ГОСУДАРСТВЕННЫЙ КОММИТЕТ СВЯЗИ, ИНФОРМАТИЗАЦИИ И ТЕЛЕКОМУНИКАЦИОННЫХ ТЕХНОЛОГИЙ
ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

КУРСОВАЯ РАБОТА

по курсу «Системное программное обеспечение»

на тему: Синхронизация параллельно выполняющихся процессов в многозадачных средах

Выполнил студент
группы № 542-12
Якубов Алишер

Преподаватель
Ф.И.О. Акбарова М.Х.

Ташкент-2014

Введение

К моменту появления персональных компьютеров в мире существовало несколько технических решений позволяющих реализовать многозадачность на больших машинах. В бывшем СССР это были машины серии ЕС и болгарские ИЗОТ. Они теоретически позволяли подключать до 255 терминалов, где каждому терминалу выделялось некоторое количество ресурсов компьютера и процессорного времени. На практике нормальная работа такого комплекса обеспечивалась при наличии не более 25-30 терминалов, или меньше при сложных задачах.

Для персональных ЭВМ многозадачность не вводилась принципиально. Ведь исходя из названия PC – “Personal Computer” предполагалось, что работать будет один человек с одной текущей задачей. В качестве операционной системы была принята переработанная система CP/M под названием MS-DOS. Она так же не предусматривала многозадачности. Основная проблема разработки многозадачной операционной системы это не реентерабельность ее функций. То есть если один процесс запустил функцию чтения файла, то другой процесс не сможет не только обращаться к файлам, но и вообще вызвать другие ее функции. Для этого необходима поддержка на уровне процессора, которая была введена с разработкой линейки 286.

Многозадачность и многопоточность 3
Режимы многозадачности 4
Многозадачность в DOS 4
Невытесняющая многозадачность 5
Реентерабельность Маньяна и последовательная очередь сообщений 7
Решения, использующие многопоточность 8
Многопоточная архитектура 10
Коллизии, возникающие при использовании потоков 11
Преимущества Windows 12
Новая усовершенствованная многопоточная программа 14
О использовании функции Sleep 15
Критический раздел 15
Объект Мютекс 18
Уведомления о событиях 19
Локальная память потока 19

Реализация многопоточности в Delphi

Многозадачность и многопоточность

Многозадачность (multitasking) – это способность операционной системы выполнять несколько программ одновременно. В основе этого принципа лежит использование операционной системой аппаратного таймера для выделения отрезков времени (time slices) для каждого из одновременно выполняемых процессов. Если эти отрезки времени достаточно малы, и машина не перегружена слишком большим числом программ, то пользователю кажется, что все эти программы выполняются параллельно.

Идея многозадачности не нова. Многозадачность реализуется на больших компьютерах типа мэйнфрэйм (mainframe), к которым подключены десятки, а иногда и сотни, терминалов. У каждого пользователя, сидящего за экраном такого терминала, создается впечатление, что он имеет эксклюзивный доступ ко всей машине. Кроме того, операционные системы мэйнфрэймов часто дают возможность пользователям перевести задачу в фоновый режим, где они выполняются в то время, как пользователь может работать с другой программой.

Для того, чтобы многозадачность стала реальностью на персональных компьютерах, потребовалось достаточно много времени. Но, кажется, сейчас мы приближаемся к эпохе использования многозадачности на ПК (PC). Как мы увидим вскоре, некоторые расширенные 16-разрядные версии Windows поддерживают многозадачность, а имеющиеся теперь в нашем распоряжении Windows NT и Windows 95 – 32-разрядные версии Windows, поддерживают кроме многозадачности еще и многопоточность (multithreading).

Многопоточность – это возможность программы самой быть многозадачной. Программа может быть разделена на отдельные потоки выполнения (threads), которые, как кажется, выполняются параллельно. На первый взгляд эта концепция может показаться едва ли полезной, но оказывается, что программы могут использовать многопоточность для выполнения протяженных во времени операций в фоновом режиме, не вынуждая поль-

зователя надолго отрываться от машины.

Режимы многозадачности

На заре использования персональных компьютеров некоторые отстаивали идею многозадачности для будущего, но многие ломали головы над вопросом: какая польза от многозадачности на однопользовательской машине? В действительности оказалось, что многозадачность – это именно то, что необходимо пользователям, даже не подозревавшим об этом.

