

Пермский филиал федерального государственного автономного
образовательного учреждения высшего образования
«Национальный исследовательский университет
«Высшая школа экономики»

Факультет экономики, менеджмента и бизнес-информатики

Чепокhov Елизар Сергеевич

СИНХРОНИЗАЦИЯ

Реферат

студента образовательной программы «Программная инженерия»
по направлению подготовки 09.03.04 Программная инженерия

Доцент кафедры
информационных
технологий в бизнесе

Л. Н. Лядова

Пермь, 2020 год

Средства синхронизации процессов и потоков

Существует достаточно обширный класс средств операционной системы, с помощью которых обеспечивается взаимная синхронизация процессов и потоков. Потребность в синхронизации потоков возникает только в мультипрограммной операционной системе и связана с совместным использованием аппаратных и информационных ресурсов вычислительной системы. Синхронизация необходима для исключения гонок и тупиков при обмене данными между потоками, разделении данных, при доступе к процессору и устройствам ввода-вывода.

Любое взаимодействие процессов или потоков связано с их синхронизацией, которая заключается в согласовании их скоростей путем приостановки потока до наступления некоторого события и последующей его активизации при наступлении этого события. Синхронизация лежит в основе любого взаимодействия потоков, связано ли это взаимодействие с разделением ресурсов или с обменом данными.

Критические секции

Последовательность инструкций, *одновременное выполнение* которой может привести к неправильным результатам называется *критической секцией*. Каждая критическая секция требует реализации взаимного исключения по отношению к одной конкретной разделяемой единице данных (переменной в общей памяти, целому файлу или записи в файле и т.п.), которая выступает в качестве последовательно используемого, требующего монопольного доступа ресурса.

На критическую секцию, связанную с доступом к какому-либо разделяемому несколькими процессами информационному ресурсу, налагаются следующие *требования*:

1. в любой момент времени только один процесс может находиться в своей критической секции по данному ресурсу (это главное требование - взаимное исключение);
2. ни один процесс не должен ждать бесконечно долго входа в критическую секцию (реализация взаимного исключения не должна приводить к ошибкам или невозможности выполнения процессами своих функций - взаимное исключение только устанавливает порядок доступа к общим ресурсам, исключая их разрушение);
3. ни один процесс не может находиться в своей критической секции бесконечно долго (это следствие предыдущего требования - все процессы в течение приемлемого времени должны получить доступ к разделяемым данным для выполнения своих функций);
4. никакой процесс, находящийся вне своей критической секции, не должен задерживать выполнение других процессов, ожидающих входа в свои критические секции.

Использование критической секции

В качестве примера было решено привести вывод различными потоками сообщения в консоли. В данном случае *cout* является разделяемым ресурсом, за который «борются» 4 потока. Код программы выглядит следующим образом:

```
#include "pch.h"
#include <iostream>
#include <thread>

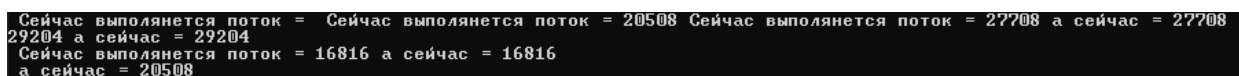
using namespace std;
void Inc()
{
    cout << " Сейчас выполняется поток = " << this_thread::get_id()
        << " а сейчас = " << this_thread::get_id() << '\n';
}

int main()
{
    setlocale(0, "");
    thread t1(Inc);
    thread t2(Inc);
    thread t3(Inc);
    thread t4(Inc);

    t1.join();
    t2.join();
    t3.join();
    t4.join();
    return 0;
}
```

В этом коде поток создается с помощью *thread* – реализации многопоточной работы с помощью библиотеки C++. C++ *thread* в конечном счете все равно вызовет *CreateThread*, это сделает C++ Runtime. Так что по сути это то же самое, что изначально создать процесс *CreateThread*, однако выглядит более компактно. Кроме того, использование *CreateThread* сразу сделает невозможной компиляцию программы под другой операционкой, поскольку в разной ОС различные функции C++. Но в следующих примерах все же было решено использовать *CreateThread*.

