oct 01, 19 23:38 **WriterThread.h** Page 1/1

```
#ifndef _WRITER_THREAD_H_
#define _WRITER_THREAD_H_

#include "ProtectedQueue.h"
#include "Writer.h"
#include <thread>
#include <vector>
#include <cstdbool>

class WriterThread {
    Writer& writer;
    std::thread thread;
    std::vector<ProtectedQueue>& queues;

    private:
        void write_file();

        bool queues_are_open();

        bool queues_are_empty();

    public:
        explicit WriterThread(std::vector<ProtectedQueue>& queues,
                              Writer& writer);

        void run();

        void join();

        ~WriterThread();
};

#endif
```

oct 01, 19 23:38 **WriterThread.cpp** Page 1/1

```
#include "WriterThread.h"
#include "ProtectedQueue.h"
#include "Block.h"
#include "Writer.h"
#include <vector>
#include <fstream>
#include <cstdbool>
#include <iostream>

/*--------------Public--------------*/
WriterThread::WriterThread(std::vector<ProtectedQueue>& queues,
                           Writer& writer):
                           writer(writer),
                           queues(queues) {}

void WriterThread::run() {
    this→thread = std::thread(&WriterThread::write_file, this);
}

void WriterThread::join() {
    this→thread.join();
}

/*--------------Private-------------*/
void WriterThread::write_file() {
    while (this→queues_are_open() ∨ ¬this→queues_are_empty()) {
        for (size_t i = 0; i < queues.size(); i++) {
            Block* block = this→queues[i].pop();
            if (block) {
                block→print_in_file(this→writer);
                delete block;
            }
        }
    }
}

bool WriterThread::queues_are_open() {
    bool open = false;
    for (size_t i = 0; i < this→queues.size(); i++) {
        if (¬this→queues[i].closed()) {
            open = true;
        }
    }
    return open;
}

bool WriterThread::queues_are_empty() {
    bool empty = true;
    for (size_t i = 0; i < this→queues.size(); i++) {
        if (¬this→queues[i].empty()) {
            empty = false;
        }
    }
    return empty;
}

WriterThread::~WriterThread() {}
```

oct 01, 19 23:38 **Writer.h** Page 1/1

```h
#ifndef _WRITER_H_
#define _WRITER_H_

#include <fstream>
#include <cstdint>

class Writer {
    std::ostream* output; //Current output
    std::ofstream file;

    private:
        uint8_t get_byte_to_print(const char* number_by_bit);

    public:
        Writer();

        int set_file(const char* filename);

        void write_bits(uint8_t bits);

        void write_reference(uint32_t reference);

        void write_number(const char* number_by_bit);

        ~Writer();
};

#endif
```

oct 01, 19 23:38 **Writer.cpp** Page 1/1

```cpp
#include "Writer.h"
#include <fstream>
#include <iostream>
#include <cstdint>
#include <arpa/inet.h>

#define ERROR 1
#define SUCCESS 0
#define BINARY +2

/*--------------Public--------------*/
Writer::Writer() {
    this→output = &std::cout; //Default
}

int Writer::set_file(const char* filename) {
    this→file.open(filename, std::ios::binary);
    if (this→file.is_open()) {
        this→output = &this→file;
        return SUCCESS;
    }
    return ERROR;
}

void Writer::write_reference(uint32_t reference) {
    uint32_t ref_be = htonl(reference);
    output→write((char*)&ref_be, sizeof(uint32_t));
}

void Writer::write_bits(uint8_t bits) {
    output→write((char*)&bits, sizeof(uint8_t));
}

void Writer::write_number(const char* number_by_bit) {
    uint8_t number = this→get_byte_to_print(number_by_bit);
    output→write((char*)&number, sizeof(uint8_t));
}

/*--------------Private-------------*/
uint8_t Writer::get_byte_to_print(const char* number_by_bit) {
    return (uint8_t)strtoul(number_by_bit, nullptr, BINARY);
}

Writer::~Writer() {
    if (this→file.is_open()) {
        this→file.close();
    }
}
```

oct 01, 19 23:38 **Reader.h** Page 1/1

```h
#ifndef _READER_H_
#define _READER_H_

#include "BlockBuffer.h"
#include <mutex>
#include <fstream>

class Reader {
    std::istream* input; //Current input
    std::ifstream file;
    std::mutex f_mtx;
    size_t curr_pos;
    int block_len;

    private:
        int set_block(int block_pos);

        int read_block(BlockBuffer& buffer);


    public:
        explicit Reader(int block_len);

        int set_file(const char* filename);

        int set_and_read_block(int block_pos, BlockBuffer& buffer);

        ~Reader();
};

#endif
```

