CS342 OPERATING SYSTEMS
Fall 2025, Project #1
PROCESSES, IPC, and THREADS

Assigned: Oct 06, 2025.
Due date: Oct 21, 2025; 23:59. Document Version: **1.1**

- You will develop the project in C/Linux. Submitted projects will be tested in **Ubuntu 22.04 Linux 64-bit**.
- You are not allowed to share this project and/or its solution.
- **Objectives/keywords:** Learn and practice with process creation, fork, inter-process communication, shared memory, threads, multi-process applications, multi-threaded applications.

1 PART A (50 pts): Developing a Multi-Process Application

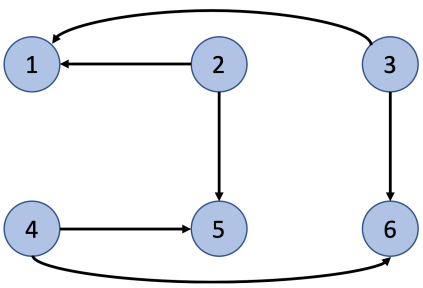
In this project project, you will develop a **multi-process** application that will process a large input file by using multiple child processes [1]. The input file will contain a large number of *ordered* integer pairs (*s*, *d*), each representing a directed edge (arc or link) in a graph. For example, a pair (5, 3) indicates a directed edge from the vertex 5 to the vertex 3. The file may contain millions of such pairs.

For a directed edge (*s*, *d*), we can consider *s* as the *source* vertex (node) and *d* as the *destination* vertex (node) of the edge (link). That means, in a pair of numbers, the first member of the pair is the source vertex number, and the second member is the destination vertex number.

Your task is to write an application that, for each vertex (destination vertex), finds the list of source vertices that point to it, along with the count of those sources. Vertices that do not have any incoming edges should not be considered as destination (should be excluded from the output).

The input file consists of pairs of integers, one pair per line. An example is shown below (together with the represented graph).

```
4 5
3 6
2 5
3 1
2 1
4 6
```



Above, each line specifies one directed edge. For instance, the first line means there is an edge from node 4 to node 5 (pointing from 4 to 5; 4 is source and 5 is destination). Vertex numbers are positive. 0 is not a valid vertex number.

Your C program will create multiple child processes to perform the task, similar in concept to a *MapReduce* system used in distributed computing [2]. The program executable will be named **`findsp`** (find sources using multiple processes) and will accept the following command-line parameters. The program source file will be named as `findsp.c`.

`findsp` `INFILE M R OUT1 OUT2 MIND MAXD SHMSIZE`

`INFILE` parameter is used to specify the name of the input text file. `M` is the number of mapper child processes. `R` is the number of reducer child processes. `OUT1` specifies the name of the final output file that will store the source list for each destination (destinations will be in sorted order; so are the vertices in a source list). `OUT2` specifies the name of the output file that will include the number of sources (count of sources) for each destination (will be in sorted order). The `MIND` and `MAXD` parameters will be used to restrict the processing to only the destinations in the range [`MIND`, `MAXD`] (included). Other destinations will be ignored and will not be included in the output. If `-1` is specified as value of `MIND` or `MAXD`, you will ignore the parameter (parameter will not have any effect). `SHMSIZE` parameter is used to determine the size of the shared memory segment to be created. The size of the shared memory segment will be $2^{\texttt{SHMSIZE}}$. For example, if `SHMSIZE` is specified as 20, then size of the segment will be 2^{20} bytes (1 MB).

Example invocations of the program are shown below:

- `findsp in.txt 5 8 outp1.txt outp2.txt -1 -1 20`
- `findsp in.txt 4 8 outp1.txt outp2.txt 2000 40000 20`

Step 1: Splitting the Input File

The main process will first split the input file into M smaller input files using a simple round-robin method. For example: Write the first pair to file 1, the second pair to file 2, and so on; cycling through all M files. The split files will be named in the following form: **split**-*k*, where *k* is the mapper number (1 <= *k* <= M). For example: split-1, split-2.

Step 2: Mapper Processes

The main process will then create M child processes, each acting as a mapper. Each mapper will read its corresponding split file, process its contents, and produce intermediate output files. A mapper will produce R intermediate files. The processing of a split file will be done while reading the data in from the file. That means, when a pair is read, it will be processed (mapped) and an

output is written to an intermediate file. Then, next pair can be read and processed.

There will be M × R intermediate files, named in the following form: **intermediate**-*i*-*j*, where *i* is mapper number (1 <= *i* <= M) and *j* is reducer number (1 <= *j* <= R).

Mapping Function

For each pair (*s*, *d*) read from an input file, you will first compute the reducer index *k* as *k* = (*d* % R) + 1. Then you will reverse the pair and write it to the intermediate file with that index. That means the pair will be written in reverse order (*d*, *s*) to the intermediate file corresponding to reducer *k*. For example, a pair (3, 7) will be processed as follows in a mapper with id 1. Assume R is 3. Then, *k* will be (7 % 3) + 1, which is 2. Then we will write the line "7, 3" into the intermediate file with name intermediate-1-2.

