Middle East Technical University | Department of Computer Engineering

CENG 334

Introduction to Operating Systems

Spring 2025-2026

Homework 1 - Merge Operator

Due date: March 23, 2026, Monday, 23:59

1 Overview

In this homework, you will implement a **merge operator controller** program (the *merger*) that orchestrates three distinct operators—sort, filter, and unique—on CSV data. The controller reads a specification from standard input, spawns operator processes in pipelines, and merges their outputs. The overall structure forms an n-ary tree: merger nodes at internal nodes and operator chains at the leaves. You will use Unix system calls for process creation (fork), execution (exec), pipes (pipe), and I/O redirection (dup2) to build this system.

Keywords: *unix, processes, pipes, CSV, fork, exec*

2 Background

2.1 Pipelines and Processes

A **pipeline** is a sequence of processes chained together so that each process's standard output (stdout) is connected to the standard input (stdin) of the next process. You will use **bidirectional pipes** for inter-process communication. A bidirectional pipe is created with `socketpair(2)` (Unix domain stream sockets): it returns two file descriptors, and *each* descriptor can be used for both reading and writing. Thus one process can read from one end and write to the other, or use the same pair for two-way communication.

Define a macro for creating a pipe:

```
#define PIPE(fd) socketpair(AF_UNIX, SOCK_STREAM, PF_UNIX, fd)
```

After `PIPE(fd)`, `fd[0]` and `fd[1]` are two connected endpoints. For *unidirectional* use (e.g., parent writes, child reads): the parent writes to `fd[1]` and closes `fd[0]` (or keeps `fd[0]` for reading in another process); the child reads from `fd[0]` and closes `fd[1]`. Use `dup2(2)` to replace stdin or stdout with one end of the pair so that the child process simply reads from stdin and writes to stdout. Close any pipe end you do not need in each process to avoid deadlocks and to ensure the reader sees EOF when the writer finishes.

When you run a command like `sort | filter` in a shell, the shell creates one pipe (or socket pair), forks two child processes, redirects the first child's stdout to one end and the second child's stdin to the other, and then executes `sort` and `filter`. Both processes run **concurrently**; the shell waits for both to finish using `wait(2)` (or `waitpid(2)`). In this homework, your merger will set up the same kind of pipelines between the provided operators.

2.2 What the Merger Does

The name *merger* in this homework does **not** mean merge-sort, and you should not assume that the final output is globally sorted unless the specification explicitly causes that. The merger is simply the coordinator: it splits the CSV input into ranges, assigns each range to a *chain* of operators (e.g., sort then filter), starts those chains as pipelines of processes, and combines the produced outputs. The operators themselves read only from stdin and write only to stdout; they do not open the CSV file. Only the **main merger** program reads the CSV file; sub-mergers receive their data on standard input.

2.3 Tree Structure

The specification can describe a **recursive** structure: a chain may consist of a single "merger" operator, which is followed by a sub-specification (without a filename). That sub-specification describes another merger with its own chains. Thus the overall structure is an **n-ary tree**: each node is either a merger (with children that are chains) or a leaf (an operator chain). Merger nodes merge the outputs of their children; leaves are pipelines of sort/filter/unique that produce one stream each.

