sep 24, 19 16:48 **Writer.h** Page 1/1

```
#ifndef _WRITER_H_
#define _WRITER_H_

#include <vector>
#include "ProtectedBitBlockQueue.h"
#include "OutFile.h"

class Writer {
private:
  std::vector<ProtectedBitBlockQueue> vectorOfQueues;
  OutFile* outFile;
  unsigned currentQueue;
  int amountOfQueues;
public:
  explicit Writer(OutFile* outFile, int maxElements, int amountOfQueues);

  void write();

  ProtectedBitBlockQueue* getQueueFor(int thread);

  void nextQueue();

  ~Writer();
};


#endif
```

sep 24, 19 16:48 **Writer.cpp** Page 1/1

```
#include "Writer.h"
#define FIRST_QUEUE 0

Writer::Writer(OutFile *outFile, int maxElements, int amountOfQueues) {
  for (int i = 0; i < amountOfQueues; i++){
    vectorOfQueues.push_back(ProtectedBitBlockQueue(maxElements));
  }
  this → outFile = outFile;
  this → amountOfQueues = amountOfQueues;
  currentQueue = FIRST_QUEUE;
}

void Writer::write() {
  while (¬vectorOfQueues.at(currentQueue).donePoping()) {
    BitBlock bitBlock = std::move(vectorOfQueues.at(currentQueue).pop());
    bitBlock.writeTo(outFile);
    nextQueue();
  }
}

ProtectedBitBlockQueue* Writer::getQueueFor(int thread) {
  return &vectorOfQueues.at(thread);
}

void Writer::nextQueue() {
  unsigned newQueue = currentQueue + 1;
  if (newQueue < vectorOfQueues.size()) {
    currentQueue = newQueue;
  } else {
    currentQueue = FIRST_QUEUE;
  }
}

Writer::~Writer() {
}
```

sep 24, 19 16:48 **Thread.h** Page 1/1

```
#ifndef _THREAD_H_
#define _THREAD_H_

#include <thread>

class Thread {
  private:
    std::thread thread;

  public:
    Thread();

    Thread(const Thread&) = delete;

    Thread& operator=(const Thread&) = delete;

    Thread(Thread∧ other);

    Thread& operator=(Thread ∧other);

    virtual void run() = 0;

    void start();

    void join();

    virtual ~Thread();
};


#endif
```

sep 24, 19 16:48 **Thread.cpp** Page 1/1

```
#include "Thread.h"

Thread::Thread() {
}

void Thread::start() {
  thread = std::thread(&Thread::run, this);
}

void Thread::join() {
  thread.join();
}

Thread::Thread(Thread ∧other) {
  this → thread = std::move(other.thread);
}

Thread& Thread::operator=(Thread ∧other) {
  this → thread = std::move(other.thread);
  return *this;
}

Thread::~Thread() {
}
```

sep 24, 19 16:48 **ProtectedInFile.h** Page 1/1

```
#ifndef _IN_FILE_H_
#define  _IN_FILE_H_
#include <istream>
#include "Block.h"
#include <mutex>
#include <fstream>
class ProtectedInFile {
  private:
    std::istream* file;
    std::ifstream fd;
    std::mutex mutex;
    unsigned lastRead;
    bool wasRead;
    int size;

    int readNumberTo(Block* block);

    int isEOF();

  public:
    explicit ProtectedInFile(const char* filename);

    int readNumbsToStartingAt(int amountOfNumb, Block* block, int position);

    ~ProtectedInFile();
};

#endif
```

