

Санкт-Петербургский Политехнический Университет Петра Великого  
Институт компьютерных наук и технологий  
Кафедра компьютерных систем и программных технологий

# Операционные системы

Отчет по лабораторной работе №7  
Средства синхронизации потоков и процессов в ОС Windows

**Работу выполнил:**

Черевичник Андрей

Группа: 43501/3

**Преподаватель:**

Малышев Игорь Алексеевич

Санкт-Петербург  
2017

# Содержание

<b>1</b>	<b>Лабораторная работа №7</b>	<b>2</b>
1.1	Цель работы . . . . .	2
1.2	Программа работы . . . . .	2
1.3	Ход работы . . . . .	3
1.3.1	Глава 1. Средства синхронизации . . . . .	3
1.3.2	Глава 2. Задача «Читатели и писатели» . . . . .	22
1.3.3	Глава 3. Задача «Обедающие философы» . . . . .	54
1.4	Вывод . . . . .	57

# Лабораторная работа №7

## 1.1 Цель работы

Изучение средств синхронизации доступа к ресурсам потоков и процессов в ОС семейства Windows.

## 1.2 Программа работы

### Глава 1. Средства синхронизации

1. Создать два потока: производителя и потребителя. Потоки разделяют целочисленный массив, в который заносятся производимые и извлекаются потребляемые данные. Для наглядности и контроля за происходящим в буфер помещается нарастающее значение, однозначно идентифицирующее производителя и номер его очередной посылки.
2. Синхронизировать программу мьютексами.
3. Синхронизировать программу семафорами.
4. Синхронизировать программу критическими секциями.
5. Синхронизировать программу событиями.
6. Синхронизировать программу критическими секциями с использованием условных переменных.

### Глава 2. Задача "Читатели и писатели"

1. Решить задачу одного писателя и N читателей. Для синхронизации разрешено использовать только объекты-события, в качестве разделяемого ресурса – разделяемую память (share memory). Писатель пишет в share memory сообщение и ждет, пока все читатели не прочитают данное сообщение.
2. Решение задачи для разных процессов.
3. Модифицировать предложенное решение таким образом, чтобы «читатели» не имели доступа к памяти по записи.
4. Предложить более рациональное решение задачи, используя другие средства синхронизации или их сочетание.
5. Разработать клиент-серверное приложение для полной задачи «читатели - писатели» с собственной системой ограничений на доступ каждого «читателя» к информации. Программа должна поддерживать сетевое функционирование.

### Глава 3. Задача "Обедающие философы"

1. Составить модель и программу для задачи: в одном пансионе, открытом богатым филантропом, были собраны пять знаменитых философов. Предаваясь размышлениям, они независимо друг от друга заходили обедать в общую столовую. В столовой стоял стол, вокруг которого были поставлены стулья. Каждому философу свой стул. Слева от философа лежало по вилке, а в центре стола стояла большая тарелка спагетти. Спагетти можно было есть только двумя вилками, а потому, сев за стол, философ должен был взять вилку соседа справа (если она, конечно, свободна).

## 1.3 Ход работы

### 1.3.1 Глава 1. Средства синхронизации

В ОС Windows существует множество способов для синхронизации потоков и процессов:

1. Мьютексы.
2. Семафоры.
3. Критические секции.
4. Объекты события.
5. Условные переменные.
6. Функции ожидания.

Каждый из этих способов синхронизации имеет свои преимущества и недостатки.

#### 1. Создание многопоточного приложения для последующей синхронизации

Для демонстрации работы каждого из этих средств синхронизации была написана программа, которая имеет производителя и потребителя. Потоки разделяют целочисленный массив, в который заносятся производимые и извлекаются потребляемые данные. Для наглядности и контроля за происходящим в буфер помещается нарастающее значение, однозначно идентифицирующее производителя и номер его очередной посылки.

Главный поток определяет конфигурацию программы, создает все структуры данных и потоки. После создания всех объектов поток ожидает завершения созданных потоков, а затем производит удаление всех созданных объектов.

В программе предусматривается возможность настройки:

1. Количества писателей и читателей.
2. Установки задержек для читателей и писателей.
3. Времени жизни приложения.
4. Размера очереди сообщения.

В качестве общего ресурса используется очередь FIFO. Потоки-писатели добавляют в очередь сообщения, потоки-читатели забирают.

Реализация программы:

```
1 #include <stdio.h>
2 #include <windows.h>
3 #include <conio.h>
4 #include <iostream>
5 #include <fstream>
6 #include <string>
7 #include <vector>
8
9 // Структура начальной конфигурации приложения
10 struct Configuration {
11     // Количество потоков читателей
12     int readersCount;
13     // Количество потоков писателей
14     int writersCount;
15     // Размер очереди
16     int queueSize;
17     // Задержка читателей в миллисекундах
18     int readersDelay;
19     // Задержка писателей в миллисекундах
20     int writersDelay;
21     // Время жизни приложения
22     int timeToLive;
23 };
```

```

24
25 // Очередь FIFO
26 struct Queue {
27     // Массив сообщений
28     char** messages;
29     // Индекс записи
30     int writeIndex;
31     // Индекс чтения
32     int readIndex;
33     // Размер очереди
34     int queueSize;
35     // Флаг заполненности очереди
36     bool isFull;
37 };
38
39 static const char* CONFIG_PATH = "config.ini";
40 static const __int64 TIMER_CONSTANT = -1 * 10000000;
41 static const char* EXIT_MESSAGE = "q";
42
43 // Глобальные структуры конфигурации и очереди
44 struct Configuration globalConfig;
45 struct Queue globalQueue;
46
47 // Массив, содержащий дескрипторы всех потоков
48 std::vector<HANDLE> handlers;
49 // Переменная, которая завершает все потоки
50 bool interrupt = false;
51
52 HANDLE mutex;
53
54 // Функция для заполнения структуры конфигурации
55 void setConfig(const std::string filename, Configuration* config);
56 // Получение параметра структуры конфигурации
57 void getConfigParam(int* parametr, std::istream& stream);
58 // Функция, конфигурирующая и запускающая таймер
59 HANDLE createAndStartWaitableTimer(const unsigned int secondsCount);
60
61 // Функция, создающая все потоки
62 void createAllThreads(Configuration* config);
63 // Обработчики потоков
64 DWORD WINAPI threadReaderExecutor(LPVOID argument);
65 DWORD WINAPI threadWriterExecutor(LPVOID argument);
66 DWORD WINAPI threadTimeManagerExecutor(LPVOID argument);
67
68 int main(int argc, char* argv[]) {
69     // Получение названия файла конфигурации
70     std::string configFilename;
71     configFilename = (argc < 2) ? CONFIG_PATH : argv[1];
72     setConfig(configFilename, &globalConfig);
73
74     // Создание всех потоков
75     createAllThreads(&globalConfig);
76
77     // Заполнение структуры очереди
78     globalQueue.isFull = false;
79     globalQueue.readIndex = 0;
80     globalQueue.writeIndex = 0;
81     globalQueue.queueSize = globalConfig.queueSize;
82     globalQueue.messages = new char*[globalConfig.queueSize];
83
84     // Старт всех потоков
85     for(auto& current : handlers)
86         ResumeThread(current);
87
88     // Ожидание завершения всех потоков
89     WaitForMultipleObjects(handlers.size(), &(handlers[0]), TRUE, INFINITE);

```

```

90
91 // Закрываем дескрипторы потоков
92 for(auto& current : handlers)
93     CloseHandle(current);
94
95 // Закрываем дескриптор мьютекса
96 CloseHandle(mutex);
97
98 std::cout << "Press \"Enter\" to exit." << std::endl;
99 std::getchar();
100
101 return 0x0;
102 }
103
104 void getConfigParam(int* parametr, std::istream& stream) {
105     // Получение параметра структуры конфигурации
106
107     std::string line;
108
109     if(stream.eof())
110         throw std::exception();
111
112     std::getline(stream, line);
113     *parametr = std::stoi(line);
114 }
115
116 void setConfig(const std::string configFilename, Configuration* config) //функция установки
    конфигурации
117 {
118     // Открываем конфигурационный файл
119     auto stream = std::ifstream(configFilename.data());
120     if(!stream.is_open()) {
121         std::cerr << "It's impossible to open config file." << std::endl;
122         exit(0x1);
123     }
124
125     // Заполняем структуру конфигурации
126     std::string line;
127     try {
128         getConfigParam(&(config->readersCount), stream);
129         getConfigParam(&(config->readersDelay), stream);
130         getConfigParam(&(config->writersCount), stream);
131         getConfigParam(&(config->writersDelay), stream);
132         getConfigParam(&(config->queueSize), stream);
133         getConfigParam(&(config->timeToLive), stream);
134     }
135     catch(const std::exception& exception) {
136         std::cerr << "It's impossible to parse config file." << std::endl;
137         stream.close();
138         exit(0x2);
139     }
140
141     // Проверка корректности полученных данных
142     if(
143         config->readersCount <= 0 || config->readersDelay <= 0 ||
144         config->writersCount <= 0 || config->writersDelay <= 0 ||
145         config->queueSize <= 0 || config->timeToLive == 0
146     ) {
147         std::cerr << "Wrong config file values." << std::endl;
148         stream.close();
149         exit(0x3);
150     }
151
152     stream.close();
153
154     std::cout << "—— Current config ——" << std::endl

```

```

155     << "Readers count: " << config->readersCount << std::endl
156     << "Readers delay: " << config->readersDelay << std::endl
157     << "Writers count: " << config->writersCount << std::endl
158     << "Writers delay: " << config->writersDelay << std::endl
159     << "Queue size: " << config->queueSize << std::endl
160     << "Time to live: " << config->timeToLive << std::endl << std::endl;
161 }
162
163 void createAllThreads(Configuration * config) {
164     // Создание всех потоков читателей
165     for(int readerIndex = 0; readerIndex < config->readersCount; ++readerIndex) {
166         HANDLE reader = CreateThread(nullptr, NULL, threadReaderExecutor, LPVOID(readerIndex)
167         , CREATE_SUSPENDED, nullptr);
168         if(!reader) {
169             std::cerr << "It's impossible to create reader." << std::endl;
170             exit(0x4);
171         }
172         handlers.push_back(reader);
173     }
174
175     std::cout << "Readers have been successfully created." << std::endl;
176
177     // Создание всех потоков писателей
178     for(int writerIndex = 0; writerIndex < config->readersCount; ++writerIndex) {
179         HANDLE writer = CreateThread(nullptr, NULL, threadWriterExecutor, LPVOID(writerIndex)
180         , CREATE_SUSPENDED, nullptr);
181         if(!writer) {
182             std::cerr << "It's impossible to create reader." << std::endl;
183             exit(0x5);
184         }
185         handlers.push_back(writer);
186     }
187
188     std::cout << "Writers have been successfully created." << std::endl;
189
190     // Создание временного обработчика
191     HANDLE timeManager = CreateThread(nullptr, NULL, threadTimeManagerExecutor, LPVOID(
192     config->timeToLive), CREATE_SUSPENDED, nullptr);
193     if(!timeManager) {
194         std::cerr << "It's impossible to create time manager." << std::endl;
195         exit(0x6);
196     }
197
198     handlers.push_back(timeManager);
199
200     std::cout << "Time manager has been successfully created." << std::endl;
201 }
202
203 HANDLE createAndStartWaitableTimer(const unsigned int secondsCount) //создание, установка и
204     запуск таймера
205 {
206     // Вычисляем временные характеристики таймера
207     __int64 endTimeValue = TIMER_CONSTANT * secondsCount;
208     LARGE_INTEGER endTimeStruct;
209     endTimeStruct.LowPart = DWORD(endTimeValue & 0xFFFFFFFF);
210     endTimeStruct.HighPart = LONG(endTimeValue >> 0x20);
211
212     // Создаем таймер
213     HANDLE timer = CreateWaitableTimer(nullptr, false, nullptr);
214     SetWaitableTimer(timer, &endTimeStruct, NULL, nullptr, nullptr, false);
215     return timer;
216 }

```

```

217
218 DWORD WINAPI threadTimeManagerExecutor(LPVOID argument) {
219     int timeToLive = int(argument);
220     std::string message;
221
222     // Если время жизни отрицательное, то выходим по введенной строке
223     if(timeToLive < 0)
224         while(true) {
225             message.clear();
226             std::getline(std::cin, message);
227
228             if(message == EXIT_MESSAGE) {
229                 interrupt = true;
230                 break;
231             }
232         }
233     // Если время жизни положительное, то выходим по таймеру
234     else {
235         HANDLE timer = createAndStartWaitableTimer(timeToLive);
236         WaitForSingleObject(timer, INFINITE);
237         interrupt = true;
238         CloseHandle(timer);
239     }
240
241     printf("Time manager finished.\n");
242     return NULL;
243 }
244
245 DWORD WINAPI threadReaderExecutor(LPVOID argument) {
246     int threadId = int(argument);
247     while(!interrupt) {
248         if(globalQueue.readIndex != globalQueue.writeIndex || globalQueue.isFull) {
249             // Если в очереди есть данные
250             printf("Reader #%d. Get data \"%s\" from position %d.\n", threadId, globalQueue.
messages[globalQueue.readIndex], globalQueue.readIndex);
251
252             // Берем данные из очереди
253             delete [] globalQueue.messages[globalQueue.readIndex];
254             globalQueue.messages[globalQueue.readIndex] = nullptr;
255             globalQueue.isFull = false;
256
257             printf("Queue size: %d.\n", globalQueue.queueSize);
258             printf("Queue write index: %d.\n", globalQueue.writeIndex);
259
260             std::getchar();
261
262             // Вычисляем индекс читателя
263             globalQueue.readIndex = (globalQueue.readIndex + 1) % globalQueue.queueSize;
264         }
265
266         Sleep(globalConfig.readersDelay);
267     }
268
269     printf("Reader finished.\n");
270     return NULL;
271 }
272
273 DWORD WINAPI threadWriterExecutor(LPVOID argument) {
274     int threadId = int(argument);
275
276     int messageIndex = 0;
277     std::string message;
278     while(!interrupt) {
279         if(globalQueue.readIndex != globalQueue.writeIndex || !globalQueue.isFull) {
280             // Если в очереди нет данных

```



```

282 // Записываем данные в очередь
283 message = "Writer #" + std::to_string(threadId) + ". Message number #" + std::
to_string(messageIndex++) + ".";
284 globalQueue.messages[globalQueue.writeIndex] = _strdup(message.data());
285
286 printf("Writer #%d. Put data \"%s\" in position %d.\n", threadId, globalQueue.
messages[globalQueue.writeIndex], globalQueue.writeIndex);
287
288 // Вычисляем индекс писателя
289 globalQueue.writeIndex = (globalQueue.writeIndex + 1) % globalQueue.queueSize;
290 // Получаем флаг заполненности очереди
291 globalQueue.isFull = globalQueue.writeIndex == globalQueue.readIndex;
292 }
293
294 Sleep(globalConfig.writersDelay);
295 }
296
297 printf("Writer finished.\n");
298 return NULL;
299 }

```

Результат работы программы:

```
C:\Users\Desktop\old.exe
----- Current config -----
Readers count: 4
Readers delay: 500
Writers count: 7
Writers delay: 100
Queue size: 10
Time to live: 1

Readers have been successfully created.
Writers have been successfully created.
Time manager has been successfully created.
Writer #2. Put data "Writer #2. Message number #0." in position 0.
Writer #0. Put data "Writer #0. Message number #0." in position 1.
Writer #3. Put data "Writer #3. Message number #0." in position 1.
Reader #2. Get data "Writer #2. Message number #0." from position 0.
Queue size: 10.
Queue write index: 3.
Writer #1. Put data "Writer #1. Message number #0." in position 3.
Reader #3. Get data "(null)" from position 0.
Queue size: 10.
Queue write index: 4.
Writer #2. Put data "Writer #2. Message number #1." in position 4.
Writer #1. Put data "Writer #1. Message number #1." in position 4.
Writer #3. Put data "Writer #3. Message number #1." in position 4.
Writer #0. Put data "Writer #0. Message number #1." in position 4.
Writer #2. Put data "Writer #2. Message number #2." in position 8.
Writer #0. Put data "Writer #0. Message number #2." in position 8.
```

Рис. 1.1

Можно заметить, что результирующие данные не синхронизированны: сообщения появляются вразнобой и проскакивают значения (null). Далее, при помощи различных методов синхронизации мы постараемся избежать этой ситуации.

## 2. Синхронизация мьютексами

Мьютексы гарантируют потокам взаимоисключающий доступ к единственному ресурсу. Они содержат счетчик числа пользователей, счетчик рекурсии и переменную, в которой запоминается идентификатор потока.

Мьютексы ведут себя точно так же, как и критические секции. Однако, если последние являются объектами пользовательского режима, то мьютексы являются объектами ядра. Кроме того, мьютексы позволяют синхронизировать доступ к ресурсу нескольких потоков из разных процессов. При этом можно задать максимальное время ожидания доступа к ресурсу.

Идентификатор потока определяет, какой поток захватил мьютекс, а счетчик рекурсий — сколько раз. У мьютексов много применений, и это наиболее часто используемые объекты ядра. Как правило, с их помощью защищают блок памяти, к которому обращается множество потоков.

