Operating Systems
Practical Assignment

Eduardo Freitas Fernandes

2025

Contents

1 Overview

In this practical assignment, it's requested to implement a program that simulates the **communication** between a server and a client. The client has a set of **requests** to send to the server, and the server processes the request and sends the result to the client. The context of the requests is about **metadata** relative to documents stored locally on the machine.

The presented implementation uses system calls like `mkfifo()` and `open()` to establish the communication between server and client, `fork()` and `exec()` variants to consult data in the indexed files, and some others.

The program stores this metadata in a **file** (disk) and some in **memory** (cache). It's possible to choose the size of the **cache** and the **eviction policy** to use. I've developed some **Bash scripts** to test and analyse the impact of different cache sizes and types.

2 Architecture

This program is organized in two main parts, the **client** and the **server**. The **client** view is smaller and simpler, given that it only defines the request specified by the user and receives and shows the response given by the server. The **server** component of this program is more interesting and bigger, since it has to manage client-server communication, memory and disk operations. In the next **two subsections** I will make a **detailed** description of those two parts. The program was built upon the following modules:

- `dclient`
- `dserver`
- `client_ops`
- `server_ops`
- `defs`
- `document`
- `free_list`
- `index_table`
- `cache`
- `fifo_cache`
- `rand_cache`
- `lru_cache`
- `utils`

The `utils` module is a general purpose module, used to implement functions needed in other modules but do not fit it's context. The `document` module is used to store the metadata relative to the real document, stored locally.

```
typedef struct document {
    char title[TITLE_SIZE];
    char authors[AUTHORS_SIZE];
    char year[YEAR_SIZE];
```

```
    char path[PATH_SIZE];
} Document;
```

The `defs` module (it's just a header file) defines **macros** essential for all the other modules, like the name of the file to store the metadata, sizes for the metadata and a `struct` relative to a **Request**.

```
typedef struct {
    pid_t client;
    Operation operation;
    char title[TITLE_SIZE];
    char authors[AUTHORS_SIZE];
    char year[YEAR_SIZE];
    char path[PATH_SIZE];
} Request;
```

The `client` field indicates the process identifier of the process (client) that sent the request. The `operation` field indicates the type of request to perform, the available values will be presented ahead. The remaining fields are used to transport the arguments necessary to execute the request.

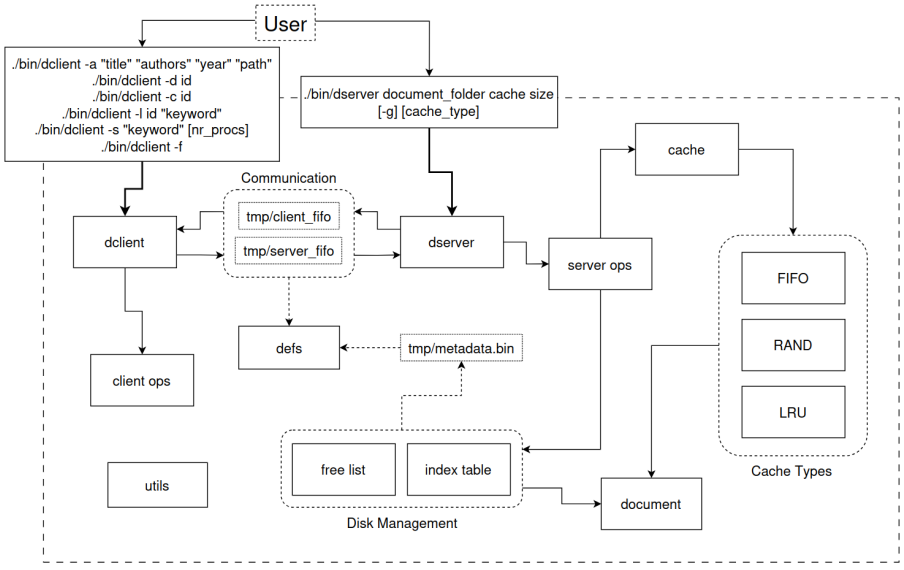

Figure 1: Program Diagram

In Figure 1 we can see the usage of every module and how they interlink with each other. The user starts by turning on the server. The client them sends a request to the server through `SERVER_FIFO`. The indexed metadata is being kept in the `STORAGE_FILE` file and managed by the **Free List** and

Index Table. The `server_ops` module uses these two data structures and the **Cache** to answer the requests. The server then sends a reply through `CLIENT_FIFO` to the user.

2.1 Client

The modules that belong to this program component are:

- `dclient`: main program that the user uses to send requests, receive replies and show it's content.
- `client_ops`: this module ensures the request sent by the user is valid and takes care of the output for the user.