oct 01, 19 23:38 **Reader.cpp** Page 1/1

```cpp
#include "Reader.h"
#include "BlockBuffer.h"
#include <mutex>
#include <fstream>
#include <cstdint>
#include <cstring>

#define ERROR 1
#define SUCCESS 0

/*--------------Public--------------*/
Reader::Reader(int block_len) {
    this→input = &std::cin; //Default
    this→block_len = block_len;
}

int Reader::set_file(const char* filename) {
    this→file.open(filename, std::ios::binary);
    if (this→file.is_open()) {
        this→input = &this→file;
        return SUCCESS;
    }
    return ERROR;
}

int Reader::set_and_read_block(int block_pos, BlockBuffer& buffer) {
    std::unique_lock<std::mutex> lock(this→f_mtx);
    if (this→set_block(block_pos) ≡ ERROR) {
        return 0;
    }
    return this→read_block(buffer);
}

/*--------------Private-------------*/
int Reader::set_block(int block_pos) {
    int pos = block_pos * this→block_len * sizeof(uint32_t);
    if (this→input→seekg(pos, std::ios_base::beg)) {
        return SUCCESS;
    }
    this→input→clear();
    return ERROR;
}

int Reader::read_block(BlockBuffer& buffer) {;
    char number[DW_BYTES];
    memset(number, 0, DW_BYTES * sizeof(char));

    while (¬buffer.is_full() ∧ this→input→read(number, DW_BYTES)) {
        buffer.add_number(number);
    }
    if (this→input→eof() ∨ this→input→fail()) {
        this→input→clear();
    }

    return buffer.numbers_stored();
}

Reader::~Reader() {
    if (this→file.is_open()) {
        this→file.close();
    }
}
```

oct 01, 19 23:38 **ProtectedQueue.h** Page 1/1

```h
#ifndef _QUEUE_H_
#define _QUEUE_H_

#include "Block.h"
#include <queue>
#include <mutex>
#include <cstdbool>
#include <condition_variable>

class ProtectedQueue {
    size_t max_q_len;
    std::queue<Block*> queue;
    std::condition_variable cv;
    std::mutex q_mtx;
    bool q_closed;

    public:
        explicit ProtectedQueue(size_t max_q_len);

        ProtectedQueue(ProtectedQueue∧ p_queue);

        void push(Block* block);

        bool empty();

        void close();

        bool closed();

        Block* pop();

        ~ProtectedQueue();
};

#endif
```

oct 01, 19 23:38 **ProtectedQueue.cpp** Page 1/2

```cpp
#include "ProtectedQueue.h"
#include "Block.h"
#include <cstdbool>
#include <iostream>
#include <mutex>

/*--------------Public--------------*/
ProtectedQueue::ProtectedQueue(size_t max_q_len) {
    this→max_q_len = max_q_len;
    this→q_closed = false;
}

ProtectedQueue::ProtectedQueue(ProtectedQueue∧ p_queue):
    queue(p_queue.queue)
{
    this→max_q_len = p_queue.max_q_len;
    this→q_closed = p_queue.q_closed;

    p_queue.max_q_len = 0;
    p_queue.q_closed = true;
}

void ProtectedQueue::push(Block* block) {
    std::unique_lock<std::mutex> lock(this→q_mtx);
    while (this→queue.size() ≥ this→max_q_len) {
        this→cv.wait(lock);
    }

    this→queue.push(block);
    this→cv.notify_all();
}

Block* ProtectedQueue::pop() {
    std::unique_lock<std::mutex> lock(this→q_mtx);
    while (this→queue.empty() ∧ ¬this→q_closed) {
        this→cv.wait(lock);
    }

    if (this→queue.empty()) {
        //The queue is not receiving more elements
        return nullptr;
    } else {
        Block* block = this→queue.front();
        this→queue.pop();
        this→cv.notify_all();
        return block;
    }
}


bool ProtectedQueue::empty() {
    std::unique_lock<std::mutex> lock(this→q_mtx);
    return this→queue.empty();
}

void ProtectedQueue::close() {
    std::unique_lock<std::mutex> lock(this→q_mtx);
    this→q_closed = true;
    //Notify free pass to take!
    this→cv.notify_all();
}

bool ProtectedQueue::closed() {
    std::unique_lock<std::mutex> lock(this→q_mtx);
    return this→q_closed;
}
```

oct 01, 19 23:38 **ProtectedQueue.cpp** Page 2/2

```
67
68  /*--------------Private--------------*/
69  ProtectedQueue::~ProtectedQueue() {}
```

