GSB/RF
October 1, 2003

Project 2 — Sorting

This is the second project in the course *External Memory Algorithms and Data Structures*. The project should be done in groups of 2–3 persons, and the report should be handed in by *Wednesday, November 5, 2003*. The project should be programmed in C or C++ on a Unix system.

In this project, we study the behavior of various sorting algorithms when used in external memory. More specifically, the goal of the project is to

1. Use different I/O approaches to manipulate *streams* of data to/from external memory.
2. Develop an efficient algorithm for merging several streams of data to/from external memory.
3. Implement multiway *Mergesort* in external memory.
4. Implement *Radixsort* in external memory.
5. Compare these implementations with the standard *Heapsort* and *Quicksort* algorithms.

As part of the project, you will need to implement a *Heap*, as well as *Heapsort* and *Quicksort*. To keep the project down in size, and to increase comparability of results between different projects, we require you to use the code in the book

> Robert Sedgewick: *Algorithms in C, Parts 1–4*, third edition. Addison-Wesley, 1998, ISBN 0201314525.

This code can be found online at

`www.cs.princeton.edu/~rs/Algs3.c1-4/code.txt` .

Tasks:

1. Write a program which takes two arguments n and *filename* and creates a file named *filename* containing n random 32-bit integers.
2. The implementation of *Mergesort* and *Radixsort* should use the concept of *streams*. There should be two different kinds of streams: *input streams* and *output streams*. An input stream should at least support the operations `open` (open 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). An output stream should support the operations `create` (create a new stream), `write` (write an element to an existing stream), and `close` (close the existing stream).

   Make five implementations of streams, each using a different one of the following four I/O mechanisms. In all four cases, the actual data of the stream should be stored in a simple file.

   (a) Reading and writing is done one element at a time by the `read` and `write` system calls.

   (b) Reading and writing is done by the `fread` and `fwrite` functions from the `stdio` library. These implement their own (fixed) buffering mechanism.

(c) Reading and writing is handled as in (a), except that now the stream is equipped with a buffer in internal memory of size B, and whenever the buffer becomes empty/full, the next B elements are read/written from/to the file.

(d) Reading and writing is handled as in (c), but now trying to parallelize CPU work and I/Os by using *double buffering*: each stream has two buffers, where one is transfered from/to disk while the other is used for reading/writing the stream. When the latter is empty/full and the transfer of the former has completed, the roles of the buffers are interchanged. For this to work, you must use asynchronous I/O or threads.

(e) Reading and writing is handled by mapping the file containing the stream to internal memory using `mmap` and scanning the stream as if it was an array in internal memory.

For each of the implementations (a), (b), and (e), as well as for (c) and (d) with various values of B (including very large ones), perform the experiment of opening k streams and n times read/write one element to/from each of the streams. For each implementation, do this for a large n and for $k = 1, 2, 4, 8, \ldots, \textsc{Max}$, where $\textsc{Max}$ is the maximal number of streams allowed by the operating system. For each of the four stream implementations, identify their properties and limitations, and try to single out a winner.

3. Implement a d-way merging algorithm that given d sorted input streams creates an output stream containing the elements from the input streams in sorted order. The merging should be based on the priority queue structure *Heap*.

4. Implement a *Mergesort* algorithm for sorting 32-bit integers. The program should take parameters n, m, and d, and should proceed by the following steps.

   (a) Read the input file and split it into $\lceil n/m \rceil$ streams, each of size $\leq m$. Each stream that is created should be sorted in internal memory using *Quicksort* before writing it to external memory.

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

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

5. Implement a version of the *Mergesort* algorithm above which tries to parallelize CPU work and I/O in the first phase by restricting the initial sorted streams to be of length $\leq m/2$, and then sorting one stream while transferring another to/from disk. For this to work, you should use threads.

6. Implement a *Radixsort* algorithm for sorting 32-bit integers. The program should take parameters n and s, and should proceed by the following steps.

   (a) Scan the input while distributing the elements into 2^s streams according to the s least significant bits of the integers.

   (b) Repeatedly concatenate the streams, and distribute the elements according to the next s bits of the integers. For the last distribution, less than s bits may be left.

   (c) Concatenate the last set of streams into one.

   Note: It is probably more efficient just to read from the previous set of streams in order, without actually concatenating them first.

7. Perform experiments with the two *Mergesort* and the *Radixsort* programs, using the best of the stream implementations from above. The data should be random 32-bit integers. Try different values for n, m, d, and s, and identify what are good choices of m, d, and s for the various sizes.

8. Implement the standard *Heapsort* and *Quicksort* algorithms which sorts an array of n integers.

9. For various sizes of data (in particular sizes too large for main memory), compare the running time of the best of your *Mergesort* algorithms, the best of your *Radixsort* algorithms, the *Heapsort* algorithm and the *Quicksort* algorithm.

Hints:

1. Check the free disk space before creating the huge files, e.g. using the `df` command.

2. 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 send over the network. Be careful to check that the disk used has sufficient space left for running your program. Always leave significant additional disk space unused.

3. Remember to remove the files created.

4. Read the hints of Project 1 again.