Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

**Курсовой проект**

**«Быки и коровы на memory** **map»**

Группа: М80-206Б-22

Студент: Голубев Т. Д.

Преподаватель: Миронов Е.С.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 08.01.24

Москва, 2024

**Постановка задачи**

Необходимо спроектировать и реализовать программный прототип в соответствии с выбранным вариантом. Произвести анализ и сделать вывод на основании данных, полученных при работе программного прототипа.

Проектирование консольной клиент-серверной игры

Создать собственную игру более, чем для одного пользователя. Игра может быть устроена по принципу: клиент-клиент, сервер-клиент. Необходимо написать 2 программы: сервер и клиент. Сначала запускается сервер, а далее клиенты соединяются с сервером. Сервер координирует клиентов между собой. При запуске клиента игрок может выбрать одно из следующих действий (возможно больше, если предусмотрено вариантом):

1. Создать игру, введя ее имя,
2. Присоединиться к одной из существующих игр по имени игры. «Быки и коровы» (угадывать необходимо слова). Общение между сервером и клиентом необходимо организовать при помощи memory map. При создании каждой игры необходимо указывать количество игроков, которые будут участвовать. То есть угадывать могут несколько игроков. Если кто-то из игроков вышел из игры, то игра должна быть продолжена.

**Общий метод и алгоритм решения**

Сервер принимает запросы от клиентов. Среди запросов может быть создание игры. Это запускает новый процесс «Сервер игры». Клиенты подключаются к этому серверу, общаются с ним. Игра происходит через запросы к этому серверу.

Процессы общаются с помощью memory map. Существует структура «Сообщение», которая состоит из полей «Тип» и «Данные». Общение клиента с сервером осуществляется с помощью этих сообщений, которые размещаются в memory map.

Во время запуска игры случайно выбирается слово из базы данных. Собираются ответы пользователей и проверяется их правильность. Пользователям сообщается количество «быков» (угаданных букв с точностью до позиции) и «коров» (угаданных букв с неверной позицией). Игра завершается, когда хотя бы один из игроков угадал слово.

