Лабораторная работа «Реализация монитора Хоара «Читатели-писатели» под ОС Windows» (Hoare C.A.R.)

В лабораторной работе необходимо разработать многопоточное приложение, используя API ОС Windows такие как, потоки, события (event) и мьютексы (mutex). Потоки разделяют единственную глобальную переменную. Приложение реализует монитор Хоара «Читатели-писатели».

Монитор Хоара «Читатели-писатели».

Задача «Читатели-писатели» является одной из широко известных. В ОС Linux задача «читатели-писатели» реализована с помощью r-w семафоров.

Для этой задачи характерно наличие двух типов процессов: процессов «читателей», которые могут только читать данные, и процессов «писателей», которые могут только изменять данные. Читатели могут работать параллельно, поскольку они друг другу не мешают (данные не изменяют), а писатели могут работать только в режиме монопольного доступа: только один писатель может получить доступ к разделяемой переменной, причем, когда работает писатель, то другие писатели и читатели не могут получить доступ к этой переменной. Рассмотрим монитор Хоара «Читатели-писатели», для которого характерно наличие четырех процедур: start_read(), stop_read(), start_write(), stop_write() (листинг 1).

```
RESOURCE MONITOR;
var
    active_readers : integer;
     active_writer : logical;
    can_read, can_write : conditional;
procedure star_read
    begin
        if (active_writer or turn(can_write)) then wait(can_read);
        active_readers++; //инкремент читатетей
        signal(can_read);
    end;
procedure stop_read
     begin
         active_readers--; //декремент читателей
         if (active_readers = 0) then signal(can_write);
      end;
procedure start_write
       begin
           if ((active_readers > 0) or active_writer) then wait(can_write);
           active_writer:= true;
       end;
procedure stop_write
        begin
            active_writer:= false;
            if (turn(can_read) then signal(can_read)
```

```
                else signal(can_write);
end;
begin
    active_readers:=0;
    active_writer:=false;
end.
```

Листинг 1

Когда число читателей равно 0, процесс писатель получает возможность начать работу. Новый процесс читатель не сможет начать свою работу пока работает процесс писатель и не появится истинное значение условия can_read.

Писатель может начать свою работу, когда условие can_write станет равно истине (true).

Когда процессу читателю нужно выполнить чтение, он вызывает процедуру start_read. Если читатель заканчивает читать, то он вызывает процедуру stop_read. При входе в процедуру star_read новый процесс читатель сможет начать работать, если нет процесса писателя, изменяющего данные, в которых заинтересован читатель, и нет писателей, ждущих свою очередь (turn(can_write)), чтобы изменить эти данные. Второе условие нужно для предотвращения бесконечного откладывания процессов писателей в очереди писателей.

Процедура start_read завершается выдачей сигнала signal(can_read), чтобы следующий читатель в очереди читателей смог начать чтение. Каждый следующий читатель, начав чтение выдает signal(can_read), активизирует следующего читателя в очереди читателей. В результате возникает цепная реакция активизации читателей и она будет идти до тех пор, пока не активизируются все ожидающие читатели.

«Цепная реакция» читателей является отличительной особенностью данного решения, которое эффективно «запускает» параллельное выполнение читателей.

Процедура stop_read уменьшает количество активных читателей: читателей, начавших чтение. После ее многократного выполнения количество читателей может стать равным нулю. Если число читателей равно нулю, выполняется signal(can_write), активизирующий писателя из очереди писателей.

Когда писателю необходимо выполнить запить, он вызывает процедуру start_write. Для обеспечения монопольного доступа писателя к разделяемым данным, если есть читающие процессы или другой активный писатель, то писателю придется подождать, когда будет установлено значение «истина» в переменной типа условие can_write. Когда писатель получает возможность работать логической переменной can_write присваивается значение «истина», что заблокирует доступ других процессов писателей к разделяемым данным.

Когда писатель заканчивает работу, предпочтение отдается читателям при условии, что очередь ждущих читателей не пуста. Иначе для писателей устанавливается переменная can_write. Таким образом исключается бесконечное откладывание читателей.