Многозадачность в DOS

Микропроцессор Intel 8088, использовавшийся в первых ПК, не был специально разработан для реализации многозадачности. Частично проблема (как было показано в предыдущей главе) заключалась в недостатках управления памятью. В то время, как множество программ начинает и заканчивает свое выполнение, многозадачная операционная система должна осуществлять перемещение блоков памяти для объединения свободного пространства. На процессоре 8088 это было невозможно реализовать в стиле, прозрачном для приложений.

Сама DOS не могла здесь чем-либо существенно помочь. Будучи разработанной таким образом, чтобы быть маленькой и не мешать приложениям, DOS поддерживала, кроме загрузки программ и обеспечения им доступа к файловой системе, еще не так много средств.

Тем не менее, творческие программисты, работавшие с DOS на заре ее появления, нашли путь преодоления этих препятствий, преимущественно при использовании резидентных (terminate-and-stay-resident, TSR) программ. Некоторые TSR-программы, такие как спулер печати, использовали прерывание аппаратного таймера для выполнения процесса в фоновом режиме. Другие, подобно всплывающим (popup) утилитам, таким как SideKick, могли выполнять одну из задач переключения – приостановку выполнения приложения на время работы утилиты. DOS также была усовершенствована для обеспечения поддержки резидентных программ.

Некоторые производители программного обеспечения пытались соз-

дать многозадачные оболочки или оболочки, использующие переключение между задачами, как надстройки над DOS (например, Quarterdeck's DeskView), но только одна из этих оболочек получила широкое распространение на рынке. Это, конечно. Windows.

Невытесняющая многозадачность

Когда Microsoft выпустила на рынок Windows 1.0 в 1985 году, это было еще в большой степени искусственным решением, придуманным для преодоления ограничений MS DOS. В то время Windows работала в реальном режиме (real mode), но даже тогда она была способна перемещать блоки физической памяти (одно из необходимых условий многозадачности) и делала это, хотя и не очень прозрачно для приложений, но все-таки вполне удовлетворительно.

В графической оконной среде многозадачность приобретает гораздо больший смысл, чем в однопользовательской операционной системе, использующей командную строку. Например, в классической операционной системе UNIX, работающей с командной строкой, существует возможность запускать программы из командной строки так, чтобы они выполнялись в фоновом режиме. Однако, любой вывод на экран из программы должен быть переадресован в файл, иначе этот вывод смешается с текущим содержимым экрана.

Оконная оболочка позволяет нескольким программам выполняться совместно, разделяя один экран. Переключение вперед и назад становится тривиальным, существует возможность быстро передавать данные из одной программы в другую, например, разместить картинку, созданную в программе рисования, в текстовом файле, образованном с помощью текстового процессора. Передача данных поддерживалась в различных версиях Windows: сначала с использованием папки обмена (clipboard), позднее – посредством механизма динамического обмена данными (Dynamic Data Exchange, **DDE),** сейчас – через внедрение и связывание объектов (Object Linking and Embedding, OLE).

И все же, реализованная в ранних версиях Windows многозадачность не была традиционной вытесняющей, основанной на выделении отрезков времени, как в многопользовательских операционных системах. Такие операционные системы используют системный таймер для периодического прерывания выполнения одной задачи и запуска другой. 16-разрядные версии Windows поддерживали так называемую невытесняющую многозадачность (non-preemptive multitasking). Такой тип многозадачности был возможен благодаря основанной на сообщениях архитектуре Windows. В общем случае, Windows-программа находилась в памяти и не выполнялась до тех пор, пока не получала сообщение. Эти сообщения часто являлись прямым или косвенным результатом ввода информации пользователем с клавиатуры или мыши. После обработки сообщения программа возвращала управление обратно Windows.

16-разрядные версии Windows не имели возможности произвольно переключать управление с одной Windows-программы на другую, основываясь на квантах времени таймера. Переключение между задачами происходило в момент, когда программа завершала обработку сообщения и возвращала управление Windows. Такую невытесняющую многозадачность называют также кооперативной многозадачностью (cooperative multitasking) потому, что она требует некоторого согласования между приложениями. Одна Windows-программа могла парализовать работу всей системы, если ей требовалось много времени для обработки сообщения.