sep 24, 19 16:48 **ProtectedInFIle.cpp** Page 1/2

```
#include "ProtectedInFile.h"
#include <iostream>
#include <cstdint>
#include <cstring>
#include <netinet/in.h>
#include "Lock.h"
#define END_OF_FILE -1
#define OK 0
#define NUM_SIZE 4
#define CIN '-'

ProtectedInFile::ProtectedInFile(const char* filename) {
  if (*filename == CIN) {
    file = &std::cin;
  } else {
    fd = std::ifstream(filename, std::ios::in|std::ios::binary);
    file = &fd;
  }
  lastRead = 0;
  file -> seekg(0, file -> end);
  size = file -> tellg();
  file -> seekg(0, file -> beg);
}

int ProtectedInFile::readNumbsToStartingAt(int amountOfNumb, Block *block,
    int position) {
  mutex.lock();
  file -> seekg(position, file -> beg);
  int fileState = OK;
  if (file -> tellg() >= size) {
    fileState = END_OF_FILE;
  }
  wasRead = false;
  while (block -> hasSpace() && fileState == OK) {
    fileState = readNumberTo(block);
  }
  while (block -> hasSpace() && wasRead) {
    block -> addNumber(lastRead);
  }
  mutex.unlock();
  return  fileState;
}

int ProtectedInFile::readNumberTo(Block *block) {
  int fileState = isEOF();
  if (fileState == OK) {
    char* num = new char [NUM_SIZE];
    file -> read(num, NUM_SIZE);
    uint32_t number;
    memcpy(&number, num, sizeof(char) * NUM_SIZE);
    delete[] num;
    number = ntohl(number);
    fileState = isEOF();
    if (fileState == OK) {
      lastRead = number;
      block -> addNumber(lastRead);
      wasRead = true;
    }
  }
  return fileState;
}

int ProtectedInFile::isEOF() {
  int returnValue = OK;
  if (file -> eof()) {
    file -> clear();
```

sep 24, 19 16:48 **ProtectedInFIle.cpp** Page 2/2

```
67      returnValue = END_OF_FILE;
68    }
69    return returnValue;
70  }
71
72  ProtectedInFile::~ProtectedInFile(){
73    if (fd.is_open()) {
74      fd.close();
75    }
76  }
```

sep 24, 19 16:48 **ProtectedBitBlockQueue.h** Page 1/1

```
1   #ifndef _PROTECTED_BLOCK_QUEUE_H_
2   #define _PROTECTED_BLOCK_QUEUE_H_
3
4   #include <queue>
5   #include <mutex>
6   #include <condition_variable>
7   #include "BitBlock.h"
8   #include "OutFile.h"
9
10  class ProtectedBitBlockQueue {
11    private:
12      std::queue<BitBlock> queue;
13      std::mutex m;
14      std::condition_variable popCondition;
15      std::condition_variable pushCondition;
16      unsigned maxElements;
17      bool donePushing;
18      bool popAvailable;
19
20    public:
21      explicit ProtectedBitBlockQueue(int maxAmountOfElements);
22
23    ProtectedBitBlockQueue(const ProtectedBitBlockQueue& other);
24
25      void push(BitBlock bitBlock);
26
27      BitBlock pop();
28
29      bool donePoping();
30
31      void done(bool processState);
32
33      ~ProtectedBitBlockQueue();
34  };
35
36
37  #endif
```

sep 24, 19 16:48 **ProtectedBitBlockQueue.cpp** Page 1/1

```cpp
#include "ProtectedBitBlockQueue.h"
#include "Lock.h"

ProtectedBitBlockQueue::ProtectedBitBlockQueue(int maxAmountOfElements) {
  maxElements = maxAmountOfElements;
  donePushing = false;
  popAvailable = false;
}

ProtectedBitBlockQueue::ProtectedBitBlockQueue
  (const ProtectedBitBlockQueue &other):queue(other.queue) {
  this → maxElements = other.maxElements;
  this → donePushing = other.donePushing;
  this → popAvailable = other.popAvailable;
}

void ProtectedBitBlockQueue::push(BitBlock bitBlock) {
  std::unique_lock<std::mutex> lock(m);
  while (queue.size() ≥ maxElements) {
    pushCondition.wait(lock);
  }
  queue.push(std::move(bitBlock));
  done(false);
}

BitBlock ProtectedBitBlockQueue::pop() {
  std::unique_lock<std::mutex> lock(m);
  BitBlock bitBlock(0, 0, 0);
  while (¬popAvailable ∧ ¬donePushing) {
    popCondition.wait(lock);
  }
  if (queue.size() > 0) {
    bitBlock = std::move(queue.front());
    queue.pop();
    popAvailable = false;
  }
  pushCondition.notify_all();
  return std::move(bitBlock);
}

bool ProtectedBitBlockQueue::donePoping() {
  bool answer = false;
  if (donePushing ∧ queue.size() ≡ 0) {
    answer = true;
  }
  return  answer;
}

void ProtectedBitBlockQueue::done(bool processState) {
  popAvailable = true;
  donePushing = processState;
  popCondition.notify_all();
}

ProtectedBitBlockQueue::~ProtectedBitBlockQueue() {
}
```

