Linked List

stupid.os needs a file system

Topics:

- Programming with a Linked List
- Programming with a Queue
- Programming with an Array
- Understanding a simplified form of an Operating System algorithm

Outcomes:

- Define data structures (types) such as heaps, balanced trees, hash tables.
- Identify, construct, and clearly define a data structure that is useful for modeling a given problem.
- Use a specific algorithmic technique in solving a given problem (e.g., i can write a dynamic program that solves a shortest path problem).
- Combine fundamental data structures and algorithmic techniques in building a complete algorithmic solution to a given problem.

Description

Hello programmer, stupid.os needs a new file system built. We need to complete a prototype demonstrating how we can link together file chunks to store files on the hard drive. The makers of stupid.os remind us that they don't have any time or care for little things like directories, nope it's just one big folder for their users!

Use the following Guidelines:

- Give identifiers semantic meaning and make them easy to read (examples numStudents, grossPay, etc).
- Keep identifiers to a reasonably short length.
- User upper case for constants. Use title case (first letter is upper case) for classes. Use lower case with uppercase word separators for all other identifiers (variables, methods, objects).
- Use tabs or spaces to indent code within blocks (code surrounded by braces). This includes classes, methods, and code associated with ifs, switches and loops. Be consistent with the number of spaces or tabs that you use to indent.
- Use white space to make your program more readable.

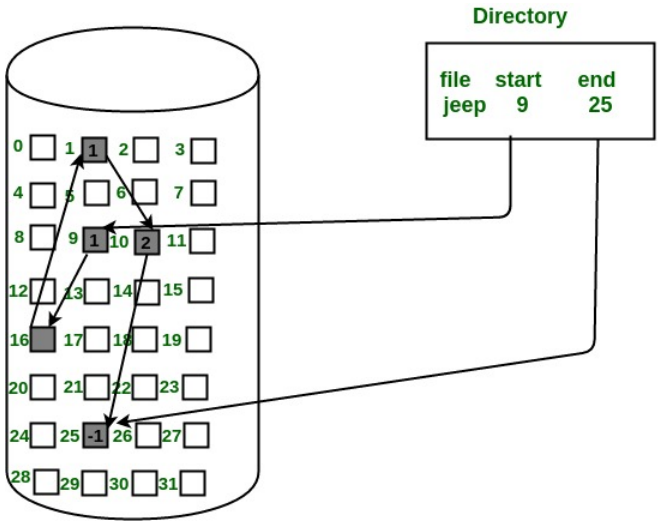
Important Note:

All submitted assignments must begin with the descriptive comment block. To avoid losing trivial points, make sure this comment header is included in every assignment you submit, and that it is updated accordingly from assignment to assignment.

Programming Assignment:

Instructions:

This assignment will have you using several data structures to abstract the idea of a Linked File Allocation file system.



Every operating system has a File System that organizes and keeps track of the files on the hard drive. However, files aren't just single giant blobs. A hard drive is a very organized structure that stores information in sectors and blocks. Blocks of memory of memory are usually only about 512 - 4096 bytes.

This means that every file is partitioned into blocks to be stored on the hard drive. Storing these blocks contiguously is possible, but highly restrictive. We can use other methods using data structures to solve this problem. One method is to used Linked Lists to connect all the file blocks together, that is the method we will be simulating in this assignment.

To achieve this, we will be using a very large array as an abstraction for the hard drive. We'll be using a queue to keep track of available blocks. We'll be using our linked list to keep track of files and filenames.

Specifications:

Part 1 – Queue

The first thing you need to do is to build a Templated Queue to be used in this assignment.

Create a library file: queue_<lastname>.hpp to contain your Queue data structure. This should be a simple modification of the ideas of a linked list.

Method	Description
Queue()	Constructor
~Queue()	Destructor
+enqueue(T):void	Adds item to the queue
+dequeue():void	Removes the front node of the queue
+front():T	Returns the item at the front of the queue without removal
+size():int	Returns the number of items in the queue
+empty():void	Empties the queue
+isEmpty():bool	Is the queue empty?

Part 2 – Modify you Linked List

To facilitate our file system, we need to be able to add to our Linked List in a sorted way.