Хотя невытесняющая многозадачность была основным типом многозадачности в 16-разрядных версиях Windows, некоторые элементы вытесняющей (примитивной, preemptive) многозадачности в них тоже присутствовали.

Windows использовала вытесняющую многозадачность для выполнения DOS-программ, а также позволяла библиотекам динамической компоновки (DLL) получать прерывания аппаратного таймера для задач мультимедиа.

16-разрядные версии Windows имели некоторые особенности, которые помогали программистам если не разрешить, то, по крайней мере, справиться с ограничениями, связанными с невытесняющей многозадачностью. Наиболее известной является отображение курсора мыши в виде песочных часов. Конечно, это не решение проблемы, а только лишь возможность дать знать пользователю, что программа занята выполнением протяженной во времени работы, и что система какое-то время будет недоступна. Другим частичным решением является использование системного таймера Windows, что позволяет выполнять какие-либо действия периодически. Таймер часто используется в приложениях типа часов и приложениях, работающих с анимацией.

Другим решением по преодолению ограничений невытесняющей многозадачности является вызов функции *PeekMessage,* как мы видели в программе RANDRECT в главе 4. Обычно программа использует вызов функции *GetMes-sage* для извлечения сообщений из очереди. Однако, если в данный момент времени очередь сообщений пуста, то функция *GetMessage* будет ждать поступления сообщения в очередь, а затем возвратит его. Функция *PeekMessage* работает иначе – она возвращает управление программе даже в том случае, если нет сообщений в очереди. Таким образом, выполнение работы, требующей больших затрат времени, будет продолжаться до того момента, пока в очереди не появятся сообщения для данной или любой другой программы.

Presentation Manager и последовательная очередь сообщений

Первой попыткой фирмы Microsoft (в сотрудничестве с IBM) внедрить многозадачность в квази-DOS/Windows оболочку была система OS/2 и Presentation Manager (PM). Хотя OS/2, конечно, поддерживала вытесняющую многозадачность, часто казалось, что это вытеснение не было перенесено в PM. Дело в том, что PM выстраивал в очередь сообщения, формируемые в результате пользовательского ввода от клавиатуры или мыши. Это означает, что PM не предоставляет программе такое пользовательское

сообщение до тех пор, пока предыдущее сообщение, введенное пользователем, не будет полностью обработано.

Хотя сообщения от клавиатуры или мыши – это только часть множества сообщений, которые может получить программа в PM или Windows, большинство других сообщений являются результатом событий, связанных с клавиатурой или мышью. Например, сообщение от меню команд является результатом выбора пункта меню с помощью клавиатуры или мыши. Сообщение от клавиатуры или мыши не будет обработано до тех пор, пока не будет полностью обработано сообщение от меню.

Основная причина организации последовательной очереди сообщений состоит в том, чтобы отследить все действия пользователя. Если какое-либо сообщение от клавиатуры или мыши вызывает переход фокуса ввода от одного окна к другому, то следующее сообщение клавиатуры должно быть направлено в окно, на которое установился фокус ввода. Таким образом, система не знает, в какое окно передавать сообщение на обработку до тех пор, пока не будет обработано предыдущее сообщение.

В настоящее время принято соглашение о том, что не должно быть возможности для какого-либо одного приложения парализовать работу всей системы, и что требуется использовать непоследовательную очередь сообщений, поддерживаемую системами Windows 95 и Windows NT. Если одна программа занята выполнением протяженной во времени операции, то существует возможность переключить фокус ввода на другое приложение.

Решения, использующие многопоточность

Выше был рассмотрен Presentation Manager операционной системы OS/2 только из-за того, что это была первая оболочка, которая подготовила сознание некоторых ветеранов программирования под Windows (в том числе и автора) к введению многопоточности. Интересно, что *ограниченная* поддержка многопоточности в PM дала программистам основную идею организации программ, использующих многопоточность. Хотя эти

ограничения сейчас преимущественно преодолены в Windows 95, тем не менее уроки, полученные при работе с более ограниченными системами, остаются актуальными и по сей день.