**mmap.h**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166 | #pragma once  #include <unistd.h>  #include <iostream>  #include <sys/mman.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <exception>  #include <string>  **namespace** bc {  **enum** ModeFlags {  read = PROT\_READ,  write = PROT\_WRITE,  exec = PROT\_EXEC,  none = PROT\_NONE  };  **template** <**class** **T**>  **class** **MemoryMap** {  **public:**  MemoryMap() = **delete**;  MemoryMap(**const** std::string& s, **size\_t** size, **int** mode);  **void** **delete\_shm\_file**();  ~MemoryMap();  MemoryMap(**const** MemoryMap<T>& other);  MemoryMap(MemoryMap<T>&& other) noexcept;  MemoryMap<T>& **operator**=(**const** MemoryMap<T>& other);  MemoryMap<T>& **operator**=(MemoryMap<T>&& other) noexcept;  T\* data() **const** noexcept;  **size\_t** size() **const** noexcept;  T& **operator**[](**int** idx);  **const** T& **operator**[](**int** idx) **const**;  **const** std::string& name() **const**;  **private:**  T\* \_data;  **int** \_fd;  std::string \_name;  **size\_t** \_size;  **int** \_mode;  };  **template** <**class** **T**>  MemoryMap<T>::MemoryMap(**const** std::string& name, **size\_t** size, **int** mode) : \_name{name}, \_size{size}, \_mode{mode} {  \_fd = shm\_open(name.c\_str(), O\_CREAT | O\_RDWR, S\_IREAD | S\_IWRITE);  **if** (ftruncate(\_fd, **sizeof**(T) \* size) != **0**) {  **throw** std::runtime\_error("ftruncate error");  }  **if** (\_fd == -**1**) {  **throw** std::runtime\_error("shm\_open error");  }  \_data = (T\*) mmap(NULL, size, mode, MAP\_SHARED, \_fd, **0**);  **if** (\_data == MAP\_FAILED) {  **throw** std::runtime\_error("mmap error");  }  }  **template** <**class** **T**>  **void** MemoryMap<T>::delete\_shm\_file() {  **int** error\_code = shm\_unlink(\_name.c\_str());  **if** (error\_code == -**1**) {  **throw** std::runtime\_error("shm\_unlink");  }  }  **template** <**class** **T**>  MemoryMap<T>::~MemoryMap() {  munmap(\_data, \_size);  }  **template** <**class** **T**>  MemoryMap<T>::MemoryMap(**const** MemoryMap<T>& other) :  MemoryMap<T>(other.\_name, other.\_size, other.\_mode) {}  **template** <**class** **T**>  MemoryMap<T>::MemoryMap(MemoryMap<T>&& other) noexcept :  \_data{other.\_data},  \_fd{std::move(other.\_fd)},  \_name{std::move(other.\_name)},  \_size{std::move(other.\_size)},  \_mode{std::move(other.\_mode)} {}  **template** <**class** **T**>  MemoryMap<T>& MemoryMap<T>::**operator**=(**const** MemoryMap<T>& other) {  \_name = other.\_name;  \_size = other.\_size;  \_mode = other.\_mode;  \_fd = shm\_open(\_name.c\_str(), O\_CREAT | O\_RDWR, S\_IREAD | S\_IWRITE);  **if** (ftruncate(\_fd, **sizeof**(T) \* \_size) != **0**) {  **throw** std::runtime\_error("ftruncate error");  }  **if** (\_fd == -**1**) {  **throw** std::runtime\_error("shm\_open error");  }  \_data = (T\*) mmap(NULL, \_size, other.\_mode, MAP\_SHARED, \_fd, **0**);  **if** (\_data == MAP\_FAILED) {  **throw** std::runtime\_error("mmap error");  }  **return** \***this**;  }  **template** <**class** **T**>  MemoryMap<T>& MemoryMap<T>::**operator**=(MemoryMap<T>&& other) noexcept {  \_data = other.\_data;  \_fd = std::move(other.\_fd);  \_name = std::move(other.\_name);  \_size = std::move(other.\_size);  \_mode = std::move(other.\_mode);  **return** \***this**;  }  **template** <**class** **T**>  T\* MemoryMap<T>::data() **const** noexcept {  **return** \_data;  }  **template** <**class** **T**>  **size\_t** MemoryMap<T>::size() **const** noexcept {  **return** \_size;  }  **template** <**class** **T**>  T& MemoryMap<T>::**operator**[](**int** idx) {  **if** (idx > \_size - **1**) {  **throw** std::range\_error("out of range");  }  **return** \_data[idx];  }  **template** <**class** **T**>  **const** T& MemoryMap<T>::**operator**[](**int** idx) **const** {  **if** (idx > \_size - **1**) {  **throw** std::range\_error("out of range");  }  **return** \_data[idx];  }  **template** <**class** **T**>  **const** std::string& MemoryMap<T>::name() **const** {  **return** \_name;  }  **template** <**class** **T**>  **void** str\_to\_mmap(**const** std::string& str, MemoryMap<T>& mmap, **int** start\_idx) {  **if** ((mmap.size() - start\_idx) < str.size()) **throw** std::logic\_error("string is too long");  **int** j = start\_idx;  **for** (**int** i = **0**; i < str.size(); ++i) {  mmap[j] = str[i];  ++j;  }  mmap[start\_idx + str.size()] = '\0';  }  **template** <**class** **T**>  std::string mmap\_to\_str(**const** MemoryMap<T>& mmap, **int** start\_idx) {  std::string str(mmap.size() - start\_idx, ' ');  **int** i = start\_idx;  **int** j = **0**;  **while** (mmap[i] != '\0') {  str[j] = mmap[i];  ++i;  ++j;  }  **return** str;  }  } // bulls & cows |