Each pair of output should be written on a separate line. After a mapper finishes processing its input split file and generating its intermediate files, it will terminate. The parent process will wait for all mappers to terminate before proceeding.

Step 3: Reducer Processes

Once all mappers have finished, the parent will create R reducer processes. It will also create a **shared memory segment** to store tables produced by each reducer process [3].

Each reducer process *k* (1 ≤ *k* ≤ R) will read all intermediate files with names **intermediate**-*i*-*k* (where *i* ranges from 1 to M) and will sort the content (the pairs) according to first destination (*d*), then source (*s*). Then it will go over the sorted list and will group the pairs with respect to the destination. That means for each unique destination vertex *d*, it will build a list of sources (in sorted order) that have an edge to that destination. The reducer will also count the number of sources for a destination *d*.

Then a reducer *k* will write to its output file, for each destination *d*, the destination number and its corresponding list of sources. The name of the output file will be in the form **output**-*k*. A destination and all its source vertices will be written in a single line in the output file. For example, if we have 3, 7, and 1 as the list of source vertices pointing to destination 4, we will write the following line to the output file.

4: 1 3 7

Each reducer will also store (*d*, count) pairs in its assigned part of the shared memory segment. For the above example, since vertex 4 has 3 vertices pointing to it, it will write the pair 4, 3 to the shared memory. Each reducer will use a part of the shared memory segment. You can consider that the

shared memory segment is divided into R parts, each for a different reducer. After finishing processing of its input intermediate files, a reducer will terminate.

Step 4: Parent Process Finalization

When all reducers have terminated, the parent process will merge the reducer output files into one final output file (`OUT1`). The files should be merged so that the output is in sorted order. Additionally, the parent process will combine the data from all reducers' shared memory parts into a single table and write its content to final output file 2 (`OUT2`) in sorted order with respect to *d*.

After generating both output files, the parent process will remove the shared memory segment and will terminate.

Below is an example. Assume M is 3 and R is 2. We see a sample content for various files.

<table>
<tr><th>INFILE</th><th>Split Files</th><th>Intermediate Files (Output of Mappers)</th><th>Output of Reducers</th><th>Final Output Files</th></tr>
<tr><td rowspan="7">in.txt
2 4
3 5
2 5
1 5
3 4
2 6
5 6
4 2
5 4
6 2</td><td rowspan="2">split-1
2 4
1 5
5 6
6 2</td><td>intermediate-1-1
4 2
6 5
2 6</td><td>output-1
2: 4 6
4: 2 3 5
6: 4 5</td><td rowspan="4">outp1.txt
2: 4 6
4: 2 3 5
5: 1 2 3
6: 4 5</td></tr>
<tr><td>intermediate-1-2
5 1</td><td></td></tr>
<tr><td rowspan="2">split-2
3 5
3 4
4 2</td><td>intermediate-2-1
4 3
2 4</td><td></td></tr>
<tr><td>intermediate-2-2
5 3</td><td>output-2
5: 1 2 3</td></tr>
<tr><td rowspan="2">split-3
2 5
2 6
5 4</td><td>intermediate-3-1
6 4
4 5</td><td></td><td rowspan="2">outp2.txt
2: 2
4: 3
6: 2
5: 3</td></tr>
<tr><td>intermediate-3-2
5 2</td><td></td></tr>
</table>

Figure below shows the runtime components (processes) of the application and the flow of data.

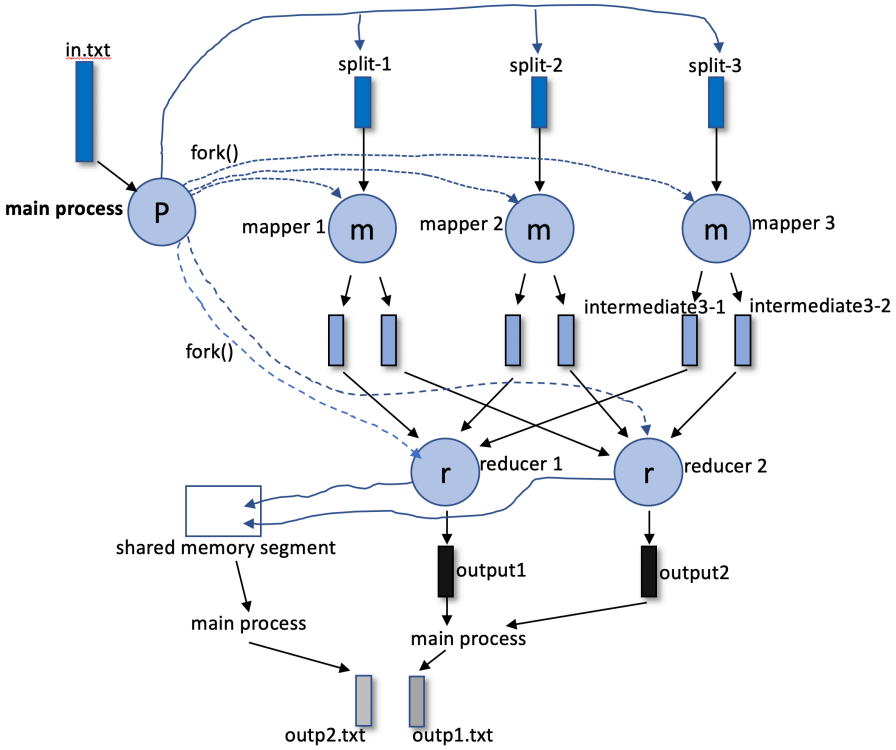


Figure 1. Runtime structure of the application.