В многопоточной среде программы могут быть разделены на части, называемые потоками выполнения (threads), которые выполняются одновременно. Поддержка многопоточности оказывается лучшим решением проблемы последовательной очереди сообщений в PM и приобретает полный смысл при ее реализации в Windows 95.

В терминах программы "поток" – это просто функция, которая может также вызывать другие функции программы. Программа начинает выполняться со своего главного (первичного) потока, который в традиционных программах на языке С является функцией *main*, а в Windows-программах – *WinMain*. Будучи выполняемой, функция может создавать новые потоки обработки, выполняя системный вызов с указанием функции инициализации потока (initial threading function). Операционная система в вытесняющем режиме переключает управление между потоками подобно тому, как она это делает с процессами.

В PM системы OS/2 любой поток может либо создавать очередь сообщений, либо не создавать. PM-поток должен создавать очередь сообщений, если он собирается создавать окно. С другой стороны, поток может не создавать очередь сообщений, если он осуществляет только обработку данных или графический вывод. Поскольку потоки, не создающие очереди сообщений, не обрабатывают сообщения, то они не могут привести к "зависанию" системы. На поток, не имеющий очереди сообщений, накладывается только одно ограничение – он не может посылать асинхронное сообщение в окно потока, имеющего очередь сообщений, или вызывать какую-либо функцию, если это приведет к посылке сообщения. (Однако эти потоки могут *посылать* синхронные сообщения потокам, имеющим очередь сообщений.)

Таким образом, программисты, работавшие с PM, научились разби-

вать свои программы на один поток с очередью сообщений, создающий все окна и обрабатывающий сообщения для них, и один или несколько потоков, не имеющих очередей сообщений, и выполняющих продолжительные действия в фоновом режиме. Кроме того, программисты, работавшие с PM, узнали о "правиле 1/10 секунды". Оно состоит в том, что поток с очередью сообщений тратит не более 1/10 секунды на обработку любого сообщения. Все, что требует большего времени, следовало выделять в отдельный поток. Если все программисты придерживались этого правила, то никакая PM-программа не могла вызвать зависание системы более чем на 1/10 секунды.

Многопоточная архитектура

Как уже отмечалось выше, ограничения PM дали программистам основные идеи для понимания того, как использовать множество потоков в программе, выполняемой в графической среде. Ниже приведены наши рекомендации по архитектуре многопоточных программ: первичный или главный (primary) поток вашей программы создает все окна и соответствующие им оконные процедуры, необходимые в программе и обрабатывает все сообщения для этих окон. Все остальные потоки – это просто фоновые задачи. Они не имеют интерактивной связи с пользователем, кроме как через первичный поток.

Один из способов добиться этого состоит в том, чтобы первичный поток обрабатывал пользовательский ввод и другие сообщения, возможно создавая при этом вторичные (secondary) потоки в процессе. Эти вторичные потоки выполняют не связанные с пользователем задачи.

Другими словами, первичный поток вашей программы является губернатором, а вторичные потоки – свитой губернатора. Губернатор поручает всю большую работу своим помощникам на то время, пока он осуществляет контакты с внешним миром. Поскольку вторичные потоки являются членами свиты, они не могут проводить свои пресс-конференции. Они скромно выполняют каждый свое задание, делают отчет губернатору и

ждут новых указаний.

Потоки внутри отдельной программы являются частями одного процесса, поэтому они разделяют все ресурсы процесса, такие как память и открытые файлы. Поскольку потоки разделяют память, отведенную программе, то они разделяют и статические переменные. Однако, у каждого потока есть свой собственный стек, и значит, автоматические переменные являются уникальными для каждого потока. Каждый поток, также, имеет свое состояние процессора, которое сохраняется и восстанавливается при переключении между потоками.

Коллизии, возникающие при использовании потоков

Собственно разработка, программирование и отладка сложного многопоточного приложения являются, естественно, самыми сложными задачами, с которыми приходится сталкиваться программисту для Windows. Поскольку в системе с вытесняющей многозадачностью поток может быть прерван в любой момент для переключения на другой поток, то может случайно произойти любое нежелательное взаимодействие между двумя потоками.