**mutex.h**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | #pragma once  #include <pthread.h>  #include <string>  #include "mmap.h"  /\* This is my process-shared mutex \*/  **namespace** bc {  **enum** MutexFlag {  create,  connect  };  **class** **Mutex** {  **private:**  std::string \_name;  MemoryMap<**pthread\_mutex\_t**> \_mtx;  **public:**  Mutex(**const** std::string& name, MutexFlag flag);  Mutex(**const** Mutex& other);  Mutex(Mutex&& other) noexcept;  ~Mutex() noexcept;  Mutex& **operator**=(**const** Mutex& other);  Mutex& **operator**=(Mutex&& other) noexcept;  **void** **lock**();  **void** **unlock**();  **const** std::string& name() **const**;  **void** **delete\_for\_all**(); // it delete shmfile, should be called by server  };  } // bulls & cows |

**mutex.cpp**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | #include "mutex.h"  **using** **namespace** bc;  Mutex::Mutex(**const** std::string& name, MutexFlag flag) : \_name{name}, \_mtx(name, **1**, ModeFlags::write | ModeFlags::read) {  **if** (flag == MutexFlag::create) {  **pthread\_mutexattr\_t** attr;  pthread\_mutexattr\_init(&attr);  pthread\_mutexattr\_setpshared(&attr, PTHREAD\_PROCESS\_SHARED);  pthread\_mutex\_init(&\_mtx[**0**], &attr);  }  }  Mutex::Mutex(**const** Mutex& other) : \_name{other.\_name}, \_mtx{other.\_mtx} {}  Mutex::Mutex(Mutex&& other) noexcept : \_name{std::move(other.\_name)}, \_mtx{std::move(other.\_mtx)} {}  Mutex::~Mutex() noexcept {  // there is no memory leak because of destructor of bc::MemoryMap  }  Mutex& Mutex::**operator**=(**const** Mutex& other) {  \_name = other.\_name;  \_mtx = other.\_mtx;  **return** \***this**;  }  Mutex& Mutex::**operator**=(Mutex&& other) noexcept {  \_name = std::move(other.\_name);  \_mtx = std::move(other.\_mtx);  **return** \***this**;  }  **void** Mutex::lock() {  **int** res = pthread\_mutex\_lock(&\_mtx[**0**]);  **if** (res != **0**) **throw** std::runtime\_error(std::to\_string(res));  }  **void** Mutex::unlock() {  **int** res = pthread\_mutex\_unlock(&\_mtx[**0**]);  **if** (res != **0**) **throw** std::runtime\_error(std::to\_string(res));  }  **const** std::string& Mutex::name() **const** {  **return** \_name;  }  **void** Mutex::delete\_for\_all() {  \_mtx.delete\_shm\_file();  } |

**server.h**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | #pragma once  #include <string>  #include <unistd.h>  #include "mmap.h"  #include "mutex.h"  #include "message.h"  **namespace** bc {  **class** **Server** {  **private:**  **int** \_id;  MemoryMap<**char**> \_mmap;  Mutex \_mtx;  **static** **const** **int** \_mmap\_size = **10000**;  **public:**  Server();  ~Server();  **void** **create\_game**(**const** std::string& game\_name, **int** players);  **int** id() **const**;  **void** **send\_message**(**const** Message& msg);  Message **get\_message**();  **void** **clear\_mmap**();  };  } // bulls & cows |