Мьютекс создается следующей функцией:

```
HANDLE WINAPI CreateMutex(
    _In_opt_ LPSECURITY_ATTRIBUTES lpMutexAttributes,
    _In_     BOOL bInitialOwner,
    _In_opt_ LPCTSTR lpName
);
```

Для открытия мьютекса используется функция:

```
HANDLE WINAPI OpenMutex(
    _In_ DWORD dwDesiredAccess,
    _In_ BOOL bInheritHandle,
```

```

    _In_ LPCTSTR lpName
);

```

Для захвата ресурса используются функции семейства Wait: WaitForSingleObject, WaitForMultipleObjects и др.

```

DWORD WINAPI WaitForSingleObject(
    _In_ HANDLE hHandle,
    _In_ DWORD dwMilliseconds
);

```

Для мьютексов сделано одно исключение в правилах перехода объектов ядра из одного состояния в другое. Допустим, поток ждет освобождения занятого объекта мьютекса. В этом случае поток обычно засыпает (переходит в состояние ожидания). Однако система проверяет, не совпадает ли идентификатор потока, пытающегося захватить мьютекс, с аналогичным идентификатором у мьютекса. Если они совпадают, система по-прежнему выделяет потоку процессорное время, хотя мьютекс все еще занят.

Мьютекс освобождается функцией ReleaseMutex:

```

BOOL WINAPI ReleaseMutex(
    _In_ HANDLE hMutex
);

```

Эта функция уменьшает счетчик рекурсии в мьютексе на 1. Если данный объект передавался во владение потоку неоднократно, поток обязан вызвать ReleaseMutex столько раз, сколько необходимо для обнуления счетчика рекурсии. Как только счетчик станет равен 0, переменная, хранящая идентификатор потока, тоже обнулится, и мьютекс освободится.

Изменения в исходной программе, для синхронизации мьютексами:

```

1  ...
2
3  // Глобальный дескриптор
4  HANDLE mutex;
5
6  ...
7
8  int main(int argc, char* argv[]) {
9      ...
10
11     // Создание мьютекса для синхронизации
12     mutex = CreateMutex(nullptr, FALSE, nullptr);
13     if(!mutex) {
14         std::cerr << "It's impossible to create mutex." << std::endl;
15         return 0x7;
16     }
17
18     ...
19
20     // Закрываем дескриптор мьютекса
21     CloseHandle(mutex);
22
23     ...
24 }
25
26 ...
27
28 DWORD WINAPI threadReaderExecutor(LPVOID argument) {
29     ...
30
31     while(!interrupt) {
32         // Захват объекта синхронизации
33         WaitForSingleObject(mutex, INFINITE);
34
35         if(globalQueue.readIndex != globalQueue.writeIndex || globalQueue.isFull) {
36             ...
37         }
38

```

```

39 // Освобождение объекта синхронизации
40 ReleaseMutex(mutex);
41
42 Sleep(globalConfig.readersDelay);
43 }
44
45 ...
46 }
47
48 ...
49
50 DWORD WINAPI threadWriterExecutor(LPVOID argument) {
51     ...
52
53     while(!interrupt) {
54         // Захват объекта синхронизации
55         WaitForSingleObject(mutex, INFINITE);
56
57         if(globalQueue.readIndex != globalQueue.writeIndex || !globalQueue.isFull) {
58             ...
59         }
60
61         // Освобождение объекта синхронизации
62         ReleaseMutex(mutex);
63
64         Sleep(globalConfig.writersDelay);
65     }
66
67     ...
68 }

```

Результат работы программы после синхронизации мьютексами при различных конфигурационных параметрах:

```

C:\Users\Desktop\mutex.exe
----- Current config -----
Readers count: 7
Readers delay: 100
Writers count: 7
Writers delay: 100
Queue size: 10
Time to live: 3

Readers have been successfully created.
Writers have been successfully created.
Time manager has been successfully created.
Writer #3. Put data "Writer #3. Message number #0." in position 0.
Reader #4. Get data "Writer #3. Message number #0." from position 0.
Writer #4. Put data "Writer #4. Message number #0." in position 1.
Writer #2. Put data "Writer #2. Message number #0." in position 2.
Writer #5. Put data "Writer #5. Message number #0." in position 3.
Reader #1. Get data "Writer #4. Message number #0." from position 1.
Writer #1. Put data "Writer #1. Message number #0." in position 4.
Writer #0. Put data "Writer #0. Message number #0." in position 5.
Writer #6. Put data "Writer #6. Message number #0." in position 6.
Reader #5. Get data "Writer #2. Message number #0." from position 2.
Reader #0. Get data "Writer #5. Message number #0." from position 3.
Writer #6. Put data "Writer #6. Message number #1." in position 7.
Writer #0. Put data "Writer #0. Message number #1." in position 8.
Reader #5. Get data "Writer #1. Message number #0." from position 4.
Writer #2. Put data "Writer #2. Message number #1." in position 9.

```

Рис. 1.2

### 3. Синхронизация семафорами

Семафоры используются для учета ресурсов. Как и все объекты ядра, они содержат счетчик числа пользователей, но, кроме того, поддерживают два 32 битных значения со знаком: одно определяет максимальное число ресурсов (контролируемое семафором), другое используется как счетчик текущего числа ресурсов.

```

C:\Users\Desktop\mutex.exe
----- Current config -----
Readers count: 7
Readers delay: 500
Writers count: 7
Writers delay: 100
Queue size: 10
Time to live: 3

Readers have been successfully created.
Writers have been successfully created.
Time manager has been successfully created.
Writer #2. Put data "Writer #2. Message number #0." in position 0.
Writer #0. Put data "Writer #0. Message number #0." in position 1.
Reader #1. Get data "Writer #2. Message number #0." from position 0.
Writer #6. Put data "Writer #6. Message number #0." in position 2.
Reader #4. Get data "Writer #0. Message number #0." from position 1.
Writer #3. Put data "Writer #3. Message number #0." in position 3.
Writer #1. Put data "Writer #1. Message number #0." in position 4.
Reader #5. Get data "Writer #6. Message number #0." from position 2.
Writer #5. Put data "Writer #5. Message number #0." in position 5.
Writer #4. Put data "Writer #4. Message number #0." in position 6.
Reader #2. Get data "Writer #3. Message number #0." from position 3.
Writer #2. Put data "Writer #2. Message number #1." in position 7.
Writer #3. Put data "Writer #3. Message number #1." in position 8.
Writer #4. Put data "Writer #4. Message number #1." in position 9.
Writer #1. Put data "Writer #1. Message number #1." in position 0.
Writer #5. Put data "Writer #5. Message number #1." in position 1.
Writer #0. Put data "Writer #0. Message number #1." in position 2.
Writer #6. Put data "Writer #6. Message number #1." in position 3.
Reader #6. Get data "Writer #1. Message number #0." from position 4.
Reader #2. Get data "Writer #5. Message number #0." from position 5.

```

Рис. 1.3

Семафор создается вызовом функции CreateSemaphore:

```

HANDLE WINAPI CreateSemaphore(
    _In_opt_ LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
    _In_     LONG                  lInitialCount,
    _In_     LONG                  lMaximumCount,
    _In_opt_ LPCTSTR               lpName
);

```

Получить существующий семафор можно с помощью функции OpenSemaphore:

```

HANDLE WINAPI OpenSemaphore(
    _In_ DWORD dwDesiredAccess,
    _In_ BOOL bInheritHandle,
    _In_ LPCTSTR lpName
);

```

Для захвата ресурса используются функции семейства Wait: WaitForSingleObject, WaitForMultipleObjects и др.

Семафор освобождается функцией ReleaseSemaphore:

```

BOOL WINAPI ReleaseSemaphore(
    _In_ HANDLE hSemaphore,
    _In_ LONG lReleaseCount,
    _Out_opt_ LPLONG lpPreviousCount
);

```

Изменения в исходной программе, для синхронизации семафорами:

```

1 ...
2
3 // Глобальный дескриптор
4 HANDLE semaphore;
5
6 ...

```

```

7
8 int main(int argc, char* argv[]) {
9     ...
10
11     // Создание семафора для синхронизации
12     semaphore = CreateSemaphore(nullptr, 1, 1, nullptr);
13     if(!semaphore) {
14         std::cerr << "It's impossible to create semaphore." << std::endl;
15         return 0x7;
16     }
17
18     ...
19
20     // Закрываем дескриптор семафора
21     CloseHandle(semaphore);
22
23     ...
24 }
25
26 ...
27
28 DWORD WINAPI threadReaderExecutor(LPVOID argument) {
29     ...
30
31     while(!interrupt) {
32         // Захват объекта синхронизации
33         WaitForSingleObject(semaphore, INFINITE);
34
35         if(globalQueue.readIndex != globalQueue.writeIndex || globalQueue.isFull) {
36             ...
37         }
38
39         // Освобождение объекта синхронизации
40         ReleaseSemaphore(semaphore, 1, nullptr);
41
42         Sleep(globalConfig.readersDelay);
43     }
44
45     ...
46 }
47
48 ...
49
50 DWORD WINAPI threadWriterExecutor(LPVOID argument) {
51     ...
52
53     while(!interrupt) {
54         // Захват объекта синхронизации
55         WaitForSingleObject(semaphore, INFINITE);
56
57         if(globalQueue.readIndex != globalQueue.writeIndex || !globalQueue.isFull) {
58             ...
59         }
60
61         // Освобождение объекта синхронизации
62         ReleaseSemaphore(semaphore, 1, nullptr);
63
64         Sleep(globalConfig.writersDelay);
65     }
66
67     ...
68 }

```

Результат работы программы после синхронизации семафорами при различных конфигурационных параметрах:

```
C:\Users\Desktop\semaphore.exe
----- Current config -----
Readers count: 7
Readers delay: 100
Writers count: 7
Writers delay: 100
Queue size: 10
Time to live: 3

Readers have been successfully created.
Writers have been successfully created.
Time manager has been successfully created.
Writer #3. Put data "Writer #3. Message number #0." in position 0.
Reader #1. Get data "Writer #3. Message number #0." from position 0.
Writer #6. Put data "Writer #6. Message number #0." in position 1.
Reader #5. Get data "Writer #6. Message number #0." from position 1.
Writer #4. Put data "Writer #4. Message number #0." in position 2.
Reader #4. Get data "Writer #4. Message number #0." from position 2.
Writer #0. Put data "Writer #0. Message number #0." in position 3.
Writer #2. Put data "Writer #2. Message number #0." in position 4.
Writer #5. Put data "Writer #5. Message number #0." in position 5.
Reader #2. Get data "Writer #0. Message number #0." from position 3.
Writer #1. Put data "Writer #1. Message number #0." in position 6.
Reader #3. Get data "Writer #2. Message number #0." from position 4.
Reader #1. Get data "Writer #5. Message number #0." from position 5.
Reader #0. Get data "Writer #1. Message number #0." from position 6.
Writer #3. Put data "Writer #3. Message number #1." in position 7.
Writer #1. Put data "Writer #1. Message number #1." in position 8.
```

Рис. 1.4

```
C:\Users\Desktop\semaphore.exe
----- Current config -----
Readers count: 7
Readers delay: 500
Writers count: 7
Writers delay: 100
Queue size: 10
Time to live: 1

Readers have been successfully created.
Writers have been successfully created.
Time manager has been successfully created.
Writer #0. Put data "Writer #0. Message number #0." in position 0.
Writer #1. Put data "Writer #1. Message number #0." in position 1.
Reader #6. Get data "Writer #0. Message number #0." from position 0.
Reader #3. Get data "Writer #1. Message number #0." from position 1.
Writer #5. Put data "Writer #5. Message number #0." in position 2.
Writer #2. Put data "Writer #2. Message number #0." in position 3.
Writer #6. Put data "Writer #6. Message number #0." in position 4.
Writer #3. Put data "Writer #3. Message number #0." in position 5.
Writer #4. Put data "Writer #4. Message number #0." in position 6.
Writer #1. Put data "Writer #1. Message number #1." in position 7.
Writer #5. Put data "Writer #5. Message number #1." in position 8.
Writer #0. Put data "Writer #0. Message number #1." in position 9.
Writer #3. Put data "Writer #3. Message number #1." in position 0.
Writer #4. Put data "Writer #4. Message number #1." in position 1.
Reader #4. Get data "Writer #5. Message number #0." from position 2.
Reader #3. Get data "Writer #2. Message number #0." from position 3.
Reader #1. Get data "Writer #6. Message number #0." from position 4.
Reader #2. Get data "Writer #3. Message number #0." from position 5.
```

Рис. 1.5

#### 4. Синхронизация критическими секциями

Критическая секция — небольшой участок кода, требующий монопольного доступа к каким-то общим данным. Она позволяет сделать так, чтобы одновременно только один поток получал доступ к определенному ресурсу.

Важное преимущество критической секции перед мьютексами и семафорами заключается в том, что она работает намного быстрее, ввиду того что не является объектом ядра.

Инициализация критической секции производится функцией

```
void WINAPI InitializeCriticalSection(
```

```

    _Out_ LPCRITICAL_SECTION lpCriticalSection
);

```

Захват секции производится функцией EnterCriticalSection:

```

void WINAPI EnterCriticalSection(
    _Inout_ LPCRITICAL_SECTION lpCriticalSection
);

```

Освобождение секции производится функцией LeaveCriticalSection:

```

void WINAPI LeaveCriticalSection(
    _Inout_ LPCRITICAL_SECTION lpCriticalSection
);

```

Удаление критической секции производится функцией DeleteCriticalSection:

```

void WINAPI DeleteCriticalSection(
    _Inout_ LPCRITICAL_SECTION lpCriticalSection
);

```

Изменения в исходной программе, для синхронизации критическими секциями:

```

1  ...
2
3  // Глобальный объект
4  CRITICAL_SECTION criticalSection;
5
6  ...
7
8  int main(int argc, char* argv[]) {
9      ...
10
11     // Инициализируем критическую секцию
12     InitializeCriticalSection(&criticalSection);
13
14     ...
15
16     // Удаляем критическую секцию
17     DeleteCriticalSection(&criticalSection);
18
19     ...
20 }
21
22 ...
23
24 DWORD WINAPI threadReaderExecutor(LPVOID argument) {
25     ...
26
27     while(!interrupt) {
28         // Захват объекта синхронизации
29         EnterCriticalSection(&criticalSection);
30
31         if(globalQueue.readIndex != globalQueue.writeIndex || globalQueue.isFull) {
32             ...
33         }
34
35         // Освобождение объекта синхронизации
36         LeaveCriticalSection(&criticalSection);
37
38         Sleep(globalConfig.readersDelay);
39     }
40
41     ...
42 }
43

```



```

44 ...
45
46 DWORD WINAPI threadWriterExecutor(LPVOID argument) {
47     ...
48     while(!interrupt) {
49         // Захват объекта синхронизации
50         EnterCriticalSection(&criticalSection);
51
52         if(globalQueue.readIndex != globalQueue.writeIndex || !globalQueue.isFull) {
53             ...
54         }
55
56         // Освобождение объекта синхронизации
57         LeaveCriticalSection(&criticalSection);
58
59         Sleep(globalConfig.writersDelay);
60     }
61
62     ...
63 }
64

```

Результат работы программы после синхронизации критическими секциями при различных конфигурационных параметрах:

```

C:\Users\Desktop\crit.exe
----- Current config -----
Readers count: 7
Readers delay: 100
Writers count: 7
Writers delay: 100
Queue size: 10
Time to live: 3

Readers have been successfully created.
Writers have been successfully created.
Time manager has been successfully created.
Writer #2. Put data "Writer #2. Message number #0." in position 0.
Reader #1. Get data "Writer #2. Message number #0." from position 0.
Writer #1. Put data "Writer #1. Message number #0." in position 1.
Reader #5. Get data "Writer #1. Message number #0." from position 1.
Writer #5. Put data "Writer #5. Message number #0." in position 2.
Writer #0. Put data "Writer #0. Message number #0." in position 3.
Writer #3. Put data "Writer #3. Message number #0." in position 4.
Writer #6. Put data "Writer #6. Message number #0." in position 5.
Writer #4. Put data "Writer #4. Message number #0." in position 6.
Reader #4. Get data "Writer #5. Message number #0." from position 2.
Reader #3. Get data "Writer #0. Message number #0." from position 3.
Reader #6. Get data "Writer #3. Message number #0." from position 4.
Writer #3. Put data "Writer #3. Message number #1." in position 7.
Reader #2. Get data "Writer #6. Message number #0." from position 5.
Reader #0. Get data "Writer #4. Message number #0." from position 6.
Writer #6. Put data "Writer #6. Message number #1." in position 8.
Writer #0. Put data "Writer #0. Message number #1." in position 9.
Writer #2. Put data "Writer #2. Message number #1." in position 0.
Reader #1. Get data "Writer #3. Message number #1." from position 7.
Writer #1. Put data "Writer #1. Message number #1." in position 1.
Writer #5. Put data "Writer #5. Message number #1." in position 2.
Reader #5. Get data "Writer #6. Message number #1." from position 8.

```

Рис. 1.6

## 5. Синхронизация событиями

События - самая примитивная разновидность объектов ядра. Они содержат счетчик числа пользователей (как и все объекты ядра) и две булевы переменные: одна сообщает тип данного объекта-события, другая — его состояние (свободен или занят). События просто уведомляют об окончании какой-либо операции. Объекты-события бывают двух типов: со сбросом вручную и с автосбросом. Первые позволяют возобновлять выполнение сразу нескольких ждущих потоков, вторые только одного.