Данный пример наиболее наглядно показывает, когда какой поток выполнялся. Конечно, сообщения «Сейчас выполняется поток =», «а сейчас =» могут быть написаны любым потоком, однако кое-какая последовательность получения доступа к консоли прослеживается. Например на данном примере первый поток (любой из четырех) вывел первую часть предложения, после чего другой поток отобрал доступ к выводу и так же вывел первую часть предложения, после этого сообщение мог вывести любой из 4 потоков и т.д.



```
Сейчас выполняется поток = Сейчас выполняется поток = 20508 Сейчас выполняется поток = 27708 а сейчас = 27708
29204 а сейчас = 29204
Сейчас выполняется поток = 16816 а сейчас = 16816
а сейчас = 20508
```

Чтобы предотвратить некорректное выполнение конкурирующих за доступ к общим данным потоков было решено воспользоваться критической секцией – наиболее простым механизмом синхронизации доступа к разделяемым ресурсам. Их можно использовать для синхронизации потоков, работающих в рамках одного процесса. Поскольку в данной программе не создаются дополнительные процессы, было

решено воспользоваться данным способом решения проблемы взаимного исключения.

`CRITICAL_SECTION` `csWindowsPaint`; располагается в области глобальных переменных, доступной всем выполняющимся потокам процесса. Так как у каждого процесса свое собственное адресное пространство, адрес критической секции *нельзя передать другим процессам*. Структура `CRITICAL_SECTION` и указатели на нее определены в файле `winbase.h` (он автоматически включается при включении файла `windows.h`), поэтому для модификации кода программы была добавлена библиотека `windows.h`. Код новой программы представлен ниже.

```
#include "pch.h"
#include <iostream>
#include <thread>
#include "windows.h"

using namespace std;

CRITICAL_SECTION criticalsect;
void Inc()
{
    EnterCriticalSection(&criticalsect);
    cout << " Сейчас выполняется поток = " << this_thread::get_id()
         << " а сейчас = " << this_thread::get_id() << '\n';
    LeaveCriticalSection(&criticalsect);
}

int main()
{
    InitializeCriticalSection(&criticalsect);
    setlocale(0, "");
    thread t1(Inc);
    thread t2(Inc);
    thread t3(Inc);
    thread t4(Inc);

    t1.join();
    t2.join();
    t3.join();
    t4.join();
    return 0;
}
```

Перед использованием критическая секция была проинициализирована в `main` с помощью функции `InitializeCriticalSection`, которой передается единственный параметр — адрес структуры типа `CRITICAL_SECTION`. Для входа в критическую секцию использовалась функция `EnterCriticalSection`. Если один из потоков процесса вошел в критическую секцию, при попытке других потоков войти в нее они будут переведены в состояние ожидания, пока поток, занявший критическую секцию, не выйдет из нее с помощью функции `LeaveCriticalSection`. Таким образом гарантируется, что фрагмент кода, заключенный между вызовами функций, представляющих вход в критическую секцию и выход из нее, будет выполняться потоками одного процесса последовательно. Выход одного потока из занятой им критической секции может активизировать следующий поток, ожидающий входа в указанную параметром критическую секцию.

Результат выполнения модифицированной программы:

```
Сейчас выполняется поток = 23808 а сейчас = 23808
Сейчас выполняется поток = 20512 а сейчас = 20512
Сейчас выполняется поток = 20840 а сейчас = 20840
Сейчас выполняется поток = 6372 а сейчас = 6372
```

Реализация задачи «писателей-читателей» при однократной записи и считывании данных

В задаче писателей-читателе для организации взаимодействия двух процессов, обмена сообщениями между ними используется буфер, в который сообщения помещаются в виде отдельных записей. Существует два процесса, один из которых является процессом-производителем (“писателем”), помещающим данные в буфер, а второй - процессом-потребителем (“читателем”), считывающим из буфера данные, записанные в него первым процессом, в порядке их размещения в буфере.