**server.cpp**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | #include "server.h"  **using** **namespace** bc;  Server::Server() : \_id{getpid()},  \_mmap("bc\_" + std::to\_string(\_id) + "\_mmap", \_mmap\_size, ModeFlags::read | ModeFlags::write),  \_mtx("bc\_" + std::to\_string(\_id) + "\_mutex", MutexFlag::create) {  clear\_mmap();  }  Server::~Server() {  \_mtx.delete\_for\_all();  \_mmap.delete\_shm\_file();  }  **void** Server::create\_game(**const** std::string& game\_name, **int** players) {  **int** pid = fork();  **if** (pid == -**1**) **throw** std::runtime\_error("fork error");  **if** (pid == **0**) {  **int** er = execl("../build/kp\_game", "../build/kp\_game", game\_name.c\_str(), std::to\_string(players).c\_str(), NULL);  **if** (er == -**1**) {  std::cout << "create er: " << errno << std::endl;  exit(**1**);  }  }  }  **int** Server::id() **const** {  **return** \_id;  }  **void** Server::send\_message(**const** Message& msg) {  \_mtx.lock();  \_mmap[**0**] = msg.type;  str\_to\_mmap(msg.data, \_mmap, **1**);  \_mtx.unlock();  }  Message Server::get\_message() {  Message msg;  \_mtx.lock();  msg.type = (MessageType) \_mmap[**0**];  **if** (msg.type == MessageType::wait) {  \_mtx.unlock();  **return** msg;  }  msg.data = mmap\_to\_str(\_mmap, **1**);  \_mtx.unlock();  **return** msg;  }  **void** Server::clear\_mmap() {  \_mtx.lock();  \_mmap[**0**] = MessageType::wait;  \_mmap[**1**] = '\0';  \_mtx.unlock();  } |

**game.h**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | #pragma once  #include <set>  #include <fstream>  #include <random>  #include "mmap.h"  #include "mutex.h"  #include "message.h"  **namespace** bc {  **class** **Game** {  **private:**  MemoryMap<**char**> \_mmap;  Mutex \_mtx;  std::string \_name;  **int** \_players;  std::string \_word;  **int** \_words; // count of  std::set<**char**> \_letters;  std::ifstream \_db;  **int** \_winner\_id;  **static** **const** **int** \_mmap\_size = **10000**;  **public:**  Game(**const** std::string& name, **int** players, **const** std::string& db\_name);  Game(**const** Game& other) = **delete**;  ~Game() noexcept;  **void** **send\_message**(**const** Message& msg);  Message **get\_message**();  **void** **clear\_mmap**();  **const** std::string& word() **const**;  **int** players() **const**;  **int**& players();  **const** std::string& name() **const**;  MemoryMap<**char**>& mmap();  Mutex& mtx();  **int** winner\_id() **const**;  std::string check\_word(**const** std::string& word, **int** player\_id);  **private:**  std::string \_generate\_word();  };  } // bulls & cows |

