Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

**«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ**

**УНИВЕРСИТЕТ ИМЕНИ Н.Э.БАУМАНА»**

**Факультет «Информатика и системы управления»**

**Кафедра «Программное обеспечение ЭВМ и информационные технологии» (ИУ7)**

**РАСЧЁТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**к курсовому проекту по дисциплине операционные системы на тему:**

**Реализация задачи «Читатели – писатели»**

**при помощи монитора Хоара и**

**объектов ядра Microsoft Windows**

Работу выполнил:

студент группы ВИУ7-61

Шафорост Б.Н.

Научный руководитель:

доцент кафедры ИУ7

Рязанова Н.Ю.

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_\_\_

Москва 2015г.

**Оглавление**

Введение.........................................................................................................................................3

Глава 1. Процессы и потоки. Многозадачность в операционных системах........................4   
 1.1. Процессы...............................................................................................................4

1.2. Потоки....................................................................................................................4 1.3. Многозадачность...................................................................................................5

Глава 2. Объекты синхронизации и функции ожидания в операционной системе Windows...................................................................................................................................8   
 2.1. Функции ожидания...............................................................................................8

2.2. Событие (Event)...................................................................................................10 2.3. Мьютекс...............................................................................................................12

Глава 3. Задача о читателях-писателях...............................................................................15   
 3.1. Постановка задачи..............................................................................................15

3.2. Монитор Хоара....................................................................................................16

3.3. Решение задачи...................................................................................................18 3.4. Код программы на языке С++............................................................................19

Заключение............................................................................................................................21

Список литературы...............................................................................................................22

# Введение

Одной из характерных особенностей развития цифровой техники последних лет становится стремление производителей вычислительных машин увеличить их производительность не за счёт увеличения быстродействия отдельного процессорного ядра, а за счёт увеличения числа ядер процессора. Появление новых компьютеров с несколькими ядрами позволяет решать задачи более гибко и предоставлять пользователю более удобные программы. Однако наиболее важным назначением компьютерной техники по-прежнему остается использование её в том направлении, для которого она собственно и создавалась, а именно, для решения больших задач, требующих выполнения громадных объемов вычислений. Современные операционные системы поддерживают многозадачность. Поэтому перед программистами поставлена очень важная цель: обеспечить максимально эффективное использование ресурсов оборудования, а также беспрепятственное и безошибочное выполнение нескольких задач одновременно.

Многопоточные приложения предоставляют пользователю больше возможностей по сравнению с однопоточными. В то же время, разработчик программного обеспечения получает больше возможностей для реализации архитектурных решений. Однако, использование многопоточности сопряжено и с усложнением структуры программы, увеличением сложности в ее проектировании и при поддержке.

В данной курсовой работе будет рассмотрена многозадачность операционной системы как принцип, позволяющий реализовать одновременное выполнение нескольких программ на одном процессоре ЭВМ. Будут показаны механизмы реализации многозадачности средствами ядра операционной системы, рассмотрены объекты синхронизации ядра операционной системы Windows: событие и мьютекс, а также функции ожидания этих событий. В качестве примера будет приведена классическая задача «читатели-писатели» и реализовано её решение с помощью монитора Хоара и объектов синхронизации Windows (события и мьютекс) на языке программирования C++.

**Глава 1. Процессы и потоки.**

**Многозадачность в операционных системах.**

## 1.1. Процессы.

Процесс обычно определяют как экземпляр выполняемой программы, и он состоит из двух компонентов:

1. Объекта ядра, через который операционная система управляет процессом. Там же хранится статистическая информация о процессе.
2. Адресного пространства, в котором содержится код и данные всех EXE и DLL модулей. Именно в нем находятся области памяти, динамически распределяемой для стеков, потоков и других нужд.[[1]](#footnote-2)

Процессы инертны. Чтобы процесс что-нибудь выполнил, в нем нужно создать поток. Именно потоки отвечают за исполнение кода, содержащегося в адресном пространстве процесса. Один процесс может владеть несколькими потоками, и тогда они "одновременно" исполняют код в адресном пространстве процесса.

Для этого каждый поток должен располагать собственным набором регистров процессора и собственным стеком. В каждом процессе есть минимум один поток. Если бы у процесса не было ни одного потока, ему нечего было бы делать, и система автоматически уничтожила бы его вместе с выделенным ему адресным пространством.

Чтобы все эти потоки работали, операционная система отводит каждому из них определенное процессорное время. Выделяя потокам отрезки времени (называемые квантами), она создает тем самым иллюзию одновременного выполнения потоков.

При создании процесса первый (точнее, первичный) поток создается системой автоматически. Далее этот поток может породить другие потоки, те в свою очередь — новые и т. д.

## 1.2. Потоки.

Любой поток состоит из двух компонентов:

1. Объекта ядра, через который операционная система управляет потоком. Там же хранится статистическая информация о потоке.
2. Стека потока, который содержит параметры всех функций и локальные переменные, необходимые потоку для выполнения кода.