Для того чтобы реализовать данную задачу при однократной записи и считывании данных было решено использовать объекты семафоры (однако можно было бы использовать и Mutex, поскольку лимит надо сделать на 1 запись, что можно воплотить и с помощью мьютексов). Его преимущество состоит в том, что он позволяет установить счетчик при организации доступа к ресурсу: возможность параллельной работы с ресурсом обеспечивается для заранее определенного ограниченного числа задач (потоков). Потоки, пытающиеся получить доступ к ресурсам сверх установленного лимита, будут переведены в состояние ожидания, пока какой-либо поток, получивший доступ к ресурсу раньше, не освободит его. В данной задаче необходимо поставить лимит на 1 поток, обращающийся к процедуре чтения и записи.

Код выглядит следующим образом:

```
#include "pch.h"
#include <windows.h>
#include <stdio.h>
#include <process.h>
#include <iostream>
#include <fstream>
#include <string>
#include <thread>
#include <mutex>
#include <time.h>

using namespace std;
HANDLE hSem;
unsigned long wThread;
unsigned long rThread;
HANDLE wThr;
HANDLE rThr;

static string students[1] = {};

void Writer(void*)
{
    ofstream file("Entry.txt");
    string text;
    cout << "Input text: ";
    cin >> text;
```

```

        file << text;
        file.close();
        ReleaseSemaphore(hSem, 1, NULL);
    }

void Reader(void*)
{
    WaitForSingleObject(hSem, INFINITE);
    char buff[50];
    ifstream file("Entry.txt");
    while (file.getline(buff, 50))
    {
        cout << buff << endl;
    }
    file.close();
    cout << buff << endl;
    ReleaseSemaphore(hSem, 1, NULL);
}

int main(void)
{
    hSem = CreateSemaphore(NULL, 0, 1, NULL);
    wThr = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Writer, NULL, 0, &wThread);
    rThr = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Reader, NULL, 0, &rThread);
    WaitForSingleObject(rThr, INFINITE);

    return 0;
}

```

В качестве писателя выступает пользователь, запустивший приложение. Данная программа позволяет ввести сообщение в консоли, после чего оно записывается в файл. При чтении сообщение достаётся из файла и выводится в консоли.

Для создания семафора вызывается функция `CreateSemaphore` с параметрами:

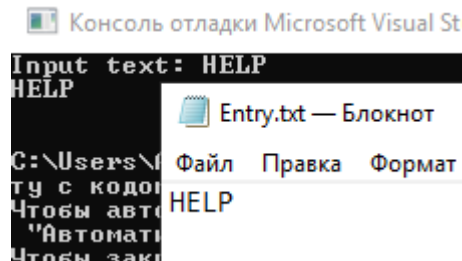
- `LPSECURITY_ATTRIBUTES lpSemaphoreAttributes` – атрибуты защиты объекта;
- `LONG lInitialCount` – начальное значение счетчика семафора;
- `LONG lMaximumCount` – максимальное значение счетчика семафора;
- `LPCTSTR lpName` – адрес строки, содержащей имя объекта `Semaphore`.

Поскольку данный семафор используется в рамках одного приложения, последнее значение указано как `NULL`. Значение счетчика семафора уменьшается с помощью функции, которая позволяют захватить ресурс или организовать его ожидание, – функций `WaitForSingleObject`. Функция `ReleaseSemaphore` увеличивает значение счетчика. При вызове ей передаются следующие параметры:

- `HANDLE hSemaphore` – описатель семафора;
- `LONG cReleaseCount` – значение инкремента (счетчик увеличивается на заданную положительную величину);
- `LPLONG lplPreviousCount` – адрес переменной для записи предыдущего значения (значения, предшествующего вызову функции) счетчика.

Поскольку в начальном значении счетчика семафора было указано число 0, то поток для чтения не может войти в блок считывания информации, поскольку там

установлено `WaitForSingleObject(hSem, INFINITE)` – а вычитать не из чего, минимальное значение семафора и так равно 0. Однако поток для записи, в конце метода записи увеличивает значение семафора с помощью `ReleaseSemaphore(hSem, 1, NULL)`, и теперь другой поток может произвести чтение. Таким образом была решена проблема взаимного исключения.



