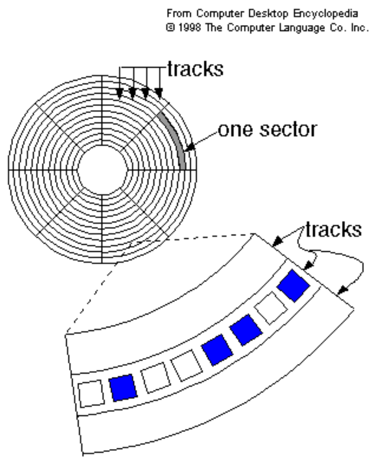
CS380 Algorithm Design & Analysis
Assignment 2: Disk Scheduling

Date Assigned: Friday, September 13, 2013
Part 1 Due: Monday, September 23, 2013 @ 11:59pm
Part 2 Due: Friday, September 27, 2013 @ 11:59pm
Total Points: 50pts (30 points for part 1, 20 points for part 2)

For this assignment you will implement a Heap and use that heap to simulate the scheduling algorithm of a hard drive. You will build this project in two stages: 1) a Heap for ints 2) a Heap for DiskIOs

For this assignment, we will consider the disk to be a single platter or surface with a read/write head that can move radially across the surface of the disk. The disk surface is divided up into a set of concentric circles called tracks and each track is divided into segments called sectors. Sectors are not important for this project.


From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.
tracks
one sector
tracks

Our disk for this assignment has 200 tracks (0 - 199) and 20 sectors (0 - 19) per track. By the time an I/O request arrives from the user to the disk, it takes the following form:

```
r1 = (id#,R/W,Sr,Tr)

where  id#  - is the user's unique identification number (0 - 47)
       R/W  - is the request to read or write to the disk
       Sr   - is the sector number (0 - 19)
       Tr   - is the track number (0 - 199)
```

You are to write a C++ program using object-oriented design that will test the following disk scheduling policy:

Shortest Seek Time First (SSTF) - This policy uses a priority queue (a min heap) of I/O requests keyed by the distance between the target track of the IO operation and the track on which the read head is currently positioned: key = abs(TargetTrack - CurrentTrack). This algorithm always chooses the IO request that will move the read head this shortest distance thereby spending less time seeking and more time transmitting data.

Notes:

- I have provided Heap.h, HNode.h, and ComparableItem.h
- Implement your heap stored as an STL Vector
    1. http://www.cplusplus.com/reference/vector/vector/
- This heap must be configurable as a Max or Min heap. The default is a Max heap. You must implement the definitions for all of the functions listed in a file called "Heap.cpp". Do not change any of the function names or data types.
    1. This Heap has a similar interface to SortableArray.
    2. This Heap implements all the common heap methods we discussed in class
    3. This Heap must also implement the Visitor design pattern (Heap::updateEachKey()). See the URL in the code for more information. This method must visit each item in the Heap and call HNode::updateKey() on each item in the Heap. After that, Heap::buildHeap() will be called.
- I have provided you with a file "HNode.h" containing an abstract class with pure virtual functions that must be overloaded.
    1. The HNode subclasses ComparableItem from the first assignment.
    2. HNode::changeKey() sets the key of the current node
    3. HNode::outputNode() outputs the contents of the current node.
    4. HNode::updateKey() will use the data passed into the method to calculate the new key.
    5. The operator<< for HNode must call HNode::outputNode()
- You must subclass "HNode.h" to hold the specific data required for the project.
    1. For Part One: a Heap of ints
    2. For Part Two: a Heap of DiskIOs (hint: the key for DiskIOs is also an int!)
- All of your output will be displayed to the screen for this assignment.
- Do NOT use structs in this assignment.

Warning: This is not a last minute assignment. You should begin working on this assignment today so that you have plenty of time to iron out any problems you may encounter.

This assignment will also hone your skills in object-oriented design. I encourage you to come and talk to me about the design of your program. Part of your grade will be allocated to how well you designed your program.

Bring your design to me for review!

What to Complete for Part 1 Integer Heap

- Implement the complete class Heap to perform all of the standard heap functionality.

- You should have thought ahead to Part 2 and designed the entire set of classes at this point.
- Subclass the HNode class to hold the data needed for an integer priority queue and implement the virtual functions (IntNode). You can add any other functions that you might need.
- Write a driver to test your implementations. The driver must do the following:
    1. Read the following integers one at a time from a file and insert them into a **Max** Heap: <5, 3, 17, 10, 84, 19, 6, 22, 9>.
    2. Display the Heap (using <<)
    3. Display the result of calling the function **`heapExtreme`**.
    4. Insert the value 20
    5. Display the Heap (using <<)
    6. Extract each item one by one, printing them to the screen. You should receive the items in descending order.
    7. Do all of the above steps again, except for a **Min** Heap.