Потоки всегда создаются в контексте какого-либо процесса, и вся их жизнь проходит только в его границах. На практике это означает, что потоки исполняют код и манипулируют данными в адресном пространстве процесса. Поэтому, если два и более потоков выполняется в контексте одного процесса, все они делят одно адресное пространство. Потоки могут исполнять один и тот же код и манипулировать одними и теми же данными, а также совместно использовать описатели объектов ядра, поскольку таблица описателей создается не в отдельных потоках, а в процессах.

Процессы используют куда больше системных ресурсов, чем потоки. Причина кроется в адресном пространстве. Создание виртуального адресного пространства для процесса требует значительных системных ресурсов. При этом ведется масса всяческой статистики, на что уходит немало памяти. В адресное пространство загружаются EXE- и DLL-файлы, а значит, нужны файловые ресурсы. С другой стороны, потоку требуются лишь соответствующий объект ядра и стек, объем статистических сведений о потоке невелик и много памяти не занимает.[[2]](#footnote-3)

Для создания потоков можно использовать функцию \_beginthread. Она имеет следующий прототип:

uintptr\_t \_beginthread(

void( \_\_cdecl \*start\_address )( void \* ),

unsigned stack\_size,

void \*arglist

);

start\_address - начальный адрес процедуры, который начинает выполнение нового потока.

stack\_size - размер стека для нового потока или 0.

arglist список аргументов, который должен быть передан новому потоку, или NULL.[[3]](#footnote-4)

## 1.3. Многозадачность

Многозадачная операционная система делит доступное процессорное время между процессами или потоками, которым оно необходимо. Современные системы Windows, Mac OS, Linux, iOS, Android поддерживают многозадачность. Windows в современной версии распределяет кванты времени на каждый поток, который выполняется. Выполняемый в данное время поток приостанавливается, когда истекает квант его времени, позволяя начать исполнение другого потока. Когда система переключается от одного потока к другому, она сохраняет контекст потока и восстанавливает сохраненный контекст следующего потока в куче.

Длина кванта времени зависит от операционной системы и процессора. Поскольку каждый квант времени мал (около 20 мс),[[4]](#footnote-5) пользователю кажется, что многие потоки выполняются одновременно.

Для пользователя преимущества заключаются в том, что многозадачность – это возможность использовать несколько приложений одновременно. Например, пользователь может редактировать файл в одном приложении, пока другое просчитывает таблицу.

Для разработчика приложений многозадачность предоставляет возможность создавать приложения, которые используют более одного процесса и создавать процессы, использующие более одного потока выполнения. Например, процесс может иметь поток пользовательского интерфейса, который управляет взаимодействием с пользователем (ввод с клавиатуры или мыши) и рабочие потоки, которые выполняют другие задачи, пока пользовательский интерфейс ждет пользовательского ввода. Если вы даете потоку пользовательского интерфейса высший приоритет, приложение будет более чувствительно к пользователю, пока рабочие потоки используют процессор во время, когда нет пользовательского ввода.

Существует две формы многозадачности: процессная и потоковая. Основанная на процессах многозадачность управляет конкурентным исполнением программ. Поточная многозадачность управляет одновременным исполнением частей одной и той же программы.

С/C++ не содержит встроенной поддержки многопоточности, вместо этого он опирается на операционную систему.

При одновременном доступе нескольких процессов (или нескольких потоков одного процесса) к какому-либо ресурсу возникает проблема синхронизации. Поскольку поток в Win32 может быть остановлен в любой, заранее ему неизвестный момент времени, возможна ситуация, когда один из потоков не успел завершить модификацию ресурса (например, отображенной на файл области памяти), но был остановлен, а другой поток попытался обратиться к тому же ресурсу. В этот момент ресурс находится в несогласованном состоянии, и последствия обращения к нему могут быть самыми неожиданными — от порчи данных до нарушения защиты памяти.

Классическим примером взаимодействия процессов и потоков в многопоточном приложении является задача "читатели-писатели", которая моделирует доступ к базе данных. По условию задачи, несколько процессов "писателей" делают запись в общем буфере, из которого читают несколько процессов "читателей". Любое число читателей может одновременно иметь доступ к буферу, но только один писатель может работать с ним в каждый момент времени.

Далее рассмотрим средства, с помощью которых реализуется многозадачность в операционной системе Windows.

# Глава 2. Объекты синхронизации и функции ожидания в операционной системе Windows.

В операционных системах Windows объектами синхронизации называются объекты ядра, которые могут находиться в одном из двух состояний: сигнальном (signaled) и несигнальном (nonsignaled). Объекты синхронизации могут быть разбиты на несколько классов.

