components:

1. Storage subsystem:

Our storage subsystem consists of four independent disk mechanisms to store fixed size disk blocks. The schematic of a typical disk subsystem is discussed in Section 2.3.2 of the textbook. You can implement each disk using a single UNIX/Windows file. For our homework, assume a block size of 128 bytes and a disk size of 1 MB (you will use four 1 MB files). To store or retrieve these fixed size blocks, the simulated disk system incurs rotational latency, seek latency as well the actual transfer time to read or write a block. For simplicity, we will assume that the system takes a constant time of two seconds per operation. For example, each write operation would incur seek (2), rotational latency (2) and a write delay (2) for a total of six seconds. Operations on a single disk is sequential, the disk cannot be used while processing another request. Our storage subsystem consists of four such disk mechanisms to increase throughput (by exploiting the parallelism inherent in using four disks).
For this homework, each drive will implement the following functionality:

1. readDisk (*blockNumber*) – return the block of data stored in the disk at *blockNumber*. Note that it is not an error to request to read a *blockNumber* that was never written by a previous write() operation; the data returned is undefined, however.
2. writeDisk(*blockNumber*, *data*) – write the *data* in the *blockNumber* specified. Reading and writing past the end of the disk is an error.

New read and write operations wait for pending readDisk() or writeDisk() operations. The file system will use the different disks to achieve higher throughput.

Systems calls of interest: lseek(), read(), write(), open(), create(), close()

2. File system:

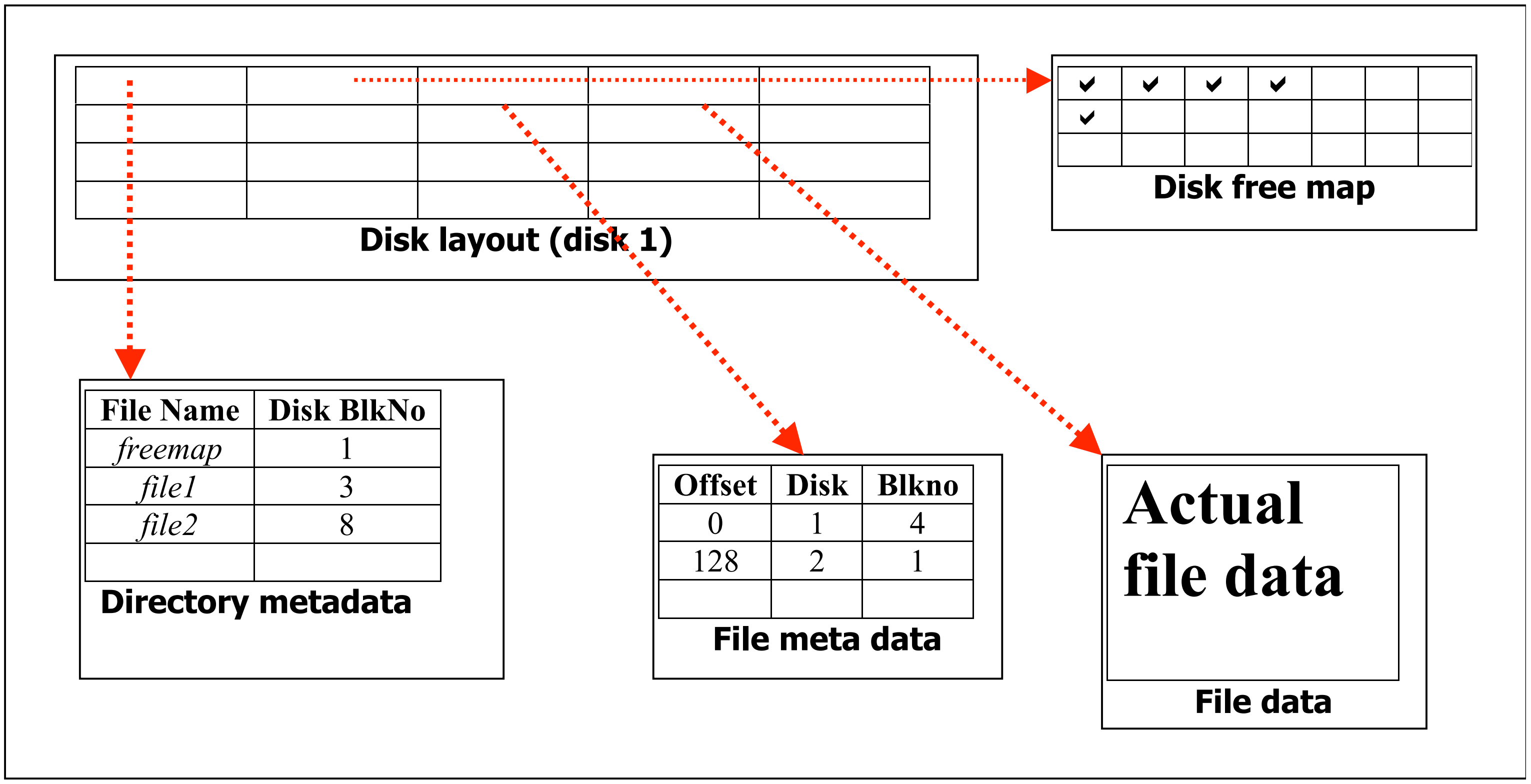
We build our file system on top of this storage subsystem. The system uses a directory meta-data structure to maintain information about the various files. For our system, we assume a flat directory structure (there are no sub-directories). The directory entry stores information about the name of a file and the disk block number containing the file meta-data. The file meta-data contains a list of block and disk pointers to the actual blocks that make up this file. The directory as well as the file meta-data blocks should grow as needed (to accommodate large files). Note that you have to store this meta-information (directory and file pointer) in the disk storage itself. Assume a fixed location for the start of the directory structure (e.g. block 1 in disk 1) to make it easier to boot strap the system.