Потоки Windows

Функция потока может выполнять любые задачи. Рано или поздно она закончит свою работу и вернет управление. В этот момент Ваш поток остановится, память, отведенная под его стек, будет

освобождена, а счетчик пользователей его объекта ядра «поток» уменьшится на 1. Когда счетчик обнулится, этот объект ядра будет разрушен.

Но, как и объект ядра «процесс», он может жить гораздо дольше, чем сопоставленный с ним поток.

А теперь поговорим о самых важных вещах, касающихся функций потоков.

- В отличие от входной функции первичного потока, у которой должно быть одно из четырех имен: main, wmain, WinMain или wWinMain, — функцию потока можно назвать как угодно. Однако, если в программе несколько функций потоков, то необходимо присвоить им разные имена, иначе компилятор или компоновщик решит, что создаётся несколько реализаций одной функции.

- Поскольку входным функциям главного потока (процесса) передаются строковые параметры, они существуют в ANSI- и Unicode-версиях: main — wmain и WinMain — wWinMain. Но функциям потоков передается единственный параметр, смысл которого определяется разработчиком, а не операционной системой. Поэтому здесь нет проблем с ANSI/Unicode.

- Функция потока должна возвращать значение, которое будет использоваться как код завершения потока. Здесь полная аналогия с библиотекой C/C++: код завершения главного потока становится кодом завершения процесса.

- Функции потоков (да и все разработанные функции) должны по мере возможности обходиться своими параметрами и локальными переменными. Так как к статической или глобальной переменной могут одновременно обратиться несколько потоков, есть риск неверного изменения ее содержимого. Однако параметры и локальные переменные создаются в стеке потока, поэтому они в гораздо меньшей степени подвержены влиянию другого потока.

Функция CreateThread()

Для создания дополнительных потоков, нужно вызвать из первичного потока функцию CreateThread():

```
HANDLE CreateThread(
LPSECURITY_ATTRIBUTES lpsa;
DWORD cbStack;
```

LPTHREAD_START_ROUTINE lpStartAddr;
LPVOID lpvThreadParam;
DWORD fdwCreate;
LPDWORD lpIDThread);

1. При каждом вызове этой функции система создает объект ядра «поток». Структура данных используется операционной системой для управления потоком и хранит статистическую информацию о потоке. **HANDLE** (дескриптор) этого объектам – значение, которое возвращает функция **CreateThread().** Система выделяет память под стек потока в адресном пространстве процесса.
2. Инициализируется код завершения потока (регистрируемый в объекте ядра «поток») идентификатором STILL_ACTIVE и присваивает счетчику простоя потока (thread’s suspend count) единицу. Последний тоже запоминается в объекте ядра «поток».
3. Для нового потока создается структура CONTEXT.
4. Формирует стек потока, для чего резервирует в адресном пространстве процесса регион размером в две страницы.
5. Значения lpStartAddr и lpvThreadParm помещаются в стек.
6. Инициализирует регистры: указатель стека и счетчик команд в структуре CONTEXT потока.
   Параметры CreatThread():

- **lpsa –** указатель на структуру SECURITY_ATTRIBUTES. Для установки атрибутов защиты, определенных по умолчанию, в параметр записывается NULL.
- **cbStack –** определяет, какую часть адресного пространства поток может использовать под стек. Каждый поток имеет отдельный стек. Если указывается 0, то размер стека устанавливается по умолчанию.
- **lpStartAddr** – определяет адрес функции потока, с которой следует начать выполнение потока.
- **lpvThreadParm** – указатель на переменную, которая передается в поток.
- **fdwCreate** – определяет флаги, управляющие созданием потока. Может иметь одно из двух значений: 0 – исполнение потока начинается немедленно, CREATE_SUSPENDED – поток находится в состоянии ожидания.
- **lpIDThread** – адрес переменной типа DWORD, в которую функция возвращает идентификатор потока.

Новый поток выполняется в контексте того же процесса, что и родительский поток. Поэтому он

получает доступ ко всем описателям объектов ядра, всей памяти и стекам всех потоков в процессе. За счет этого потоки в рамках одного процесса могут легко взаимодействовать друг с другом.

Другими словами, поток своего адресного пространства не имеет, а выполняется в адресном пространстве процесса параллельно с другими потоками, если они созданы, и может разделять с ними глобальные переменные. Владельцем ресурсов является процесс.