К первому классу относятся объекты синхронизации, которые служат только для решения задачи синхронизации параллельных потоков. К таким объектам синхронизации в Windows относятся:

* событие (event);
* мьютекс (mutex);
* семафор (semaphore).

Ко второму классу объектов синхронизации относится ожидающий таймер (waitable timer), который переходит в сигнальное состояние по истечении заданного интервала работы.

К третьему классу объектов синхронизации относятся объекты, которые переходят в сигнальное состояние по завершении своей работы или при получении некоторого сообщения. Примерами таких объектов синхронизации являются потоки и процессы. Пока эти объекты выполняются, они находятся в несигнальном состоянии. Если выполнение этих объектов заканчивается, то они переходят в сигнальное состояние.[[5]](#footnote-6)

**2. 1. Функции ожидания.**

Функции ожидания в Windows - это такие функции, параметрами которых являются объекты синхронизации. Эти функции обычно используются для блокировки потоков. Рассмотрим две основные функции ожидания: WaitForSingleObject (функция, ожидающая единственный объект) и WaitForMultipleObject (функция, ожидающая несколько объектов).

Для ожидания перехода в сигнальное состояние одного объекта синхронизации используется функция WaitForSingleObject, которая имеет следующий прототип:

DWORD WaitForSingleObject(

HANDLE hHandle, // дескриптор объекта

DWORD dwMilliseconds // интервал ожидания в миллисекундах

);

Функция WaitForSingleObject в течение интервала времени, равного значению параметра dwMilliseconds, ждёт перехода объекта синхронизации, дескриптор которого задаётся параметром hHandle, в сигнальное состояние. Если значение параметра dwMilliseconds равно нулю, то функция только поверяет состояние объекта синхронизации. Если же значение параметра dwMilliseconds равно INFINITE, то функция ждёт перехода объекта синхронизации в сигнальное состояние бесконечно долго.

В случае успешного завершения функция WaitForSingleObject возвращает одно из следующих значений:

* WAIT\_OBJECT\_0 - объект перешёл в сигнальное состояние;
* WAIT\_ABANDONED - забытый мьютекс;
* WAIT\_TIMEOUT - время ожидания истекло.

Значение WAIT\_OBJECT\_0 означает, что объект синхронизации находился или перешёл в сигнальное состояние.

Значение WAIT\_ABANDONED означает, что объектом синхронизации являлся мьютекс, который не освободился завершившимся потоком. В этом случае мьютекс освобождается операционной системой и потому также переходит в сигнальное состояние. Такой мьютекс называется забытый или заброшенный мьютекс (abandoned mutex).

Значение WAIT\_TIMEOUT означает, что время ожидания истекло, а объект синхронизации так и не перешёл в сигнальное состояние.

В случае неудачи функция WaitForSingleObject возвращает значение WAIT\_FAILED.

Для ожидания перехода в сигнальное состояние нескольких объектов синхронизации или одного из нескольких объектов синхронизации используется функция WaitForMultipleObject, которая имеет следующий прототип:

DWORD WaitForMultipleObjects(

DWORD nCount, // количество объектов

CONST HANDLE \*lpHandles, // массив дескрипторов объектов

BOOL bWaitAll, // режим ожидания

DWORD dwMilliseconds // интервал ожидания в миллисекундах

);

Функция WaitForMultipleObject работает следующим образом. Если значение параметра bWaitAll равно TRUE, то эта функция в течение интервала времени, равного значению dwMilliseconds, ждёт пока все объекты синхронизации, дескрипторы которых заданы в массиве lpHandles, перейдут в сигнальное состояние. Если же значение параметра bWaitAll равно FALSE, то эта функция в течение заданного интервала времени ждёт, пока любой из заданных объектов синхронизации перейдёт в сигнальное состояние. Если значение параметра dwMilliseconds равно нулю, то функция только поверяет состояние объектов синхронизации. Если же значение параметра dwMilliseconds равно INFINITE, то функция ждёт перехода объектов синхронизации в сигнальное состояние бесконечно долго.

**2. 2. Событие (Event)**

Событием называется оповещение о некотором выполненном действии. В программировании события используются для оповещения одного потока о том, что другой поток выполнил некоторое действие. Сама же задача оповещения одного потока о некотором действии, которое совершил другой поток называется задачей условной синхронизации или иногда задачей оповещения. В операционных системах Windows события описываются объектами ядра Events. При этом различают два типа событий:

* события с ручным сбросом;
* события с автоматическим сбросом.

Различие между этими типами событий заключается в том, что событие с ручным сбросом можно перевести в несигнальное состояние только посредством вызова функции ResetEvent, а событие с автоматическим сбросом переходит в несигнальное состояние как при помощи функции ResetEvent, так и при помощи функции ожидания. При этом отметим, что если события с автоматическим сбросом ждут несколько потоков, используя функцию WaitForSingleObject, то из состояния ожидания освобождается только один из этих потоков.