Событие создается вызовом функции `CreateEvent`:

```

C:\Users\Desktop\crit.exe
----- Current config -----
Readers count: 7
Readers delay: 500
Writers count: 7
Writers delay: 100
Queue size: 10
Time to live: 1

Readers have been successfully created.
Writers have been successfully created.
Time manager has been successfully created.
Writer #2. Put data "Writer #2. Message number #0." in position 0.
Writer #1. Put data "Writer #1. Message number #0." in position 1.
Writer #3. Put data "Writer #3. Message number #0." in position 2.
Reader #2. Get data "Writer #2. Message number #0." from position 0.
Writer #0. Put data "Writer #0. Message number #0." in position 3.
Writer #4. Put data "Writer #4. Message number #0." in position 4.
Reader #6. Get data "Writer #1. Message number #0." from position 1.
Reader #1. Get data "Writer #3. Message number #0." from position 2.
Reader #5. Get data "Writer #0. Message number #0." from position 3.
Writer #5. Put data "Writer #5. Message number #0." in position 5.
Writer #6. Put data "Writer #6. Message number #0." in position 6.
Writer #2. Put data "Writer #2. Message number #1." in position 7.
Writer #6. Put data "Writer #6. Message number #1." in position 8.
Writer #4. Put data "Writer #4. Message number #1." in position 9.
Writer #3. Put data "Writer #3. Message number #1." in position 0.
Writer #0. Put data "Writer #0. Message number #1." in position 1.
Writer #1. Put data "Writer #1. Message number #1." in position 2.
Writer #5. Put data "Writer #5. Message number #1." in position 3.
Reader #1. Get data "Writer #4. Message number #0." from position 4.
Reader #3. Get data "Writer #5. Message number #0." from position 5.
Reader #6. Get data "Writer #6. Message number #0." from position 6.
Reader #0. Get data "Writer #2. Message number #1." from position 7.
Reader #2. Get data "Writer #6. Message number #1." from position 8.
Reader #4. Get data "Writer #4. Message number #1." from position 9.

```

Рис. 1.7

```

HANDLE WINAPI CreateEvent(
    _In_opt_ LPSECURITY_ATTRIBUTES lpEventAttributes,
    _In_     BOOL                   bManualReset,
    _In_     BOOL                   bInitialState,
    _In_opt_ LPCTSTR               lpName
);

```

Для захвата ресурса используются функции семейства Wait: WaitForSingleObject, WaitForMultipleObjects и др.

Освобождение секции производится функцией SetEvent:

```

BOOL WINAPI SetEvent(
    _In_ HANDLE hEvent
);

```

Изменения в исходной программе, для синхронизации событиями:

```

1 ...
2
3 // Глобальный дескриптор
4 HANDLE event;
5
6 ...
7
8 int main(int argc, char* argv[]) {
9     ...
10
11     // Создаем событие
12     event = CreateEvent(nullptr, false, true, nullptr);
13     if (!event) {
14         std::cerr << "It's impossible to create event." << std::endl;
15         return 0x7;

```

```

16     }
17
18     ...
19
20     // Закрываем дескриптор события
21     CloseHandle(event);
22
23     ...
24 }
25
26 ...
27
28 DWORD WINAPI threadReaderExecutor(LPVOID argument) {
29     ...
30
31     while(!interrupt) {
32         // Захват объекта синхронизации
33         WaitForSingleObject(event, INFINITE);
34
35         if(globalQueue.readIndex != globalQueue.writeIndex || globalQueue.isFull) {
36             ...
37         }
38
39         // Освобождение объекта синхронизации
40         SetEvent(event);
41
42         Sleep(globalConfig.readersDelay);
43     }
44
45     ...
46 }
47
48 ...
49
50 DWORD WINAPI threadWriterExecutor(LPVOID argument) {
51     ...
52
53     while(!interrupt) {
54         // Захват объекта синхронизации
55         WaitForSingleObject(event, INFINITE);
56
57         if(globalQueue.readIndex != globalQueue.writeIndex || !globalQueue.isFull) {
58             ...
59         }
60
61         // Освобождение объекта синхронизации
62         SetEvent(event);
63
64         Sleep(globalConfig.writersDelay);
65     }
66
67     ...
68 }

```

Результат работы программы после синхронизации событиями при различных конфигурационных параметрах:

```

C:\Users\Desktop\event.exe
----- Current config -----
Readers count: 7
Readers delay: 100
Writers count: 7
Writers delay: 100
Queue size: 10
Time to live: 3

Readers have been successfully created.
Writers have been successfully created.
Time manager has been successfully created.
Writer #4. Put data "Writer #4. Message number #0." in position 0.
Writer #1. Put data "Writer #1. Message number #0." in position 1.
Reader #2. Get data "Writer #4. Message number #0." from position 0.
Reader #5. Get data "Writer #1. Message number #0." from position 1.
Writer #2. Put data "Writer #2. Message number #0." in position 2.
Reader #3. Get data "Writer #2. Message number #0." from position 2.
Writer #3. Put data "Writer #3. Message number #0." in position 3.
Writer #0. Put data "Writer #0. Message number #0." in position 4.
Writer #6. Put data "Writer #6. Message number #0." in position 5.
Writer #5. Put data "Writer #5. Message number #0." in position 6.
Reader #1. Get data "Writer #3. Message number #0." from position 3.
Reader #0. Get data "Writer #0. Message number #0." from position 4.
Writer #5. Put data "Writer #5. Message number #1." in position 7.
Reader #1. Get data "Writer #6. Message number #0." from position 5.
Writer #6. Put data "Writer #6. Message number #1." in position 8.
Reader #3. Get data "Writer #5. Message number #0." from position 6.
Writer #0. Put data "Writer #0. Message number #1." in position 9.
Writer #2. Put data "Writer #2. Message number #1." in position 0.
Writer #3. Put data "Writer #3. Message number #1." in position 1.
Reader #2. Get data "Writer #5. Message number #1." from position 7.
Reader #6. Get data "Writer #6. Message number #1." from position 8.
Reader #5. Get data "Writer #0. Message number #1." from position 9.
Reader #4. Get data "Writer #2. Message number #1." from position 0.

```

Рис. 1.8

```

C:\Users\Desktop\event.exe
----- Current config -----
Readers count: 7
Readers delay: 500
Writers count: 7
Writers delay: 100
Queue size: 10
Time to live: 1

Readers have been successfully created.
Writers have been successfully created.
Time manager has been successfully created.
Writer #1. Put data "Writer #1. Message number #0." in position 0.
Writer #0. Put data "Writer #0. Message number #0." in position 1.
Writer #3. Put data "Writer #3. Message number #0." in position 2.
Reader #5. Get data "Writer #1. Message number #0." from position 0.
Writer #6. Put data "Writer #6. Message number #0." in position 3.
Reader #6. Get data "Writer #0. Message number #0." from position 1.
Writer #2. Put data "Writer #2. Message number #0." in position 4.
Writer #4. Put data "Writer #4. Message number #0." in position 5.
Writer #5. Put data "Writer #5. Message number #0." in position 6.
Writer #1. Put data "Writer #1. Message number #1." in position 7.
Writer #5. Put data "Writer #5. Message number #1." in position 8.
Writer #4. Put data "Writer #4. Message number #1." in position 9.
Writer #2. Put data "Writer #2. Message number #1." in position 0.
Writer #3. Put data "Writer #3. Message number #1." in position 1.
Reader #1. Get data "Writer #3. Message number #0." from position 2.
Reader #4. Get data "Writer #6. Message number #0." from position 3.
Reader #3. Get data "Writer #2. Message number #0." from position 4.
Reader #5. Get data "Writer #4. Message number #0." from position 5.
Reader #6. Get data "Writer #5. Message number #0." from position 6.
Reader #0. Get data "Writer #1. Message number #1." from position 7.
Reader #2. Get data "Writer #5. Message number #1." from position 8.
Writer #1. Put data "Writer #1. Message number #2." in position 2.
Writer #6. Put data "Writer #6. Message number #1." in position 3.

```

Рис. 1.9

## 6. Синхронизация критическими секциями с использованием условных переменных

Сами по себе условные переменные не могут применяться, а только в сочетании с другими средствами синхронизации (например, с критическими секциями). Условные переменные отсутствуют в Windows 2003 и XP.

Инициализация условной переменной производится функцией `InitializeConditionVariable`:

```
VOID WINAPI InitializeConditionVariable(  
    _Out_ PCONDITION_VARIABLE ConditionVariable  
);
```

Для ожидания восстановления условной переменной используется функция `SleepConditionVariableCS`:

```
BOOL WINAPI SleepConditionVariableCS(  
    _Inout_ PCONDITION_VARIABLE ConditionVariable,  
    _Inout_ PCRITICAL_SECTION CriticalSection,  
    _In_     DWORD dwMilliseconds  
);
```

Для пробуждения потоков, ожидающих условной переменной используются функция:

```
VOID WINAPI WakeConditionVariable(  
    _Inout_ PCONDITION_VARIABLE ConditionVariable  
);
```

Изменения в исходной программе, для синхронизации критическими секциями с использованием условных переменных:

```
1  ...  
2  
3  static const int TIMEOUT = 5000;  
4  
5  CRITICAL_SECTION criticalSection;  
6  CONDITION_VARIABLE readConditionVariable;  
7  CONDITION_VARIABLE writeConditionVariable;  
8  
9  ...  
10  
11 int main(int argc, char* argv[]) {  
12     ...  
13  
14     // Инициализируем критическую секцию и условные переменные  
15     InitializeCriticalSection(&criticalSection);  
16     InitializeConditionVariable(&readConditionVariable);  
17     InitializeConditionVariable(&writeConditionVariable);  
18  
19     ...  
20  
21     // Ожидание завершения всех потоков  
22     WaitForMultipleObjects(handlers.size(), &(handlers[0]), TRUE, TIMEOUT);  
23  
24     ...  
25  
26     // Удаляем критическую секцию  
27     DeleteCriticalSection(&criticalSection);  
28  
29     ...  
30 }  
31  
32 ...  
33  
34 DWORD WINAPI threadReaderExecutor(LPVOID argument) {  
35     int threadId = int(argument);  
36     while(!interrupt) {  
37         EnterCriticalSection(&criticalSection);  
38
```

```

39 // Ожидаем пока в очереди не освободится место
40 while(!(globalQueue.readIndex != globalQueue.writeIndex || globalQueue.isFull))
41     SleepConditionVariableCS(&readConditionVariable, &criticalSection, INFINITE);
42
43 // Если в очереди есть данные
44 printf("Reader #%d. Get data \"%s\" from position %d.\n", threadId, globalQueue.
messages[globalQueue.readIndex], globalQueue.readIndex);
45
46 // Берем данные из очереди
47 delete [] globalQueue.messages[globalQueue.readIndex];
48 globalQueue.messages[globalQueue.readIndex] = nullptr;
49 globalQueue.isFull = false;
50
51 // Вычисляем индекс читателя
52 globalQueue.readIndex = (globalQueue.readIndex + 1) % globalQueue.queueSize;
53
54 // Посылаем сигнал потокамписателям—
55 WakeConditionVariable(&writeConditionVariable);
56 // Освобождение объекта синхронизации
57 LeaveCriticalSection(&criticalSection);
58
59 Sleep(globalConfig.readersDelay);
60 }
61
62 printf("Reader finished.\n");
63 return NULL;
64 }
65
66 ...
67
68 DWORD WINAPI threadWriterExecutor(LPVOID argument) {
69     int threadId = int(argument);
70
71     int messageIndex = 0;
72     std::string message;
73     while(!interrupt) {
74         // Захват объекта синхронизации
75         EnterCriticalSection(&criticalSection);
76
77         // Ожидаем пока в очереди не освободится место
78         while(!(globalQueue.readIndex != globalQueue.writeIndex || !globalQueue.isFull))
79             SleepConditionVariableCS(&writeConditionVariable, &criticalSection, INFINITE);
80
81         // Записываем данные в очередь
82         message = "Writer #" + std::to_string(threadId) + ". Message number #" + std::
to_string(messageIndex++) + ".";
83         globalQueue.messages[globalQueue.writeIndex] = _strdup(message.data());
84
85         printf("Writer #%d. Put data \"%s\" in position %d.\n", threadId, globalQueue.
messages[globalQueue.writeIndex], globalQueue.writeIndex);
86
87         // Вычисляем индекс писателя
88         globalQueue.writeIndex = (globalQueue.writeIndex + 1) % globalQueue.queueSize;
89
90         if(globalQueue.isFull)
91             printf("Writer #%d. Queue is full.", threadId);
92
93         // Посылаем сигнал потокамчитателям—
94         WakeConditionVariable(&readConditionVariable);
95         // Освобождение объекта синхронизации
96         LeaveCriticalSection(&criticalSection);
97
98         Sleep(globalConfig.writersDelay);
99     }
100
101     printf("Writer finished.\n");

```

```

102     return NULL;
103 }

```

Результат работы программы после синхронизации критическими секциями с использованием условных переменных при различных конфигурационных параметрах:

```

C:\Users\Desktop\var.exe
----- Current config -----
Readers count: 7
Readers delay: 100
Writers count: 7
Writers delay: 100
Queue size: 10
Time to live: 3

Readers have been successfully created.
Writers have been successfully created.
Time manager has been successfully created.
Writer #3. Put data "Writer #3. Message number #0." in position 0.
Reader #4. Get data "Writer #3. Message number #0." from position 0.
Writer #0. Put data "Writer #0. Message number #0." in position 1.
Reader #3. Get data "Writer #0. Message number #0." from position 1.
Writer #1. Put data "Writer #1. Message number #0." in position 2.
Reader #1. Get data "Writer #1. Message number #0." from position 2.
Writer #4. Put data "Writer #4. Message number #0." in position 3.
Writer #2. Put data "Writer #2. Message number #0." in position 4.
Reader #6. Get data "Writer #4. Message number #0." from position 3.
Writer #5. Put data "Writer #5. Message number #0." in position 5.
Reader #5. Get data "Writer #2. Message number #0." from position 4.
Writer #6. Put data "Writer #6. Message number #0." in position 6.
Reader #0. Get data "Writer #5. Message number #0." from position 5.
Reader #2. Get data "Writer #6. Message number #0." from position 6.
Writer #3. Put data "Writer #3. Message number #1." in position 7.
Reader #2. Get data "Writer #3. Message number #1." from position 7.
Writer #4. Put data "Writer #4. Message number #1." in position 8.
Reader #1. Get data "Writer #4. Message number #1." from position 8.
Writer #0. Put data "Writer #0. Message number #1." in position 9.
Reader #3. Get data "Writer #0. Message number #1." from position 9.
Writer #2. Put data "Writer #2. Message number #1." in position 0.

```

Рис. 1.10

```

C:\Users\Desktop\var.exe
----- Current config -----
Readers count: 7
Readers delay: 500
Writers count: 7
Writers delay: 100
Queue size: 10
Time to live: 1

Readers have been successfully created.
Writers have been successfully created.
Time manager has been successfully created.
Writer #1. Put data "Writer #1. Message number #0." in position 0.
Reader #4. Get data "Writer #1. Message number #0." from position 0.
Writer #2. Put data "Writer #2. Message number #0." in position 1.
Reader #1. Get data "Writer #2. Message number #0." from position 1.
Writer #0. Put data "Writer #0. Message number #0." in position 2.
Reader #2. Get data "Writer #0. Message number #0." from position 2.
Writer #3. Put data "Writer #3. Message number #0." in position 3.
Reader #3. Get data "Writer #3. Message number #0." from position 3.
Writer #5. Put data "Writer #5. Message number #0." in position 4.
Writer #4. Put data "Writer #4. Message number #0." in position 5.
Reader #0. Get data "Writer #5. Message number #0." from position 4.
Writer #6. Put data "Writer #6. Message number #0." in position 6.
Reader #6. Get data "Writer #4. Message number #0." from position 5.
Reader #5. Get data "Writer #6. Message number #0." from position 6.
Writer #0. Put data "Writer #0. Message number #1." in position 7.
Writer #6. Put data "Writer #6. Message number #1." in position 8.
Writer #2. Put data "Writer #2. Message number #1." in position 9.
Writer #4. Put data "Writer #4. Message number #1." in position 0.

```

Рис. 1.11

### 1.3.2 Глава 2. Задача «Читатели и писатели»

Задача заключается в реализации одного писателя и множества читателей. Писатель пишет сообщение и ожидает пока все читатели его не прочтут.

#### 1. Использование разделяемой памяти и одного процесса для решения задачи

Система может проецировать на оперативную память не только файл размещения, но и любой другой файл. Приложения могут использовать эту возможность. Это может использоваться для обеспечения более быстрого доступа к файлам, а также для совместного использования памяти.

Такие объекты называются проекциями файлов на оперативную память. Для создания проекции файла вызывается функция `CreateFileMapping`:

```
HANDLE WINAPI CreateFileMapping(
    _In_     HANDLE      hFile,
    _In_opt_ LPSECURITY_ATTRIBUTES lpAttributes,
    _In_     DWORD       flProtect,
    _In_     DWORD       dwMaximumSizeHigh,
    _In_     DWORD       dwMaximumSizeLow,
    _In_opt_ LPCTSTR     lpName
);
```

Для получения дескриптора уже созданного отображения используется функция `OpenFileMapping`:

```
HANDLE WINAPI OpenFileMapping(
    _In_  DWORD      dwDesiredAccess,
    _In_  BOOL       bInheritHandle,
    _In_  LPCTSTR    lpName
);
```

Отображение файла на память процесса осуществляется с помощью функции `MapViewOfFile`:

```
LPVOID WINAPI MapViewOfFile(
    _In_  HANDLE      hFileMappingObject,
    _In_  DWORD       dwDesiredAccess,
    _In_  DWORD       dwFileOffsetHigh,
    _In_  DWORD       dwFileOffsetLow,
    _In_  SIZE_T      dwNumberOfBytesToMap
);
```

Для очистки памяти используется функция `UnmapViewOfFile`:

```
BOOL WINAPI UnmapViewOfFile(
    _In_  LPCVOID     lpBaseAddress
);
```

Для синхронизации потоков потребовалось пять объектов-событий:

1. *eventCanRead* – означает, что поток-писатель записал сообщение в память, и его могут читать потоки-читатели (ручной сброс);
2. *eventCanWrite* – означает, что все потоки-читатели получили сообщение и готовы к приему следующего (автосброс);
3. *eventAllRead* – требуется для приведения в готовность потоков-читателей (ручной сброс);
4. *eventChangeCount* – событие для разрешения работы со счетчиком (количество потоков-читателей, которые прошли заданный участок кода) (автосброс);
5. *eventExit* – устанавливается потоком-планировщиком (ручной сброс).

