I/O-Efficient Merge Sort*

Due: Wednesday, September 28, 2011
Turn in at the start of class.

Overview

In this assignment you will implement an I/O-efficient merge sort. You will explore several different ways to move data to/from disk using streams. The project should be done in groups of 2 (if we have an odd number, I will allow 1 group of 3). It should be programed in C or C++ on a Unix system and tested on a system that has a hard drive (no Mac Book Airs or similar flash memory only devices).

The assignment will be graded on a report that you will write describing the implementation and the experimental work. We will inspect the source code as needed. Hand in the report at the start of class Wednesday, September 28.

The project will use a *heap*, and implement *heapsort* and *quicksort*. Only for these aspects are you allowed to use existing code; all other code must be written for the project. For example you can use the code from the following book:

Robert Sedgewick: *Algorithms in C, Parts 14*, third edition. Addison-Wesley, 1998, ISBN 0201314525

and can be found online at `www.cs.princeton.edu/~rs/Algs3.c1-4/code.txt`

Attribution: This project is based heavily off of a project assigned by Lars Arge (`www.daimi.au.dk/~large/ioS11/project1.pdf`), with permission.

1 Tasks

Follow the outline of tasks below and document your finding in a report. Following this outline in the report will make grading easier :).

1.1 Streams

Your merge sort implementation will use streams. You will need two types: *input streams* and *output streams*. The input stream should support the following operations: `open` (opening an existing stream for reading), `read_next` (read the next element from the stream), and `end_of_stream` (return `true` if the end of the stream has been reached). The output stream should support the following operations: `create` (create a new stream), `write` (write an element to an existing stream), and `close` (close the existing stream).

Use the following I/O-mechanisms to implement streams in four different ways. In each implementation, the actual data should be stored in a simple file (i.e. not just in the program memory).

(a) Read and Write using one element at a time with `read` and `write` system calls.

(b) Read and Write using `fread` and `fwrite` functions from `stdio` library (this library has implemented its own buffering mechanism for these functions).

(c) Read and Write, are implemented as in (a), except now the stream is equipped with a buffer of size B in internal memory. Whenever the buffer becomes empty/full the next B elements are read/written from/to the file.

(d) Read and Write is performed by mapping and unmapping a B element portion of the file into internal memory using `mmap`/`munmap`.

Documentation for these functions and system calls can be easily found online with an appropriate search.

Experiment with each stream implementation, and determine a "winner." The goal is to determine which works the best and to clearly document the process of this discovery in the report. What are the limitations and advantages of each implementation. For example:

- Try opening k streams and read/write one element to/from each of the streams N times. Try for large N and $k = 1, 2, 4, 8, \ldots$ MAX where MAX is the maximum number of streams allowed by the operating system.
- For (c) and (d) experiment with different values of B (including very large ones). Identify the optimal values.

1.2 d-Way Merge

Implement a d-way merging algorithm; given d sorted input streams of 32-bit integers create a single output stream containing the elements from the input stream in sorted order. The merging should be based on the priority queue structure *heap*.

1.3 External MergeSort

Implement an external memory Mergesort algorithm for sorting 32-bit integers. The program should take an input file and parameters N, M, and d as input and sort as follows.

(a) Read the input file and split into $\lceil N/M \rceil$ streams, each of size at most M. As each stream that is created, it should be sorted in internal memory using *quicksort* before writing to disk.

(b) Store the references to the $\lceil N/M \rceil$ streams in a queue (if necessary in external memory).

(c) Repeatedly until only one stream remains, merge the d (or fewer) first streams in the queue and put the resulting stream at the end of the queue.

1.4 External MergeSort

Experiment with your *mergesort* implementation using the best stream implementations from 1.1 and with random 32-bit integers. Try different values of N, M, and d. Identify what are good choices for these parameters for various input sizes (including very large ones). Document everything you discover in your report.

1.5 External MergeSort

Implement the standard (interal memory) *heapsort* and *mergesort* algorithms. Compare against the best of your external memory *mergesort* algorithms. Discuss what you find in your report.

1.6 Code Base

Provide a pointer to your code base so we can inspect the source code.

Make sure not to modify this code prior to receiving a grade so the time stamps can be verified.

Describe the system(s) that you used for the experiments. The optimal values of B and M may change as you switch systems.

2 Hints

1. Run your programs a suitable number of times for each input size and use the average running time. This should level out fluctuations due to other processes.

2. There are quite a number of UNIX commands to time the execution of the program. Among these are `time` and `timex`. In some shells (e.g. csh), there are also built-in version of `time`. There are also corresponding C library routines called `times`, `getrusage`, and `gettimeofday`. The availability may differ from machine to machine.

3. Do not create files in an NFS mounted directory. The files could be created in the directory `/tmp/<username>` (i.e. on the local harddisk) to avoid the files to be sent over the network. Be careful to check that the disk has sufficient space left for running your program. Always leave significant additional disk space unused.

4. Do not run experiments on file servers! Rather, try to find a machine with no other user processes executing. A personal laptop is fine as long as it has a hard drive.

5. Check the free disk space before creating huge files, e.g. using the `df` command. Remember to remove the created files when you are finished.

6. Use C or C++. Past students attempting this assignment in Java and other languages encountered numerous problems.