Создаются события вызовом функции CreateEvent, которая имеет следующий прототип:

HANDLE CreateEvent (

LPSECURITY\_ATTRIBUTES lpSecurityAttributes, //атрибуты защиты

BOOL bManualReset /\*Тип события. Определяет, будет ли Event переключаемым вручную (TRUE) или автоматически (FALSE)\*/

BOOL bInitialState, /\*Задает начальное состояние. Если TRUE, то объект изначально находится в сигнальном состоянии\*/

LPCTSTR lpName //Имя события (или NULL, если имя не требуется)

);

Основную смысловую нагрузку несут второй и третий параметры. Если значение параметра равно TRUE, то создаётся событие с ручным сбросом, в противном случае - с автоматическим сбросом. Если значение параметра bInitialState равно TRUE, то начальное состояние события является сигнальным, в противном случае - несигнальным. Параметр lpName задаёт имя события, которое позволяет обращаться к нему из потоков, выполняющихся в разных процессах. Этот параметр может быть равен NULL, тогда создаётся безымянное событие.

В случае удачного завершения функция CreateEvent возвращает дескриптор события, а в случае неудачи - значение NULL. Если событие с заданным именем уже существует, то функция CreateEvent возвращает дескриптор этого события, а функция GetLastError, вызванная после функции CreateEvent, вернёт значение ERROR\_ALREADY\_EXISTS.

Для перевода любого события в сигнальное состояние используется функция SetEvent, которая имеет следующий прототип:

BOOL SetEvent (

HANDLE hEvent //дескриптор события

);

При успешном завершении эта функция возвращает ненулевое значение, а в случае неудачи - FALSE.

Функция ResetEvent используется для перевода любого события в несигнальное состояние. Эта функция имеет следующий прототип:

BOOL ResetEvent (

HANDLE hEvent //дескриптор события

);

При успешном завершении эта функция возвращает ненулевое значение, а в случае неудачи - FALSE.

Доступ к существующему событию можно открыть с помощью одной из функций CreateEvent или OpenEvent. Если для этой цели используется функция CreateEvent, то значения параметров bManualReset и bInitialState этой функции игнорируются, так

как они уже установлены другим потоком, а поток, вызвавший эту функцию, получает полный доступ к событию с именем, заданным параметром lpName.

Теперь рассмотрим функцию OpenEvent, которая используется в случае, если известно, что событие с заданным именем уже существует. Эта функция имеет следующий прототип:

HANDLE OpenEvent (

DWORD dwDesiredAccess, // флаги доступа

BOOL bInheritHandle, // режим наследования

LPCTSTR lpName // имя события

);

Параметр dwDesiredAccess определяет доступ к событию, и он может быть равен любой логической комбинации следующих флагов:

* EVENT\_ALL\_ACCESS
* EVENT\_MODIFY\_STATE
* SYNCHRONIZE

Флаг EVENT\_ALL\_ACCESS означает, что поток может выполнять над событием любые действия. Флаг EVENT\_MODIFY\_STATE означает, что поток может использовать функции SetEvent и ResetEvent для изменения состояния события. Флаг SYNCHRONIZE означает, что поток может использовать событие в функциях ожидания.

**2.3. Мьютекс**

Для решения проблемы взаимного исключения между параллельными потоками, выполняющимися в контексте разных процессов, в операционных системах Windows используется объект ядра мьютекс. Слово мьютекс является переводом английского слова mutex, которое в свою очередь является сокращением от выражения mutual exclusion, что на русском языке значит взаимное исключение. Мьютекс находится в сигнальном состоянии, если он не принадлежит ни одному потоку. В противном случае мьютекс находится в несигнальном состоянии. Одновременно мьютекс может принадлежать только одному потоку.

Потоки, ждущие сигнала мьютекса, обслуживаются в порядке FIFO, т.е. потоки становятся в очередь к мьютексу с дисциплиной обслуживания "первый пришёл - первый вышел".

Создается мьютекс вызовом функции CreateMutex, которая имеет следующий прототип:

HANDLE CreateMutex(

LPSECURITY\_ATTRIBUTES lpMutexAttributes, // атрибуты защиты

BOOL bInitialOwner, // начальный владелец мьютекса

LPCTSTR lpName // имя мьютекса

);

Если значение параметра bInitialOwner равно TRUE, то мьютекс сразу переходит во владение потоку, которым он был создан. В противном случае вновь созданный мьютекс свободен. Поток, создавший мьютекс, имеет все права доступа к этому мьютексу.

Значение параметра lpName определяет уникальное имя мьютекса для всех процессов, выполняющихся под управлением операционной системы. Это позволяет обращаться к мьютексу из других процессов, запущенных под управением этой же операционной системы. Значение параметра lpName может быть пустой указатель NULL. В этом случае система создаёт безымянный мьютекс. Следует также отметить, что имена мьютексов являются чувствительными к нижнему и верхнему регистрам.