Одной из основных ошибок в многопоточных программах является так называемое состояние гонки (race condition). Это случается, если программист считает, что один поток закончит выполнение своих действий, например, подготовку каких-либо данных, до того, как эти данные потребуются другому потоку. Для координации действий потоков операционным системам необходимы различные формы синхронизации. Одной из таких форм является семафор (semaphore), который позволяет программисту приостановить выполнение потока в конкретной точке программы до тех пор, пока он не получит от другого потока сигнал о том, что он может возобновить работу. Похожи на семафоры критические разделы (critical sections), которые представляют собой разделы кода, во время выполнения которого, поток не может быть прерван.

Но использование семафоров может привести к другой распростра-

ненной ошибке, связанной с потоками, которая называется тупиком (deadlock). Это случается, когда два потока блокируют выполнение друг друга, а для того, чтобы их разблокировать необходимо продолжить работу.

К счастью, 32-разрядные программы более устойчивы к определенным проблемам, включая проблемы с потоками, чем 16-разрядные программы. Например, предположим, что один поток выполняет простое действие:

lCount++ ;

где *lCount* – 32-разрядная глобальная переменная типа длинное целое, используемая другими потоками. В 16-разрядной программе, в которой такой оператор языка С транслируется в две инструкции машинного кода (сначала инкрементируется младшие 16 разрядов, а затем добавляется перенос в старшие 16 разрядов). Допустим, что операционная система прервала поток между этими двумя инструкциями машинного кода. Если переменная *lCount* имела значение $0000FFFF, то после выполнения первой инструкции машинного кода *lCount* будет иметь нулевое значение. Если в этот момент произойдет прерывание потока, то другой поток получит нулевое значение переменной *lCount*. Только после окончания этого потока значение *lCount* будет увеличено на единицу до своего истинного значения $00010000.

Такого рода ошибка может быть никогда не выявлена, поскольку довольно редко приводит к проблемам во время выполнения. Для 16-разрядных программ наилучший путь предотвратить такую ошибку – это поместить данное выражение в критический раздел, в рамках которого поток не может быть прерван. В 32-разрядной программе, однако, приведенное выражение является абсолютно корректным, поскольку оно компилируется в одну инструкцию машинного кода.

Преимущества Windows

Операционные системы Windows 95 и Windows NT не имеют последовательной очереди сообщений. Такое решение кажется очень хорошим:

если программа выполняет длительную обработку сообщения, то курсор мыши принимает форму песочных часов при расположении над окном этой программы, и изменяется на обычную стрелку, если он располагается над окном другой программы. Простым щелчком кнопкой мыши можно перевести другое окно на передний план.

Однако пользователь по-прежнему не может работать с приложением, выполняющим длительную операцию, поскольку выполнение длительной операции предотвращает получение сообщений программой. А это нежелательно. Программа должна быть всегда открыта для сообщений, а это требует использования вторичных потоков.

В Windows 95 и Windows NT не существует различия между потоками, имеющими очередь сообщений, и потоками без очереди сообщений. При создании каждый поток получает свою собственную очередь сообщений. Это снижает число ограничений, существующих для потоков в PM-программе. (Однако, в большинстве случаев все еще обработка ввода и сообщений осуществляется в одном потоке, а протяженные во времени задачи передаются другим потокам, которые не создают окон.) Такая схема организации приложения, как мы увидим, почти всегда является наиболее разумной.

В Windows 95 и Windows NT есть функция, которая позволяет одному потоку уничтожить другой поток, принадлежащий тому же процессу. Как вы обнаружите, когда начнете писать многопоточные приложения под Windows, иногда это очень удобно. Ранние версии операционной системы OS/2 не содержали функции для уничтожения потоков.

Windows 95 и Windows NT поддерживают так называемую локальную память потока (thread local storage, TLS). Для того чтобы понять, что это такое, вспомним о том, что статические переменные, как глобальные так и локальные по отношению к функциям, разделяются между потоками, поскольку они расположены в зоне памяти данных процесса. Автоматические переменные (которые являются всегда локальными по отношению к

функции) – уникальны для каждого потока, т. к. они располагаются в стеке, а каждый поток имеет свой стек.