3. File operations

Your file system will support the following operations:

1. createFile(*filename*) – create a file with the given name in the file system. (add a directory entry for the given *filename*)
2. deleteFile(*filename*) – deletes the given file name from the file system. (removes the entry from the directory structure and free any data blocks allocated to the file)
3. readFile(*filename*, *offset*, *size*) – reads the data stored in *filename*, from the given *offset* up to *size* bytes. Note that this operation may translate into multiple readDisk() operations to retrieve all the data.
4. writeFile(*filename*, *offset*, *size*,) – writes the specified data into the storage subsystem. New disk slots are allocated for this file as needed.

To implement this system, you will have to maintain a free segment map that shows the unallocated disk blocks. Of course, this disk map should also be stored in the disk storage for subsequent use. Our system architecture is illustrated in the following figure.



To further clarify the operation of our system, here is the typical sequence of events required to implement readFile() operation.

1. readDisk() – read directory meta-data, search the entries for the required file and locate the block number for file meta-data
2. readDisk() – read the located file structure meta-data. Search this structure to locate the block numbers of all the required data blocks
3. while(blocks to read)
4. readDisk() – read file data

Testing your program:

You will develop a driver program that will test your file system. The driver will read a sequence of file operations from a text file using the following format:

<arrival time> <operation> <args>
where arrival time (in seconds since epoch) is the time that the operation should commence, operation is
0 for createFile(),
1 for readFile(),
2 for writeFile() and
3 for deleteFile().

For each operation in this trace input, you will invoke the corresponding system call in your file system. After the completion of each file operation, you will print the following output

<current time> <arrival time> <operation> <time required for operation>

Submission

You should utilize threads for ease of implementing the various queue processing (for example, you can use one thread per disk to simulate the seek, rotational latency and operation time intervals, one queue to read the trace at the specified time etc.). **In general, start early. Develop your algorithm in paper first and not on the keyboard.**

Submit your project, along with a succinct report called REPORT.txt (plain text is fine) describing your approach, the merits of your approach and compilation instructions. You will turn in your complete project as a single tar file. On wizard, please use ~surendar/Public/bin/turnin HWP1 <your tar file> to submit your assignment. You can submit your assignment multiple times. I will only use the latest submission. To see the files that you had submitted, try turnin HWP1. Remember, I may randomly choose students who will be asked to explain their approach in person.

Evaluate your implementation on the following issues in the REPORT.txt:

- **Caching:** Caching is a popular technique to reduce access latency. For example, caching the directory meta-data can save on an extra disk read. Discuss how caching can help your implementation and the problems that you would have to address.
- **Reliability:** File systems have to offer reliable data storage. Discuss conditions under which your file system can become inconsistent (for example, what happens if your file system crashes half way through a write operation?)
- **Access patterns:** File system access patterns affect the performance. For example, sequential file access may benefit from striped data storage (across the 4 disks). Discuss the access pattern that will benefit from your implementation strategy.

Future Home Work Possibilities

- **Disk block placement algorithms**: We will Implement a realistic disk model with more accurate seek, rotational latency and transfer times. You will place data objects more intelligently to exploit the current disk head location
- **Raid storage**: You will implement the raid mechanisms to achieve better reliability and throughput.