Например: CreateThread(NULL, 0, &ThreadProc, &hMutex, 0, NULL);

Аргументы слева направо — (1) NULL указывает, что атрибуты защиты будут установлены по умолчанию, (2) размер стека в байтах, округляется до размера страницы, если ноль, то берется размер по умолчанию, (3) указатель на процедуру, с которой следует начать выполнение потока, (4) аргумент процедуры, переданной предыдущим аргументом, обычно здесь передается указатель на некую структуру, (5) флаги, например, можно создать приостановленный поток (CREATE_SUSPENDED), а затем запустить его с помощью ResumeThread, (6) куда записать ThreadId созданного потока.

В случае успеха процедура возвращает дескриптор созданного потока. В случае ошибки возвращается NULL, а подробности можно узнать через GetLastError.

Wait-функции

Wait-функции позволяют потоку в любой момент приостановиться и ждать освобождения какого-либо объекта ядра. Из всего семейства этих функций чаще всего используется WaitForSingleObject:

DWORD WaitForSingleObject(
HANDLE hObject,
DWORD dwMilliseconds);

Когда поток вызывает эту функцию, первый параметр, hObject, идентифицирует объект ядра, поддерживающий состояния «свободен-занят». (То есть любой объект, упомянутый в списке из предыдущего раздела.) Второй параметр, dwMilliseconds, указывает, сколько времени (в миллисекундах) поток готов ждать освобождения объекта.

Следующий вызов сообщает системе, что поток будет ждать до тех пор, пока не завершится процесс, идентифицируемый описателем hProcess:
WaitForSingleObject(hProcess, INFINITE);

В данном случае константа INFINITE, передаваемая во втором параметре, подсказывает системе, что вызывающий поток готов ждать этого события хоть целую вечность. Именно эта константа обычно и передается функции WaitForSingleObject, но Вы можете указать любое значение в миллисекундах. Кстати, константа INFINITE определена как 0xFFFFFFFF (или –1). Разумеется, передача INFINITE не всегда безопасна.

Если объект так и не перейдет в свободное состояние, вызывающий поток никогда
не проснется, но тратить процессорное время он при этом не будет.
Пример, иллюстрирующий, как вызывать WaitForSingleObject со значением тайм-
маута, отличным от INFINITE:

```
DWORD dw = WaitForSingleObject(hProcess, 5000);

switch (dw) {
case WAIT_OBJECT_0:
// процесс завершается
break;
case WAIT_TIMEOUT:
// процесс не завершился в течение 5000 мс
break;
case WAIT_FAILED:
// неправильный вызов функции (неверный описатель?)
break;
}
```

Данный код сообщает системе, что вызывающий поток не должен получать процессорное время, пока не завершится указанный процесс или не пройдет 5000 мс (в зависимости от того, что случится раньше). Поэтому функция вернет управление либо до истечения 5000 мс, если процесс завершится, либо примерно через 5000 мс, если процесс к тому времени не закончит свою работу. Заметьте, что в параметре dwMilliseconds можно передать 0, и тогда WaitForSingleObject немедленно вернет управление.

Возвращаемое значение функции WaitForSingleObject указывает, почему вызываю-
щий поток снова стал планируемым. Если функция возвращает WAIT_OBJECT_0, объект свободен, а если WAIT_TIMEOUT — заданное время ожидания (таймаут) истекло.

При передаче неверного параметра (например, недопустимого описателя) WaitForSingleObject возвращает WAIT_FAILED. Чтобы выяснить конкретную причину ошибки, вызовите функцию GetLastError.

WaitForSingleObject(hHandle, dwTimeout);

WaitForSingleObject ждет, когда объект дескриптор которого передается первым параметром, перейдет в сигнальное состояние (signaled state). Кроме того, в зависимости от типа объекта, процедура может менять его состояние. WaitForSingleObject может быть применен к событиям, мьютексам, семафорам, потокам, процессам и т.п.

Например, если hHandle представляет собой дескриптор мьютекса, процедура ждет, когда мьютекс освободится, а затем захватывает его. Если же hHandle является дескриптором потока, то процедура просто ждет его завершения. Второй аргумент задает время ожидания в миллисекундах. Можно ждать вечно, передав специальное значение INFINITE. Если указать ноль, процедура не переходит в режим ожидания, а возвращает управление немедленно.