**game.cpp**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108 | #include "game.h"  **using** **namespace** bc;  Game::Game(**const** std::string& name, **int** players, **const** std::string& db\_name) : \_name{name},  \_mmap{"bcgame\_" + name + "\_mmap", \_mmap\_size, ModeFlags::read | ModeFlags::write},  \_mtx{"bcgame\_" + name + "\_mutex", MutexFlag::create},  \_db{db\_name, std::ios::binary | std::ios::in},  \_players{players},  \_winner\_id{**0**} {  **if** (!\_db.good()) **throw** std::runtime\_error("Can't open database file");  \_db.read(**reinterpret\_cast**<**char**\*>(&\_words), **sizeof**(**int**));  \_word = \_generate\_word();  **for** (**char** letter : \_word) {  \_letters.insert(letter);  }  }  Game::~Game() noexcept {  \_db.close();  \_mmap.delete\_shm\_file();  \_mtx.delete\_for\_all();  }  **void** Game::send\_message(**const** Message& msg) {  \_mtx.lock();  \_mmap[**0**] = msg.type;  str\_to\_mmap(msg.data, \_mmap, **1**);  \_mtx.unlock();  }  Message Game::get\_message() {  Message msg;  \_mtx.lock();  msg.type = (MessageType) \_mmap[**0**];  msg.data = mmap\_to\_str(\_mmap, **1**);  \_mtx.unlock();  **return** msg;  }  std::string Game::\_generate\_word() {  std::random\_device dev;  std::mt19937 rng(dev());  std::uniform\_int\_distribution<std::mt19937::result\_type> dist(**0**, \_words - **1**);  \_db.seekg(**sizeof**(**int**) + **sizeof**(**char**) \* **5** \* dist(rng), std::ios::beg);  **char** raw\_word[**5**];  \_db.read(raw\_word, **sizeof**(**char**) \* **5**);  std::string word(raw\_word);  **return** word;  }  **const** std::string& Game::word() **const** {  **return** \_word;  }  **void** Game::clear\_mmap() {  \_mtx.lock();  \_mmap[**0**] = MessageType::wait;  \_mmap[**1**] = '\0';  \_mtx.unlock();  }  **int** Game::players() **const** {  **return** \_players;  }  **int**& Game::players() {  **return** \_players;  }  **const** std::string& Game::name() **const** {  **return** \_name;  }  MemoryMap<**char**>& Game::mmap() {  **return** \_mmap;  }  Mutex& Game::mtx() {  **return** \_mtx;  }  **int** Game::winner\_id() **const** {  **return** \_winner\_id;  }  std::string Game::check\_word(**const** std::string& word, **int** player\_id) {  **int** cows = **0**;  std::string str\_cows = "";  **int** bulls = **0**;  std::string str\_bulls = "";  **for** (**int** i = **0**; i < word.size(); ++i) {  **if** (word[i] == \_word[i]) {  ++bulls;  str\_bulls += word[i];  str\_bulls += ' ';  } **else** **if** (\_letters.find(word[i]) != \_letters.end()) {  ++cows;  str\_cows += word[i];  str\_cows += ' ';  }  }  **if** (bulls == \_word.size() && \_winner\_id == **0**) {  \_winner\_id = player\_id;  }  std::string ans = std::to\_string(cows) + ' ' + str\_cows + ' ' + std::to\_string(bulls) + str\_bulls;  **return** ans;  } |

**client.h**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | #pragma once  #include <unistd.h>  #include <string>  #include <thread>  #include <chrono>  #include "mmap.h"  #include "mutex.h"  #include "message.h"  **namespace** bc {  **class** **Client** {  **private:**  **int** \_id;  MemoryMap<**char**> \_mmap;  Mutex \_mtx;  **int** \_server\_id;  **static** **const** **int** \_mmap\_size = **10000**;  **public:**  Client(**int** server\_id);  ~Client();  **int** id() **const**;  **void** **create\_game**(**const** std::string& game\_name, **int** players);  **void** **connect\_to\_game**(**const** std::string& game\_name);  **void** **connect\_to\_server**();  **void** **send\_message**(**const** Message& msg);  **bool** **ping**();  **void** **clear\_mmap**();  Message **get\_message**();  };  } // bulls & cows |

