Project #2: Synchronization

Assigned: Oct 24, 2023 Document version: 1.3
Due date: Nov 16, 2023

- This project will be done in groups of two students. You can do it individually as well. You will program in C/Linux. Programs will be tested in Ubuntu 22.04 Linux 64-bit.
- **Objectives**: Practice with thread and process synchronization; POSIX Pthreads mutex and condition variables; low level I/O with binary files; Hash tables; Concurrent data structures; key-value stores, IPC, message queues, lseek(), mmap() calls; Direct access to files; multi-threaded server; multi-threaded client.

---

1. Project Specification (80 pts)

In this project you will develop a **key-value store** (KVS) server and client. The server will be multi-threaded. It will be able to store a large number of data items, i.e., key-value pairs, in the data store (a set of data files). The client program will contact the server and request operations (PUT, DEL, GET) to be performed on the key-value store. The client program will also be multi-threaded, so that it can issue concurrent requests. The server and client programs will communicate with each other by using two POSIX message queues.

The **server program** will be called as `serverk` and will be invoked with the following options.

`serverk -d dcount -f fname -t tcount -s vsize -m mqname`

- The `-d dcount` option specifies the number of data files that will store data items persistently. Let D denote the value of the `dcount` parameter. Then the server will use D files (file 1, ..., file D) to store the data. The data items (key-value pairs) will be partitioned into these D files (D sets) with a simple mapping: a key k and its value will go into file $(k \mod D) + 1$.
- The `-f fname` option is used to generate file names for the data files. The name of a datafile K ($1 \leq K \leq D$) will be `fname`K (e.g., `dataset1`). Data files will be binary files. If the files do not exist, they will be created when the server is started.

- The `-t tcount` option is used to specify the number of **worker threads** that will work on the requests concurrently. Each worker thread will work on another request. Let M denote the value of the `tcount` option. Then M worker threads will be created. M does not have to be equal to D. These M threads will be created in the beginning, when the server is started. Then they will await for work by issuing a wait operation on one or more condition variable(s). When a request arrives to the server, one of the waiting threads will be waken up (by signalling or broadcasting on a condition variable) and will be given the request. The thread will handle the request.
- The `-s vsize` option is used to specify the size of the value of a data item (of a key-value pair). Let L denote the value of the `vsize` parameter. All data item values will of the same size L (that means we are using fixed length key-value pairs).
- The `-m mqname` option is used to specify a string that will be used to name the message queues. There will be two message queues, and they will be named as `mqname1` and `mqname2` (for example: `mq1` and `mq2`). The first message queue will be used for client-to-server communication; the second one for for server-to-client communication.

The server process will also have a **front-end (FE)** thread created (FE in the figure). The FE thread will receive the requests from the first message queue (one by one) and will pass them to the worker threads. It will need to invoke a signal operation on a **condition variable** to wake a worker thread to handle a request. If you wish, if needed, your FE may store messages in a buffer (queue) temporarily (this buffering is optional). After processing a request, a worker thread will send its **reply message** directly to the second message queue (to be received by the client program). When a worker thread has no work to do, it will wait on a condition variable.

A **client program** will send requests to the server. There are three types of requests that can be sent: PUT, DEL (Delete), and GET. The **PUT request** will include a key-value pair and will ask the server to add that pair to the data store. If the key-value pair already exists in the data store, the value will be replaced (overwritten). The server will return a reply message to indicate if the operation was successful or not. The **DEL (delete) request** will include a key and will ask the server to delete the respective key-value pair from the data store. The server will return a reply that will indicate if the operation was successful or not. The **GET request** will include a key and will ask the server to retrieve and return the respective value from the data store. The server will return a reply that will indicate if the operation was successful. If the operation was successful (the related data item is found), the reply will also include the respective value.