Create new methods to help support this:

- sortedInsert
  - This will take data into the linked list and keep it in a ascending sort order

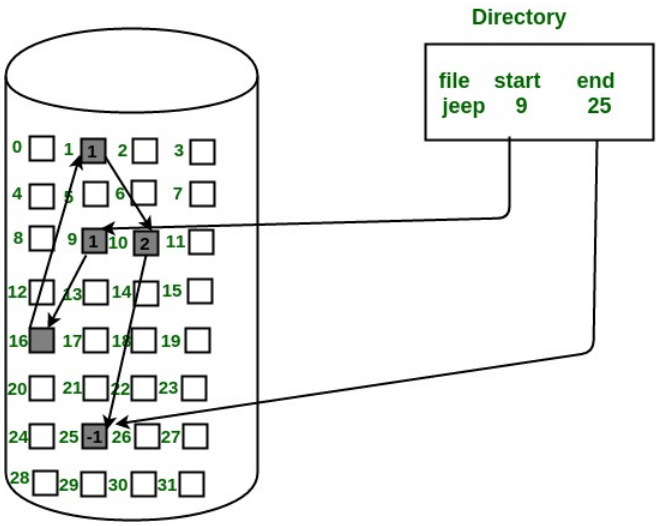
Part 3 – File Class

Create a file class to represent a file being stored on the hard drive.

- Properties
  - Filename
  - LinkedList containing integers
    - These will correspond to the indexes in the simulated hard drive where the "contents" are being store
- Methods
  - getFileName
    - returns the file name
  - addBlock(int)
    - Adds an index to the linked list
  - fileSize
    - Returns how big the file is (list size)
  - getFileBlocks
    - Returns the indexes that make up the file

When a file is stored on the "hard drive", each character of the contents string will be put in an index. We keep these indexes in a linked list in the File object so we can put the file back together when we want to.

You can't just store a start and end index because as files get added and removed from the "hard drive" the available indexes will become more jumbled hence the need for the linked list to keep track of it all.



Part 4 – The File Manager

Create a class named FileManager that will simulate the File Manager of the operating system and the hard drive.

Property	
-hardDrive:char*	A character array to simulate the hard drive
-blocksAvailable:Queue<int>	Stores the available blocks on the hard drive. Initially this will be every index in order.
-files:LinkedList<File>	Keeps track of all the file objects, sorted by file name
Method	
+addFile(name:string, contents:string)	Builds a new file, puts the contents on the hard drive and in the files list (remember sorted)
+deleteFile(name:string)	Deletes a file, put the chunks that file used to occupy back into the blocksAvailable queue
+readFile(name:string):string	Look up a file and return the contents of that file
+getFileNames():std::vector<string>	Get all the file names out as a vector
#findFileByName(std::string):File	A method to search for a File object and return it (or a pointer to it)

This is the most important class since it is the one that will primarily get tested.

The File Class is used to represent each individual file, the FileManager class organizes them all. It keeps track of all the existing files with a LinkedList that is sorted by filename and facilitates the addition and removal of files from the hard drive.

Part 5 – Simulation

To demonstrate our software prototype work we need to be able to perform the actions a user might want to do.

Process a command line argument.

The user should run the program with a -s <size> to set the size of the simulated hard drive.

The user should be able to run the program with a -f <file> to load an initial state of the hard drive.

The file will contain filenames and strings for contents.

<filename>:<contents>

Read each line and parse the filename and contents and add them to your file manager.

Example usages (if my executable is called exe):

```
exe -s <size>
```

exe -s <size> -f <file>

User interface

You will provide the user with a simple user interface that will let them add, remove or read a file.

```
Menu:
1 - Show files on hard drive
2 - Add a file
3 - Delete a file
4 - Output a file
0 - Exit simulation
```

Option 1

When the user selects option #1 – simply show all the filenames and the size of the files.

```
Filename                 size
Foo.txt                 5 blk
a.t                    15 blk
bob.test                4 blk
random.a              101 blk
```

Show the filename left justified using at least 40 characters worth of space. Show the block count right justified using at least 10 characters worth of space followed by "blk".

Hint: either use printf() or <iomanip>

Option 2

When the user selects option #2 – prompt them for a filename and a string for the contents.

```
Enter filename: <user inputs file name>
Enter content string: <user inputs content string>
```

After getting this information create a new file entry for the system. Store the characters of the content string one at a time in the hard drive.

Option 3

When the user selects option #3 – prompt them for a filename.

```
Enter filename: <user inputs file name>
```

Delete the file from the simulated hard drive.

Option 4

When the user selects option #4 – prompt them for a filename.

```
Enter filename: <user inputs file name>
```

Output the content string that exists on the simulated hard drive for that file.