Поток-писатель ждет, пока все потоки-читатели прочитают сообщение, а затем пишет новое сообщение. После записи он разрешает чтение.



Решение задачи «Читатели и писатели»:

```

1 #include <stdio.h>
2 #include <windows.h>
3 #include <iostream>
4 #include <fstream>
5 #include <string>
6 #include <vector>
7
8 // Структура начальной конфигурации приложения
9 struct Configuration {
10     // Количество потоков читателей
11     int readersCount;
12     // Количество потоков писателей
13     int writersCount;
14     // Размер очереди
15     int queueSize;
16     // Задержка читателей в миллисекундах
17     int readersDelay;
18     // Задержка писателей в миллисекундах
19     int writersDelay;
20     // Время жизни приложения
21     int timeToLive;
22 };
23
24 static const char* CONFIG_PATH = "config.ini";
25 static const __int64 TIMER_CONSTANT = -1 * 10000000;
26 static const char* EXIT_MESSAGE = "q";
27 static const char* SHARE_MEMORY_NAME = "$$SharedMemory$$";
28 static const int BUFFER_SIZE = 1024;
29
30 // Глобальные структуры конфигурации и очереди
31 struct Configuration globalConfig;
32
33 // Массив, содержащий дескрипторы всех потоков
34 std::vector<HANDLE> handlers;
35 // Переменная, которая завершает все потоки
36 bool interrupt = false;
37
38 // События для синхронизации
39 HANDLE eventCanRead, eventCanWrite, eventChangeCount, eventExit, eventAllRead;
40 // Переменные для синхронизации работы потоков
41 int readersCount = 0, readyReadersCount = 0;
42 // Отображение файла
43 HANDLE fileMapping;
44 // Указатели на отображаемую память
45 LPVOID writersMap, readersMap;
46
47 // Функция для заполнения структуры конфигурации
48 void setConfig(const std::string filename, Configuration* config);
49 // Получение параметра структуры конфигурации
50 void getConfigParam(int* parametr, std::istream& stream);
51 // Функция, конфигурирующая и запускающая таймер
52 HANDLE createAndStartWaitableTimer(const unsigned int secondsCount);
53
54 // Функция, создающая все потоки
55 void createAllThreads(Configuration* config);
56 // Обработчики потоков
57 DWORD WINAPI threadReaderExecutor(LPVOID argument);
58 DWORD WINAPI threadWriterExecutor(LPVOID argument);
59 DWORD WINAPI threadTimeManagerExecutor(LPVOID argument);
60
61 int main(int argc, char* argv[]) {
62     // Получение названия файла конфигурации
63     std::string configFileFilename;
64     configFileFilename = (argc < 2) ? CONFIG_PATH : argv[1];
65     setConfig(configFileFilename, &globalConfig);

```

```

66
67 // Создание всех потоков
68 createAllThreads(&globalConfig);
69
70 // Создание отображаемого файла
71 fileMapping = CreateFileMapping(INVALID_HANDLE_VALUE, nullptr, PAGE_READWRITE, NULL,
72     BUFFER_SIZE, SHARE_MEMORY_NAME);
73 if (!fileMapping) {
74     std::cerr << "It's impossible to create shared file." << std::endl;
75     return 0x7;
76 }
77
78 // Отображение файла на адресное пространство процесса
79 writersMap = MapViewOfFile(fileMapping, FILE_MAP_WRITE, NULL, NULL, NULL);
80 readersMap = MapViewOfFile(fileMapping, FILE_MAP_READ, NULL, NULL, NULL);
81 if (!writersMap || !readersMap) {
82     std::cerr << "It's impossible to view map." << std::endl;
83     return 0x8;
84 }
85
86 // Инициализация событий для синхронизации
87 eventCanRead = CreateEvent(nullptr, true, false, nullptr);
88 eventCanWrite = CreateEvent(nullptr, false, false, nullptr);
89 eventAllRead = CreateEvent(nullptr, true, true, nullptr);
90 eventChangeCount = CreateEvent(nullptr, false, true, nullptr);
91 eventExit = CreateEvent(nullptr, true, false, nullptr);
92 if (!eventCanRead || !eventCanWrite || !eventAllRead || !eventChangeCount || !eventExit)
93 {
94     std::cerr << "It's impossible to create event." << std::endl;
95     return 0x9;
96 }
97
98 // Старт всех потоков
99 for(auto& current: handlers)
100     ResumeThread(current);
101
102 // Ожидание завершения всех потоков
103 WaitForMultipleObjects(handlers.size(), &(handlers[0]), TRUE, INFINITE);
104
105 // Закрываем дескрипторы потоков
106 for(auto& current: handlers)
107     CloseHandle(current);
108
109 // Закрываем описатели объектов синхронизации
110 CloseHandle(eventCanRead);
111 CloseHandle(eventCanWrite);
112 CloseHandle(eventAllRead);
113 CloseHandle(eventChangeCount);
114 CloseHandle(eventExit);
115
116 // Удаляем разделяемую память
117 UnmapViewOfFile(writersMap);
118 UnmapViewOfFile(readersMap);
119
120 CloseHandle(fileMapping);
121
122 std::cout << "Press \"Enter\" to exit." << std::endl;
123 std::getchar();
124
125 return 0x0;
126 }
127
128 void getConfigParam(int* parametr, std::istream& stream) {
129     // Получение параметра структуры конфигурации
130     std::string line;

```

```

130
131     if (stream.eof())
132         throw std::exception();
133
134     std::getline(stream, line);
135     *parametr = std::stoi(line);
136 }
137
138 void setConfig(const std::string configFilename, Configuration* config) //функция установки
    конфигурации
139 {
140     // Открываем конфигурационный файл
141     auto stream = std::ifstream(configFilename.data());
142     if (!stream.is_open()) {
143         std::cerr << "It's impossible to open config file." << std::endl;
144         exit(0x1);
145     }
146
147     // Заполняем структуру конфигурации
148     std::string line;
149     try {
150         getConfigParam(&(config->readersCount), stream);
151         getConfigParam(&(config->readersDelay), stream);
152         getConfigParam(&(config->writersCount), stream);
153         getConfigParam(&(config->writersDelay), stream);
154         getConfigParam(&(config->queueSize), stream);
155         getConfigParam(&(config->timeToLive), stream);
156     }
157     catch (const std::exception& exception) {
158         std::cerr << "It's impossible to parse config file." << std::endl;
159         stream.close();
160         exit(0x2);
161     }
162
163     // Проверка корректности полученных данных
164     if (
165         config->readersCount <= 0 || config->readersDelay <= 0 ||
166         config->writersCount <= 0 || config->writersDelay <= 0 ||
167         config->queueSize <= 0 || config->timeToLive == 0
168     ) {
169         std::cerr << "Wrong config file values." << std::endl;
170         stream.close();
171         exit(0x3);
172     }
173
174     stream.close();
175
176     std::cout << "——— Current config ———" << std::endl
177         << "Readers count: " << config->readersCount << std::endl
178         << "Readers delay: " << config->readersDelay << std::endl
179         << "Writers count: " << config->writersCount << std::endl
180         << "Writers delay: " << config->writersDelay << std::endl
181         << "Queue size: " << config->queueSize << std::endl
182         << "Time to live: " << config->timeToLive << std::endl << std::endl;
183 }
184
185 void createAllThreads(Configuration * config) {
186     // Создание всех потоков читателей
187     for (int readerIndex = 0; readerIndex < config->readersCount; ++readerIndex) {
188         HANDLE reader = CreateThread(nullptr, NULL, threadReaderExecutor, LPVOID(readerIndex)
189             , CREATE_SUSPENDED, nullptr);
189         if (!reader) {
190             std::cerr << "It's impossible to create reader." << std::endl;
191             exit(0x4);
192         }
193     }

```

```

194     handlers.push_back(reader);
195 }
196
197 std::cout << "Readers have been successfully created." << std::endl;
198
199
200 // Создание всех потоков писателей
201 for(int writerIndex = 0; writerIndex < config->readersCount; ++writerIndex) {
202     HANDLE writer = CreateThread(nullptr, NULL, threadWriterExecutor, LPVOID(writerIndex)
203     , CREATE_SUSPENDED, nullptr);
204     if(!writer) {
205         std::cerr << "It's impossible to create reader." << std::endl;
206         exit(0x5);
207     }
208     handlers.push_back(writer);
209 }
210
211 std::cout << "Writers have been successfully created." << std::endl;
212
213
214 // Создание временного обработчика
215 HANDLE timeManager = CreateThread(nullptr, NULL, threadTimeManagerExecutor, LPVOID(
216     config->timeToLive), CREATE_SUSPENDED, nullptr);
217 if(!timeManager) {
218     std::cerr << "It's impossible to create time manager." << std::endl;
219     exit(0x6);
220 }
221
222 handlers.push_back(timeManager);
223
224 std::cout << "Time manager has been successfully created." << std::endl;
225 }
226
227 HANDLE createAndStartWaitableTimer(const unsigned int secondsCount) //создание, установка и
228     запуск таймера
229 {
230     // Вычисляем временные характеристики таймера
231     __int64 endTimeValue = TIMER_CONSTANT * secondsCount;
232     LARGE_INTEGER endTimeStruct;
233     endTimeStruct.LowPart = DWORD(endTimeValue & 0xFFFFFFFF);
234     endTimeStruct.HighPart = LONG(endTimeValue >> 0x20);
235
236     // Создаем таймер
237     HANDLE timer = CreateWaitableTimer(nullptr, false, nullptr);
238     SetWaitableTimer(timer, &endTimeStruct, NULL, nullptr, nullptr, false);
239     return timer;
240 }
241
242 DWORD WINAPI threadTimeManagerExecutor(LPVOID argument) {
243     int timeToLive = int(argument);
244     std::string message;
245
246     // Если время жизни отрицательное, то выходим по введенной строке
247     if(timeToLive < 0)
248         while(true) {
249             message.clear();
250             std::getline(std::cin, message);
251
252             if(message == EXIT_MESSAGE) {
253                 interrupt = true;
254                 break;
255             }
256         }
257     // Если время жизни положительное, то выходим по таймеру
258     else {

```

```

257     HANDLE timer = createAndStartWaitableTimer(timeToLive);
258     WaitForSingleObject(timer, INFINITE);
259     SetEvent(eventExit);
260     interrupt = true;
261     CloseHandle(timer);
262 }
263
264 printf("Time manager finished.\n");
265 return NULL;
266 }
267
268 DWORD WINAPI threadReaderExecutor(LPVOID argument) {
269     int threadId = int(argument);
270
271     // Задаем массив событий для функции ожидания
272     HANDLE readerHandlers[2];
273     readerHandlers[0] = eventExit;
274     readerHandlers[1] = eventCanRead;
275
276     while(!interrupt) {
277         // Ожидаем, пока все прочитают
278         WaitForSingleObject(eventAllRead, INFINITE);
279         // Узнаем сколько потоков читателей прошло данную границу
280         WaitForSingleObject(eventChangeCount, INFINITE);
281
282         ++readyReadersCount;
283         if(readyReadersCount == globalConfig.readersCount) {
284             readyReadersCount = 0;
285             // Если все прошли то закрываем событие
286             ResetEvent(eventAllRead);
287             // Разрешаем писать
288             SetEvent(eventCanWrite);
289         }
290
291         // Разрешаем изменять счетчик
292         SetEvent(eventChangeCount);
293
294         DWORD analyzeEvent = WaitForMultipleObjects(2, readerHandlers, false, INFINITE);
295         switch(analyzeEvent) {
296             case WAIT_OBJECT_0:
297                 // Событие завершения потока
298                 printf("Reader #%d finished.\n", threadId);
299                 return NULL;
300
301             case (WAIT_OBJECT_0 + 1):
302                 // Событие чтения
303                 printf("Reader #%d. Read message \"%s\"\n", threadId, (char *) readersMap);
304
305                 // Ожидание уменьшение счетчика количества читателей
306                 WaitForSingleObject(eventChangeCount, INFINITE);
307
308                 // Уменьшаем счетчик количества читателей
309                 --readersCount;
310                 if(readersCount == 0) {
311                     // Запрещаем читать, если мы прочитали последние
312                     ResetEvent(eventCanRead);
313                     // Открываем границу
314                     SetEvent(eventAllRead);
315                 }
316
317                 SetEvent(eventChangeCount);
318                 break;
319
320             default:
321                 std::cerr << "Error in function WaitForMultipleObjects in reader executor." << std
::endl;

```

```

322     exit(0xA);
323 }
324 }
325
326 printf("Reader #%d finished.\n", threadId);
327 return NULL;
328 }
329
330 DWORD WINAPI threadWriterExecutor(LPVOID argument) {
331     int threadId = int(argument);
332
333     int messageIndex = 0;
334
335     // Задаем массив событий для функции ожидания
336     HANDLE writerHandlers[2];
337     writerHandlers[0] = eventExit;
338     writerHandlers[1] = eventCanWrite;
339
340     while(!interrupt) {
341         DWORD analyzeEvent = WaitForMultipleObjects(2, writerHandlers, false, INFINITE);
342         switch(analyzeEvent) {
343             case WAIT_OBJECT_0:
344                 // Событие завершения потока
345                 printf("Writer #%d finished.\n", threadId);
346                 return NULL;
347
348             case WAIT_OBJECT_0 + 1:
349                 // Событие записи
350
351                 ++messageIndex;
352                 readersCount = globalConfig.readersCount;
353
354                 // Запись в разделяемую память
355                 sprintf_s((char *) writersMap, BUFFER_SIZE, "Writer #%d. Message number #%d.",
threadId, messageIndex);
356
357                 printf("Writer #%d. Put data \"%s\".\n", threadId, (char *) writersMap);
358
359                 // Разрешаем читателям прочитать сообщение и ставим событие в занятое
360                 SetEvent(eventCanRead);
361                 break;
362
363             default:
364                 std::cerr << "Error in function WaitForMultipleObjects in writer executor." << std
::endl;
365                 exit(0xA);
366             }
367         }
368
369         printf("Writer #%d finished.\n", threadId);
370         return NULL;
371     }

```

Результат решения задачи:

```
C:\Users\Desktop\p2.1.exe
----- Current config -----
Readers count: 4
Readers delay: 1
Writers count: 1
Writers delay: 1
Queue size: 10
Time to live: 1

Readers have been successfully created.
Writers have been successfully created.
Time manager has been successfully created.
Writer #0. Put data "Writer #0. Message number #1.".
Reader #0. Read message "Writer #0. Message number #1."
Reader #2. Read message "Writer #0. Message number #1."
Reader #1. Read message "Writer #0. Message number #1."
Reader #3. Read message "Writer #0. Message number #1."
Writer #1. Put data "Writer #1. Message number #1.".
Reader #2. Read message "Writer #1. Message number #1."
Reader #0. Read message "Writer #1. Message number #1."
Reader #3. Read message "Writer #1. Message number #1."
Reader #1. Read message "Writer #1. Message number #1."
Writer #0. Put data "Writer #0. Message number #2.".
Reader #0. Read message "Writer #0. Message number #2."
Reader #1. Read message "Writer #0. Message number #2."
Reader #3. Read message "Writer #0. Message number #2."
Reader #2. Read message "Writer #0. Message number #2."
Writer #3. Put data "Writer #3. Message number #1.".
Reader #0. Read message "Writer #3. Message number #1."
Reader #1. Read message "Writer #3. Message number #1."
Reader #2. Read message "Writer #3. Message number #1."
Reader #3. Read message "Writer #3. Message number #1."
Writer #2. Put data "Writer #2. Message number #1.".
Reader #0. Read message "Writer #2. Message number #1."
Reader #1. Read message "Writer #2. Message number #1."
Reader #3. Read message "Writer #2. Message number #1."
Reader #2. Read message "Writer #2. Message number #1."
Writer #1. Put data "Writer #1. Message number #2.".
Reader #0. Read message "Writer #1. Message number #2."
Reader #2. Read message "Writer #1. Message number #2."
Reader #3. Read message "Writer #1. Message number #2."
Reader #1. Read message "Writer #1. Message number #2."
```

Рис. 1.12

После того, как писатель положил значение в разделяемую память, началось ожидание чтения всеми читателями. Только после того, как все читатели прочитали данные генерируются следующие.

## 2. Использование разделяемой памяти и нескольких процессов для решения задачи

В данной программе главный поток и поток-писатель будут принадлежать одному процессу, а потоки-читатели разным. Каждому событию и разделяемой памяти теперь приписывается уникальное имя, для того чтобы к ним можно было обратиться из другого процесса.