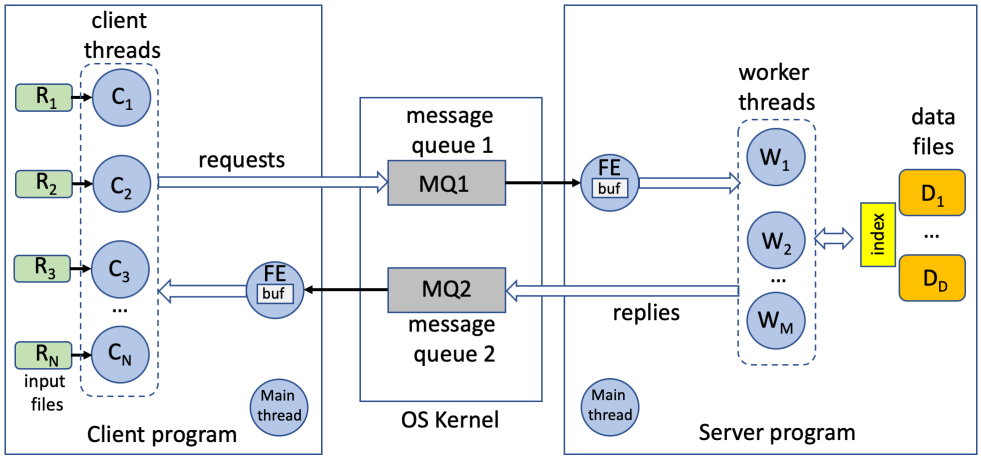

Figure 1: Diagram showing the run-time structure of the Server and Client Programs. Assume we have N client threads created at the client and M worker threads created at the server. The server has D data files.

The client program will be multi-threaded as well. It will be able to use multiple **client threads** to issue requests concurrently. A client thread will read the request descriptions from an input text file. After creating a request message, a client thread will send it directly to the first message queue (MQ1 in the figure), from where the server can receive it. After sending the request to the message queue, the client thread will wait on a **condition variable** until the reply is received by the client program, at which time it will be waken up via a signal (or broadcast) operation.

The client program will be named as `clientk` and will be invoked with the following options.

`clientk -n clicount -f fname -s vsize -m mqname -d dlevel`

- The `-n clicount` option is used to specify the number of client threads that will be created. Let N denote the value of the `clicount` parameter. Then N client threads will be created. Each thread will simulate another client.
- The `-f fname` option is used to specify a name that will be used to generate names for the input files that store request specifications. We call these input files as request files. There will be N input files (file 1, ..., file N). The name of an input file K ($1 \leq K \leq N$) will be `fname`K (for example: `reqfile1`). Each client thread will have its own input request file. Input files will be ascii text files (not binary files).

- The `-s vsize` option is the same with the `-s` option in the server program. Let L denote the value of the `vsize` parameter. L must be the same in both client and server programs.
- The `-m mqname` option is used to specify a message queue name, as in the server program.
- The `-d dlevel` option is used to specify a debug level, which will affect the information messages that will be printed out to the screen by the client program. If `dlevel` is set to 0, no information will be printed out. If it is set to 1, information about each request created and the respective reply received will be printed out, so that the user can see what is going on.

At the client program, we will also have a **front-end** (FE) thread. The FE thread will receive reply messages from the second message queue and will pass them to the client threads. If you wish, if needed, your FE may buffer the messages in a buffer (queue) temporally (this buffering is optional). A reply message will be passed to the client thread that issued the corresponding request. To pass the reply, the FE thread will wake up the respective client thread by issuing a signal (or broadcast) operation on the condition variable where the client thread is waiting.

The **input file** for a client thread will contain a sequence of request descriptions (one request description per line). The client thread will read its request descriptions one line at a time. For each request line read, the client thread will create a request and will send it to the first message queue (MQ1 in the figure). Then it will wait for the respective reply. After reply is received, the client thread will read another request line and will create another request, and so on.

Each line in a request file will specify another request. The format of a request line is as follows:

`request key [value]`

The `request` can be `PUT`, `DEL`, or `GET`. The `key` will be a positive integer. A value will be specified only for a `PUT` request. The specified value can only contain alphanumeric characters (i.e., [a-z], [A-Z], [0-9]), therefore will be a single-word string. The client thread will put this string into a character (byte) array of size L (fixed value length). Before that, the whole array needs to be initialized to character zero, i.e., the NULL character. Then the word is put at the beginning of the array. For example, if the word (i.e., value) is `this`, then the content of the value array can be:

```
't', 'h', 'i', 's', '\0', '\0', ..., '\0'.
```

The number of characters stored in the array is L.