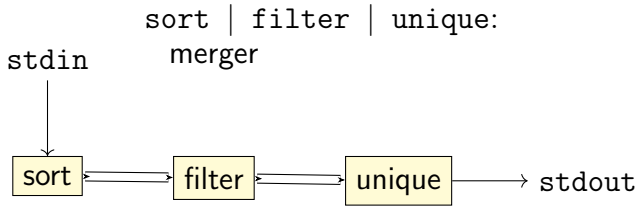

Figure 1: Operator Chain Pipeline

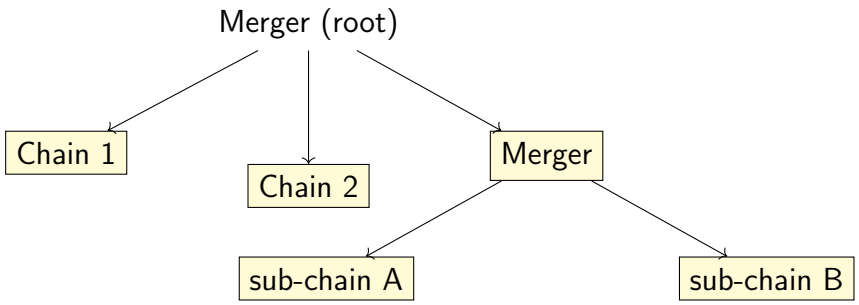

Figure 2: N-ary Merger Tree

3 Input Format

The controller reads its input from **stdin** line by line. The first line describes the top-level merger and the CSV file; the following lines describe each operator chain (or, in the recursive case, a sub-merger). All line numbers in chain lines are **1-based** and **inclusive**.

3.1 First Line (Top-Level Only)

```
filename num_chains
```

- `filename`: Path to the CSV file. Only the **main** merger program opens this file; sub-mergers receive their data on standard input.
- `num_chains`: Number of operator chains that follow. The next `num_chains` lines each describe one chain.

The top-level merger reads the **entire CSV file**. It does not take a merger-level start/end range. Range selection is done only by the chain lines that follow.

3.2 Operator Chain Format

Each chain is given on **one line**. To avoid confusion with operator arguments, **the line range always comes first**, then the pipeline (shell-like, with `|` between operators):

```
start_line end_line operator1 args1 | operator2 args2 | ... | operatorN argsN
```

Example:

```
5 50 sort -c 2 -t num | filter -c 2 -ge 36
```

This chain is responsible for CSV lines 5 through 50 (inclusive). It runs `sort` on column 2 (numeric), then pipes the result to `filter`, which keeps rows where column 2 is greater than or equal to 36.

- `start_line` and `end_line`: Inclusive range of CSV line numbers that this chain will process. The merger will read these lines from the file and feed them to the first operator's stdin.
- The rest of the line is a pipeline: operator names and their arguments, separated by spaces, with `|` between operators (as in a Unix shell). Each operator (sort, filter, unique) has its own arguments (e.g., `-c 2 -t num`).

Size limits:

- The entire operator chain (one line) must fit in 10240 characters.
- Each single element (operator name or one argument value) is at most 129 bytes.

3.3 Recursive Case

A chain may consist of a single operator named `merger`. In that case, the chain does not run a pipeline of sort/filter/unique; instead, the parent merger starts a sub-merger. You may realize this either by forking a child and then executing the **same merger binary again**, or by recursively running the same merger logic without an exec. The sub-merger has **no filename** in its specification; the parent writes the sub-input to the sub-merger's stdin, then writes the CSV lines for that chain's range to the same stdin. The sub-merger processes **all CSV lines it receives on stdin**; it does not take a merger-level start/end range.

Format of a chain that is a recursive merger:

```
start_line end_line merger
```

The **next line** is the sub-merger's first line (no filename):

```
num_chains
```

Then come `num_chains` lines, each describing a sub-chain (each of which may again be an operator pipeline or a recursive merger). This recursion builds the n-ary tree.

Important: In sub-merger input, the line numbers in each chain line are **relative to the sub-merger's own CSV block**. In other words, when the parent sends only lines 51–100 to a sub-merger, the sub-merger treats the first received CSV line as local line 1, the second as local line 2, and so on. This makes it natural to implement a sub-merger either recursively or by executing the same binary again as a separate process.

Example with recursion:

```
data.csv 2
1 50 sort -c 1 -t text
51 100 merger
2
1 25 filter -c 2 -t num -ge 30
26 50 filter -c 2 -t num -l 30
```

The top-level merger has two chains and reads the whole file. The first chain (lines 1–50) runs `sort`. The second chain (lines 51–100) is a sub-merger with two chains over its *local* block: one for local lines 1–25 (filter ≥ 30) and one for local lines 26–50 (filter < 30). The sub-merger merges those two streams and sends the result to the top-level merger.