Мьютекс захватывается потоком посредством любой функции ожидания, а освобождается функцией ReleaseMutex, которая имеет следующий прототип:

BOOL ReleaseMutex (

HANDLE hMutex // дескриптор мьютекса

);

В случае успешного завершения функция ReleaseMutex возвращает значение TRUE, в случае неудачи – FALSE. Если поток освобождает мьютекс, которым он не владеет, то функция ReleaseMutex возвращает значение FALSE.

Для доступа к существующему мьютексу поток может использовать одну из функций CreateMutex или OpenMutex. Функция CreateMutex используется в тех случаях, когда поток не знает, создан или нет мьютекс с указанным именем другим потоком. В этом случае значение параметра bInitialOwner нужно установить в FALSE, так как невозможно определить какой из потоков создает мьютекс. Если поток использует для доступа к уже созданному мьютексу функцию CreateMutex, то он получает полный доступ к этому мьютексу. Для того чтобы получить доступ к уже созданному мьютексу, поток может также использовать функцию OpenMutex, которая имеет следующий прототип:

HANDLE OpenMutex(

DWORD dwDesiredAccess, // доступ к мьютексу

BOOL bInheritHandle // свойство наследования

LPCTSTR lpName // имя мьютекса

);

Параметр dwDesiredAccess может принимать одно из двух значений:

* MUTEX\_ALL\_ACCESS - полный доступ;
* SYNCHRONIZE - синхронизация.

В первом случае поток получает полный доступ к мьютексу. Во втором случае поток может использовать мьютекс только в функциях ожидания, чтобы захватить мьютекс или в функции ReleaseMutex для его освобождения.

Параметр bInheritHandle определяет свойство наследования мьютекса. Если значение этого параметра равно TRUE, то дескриптор открываемого мьютекса является наследуемым. В противном случае дескриптор не наследуется.

В случае удачного завершения функция OpenMutex возвращает дескриптор открытого мьютекса, а в случае неудачи возвращает значение NULL.

# Глава 3. Задача о читателях-писателях.

**3.1. Постановка задачи.**

Задача о читателях-писателях — одна из важнейших задач параллельного программирования. Формулируется она так:

Есть область памяти, позволяющая чтение и запись. Несколько потоков имеют к ней доступ, при этом одновременно могут читать сколько угодно потоков, но писать — только один. Как обеспечить такой режим доступа?

Произвольное число потоков пытается получить доступ к некоему разделяемому ресурсу. Каким-то потокам («писателям») нужно модифицировать данные, а каким-то («читателям») — лишь прочесть эти данные. Синхронизация такого процесса необходима хотя бы потому, что должны быть выполнены следующие правила:

1. Когда один поток что-то пишет в область общих данных, другие этого делать не могут;
2. Когда один поток что-то пишет в область общих данных, другие не могут ни чего считывать оттуда;
3. Когда один поток считывает что-то из области общих данных, другие не могут туда ничего записывать;
4. Когда один поток считывает что-то из области общих данных, другие тоже могут это делать.

Посмотрим на проблему в контексте базы данных. Правило 1 необходимо потому, что мы, конечно же, не можем позволить одновременно обновлять одну и ту же запись. Иначе информация в записи будет повреждена.

Правило 2 запрещает доступ к записи, обновляемой в данный момент другим пользователем. Будь то иначе, один пользователь считывал бы запись, когда другой пользователь изменял бы ее содержимое.

Правило 3 служит тем же целям, что и правило 2. И действительно, какая разница, кто первый получит доступ к данным: тот, кто записывает, или тот, кто считывает, — все равно одновременно этого делать нельзя.

И, наконец, последнее правило. Оно введено для большей эффективности работы баз данных. Если никто не модифицирует записи в базе данных, все пользователи могут свободно читать любые записи. [[6]](#footnote-7)

**3.2. Монитор Хоара.**

Рассмотрим некоторый ресурс, который разделяется между процесса­ми каким-либо планировщиком. Каждый раз, когда процесс желает полу­чить в свое распоряжение какие-то ресурсы, он должен обратиться к планировщику. Этот планировщик должен иметь переменные, с помощью которых он отслеживает, занят ресурс или свободен. Процедуру планировщика разделя­ют все процессы, и каждый процесс может в любой момент захотеть обратиться к планировщику. Но планировщик не в состоянии обслуживать более одного про­цесса одновременно. Такая процедура-планировщик и представляет собой при­мер монитора.