Any **data value** sent from the client to the server (corresponding to a key), or from the server to the client, will be L characters (bytes) long. A stored data value, stored in a data file, should also be L bytes long (even if there is just one alphanumeric character in the value).

When the client sends a GET request to the server (together with a key), the server will return the corresponding value. When the client receives the value back, it can extract the word (alphanumeric character sequence) from it.

An example request file content is given below.

```
PUT 100 This
PUT 101 That
PUT 102 Hello
PUT 200 Bye
PUT 210 The
DEL 102
DEL 1200
DEL 100
GET 250
GET 200
PUT 220 A
PUT 221 B
GET 210
PUT 300 string1
PUT 301 string2
GET 30
PUT 101 Thatis
```

We should be able to run the client program also in **interactive** mode. That means, instead of reading requests from input text files, the program will take them from the user. The client will be run in this interactive mode if the `clicount` argument is set to `0` (zero). In this case, the `-f fname` option can be set to any string (will have no affect). In interactive mode, one client thread will be created and it will interact with the user. It will wait for and take a request description from the user and will create a request. After receiving the respective reply from the server, it will wait for another request description from the user. The format of an interactive request description is the same with the format of a request description that can be read from an input file.

In interactive mode, we can send another type of request, a **DUMP request**. When we send this request, the server will dump (write) the content of all data store files

into a single **output** text (ascii) file. The name of the file will be specified with the DUMP request (for example: `DUMP outfile.txt`). Each line of the output file will contain a key-value pair. First the integer key will appear. Then a SPACE character will follow. Then the value will appear. The value will only contain the alphanumeric characters (printable characters); it will not contain the NULL characters. In this way all data items will be written to the output file. We can use this file to check if our previous operations have modified the data store properly. The output does not have to sorted. We will use the `sort` command to sort the file. An example output can be as follows:

```
120 this
100 B
130 that
200 T
```

In interactive mode, the client program should also support a **QUIT request** and a **QUITSERVER** request. If the user issues the QUIT request at the client (by typing `QUIT` at the client prompt), the client program will terminate (but not the server). If the user issues the QUITSERVER request at the client (by typing `QUITSERVER` at the client prompt), both the server program and client program will terminate (first server, then client). All message queues will be deleted by the server before termination.

Example invocations of the server program are shown below.

```
./serverk -d 3 -f dataset -t 5 -s 64 -m mq
./serverk -d 2 -f dbfile -t 4 -s 1024 -m mqueue
```

Example invocations of the client program are shown below.

```
./clientk -n 10 -f rfile -s 64 -m mq -d 0
./clientk -n 8 -f rfile  -s 1024 -m mqueue -d 1
./clientk -n 0 -f anything -s 1024 -m mqueue -d 1
```

At the server, if you wish, you can **map** a data store file into memory using the `mmap()` system call. Then you can access the file content with memory pointers. Instead, if you wish, you can access the file content by using ordinary `read()` and `write()` system calls and `lseek()` system call. The `lseek()` system call moves the file position pointer to a given file offset. Then, the next read or write operation takes place from/to that position in the file.

When a worker thread receives a request to process, it will first search an **index** structure in memory to find the position (offset) of the data item in the data file that is responsible for the given key. Therefore, you will first read the data store and generate an index, when the server is started (if there is a data store already created). Then, the server will maintain the index as operations are being done on the data store.

The index will be implemented as a set of **hash tables** or a set of **sorted linked lists** (one table or list per data store file). For hash-based indexing, the size of the table can be a reasonable value, like 1024. An index structure (a hash table or a sorted linked list) will keep the file addresses (offsets) of the key-value pairs stored in the respective data store file. Hence it will map a given key to its offset in the file (if key exists in the data store). When a worker thread receives a request, it will first look at the index structure to map the given key into a file offset (i.e., the position of the data item in the data file). The worker thread can then use the file offset to *directly* access the data item from the data file. For a PUT operation, if the specified key is not in the data store file, the data item (key-value pair) will be added (i.e., appended) to the end of the data file.