3.4 Example Input (No Recursion)

```
data.csv 2
5 50 sort -c 1 -t num -r | filter -c 2 -t text -g "threshold"
51 100 filter -c 2 -t num -ge 36
```

Top-level: file `data.csv`, two chains. First chain: lines 5–50, sort by column 1 (numeric, reverse) then filter. Second chain: lines 51–100, filter only.

Additional, more complex input examples (for example, longer three-stage pipelines and inputs with four or five top-level children) will be posted on the course website.

4 Provided Parser

We provide parser source code (`merger_parser.c`, `merger_parser.h`). You compile these sources directly with your merger; no separate library build is required. The parser reads the input line by line and builds a tree of **merger nodes** and **operator chains**. Each merger node has a list of chains; each chain is either a pipeline of operators (sort, filter, unique) or a recursive merger (a single "merger" operator followed by a sub-specification). The parser uses fixed-size buffers; see the header for limits (`CHAIN_BUF_SIZE`, `ELEMENT_MAX_SIZE`, `MAX_OPERATORS`, `MAX_CHAINS`).

4.1 API overview

`parse_merger_input(FILE *f)` reads from `f` (e.g., `stdin`) and returns the root merger node, or `NULL` on parse error. It consumes the first line and then `num_chains` chain lines; when a chain is "`merger`", it recursively parses the sub-input (without a filename). `free_merger_tree(root)` frees the entire tree, including all sub-merger nodes. The exact data structures (`merger_node_t`, `operator_chain_t`, `operator_t`) are defined in the provided header; you do not need to implement the parser yourself.

5 Operator Specifications

The operator programs `sort`, `filter`, and `unique` are **provided to you**. In this homework, you do **not** implement these operators; you execute them with the correct arguments and connect their stdin/stdout correctly. Operators **do not read the CSV file**. They read all input from **standard input** and write to standard output. The merger opens the file, selects the line range for each chain, and writes those lines to the first operator's stdin. Thus operators are pure stream processors: they only see a stream of CSV lines and their command-line arguments (column index, type, comparison, etc.).

All operators share two mandatory concepts:

- `-c #` or `--column=#`: Column ID (1-based). The operator acts on this column.
- `-t text`, `-t num`, or `-t date`: Type of the column (for comparison and type checking). If a row has a value in that column that does not match the type (e.g., non-numeric in a `num` column), the operator must discard that row and log a message to stderr.

5.1 sort

Sorts the input by the given column. Comparison rules: **text** and **date** use string comparison (e.g., `strcmp`); **date** should follow ISO 8601 (YYYY-MM-DD). **num** uses numeric comparison (e.g., `strtod` then compare).

- Optional `-r` or `--reverse`: Descending order.
- Sort must see the full input before producing output: read until EOF, then sort, then write. It cannot stream.

5.2 filter

Keeps only rows that satisfy a comparison on the given column. The comparison is given by one of `-g` (greater), `-l` (less), `-e` (equal), `-ge`, `-le`, `-ne`, followed by a value (text, number, or date according to `-t`). Filter can process line-by-line: read a line, compare the column value, output the line or skip it.

- Type mismatch: if the value in the column cannot be interpreted according to `-t`, discard the row and log to stderr.

5.3 unique

Removes duplicate rows with respect to the given column: among rows with the same value in that column, keep only the **first** occurrence. Can be implemented by streaming (e.g., keep a set of seen values and output only when the value has not been seen before).

6 Execution

The **main program (merger)** is the only one that reads the CSV file, and it reads the **whole file**. For operator chains, it selects the relevant lines and feeds them to the first operator's stdin. For a chain that is a **sub-merger**, the main merger provides the sub-input (spec) and exactly the CSV lines for that chain's range to the sub-merger (e.g., via a pipe to the sub-merger's stdin). The sub-merger then processes **all CSV lines it receives** and interprets its chain ranges relative to that local block. You may implement the sub-merger by forking and executing the merger program again, or by running the sub-merger logic in the same process (e.g., recursively); **exec is not required** for sub-mergers. A common way to distinguish the two modes when using the same binary is to inspect the command-line arguments or another explicit mode indicator (for example, one mode for top-level execution with a filename argument and another mode for sub-merger execution without one). The **operators** do not read the file; they only read from stdin. The merger combines chain outputs in **chain order** (order of appearance). If you keep a list of currently readable child outputs, then whenever one child reaches EOF you should remove that child from the active read list and continue with the remaining open outputs until all are exhausted. It must reap all children (no zombie processes). At the end, the merger must print an **exit status block** to **stdout** so that it can be distinguished from the merged data. Format (only for **operator** processes, not sub-mergers):