sep 24, 19 16:48 **OutFile.h** Page 1/1

```cpp
#ifndef _OUT_FILE_H_
#define _OUT_FILE_H_

#include <ostream>
#include <iostream>
#include <fstream>
class OutFile {
  private:
    std::ostream* outFile;
    std::ofstream fd;

  public:
    explicit OutFile(const char* filename);

    void write(char* buf, int bytesToWrite);

    ~OutFile();
};


#endif
```

sep 24, 19 16:48 **OutFile.cpp** Page 1/1

```cpp
#include "OutFile.h"
#include <fstream>
#define COUT '-'

OutFile::OutFile(const char* filename) {
  if (*filename ≡ COUT) {
    outFile = &std::cout;
  } else {
    fd = std::ofstream(filename, std::ios::binary);
    outFile = &fd;
    }
}

void OutFile::write(char *buf, int bytesToWrite) {
  outFile → write(buf, bytesToWrite);
}

OutFile::~OutFile() {
  if (fd.is_open()) {
    fd.close();
  }
}
```

sep 24, 19 16:48 **main.cpp** Page 1/1

```cpp
#include "ProtectedInFile.h"
#include "FileCompressor.h"
#include "OutFile.h"
#include "Writer.h"
#include <vector>
int main(int argc, char const *argv[]) {
  ProtectedInFile file(argv[4]);
  OutFile outFile(argv[5]);
  int numbsPerBlock = atoi(argv[1]);
  int amountOfThreads = atoi(argv[2]);
  int elementsPerQueue = atoi(argv[3]);
  Writer writer(&outFile, elementsPerQueue, amountOfThreads);
  std::vector<FileCompressor*> threads;
  for (int i = 0; i < amountOfThreads; i++) {
    threads.push_back(new FileCompressor(&file, writer.getQueueFor(i),
        numbsPerBlock, amountOfThreads, i));
    threads[i] → start();
  }
  writer.write();
  for (int i = 0; i < (*argv[2] - '0'); i++) {
    threads[i] → join();
    delete threads[i];
  }
  return 0;
}
```

sep 24, 19 16:48 **Lock.h** Page 1/1

```
#ifndef TP2_LOCK_H
#define TP2_LOCK_H

#include <mutex>

class Lock {
  private:
    std::mutex& mutex;
  public:
    explicit Lock(std::mutex& aMutex);

    ~Lock();
};
#endif
```

sep 24, 19 16:48 **Lock.cpp** Page 1/1

```
#include "Lock.h"

Lock::Lock(std::mutex &aMutex) :mutex(aMutex) {
  mutex.lock();
}

Lock::~Lock() {
  mutex.unlock();
}
```

sep 24, 19 16:48 **FileCompressor.h** Page 1/1

```
#ifndef _FILE_COMPRESSOR_H_
#define  _FILE_COMPRESSOR_H_
#include "ProtectedInFile.h"
#include "ProtectedBitBlockQueue.h"
#include "Block.h"
#include "Thread.h"
class FileCompressor : public Thread {
  private:
    ProtectedInFile* inFile;
    ProtectedBitBlockQueue* queue;
    Block block;
    int numbsPerBlock;
    int numbOfThreads;
    int myNumb;

  public:
    explicit FileCompressor(ProtectedInFile* in, ProtectedBitBlockQueue* queue,
        int numbsPerBlock, int numbOfThreads, int myNumb);

    void run();

    ~FileCompressor();
};

#endif
```