Иногда бывает удобно использовать для двух и более потоков одну и ту же функцию, а статические данные использовать уникальные для каждого потока. Это и есть пример использования локальной памяти потока. Существует несколько вызовов функций Windows для работы с локальной памятью потока. Фирма Microsoft ввела расширение в компилятор C, которое позволяет использовать локальную память потока более прозрачным для программиста образом.

Новая усовершенствованная многопоточная программа

Иногда имеет место тенденция использовать в программе каждую возможность, предлагаемую операционной системой. Нет смысла использовать множество потоков в программе, которая в этом не нуждается. Если программа выводит на экран курсор в виде песочных часов на достаточно долгий период времени, или, если она использует функцию *PeekMessage* для того, чтобы избежать появления курсора в виде песочных часов, то тогда идея реструктуризации программы в многопоточную, вероятно, может оказаться хорошей. В противном случае, вы только усложните себе работу и, возможно, внесете в программу новые ошибки.

Есть даже некоторые ситуации, когда появление курсора мыши в виде песочных часов, может быть совершенно подходящим. Выше уже упоминалось "правило 1/10 секунды". Так вот, загрузка большого файла в память может потребовать больше времени, чем 1/10 секунды. Значит ли это, что функции загрузки файла должны были быть реализованы с использованием разделения на потоки? Совсем необязательно. Когда пользователь дает программе команду открыть файл, то он или она обычно хочет, чтобы операционная система выполнила ее немедленно. Выделение процесса загрузки файла в отдельный поток просто приведет к усложнению программы. Не стоит это делать даже ради того, чтобы похвастаться перед друзьями, что вы пишите многопоточные приложения.

О использовании функции Sleep

Выше было показано, как лучше организовать архитектуру программы, использующей многопоточность, а именно, чтобы первичный поток создавал все окна в программе, содержал все оконные процедуры этих окон и обрабатывал все сообщения. Вторичные потоки выполняют фоновые задачи или задачи, протяженные во времени.

Однако, предположим, что требуется реализовать анимацию во вторичном потоке. Обычно анимация в Windows осуществляется с использованием сообщения WM_TIMER. Но если вторичный поток не создает окно, то он не может получить это сообщение. А без задания определенного темпа анимация могла бы осуществляться слишком быстро.

Решение состоит в использовании функции *Sleep*. Поток вызывает функцию *Sleep* для того, чтобы добровольно отложить свое выполнение. Единственный параметр этой функции – время, задаваемое в миллисекундах. Функция *Sleep* не осуществляет возврата до тех пор, пока не истечет указанное время. В течение него выполнение потока приостанавливается и выделения для него процессорного времени не происходит (хотя очевидно, что для потока все-таки требуется какое-то незначительное время, за которое система должна определить, пора возобновлять выполнение потока или нет).

Если параметр функции *Sleep* задан равным нулю, то поток будет лишен остатка выделенного ему кванта процессорного времени.

Когда поток вызывает функцию *Sleep*, задержка на заданное время относится к этому потоку. Система продолжает выполнять другие потоки этого и других процессов

Критический раздел

В однозадачной операционной системе обычные программы не нуждаются в "светофорах" для координации их действий. Они выполняются так, как будто они являются хозяевами дороги, по которой они следуют. Не существует ничего, что могло бы вмешаться в то, что они делают.

Даже в многозадачной операционной системе большинство программ выполняются независимо друг от друга. Но некоторые проблемы все же могут возникнуть. Например, двум программам может понадобиться читать и писать в один файл в одно и то же время. Для таких случаев операционная система поддерживает механизм разделения файлов (shared files) и блокирования отдельных фрагментов файла (record locking).

Однако, в операционной системе, поддерживающей многопоточность, такое решение может внести путаницу и создать потенциальную опасность. Разделение

данных между двумя и более потоками является общим случаем. Например, один поток может обновлять одну или более переменных, а другой может использовать эти переменные. Иногда в этой ситуации может возникнуть проблема, а иногда – нет. (Помните, что операционная система может переключать управление потоками только между инструкциями машинного кода. Если простое целое число разделяется между двумя потоками, то изменение этой переменной обычно осуществляется одной инструкцией машинного кода, и потенциальные проблемы сводятся к минимуму.)