- **Only operator processes** (sort, filter, unique) are reported; sub-merger processes are not.
- One line per exiting operator, in chain order and operator order within each chain. Each line has the form: `EXIT-STATUS <pid> <status>` (the literal `EXIT-STATUS`, a space, the process id, a space, and the exit status integer). So **every line** in the block starts with `EXIT-STATUS`.

Example: two chains, first chain has sort and filter (two operators), second has one filter; all exit 0:

```
EXIT-STATUS 12345 0
EXIT-STATUS 12346 0
EXIT-STATUS 12347 0
```

The merged CSV output appears first on stdout; then these lines. Grading compares the data part exactly and the status values (ignoring pids) in order.

6.1 High-Level Steps

1. Parse the specification from stdin using the provided `parse_merger_input(stdin)`. You get a tree rooted at a `merger_node_t`.
2. For each chain of this node (index `i = 0 .. num_chains-1`):
   (a) If this chain is a recursive (sub-)merger: arrange for the sub-merger to receive its input (the sub-spec and the CSV lines for this chain's range). You may do this by forking a child and connecting its stdin to a pipe you write to, then writing the sub-spec and CSV lines and closing the pipe; the child then runs the sub-merger (e.g., by executing the merger program or by calling the same logic recursively). In the parent, read the sub-merger's output from the other pipe until EOF, then wait for the child and store its exit status.
   (b) Else (operator pipeline): Create a pipe to feed input to the first operator. Create one process per operator and connect them so that the first reads from the feed pipe and each subsequent one reads from the previous operator's output; the last operator's output goes to a pipe the merger reads from. In the parent, write the CSV lines for this chain's range to the feed pipe and close it. Wait for all operator processes and store the last operator's exit status.

3. Read from each chain's output pipe **in order** (chain 0, then chain 1, ...) and print the concatenated stream to stdout.

4. Reap all child processes (ensure no zombies). Print the exit status block to stdout in the exact format given above. Free the tree with `free_merger_tree`.

Important synchronization note: Your design must allow writing input into a chain/sub-merger and reading output from it to progress **concurrently**. Otherwise, for large inputs, a writer may block because the other side is not being drained yet, which can lead to deadlock. A straightforward way is to use an extra helper process so that one side feeds input while another side consumes output and performs the merge.

6.2 Bidirectional pipes and pipeline setup (Instructional)

Use a bidirectional pipe: create a channel with two ends; each end can be used for reading or writing. For one-way data flow, assign one end to the writer and the other to the reader. In every process, close any end you do not use so that readers see end-of-input when the writer is done.

Operator pipeline (one chain), K operators:

```
create pipe P_feed (merger writes CSV lines here)
create pipe P_out  (last operator writes result here)
for i = 0 to K-2: create pipe P_i (connects operator i output to operator i+1 input)
for i = 0 to K-1:
    create new process
    in that process:
        make stdin  = read end of (P_feed if i=0 else P_{i-1})
        make stdout = write end of (P_i if i<K-1 else P_out)
        close all other pipe ends in this process
        run operator i
in merger: write CSV lines to P_feed, close P_feed
in merger: after all operator processes finish, read result from P_out
```

Sub-merger (one chain):

```
create pipe P_to_sub   (merger writes spec + CSV here)
create pipe P_from_sub (sub-merger writes merged result here)
create new process
in that process:
    make stdin  = read end of P_to_sub
    make stdout = write end of P_from_sub
    close other ends in this process
    run sub-merger
in merger: write spec and CSV lines to P_to_sub, close it
in merger: read from P_from_sub until EOF, then wait for child
```

In practice, the important point is not the exact mechanism but that feeding input and consuming output happen concurrently enough to avoid blocking on full pipes.