Таким образом, монитор — это механизм организации параллелизма, который содержит как данные, так и процедуры, необходимые для реализации динамиче­ского распределения конкретного общего ресурса или группы общих ресурсов.[[7]](#footnote-8) Процесс, желающий получить доступ к разделяемым переменным, должен обра­титься к монитору, который либо предоставит доступ, либо откажет в нем. Необ­ходимость входа в монитор с обращением к какой-либо его процедуре (напри­мер, с запросом на выделение требуемого ресурса) может возникать у многих процессов. Однако вход в монитор находится под жестким контролем — здесь осуществляется взаимоисключение процессов, так что в каждый момент време­ни только одному процессу разрешается войти в монитор. Процессам, которые хотят войти в монитор, когда он уже занят, приходится ждать, причем режимом ожидания автоматически управляет сам монитор. При отказе в доступе монитор блокирует обратившийся к нему процесс и определяет условие, по которому процесс ждет. Проверка условия выполняется самим монитором, который и де­блокирует ожидающий процесс. Поскольку механизм монитора гарантирует взаимоисключение процессов, отсутствуют серьезные проблемы, связанные с ор­ганизацией параллельных взаимодействующих процессов.

Внутренние данные монитора могут быть либо глобальными (относящимися ко всем процедурам монитора), либо локальными (относящимися только к одной конкретной процедуре). Ко всем этим данным можно обращаться только изнут­ри монитора; процессы, находящиеся вне монитора и, по существу, только вызы­вающие его процедуры, просто не могут получить доступ к данным монитора. При первом обращении монитор присваивает своим переменным начальные зна­чения. При каждом последующем обращении используются те значения пере­менных, которые сохранились от предыдущего обращения.

Если процесс обращается к некоторой процедуре монитора и обнаруживается, что соответствующий ресурс уже занят, эта процедура монитора выдает команду ожидания с указанием условия ожидания. Процесс мог бы оставаться внут­ри монитора, однако это противоречит принципу взаимоисключения, если в мо­нитор затем вошел бы другой процесс. Поэтому процесс, переводящийся в ре­жим ожидания, должен вне монитора ждать того момента, когда необходимый ему ресурс освободится.

Со временем процесс, который занимал данный ресурс, обратится к монитору, чтобы возвратить ресурс системе. Соответствующая процедура монитора при этом может просто принять уведомление о возвращении ресурса, а затем ждать, пока не поступит запрос от другого процесса, которому потребуется этот ресурс. Од­нако может оказаться, что уже имеются процессы, ожидающие освобождения данного ресурса. В этом случае монитор выполняет команду извещения (сигна­лизации), чтобы один из ожидающих процессов мог получить данный ре­сурс и покинуть монитор. Если процесс сигнализирует о возвращении (иногда называемом освобождением) ресурса и в это время нет процессов, ожидающих данного ресурса, то подобное оповещение не вызывает никаких других последст­вий кроме того, что монитор, естественно, вновь вносит ресурс в список свобод­ных. Очевидно, что процесс, ожидающий освобождения некоторого ресурса, дол­жен находиться вне монитора, чтобы другой процесс имел возможность войти в монитор и возвратить ему этот ресурс.

Чтобы гарантировать, что процесс, находящийся в ожидании некоторого ресур­са, со временем получит этот ресурс, делается так, что ожидающий процесс име­ет более высокий приоритет, чем новый процесс, пытающийся войти в мони­тор. В противном случае новый процесс мог бы перехватить ожидаемый ресурс до того, как ожидающий процесс вновь войдет в монитор. Если допустить много­кратное повторение подобной нежелательной ситуации, то ожидающий процесс мог бы откладываться бесконечно. Однако для систем реального времени можно допустить использование дисциплины обслуживания на основе абсолютных или динамически изменяемых приоритетов.

Использование монитора в качестве основного средства синхронизации и связи освобождает процессы от необходимости явно разделять между собой информа­цию. Напротив, доступ к разделяемым переменным всегда ограничен телом мо­нитора, и поскольку мониторы входят в состав ядра операционной системы, то разделяемые переменные становятся системными переменными. Это автомати­чески исключает критические интервалы (так как в каждый момент монитором может пользоваться только один процесс, то два процесса никогда не могут по­лучить доступ к разделяемым переменным одновременно).

Монитор является пассивным объектом в том смысле, что это не процесс; его процедуры выполняются только по требованию процесса.

**3.3. Решение задачи.**

Для решения задачи читателей-писателей, в которой будет решаться проблема синхронизации параллельных потоков, мы будем использовать монитор Хоара, описанный выше. Программу будем писать на языке С++.

Прежде всего, для работы с монитором нам необходимо создать параллельные потоки, имитирующие работу писателей и читателей. Количество писателей определяется переменной number\_of\_writers, и количество читателей - переменной number\_of\_readers (в нашей программе: 4 и 3 соответственно).

В качестве разделяемого ресурса выступает целочисленная переменная resourse. При каждом обращении к ней читатели должны считывать ее значение, а писатели увеличивать её на единицу. Считывание проходит одновременно всеми читателями, а запись может выполняться только одним писателем в порядке очереди.