Реализация процесса-писателя:

```
1 #include <stdio.h>
2 #include <windows.h>
3 #include <iostream>
4 #include <fstream>
5 #include <string>
6 #include <vector>
7
8 // Структура начальной конфигурации приложения
9 struct Configuration {
10     // Количество потоков читателей
11     int readersCount;
12     // Количество потоков писателей
13     int writersCount;
14     // Размер очереди
15     int queueSize;
16     // Задержка читателей в миллисекундах
17     int readersDelay;
```

```

18 // Задержка писателей в миллисекундах
19 int writersDelay;
20 // Время жизни приложения
21 int timeToLive;
22 };
23
24 static const char* CONFIG_PATH = "config.ini";
25 static const __int64 TIMER_CONSTANT = -1 * 10000000;
26 static const char* EXIT_MESSAGE = "q";
27 static const char* SHARE_MEMORY_NAME = "$$SharedMemory$$";
28 static const int BUFFER_SIZE = 1024;
29 static const int OFFSET = sizeof(int) * 2;
30 static constexpr char* READER_PROCESS_PATH = "p2.2.r.exe";
31
32 static const char* EVENT_CAN_READ_NAME = "$$EventCanRead$$";
33 static const char* EVENT_CAN_WRITE_NAME = "$$EventCanWrite$$";
34 static const char* EVENT_CHANGE_COUNT_NAME = "$$EventChangeCount$$";
35 static const char* EVENT_EXIT_NAME = "$$EventExit$$";
36 static const char* EVENT_ALL_READ_NAME = "$$EventAllRead$$";
37
38 // Глобальные структуры конфигурации и очереди
39 struct Configuration globalConfig;
40
41 // Массив, содержащий дескрипторы всех потоков
42 std::vector<HANDLE> handlers;
43 // Переменная, которая завершает все потоки
44 bool interrupt = false;
45
46 // События для синхронизации
47 HANDLE eventCanRead, eventCanWrite, eventChangeCount, eventExit, eventAllRead;
48 // Отображение файла
49 HANDLE fileMapping;
50 // Указатели на отображаемую память
51 LPVOID writersMap;
52
53 // Функция для заполнения структуры конфигурации
54 void setConfig(const std::string filename, Configuration* config);
55 // Получение параметра структуры конфигурации
56 void getConfigParam(int* parametr, std::istream& stream);
57 // Функция, конфигурирующая и запускающая таймер
58 HANDLE createAndStartWaitableTimer(const unsigned int secondsCount);
59
60 // Функция, создающая все потоки
61 void createAllThreads(Configuration* config);
62 // Обработчики потоков
63 DWORD WINAPI threadWriterExecutor(LPVOID argument);
64 DWORD WINAPI threadTimeManagerExecutor(LPVOID argument);
65
66 int main(int argc, char* argv[]) {
67     // Получение названия файла конфигурации
68     std::string configFilename;
69     configFilename = (argc < 2) ? CONFIG_PATH : argv[1];
70     setConfig(configFilename, &globalConfig);
71
72     // Создание всех потоков
73     createAllThreads(&globalConfig);
74
75     // Создание отображаемого файла
76     fileMapping = CreateFileMapping(INVALID_HANDLE_VALUE, nullptr, PAGE_READWRITE, NULL,
77     BUFFER_SIZE, SHARE_MEMORY_NAME);
78     if(!fileMapping) {
79         std::cerr << "It's impossible to create shared file." << std::endl;
80         return 0x7;
81     }
82     // Отображение файла на адресное пространство процесса

```



```

83 writersMap = MapViewOfFile(fileMapping, FILE_MAP_WRITE, NULL, NULL, NULL);
84 if(!writersMap) {
85     std::cerr << "It's impossible to view map." << std::endl;
86     return 0x8;
87 }
88
89 *((int*) writersMap) = 0;
90 *(((int*) writersMap) + 1) = globalConfig.readersCount;
91
92 // Инициализация событий для синхронизации
93 eventCanRead = CreateEvent(nullptr, true, false, EVENT_CAN_READ_NAME);
94 eventCanWrite = CreateEvent(nullptr, false, false, EVENT_CAN_WRITE_NAME);
95 eventAllRead = CreateEvent(nullptr, true, true, EVENT_ALL_READ_NAME);
96 eventChangeCount = CreateEvent(nullptr, false, true, EVENT_CHANGE_COUNT_NAME);
97 eventExit = CreateEvent(nullptr, true, false, EVENT_EXIT_NAME);
98 if(!eventCanRead || !eventCanWrite || !eventAllRead || !eventChangeCount || !eventExit)
99 {
100     std::cerr << "It's impossible to create event." << std::endl;
101     return 0x9;
102 }
103
104 // Старт всех потоков
105 for(auto& current : handlers)
106     ResumeThread(current);
107
108 // Ожидание завершения всех потоков
109 WaitForMultipleObjects(handlers.size(), &(handlers[0]), TRUE, INFINITE);
110
111 // Закрываем дескрипторы потоков
112 for(auto& current : handlers)
113     CloseHandle(current);
114
115 // Закрываем описатели объектов синхронизации
116 CloseHandle(eventCanRead);
117 CloseHandle(eventCanWrite);
118 CloseHandle(eventAllRead);
119 CloseHandle(eventChangeCount);
120 CloseHandle(eventExit);
121
122 // Удаляем разделяемую память
123 UnmapViewOfFile(writersMap);
124
125 CloseHandle(fileMapping);
126
127 std::cout << "Press \"Enter\" to exit." << std::endl;
128 std::getchar();
129
130 return 0x0;
131 }
132
133 void getConfigParam(int* parametr, std::istream& stream) {
134     // Получение параметра структуры конфигурации
135     std::string line;
136
137     if(stream.eof())
138         throw std::exception();
139
140     std::getline(stream, line);
141     *parametr = std::stoi(line);
142 }
143
144 void setConfig(const std::string configFilename, Configuration* config) //функция установки
    конфигурации
145 {
146     // Открываем конфигурационный файл

```

```

147 auto stream = std::ifstream(configFilename.data());
148 if(!stream.is_open()) {
149     std::cerr << "It's impossible to open config file." << std::endl;
150     exit(0x1);
151 }
152
153 // Заполняем структуру конфигурации
154 std::string line;
155 try {
156     getConfigParam(&(config->readersCount), stream);
157     getConfigParam(&(config->readersDelay), stream);
158     getConfigParam(&(config->writersCount), stream);
159     getConfigParam(&(config->writersDelay), stream);
160     getConfigParam(&(config->queueSize), stream);
161     getConfigParam(&(config->timeToLive), stream);
162 }
163 catch(const std::exception& exception) {
164     std::cerr << "It's impossible to parse config file." << std::endl;
165     stream.close();
166     exit(0x2);
167 }
168
169 // Проверка корректности полученных данных
170 if(
171     config->readersCount <= 0 || config->readersDelay <= 0 ||
172     config->writersCount <= 0 || config->writersDelay <= 0 ||
173     config->queueSize <= 0 || config->timeToLive == 0
174 ) {
175     std::cerr << "Wrong config file values." << std::endl;
176     stream.close();
177     exit(0x3);
178 }
179
180 stream.close();
181
182 std::cout << "—— Current config ——" << std::endl
183 << "Readers count: " << config->readersCount << std::endl
184 << "Readers delay: " << config->readersDelay << std::endl
185 << "Writers count: " << config->writersCount << std::endl
186 << "Writers delay: " << config->writersDelay << std::endl
187 << "Queue size: " << config->queueSize << std::endl
188 << "Time to live: " << config->timeToLive << std::endl << std::endl;
189 }
190
191 void createAllThreads(Configuration * config) {
192     STARTUPINFO startupInformation;
193     ZeroMemory(&startupInformation, sizeof(startupInformation));
194     startupInformation.cb = sizeof(startupInformation);
195
196     PROCESS_INFORMATION processInformation;
197
198     std::string readerPath;
199     // Создание всех процессов читателей
200     for(int readerIndex = 0; readerIndex < config->readersCount; ++readerIndex) {
201         readerPath = std::string(READER_PROCESS_PATH) + " " + std::to_string(readerIndex);
202
203         BOOL reader = CreateProcess(nullptr, _strdup(readerPath.data()), nullptr, nullptr,
204             FALSE, CREATE_NEW_CONSOLE, nullptr, nullptr, &startupInformation, &processInformation);
205         if(!reader) {
206             std::cerr << "It's impossible to create reader." << std::endl;
207             exit(0x5);
208         }
209     }
210
211     std::cout << "Readers have been successfully created." << std::endl;

```

```

211
212
213 // Создание всех потоков писателей
214 for(int writerIndex = 0; writerIndex < config->writersCount; ++writerIndex) {
215     HANDLE writer = CreateThread(nullptr, NULL, threadWriterExecutor, LPVOID(writerIndex)
216         , CREATE_SUSPENDED, nullptr);
217     if(!writer) {
218         std::cerr << "It's impossible to create writer." << std::endl;
219         exit(0x5);
220     }
221     handlers.push_back(writer);
222 }
223
224 std::cout << "Writers have been successfully created." << std::endl;
225
226
227 // Создание временного обработчика
228 HANDLE timeManager = CreateThread(nullptr, NULL, threadTimeManagerExecutor, LPVOID(
229     config->timeToLive), CREATE_SUSPENDED, nullptr);
230 if(!timeManager) {
231     std::cerr << "It's impossible to create time manager." << std::endl;
232     exit(0x6);
233 }
234 handlers.push_back(timeManager);
235
236 std::cout << "Time manager has been successfully created." << std::endl;
237 }
238
239 HANDLE createAndStartWaitableTimer(const unsigned int secondsCount) //создание, установка и
    запуск таймера
240 {
241     // Вычисляем временные характеристики таймера
242     __int64 endTimeValue = TIMER_CONSTANT * secondsCount;
243     LARGE_INTEGER endTimeStruct;
244     endTimeStruct.LowPart = DWORD(endTimeValue & 0xFFFFFFFF);
245     endTimeStruct.HighPart = LONG(endTimeValue >> 0x20);
246
247     // Создаем таймер
248     HANDLE timer = CreateWaitableTimer(nullptr, false, nullptr);
249     SetWaitableTimer(timer, &endTimeStruct, NULL, nullptr, nullptr, false);
250     return timer;
251 }
252
253 DWORD WINAPI threadTimeManagerExecutor(LPVOID argument) {
254     int timeToLive = int(argument);
255     std::string message;
256
257     // Если время жизни отрицательное, то выходим по введенной строке
258     if(timeToLive < 0)
259         while(true) {
260             message.clear();
261             std::getline(std::cin, message);
262
263             if(message == EXIT_MESSAGE) {
264                 interrupt = true;
265                 break;
266             }
267         }
268     // Если время жизни положительное, то выходим по таймеру
269     else {
270         HANDLE timer = createAndStartWaitableTimer(timeToLive);
271         WaitForSingleObject(timer, INFINITE);
272         SetEvent(eventExit);
273         interrupt = true;

```

```

274     CloseHandle(timer);
275 }
276
277 printf("Time manager finished.\n");
278 return NULL;
279 }
280
281 DWORD WINAPI threadWriterExecutor(LPVOID argument) {
282     int threadId = int(argument);
283
284     int messageIndex = 0;
285
286     // Задаем массив событий для функции ожидания
287     HANDLE writerHandlers[2];
288     writerHandlers[0] = eventExit;
289     writerHandlers[1] = eventCanWrite;
290
291     while(!interrupt) {
292         DWORD analyzeEvent = WaitForMultipleObjects(2, writerHandlers, false, INFINITE);
293         switch(analyzeEvent) {
294             case WAIT_OBJECT_0:
295                 // Событие завершения потока
296                 printf("Writer #%d finished.\n", threadId);
297                 return NULL;
298
299             case (WAIT_OBJECT_0 + 1):
300                 // Событие записи
301                 ++messageIndex;
302
303                 // Запись в разделяемую память
304                 sprintf_s(((char *) writersMap) + OFFSET, BUFFER_SIZE, "Writer #%d. Message number
305                               #%d.", threadId, messageIndex);
306
307                 printf("Writer #%d. Put data \"%s\".\n", threadId, ((char *) writersMap) + OFFSET);
308
309                 WaitForSingleObject(eventChangeCount, INFINITE);
310
311                 *(((int *) writersMap) += globalConfig.readersCount;
312                 *(((int *) writersMap) + 1) += globalConfig.readersCount;
313
314                 SetEvent(eventChangeCount);
315
316                 // Разрешаем читателям прочитать сообщение и ставим событие в занятое
317                 SetEvent(eventCanRead);
318                 break;
319
320             default:
321                 std::cerr << "Error in function WaitForMultipleObjects in writer executor." << std
322                               ::endl;
323                 exit(0xA);
324         }
325     }
326
327     printf("Writer #%d finished.\n", threadId);
328     return NULL;
329 }

```

Реализация процесса-читателя:

```

1 #include <stdio.h>
2 #include <iostream>
3 #include <windows.h>
4 #include <string>
5
6 static const int OFFSET = sizeof(int) * 2;
7
8 static const char* SHARE_MEMORY_NAME = "$$SharedMemory$$";

```

```

9 // Названия событий
10 static const char* EVENT_CAN_READ_NAME = "$EventCanRead$";
11 static const char* EVENT_CAN_WRITE_NAME = "$EventCanWrite$";
12 static const char* EVENT_CHANGE_COUNT_NAME = "$EventChangeCount$";
13 static const char* EVENT_EXIT_NAME = "$EventExit$";
14 static const char* EVENT_ALL_READ_NAME = "$EventAllRead$";
15
16 // События для синхронизации
17 HANDLE eventCanRead, eventCanWrite, eventChangeCount, eventExit, eventAllRead;
18
19 int main(int argc, char* argv[]) {
20     if(argc != 2) {
21         std::cerr << "Wrong count of arguments." << std::endl;
22         return 0x1;
23     }
24
25     // Получаем идентификатор из аргумента командной строки
26     int readerId = std::stoi(argv[1]);
27
28     std::cout << "Reader #" << readerId << " started." << std::endl;
29
30     // Инициализация событий для синхронизации
31     eventCanRead = OpenEvent(EVENT_ALL_ACCESS, false, EVENT_CAN_READ_NAME);
32     eventCanWrite = OpenEvent(EVENT_ALL_ACCESS, false, EVENT_CAN_WRITE_NAME);
33     eventAllRead = OpenEvent(EVENT_ALL_ACCESS, false, EVENT_ALL_READ_NAME);
34     eventChangeCount = OpenEvent(EVENT_ALL_ACCESS, false, EVENT_CHANGE_COUNT_NAME);
35     eventExit = OpenEvent(EVENT_ALL_ACCESS, false, EVENT_EXIT_NAME);
36     if(!eventCanRead || !eventCanWrite || !eventAllRead || !eventChangeCount || !eventExit)
37     {
38         std::cerr << "It's impossible to open event." << std::endl;
39         return 0x2;
40     }
41
42     // Открытие отображаемого файла
43     HANDLE fileMapping = OpenFileMapping(FILE_MAP_ALL_ACCESS, false, SHARE_MEMORY_NAME);
44     if(!fileMapping) {
45         std::cerr << "It's impossible to open shared file." << std::endl;
46         return 0x3;
47     }
48
49     LPVOID readersMap = MapViewOfFile(fileMapping, FILE_MAP_ALL_ACCESS, NULL, NULL, NULL);
50     if(!readersMap) {
51         std::cerr << "It's impossible to view map." << std::endl;
52         return 0x4;
53     }
54
55     // Задаем массив событий для функции ожидания
56     HANDLE readerHandlers[2];
57     readerHandlers[0] = eventExit;
58     readerHandlers[1] = eventCanRead;
59
60     while(true) {
61         // Ожидаем, пока все прочитают
62         WaitForSingleObject(eventAllRead, INFINITE);
63         // Узнаем сколько потоков читателей прошло данную границу
64         WaitForSingleObject(eventChangeCount, INFINITE);
65
66         — *(((int *) readersMap) + 1);
67         if(*(((int *) readersMap) + 1) == 0) {
68             // Если все прошли то закрываем событие
69             ResetEvent(eventAllRead);
70             // Разрешаем писать
71             SetEvent(eventCanWrite);
72         }
73         SetEvent(eventChangeCount);

```

```

74     DWORD analyzeEvent = WaitForMultipleObjects(2, readerHandlers, false, INFINITE);
75     switch(analyzeEvent) {
76     case WAIT_OBJECT_0:
77         // Событие завершения потока
78         printf("Reader #%d finished.\n", readerId);
79         goto exit;
80
81     case (WAIT_OBJECT_0 + 1):
82         // Событие чтения
83         printf("Reader #%d. Get data \"%s\".\n", readerId, ((char *) readersMap) + OFFSET);
84
85         // Ожидание уменьшение счетчика количества читателей
86         WaitForSingleObject(eventChangeCount, INFINITE);
87
88         // Уменьшаем счетчик количества читателей
89         — *((int *) readersMap);
90         if (*((int *) readersMap) == 0) {
91             // Запрещаем читать, если мы прочитали последние
92             ResetEvent(eventCanRead);
93             // Открываем границу
94             SetEvent(eventAllRead);
95         }
96
97         SetEvent(eventChangeCount);
98         break;
99
100    default:
101        std::cerr << "Error in function WaitForMultipleObjects in reader executor." << std
102        ::endl;
103        exit(0x5);
104    }
105
106    exit:
107    // Закрываем описатели объектов синхронизации
108    CloseHandle(eventCanRead);
109    CloseHandle(eventCanWrite);
110    CloseHandle(eventAllRead);
111    CloseHandle(eventChangeCount);
112    CloseHandle(eventExit);
113
114    // Удаляем разделяемую память
115    UnmapViewOfFile(readersMap);
116
117    CloseHandle(fileMapping);
118
119    std::cout << "Press \"Enter\" to exit." << std::endl;
120    std::getchar();
121
122    return 0x0;
123 }