oct 01, 19 23:38 **main.cpp** Page 1/1

```
1   #include "Compressor.h"
2   #include <iostream>
3   #include <cstring>
4   #include <cstdlib>
5   #include <sstream>
6
7   #define ERROR 1
8   #define SUCCESS 0
9   #define NO_FILE "-"
10
11  int main(int argc, char *argv[]) {
12      if (argc ≠ 6) {
13          std::cerr << "Parametros invalidos" << '\n';
14          return ERROR;
15      }
16
17      char* end_1; char* end_2; char* end_3;
18      int block_len = strtol(argv[1], &end_1, 10);
19      int num_thrds = strtol(argv[2], &end_2, 10);
20      int max_q_len = strtol(argv[3], &end_3, 10);
21
22      if (*end_1 ∨ *end_2 ∨ *end_3) {
23          std::cerr << "Los parametros numericos de entrada son invalidos" <<'\n';
24      }
25
26      Compressor compressor(num_thrds, max_q_len, block_len);
27
28      if (strncmp(NO_FILE, argv[4], 1) ≠ 0) {
29          if (compressor.set_input_file(argv[4]) ≡ ERROR) {
30              std::cerr << "El archivo no se pudo abrir o no existe" << '\n';
31              return ERROR;
32          }
33      }
34      if (strncmp(NO_FILE, argv[5], 1) ≠ 0) {
35          if (compressor.set_output_file(argv[5]) ≡ ERROR) {
36              std::cerr << "El archivo no se pudo crear" << '\n';
37              return ERROR;
38          }
39      }
40
41      compressor.compress();
42      return SUCCESS;
43  }
```

oct 01, 19 23:38 **CompressorThread.h** Page 1/1

```
#ifndef _COMPRESSOR_THREAD_H_
#define _COMPRESSOR_THREAD_H_

#include "ProtectedQueue.h"
#include "BlockBuffer.h"
#include "Reader.h"
#include <cstdint>
#include <thread>

class CompressorThread {
    BlockBuffer buffer;
    int curr_block;
    int off_blocks;

    Reader& reader;

    ProtectedQueue& queue;
    std::thread thread;

    private:
        void compress();

    public:
        CompressorThread(size_t block_len,
                         int start,
                         int off_block,
                         Reader& reader,
                         ProtectedQueue& queue);

        void run();

        void join();

        ~CompressorThread();
};

#endif
```

oct 01, 19 23:38 **CompressorThread.cpp** Page 1/1

```
#include "CompressorThread.h"
#include "ProtectedQueue.h"
#include "BlockBuffer.h"
#include "Block.h"
#include "Reader.h"
#include <iostream>
#include <cstring>
#include <cstdint>
#include <mutex>

#define DW_BYTES 4
#define SUCCESS 0
#define ERROR 1

/*--------------Public--------------*/
CompressorThread::CompressorThread(size_t block_len,
                                   int start,
                                   int off_blocks,
                                   Reader& reader,
                                   ProtectedQueue& queue):
                                   buffer(block_len),
                                   reader(reader),
                                   queue(queue)
{
    this→off_blocks = off_blocks;
    this→curr_block = start;
}

void CompressorThread::run() {
    this→thread = std::thread(&CompressorThread::compress, this);
}

void CompressorThread::join() {
    this→thread.join();
}

/*-------------Private--------------*/
void CompressorThread::compress() {
    while (this→reader.set_and_read_block(this→curr_block,
                                          this→buffer) > 0) {
        Block* block = this→buffer.create_compressed_block();
        this→queue.push(block);
        this→curr_block = this→curr_block + this→off_blocks;
    }
    this→queue.close();
}

CompressorThread::~CompressorThread() {
    //Dont do anything
}
```

oct 01, 19 23:38 **Compressor.h** Page 1/1

```h
#ifndef _COMPRESSOR_H_
#define _COMPRESSOR_H_

#include "CompressorThread.h"
#include "WriterThread.h"
#include "Reader.h"
#include "Writer.h"
#include <vector>
#include <mutex>
#include <iostream>

class Compressor {
    Reader reader;
    Writer writer;

    WriterThread* wtr_thread;
    std::vector<ProtectedQueue> queues;
    std::vector<CompressorThread*> cmp_threads;

    private:
        void init_queues(size_t max_q_len, int num_thrds);

        void init_threads(size_t block_len, int num_thrds);

    public:
        explicit Compressor(int num_thrds, size_t max_q_len, size_t block_len);

        int set_input_file(const char* i_filename);

        int set_output_file(const char* o_filename);

        void compress();

        ~Compressor();
};

#endif
```