Для доступа к разделяемой переменной (ресурсу) в программе будут использованы события (Event). Для читателей это can\_read\_event (со сбросом вручную), а для писателей - can\_write\_event (с автоматическим сбросом). Читатели будут ожидать завершение работы всех ждущих писателей (функция start\_read), а писатели будут ожидать как завершение работы читателей, так и завершение работы писателей, стоящих в очереди (функция start\_write).

Также в программе будет использован мьютекс rw\_mutex для защиты разделяемого ресурса - чтобы гарантировать, что одновременно два писателя не получат к нему доступ.

Функции stop\_read и stop\_write осуществляют проверки состояния монитора на момент завершения операций считывания или записи ресурсов и, инициируют соответствующие события (функции SetEvent), которые запускают потоки, ожидающие в очереди.

Программа завершает свою работу при условии, что ресурс достиг максимальной отметки (значения) maximum\_resourse.

## 

## 3.4. Код программы на языке С++

#include"stdafx.h"

#include"stdio.h"

#include<conio.h>

#include<iostream>

#include<tchar.h>

#include<Windows.h>

#include<process.h>

using namespace std;

const int number\_of\_readers = 3;

const int number\_of\_writers = 4;

const int wait\_time\_writers = 20;

const int wait\_time\_readers = 20;

const int maximum\_resourse = 500;

bool writerlock = 0;

int queue\_canwrite = 0;

int queue\_canread = 0;

int readers = 0;

int resourse = 0; //разделяемая переменная

HANDLE can\_read\_event;//событие, пропускающее читателей (со сбросом вручную)

HANDLE can\_write\_event; //событие, пропускающее писателей (с автосбросом)

HANDLE rw\_mutex; //мьютекс, предоставляющий монопольный доступ к ресурсу

void start\_read() //открываем работу читателей

{

InterlockedIncrement((LONG\*)&queue\_canread);

if (writerlock || queue\_canwrite)

{

WaitForSingleObject(can\_read\_event, INFINITE);

}

InterlockedDecrement((LONG\*)&queue\_canread);

InterlockedIncrement((LONG\*)&readers);

}

void stop\_read() //закрываем работу читателей

{

InterlockedDecrement((LONG\*)&readers);

if (!queue\_canread)

{

ResetEvent(can\_read\_event);

SetEvent(can\_write\_event);

}

}

void start\_write() //открываем работу писателей

{

InterlockedIncrement((LONG\*)&queue\_canwrite);

if (readers || writerlock)

{

WaitForSingleObject(can\_write\_event, INFINITE);

}

InterlockedDecrement((LONG\*)&queue\_canwrite);

InterlockedIncrement((LONG\*)&writerlock);

}

void stop\_write() //закрываем работу писателей

{

InterlockedDecrement((LONG\*)&writerlock);

if (queue\_canread)

{

SetEvent(can\_read\_event);

}

else

{

SetEvent(can\_write\_event);

}

}

void read(void\*parametr)

{

int \* number = (int \*)parametr;

while (resourse < maximum\_resourse)

{

Sleep(rand() % wait\_time\_readers);

start\_read();

printf("Hello! I`m reader. My number is: %d; Resourse = %d\n", \*number, resourse);

stop\_read();

}

}

void write(void\*parametr)

{

int \* number = (int \*)parametr;

while (resourse < maximum\_resourse)

{

Sleep(rand() % wait\_time\_writers);

start\_write();

WaitForSingleObject(rw\_mutex, INFINITE);

resourse++;

printf("Hello! I`m writer. My number is: %d; Resourse = %d\n", \*number, resourse);

ReleaseMutex(rw\_mutex);

stop\_write();

}

}

int main(){

//нумеруем читателей и писателей

int rw\_count = number\_of\_readers + number\_of\_writers;

int \*read\_write\_num = new int[number\_of\_readers + number\_of\_writers];

for (int i = 0; i < rw\_count; i++)

{

read\_write\_num[i] = i;

}

//создаем мьютекс и события

rw\_mutex = CreateMutex(NULL, FALSE, NULL); //создаем мьютекс

can\_read\_event = CreateEvent(NULL, TRUE, TRUE, NULL); //сброс вручную

can\_write\_event = CreateEvent(NULL, FALSE, TRUE, NULL); // с автосбросом

//запуск читателей и писателей

HANDLE hThread[number\_of\_readers + number\_of\_writers];

for (int i = 0; i < rw\_count; i++)

{

void \*parametr = &read\_write\_num[i];

if (i < number\_of\_readers)

hThread[i] = (HANDLE)\_beginthread(read, 0, parametr);

else

hThread[i] = (HANDLE)\_beginthread(write, 0, parametr);

}

\_getch();

return 0;

}

# Заключение

В современных условиях развития информационных технологий, многопоточные программы, несомненно, имеют явное преимущество перед однопоточными. Они позволяют более эффективно решать различные прикладные задачи, обеспечивают более высокое быстродействие, а также предоставляют программисту больше свободы для взаимодействия с пользователем. Однако с точки зрения архитектуры, подобные программы, несомненно, являются более сложными в проектировании и написании, поскольку неизбежно возникает необходимость решать вопрос взаимодействия между различными потоками и процессами.