The user calls the `dclient` binary with a flag (e.g. -a, -d, -s) and the necessary arguments for that operation. Through the `client_ops` module, the program validates the given arguments, i.e. checks if the number of arguments is correct. The client creates a pipe with name (FIFO) (e.g. `client_fifo_24`), that will be used to receive the response sent by the server. This FIFO is unique and the only process that will read data from it is the client who created it. After this, the client opens the FIFO created by the server to send a request. This FIFO is shared by all the clients, so every request is sent by this FIFO named `SERVER_FIFO`. The client sends the request using the system call `write()`, and depending on the operation, opens the private FIFO and, using the system call `read()`, receives the response from the server. Once again, the `client_ops` module helps the client and shows the response.

2.2 Server

The modules that belong to this program component are:

- `dserver`: main program that the user uses to start the server, receive requests and do operations.
- `server_ops`: module that contains all the server operations, whether it's request processing, disk or memory management.
- `free_list`: data structure used to store available document identifiers
- `index_table`: data structure used to indicate which documents are valid

The **cache** modules are managed by the server, so they should be included in this listing; however, the cache is a topic I want to discuss separately.
The server is the main piece of the program. It's the first binary to be executed. The server starts by creating the main channel `SERVER_FIFO`

to receive requests from clients. Then creates (or uploads) the necessary data structures to run and enters a loop to receive the request sent by other processes. After receiving a `SHUTDOWN` message, it frees all the data structures and records the necessary information in disk.

2.2.1 Operations

The available request types are:

- `INDEX`: index metadata relative to a file
- `REMOVE`: remove an index of metadata
- `CONSULT`: retrieve the metadata relative to a file
- `COUNT_WORD`: count the number of lines a keyword occurs in a document
- `LIST_WORD`: list the documents that contain a keyword
- `SHUTDOWN`: shut down the server
- `KILL`: tell the server a child process finished its execution

To **index** metadata about a document, the server records a `Document struct` in disk (`STORAGE_FILE`) with the help of the **Free List** and **Index Table**. After this, the metadata is also recorded in **memory** (Cache). To **remove** an indexation, the server marks the document as invalid in the **Index Table**, and reuses the document identifier through the **Free List**. These operations are done by the original server process, so other clients have to wait for these operations to finish.

NOTE: The following operations are consultations, meaning they do not modify the stored metadata. Therefore, clients should not have to wait for others performing the same operations, as specified in the project brief. However, implementing a cache is also required. To ensure the cache content persists between operations, it must be maintained by a single process (since each process has its own instance). To balance both requirements, I decided that the `CONSULT` and `COUNT_WORD` operations will be handled by the server, meaning clients will have to wait in these cases. The `LIST_WORD` operation, on the other hand, is performed by a child process, primarily because it iterates over all documents and is more time-consuming, while the other two operations are very fast as they access only a single document.

The `CONSULT` operation is meant to show the metadata indexed relative to a document. The server uses the **Index Table** to see if the document is indexed, then goes to `STORAGE_FILE` (or cache) and reads the metadata and sends it to the client. The `COUNT_WORD` operation is similar, only with some

additional tasks. The server reads the metadata, grabs the filename and runs `grep -c` to count the number of lines the keyword occurs, and sends the response. `COUNT_WORD` task uses an **anonymous pipe** (created with the system call `pipe()`) to send the count to server.

The `LIST_WORD` operation is done by another process, so that the server can do other tasks and other clients don't have to wait. The default number of processes to execute this task is 1. Through the **Index Table** we can obtain an array with the valid document identifiers. We then divide the identifiers by the number of processes, and each one takes care of a chunk. This operation uses a single **anonymous pipe** to communicate with the child processes, charged of each chunk. If a process finds a document with the keyword, it sends the identifier to the parent process, that stores the ID. An important thing to say is that the processes access the `STORAGE_FILE`, but they don't share the offset (they each use the system call `open()` to avoid this). After creating all the necessary processes, the parent process enters a loop and reads all the document IDs sent in the anonymous pipe. After this is done, it waits (system call `wait()`) for every child process.

NOTE: Each of the operations described above send a **response** to the client, so they either create a child process and he does it, or the current process does it (`COUNT_WORD` and `LIST_WORD` cases).

The `SHUTDOWN` command tells the server to turn off. The client does not receive an answer. The server records the necessary data in disk (Free List and Index Table) and free the allocated space for the data structures.

The `KILL` instruction is used by the child processes of the server. This operation tells the server that a process has finished it's work and it's in zombie mode, waiting for someone to get its return value. This trick allows the server to block in a single place, in the FIFO named `SERVER_FIFO`.