Give 30 characters space and left justification for the file name. Set the file contents to be right justified.

Example:

```
Filename                       Contents
hRKz8qcv0i                     VrghdkYt43CYOHtsdjfbYSdEN6ElYhNuqnch7xX5FLty
```

Option 5

Exits the program.

NOTES

- I've left many implementations details up to you to decide. The FileManager class is the one that's primary interface. We'll be shoving data into it.
    - I'm letting you decide if you want to try to move/copy stack-objects or do everything on the heap.
    - If you want to approach this more like a Java project, then use pointer and allocate object to the heap.
        - Remember you have to clean up that memory.
        - This also might mean that your template might look weird.
            - LinkedList<File*>* list;
- You are possibly going to run into a lot of C++ nuance problems compared to Java
    - Depending on how you approach coding this you might need to build Copy Constructors for your classes.
    - You might need to build destructors.
- You can add whatever method you need to make this work. However, the core functionality must be present and you MUST use LinkedLists for the file indexes in the File object.

Grading of Programming Assignment

Your software will be graded through GradeScope. The specifications portion of the rubric will be determined by auto-grading tests. A grader will judge the "soft" skills portion of the rubric.

Rubric:

	Levels of Achievement						
Criteria	A	B	C	D	E	U	F
Specifications Weight 50.00%	100 % The program works and meets all of the specifications.	85 % The program works and produces the correct results and displays them correctly. It also meets most of the other specifications.	75 % The program produces mostly correct results but does not display them correctly and/or missing some specifications	65 % The program produces partially correct results, display problems and/or missing specifications	35 % Program compiles and runs and attempts specifications, but several problems exist	20 % Code does not compile and run. Produces excessive incorrect results	0 % Code does not compile. Barely an attempt was made at specifications.
Code Quality Weight 20.00%	100 % Code is written clearly	85 % Code readability is less	75 % The code is readable only by someone who knows what it is supposed to be doing.	65 % Code is using single letter variables, poorly organized	35 % The code is poorly organized and very difficult to read.	20 % Code uses excessive single letter identifiers. Excessively poorly organized.	0 % Code is incomprehensible
Documentation Weight 15.00%	100 % Code is very well commented	85 % Commenting is simple but solid	75 % Commenting is severely lacking	65 % Bare minimum commenting	35 % Comments are poor	20 % Only the header comment exists identifying the student.	0 % Non existent
Efficiency Weight 15.00%	100 % The code is extremely efficient without sacrificing readability and understanding.	85 % The code is fairly efficient without sacrificing readability and understanding.	75 % The code is brute force but concise.	65 % The code is brute force and unnecessarily long.	35 % The code is huge and appears to be patched together.	20 % The code has created very poor runtimes for much simpler faster algorithms.	0 % Code is incomprehensible

What to Submit?

You will submit the following files to GradeScope:

- LinkedList_<yourname>.hpp
- Queue_<yourname>.hpp
- FileSystem_<yourname>.h
  - Contains File and FileManager declarations
- FileSystem_<yourname>.cpp
  - Contains File and FileManager definitions
- stupidOS_<yourname>.cpp
  - This is where your main() and major interface logic should be
- Makefile
  - Compile the project to an executable named stupidos

Academic Integrity and Honor Code.

You are encouraged to cooperate in study group on learning the course materials. However, you may not cooperate on preparing the individual assignments. Anything that you turn in must be your own work: You must write up your own solution with your own understanding. If you use an idea that is found in a book or from other sources, or that was developed by someone else or jointly with some group, make sure you acknowledge the source and/or the names of the persons in the write-up for each problem. When you help your peers, you should never show your work to them. All assignment questions must be asked in the course discussion board. Asking assignment questions or making your assignment available in the public websites before the assignment due will be considered cheating.

*The instructor and the TA will **CAREFULLY** check any possible proliferation or plagiarism. We will use the document/program comparison tools like MOSS (Measure Of Software Similarity: http://moss.stanford.edu/) to check any assignment that you submitted for grading. The Ira A. Fulton Schools of Engineering expect all students to adhere to ASU's policy on Academic Dishonesty. These policies can be found in the Code of Student Conduct:*

http://www.asu.edu/studentaffairs/studentlife/judicial/academic_integrity.htm

ALL cases of cheating or plagiarism will be handed to the Dean's office. Penalties include a failing grade in the class, a note on your official transcript that shows you were punished for cheating, suspension, expulsion and revocation of already awarded degrees.