**client.cpp**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | #include "client.h"  **using** **namespace** bc;  Client::Client(**int** server\_id) : \_id{getpid()}, \_server\_id{server\_id},  \_mmap("bc\_" + std::to\_string(server\_id) + "\_mmap", \_mmap\_size, ModeFlags::read | ModeFlags::write),  \_mtx("bc\_" + std::to\_string(server\_id) + "\_mutex", MutexFlag::connect) {}  Client::~Client() {}  **int** Client::id() **const** {  **return** \_id;  }  **void** Client::create\_game(**const** std::string& game\_name, **int** players) {  // 4 = players\_num + ' ' + state + end of str  **if** (game\_name.size() > \_mmap\_size - **4**) **throw** std::logic\_error("Too long game name");  Message msg;  msg.type = MessageType::server\_create\_game;  msg.data = game\_name + ' ' + std::to\_string(players) + '\0';  send\_message(msg);  }  **void** Client::connect\_to\_game(**const** std::string& game\_name) {  MemoryMap<**char**> mmap("bcgame\_" + game\_name + "\_mmap", \_mmap\_size, ModeFlags::read | ModeFlags::write);  \_mmap = mmap;  Mutex **mtx**("bcgame\_" + game\_name + "\_mutex", MutexFlag::connect);  \_mtx = mtx;  }  **void** Client::connect\_to\_server() {  MemoryMap<**char**> mmap("bc\_" + std::to\_string(\_server\_id) + "\_mmap", \_mmap\_size, ModeFlags::read | ModeFlags::write);  \_mmap = mmap;  Mutex **mtx**("bc\_" + std::to\_string(\_server\_id) + "\_mutex", MutexFlag::connect);  \_mtx = mtx;  }  **void** Client::send\_message(**const** Message& msg) {  \_mtx.lock();  \_mmap[**0**] = msg.type;  str\_to\_mmap(msg.data, \_mmap, **1**);  \_mtx.unlock();  }  Message Client::get\_message() {  Message msg;  \_mtx.lock();  msg.type = (MessageType) \_mmap[**0**];  msg.data = mmap\_to\_str(\_mmap, **1**);  \_mtx.unlock();  **return** msg;  }  **bool** Client::ping() {  Message msg;  msg.type = MessageType::ping;  msg.data = "ping";  send\_message(msg);  msg = get\_message();  **int** attempts = **0**;  **while** (msg.type == MessageType::ping) {  ++attempts;  std::this\_thread::sleep\_for(std::chrono::milliseconds(**100**));  msg = get\_message();  **if** (attempts == **20**) **break**;  }  **if** (msg.type == MessageType::wait || msg.type == MessageType::start\_round) **return** true;  **return** false;  }  **void** Client::clear\_mmap() {  \_mtx.lock();  \_mmap[**0**] = MessageType::wait;  \_mmap[**1**] = '\0';  \_mtx.unlock();  } |

**message.h**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #pragma once  #include <string>  **namespace** bc {  **enum** MessageType {  wait,  server\_create\_game,  ping,  start\_round,  end\_game,  guess,  ans\_guess  };  **struct** Message {  MessageType type;  std::string data;  };  } // bulls & cows |

**server\_main.cpp**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | #include <iostream>  #include <thread>  #include <chrono>  #include <sstream>  #include "server.h"  **using** **namespace** bc;  **void** **routine**(Server& ser) {  Message msg;  **while** (true) {  msg = ser.get\_message();  **switch** (msg.type) {  **case** MessageType::wait: {  std::this\_thread::sleep\_for(std::chrono::milliseconds(**1000**));  **break**;  }  **case** MessageType::server\_create\_game: {  std::istringstream oss(msg.data);  std::string game\_name;  **int** players;  oss >> game\_name >> players;  try {  ser.create\_game(game\_name, players);  } **catch** (std::exception& ex) {  std::cout << "Error: " << ex.what() << std::endl;  }  std::cout << "Game **\"**" << game\_name << "**\"** has been created" << std::endl;  ser.clear\_mmap();  **break**;  }  **case** MessageType::ping: {  ser.clear\_mmap();  **break**;  }  }  }  }  **int** **main**() {  std::cout << "Starting server" << std::endl;  Server ser;  std::cout << "Server id: " << ser.id() << std::endl;  routine(ser);  **return** **0**;  } |