Реализация задачи «писателей-читателей» при работе с циклическим буфером при условии, что выполняется только один «писатель» и только один «читатель».

Проблема в этом случае состоит в том, что “читатель” не должен выполнять чтение из пустого буфера, в который еще не поместили информацию, а также не должен считывать одну и ту же информацию дважды; а “писатель” не должен пытаться писать информацию в переполненный буфер или переписывать записи, которые еще не были прочитаны “читателем”.

В данном случае требуется не только организовать взаимное исключение при доступе к общим данным, но и вести учет ресурсов, в качестве которых выступают свободные для записи области буфера (для процесса-производителя) и записи, размещенные в буфере “писателем” (для процесса-потребителя).

В связи с поставленной задачей было решено использовать для решения проблемы взаимного исключения 2 семафора (для чтения и записи) и мьютекс (для поддержания порядка между двумя процессами – записью и чтением).

Главное отличие мьютекса от критической секции является то, что его могут использовать для синхронизации задач, выполняющихся в рамках различных процессов. В отличие от семафора в мьютексе нельзя задать счетчик при организации доступа к ресурсу (он может быть только в 2 состояниях – отмеченном и неотмеченном).

```
#include "pch.h"
#include <iostream>
#include <windows.h>
#include <fstream>

using namespace std;
HANDLE hSemWrite;
HANDLE hSemRead;
HANDLE hMut;

unsigned long wThrID;
unsigned long rThrID;

HANDLE wThr;
HANDLE rThr;
```

```

HANDLE hThreads[2]; // для двух потоков

int arr[3]; // массив играющий роль буфера
int wasReaded;
int forW;

void Reader(void*)
{
    for (int j = 0; j < 10; j++)
    {
        WaitForSingleObject(hSemRead, INFINITE);
        WaitForSingleObject(hMut, INFINITE);

        cout << "I read this " << arr[wasReaded] << endl;
        wasReaded++;
        if (wasReaded > 2)
            wasReaded = 0;

        ReleaseMutex(hMut);
        ReleaseSemaphore(hSemWrite, 1, NULL);
    }
}

void Writer(void*)
{
    for (int i = 0; i < 10; i++)
    {
        WaitForSingleObject(hSemWrite, INFINITE);
        WaitForSingleObject(hMut, INFINITE);

        arr[forW] = i + 1;
        cout << "I write this " << arr[forW] << endl;
        forW++;
        if (forW == 3)
        {
            forW = 0;
        }

        ReleaseMutex(hMut);
        ReleaseSemaphore(hSemRead, 1, NULL);
    }
}

int main(void)
{
    hSemWrite = CreateSemaphore(NULL, 3, 3, NULL);
    hSemRead = CreateSemaphore(NULL, 0, 3, NULL);
    hMut = CreateMutex(NULL, FALSE, NULL);
    wThr = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Writer, NULL, 0, &wThrID);
    rThr = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Reader, NULL, 0, &rThrID);
    hThreads[0] = wThr;
    hThreads[1] = rThr;
    WaitForMultipleObjects(2, hThreads, TRUE, INFINITE);
    return 0;
}

```

Семафоры создаются с помощью функции CreateSemaphore, причем семафор для записи в качестве начального и максимального значения имеет одно и то же число – 3 – длина массива, который в этом случае играет роль буфера. Семафор для чтения изначально имеет значение 0, а максимальное равняется так же 3. Нулевое значение семафора для чтения предотвращает возможность считывания записей, до того, как их записали. А начальное значение семафора для записи обеспечивает

первоначальную запись в буфер, так как максимальное число записей, хранящихся в буфере равно 3, то после прочтения, часть записей перезаписывается. Мьютекс создается с помощью функции `CreateMutex`, которой передаются параметры:

- `LPSECURITY_ATTRIBUTES lpMutexAttributes` – атрибуты защиты объекта;
- `BOOL bInitialState` – флаг начального состояния объекта (`TRUE` – поток, создавший объект будет им владеть сразу после создания, `FALSE` – после создания `Mutex` не будет принадлежать ни одному потоку до его явного захвата с помощью специальной функции);
- `LPCTSTR lpName` – адрес строки, содержащей имя объекта `Mutex`.