6.3 Process Diagram (Example)

The following diagram illustrates execution for an example input. The main merger reads `data.csv`, has two chains: Chain 1 is responsible for **lines 1–50** and runs `sort | filter`; Chain 2 is responsible for **lines 51–100** and is a sub-merger with two sub-chains over its *local* input block: one for **local lines 1–25** (filter) and one for **local lines 26–50** (filter). Only the main merger reads the file; it sends the sub-spec and CSV lines 51–100 to the sub-merger's stdin. Outputs are combined in chain order.

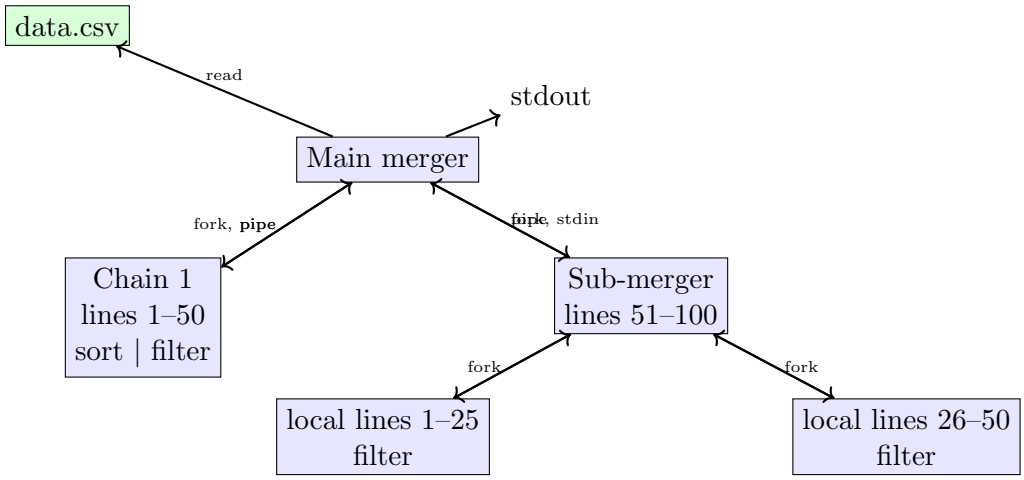

Figure 3: Execution: main merger reads `data.csv` and forks Chain 1 (lines 1–50, operator pipeline) and a sub-merger (lines 51–100). The sub-merger receives its spec and CSV on stdin and forks two filter chains over its local block (local lines 1–25 and 26–50); it combines their output and sends it to the main merger.

7 Regulations

- **Programming Language:** Your program should be coded in C or C++. Your submission will be compiled with `gcc` or `g++` on department lab machines. Make sure that your code compiles successfully.
- **Late Submission:** Late submission is allowed but with a penalty of $5 \times day \times day$.
- **Cheating:** Everything you submit must be your work. Any work used from third-party sources will be considered cheating and disciplinary action will be taken under the "zero tolerance" policy.
- **Newsgroup:** You must follow the ODTUClass for discussions and possible updates daily.
- **Grading:** This homework will be graded out of 100 and will make up 10% of your total grade. Grading is **lab exam based**: your grade for this homework is determined by the lab exam. Students who miss the lab exam will receive 0 from the homework as well.
- **Testing:** Testing will be **black-box**. Do not print any unnecessary output to stdout; grading is done by comparing your program's **stdout** to the expected output. If you need to print during debugging, use **stderr** only.

8 Submission

Submission will be done via ODTUClass. Create a tar.gz file named `hw1.tar.gz` that contains all your source code files with a makefile. The tar file should not contain any directories! The make should create an executable called `merger`. Your code should be able to be executed using the following command sequence.

```
$ tar -xf hw1.tar.gz
$ make
$ ./merger
```