В работе была рассмотрена многозадачность операционной системы как принцип, позволяющий реализовать одновременное выполнение нескольких программ на одном процессоре. Показаны механизмы реализации многозадачность средствами ядра операционной системы. Рассмотрены объекты синхронизации ядра операционной системы Windows: событие, мьютекс, а также функции ожидания этих событий. Приведена классическая задача «читатели-писатели» и реализовано её решение с помощью монитора Хоара и объектов синхронизации Windows (события и мьютекс) на языке программирования C++.

# Список литературы

1. AboutProcessesandThreads. MSDN (электронный ресурс) http://msdn.microsoft.com

/en-us/library/windows/desktop/ms681917(v=vs.85).aspx

1. Advantages of Multitasking. MSDN (электронный ресурс) http://msdn.microsoft.com/

en-us/library/windows/desktop/ms681940(v=vs.85).aspx

1. \_beginthread, \_beginthreadex. MSDN(электронный ресурс) https://msdn.microsoft.com

/ru-ru/library/kdzttdcb.aspx

1. C++ Multithreading (электронный ресурс) http://www.tutorialspoint.com/

cplusplus/cpp\_multithreading.htm

1. MutexObjects. MSDN (электронный ресурс) <http://msdn.microsoft.com/ru-ru/library/windows/desktop/ms684266(v=vs.85).aspx>
2. Multitasking. MSDN (электронный ресурс) <http://msdn.microsoft.com/en-us/library/windows/desktop/ms684259(v=vs.85).aspx>
3. UsingMutexObjects. MSDN (электронный ресурс) <http://msdn.microsoft.com/ru-ru/library/windows/desktop/ms686927(v=vs.85).aspx>
4. Windows. Википедия (электронный ресурс) https://ru.wikipedia.org/wiki/Windows
5. Анатолий Тенцер. Синхронизация процессов при работе с Windows. (электронный ресурс) <http://compress.ru/article.aspx?id=11782>
6. Задача о читателях-писателях (электронный ресурс) https://ru.wikipedia.org/wiki/

Задача\_о\_читателях-писателях

1. Классические задачи синхронизации (электронный ресурс)// http://itcentre.ru/

programming/science-work/parallel-programming/368/

1. Корнелюк В.В. Учебно-методический комплекс по учебной дисциплине "Информационные системы" для специальности "Прикладная информатика", Минск, 2015.
2. Одинцов И. О. Профессиональное программирование. Системный подход.— 2-е изд. перераб. и доп. — СПб.: БХВ-Петербург, 2004. — 624 с.: ил..
3. Побегайло А.П. Системное программирование в Windows. - СПБ .:БХВ-Петербург 2006, 1056с.: ил.
4. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64 разрядной версии Windows / Пер. с англ. — 4-е изд. — Спб.: Питер; М.: Издательство «Русская Редакция»; 2008. — 720 стр.: ил.
5. Синхронизация процессов при помощи событий и мьютексов (электронный ресурс) // ВГКС. Кафедра ПОСТ <http://www.mediascan.by/index.files/p4.pdf>
6. Таненбум Э. Современные операционные системы. 3-е изд. – СПб.: Питер, 2010. – 1120 с.: ил.

1. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64 разрядной версии Windows / Пер. с англ. — 4-е изд. — Спб.: Питер; М.: Издательство «Русская Редакция»; 2008. — 720 стр.: ил, с. 48. [↑](#footnote-ref-2)
2. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64 разрядной версии Windows / Пер. с англ. — 4-е изд. — Спб.: Питер; М.: Издательство «Русская Редакция»; 2008. — 720 стр.: ил, с. 130. [↑](#footnote-ref-3)
3. https://msdn.microsoft.com/ru-ru/library/kdzttdcb.aspx [↑](#footnote-ref-4)
4. https://ru.wikipedia.org/wiki/Windows [↑](#footnote-ref-5)
5. Побегайло А.П. Системное программирование в Windows. - СПБ .:БХВ-Петербург 2006, 1056с.: ил., с. 115. [↑](#footnote-ref-6)
6. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64 разрядной версии Windows / Пер. с англ. — 4-е изд. — Спб.: Питер; М.: Издательство «Русская Редакция»; 2008. — 720 стр.: ил, с. 268. [↑](#footnote-ref-7)
7. Одинцов И. О. Профессиональное программирование. Системный подход.— 2-е изд. перераб. и доп. — СПб.: БХВ-Петербург, 2004. — 624 с.: ил.. с. 481 [↑](#footnote-ref-8)