В случае ошибки процедура возвращает WAIT_FAILED, а подробности поможет узнать GetLastError. В случае успеха возвращается WAIT_OBJECT_0, если мы дождались перехода объекта в сигнальное состояние, и WAIT_TIMEOUT, если отключились по таймауту. Также мы можем получить WAIT_ABANDONED. Это происходит в случае, если поток, удерживающий мьютекс, завершился, не

освободив его. В этом случае мьютекс становится блокированным текущей нитью, но целостность данных, доступ к которым ограничивался мьютексом, по понятным причинам находится под вопросом.

Функция WaitForMultipleObjects аналогична WaitForSingleObject с тем исключением, что позволяет ждать освобождения сразу нескольких объектов или какого-то одного из списка объектов:

```
DWORD WaitForMultipleObjects(
DWORD dwCount,
CONST HANDLE* phObjects,
BOOL fWaitAll,
DWORD dwMilliseconds);
```

Параметр dwCount определяет количество интересующих Вас объектов ядра. Его значение должно быть в пределах от 1 до MAXIMUM_WAIT_OBJECTS (в заголовочных файлах Windows оно определено как 64). Параметр phObjects — это указатель на массив описателей объектов ядра.

WaitForMultipleObjects приостанавливает поток и заставляет его ждать освобождения либо всех заданных объектов ядра, либо одного из них. Параметр fWaitAll как раз и определяет, чего именно Вы хотите от функции. Если он равен TRUE, функция не даст потоку возобновить свою работу, пока не освободятся все объекты.

Параметр dwMilliseconds идентичен одноименному параметру функции WaitForSingleObject. Если Вы указываете конкретное время ожидания, то по его истечении функция в любом случае возвращает управление. И опять же, в этом параметре обычно передают INFINITE (будьте внимательны при написании кода, чтобы не создать ситуацию взаимной блокировки).

Возвращаемое значение функции WaitForMultipleObjects сообщает, почему возобновилось выполнение вызвавшего ее потока. Значения WAIT_FAILED и WAIT_TIMEOUT никаких пояснений не требуют. Если Вы передали TRUE в параметре fWaitAll и все объекты перешли в свободное состояние, функция возвращает значение WAIT_OBJECT_0. Если же fWaitAll приравнен FALSE, она возвращает управление, как только освобождается любой из объектов. Вы, по-видимому, захотите выяснить, какой именно объект освободился. В этом случае возвращается значение от WAIT_OBJECT_0 до WAIT_OBJECT_0 + dwCount – 1. Иначе говоря, если возвращаемое значение не равно WAIT_TIMEOUT или WAIT_FAILED, вычтите из него значение WAIT_OBJECT_0, и Вы получите индекс в массиве описателей, на который указывает второй параметр функции WaitForMultipleObjects. Индекс подскажет Вам, какой объект перешел в незанятое состояние.

WaitForMultipleObjects(dwCount, lpHandles, bWaitAll, dwTimeout);

Процедура WaitForMultipleObjects работает аналогично WaitForSingleObject, но для массива объектов. Первый аргумент задает размер массива, должен быть строго больше нуля и не превышать MAXIMUM_WAIT_OBJECTS (который равняется в точности 64). Вторым аргументом задается указатель на массив дескрипторов. В массиве могут содержаться дескрипторы на объекты разных типов, но многократное включение одного и того же дескриптора считается ошибкой. Если третий аргумент равен TRUE, процедура ждет перехода в сигнальное состояние *всех* объектов, иначе — *любого* из указанных объектов. Семантика последнего аргумента точно такая же, как и в случае с WaitForSingleObject.

Процедура возвращает WAIT_FAILED в случае ошибки и WAIT_TIMEOUT в случае отваливания по таймауту. Если процедура вернула значение от WAIT_OBJECT_0 до WAIT_OBJECT_0 + dwCount - 1, то в случае bWaitAll == TRUE все объекты из массива перешли в сигнальное состояние, а в случае bWaitAll == FALSE в сигнальное состояние перешел объект с индексом код возврата минус WAIT_OBJECT_0. Если процедура вернула значение от WAIT_ABANDONED_0 до WAIT_ABANDONED_0 + dwCount - 1, то в случае bWaitAll == TRUE все ожидаемые объекты перешли в сигнальное состояние и по крайней мере один из объектов оказался брошенным мьютексом, а в случае bWaitAll == FALSE брошенным оказался мьютекс с индексом код возврата минус WAIT_ABANDONED_0.