В качестве параметра для начального состояния был установлен флаг `FALSE`, поскольку нам надо, чтобы созданный мьютекс не принадлежал ни одному потоку, пока он не зайдет в метод `Writer`. В методах чтения и записи функция `WaitForSingleObject(hMut, INFINITE)` является вложенной в функции `WaitForSingleObject(hSemRead, INFINITE)` и `WaitForSingleObject(hSemWrite, INFINITE)` соответственно, и предотвращает возможность выполнения обеих задач одновременно. Однако в начале выполнения обоих методов стоит уменьшение значения соответствующего методу семафора на 1 или же ожидание, пока ресурс можно будет захватить. После этого выполняются действия по записи или чтению в массив и вывод в консоль, что было сделано. Затем мьютекс освобождается, после чего значения для другого семафора увеличивается на единицу – это дает доступ к другому методу. Например, в методе чтения семафор для чтения уменьшает значение на 1, но потом не прибавляет его – это происходит только в методе записи, а в методе чтения увеличивается значение только для семафора записи, что обеспечивает очередность выполнения. В самом конце с помощью `WaitForMultipleObjects(2, hThreads, TRUE, INFINITE)` с параметрами:

- `DWORD cObjects` – количество описателей объектов в массиве, для завершения которых организуется ожидание;
- `CONST HANDLE *lpObjects` – адрес массива описателей объектов, для завершения которых организуется ожидание;
- `BOOL fWaitAll` – тип ожидания: если передается значение `TRUE`, поток переводится в состояние ожидания до тех пор, пока все задачи или процессы, идентификаторы которых указаны в массиве, не завершат свою работу; если же передается значение `FALSE`, ожидание прекращается, когда один из указанных объектов завершит свою работу;
- `DWORD dwTimeOut` – время ожидания в миллисекундах.

организовывается ожидание завершения сразу нескольких потоков – для записи и чтения. Данный код позволяет не только поочередно выводить запись – чтение, но и есть несколько раз (до 3, по объему буфера) записывать данные, так как максимальное значение для семафора записи = 3, то заходя 3 раза подряд, оно станет 0, после чего доступ к записи будет закрыт до тех пор, пока не прочитают хоть одну

запись. Аналогично происходит с чтением. Если занесли 3 записи, то поток чтения может выполниться 3 раза подряд, или же выполняться по одному.

На данном скриншоте отлично демонстрируется то, что методы чтения и записи происходят поочередно, причем вначале происходит занесение записи, а потом ее чтение. Как можно заметить, в данном случае поток для записи выполнялся 2 раза подряд, но несмотря на это все данные выводятся верно, нет чтения одной и той же записи или пропущенных значений.

Реализация задачи «писателей-читателей» при работе с циклическим буфером при условии, что выполняется несколько «писателей» и несколько «читателей»