oct 01, 19 23:38 **Compressor.cpp** Page 1/1

```cpp
#include "Compressor.h"
#include "Reader.h"
#include "Writer.h"
#include "ProtectedQueue.h"
#include "CompressorThread.h"
#include <iostream>

/*--------------Public--------------*/
Compressor::Compressor(int num_thrds, size_t max_q_len, size_t block_len):
    reader(block_len)
{
    this→init_queues(max_q_len, num_thrds);
    this→init_threads(block_len, num_thrds);
    this→wtr_thread = new WriterThread(this→queues, this→writer);
}

int Compressor::set_input_file(const char* i_filename) {
    return this→reader.set_file(i_filename);
}

int Compressor::set_output_file(const char* o_filename) {
    return this→writer.set_file(o_filename);
}

void Compressor::compress() {
    for (size_t i = 0; i < cmp_threads.size(); i++) {
        this→cmp_threads[i]→run();
    }
    this→wtr_thread→run();
    for (size_t i = 0; i < cmp_threads.size(); i++) {
        this→cmp_threads[i]→join();
    }
    this→wtr_thread→join();
}

/*--------------Private--------------*/
void Compressor::init_threads(size_t block_len, int num_thrds) {
    for (int i = 0; i < num_thrds; i++) {
        CompressorThread* cmp_thread = new CompressorThread(block_len,
                                                            i,
                                                            num_thrds,
                                                            this→reader,
                                                            this→queues[i]);
        this→cmp_threads.push_back(cmp_thread);
    }
}

void Compressor::init_queues(size_t max_q_len, int num_thrds) {
    for (int i = 0; i < num_thrds; i++) {
        this→queues.emplace_back(ProtectedQueue(max_q_len));
    }
}

Compressor::~Compressor() {
    delete this→wtr_thread;
    for (size_t i = 0; i < this→cmp_threads.size(); i++) {
        delete this→cmp_threads[i];
    }
}
```

oct 01, 19 23:38 **Block.h** Page 1/1

```h
#ifndef _BLOCK_H_
#define _BLOCK_H_

#include "Writer.h"
#include <cstdint>
#include <bitset>
#include <vector>
#include <fstream>
#include "Bitset.h"

class Block {
    Bitset bitset;
    uint8_t bits;
    uint32_t ref;

    std::vector<uint32_t> numbers;

    private:
        uint32_t find_max();

        uint32_t find_min();

        void add_padding();

        void numbers_to_bits();

        void subtract_reference();

        uint8_t get_bits(uint32_t number);

    public:
        explicit Block(const std::vector<uint32_t> numbers);

        void print_in_file(Writer& writer);

        void compress();

        ~Block();
};

#endif
```

oct 01, 19 23:38 **Block.cpp** Page 1/2

```cpp
#include "Block.h"
#include "Writer.h"
#include "Bitset.h"
#include <cstdint>
#include <iostream>
#include <cmath>
#include <vector>
#include <bitset>


/*--------------Public--------------*/
Block::Block(const std::vector<uint32_t> numbers) {
    this→numbers.assign(numbers.begin(), numbers.end());
    this→ref = this→bits = 0;
}

void Block::print_in_file(Writer& writer) {
    writer.write_reference(this→ref);
    writer.write_bits(this→bits);
    this→bitset.print_in_file(writer);
}

void Block::compress() {
    this→ref = this→find_min();
    this→subtract_reference();

    uint32_t max = this→find_max();
    this→bits = this→get_bits(max);

    this→bitset.set_size(this→bits * this→numbers.size());

    this→numbers_to_bits();
}

/*--------------Private-------------*/
uint32_t Block::find_min() {
    uint32_t curr_min = this→numbers[0];
    for (size_t i = 1; i < this→numbers.size(); i++) {
        if (curr_min > this→numbers[i]) {
            curr_min = this→numbers[i];
        }
    }
    return curr_min;
}

uint32_t Block::find_max() {
    uint32_t curr_max = this→numbers[0];
    for (size_t i = 1; i < this→numbers.size(); i++) {
        if (curr_max < this→numbers[i]) {
            curr_max = this→numbers[i];
        }
    }
    return curr_max;
}

void Block::subtract_reference() {
    for (size_t i = 0; i < this→numbers.size(); i++) {
        this→numbers[i] = this→numbers[i] − this→ref;
    }
}

uint8_t Block::get_bits(uint32_t number) {
    if (number ≡ 0) {
        return 0;
    }
    return (uint8_t)log2(number) + 1;
```

```
67  }
68
69  void Block::numbers_to_bits() {
70      std::bitset<32> bitset;
71
72      for (size_t i = 0; i < this→numbers.size(); i++) {
73          bitset = this→numbers[i];
74          for (int j = 0; j < this→bits; j++) {
75              this→bitset.push_bit(bitset[this→bits − 1 − j]);
76          }
77      }
78  }
79
80  Block::~Block() {}
```



```
1   #ifndef _BLOCK_BUFFER_H_
2   #define _BLOCK_BUFFER_H_
3
4   #include "Block.h"
5   #include <cstdint>
6   #include <vector>
7   #include <cstdbool>
8   #include <iostream>
9
10  #define DW_BYTES 4
11
12  class BlockBuffer {
13      size_t block_len;
14      std::vector<uint32_t> buffer;
15
16      private:
17          void complete_buffer();
18
19      public:
20          explicit BlockBuffer(size_t block_len);
21
22          bool is_full();
23
24          int numbers_stored();
25
26          Block* create_compressed_block();
27
28          void add_number(const char* str_number);
29
30          ~BlockBuffer();
31  };
32
33  #endif
```