sep 24, 19 16:48 **FileCompressor.cpp** Page 1/1

```
#include "FileCompressor.h"
#define END_OF_FILE -1
#define OK 0
#define NUMB_SIZE 4

FileCompressor::FileCompressor(ProtectedInFile* in, ProtectedBitBlockQueue*
    queue, int numbsPerBlock, int numbOfThreads, int myNumb)
    :block(numbsPerBlock) {
  inFile = in;
  this → queue = queue;
  this → numbsPerBlock = numbsPerBlock;
  this → numbOfThreads = numbOfThreads;
  this → myNumb = myNumb;
}

void FileCompressor::run() {
  int fileState = OK;
  int pos = (numbsPerBlock * NUMB_SIZE * myNumb);
  while (fileState ≡ OK) {
    fileState = inFile → readNumbsToStartingAt(numbsPerBlock, &block, pos);
    pos += (numbsPerBlock * NUMB_SIZE * numbOfThreads);
    if (¬(block.hasSpace())) {
      block.compressTo(queue);
      block.reset();
    }
  }
  queue → done(true);
}

FileCompressor::~FileCompressor() {
}
```

sep 24, 19 16:48 **Block.h** Page 1/1

```
#ifndef _BLOCK_H_
#define  _BLOCK_H_
#include <vector>
#include "BitBlock.h"
#include "ProtectedBitBlockQueue.h"
#include "OutFile.h"

class Block {
  private:
    unsigned minNumb;
    unsigned maxNumb;
    std::vector<unsigned>::iterator iterator;
    std::vector<unsigned> numbs;
    BitBlock* bits;

    void updateMax(unsigned numberAdded);

    void updateMin(unsigned numberAdded);


  public:
    explicit Block(int amountOfNumbs);

    void addNumber(unsigned numbToAdd);

    void reset();

    bool hasSpace();

    void compressTo(ProtectedBitBlockQueue* queue);

    ~Block();
};
#endif
```

sep 24, 19 16:48 **Block.cpp** Page 1/1

```
#include "Block.h"
#include "BitBlock.h"
#include "ProtectedBitBlockQueue.h"
#define MIN_VALUE 0
#define MAX_VALUE 0xffffffff

Block::Block(int amountOfNumbs) {
  numbs.resize(amountOfNumbs);
  reset();
}

void Block::addNumber(unsigned numbToAdd) {
  if (iterator < numbs.end()){
    updateMin(numbToAdd);
    updateMax(numbToAdd);
    *iterator = numbToAdd;
    iterator ++;
  }
}

void Block::updateMax(unsigned numberAdded) {
  if (maxNumb < numberAdded) {
    maxNumb = numberAdded;
  }
}

void Block::updateMin(unsigned numberAdded) {
  if (minNumb > numberAdded) {
    minNumb = numberAdded;
  }
}
void Block::reset() {
  iterator = numbs.begin();
  minNumb = MAX_VALUE;
  maxNumb = MIN_VALUE;
}

bool Block::hasSpace() {
  bool answer = false;
  if (iterator < numbs.end()) {
    answer = true;
  }
  return answer;
}

void Block::compressTo(ProtectedBitBlockQueue *queue) {
  BitBlock bitBlock(minNumb, (maxNumb - minNumb), numbs.size());
  for (iterator = numbs.begin(); iterator < numbs.end(); iterator ++) {
    bitBlock.addNumb(*iterator - minNumb);
  }
  bitBlock.addPadding();
  queue -> push(std::move(bitBlock));
}

Block::~Block() {
}
```

sep 24, 19 16:48 **BitBlock.h** Page 1/1

```
#ifndef _BIT_BLOCK_H_
#define _BIT_BLOCK_H_
#include "OutFile.h"
#include <vector>
class BitBlock {
  private:
    unsigned reference;
    std::vector<char>::iterator iterator;
    std::vector<char> bytes;
    char aux;
    int inBit;
    bool validBlock;
    unsigned bitsPerNumb;

    void nextBit();
    unsigned calculateBitsPerNumb(unsigned maxNumb);
    int calculateBytesNeeded(int amountOfNumbs);


  public:
    explicit BitBlock(unsigned aReference, unsigned maxNumb, int amountOfNumbs);

    BitBlock(const BitBlock& other) = default;

    BitBlock(BitBlock∧ other);

    BitBlock&operator=(BitBlock∧ other);

    void addNumb(unsigned numbToAdd);

    void writeTo(OutFile *outFile);

    void addPadding();

    ~BitBlock();
};
#endif
```