По отношению к предыдущим задачам писателей-читателей, в этой проблеме составляет то, что их может быть n-ое количество. В связи с этим необходимо добавить еще дополнительные мьютексы, которые будут отслеживать, что данную запись одновременно редактирует только один писатель/читатель, тем самым предотвратив запись/чтение в одну и ту же ячейку.

```
#include "pch.h"
#include <iostream>
#include <windows.h>
#include <thread>
#define THREADCOUNT 4

using namespace std;
HANDLE hSemWrite;
HANDLE hSemRead;
HANDLE hMut;
HANDLE hMutW;
HANDLE hMutR;

HANDLE hThr;
HANDLE hThreads[THREADCOUNT];

unsigned long ThrID;
int arr[3]; // массив играющий роль буфера
int wasReaded;
int forW;
int i;
int j;

void Writer(void*)
{
    while (true) {
        WaitForSingleObject(hMutW, INFINITE);
        if (i < 9)
        {
            WaitForSingleObject(hSemWrite, INFINITE);
            WaitForSingleObject(hMut, INFINITE);

            arr[forW] = ++i;
            cout << "I write this " << arr[forW] << " thread " <<
this_thread::get_id() << endl;
            if (++forW == 3)
                forW = 0;

            ReleaseMutex(hMut);
            ReleaseMutex(hMutW);
```

```

        ReleaseSemaphore(hSemRead, 1, NULL);
    }
    else ReleaseMutex(hMutW);
}

void Reader(void*)
{
    while (true) {
        WaitForSingleObject(hMutR, INFINITE);
        if (j < 9)
        {
            WaitForSingleObject(hSemRead, INFINITE);
            WaitForSingleObject(hMut, INFINITE);

            cout << "I read this " << arr[wasReaded] << "   thread " <<
this_thread::get_id() << endl;
            wasReaded++;
            if (wasReaded > 2)
                wasReaded = 0;
            j++;

            ReleaseMutex(hMut);
            ReleaseMutex(hMutR);
            ReleaseSemaphore(hSemWrite, 1, NULL);
        }
        else ReleaseMutex(hMutR);
    }
}

int main(void)
{
    hSemWrite = CreateSemaphore(NULL, 3, 3, NULL);
    hSemRead = CreateSemaphore(NULL, 0, 3, NULL);
    hMut = CreateMutex(NULL, FALSE, NULL);
    hMutW = CreateMutex(NULL, FALSE, NULL);
    hMutR = CreateMutex(NULL, FALSE, NULL);

    for (int i = 0; i < THREADCOUNT; i++)
    {
        if (i % 2 == 0) hThr = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE)Writer, NULL, 0, &ThrID);
        else
            hThr = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Reader, NULL, 0,
&ThrID);
        hThreads[i] = hThr;
    }
    WaitForMultipleObjects(THREADCOUNT, hThreads, TRUE, 1000);
    return 0;
}

```

Код аналогичный предыдущему лишь с 2 добавленными мьютексами, которые вызываются внутри каждого метода и регулируют работу нескольких писателей и читателей, ограничивая запись в одну и ту же ячейку несколько раз, чтение аналогично. Для корректной работы hMutW и hMutR ограничивают доступ к глобальным переменным i и j, поэтому они установлены перед условием if.

```

I write this 1 thread 9476
I write this 2 thread 33892
I read this 1 thread 6108
I write this 3 thread 9476
I read this 2 thread 7532
I write this 4 thread 33892
I read this 3 thread 6108
I write this 5 thread 9476
I read this 4 thread 7532
I write this 6 thread 33892
I read this 5 thread 6108
I write this 7 thread 9476
I read this 6 thread 7532
I write this 8 thread 33892
I read this 7 thread 6108
I write this 9 thread 9476
I read this 8 thread 7532
I read this 9 thread 6108
C:\Users\Алина\Desktop\Учеба\2

```

Как можно заметить, запись и чтение реализуют различные потоки, причем все записанные значения читаются, не повторяются при записи и чтении, не теряются. Так как буфер состоит из 3 элементов, то можно заметить, что новая итерация заполнения и чтения массива организовано верно, поскольку все записанное прочитывается.

Реализация задачи «обедающих философов» («обедающих мудрецов») для произвольного числа запущенных процессов/потоков («философов»)

“Пять философов садятся обедать за круглый стол, в центре которого стоит одно блюдо со спагетти. На столе имеется пять тарелок и пять вилок между ними. Философ может начать есть, если у него есть тарелка и две вилки, которые он может взять с двух сторон от своей тарелки. Философ может отдать вилки соседям только после того, как он закончит обед”.

