GSB/RF
September 17, 1999

Project 1 — Sorting

Part B

This is the second and last part of the first project. The report on the first project (part A and part B) should be handed in by Tuesday October 5.

The second part of the first project is about implementing the *Mergesort* algorithm. The goal of this part of the project is

1. to use different I/O approaches to manipulate *streams* of data from external memory.
2. to develop an efficient algorithm for merging several streams of data to/from external memory.
3. to implement multiway *Mergesort* in external memory.

The project should be programmed in C or C++.

Tasks:

1. Write a program that takes two arguments n and *filename* and creates a file named *filename* containing n random integers (i.e. the size of the created file should be $4n$ bytes, assuming an integer requires 4 bytes).
2. The merge-sort implementation should be using the concept of *streams*. There should be two different kinds of streams: *input-streams* and *output-streams*. An input-stream 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 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 three implementations of streams using the following three I/O mechanisms. In all three cases the actual data of the stream should be stored in a simple file.

   (a) Reading and writing are done by the standard read and write operations on a Unix file (`stdio` library), one element at a time.

   (b) An input/output stream is handled as above, except that now the stream has 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.

   (c) Map the file containing the stream to internal memory using `mmap` and scan the stream as if it was an array in internal memory.
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.
4. Implement a *Mergesort* algorithm for sorting integers. The program should take parameters n, m, d, and B, 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 before writing it to external memory.

(b) Store references to the files containing 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 to the end of the queue until only one stream remains.

5. Perform experiments with the *Mergesort* program using the three different stream implementations and using different values for n, m, d, and B. The data should be random integers. Identify the properties and limitations of each of the the three stream implementations and identify in each case what are good choices of m, d, and B. Compare the running time of your *Heapsort* algorithm from Part A with th2e best of your *Mergesort* algorithms for various sizes of data (including sizes too large for main memory).

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 some additional disk space unused.

3. Remember to remove the files created.