```

Результат работы для трех потоков писателей и трех процессов читателей:

```

C:\Users\Desktop\p2.2.w.exe
----- Current config -----
Readers count: 3
Readers delay: 1
Writers count: 2
Writers delay: 1
Queue size: 10
Time to live: 1

Readers have been successfully created.
Writers have been successfully created.
Time manager has been successfully created.
Writer #0. Put data "Writer #0. Message number #1.".
Writer #2. Put data "Writer #2. Message number #1.".
Writer #1. Put data "Writer #1. Message number #1.".
Writer #0. Put data "Writer #0. Message number #2.".
Writer #2. Put data "Writer #2. Message number #2.".
Writer #1. Put data "Writer #1. Message number #2.".
Writer #0. Put data "Writer #0. Message number #3.".
Writer #2. Put data "Writer #2. Message number #3.".
Writer #1. Put data "Writer #1. Message number #3.".
Writer #0. Put data "Writer #0. Message number #4.".
Writer #2. Put data "Writer #2. Message number #4.".
Writer #1. Put data "Writer #1. Message number #4.".
Writer #0. Put data "Writer #0. Message number #5.".

C:\Users\Desktop\p2.2.r.exe
Reader #1 started.
Reader #1. Get data "Writer #0. Message number #1.".
Reader #1. Get data "Writer #2. Message number #1.".
Reader #1. Get data "Writer #1. Message number #1.".
Reader #1. Get data "Writer #0. Message number #2.".
Reader #1. Get data "Writer #2. Message number #2.".
Reader #1. Get data "Writer #1. Message number #2.".
Reader #1. Get data "Writer #0. Message number #3.".
Reader #1. Get data "Writer #2. Message number #3.".
Reader #1. Get data "Writer #1. Message number #3.".
Reader #1. Get data "Writer #0. Message number #4.".
Reader #1. Get data "Writer #2. Message number #4.".
Reader #1. Get data "Writer #1. Message number #4.".
Reader #1. Get data "Writer #0. Message number #5.".
Reader #1. Get data "Writer #2. Message number #5.".
Reader #1. Get data "Writer #1. Message number #5.".
Reader #1. Get data "Writer #0. Message number #6.".
Reader #1. Get data "Writer #2. Message number #6.".
Reader #1. Get data "Writer #1. Message number #6.".
Reader #1. Get data "Writer #0. Message number #7.".
Reader #1. Get data "Writer #2. Message number #7.".
Reader #1. Get data "Writer #1. Message number #7.".
Reader #1. Get data "Writer #0. Message number #8.".
Reader #1. Get data "Writer #2. Message number #8.".
Reader #1. Get data "Writer #1. Message number #8.".
Reader #1. Get data "Writer #0. Message number #9.".

C:\Users\Desktop\p2.2.r.exe
Reader #2 started.
Reader #2. Get data "Writer #0. Message number #1.".
Reader #2. Get data "Writer #2. Message number #1.".
Reader #2. Get data "Writer #1. Message number #1.".
Reader #2. Get data "Writer #0. Message number #2.".
Reader #2. Get data "Writer #2. Message number #2.".
Reader #2. Get data "Writer #1. Message number #2.".
Reader #2. Get data "Writer #0. Message number #3.".
Reader #2. Get data "Writer #2. Message number #3.".
Reader #2. Get data "Writer #1. Message number #3.".
Reader #2. Get data "Writer #0. Message number #4.".
Reader #2. Get data "Writer #2. Message number #4.".
Reader #2. Get data "Writer #1. Message number #4.".
Reader #2. Get data "Writer #0. Message number #5.".
Reader #2. Get data "Writer #2. Message number #5.".
Reader #2. Get data "Writer #1. Message number #5.".
Reader #2. Get data "Writer #0. Message number #6.".
Reader #2. Get data "Writer #2. Message number #6.".
Reader #2. Get data "Writer #1. Message number #6.".
Reader #2. Get data "Writer #0. Message number #7.".
Reader #2. Get data "Writer #2. Message number #7.".
Reader #2. Get data "Writer #1. Message number #7.".
Reader #2. Get data "Writer #0. Message number #8.".
Reader #2. Get data "Writer #2. Message number #8.".
Reader #2. Get data "Writer #1. Message number #8.".
Reader #2. Get data "Writer #0. Message number #9.".

Выбрать C:\Users\Desktop\p2.2.r.exe

```

Рис. 1.13

Главный поток ждет завершение всех главных потоков. Т.к. потоки-читатели, принадлежащие разным процессам, имеют каждый свою консоль, то после окончания работы они сразу не завершаются, а ждут ввода любой клавиши. Каждый поток-читатель успевает читать каждое сообщение.

### 3. Модификация программы для того, чтобы читатели не имели доступа к памяти по записи

Для этого будем хранить счетчики в процессе писателе. Потоки-читатели вместо декремента счетчика производят освобождение событий, по которым писатель декрементирует счетчик.

Когда сообщение в память записано потоком-читателем, каждый читатель получает сообщение и устанавливает событие eventCount, по которому поток-писатель декрементирует значение счетчика. Когда значение этого счетчика становится равным нулю, устанавливается событие – все прочитали сообщение.

Реализация процесса-писателя:

```

1 #include <stdio.h>
2 #include <windows.h>
3 #include <iostream>
4 #include <fstream>

```

```

5 #include <string>
6 #include <vector>
7
8 // Структура начальной конфигурации приложения
9 struct Configuration {
10     // Количество потоков читателей
11     int readersCount;
12     // Количество потоков писателей
13     int writersCount;
14     // Размер очереди
15     int queueSize;
16     // Задержка читателей в миллисекундах
17     int readersDelay;
18     // Задержка писателей в миллисекундах
19     int writersDelay;
20     // Время жизни приложения
21     int timeToLive;
22 };
23
24 static const char* CONFIG_PATH = "config.ini";
25 static const __int64 TIMER_CONSTANT = -1 * 10000000;
26 static const char* EXIT_MESSAGE = "q";
27 static const char* SHARE_MEMORY_NAME = "$$SharedMemory$$";
28 static const int BUFFER_SIZE = 1024;
29 static const int OFFSET = sizeof(int) * 2;
30 static constexpr char* READER_PROCESS_PATH = "p2.3.r.exe";
31
32 static const char* EVENT_CAN_READ_NAME = "$$EventCanRead$$";
33 static const char* EVENT_CAN_WRITE_NAME = "$$EventCanWrite$$";
34 static const char* EVENT_CHANGE_COUNT_NAME = "$$EventChangeCount$$";
35 static const char* EVENT_COUNT_NAME = "$$EventCount$$";
36 static const char* EVENT_EXIT_NAME = "$$EventExit$$";
37 static const char* EVENT_ALL_READ_NAME = "$$EventAllRead$$";
38
39 // Глобальные структуры конфигурации и очереди
40 struct Configuration globalConfig;
41
42 // Массив, содержащий дескрипторы всех потоков
43 std::vector<HANDLE> handlers;
44 // Переменная, которая завершает все потоки
45 bool interrupt = false;
46
47 // События для синхронизации
48 HANDLE eventCanRead, eventCanWrite, eventChangeCount, eventExit, eventAllRead, eventCount;
49
50 int readersDontReadyToRead, readersAlreadyRead;
51 // Отображение файла
52 HANDLE fileMapping;
53 // Указатели на отображаемую память
54 LPVOID writersMap;
55
56 // Функция для заполнения структуры конфигурации
57 void setConfig(const std::string filename, Configuration* config);
58 // Получение параметра структуры конфигурации
59 void getConfigParam(int* parametr, std::istream& stream);
60 // Функция, конфигурирующая и запускающая таймер
61 HANDLE createAndStartWaitableTimer(const unsigned int secondsCount);
62
63 // Функция, создающая все потоки
64 void createAllThreads(Configuration* config);
65 // Обработчики потоков
66 DWORD WINAPI threadWriterExecutor(LPVOID argument);
67 DWORD WINAPI threadTimeManagerExecutor(LPVOID argument);
68
69 int main(int argc, char* argv[]) {
70     // Получение названия файла конфигурации

```



```

70 std::string configFilename;
71 configFilename = (argc < 2) ? CONFIG_PATH : argv[1];
72 setConfig(configFilename, &globalConfig);
73
74 // Создание всех потоков
75 createAllThreads(&globalConfig);
76
77 // Создание отображаемого файла
78 fileMapping = CreateFileMapping(INVALID_HANDLE_VALUE, nullptr, PAGE_READWRITE, NULL,
79 BUFFER_SIZE, SHARE_MEMORY_NAME);
80 if(!fileMapping) {
81     std::cerr << "It's impossible to create shared file." << std::endl;
82     return 0x7;
83 }
84
85 // Отображение файла на адресное пространство процесса
86 writersMap = MapViewOfFile(fileMapping, FILE_MAP_WRITE, NULL, NULL, NULL);
87 if(!writersMap) {
88     std::cerr << "It's impossible to view map." << std::endl;
89     return 0x8;
90 }
91
92 *((int*) writersMap) = 0;
93 *(((int*) writersMap) + 1) = globalConfig.readersCount;
94
95 // Инициализация событий для синхронизации
96 eventCanRead = CreateEvent(nullptr, true, false, EVENT_CAN_READ_NAME);
97 eventCanWrite = CreateEvent(nullptr, false, false, EVENT_CAN_WRITE_NAME);
98 eventAllRead = CreateEvent(nullptr, true, true, EVENT_ALL_READ_NAME);
99 eventChangeCount = CreateEvent(nullptr, false, true, EVENT_CHANGE_COUNT_NAME);
100 eventExit = CreateEvent(nullptr, true, false, EVENT_EXIT_NAME);
101 eventCount = CreateEvent(nullptr, false, false, EVENT_COUNT_NAME);
102 if(!eventCanRead || !eventCanWrite || !eventAllRead || !eventChangeCount || !eventExit
103 || !eventCount) {
104     std::cerr << "It's impossible to create event." << std::endl;
105     return 0x9;
106 }
107
108 // Старт всех потоков
109 for(auto& current : handlers)
110     ResumeThread(current);
111
112 // Ожидание завершения всех потоков
113 WaitForMultipleObjects(handlers.size(), &(handlers[0]), TRUE, INFINITE);
114
115 // Закрываем дескрипторы потоков
116 for(auto& current : handlers)
117     CloseHandle(current);
118
119 // Закрываем описатели объектов синхронизации
120 CloseHandle(eventCanRead);
121 CloseHandle(eventCanWrite);
122 CloseHandle(eventAllRead);
123 CloseHandle(eventChangeCount);
124 CloseHandle(eventExit);
125 CloseHandle(eventCount);
126
127 // Удаляем разделяемую память
128 UnmapViewOfFile(writersMap);
129
130 CloseHandle(fileMapping);
131
132 std::cout << "Press \"Enter\" to exit." << std::endl;
133 std::getchar();
134
135 return 0x0;

```

```

134 }
135
136 void getConfigParam(int* parametr, std::istream& stream) {
137     // Получение параметра структуры конфигурации
138
139     std::string line;
140
141     if(stream.eof())
142         throw std::exception();
143
144     std::getline(stream, line);
145     *parametr = std::stoi(line);
146 }
147
148 void setConfig(const std::string configFilename, Configuration* config) //функция установки
    конфигурации
149 {
150     // Открываем конфигурационный файл
151     auto stream = std::ifstream(configFilename.data());
152     if(!stream.is_open()) {
153         std::cerr << "It's impossible to open config file." << std::endl;
154         exit(0x1);
155     }
156
157     // Заполняем структуру конфигурации
158     std::string line;
159     try {
160         getConfigParam(&(config->readersCount), stream);
161         getConfigParam(&(config->readersDelay), stream);
162         getConfigParam(&(config->writersCount), stream);
163         getConfigParam(&(config->writersDelay), stream);
164         getConfigParam(&(config->queueSize), stream);
165         getConfigParam(&(config->timeToLive), stream);
166     }
167     catch(const std::exception& exception) {
168         std::cerr << "It's impossible to parse config file." << std::endl;
169         stream.close();
170         exit(0x2);
171     }
172
173     // Проверка корректности полученных данных
174     if(
175         config->readersCount <= 0 || config->readersDelay <= 0 ||
176         config->writersCount <= 0 || config->writersDelay <= 0 ||
177         config->queueSize <= 0 || config->timeToLive == 0
178     ) {
179         std::cerr << "Wrong config file values." << std::endl;
180         stream.close();
181         exit(0x3);
182     }
183
184     stream.close();
185
186     std::cout << "——— Current config ———" << std::endl
187         << "Readers count: " << config->readersCount << std::endl
188         << "Readers delay: " << config->readersDelay << std::endl
189         << "Writers count: " << config->writersCount << std::endl
190         << "Writers delay: " << config->writersDelay << std::endl
191         << "Queue size: " << config->queueSize << std::endl
192         << "Time to live: " << config->timeToLive << std::endl << std::endl;
193 }
194
195 void createAllThreads(Configuration * config) {
196     STARTUPINFO startupInformation;
197     ZeroMemory(&startupInformation, sizeof(startupInformation));
198     startupInformation.cb = sizeof(startupInformation);

```

```

199 PROCESS_INFORMATION processInformation;
200
201
202 std::string readerPath;
203 // Создание всех процессов читателей
204 for(int readerIndex = 0; readerIndex < config->readersCount; ++readerIndex) {
205     readerPath = std::string(READER_PROCESS_PATH) + " " + std::to_string(readerIndex);
206
207     BOOL reader = CreateProcess(nullptr, _strdup(readerPath.data()), nullptr, nullptr,
208     FALSE, CREATE_NEW_CONSOLE | CREATE_SUSPENDED, nullptr, nullptr, &startupInformation, &
209     processInformation);
210     if(!reader) {
211         std::cerr << "It's impossible to create reader." << std::endl;
212         exit(0x5);
213     }
214     handlers.push_back(processInformation.hThread);
215 }
216
217 std::cout << "Readers have been successfully created." << std::endl;
218
219 // Создание всех потоков писателей
220 for(int writerIndex = 0; writerIndex < config->readersCount; ++writerIndex) {
221     HANDLE writer = CreateThread(nullptr, NULL, threadWriterExecutor, LPVOID(writerIndex)
222     , CREATE_SUSPENDED, nullptr);
223     if(!writer) {
224         std::cerr << "It's impossible to create reader." << std::endl;
225         exit(0x5);
226     }
227     handlers.push_back(writer);
228 }
229
230 std::cout << "Writers have been successfully created." << std::endl;
231
232
233 // Создание временного обработчика
234 HANDLE timeManager = CreateThread(nullptr, NULL, threadTimeManagerExecutor, LPVOID(
235     config->timeToLive), CREATE_SUSPENDED, nullptr);
236 if(!timeManager) {
237     std::cerr << "It's impossible to create time manager." << std::endl;
238     exit(0x6);
239 }
240
241 handlers.push_back(timeManager);
242
243 std::cout << "Time manager has been successfully created." << std::endl;
244 }
245
246 HANDLE createAndStartWaitableTimer(const unsigned int secondsCount) //создание, установка и
247     запуск таймера
248 {
249     // Вычисляем временные характеристики таймера
250     _int64 endTimeValue = TIMER_CONSTANT * secondsCount;
251     LARGE_INTEGER endTimeStruct;
252     endTimeStruct.LowPart = DWORD(endTimeValue & 0xFFFFFFFF);
253     endTimeStruct.HighPart = LONG(endTimeValue >> 0x20);
254
255     // Создаем таймер
256     HANDLE timer = CreateWaitableTimer(nullptr, false, nullptr);
257     SetWaitableTimer(timer, &endTimeStruct, NULL, nullptr, nullptr, false);
258     return timer;
259 }
260
261 DWORD WINAPI threadTimeManagerExecutor(LPVOID argument) {

```

```

260     int timeToLive = int(argument);
261     std::string message;
262
263     // Если время жизни отрицательное, то выходим по введенной строке
264     if(timeToLive < 0)
265         while(true) {
266             message.clear();
267             std::getline(std::cin, message);
268
269             if(message == EXIT_MESSAGE) {
270                 interrupt = true;
271                 break;
272             }
273         }
274     // Если время жизни положительное, то выходим по таймеру
275     else {
276         HANDLE timer = createAndStartWaitableTimer(timeToLive);
277         WaitForSingleObject(timer, INFINITE);
278         SetEvent(eventExit);
279         interrupt = true;
280         CloseHandle(timer);
281     }
282
283     printf("Time manager finished.\n");
284     return NULL;
285 }
286
287 DWORD WINAPI threadWriterExecutor(LPVOID argument) {
288     int threadId = int(argument);
289
290     int messageIndex = 0;
291
292     // Задаем массив событий для функции ожидания
293     HANDLE writerHandlers[2];
294     writerHandlers[0] = eventExit;
295     writerHandlers[1] = eventCanWrite;
296
297     while(!interrupt) {
298         readersDontReadyToRead = globalConfig.readersCount;
299         readersAlreadyRead = globalConfig.readersCount;
300
301         while(true) {
302             WaitForSingleObject(eventCount, INFINITE);
303
304             —readersDontReadyToRead;
305             if(readersDontReadyToRead == 0) {
306                 ResetEvent(eventAllRead);
307                 SetEvent(eventCanWrite);
308                 ResetEvent(eventCount);
309                 SetEvent(eventChangeCount);
310                 break;
311             }
312
313             ResetEvent(eventCount);
314             SetEvent(eventChangeCount);
315         }
316
317         DWORD analyzeEvent = WaitForMultipleObjects(2, writerHandlers, false, INFINITE);
318         switch(analyzeEvent) {
319             case WAIT_OBJECT_0:
320                 // Событие завершения потока
321                 printf("Writer #%d finished.\n", threadId);
322                 return NULL;
323
324             case (WAIT_OBJECT_0 + 1):
325                 // Событие записи

```

```

326     ++messageIndex;
327
328     // Запись в разделяемую память
329     sprintf_s(((char *) writersMap) + OFFSET, BUFFER_SIZE, "Writer #%d. Message number
330     #%d.", threadId, messageIndex);
331
332     printf("Writer #%d. Put data \"%s\".\n", threadId, (((char *) writersMap) + OFFSET));
333
334     WaitForSingleObject(eventChangeCount, INFINITE);
335
336     SetEvent(eventChangeCount);
337
338     // Разрешаем читателям прочитать сообщение и ставим событие в занятое
339     SetEvent(eventCanRead);
340     break;
341
342 default:
343     std::cerr << "Error in function WaitForMultipleObjects in writer executor." << std
344     ::endl;
345     exit(0xA);
346 }
347
348 while(true) {
349     WaitForSingleObject(eventCount, INFINITE);
350
351     —readersAlreadyRead;
352     if(readersAlreadyRead == 0) {
353         // Если последнее чтение, то запрещаем читать
354         ResetEvent(eventCanRead);
355         SetEvent(eventAllRead);
356         ResetEvent(eventCount);
357         SetEvent(eventChangeCount);
358         break;
359     }
360
361     ResetEvent(eventCount);
362     SetEvent(eventChangeCount);
363 }
364
365 Sleep(globalConfig.writersDelay);
366 }
367
368 printf("Writer #%d finished.\n", threadId);
369 return NULL;
370 }