На основе псевдокода была составлена следующая программа:

```

#include "pch.h"
#include <iostream>
#include <windows.h>
#include "windows.h"
#include "conio.h"
#include "stdlib.h"
#include "list"

#define min_time 200
#define max_time 1500
#define step 2
#define philCount 5

using namespace std;
list<int> Threads_sequence;
HANDLE iteration = CreateEvent(NULL, TRUE, TRUE, NULL);
int philNumber = 0;
CRITICAL_SECTION critSec;

HANDLE ev[philCount];
HANDLE Philosophers[philCount];

DWORD WINAPI philosopher(LPVOID arg)
{
    const int leftFork = philNumber, rightFork = philNumber + 1;
    SetEvent(iteration);

```

```

    for (int i = 0; i < step; i++)
    {
        Sleep(min_time + rand() % (max_time - min_time));
        Threads_sequence.push_back(leftFork);

        while (true)
        {
            if (Threads_sequence.front() == leftFork) {
Threads_sequence.pop_back(); break; }
            else continue;
        }

        EnterCriticalSection(&critSec);
        if (WaitForSingleObject(ev[leftFork], 1) == WAIT_TIMEOUT)
        {
            if (WaitForSingleObject(ev[rightFork], 1) == WAIT_TIMEOUT)
            {
                cout<<"Philosopher " << leftFork << " want to eat. Left and
right forks are " << leftFork << " " << rightFork << endl;
                SetEvent(ev[leftFork]);
                SetEvent(ev[rightFork]);

                LeaveCriticalSection(&critSec);
                cout << "Philosopher " << leftFork << " begin Eating \n" <<
endl;

                Sleep(min_time + rand() % (max_time - min_time));
                ResetEvent(ev[leftFork]);
                ResetEvent(ev[rightFork]);
                cout << "Philosopher " << leftFork << " end Eating " << endl;
            }
            else if (WaitForSingleObject(ev[rightFork], 1) == WAIT_OBJECT_0)
                cout << "Philosopher " << leftFork << " want to eat. But right
fork " << rightFork << " is unable " << endl;
            }
            else if (WaitForSingleObject(ev[leftFork], 1) == WAIT_OBJECT_0)
                cout << "Philosopher " << leftFork << " want to eat. But left fork "
<< leftFork << " is unable " << endl;
            else if (WaitForSingleObject(ev[rightFork], 1) == WAIT_OBJECT_0)
                cout << "Philosopher " << leftFork << " want to eat. But right fork "
<< rightFork << " is unable " << endl;
            else if ((WaitForSingleObject(ev[rightFork], 1) == WAIT_OBJECT_0) &
(WaitForSingleObject(ev[leftFork], 1) == WAIT_OBJECT_0))
                cout << "Philosopher " << leftFork << " want to eat. But left " <<
leftFork << " and right " << rightFork << " is unable " << endl;

            LeaveCriticalSection(&critSec);
            Sleep(1000);
        }
        return 0;
    }
}
int main(int argc, char argv[])
{
    InitializeCriticalSection(&critSec);
    for (int i = 0; i < philCount; i++)
    {
        ev[i] = CreateEvent(NULL, TRUE, FALSE, NULL);
        ResetEvent(ev[i]);
    }
    for (int i = 0; i < philCount; i++)
    {
        WaitForSingleObject(iteration, INFINITE);
        ResetEvent(iteration);
        philNumber = i;
        Philosophers[i] = CreateThread(NULL, 0, philosopher, NULL, 0, NULL);
    }
}

```

```
        _getch();  
        return 0;  
    }
```

В данном коде используются синхронизация задач с помощью событий, они создаются с помощью функции `CreateEvent`, которой передаются следующие параметры:

- `LPSECURITY_ATTRIBUTES lpEventAttributes` – атрибуты защиты объекта;
- `BOOL bManualReset` – флаг ручного сброса события;
- `BOOL bInitialState` – флаг начального состояния события (`TRUE` – событие создается в отмеченном состоянии, `FALSE` – начальным состоянием события будет неотмеченное состояние);
- `LPCTSTR lpName` – адрес строки, содержащей имя объекта-события.

Для перевода события в отмеченное (сигнализированное) состояние используется функция `SetEvent`, которой в качестве параметра передается описатель объекта-события. Для перевода события в неотмеченное (несигнализированное) состояние используется функция `ResetEvent`, которой в качестве параметра передается описатель объекта-события.

Событие создается для каждой вилки и это предотвращает взятие вилки другим философом, если она уже используется другим. Так же для решения задачи используется критическая секция, в ней надо подойти к столу, начать есть, после чего уйти.