**game\_main.cpp**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91 | #include <iostream>  #include <fstream>  #include <string>  #include <thread>  #include <chrono>  #include <sstream>  #include "game.h"  **using** **namespace** bc;  **const** std::string DB\_NAME = "../db/test\_db.bin";  **void** **routine**(Game& game) {  **int** players = **0**;  **int** attempts = **0**;  Message msg;  **bool** is\_game\_ended = false;  **bool** while\_cond = true;  **while** (while\_cond) {  msg = game.get\_message();  **switch** (msg.type) {  **case** MessageType::wait: {  std::this\_thread::sleep\_for(std::chrono::milliseconds(**1000**));  ++attempts;  **if** (attempts > **30**) --game.players();  **if** (players == game.players()) {  **if** (is\_game\_ended) {  while\_cond = false;  **break**;  }  Message call;  call.type = MessageType::start\_round;  call.data = "";  game.send\_message(call);  players = **0**;  attempts = **0**;  }  **break**;  }  **case** MessageType::ping: {  attempts = **0**;  ++players;  game.clear\_mmap();  **break**;  }  **case** MessageType::guess: {  attempts = **0**;  ++players;  std::istringstream iss(msg.data);  **int** pid;  std::string word;  iss >> pid >> word;  std::string ans = game.check\_word(word, pid);  **if** (game.winner\_id() != **0**) {  is\_game\_ended = true;  }  msg.type = MessageType::ans\_guess;  msg.data = std::to\_string(pid) + " " + ans;  game.send\_message(msg);  **break**;  }  **default:**  attempts = **0**;  std::this\_thread::sleep\_for(std::chrono::milliseconds(**1000**));  **break**;  }  }  msg.type = MessageType::end\_game;  msg.data = std::to\_string(game.winner\_id());  game.send\_message(msg);  }  **int** **main**(**int** argc, **char**\*\* argv) {  **if** (argc != **3**) {  std::cerr << "Wrong number of args**\n**";  **return** **1**;  }  std::string filename(argv[**1**]);  filename += "\_log.txt";  std::ofstream ofs(filename);  ofs << "Game started.**\n**";  **int** players = std::stoi(argv[**2**]);  Game game(argv[**1**], players, DB\_NAME);  ofs << "Word: " << game.word() << '\n';  routine(game);  ofs << "Winner id: " << game.winner\_id() << '\n';  ofs << "Game ended.**\n**";  ofs.close();  **return** **0**;  } |