Note: Do not delete the intermediate, split, or reducer output files. They will be checked for correctness.

2 PART B (30 pts): Developing a Multi-Threaded Application

Do the same project with **multiple threads** now. Mappers and reducers will be threads. There is no need to use shared memory. Instead you can use a set of arrays or lists. Again intermediate and output data will go into files. The program executable will be named as **`findst`**. The program source file will be named as `findst.c`. You will use POSIX Pthreads API [4].

3 PART C (20 pts): Experiments

You will design and conduct some performance (time) experiments. You will then interpret them. You will put your results, interpretations, and conclusions in a **report.pdf** file. You will include this file as part of your uploaded package. Experiments and the related report will be **20**% of the overall project grade.

You will think and decide what experiments you will perform. For example, you can run your program with different number of child processes (threads) and input file sizes, etc. Measure the time elapsed. Put your results into tables

(or plots). What is the tendency? Try to explain the results and the reasons. Put all your experimental data, analysis and interpretations into the report file.

4 SUBMISSION

Put all your files into a directory named with your Student ID. If the project is done as a group, the IDs of all students will be written, separated by a dash '-'. In a README.txt file, write your name, ID, etc. (if done as a group, all names and IDs will be included). The set of files in the directory will include README.txt, Makefile, and program source file(s). We should be able to compile your program(s) by just typing make. No binary files (executable files or .o files) will be included in your submission. Then **tar** and **gzip** the directory, and submit it to **Moodle**.

For example, a project group with student IDs 21404312 and 214052104 will create a directory named 21404312-214052104 and will put their files there. Then, they will tar the directory (package the directory) as follows:

```
tar cvf 21404312-214052104.tar 21404312-214052104
```

Then they will gzip (compress) the tar file as follows:

```
gzip 21404312-214052104.tar
```

In this way they will obtain a file called 21404312-214052104.tar.gz. Then they will upload this file to **Moodle**. For a project done individually, just the ID of the student will be used as a file or directory name.

5 LATE POLICY FOR THE PROJECT

If the deadline is missed by up to 24 hours, 25 points will be cut from the graded project and that will be the grade of the project. For example, if a late submission is graded and received 80 pts; then the grade of the project will be 80 - 25 = 55. If the deadline is missed between 24 - 48 hours, then 50 pts will be cut. Submission beyond 2 days (48 hours) will not be allowed. Note that even the submission is 1 second later after the deadline, 25 points will be cut. System assigns timestamps to the submissions. We will use those timestamps. Therefore, you are recommended to finish your submission 2-3 hours (at the latest) before the deadline. In fact, it is better if you submit one day before. You can re-submit as many times as you wish by the deadline. Each submission will overwrite the previous one. Please do not send your submission with email. Submission system will be open beyond the deadline up to 48 hours.

6 TIPS AND CLARIFICATIONS

- Start early, work incrementally.
- You will write a single Makefile that will compile both part A and part B. Your Makefile needs just to compile. It will not run a program.
- You may be called for a demo of your project. You may be asked questions. You may need to bring your computer and make a demo to the TA (if you are told to do so).
- Please read the academic integrity policy for projects and homeworks [5]. The page is also accessible from course webpage, from Grading Policy menu item.
- This is a group project. A group can have at most 3 students. If you wish, you can do the project individually. Group members can be from different sections. You can change groups in different projects.
- At the command line, M can be a number between 1 and 20.
- At the command line, R can be a number between 1 and 10.
- A vertex number can be a 32 bit positive integer.
- A vertex can have a link to itself. For example, (3, 3).
- Number of pairs in the input file can be millions.
- A pair may appear multiple times in the input file (pair repeated). But you will consider them as one edge. You will ignore repetitions. Hence repetitions will not appear in the output. You will count the unique sources pointing to a destination.
- You must not include any object or executable files in your submission.
- You will use POSIX shared memory interface [3].

7 REREFENCES

[1] A. Silberschatz, Operating System Concepts, 10th Edition (Global Edition), Wiley, 2021.

[2] Wikipedia, "MapReduce," [Online]. Available: https://en.wikipedia.org/wiki/MapReduce. [Accessed 2025].

[3] "Linux Man Pages," [Online]. Available: https://man7.org/linux/man-pages/man7/shm_overview.7.html. [Accessed 2025].

[4] Wikipedia, "Pthreads," [Online]. Available: https://en.wikipedia.org/wiki/Pthreads.

[5] CS342, "Academic Integrity Policy," 2025. [Online]. Available: https://www.cs.bilkent.edu.tr/~korpe/courses/cs342fall2025/integrity.html.