```

Реализация процесса-читателя:

```

1 #include <stdio.h>
2 #include <iostream>
3 #include <windows.h>
4 #include <string>
5
6 static const int OFFSET = sizeof(int) * 2;
7 static const int READ_DELAY = 200;
8
9 static const char* SHARE_MEMORY_NAME = "$$SharedMemory$$";
10 // Названия событий
11 static const char* EVENT_CAN_READ_NAME = "$$EventCanRead$$";
12 static const char* EVENT_CAN_WRITE_NAME = "$$EventCanWrite$$";
13 static const char* EVENT_CHANGE_COUNT_NAME = "$$EventChangeCount$$";
14 static const char* EVENT_COUNT_NAME = "$$EventCount$$";
15 static const char* EVENT_EXIT_NAME = "$$EventExit$$";
16 static const char* EVENT_ALL_READ_NAME = "$$EventAllRead$$";
17
18 // События для синхронизации
19 HANDLE eventCanRead, eventCanWrite, eventChangeCount, eventExit, eventAllRead, eventCount

```

```

20 ;
21 int main(int argc, char* argv[]) {
22     if(argc != 2) {
23         std::cerr << "Wrong count of arguments." << std::endl;
24         return 0x1;
25     }
26
27     // Получаем идентификатор из аргумента командной строки
28     int readerId = std::stoi(argv[1]);
29
30     std::cout << "Reader #" << readerId << " started." << std::endl;
31
32     CRITICAL_SECTION criticalSection;
33     InitializeCriticalSection(&criticalSection);
34
35     // Инициализация событий для синхронизации
36     eventCanRead = OpenEvent(EVENT_ALL_ACCESS, false, EVENT_CAN_READ_NAME);
37     eventCanWrite = OpenEvent(EVENT_ALL_ACCESS, false, EVENT_CAN_WRITE_NAME);
38     eventAllRead = OpenEvent(EVENT_ALL_ACCESS, false, EVENT_ALL_READ_NAME);
39     eventChangeCount = OpenEvent(EVENT_ALL_ACCESS, false, EVENT_CHANGE_COUNT_NAME);
40     eventExit = OpenEvent(EVENT_ALL_ACCESS, false, EVENT_EXIT_NAME);
41     eventCount = OpenEvent(EVENT_ALL_ACCESS, false, EVENT_COUNT_NAME);
42     if(!eventCanRead || !eventCanWrite || !eventAllRead || !eventChangeCount || !eventExit
43         || !eventCount) {
44         std::cerr << "It's impossible to open event." << std::endl;
45         return 0x2;
46     }
47
48     // Открытие отображаемого файла
49     HANDLE fileMapping = OpenFileMapping(FILE_MAP_ALL_ACCESS, false, SHARE_MEMORY_NAME);
50     if(!fileMapping) {
51         std::cerr << "It's impossible to open shared file." << std::endl;
52         return 0x3;
53     }
54
55     LPVOID readersMap = MapViewOfFile(fileMapping, FILE_MAP_READ, NULL, NULL, NULL);
56     if(!readersMap) {
57         std::cerr << "It's impossible to view map." << std::endl;
58         return 0x4;
59     }
60
61     // Задаем массив событий для функции ожидания
62     HANDLE readerHandlers[2];
63     readerHandlers[0] = eventExit;
64     readerHandlers[1] = eventCanRead;
65
66     while(true) {
67         EnterCriticalSection(&criticalSection);
68         // Ожидаем, пока все прочитают
69         WaitForSingleObject(eventAllRead, INFINITE);
70         // Узнаем сколько потоков читателей прошло данную границу
71         WaitForSingleObject(eventChangeCount, INFINITE);
72
73         SetEvent(eventCount);
74
75         DWORD analyzeEvent = WaitForMultipleObjects(2, readerHandlers, false, INFINITE);
76         switch(analyzeEvent) {
77             case WAIT_OBJECT_0:
78                 // Событие завершения потока
79                 printf("Reader #%d finished.\n", readerId);
80                 goto exit;
81
82             case (WAIT_OBJECT_0 + 1):
83                 // Событие чтения
84                 printf("Reader #%d. Get data \"%s\".\n", readerId, ((char *) readersMap) + OFFSET);

```

```

84
85     // Ожидание уменьшение счетчика количества читателей
86     WaitForSingleObject(eventChangeCount, INFINITE);
87     SetEvent(eventCount);
88     break;
89
90     default:
91         std::cerr << "Error in function WaitForMultipleObjects in reader executor." << std
92         ::endl;
93         exit(0x5);
94     }
95
96     LeaveCriticalSection(&criticalSection);
97     Sleep(READ_DELAY);
98 }
99
100 exit:
101     DeleteCriticalSection(&criticalSection);
102
103     // Закрываем описатели объектов синхронизации
104     CloseHandle(eventCanRead);
105     CloseHandle(eventCanWrite);
106     CloseHandle(eventAllRead);
107     CloseHandle(eventChangeCount);
108     CloseHandle(eventExit);
109     CloseHandle(eventCount);
110
111     // Удаляем разделяемую память
112     UnmapViewOfFile(readersMap);
113
114     CloseHandle(fileMapping);
115
116     std::cout << "Press \"Enter\" to exit." << std::endl;
117     std::getchar();
118
119     return 0x0;
120 }

```

Результат работы для трех потоков писателей и трех процессов читателей:

C:\Users\Desktop\p2.2.w.exe

```

----- Current config -----
Readers count: 3
Readers delay: 1
Writers count: 2
Writers delay: 1
Queue size: 10
Time to live: 1

Readers have been successfully created.
Writers have been successfully created.
Time manager has been successfully created.
Writer #0. Put data "Writer #0. Message number #1.".
Writer #2. Put data "Writer #2. Message number #1.".
Writer #1. Put data "Writer #1. Message number #1.".
Writer #0. Put data "Writer #0. Message number #2.".
Writer #2. Put data "Writer #2. Message number #2.".
Writer #1. Put data "Writer #1. Message number #2.".
Writer #0. Put data "Writer #0. Message number #3.".
Writer #2. Put data "Writer #2. Message number #3.".
Writer #1. Put data "Writer #1. Message number #3.".
Writer #0. Put data "Writer #0. Message number #4.".
Writer #2. Put data "Writer #2. Message number #4.".
Writer #1. Put data "Writer #1. Message number #4.".
Writer #0. Put data "Writer #0. Message number #5.".

```

C:\Users\Desktop\p2.2.r.exe

```

Reader #1 started.
Reader #1. Get data "Writer #0. Message number #1.".
Reader #1. Get data "Writer #2. Message number #1.".
Reader #1. Get data "Writer #1. Message number #1.".
Reader #1. Get data "Writer #0. Message number #2.".
Reader #1. Get data "Writer #2. Message number #2.".
Reader #1. Get data "Writer #1. Message number #2.".
Reader #1. Get data "Writer #0. Message number #3.".
Reader #1. Get data "Writer #2. Message number #3.".
Reader #1. Get data "Writer #1. Message number #3.".
Reader #1. Get data "Writer #0. Message number #4.".
Reader #1. Get data "Writer #2. Message number #4.".
Reader #1. Get data "Writer #1. Message number #4.".
Reader #1. Get data "Writer #0. Message number #5.".
Reader #1. Get data "Writer #2. Message number #5.".
Reader #1. Get data "Writer #1. Message number #5.".
Reader #1. Get data "Writer #0. Message number #6.".
Reader #1. Get data "Writer #2. Message number #6.".
Reader #1. Get data "Writer #1. Message number #6.".
Reader #1. Get data "Writer #0. Message number #7.".
Reader #1. Get data "Writer #2. Message number #7.".
Reader #1. Get data "Writer #1. Message number #7.".
Reader #1. Get data "Writer #0. Message number #8.".
Reader #1. Get data "Writer #2. Message number #8.".
Reader #1. Get data "Writer #1. Message number #8.".
Reader #1. Get data "Writer #0. Message number #9.".

```

C:\Users\Desktop\p2.2.r.exe

```

Reader #2 started.
Reader #2. Get data "Writer #0. Message number #1.".
Reader #2. Get data "Writer #2. Message number #1.".
Reader #2. Get data "Writer #1. Message number #1.".
Reader #2. Get data "Writer #0. Message number #2.".
Reader #2. Get data "Writer #2. Message number #2.".
Reader #2. Get data "Writer #1. Message number #2.".
Reader #2. Get data "Writer #0. Message number #3.".
Reader #2. Get data "Writer #2. Message number #3.".
Reader #2. Get data "Writer #1. Message number #3.".
Reader #2. Get data "Writer #0. Message number #4.".
Reader #2. Get data "Writer #2. Message number #4.".
Reader #2. Get data "Writer #1. Message number #4.".
Reader #2. Get data "Writer #0. Message number #5.".
Reader #2. Get data "Writer #2. Message number #5.".
Reader #2. Get data "Writer #1. Message number #5.".
Reader #2. Get data "Writer #0. Message number #6.".
Reader #2. Get data "Writer #2. Message number #6.".
Reader #2. Get data "Writer #1. Message number #6.".
Reader #2. Get data "Writer #0. Message number #7.".
Reader #2. Get data "Writer #2. Message number #7.".
Reader #2. Get data "Writer #1. Message number #7.".
Reader #2. Get data "Writer #0. Message number #8.".
Reader #2. Get data "Writer #2. Message number #8.".
Reader #2. Get data "Writer #1. Message number #8.".
Reader #2. Get data "Writer #0. Message number #9.".

```

Рис. 1.14

Результат работы программы не изменился. Однако теперь конструкция безопасна: читатели больше не изменяют счетчики. Счетчики расположены на стороне писателя. Общение с читателем происходит посредством событий и захватом/освобождением переменных.

#### 4. Оптимизированное решение задачи

Используем критические секции. Они обладают высоким быстродействием по сравнению с мьютексами и, так как в данном случае мы работаем всего с одним процессом, их будет достаточно. Необходимо только синхронизировать запросы от клиентов в пределах одного процесса – сервера. Клиенты не будут подозревать о существовании других клиентов.

```

1 #include <stdio.h>
2 #include <windows.h>
3 #include <iostream>
4
5 static const int WRITERS_COUNT = 4;
6 static const int READERS_COUNT = 3;

```



```

7
8 // Структура для синхронизации данных
9 struct DataSync {
10     int writerId;
11     bool canWrite;
12 };
13
14 CRITICAL_SECTION criticalSection;
15 struct DataSync dataSync;
16
17 // Обработчики потоков
18 DWORD WINAPI threadWriterExecutor(LPVOID argument);
19 DWORD WINAPI threadReaderExecutor(LPVOID argument);
20
21 int main() {
22     // Инициализация структуры синхронизации данных
23     dataSync.writerId = 0;
24     dataSync.canWrite = true;
25
26     // Инициализация критической секции
27     InitializeCriticalSection(&criticalSection);
28
29     // Создание потоковписателей—
30     for(int writerIndex = 0; writerIndex < WRITERS_COUNT; ++writerIndex) {
31         HANDLE writer = CreateThread(nullptr, NULL, threadWriterExecutor, LPVOID(writerIndex)
32         , CREATE_SUSPENDED, nullptr);
33         if(!writer) {
34             std::cerr << "It's impossible to create writer." << std::endl;
35             return 0x1;
36         }
37
38         // Запуск потока
39         ResumeThread(writer);
40         CloseHandle(writer);
41     }
42
43     // Создание потоковчитателей—
44     for(int readerIndex = 0; readerIndex < READERS_COUNT; ++readerIndex) {
45         HANDLE reader = CreateThread(nullptr, NULL, threadReaderExecutor, LPVOID(readerIndex)
46         , CREATE_SUSPENDED, nullptr);
47         if(!reader) {
48             std::cerr << "It's impossible to create reader." << std::endl;
49             return 0x2;
50         }
51
52         // Запуск потока
53         ResumeThread(reader);
54         CloseHandle(reader);
55     }
56
57     // Удаление критической секции
58     DeleteCriticalSection(&criticalSection);
59
60     std::cout << "Press \"Enter\" to exit." << std::endl;
61     std::getchar();
62
63     return 0x0;
64 }
65
66 DWORD WINAPI threadReaderExecutor(LPVOID argument) {
67     int readerId = int(argument);
68
69     while(true) {
70         EnterCriticalSection(&criticalSection);
71
72         if(!dataSync.canWrite) {

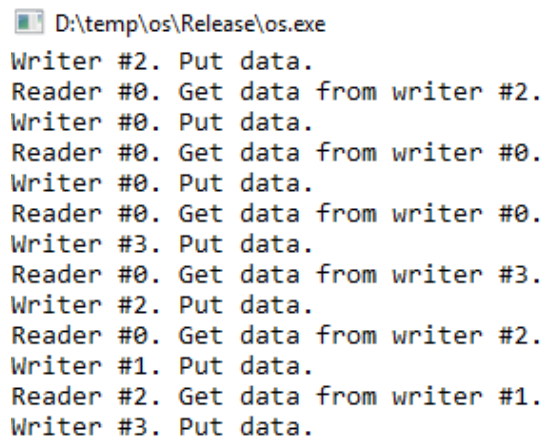
```

```

71     printf("Reader #%d. Get data from writer #%d.\n", readerId, dataSync.writerId);
72
73     // Разрешение дальнейшей записи после считывания
74     dataSync.canWrite = true;
75 }
76
77 LeaveCriticalSection(&criticalSection);
78 }
79 }
80
81 DWORD WINAPI threadWriterExecutor(LPVOID argument) {
82     int writerId = int(argument);
83
84     while(true) {
85         EnterCriticalSection(&criticalSection);
86
87         if(dataSync.canWrite) {
88             // Запись данных
89             dataSync.writerId = writerId;
90             dataSync.canWrite = false;
91
92             printf("Writer #%d. Put data.\n", writerId);
93         }
94
95         LeaveCriticalSection(&criticalSection);
96     }
97 }

```

Результат работы оптимизированной программы:



```

D:\temp\os\Release\os.exe
Writer #2. Put data.
Reader #0. Get data from writer #2.
Writer #0. Put data.
Reader #0. Get data from writer #0.
Writer #0. Put data.
Reader #0. Get data from writer #0.
Writer #3. Put data.
Reader #0. Get data from writer #3.
Writer #2. Put data.
Reader #0. Get data from writer #2.
Writer #1. Put data.
Reader #2. Get data from writer #1.
Writer #3. Put data.

```

Рис. 1.15

## 5. Разработка клиент-серверного приложения с сетевым функционированием

На сервере содержится последовательно изменяющееся значение счетчика от 0 до 5. Клиент пытается угадать текущее значение счетчика. Если значение угадано, то сервер отвечает "Yes! если не угадано, то сервер отвечает "No!".

Синхронизация доступа к переменной, которая является метафорой для любых данных, производится засчет критической секции.