2.2.2 Disk Management

The disk management is done by two data structures, the **Free List** and **Index Table**. A document identifier is an integer, and it's used to determine the position in which the metadata is kept in `STORAGE_FILE`. The `Document struct` has a static size, and the metadata is stored sequentially inside the file, so we only need the identifier to access it.

The **Index Table** is as follows:

```
typedef struct index_table {
    unsigned capacity;
```

```
    unsigned count;
    char *table;
} Index_Table;
```

It's a simple dynamic array of characters. This table is used as an array of **valid bits**. Each `char` holds 8 bits, each one relative to a document. If a bit has the value 0, it means is not valid, does not exist. This approach allows very **fast lookups** and the **space usage** is very low. A lookup is done like so: `(table[id / 8] >> (id % 8)) & 1`.

The **Free List** has the following structure:

```
struct link {
    int id;
    struct link *next;
};

typedef struct free_list {
    struct link *head;
    unsigned size;
} Free_List;
```

This data structure is used to prevent **internal fragmentation**, when removing indexes. The metadata is stored sequentially, so when an index is removed, the program stores the document identifier in a **stack**, and when the next indexation happens, it **reuses** the stored document identifier. This is a simple stack, implemented with a linked list.

Both of these data structures are **persisted** in a file (`CONTROL_FILE`), because when the server is turned off, it's necessary to know which documents are valid or not.

2.3 Cache

The modules used for the cache implementation are:

- `cache`
- `fifo_cache`
- `rand_cache`
- `lru_cache`

The **Cache** module holds a **polymorphic** cache container structure. It stores function pointers to the selected cache functions. This is necessary

because there are multiple cache implementations, this approach allows, if necessary, an expansion of the cache implementations.

The cache is an **independent** layer of this program. If no cache is used, the server accesses disk through the index table and free list, otherwise, the server requests a document from cache, if there is a **hit**, returns the document, if there is a **miss**, it accesses the disk through the same process, and then returns the document.

All the implementations have 4 components in common: an array of `Documents` to store the metadata, an array of identifiers (integers) relatives to the documents, the size of the arrays and a file descriptor (`source`), used to retrieve metadata from disk, in a cache miss.

To optimize the `LIST_WORD` operation, every cache implements a **prefetching** technique. If I search for document 30 and there is a cache miss, the cache goes to disk and brings documents 30 to 37. This saves some time, because in the `-s` task we know that if we search for document `i`, the next search will be `i+1`.

2.3.1 FIFO Eviction Policy

The FIFO cache has the following structure:

```
typedef struct fifo {
    Document *documents;
    int *identifiers;
    int back;
    int size;
    int source;
} FIFO_Cache;
```

The `back` field is used to indicate the next position to insert metadata. Once the cache is full it will start to remove the metadata added at first, following a First-In-First-Out idea.

2.3.2 RAND Eviction Policy

The RAND cache has the following structure:

```
typedef struct rand {
    Document *documents;
    int *identifiers;
    int size;
    int source;
} RAND_Cache;
```

This implementation is the simplest. When adding a set of documents, select a **random position** to place them.

2.3.3 LRU Eviction Policy

The LRU cache has the following structure:

```
typedef struct lru {
    Document *documents;
    int *identifiers;
    char *ref_bits;
    int size;
    int source;
    int back;
} LRU_Cache;
```

This cache implementation tries approximating a **Least Recently Used** eviction policy. The `ref_bits` field is an array of characters that indicates if the position is "valuable", in other words, indicates if the position was used recently. When adding metadata, search for free positions, then positions with reference bits to 0.

3 Usage

To build and run the program, you must follow these instructions in the specified order. First of all, run `make` to build the object files and create the necessary folders. After this operation, there should be two binary files (`dserver` and `dclient`) in the `bin` directory.

3.1 DServer

The server is the first binary to be executed, otherwise the program will not run properly. The server must be initiated with the following structure:

`./bin/dserver document_folder cache_size [-g] [cache_type]`

- `document_folder`: directory in which the documents are stored locally
- `cache_size`: maximum number of entries of metadata to be kept in memory
- `-g`: turn off debugging messages (optional)
- `cache_type`: select the type of eviction policy for the cache. This argument is optional. If no type is given, cache is not used.

After executing this command, it's safe to send requests to the server, through the client.

3.2 DClient

After starting up the server, the user can execute the `dclient` binary and send requests to the program server. There is no need to show the usage of the `dclient` command, the usage is explained in the project brief.

4 Critical Analysis

The last part of this assignment we are asked to analyse the performance of using multiple processes to optimize the list documents operation and to compare the multiple cache eviction policies and its different sizes. In the `scripts` directory you may find scripts to test the performance of the program. The results shown in the next two sub sections were obtained in my machine:

- **Model**:
- **Processor**:
- **Memory**:
- **Operating System**:

4.1 List Documents results

4.2 Cache results