**client\_main.cpp**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162 | #include <iostream>  #include <sstream>  #include "client.h"  **using** **namespace** bc;  **void** **help**() {  std::cout << "1. Create a game**\n**";  std::cout << "2. Connect to the game**\n**";  std::cout << "q. Exit" << std::endl;  }  **void** **game**(Client& cl) {  std::this\_thread::sleep\_for(std::chrono::milliseconds(**500**));  **if** (!cl.ping()) {  std::cerr << "Can't connect to game**\n**";  exit(**1**);  }  std::cout << "Waiting for start" << std::endl;  Message msg;  **bool** is\_new\_round\_started = false;  **while** (true) {  msg = cl.get\_message();  **if** (msg.type == MessageType::start\_round) {  std::cout << "Round started" << std::endl;  is\_new\_round\_started = true;  } **else** **if** (msg.type == MessageType::wait || msg.type == MessageType::guess) {  std::this\_thread::sleep\_for(std::chrono::milliseconds(**1000**));  } **else** **if** (msg.type == MessageType::ans\_guess) {  std::istringstream iss(msg.data);  **int** pid;  iss >> pid;  **if** (pid != cl.id()) {  std::this\_thread::sleep\_for(std::chrono::milliseconds(**100**));  **continue**;  }  **int** cows, bulls;  iss >> cows;  std::cout << "cows " << cows << ": ";  **for** (**int** i = **0**; i < cows; ++i) {  **char** letter;  iss >> letter;  std::cout << letter << ' ';  }  std::cout << '\n';  iss >> bulls;  std::cout << "bulls " << bulls << ": ";  **for** (**int** i = **0**; i < bulls; ++i) {  **char** letter;  iss >> letter;  std::cout << letter << ' ';  }  std::cout << std::endl;  cl.clear\_mmap();  } **else** **if** (msg.type == MessageType::end\_game) {  std::istringstream iss(msg.data);  **int** player\_id;  iss >> player\_id;  **if** (player\_id == cl.id()) {  std::cout << "You win!" << std::endl;  } **else** {  std::cout << "Player " << player\_id << " win." << std::endl;  }  **return**;  }  **if** (is\_new\_round\_started) {  is\_new\_round\_started = false;  std::cout << "Enter a word: ";  std::string word;  std::cin >> word;  **while** (word.size() != **5**) {  std::cout << "Word lenght must be 5" << std::endl;  std::cout << "Enter a word: ";  std::cin >> word;  }  Message guess\_msg;  guess\_msg.type = MessageType::guess;  guess\_msg.data = std::to\_string(cl.id()) + word;  msg = cl.get\_message();  **while** (msg.type != MessageType::start\_round && msg.type != MessageType::wait) {  std::this\_thread::sleep\_for(std::chrono::milliseconds(**1000**));  msg = cl.get\_message();  }  **if** (msg.type == MessageType::end\_game) {  std::istringstream iss(msg.data);  **int** player\_id;  iss >> player\_id;  **if** (player\_id == cl.id()) {  std::cout << "You win!" << std::endl;  } **else** {  std::cout << "Player " << player\_id << " win." << std::endl;  }  }  cl.send\_message(guess\_msg);  }  }  }  **void** **interface**(Client& cl) {  **char** cmd;  std::cout << "Type <h> for help page" << std::endl;  **while** (true) {  std::cout << "> ";  std::cin >> cmd;  **switch** (cmd) {  **case** '1': {  std::string game\_name;  **int** players;  std::cout << "Enter game name: ";  std::cin >> game\_name;  std::cout << "Enter number of players: ";  std::cin >> players;  try {  cl.create\_game(game\_name, players);  } **catch** (std::exception& ex) {  std::cout << "Error: " << ex.what() << std::endl;  **break**;  }  cl.connect\_to\_game(game\_name);  game(cl);  cl.connect\_to\_server();  **break**;  }  **case** '2': {  std::string game\_name;  std::cout << "Enter game name: ";  std::cin >> game\_name;  cl.connect\_to\_game(game\_name);  game(cl);  cl.connect\_to\_server();  **break**;  }  **case** 'q': {  **return**;  }  **case** 'h': {  help();  **break**;  }  **default:** {  std::cout << "Wrong command" << std::endl;  **break**;  }  }  }  }  **int** **main**() {  std::cout << "Starting client" << std::endl;  std::cout << "Enter server id: ";  **int** server\_id;  std::cin >> server\_id;  Client cl(server\_id);  std::cout << "Your client id: " << cl.id() << std::endl;  **if** (!cl.ping()) {  std::cerr << "Can't connect to server**\n**";  **return** **1**;  }  interface(cl);  **return** **0**;  } |

**Протокол работы программы**

**Тестирование:**

cat\_mood@nuclear-box:~/programming/mai-os-labs/kp/build$ ./kp\_server

Starting server

Server id: 13996

Game "game" has been created

cat\_mood@nuclear-box:~/programming/mai-os-labs/kp/build$ ./kp\_client

Starting client

Enter server id: 13996

Your client id: 14035

Type <h> for help page

> h

1. Create a game

2. Connect to the game

q. Exit

> 2

Enter game name: game

Waiting for start

Round started

Enter a word: which

cows 0:

bulls 0:

Round started

Enter a word: about

cows 0:

bulls 5: a b o u t

You win!

cat\_mood@nuclear-box:~/programming/mai-os-labs/kp/build$ ./kp\_client

Starting client

Enter server id: 13996

Your client id: 14137

Type <h> for help page

> 1

Enter game name: game

Enter number of players: 2

Waiting for start

Round started

Enter a word: there

cows 1: t

bulls 0:

Round started

Enter a word: about

cows 0:

bulls 5: a b o u t

Player 14035 win.

**Вывод**

В ходе лабораторной работы я получил опыт работы с shared memory, shared mutex, разработки игр. Я столкнулся с проблемой синхронизации процессов, которую решил с помощью shared mutex и таймера прерывания процесса. Я разработал собственные обёртки и интерфейсы для shared mutex и memory map, сделал систему сообщений, с помощью которой процессы общаются между собой, отладил эту систему.