Реализация серверной части:

```

1 #include <winsock2.h>
2 #include <ws2tcpip.h>
3 #include <windows.h>
4 #include <stdio.h>
5 #include <vector>
6 #include <string>
7 #include <iostream>
8

```

```

9 #pragma comment (lib , "Ws2_32.lib")
10
11 static const char* PORT = "65100";
12 static const int BACKLOG = 5;
13 static const int BUFFER_SIZE = 1024;
14 static const int NUMBER_LIMIT = 5;
15 static const int SLEEP_DURATION = 500;
16
17 struct SharedData {
18     int writerId;
19 };
20
21 // Массивы с потоками и сокетами
22 std::vector<HANDLE> writerThreads;
23 std::vector<SOCKET> clientSockets;
24
25 // Серверный сокет
26 SOCKET serverSocket;
27 // Критическая секция
28 CRITICAL_SECTION criticalSection;
29
30 struct SharedData sharedData;
31 int counter = 0;
32
33 // Обработчик потокаписателя—
34 DWORD WINAPI threadWriterExecutor(LPVOID argument);
35
36 int main(int argc, char* argv[]) {
37     // Получение порта
38     std::string port = PORT;
39     if(argc < 2)
40         std::cout << "Using default port: " << port << "." << std::endl;
41     else
42         port = argv[1];
43
44     // Инициализация библиотеки
45     WSADATA wsaData;
46     int wsaStartup = WSASStartup(MAKEWORD(2, 2), &wsaData);
47     if(wsaStartup != 0) {
48         std::cerr << "It's impossible to startup wsa." << std::endl;
49         return 0x1;
50     }
51
52     struct addrinfo hints;
53     ZeroMemory(&hints, sizeof(hints));
54     hints.ai_family = AF_INET;
55     hints.ai_socktype = SOCK_STREAM;
56     hints.ai_protocol = IPPROTO_TCP;
57     hints.ai_flags = AI_PASSIVE;
58
59     struct addrinfo* result = nullptr;
60     int addressInfo = getaddrinfo(nullptr, port.data(), &hints, &result);
61     if(addressInfo != 0) {
62         std::cerr << "It's impossible to get address info." << std::endl;
63         WSACleanup();
64         return 0x2;
65     }
66
67     // Создание серверного сокета
68     serverSocket = socket(result->ai_family, result->ai_socktype, result->ai_protocol);
69     if(serverSocket == INVALID_SOCKET) {
70         std::cerr << "It's impossible to create socket." << std::endl;
71         WSACleanup();
72         return 0x3;
73     }
74

```

```

75     std::cout << "Server socket " << serverSocket << " created." << std::endl;
76
77     // Биндим сервер на определенный адрес
78     int serverBind = bind(serverSocket, result->ai_addr, int(result->ai_addrlen));
79     if(serverBind == SOCKET_ERROR) {
80         std::cerr << "It's impossible to bind socket." << std::endl;
81         closesocket(serverSocket);
82         WSACleanup();
83         return 0x4;
84     }
85
86     freeaddrinfo(result);
87
88     // Слушаем сокет
89     int serverListen = listen(serverSocket, BACKLOG);
90     if(serverListen == SOCKET_ERROR) {
91         std::cerr << "It's impossible to listen socket." << std::endl;
92         closesocket(serverSocket);
93         WSACleanup();
94         return 0x5;
95     }
96
97     // Инициализация критической секции
98     InitializeCriticalSection(&criticalSection);
99     sharedData.writerId = 0;
100
101     std::cout << "Wait clients." << std::endl;
102
103     int threadIndex = 0;
104     while(true) {
105         // Прием клиента
106         SOCKET client = accept(serverSocket, nullptr, nullptr);
107
108         if(client == INVALID_SOCKET)
109             continue;
110
111         // Добавляем сокет в коллекцию
112         clientSockets.push_back(client);
113
114         // Создаем поток
115         HANDLE writer = CreateThread(nullptr, NULL, threadWriterExecutor, LPVOID(threadIndex)
116         , NULL, nullptr);
117         if(!writer) {
118             std::cerr << "It's impossible to create thread." << std::endl;
119             return 0x6;
120         }
121
122         // Добавляем поток в коллекцию
123         writerThreads.push_back(writer);
124         ++threadIndex;
125     }
126
127     return 0x0;
128 }
129
130 DWORD WINAPI threadWriterExecutor(LPVOID argument) {
131     int writerId = int(argument);
132
133     SOCKET writeSocket = clientSockets[writerId];
134
135     char charNumber;
136     char buffer[BUFFER_SIZE];
137     while(true) {
138         ZeroMemory(buffer, BUFFER_SIZE);
139
140         // Получаем один байт

```

```

140 while(true)
141     if(recv(writeSocket, &charNumber, 1, NULL) > 0)
142         break;
143
144 EnterCriticalSection(&criticalSection);
145
146 // Формируем новое число
147 ++counter;
148 if(counter > NUMBER_LIMIT)
149     counter = 0;
150
151 printf("Client #%d. Get number = %c, counter = %d.\n", writerId, charNumber, counter)
152 ;
153
154 // Получаем число из байта
155 int number = charNumber - 0x30;
156 if(number <= 0 || number > 9) {
157     counter = 0;
158     printf("Client reset information to NULL.\n");
159 }
160
161 // Формируем результат
162 if(number == counter)
163     strcpy_s(buffer, "Yes!");
164 else
165     strcpy_s(buffer, "No!");
166
167 // Отправляем результат читателю
168 send(writeSocket, buffer, sizeof(buffer), NULL);
169
170 LeaveCriticalSection(&criticalSection);
171 Sleep(SLEEP_DURATION);
172 }
173 return NULL;
174 }

```

Реализация клиентской части:

```

1 #include <winsock2.h>
2 #include <ws2tcpip.h>
3 #include <windows.h>
4 #include <string>
5 #include <iostream>
6
7 #pragma comment(lib, "Ws2_32.lib")
8
9 static const char* PORT = "65100";
10 static const char* IP = "127.0.0.1";
11 static const int BACKLOG = 5;
12 static const int BUFFER_SIZE = 1024;
13 static const int NUMBER_LIMIT = 5;
14 static const int SLEEP_DURATION = 200;
15
16 SOCKET clientSocket;
17
18 int main(int argc, char* argv[]) {
19     // Получение порта и адреса
20     std::string port = PORT;
21     std::string ip = IP;
22     if(argc < 3) {
23         std::cout << "Using default ip: " << ip << "." << std::endl;
24         std::cout << "Using default port: " << port << "." << std::endl;
25     }
26     else {
27         ip = argv[1];
28         port = argv[2];

```

```

29 }
30
31 // Инициализация библиотеки
32 WSADATA wsaData;
33 int wsaStartup = WSAStartup(MAKEWORD(2, 2), &wsaData);
34 if(wsaStartup != 0) {
35     std::cerr << "It's impossible to startup wsa." << std::endl;
36     return 0x1;
37 }
38
39 struct addrinfo hints;
40 ZeroMemory(&hints, sizeof(hints));
41 hints.ai_family = AF_INET;
42 hints.ai_socktype = SOCK_STREAM;
43 hints.ai_protocol = IPPROTO_TCP;
44
45 // Получение информации об адресе
46 struct addrinfo* addressResult = nullptr;
47 int addressInfo = getaddrinfo(ip.data(), port.data(), &hints, &addressResult);
48 if(addressInfo != 0) {
49     std::cerr << "It's impossible to get address info." << std::endl;
50     WSACleanup();
51     return 0x2;
52 }
53
54 // Пробуем создать сокет и подключиться
55 for(auto currentPtr = addressResult; currentPtr != nullptr; currentPtr = currentPtr->
56     ai_next) {
57     clientSocket = socket(currentPtr->ai_family, currentPtr->ai_socktype, currentPtr->
58     ai_protocol);
59     if(clientSocket == INVALID_SOCKET) {
60         std::cerr << "It's impossible to create socket." << std::endl;
61         WSACleanup();
62         return 0x3;
63     }
64
65     auto serverConnection = connect(clientSocket, currentPtr->ai_addr, int(currentPtr->
66     ai_addrlen));
67     if(serverConnection != SOCKET_ERROR)
68         break;
69
70     closesocket(clientSocket);
71     clientSocket = INVALID_SOCKET;
72 }
73
74 freeaddrinfo(addressResult);
75
76 if(clientSocket == INVALID_SOCKET) {
77     std::cerr << "It's impossible to create connection." << std::endl;
78     WSACleanup();
79     return 0x4;
80 }
81
82 std::cout << "Socket has been successfully created." << std::endl
83     << "Connection established." << std::endl;
84
85 char buffer[BUFFER_SIZE];
86 while(true) {
87     // Формируем число
88     char number = rand() % NUMBER_LIMIT + 0x30;
89
90     // Посылаем число клиенту
91     send(clientSocket, &number, 1, NULL);
92
93     // Считываем ответ с сервера
94     ZeroMemory(buffer, BUFFER_SIZE);

```

```

92     if(recv(clientSocket , buffer , BUFFER_SIZE, NULL) > 0)
93         std::cout << "Get answer from server: " << buffer << std::endl;
94
95     Sleep(SLEEP_DURATION);
96 }
97 }

```

Результат отгадывания тремя клиентами одновременно:

```

C:\Users\Desktop\server.exe
Client #2. Get number = 0, counter = 2.
Client reset information to NULL.
Client #1. Get number = 4, counter = 1.
Client #0. Get number = 4, counter = 2.
Client #2. Get number = 2, counter = 3.
Client #1. Get number = 4, counter = 4.
Client #0. Get number = 3, counter = 5.
Client #2. Get number = 3, counter = 6.
Client #1. Get number = 1, counter = 7.
Client #0. Get number = 4, counter = 8.
Client #2. Get number = 4, counter = 9.
Client #1. Get number = 3, counter = 10.
Client #0. Get number = 2, counter = 11.
Client #2. Get number = 4, counter = 12.
Client #1. Get number = 3, counter = 13.
Client #0. Get number = 2, counter = 14.
Client #2. Get number = 1, counter = 15.
Client #1. Get number = 4, counter = 16.
Client #0. Get number = 2, counter = 17.
Client #2. Get number = 0, counter = 18.
Client reset information to NULL.
Client #1. Get number = 3, counter = 1.
Client #0. Get number = 3, counter = 2.
Client #2. Get number = 1, counter = 3.
Client #1. Get number = 4, counter = 4.
Client #0. Get number = 2, counter = 5.
Client #2. Get number = 0, counter = 6.
Client reset information to NULL.
Client #1. Get number = 4, counter = 1.
Client #0. Get number = 3, counter = 2.
Client #2. Get number = 4, counter = 3.
Client #1. Get number = 4, counter = 4.
Client #0. Get number = 4, counter = 5.
Client #2. Get number = 4, counter = 6.
Client #1. Get number = 2, counter = 7.
Client #0. Get number = 0, counter = 8.
Client reset information to NULL.
Client #2. Get number = 1, counter = 1.

C:\Users\Desktop\client.exe
Get answer from server: No!
Get answer from server: No!
Get answer from server: No!
Get answer from server: No!
Get answer from server: No!
Get answer from server: Yes!
Get answer from server: Yes!
Get answer from server: No!
Get answer from server: Yes!

C:\Users\Desktop\client.exe
Get answer from server: No!
Get answer from server: No!
Get answer from server: No!
Get answer from server: Yes!
Get answer from server: Yes!
Get answer from server: No!
Get answer from server: No!
Get answer from server: No!
Get answer from server: No!
Get answer from server: No!
Get answer from server: Yes!
Get answer from server: No!

C:\Users\Desktop\client.exe
Get answer from server: No!
Get answer from server: No!
Get answer from server: Yes!
Get answer from server: Yes!
Get answer from server: Yes!
Get answer from server: No!
Get answer from server: No!
Get answer from server: No!
Get answer from server: No!
Get answer from server: No!
Get answer from server: Yes!
Get answer from server: No!

```

Рис. 1.16

### 1.3.3 Глава 3. Задача «Обедающие философы»

Формулировка задачи: в одном пансионе, открытом богатым филантропом, были собраны пять знаменитых философов. Предаваясь размышлениям, они независимо друг от друга заходили обедать в общую столовую. В столовой стоял стол, вокруг которого был поставлены стулья. Каждому философу свой стул. Слева от философа лежало по вилке, а в центре стола стояла большая тарелка спагетти. Спагетти можно было есть только двумя вилками, а потому, сев за стол, философ должен был взять вилку соседа справа (если она, конечно, свободна).

#### 1. Реализация модели и программы

Используем мьютексы, чтобы ограничить доступ к вилкам, если они заняты другими философами. Мьютекс в данном случае – самое очевидное и элегантное решение, поскольку вилки являются метафорой для

общих данных, их количество ограничено.

Для избежание дедлоков была вызывается функция `std::lock` в обработчике потока. Также удобно использовать функции `std::bind` и конструктор `std::thread` для передачи в обработчик потока множества аргументов. Таким образом расширенные возможности стандартной библиотеки C++ являются хорошим решением для реализации данной задачи:

```
1 #include <algorithm>
2 #include <iostream>
3 #include <memory>
4 #include <mutex>
5 #include <thread>
6 #include <vector>
7
8 static const int PHILOSOPHERS_COUNT = 5;
9
10 struct Fork {
11     std::mutex mutex;
12 };
13
14 void* threadEatExecutor(Fork* leftChopstick, Fork* rightChopstick, const int
    philosopherNumber, const int leftChopstickNumber, const int rightChopstickNumber);
15
16 int main() {
17     // Массив с вилками для еды
18     std::vector<std::unique_ptr<Fork>> chopsticks(PHILOSOPHERS_COUNT);
19
20     // Создание всех вилок
21     for(int chopstickIndex = 0; chopstickIndex < PHILOSOPHERS_COUNT; ++chopstickIndex) {
22         auto chopstick = std::make_unique<Fork>();
23         chopsticks[chopstickIndex] = std::move(chopstick);
24     }
25
26     // Создаем потоки инструментами стандартной библиотеки std::thread, потому что она позволяет
    передавать в функцию множество аргументов
27
28     // Создание массива потоков
29     std::vector<std::shared_ptr<std::thread>> threads(PHILOSOPHERS_COUNT);
30
31     // Биндим функцию задаем( количество аргументов)
32     auto bindThread = std::bind(&threadEatExecutor, std::placeholders::_1, std::
    placeholders::_2, std::placeholders::_3, std::placeholders::_4, std::placeholders::_5)
    ;
33     // Запускаем поток с определенными аргументами
34     auto sharedThread = std::make_shared<std::thread>(bindThread, chopsticks.front().get(),
    chopsticks.back().get(), 1, 1, PHILOSOPHERS_COUNT);
35     threads[0] = std::move(sharedThread);
36
37     for(int threadIndex = 1; threadIndex < PHILOSOPHERS_COUNT; ++threadIndex) {
38         // Биндим функцию задаем( количество аргументов)
39         bindThread = std::bind(&threadEatExecutor, std::placeholders::_1, std::placeholders::_
    2, std::placeholders::_3, std::placeholders::_4, std::placeholders::_5);
40         // Запускаем поток с определенными аргументами
41         sharedThread = std::make_shared<std::thread>(bindThread, chopsticks[threadIndex - 1].
    get(), chopsticks[threadIndex].get(), threadIndex + 1, threadIndex, threadIndex + 1);
42         threads[threadIndex] = std::move(sharedThread);
43     }
44
45     std::for_each(threads.begin(), threads.end(), std::mem_fn(&std::thread::join));
46
47     std::cout << std::endl << "Press \"Enter\" to exit." << std::endl;
48     std::getchar();
49
50     return 0x0;
51 }
52
53 void* threadEatExecutor(Fork* leftChopstick, Fork* rightChopstick, const int
```

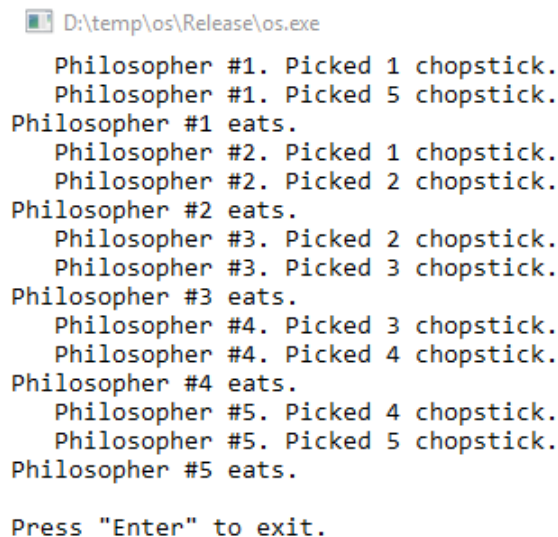


```

54 philosopherNumber, const int leftChopstickNumber, const int rightChopstickNumber) {
55 if(leftChopstick == rightChopstick) {
56     std::cerr << "Chopsticks are same." << std::endl;
57     exit(0x1);
58 }
59 // Для того чтобы не было дедлока
60 std::lock(leftChopstick->mutex, rightChopstick->mutex);
61
62 // Блокируем первую вилку
63 std::lock_guard<std::mutex>(leftChopstick->mutex, std::adopt_lock);
64
65 printf("    Philosopher #%d. Picked %d chopstick.\n", philosopherNumber,
66     leftChopstickNumber);
67
68 // Блокируем вторую вилку
69 std::lock_guard<std::mutex>(rightChopstick->mutex, std::adopt_lock);
70
71 printf("    Philosopher #%d. Picked %d chopstick.\n", philosopherNumber,
72     rightChopstickNumber);
73
74 printf("Philosopher #%d eats.\n", philosopherNumber);
75 return nullptr;
76 }

```

Результат решения задачи:



```

D:\temp\os\Release\os.exe
    Philosopher #1. Picked 1 chopstick.
    Philosopher #1. Picked 5 chopstick.
    Philosopher #1 eats.
    Philosopher #2. Picked 1 chopstick.
    Philosopher #2. Picked 2 chopstick.
    Philosopher #2 eats.
    Philosopher #3. Picked 2 chopstick.
    Philosopher #3. Picked 3 chopstick.
    Philosopher #3 eats.
    Philosopher #4. Picked 3 chopstick.
    Philosopher #4. Picked 4 chopstick.
    Philosopher #4 eats.
    Philosopher #5. Picked 4 chopstick.
    Philosopher #5. Picked 5 chopstick.
    Philosopher #5 eats.

Press "Enter" to exit.

```

Рис. 1.17

Каждый философ действительно использовал только собственную вилку и вилку соседа только в то время пока они были свободны.

## 1.4 Вывод

В ходе работы были изучены такие средства синхронизации в ОС Windows как: семафоры, мьютексы, критические секции, объекты-события, условные переменные. Самым быстрым средством считается критическая секция, т.к. она выполняется в режиме задачи и максимально упрощена.

- Мьютексы являются аналогом семафора с двумя возможными значениями. Одновременно доступ к мьютексу имеет только один поток. Мьютексы могут быть использованы для синхронизации как потоков, так и процессов.
- Семафоры позволяют захватить себя нескольким потокам, после чего захват будет невозможен, пока один из ранее захвативших семафор потоков не освободит его. Семафоры применяются для ограничения количества потоков, одновременно работающих с ресурсами. В Windows семафоры и мьютексы могут иметь имя, через которое другие процессы могут получить доступ к объектам.
- Критическая секция позволяет выделить участок кода, где поток получает доступ к разделяемому ресурсу, и предотвратить одновременное использование ресурса. К критической секции одномоментно имеет доступ только один поток. Данное средство не может быть применено для синхронизации процессов. Критическая секция не является объектом ядра, поэтому использовать функции семейства Wait не представляется возможным.
- События используются для уведомления ожидающих потоков о наступлении какого-либо события. Существует два вида событий - с ручным и автоматическим сбросом. События с ручным сбросом используются для уведомления сразу нескольких потоков. При использовании события с автоматическим сбросом уведомление получит и продолжит свое выполнение только один ожидающий поток, остальные будут ожидать дальше.