What to Submit for Part 1

- Name your project 02DiskIO_PUNetID.

- Your project MUST be in your Subversion repository by 11:59pm on the day that it is due. I expect to see a large number of commits to your repository and useful log messages.
- Submit a hard copy of the files starting with the file containing main, followed by the other classes where the header file of a class is always just before the cpp file.
- A summary of the time that you spent working on this assignment, and what slowed you down the most. Put this in a text file in your Visual Studio project.
- **IF YOU GET THIS PORTION DONE EARLY:** Email me the revision number that contains your solution to part 1. And then start on Part 2. I will pick that revision out of Subversion rather than just checking out the current version at 11:59pm.

What to Complete for Part 2 Disk IO Heap

- Use a **Min** heap for Disk IO Scheduling.

- You should not need to modify your Heap.cpp for this porition!

  The DiskIO Scheduling Algorithm you must implement

- **First**, initialize your request queue from the data in the file "init.dat". This will give the effect of having several I/O requests queued up. **Next** you are to read a line of data from the file "diskio.dat", queue up the request using the disk scheduling policy, and **then** process a request (the root of the min heap). **Update** the keys in the min heap based on the current location of the head (using Heap::updateEachKey()). Proceed in this manner until the request queue is empty and the file is at EOF. You will encounter the EOF first.
  [This loop is very similar to the loop in airport from CS 300. You need to track how long each DiskIO request is in the queue. You may choose to have a clock tick every time around the loop and track at which clock tick each DiskIO request entered and exited the queue.].
- Assume that the read/write head starts at track 0.
- The "init.dat" file requests' id numbers will be in the 0-47 range. Further, this file could be empty, but you don't need to error check the ints you are reading, they will all be valid.
- The files can contain multiple requests with the same id.
- Assuming that the disk head is on track 100 and needs to go to 105, then the tracks crossed is 5.
- The wait time is simply the number of requests the current request had to wait before being processed. The calculation of wait time does not include tracks crossed.
- Do not take the sector number into account when determining priority. If two requests have the same priority, take the request that was queued up first.

Your program must use the data found in the files "init.dat" and "diskio.dat" and report on the following:

- The total number of requests that were processed.
- The total number of tracks the read/write head had to move to process all of the I/O requests.
- The average number of tracks the read/write head had to move for each I/O request.
- The maximum number of requests that any of the I/O requests had to wait before being scheduled. Include the identification number of the I/O request that had the longest wait.
- Show what happened at each step of the disk scheduling policy. Headings should be as follows:

```
ID#    R/W     Sector     Track     Tracks Crossed     Wait Time
---    ---     ------     -----     --------------     ---------
```

DiskSchedulingOutput.txt shows the correct first few lines of output.

What to Submit for Part 2

- Your project MUST be in your Subversion repository by 11:59pm on the day that it is due. I expect to see a large number of commits to your repository and useful log messages.
- Submit a hard copy of the files starting with the file containing main, followed by the other classes where the header file of a class is always just before the cpp file. All the files must be submitted, even if they haven't changed from part 1.
- Class diagram
- A summary of the time that you spent working on this assignment, and what slowed you down the most. Add this to the text file from part one.

Notes:

A well written Heap will allow you to reuse code later in the semester!

The first milestone is a MINIMUM goal. You really should be done with the first milestone before the Exam!

Starter code: http://zeus.cs.pacificu.edu/chadd/cs380f13/Student_Heap.zip

(Win32 Console Application, NO Precompiled Headers, Empty Project)

More about Disk Scheduling:

Interactive time-shared computer systems often encounter a problem in scheduling I/O tasks for a magnetic disk. It is common to find such systems serving many terminals but having only one, or perhaps a few magnetic disk drives.

For example, such a system might service 200 terminals at any one time and have a single magnetic disk of several hundred gigabytes capacity. At any time the users could be running programs, compiling programs, and text editing where each of these activities could and do generate a mix of I/O requests.

The servicing of these requests can create a bottleneck that substantially affects the system performance. The I/O requests generated by all of the computing activities of the users are intermixed and at any time the disk can expect to have a group of I/O requests waiting for execution. Since disks can only handle one request at a time, the requests must be put into some order for service. The order chosen, called the disk-scheduling policy, is the subject of this assignment.