The **other requirements** of the project are as follows (more can be added to the **Tips and Clarifications** section).

- For simplicity, we will use fixed length keys and values.
- A **key** will be a positive integer (will be stored in the program as a `long int` type). The size of a key in a message or in a data store file will be 8 bytes (a `long int` is 8 bytes).
- A data item **value** will be of fixed length (L). The length will be a multiple of 32 bytes. Then, for example, a key-value pair will take $32 + 8 = 40$ bytes of space (in a message or in a data file). In a real key-value store, keys and values can be of variable length (and can be very long). The minimum value of L can be 32 (2^5) bytes, and the maximum value can be 2^{10} bytes. It has to be a multiple of 2^5 bytes.
- The minimum value of M (the number of worker threads at the server) can be 1, and the maximum value can be 5.
- The minimum value of N (the number of client threads at the client program) can be 1, and the maximum value can be 10.
- The minimum value of D (the number of data files at the server) can be 1, and the maximum value can be 5.
- In Linux, you can increase the number messages that a message queue can buffer. The default value is 10. You can also change the maximum message size supported.
- All messages sent between client and server must be **binary** messages (not in human readable form).

- The format of the messages, exactly what information they will contain, etc., is up to you. You can also use additional message types if you wish.
- You must use **mutex** and **condition variables** (POSIX Pthreads) to synchronize the threads at the server and also at the client.
- The message queue operations (send and receive) are **blocking**.
- Develop your program carefully so that it will not have **deadlocks**.
- The message queues will be created by the server process.
- The server should **clean up** (delete) the message queues before it terminates. In case you need, you can also delete a message queue from the command line (you can learn how from Internet).
- Normally, for a real key-value store, the clients and the server will run in different machines and will communicate with sockets, not with message queues.
- The threads must **not do any busy waiting**. They should use one or more condition variables to sleep if they need to wait. Other threads should signal or broadcast at necessary points to wake up the sleeping threads.
- You should try to have the maximum concurrency possible while processing the requests at the server. For example, if two requests (operations) executed by two worker threads are for keys mapped to different data store files, then these two operations can be executed concurrently (each thread working on another data file). If the keys are mapped to the same data file, however, then the two threads will execute the operations on the data file one at a time to avoid race conditions. You will use **mutex locks** in your program to avoid race conditions at the server and at the client.

2. Experiments (20 pts)

Design and conduct some experiments to observe the effect of using multiple worker threads on the response time of the requests, and also on the throughput (number of requests that can be served per unit time). Also try to observe the effect of value size.

At the end, write a **report** that will explain your experiments and results. Convert the report to a PDF file. You will upload this PDF file as part of your submission.

3. Submission

Put all your files into a directory named with your Student ID. If the project is done as a group, the IDs of both 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 files. We should be able to compile your programs 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`, `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.

4. Tips and Clarifications

1. Start early, work incrementally.
2. You will just write a single client program. It will simulate multiple clients by using multiple threads. In this way, you will not be required to start many client programs which will issue concurrent requests to the server. Starting one client program will be enough.
3. Buffering at an FE is optional (that means you do not have to do it).
4. As indicated earlier, in interactive mode, the client program should also support a **QUIT** request and a **QUITSERVER** request. If QUIT request (by typing `QUIT`) is issued, the client program will terminate (but not the server). If QUITSERVER request is issued (by typing `QUITSERVER`), both server and client will terminate.
5. At the server, two or more worker threads can execute multiple GET operations on the same data file (and its index) at the same time, concurrently, since GET operation is read-only. You should allow multiple GET operations to be done concurrently.
6. If you modify a data file, you need to modify the respective index structure as well. That means you need to keep the data store and its index consistent.

7. If requests are taken from input files (not interactive mode), after all operations are finished, the server should **dump** the data store into an output file called `datastoredump.txt`.