Если bWaitAll == TRUE, процедура не меняет состояния объектов до тех пор, пока все они не перейдут в сигнальное состояние. Таким образом, пока вы ждете, мьютексы могут продолжать лочиться и анлочиться другими потоками. Если bWaitAll == FALSE и сразу несколько объектов перешли в сигнальное состояние, процедура всегда работает с объектом в массиве, имеющим минимальный индекс.

Средства взаимоисключения в Windows

События (event)

Event позволяет известить один или несколько ожидающих потоков о наступлении события. События используются потоками для сигнализации, что другой поток может выполнить какую-то работу.

Event бывает:

Со сбросом вручную	Будучи установленным в сигнальное состояние, остается в нем до тех пор, пока не будет переключен явным вызовом функции ResetEvent
С автосбросом	Автоматически переключается в несигнальное состояние операционной системой, когда один из ожидающих его потоков завершается

Для создания объекта используется функция CreateEvent:

```
function CreateEvent(
   lpEventAttributes: PSecurityAttributes;  // Адрес структуры
                                            // TSecurityAttributes
    bManualReset,          // Задает, будет Event переключаемым
                           // вручную (TRUE) или автоматически (FALSE)
    bInitialState: BOOL;   // Задает начальное состояние. Если TRUE -
                           // объект в сигнальном состоянии
    lpName: PChar          // Имя или NIL, если имя не требуется
 ): THandle; stdcall;      // Возвращает идентификатор созданного
                           // объекта
```

Структура TSecurityAttributes описана, как:

```
TSecurityAttributes = record
   nLength: DWORD;                 // Размер структуры, должен
     // инициализироваться как
```

```
                                    // SizeOf(TSecurityAttributes)
   lpSecurityDescriptor: Pointer; // Адрес дескриптора защиты. В
                                    // Windows 95 и 98 игнорируется
                                    // Обычно можно указывать NIL
   bInheritHandle: BOOL;          // Задает, могут ли дочерние
                                    // процессы наследовать объект
end;
```

Если не требуется задание особых прав доступа под Windows NT или возможности наследования объекта дочерними процессами, в качестве параметра lpEventAttributes можно передавать NIL. В этом случае объект не может наследоваться дочерними процессами и ему задается дескриптор защиты «по умолчанию».

Параметр lpName позволяет разделять объекты между процессами. Если lpName совпадает с именем уже существующего объекта типа Event, созданного текущим или любым другим процессом, то функция не создает нового объекта, а возвращает идентификатор уже существующего. При этом игнорируются параметры bManualReset, bInitialState и lpSecurityDescriptor. Проверить, был ли объект создан или используется уже существующий, можно следующим образом:

```
hEvent := CreateEvent(NIL, TRUE, FALSE, 'EventName');
if hEvent = 0 then
   RaiseLastWin32Error;
 if GetLastError = ERROR_ALREADY_EXISTS then begin
   // Используем ранее созданный объект
 end;
```

Если объект используется для синхронизации внутри одного процесса, его можно объявить, как глобальную переменную и создавать без имени.

Имя объекта не должно совпадать с именем любого из существующих типов объектов: Semaphore, Mutex, Job, Waitable Timer или FileMapping.

В случае совпадения имен функция возвращает ошибку.

Если известно, что Event уже создан, для получения доступа к нему можно вместо CreateEvent воспользоваться функцией OpenEvent:

```
function OpenEvent(
  dwDesiredAccess: DWORD;  // Задает права доступа к объекту
  bInheritHandle: BOOL;    // Задает, может ли объект наследоваться
                           // дочерними процессами
  lpName: PChar            // Имя объекта
 ): THandle; stdcall;
```

Функция возвращает идентификатор объекта или 0 — в случае ошибки. Параметр dwDesiredAccess может принимать одно из следующих значений:

EVENT_ALL_ACCESS	Приложение получает полный доступ к объекту
EVENT_MODIFY_STATE	Приложение может изменять состояние объекта функциями SetEvent и ResetEvent
SYNCHRONIZE	Только для Windows NT — приложение может использовать объект только в функциях ожидания

После получения идентификатора его можно использовать. Для этого имеются следующие функции:

function SetEvent(hEvent: THandle): BOOL; stdcall;

— устанавливает объект в сигнальное состояние

function ResetEvent(hEvent: THandle): BOOL; stdcall;

— сбрасывает объект, устанавливая его в несигнальное состояние

function PulseEvent(hEvent: THandle): BOOL; stdcall

— устанавливает объект в сигнальное состояние, дает отработать всем функциям ожидания, ожидающим этот объект, а затем снова сбрасывает его.

По завершении работы с объектом он должен быть уничтожен функцией **CloseHandle.**

Рассмотрим небольшой **пример**, как на практике для синхронизации потоков использовать объекты ядра «событие».

```
// глобальный описатель события со сбросом вручную (в занятом состоянии)
HANDLE g_hEvent;
int WINAPI WinMain(...) {
// создаётся объект "событие со сбросом вручную" (в занятом состоянии)
g_hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
// порождаются три новых потока
HANDLE hThread[3];
DWORD dwThreadID;
hThread[0] = _beginthreadex(NULL, 0, WordCount, NULL, 0, &dwThreadID);
hThread[1] = _beginthreadex(NULL, 0, SpellCheck, NULL, 0, &dwThreadID);
hThread[2] = _beginthreadex(NULL, 0, GrammarCheck, NULL, 0, &dwThreadID);
OpenFileAndReadContentsIntoMemory(...);
// разрешаем всем трем потокам обращаться к памяти
SetEvent(g_hEvent);
...
```

```
}
DWORD WINAPI WordCount(PVOID pvParam) {
// ждем, когда в память будут загружены данные из файла
WaitForSingleObject(g_hEvent, INFINITE);
// обращаемся к блоку памяти
...
return(0);
}
DWORD WINAPI SpellCheck(PVOID pvParam) {
// ждем, когда в память будут загружены данные из файла
WaitForSingleObject(g_hEvent, INFINITE);
// обращаемся к блоку памяти
...
return(0);
}
DWORD WINAPI GrammarCheck(PVOID pvParam) {
// ждем, когда в память будут загружены данные из файла
WaitForSingleObject(g_hEvent, INFINITE);
// обращаемся к блоку памяти
...
return(0);
}
```

В примере при выполнении процесс создает занятое событие со сбросом вручную и записывает его описатель в глобальную переменную. Это упрощает другим потокам процесса доступ к тому же объекту-событию. Затем порождаются три потока. Они ждут, когда в память будут загружены данные (текст) из некоего файла, и потом обращаются к этим данным: один поток подсчитывает количество слов, другой проверяет орфографические ошибки, третий — грамматические. Все три функции потоков начинают работать одинаково: каждый поток вызывает WaitForSingleObject, которая приостанавливает его до тех пор, пока первичный поток не считает в память содержимое файла.

Загрузив нужные данные, главный (первичный) поток вызывает SetEvent, которая переводит событие в свободное состояние. В этот момент система пробуждает три вторичных потока, и они, получив процессорное время, обращаются к блоку памяти. Заметим, что потоки получают доступ к памяти в режиме только для чтения. Это единственная причина, по которой все три потока могут выполняться одновременно.

Если событие со сбросом вручную заменить на событие с автосбросом, программа будет вести себя совершенно иначе. После вызова главным (первичным) потоком функции SetEvent система возобновит выполнение только одного из вторичных потоков.

Какого именно — заранее сказать нельзя. Остальные два потока продолжат ожидание.