Однако, предположим, что потоки разделяют несколько переменных или структуру данных. Часто эти сложные переменные или поля структур данных должны быть согласованными между собой. Операционная система может прерывать поток в середине процесса обновления этих переменных. В этом случае поток, который затем использует эти переменные, будет иметь дело с несогласованными данными.

В результате бы возникла коллизия, и нетрудно представить себе, как такого рода ошибка может привести к краху программы. В этой ситуации нам необходимо нечто похожее на светофор, который мог бы синхронизировать и координировать работу потоков. Таким средством и является критический раздел. Критический раздел – это блок кода, при выполнении которого поток не может быть прерван.

Имеется четыре функции для работы с критическими разделами. Чтобы их использовать, вам необходимо определить объект типа критический раздел, который является глобальной переменной типа CRITICAL_SECTION. Например,

```
CRITICAL_SECTION CS ;
```

Тип данных CRITICAL_SECTION является структурой, но ее поля используются только внутри Windows. Объект типа критический раздел сначала должен быть инициализирован одним из потоков программы с помощью функции:

```
InitializeCriticalSection (&cs);
```

Эта функция создает объект критический раздел с именем *cs*. В документации содержится следующее предупреждение: "Объект критический раздел не может быть перемещен или скопирован. Процесс также не должен модифицировать объект, а должен обращаться с ним, как с "черным ящиком"."

После инициализации объекта критический раздел поток входит в критический раздел, вызывая функцию:

```
EnterCriticalSection (&cs) ;
```

В этот момент поток становится владельцем объекта. Два различных потока не могут быть владельцами одного объекта критический раздел одновременно. Следовательно, если один поток вошел в критический раздел, то следующий поток, вызывая функцию *EnterCriticalSection с* тем же самым объектом

критический раздел, будет задержан внутри функции. Возврат из функции произойдет только тогда, когда первый поток покинет критический раздел, вызвав функцию:

```
LeaveCriticalSection (&cs);
```

В этот момент второй поток, задержанный в функции *EnterCriticalSection,* станет владельцем критического раздела, и его выполнение будет возобновлено.

Когда объект критический раздел больше не нужен вашей програм-

ме, его можно удалить с помощью функции:

```
DeleteCriticalSection (&cs);
```

Это приведет к освобождению всех ресурсов системы, задействованных для поддержки объекта критический раздел.

Механизм критических разделов основан на принципе взаимного исключения (mutual exclusion). Этот термин нам еще встретится при дальнейшем рассмотрении синхронизации потоков. Только один поток может быть владельцем критического раздела в каждый конкретный момент времени. Следовательно, один поток может войти в критический раздел, установить значения полей структуры и выйти из критического раздела. Другой поток, использующий эту структуру, также мог бы войти в критический раздел перед осуществлением доступа к полям структуры, а затем выйти из критического раздела.

Обратите внимание, что возможно определение нескольких объектов типа критический раздел, например, *cs1* и *cs2*. Если в программе имеется четыре потока, и два первых из них разделяют некоторые данные, то они могут использовать первый объект критический раздел, а два других потока, также разделяющих другие данные, могут использовать второй объект критический раздел.

Обратите внимание, что надо быть весьма осторожным при использовании критического раздела в главном потоке. Если вторичный поток проводит слишком много времени в его собственном критическом разделе, то это может привести к "зависанию" главного потока на слишком большой период времени.

Объект Mutex

Существует одно ограничение в использовании критических разделов. Оно заключается в том, что их можно применять для синхронизации потоков только в рамках одного процесса. Но бывают случаи, когда необходимо синхронизировать действия потоков различных процессов, которые разделяют какие-либо ресурсы (например, память). Использовать кри-

тические разделы в такой ситуации нельзя. Вместо них подключаются объекты типа mutex (mutex object).

Составное слово "mutex" происходит из словосочетания "mutual exclusion", что означает взаимное исключение, и очень точно отражает назначение объектов. Мы хотим предотвратить возможность прерывания потока в программе до тех пор, пока не будет выполнено обновление или использование разделяемых данных.

