

The goal of this project is to implement and experiment with I/O-efficient merge-sort using several different ways of manipulating *streams* of data to/from disk. The project should be done in groups of 2–3 people and programmed in C or C++ on a Unix system. An important part of the project is to write a report that describes the implementation and (especially) the experimental work; The report (including a pointer to the implementation) should be handed in by *Friday March 11, 2005*. The evaluation of the implementation, experimentation and the report will be part of the final grade.

Part of the project will use a *Heap*, as well as *Heapsort* and *Quicksort*. Use the code from the following book for these

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

This code can also be found online at

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

Tasks:

1. The implementation of *Merge-sort* 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).

Implement streams in four different ways using the following different 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 performed one element at a time using the `read` and `write` system calls.
- (b) Reading and writing is performed using the `fread` and `fwrite` functions from the `stdio` library (these functions implement their own buffering mechanism).
- (c) Reading and writing is performed as in (a), except that now the stream is equipped with a buffer of size B in internal memory and whenever the buffer becomes empty/full the next B elements are read/written from/to the file.
- (d) Reading and writing is performed by mapping and unmapping a B element portion of the file into internal memory using `mmap/munmap`.

Experiment with each of the stream implementation. For example, try opening k streams and read/write one element to/from each of the streams N times; Try it for a large N and for $k = 1, 2, 4, 8, \dots, \text{MAX}$, where `MAX` is the maximal number of streams allowed by the operating system. For (c) and (d) also try different values of B (including very large ones). For each of the four stream implementations, identify their properties and limitations and try to single out a winner.

2. Implement a d -way merging algorithm that given d sorted input streams of 32-bit integers 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*.
3. Implement external *Merge-sort* for sorting 32-bit integers. The program should take parameters N , M , and d , and sort as follows:
 - (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 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 less) first streams in the queue and put the resulting stream at the end of the queue.
4. Perform experiments with *Merge-sort* using the best of the stream implementations from above and random 32-bit integers. Try different values for N , M , and d , and identify what are good choices for these parameters for various input sizes.
5. Implement the standard *Heapsort* and *Quicksort* algorithms for sorting an array of N integers.
6. Compare the best of your *Merge-sort* algorithms with the *Heapsort* and *Quicksort* algorithms.

Hints:

1. Run your programs a suitable number of time 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 a program, among these `time` and `timex`. In some shells (e.g. `csh`), there is also a 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, though.
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 send 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.
5. Check the free disk space before creating huge files, e.g. using the `df` command. Remember to remove the created files.