Поток, вновь ставший планируемым (выйдя из блоктровки), получает монопольный доступ к блоку памяти, где хранятся данные, считанные из файла. Перепишем функции потоков так, чтобы непосредственно перед возвратом управления они (подобно функции WinMain) вызывали SetEvent. Теперь функции потоков выглядят следующим образом:

```
DWORD WINAPI WordCount(PVOID pvParam) {
// ждем, когда в память будут загружены данные из файла
WaitForSingleObject(g_hEvent, INFINITE);
// обращаемся к блоку памяти
M
SetEvent(g_hEvent);
return(0);
}
DWORD WINAPI SpellCheck(PVOID pvParam) {
// ждем, когда в память будут загружены данные из файла
WaitForSingleObject(g_hEvent, INFINITE);
// обращаемся к блоку памяти
...
SetEvent(g_hEvent);
return(0);
}
DWORD WINAPI GrammarCheck(PVOID pvParam) {
// ждем, когда в память будут загружены данные из файла
WaitForSingleObject(g_hEvent, INFINITE);
// обращаемся к блоку памяти
...
SetEvent(g_hEvent);
return(0);
}
```

Закончив свою работу с данными, поток вызывает SetEvent, которая разрешает системе возобновить выполнение следующего из двух ждущих потоков. И опять не предсказуемо, какой поток выберет система, но так или иначе кто-то из них получит монопольный доступ к тому же блоку памяти. Когда и этот поток закончит свою работу, он тоже вызовет SetEvent, после чего с блоком памяти сможет монопольно оперировать третий, последний поток. Обратим внимание, что использование события с **автосбросом** снимает проблему с доступом вторичных потоков к памяти как для чтения, так и для записи и при этом больше не нужно ограничивать их доступ только чтением.

Этот пример четко иллюстрирует различия в применении событий со сбросом вручную и с автосбросом.

Мьютексы (mutex)

Объекты ядра «мьютексы» гарантируют потокам взаимоисключающий доступ к единственному ресурсу. Отсюда и произошло название этих объектов (mutual exclusion, mutex). 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 это — объекты ядра. Кроме того, единственный объект-мьютекс позволяет синхронизировать доступ к ресурсу нескольких потоков, принадлежащих разным процессам. Можно задать максимальное время ожидания доступа к ресурсу.

CreateMutex(NULL, FALSE, NULL);

Процедура CreateMutex() создает новый мьютекс. О первом аргументе сейчас знать не нужно. Если второй аргумент равен TRUE, созданный мьютекс будет сразу захвачен текущим потоком, если же второй аргумент равен FALSE, создается свободный мьютекс. Третий аргумент задает имя мьютекса, если нужно создать именованный мьютекс. Если указывается NULL, то мьютекс не именуется. Возвращаемые значения такие же, как у CreateThread.

ReleaseMutex(hMutex);

ReleaseMutex() освобождает ранее захваченный мьютекс. В случае успеха процедура возвращает значение, отличное от нуля. В случае ошибки возвращается ноль, а подробности доступны через GetLastError. Важно, для предотвращения тупиков (deadlock) Windows позволяет потокам захватывать один и тот же мьютекс несколько раз, но и количество вызовов ReleaseMutex() должно быть равно количеству захватов.

Sleep(dwMilliseconds);

Процедура Sleep блокирует текущий поток на заданное количество миллисекунд. Если в качестве параметра передать INFINITE, поток будет заблокирован навсегда. Если передать ноль, поток уступает оставшуюся часть своей доли процессорного времени любой другой нити с таким же приоритетом. У процедуры нет возвращаемого значения (VOID).

ExitThread(dwCode);

ExitThread завершает текущий поток с заданным кодом возврата. Объект потока при этом переходит в сигнальное состояние. Если это был последний поток в текущем процессе, процесс завершается. Код возврата конкретного потока может быть получен с помощью GetExitCodeThread.