oct 01, 19 23:38 **BlockBuffer.cpp** Page 1/1

```cpp
#include "BlockBuffer.h"
#include "Block.h"
#include <cstdint>
#include <cstdbool>
#include <iostream>
#include <cstring>
#include <arpa/inet.h>

/*--------------Public--------------*/
BlockBuffer::BlockBuffer(size_t block_len) {
    this→block_len = block_len;
}

bool BlockBuffer::is_full() {
    return this→buffer.size() ≥ this→block_len;
}

int BlockBuffer::numbers_stored() {
    return this→buffer.size();
}

void BlockBuffer::add_number(const char* str_number) {
    uint32_t number;
    memcpy(&number, str_number, 4);
    this→buffer.push_back(ntohl(number));
}

Block* BlockBuffer::create_compressed_block() {
    if (this→buffer.size() > 1) {
        //To avoid zero brick case
        this→complete_buffer();
    }
    Block*  block = new Block(buffer);
    this→buffer.clear();
    block→compress();
    return block;
}

/*--------------Private--------------*/
void BlockBuffer::complete_buffer() {
    if (this→buffer.size() < this→block_len) {
        int last_pos = this→buffer.size() − 1;
        while (this→buffer.size() < this→block_len) {
            this→buffer.push_back(this→buffer[last_pos]);
        }
    }
}

BlockBuffer::~BlockBuffer() {}
```

oct 01, 19 23:38 **Bitset.h** Page 1/1

```cpp
#ifndef _BITSET_H_
#define _BITSET_H_

#include "Writer.h"
#include <vector>
#include <cstdint>
#include <fstream>
#include <cstdbool>

class Bitset {
    int size;
    int curr_pos;
    std::vector<bool> bits;

    private:
        void complete_number();

        void calculate_padding();

        uint8_t get_byte_to_print(char* number_by_bit);

    public:
        Bitset();

        void push_bit(bool bit);

        void set_size(int size);

        void print_in_file(Writer& writer);
};

#endif
```

oct 01, 19 23:38 **Bitset.cpp** Page 1/1

```cpp
 #include "Bitset.h"
#include "Writer.h"
#include <cstring>
#include <cstdint>
#include <iostream>
#include <cstdbool>

#define BINARY +2


/*--------------Public--------------*/
Bitset::Bitset() {
    this→size = 0;
    this→curr_pos = 0;
}

void Bitset::set_size(int size) {
    this→size = size;
    this→calculate_padding();
}

void Bitset::push_bit(bool bit) {
    this→bits.push_back(bit);
    this→curr_pos++;
}

void Bitset::print_in_file(Writer& writer) {
    this→complete_number();
    char number_by_bit[8];

    int offset = 0;
    for (int i = 0; i < this→size; i++) {
        number_by_bit[i − offset] = this→bits[i]?'1':'0';
        if ((i+1) % 8 ≡ 0) {
            writer.write_number(number_by_bit);
            offset = offset + 8;
        }
    }
}

/*--------------Private-------------*/
void Bitset::calculate_padding() {
    while (this→size % 8 ≠ 0) {
        this→size++;
    }
}

void Bitset::complete_number() {
    for (int i = this→curr_pos; i < this→size; i++) {
        this→bits[i] = false;
    }
}
```

oct 01, 19 23:38 **Table of Content** Page 1/1

Table of Contents