sep 24, 19 16:48 **BitBlock.cpp** Page 1/2

```
#include "BitBlock.h"
#include <iostream>
#include <bitset>
#include "OutFile.h"
#include <netinet/in.h>
#include <bitset>
#define MAX_BIT_QUANTITY 32
#define FIRST_BIT 7
#define LAST_BIT 0
#define REFERENCE_SIZE 4

BitBlock::BitBlock(unsigned aReference, unsigned maxNumb, int amountOfNumbs) {
  bitsPerNumb = calculateBitsPerNumb(maxNumb);
  bytes.resize(calculateBytesNeeded(amountOfNumbs));
  iterator = bytes.begin();
  inBit = FIRST_BIT;
  reference = aReference;
  validBlock = true;
  if (amountOfNumbs ≡ 0) {
    validBlock = false;
  }
}

BitBlock::BitBlock(BitBlock ∧other): iterator(std::move(other.iterator)),
    bytes(std::move(other.bytes)) {
  this → reference = other.reference;
  this → aux = other.aux;
  this → bitsPerNumb = other.bitsPerNumb;
  this → inBit = other.inBit;
  this → validBlock = other.validBlock;
}

BitBlock& BitBlock::operator=(BitBlock ∧other) {
  if (this ≡ &other) {
    return  *this;
  }
  this → iterator = std::move(other.iterator);
  this → bytes = std::move(other.bytes);
  this → reference = other.reference;
  this → aux = other.aux;
  this → bitsPerNumb = other.bitsPerNumb;
  this → inBit = other.inBit;
  this → validBlock = other.validBlock;
  return *this;
}

int BitBlock::calculateBytesNeeded(int amountOfNumbs) {
  int amountOfBits = amountOfNumbs * bitsPerNumb;
  int bytesNeeded = amountOfBits / 8;
  if ((amountOfBits % 8) ≠ 0) {
    bytesNeeded ++;
  }
  return bytesNeeded;
}

unsigned int BitBlock::calculateBitsPerNumb(unsigned maxNumb) {
  std::bitset<MAX_BIT_QUANTITY>  bits(maxNumb);
  int inBit = 0;
  unsigned index = MAX_BIT_QUANTITY;
  if (maxNumb ≠ 0) {
    while (inBit ≡ 0 ∧ index > 0) {
      index −−;
      inBit = bits[index];
    }
    index ++;
  } else {
```

```
67      index = 0;
68    }
69    return index;
70  }
71
72  void BitBlock::addNumb(unsigned numbToAdd) {
73    std::bitset<MAX_BIT_QUANTITY> source(numbToAdd);
74    int amountOfBits = bitsPerNumb;
75    while (0 < amountOfBits) {
76      aux = (aux &~ (1UL << inBit)) | (source[amountOfBits − 1] << inBit);
77      nextBit();
78      amountOfBits −−;
79    }
80  }
81
82  void BitBlock::writeTo(OutFile* outFile) {
83    if (validBlock) {
84      reference = ntohl(reference);
85      outFile → write((char*) &reference, REFERENCE_SIZE);
86      outFile → write((char*) &bitsPerNumb, 1);
87      for (iterator = bytes.begin(); iterator < bytes.end(); iterator ++) {
88        outFile → write(&(*iterator), 1);
89      }
90    }
91  }
92
93  void BitBlock::addPadding() {
94    if (iterator ≠ bytes.end() ∧ inBit ≠ FIRST_BIT) {
95      while  (inBit ≥ 0) {
96        aux = (aux &~ (1UL << inBit)) | (0 << inBit);
97        inBit −−;
98      }
99      *iterator = aux;
100   }
101 }
102
103 void BitBlock::nextBit() {
104   if (inBit ≡ LAST_BIT) {
105     inBit = FIRST_BIT;
106     if (iterator ≠  bytes.end()){
107       *iterator = aux;
108       iterator ++;
109     }
110   } else {
111     inBit −−;
112   }
113 }
114
115 BitBlock::~BitBlock() {
116 }
```


Table of Contents