Пример:

```
#include <windows.h>
#define THREADS_NUMBER 10
#define ITERATIONS_NUMBER 100
#define PAUSE 10 /* ms */

DWORD dwCounter = 0;

DWORD WINAPI ThreadProc(CONST LPVOID lpParam)
 {
  CONST HANDLE hMutex = (CONST HANDLE)lpParam;
  DWORD i;
  for(i = 0; i < ITERATIONS_NUMBER; i++)
 {
    WaitForSingleObject(hMutex, INFINITE);
     dwCounter++; //разделяемый ресурс
    ReleaseMutex(hMutex);
    Sleep(PAUSE);
  }
  ExitThread(0);
}

VOID Error(CONST HANDLE hStdOut, CONST LPCWSTR szMessage) {
  DWORD dwTemp;
  TCHAR szError[256];
  WriteConsole(hStdOut, szMessage, lstrlen(szMessage), &dwTemp, NULL);
  wsprintf(szError, TEXT("LastError = %d\r\n"), GetLastError());
  WriteConsole(hStdOut, szError, lstrlen(szError), &dwTemp, NULL);
  ExitProcess(0);
}

INT main() {
  TCHAR szMessage[256];
  DWORD dwTemp, i;
  HANDLE hThreads[THREADS_NUMBER];
  CONST HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
  CONST HANDLE hMutex = CreateMutex(NULL, FALSE, NULL);
  if(NULL == hMutex) {
    Error(hStdOut, TEXT("Failed to create mutex.\r\n"));
  }

  for(i = 0; i < THREADS_NUMBER; i++) {
    hThreads[i] = CreateThread(NULL, 0, &ThreadProc, hMutex, 0, NULL);
    if(NULL == hThreads[i])
  {
      Error(hStdOut, TEXT("Failed to create thread.\r\n"));
    }
  }

  WaitForMultipleObjects(THREADS_NUMBER, hThreads, TRUE, INFINITE);
  wsprintf(szMessage, TEXT("Counter = %d\r\n"), dwCounter);
  WriteConsole(hStdOut, szMessage, lstrlen(szMessage), &dwTemp, NULL);
```

```
  for(i = 0; i < THREADS_NUMBER; i++) {
    CloseHandle(hThreads[i]);
  }
  CloseHandle(hMutex);
  ExitProcess(0);
}
```

В примере создается десять потоков, каждый из которых увеличивает глобальный счетчик на единицу в цикле (сто раз). Монопольный доступ процессов к счетчику регулируется с помощью мьютекса. Перед вызовом ExitProcess закрываются дескрипторы мьютекса и всех созданных потоков. Строго говоря, в данном конкретном случае это не требуется, но в более общем случае все дескрипторы нужно вовремя закрывать. Также следует обратить внимание, что строки «обернуты» в макрос TEXT.
Как обычно, программа может быть скомпилирована как в Visual Studio и запущена под Windows, так и с помощью MinGW и запущена под Wine.

Замечание: для реализации описанного монитора API Windows достаточно таких средств взаимоисключения как события: с авто сбросом и со сбросом вручную. Мьютекс включается в приложение искусственно (т.е. в нем нет необходимости) для знакомства в этим средством взаимоисключения.

Дополнительно: семантика Хоара и Меса

В ранних реализациях монитора (известных как семантика Хоара) оповещение условной переменной немедленно активизирует ждущий процесс и восстанавливает блокировку, тем самым гарантируется, что условие всё ещё истинно.

Реализация этого поведения сложна и очень избыточна. Также она не совместима с вытесняющей многозадачностью, когда процесс может быть прерван в произвольный момент. По этим причинам исследователи разработали множество иных семантик для условных переменных.

В современных реализациях (известных как семантика Меса) оповещение не прерывает работающий процесс, а просто переводит некоторые ждущие процессы в состояние готовности. Оповещающий процесс продолжает держать блокировку до тех пор, пока не выйдет из процедуры монитора. Побочные эффекты этого подхода в том, что оповещающий процесс не обязан соблюсти инвариант перед оповещением, а ожидающий процесс — должен повторно проверить условие, которого он дожидается. В частности, если процедура монитора включает выражение **if** *test* **then** wait(cv), другой процесс может войти в монитор после момента оповещения и изменить значение *test* до того, как ждущий процесс возобновит работу. Выражение нужно переписать так: **while** *test* **do** wait(cv), чтобы условие было пере-проверено после ожидания.

Реализации также предоставляют операцию «notifyAll», или «broadcast», которая оповещает все процессы, ждущие данное условие. Эта операция полезна, например, когда несколько процессов ждут доступности различных объёмов памяти. Освобождение памяти позволит продолжить работу кого-то из них, но планировщик не может знать, кого именно.

Примерная реализация условной переменной:

```
conditionVariable {
```

```
  int queueSize = 0;
  mutex lock;
  semaphore waiting;
  wait() {
     lock.acquire();
     queueSize++;
     lock.release();
     waiting.down();
  }
  signal() {
     lock.acquire();
     while (queueSize > 0){
        queueSize--;
        waiting.up();
     }
     lock.release();
  }
}
```

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.