Уведомления о событиях

Мы можем определить понятие большой работы как действия, выполняя которые, программа нарушит "правило 1/10 секунды". Примерами большой работы могут служить: проверка орфографии в текстовых процессорах, сортировка и индексирование файлов баз данных, пересчет электронной таблицы, печать и даже сложное рисование. Конечно, как мы уже знаем, лучшее решение состоит в следовании "правилу 1/10 секунды", т. е. в передаче большой работы вторичным потокам обработки. Эти вторичные потоки не создают окон и, значит, не ограничены "правилом 1/10 секунды".

Часто бывает, что вторичному потоку надо проинформировать первичный поток о том, что он завершился, или первичному потоку надо прервать работу, выполняемую вторичным потоком.

Локальная память потока

Глобальные переменные в многопоточных программах (так же как и любая выделенная память) разделяются между всеми потоками в программе. Локальные статические переменные функций также разделяются между всеми потоками, использующими эту функцию. Локальные автоматические переменные в функции являются уникальными для каждого потока, потому что они хранятся в стеке, а каждый поток имеет свой собственный стек.

Может возникнуть необходимость иметь постоянную область памяти, уникальную для каждого потока. Например, функция *strtok* языка C,

которая уже упоминалась в этой главе, требует такого типа память. Нет сомнений, что C его не поддерживает. В Windows 95 имеется четыре функции, поддерживающие эту память, которая называется локальной памятью потока (thread local storage, TLS).

Первичный поток вызывает функцию *JTsAlloc для* получения значения индекса:

```
dwTlsIndex = TIsAlloc () ;
```

Он может храниться в глобальной переменной или может быть передан функции потока в параметре-структуре.

Функция потока начинается с выделения памяти для структуры данных и с вызова функции *TIsSetValue,* используя индекс, полученный ранее:

```
TIsSetValue (dwTlsIndex, GlobalAlloc (GPTR, sizeof (DATA))) ;
```

Это действие устанавливает соответствие указателя с конкретным потоком и конкретным индексом в потоке. Теперь, любая функция, которой нужно использовать этот указатель (включая саму базовую функцию потока), может использовать код, подобный такому:

```
PDATA pdata ;
pdata = (PDATA) TIsGetValue (dwTlsIndex) ;
```

Теперь она может изменять значения *pdata->a* и *pdata->b.* Перед завершением функции потока необходимо освободить захваченную память:

```
GlobalFree (TIsGetValue (dwTlsIndex)) ;
```

Когда все потоки, использующие эти данные будут завершены, первичный поток освобождает индекс:

```
TIsFree (dwTlsIndex) ;
```

Полезно посмотреть как организована локальная память потока. (Мне неизвестно, как в действительности Windows 95 это делает, но описываемая схема вполне правдоподобна.) Во-первых, функция *TIsAlloc* могла бы просто выделить блок памяти (длиной 0 байт) и вернуть значение индекса, который является указателем на этот блок. Каждый раз при вызове функции *TIsSet Value* с этим индексом блок памяти увеличивается на 8 байт с помощью функции *GlobalReAlloc.* В этих 8 байтах хранятся идентификатор потока, вызывающего функцию, полученный с помощью функции

GetCurrentThreadID, и указатель, переданный функции *TlsSetValue*. Функция *TlsGetValue* просто использует идентификатор потока для поиска в таблице, а затем возвращает указатель. Функция *TZsFree* освобождает блок памяти.

Реализация многопоточности в Delphi

Стандартный мастер модулей в Delphi автоматически создает модуль содержащий класс потока с указанным именем. Весь код который необходимо вынести в отдельный поток помещается в метод класса Execute.

Базовый класс для создания потока пользователя – TThread

```
TThread = class
protected
    procedure DoTerminate; virtual;
    procedure Execute; virtual;
    procedure Synchronize(Method: TThreadMethod);
    property ReturnValue: Integer;
    property Terminated: Boolean;
  public
    constructor Create(CreateSuspended: Boolean);
    procedure Resume;
    procedure Suspend;
    procedure Terminate;
    function WaitFor: LongWord;
    property FreeOnTerminate: Boolean;
    property Handle: Thandle;
    property Priority: TthreadPriority;
    property Suspended: Boolean;
    property ThreadID: Thandle;
    property OnTerminate: TnotifyEvent;
end;
```

Процесс, породивший поток может гибко управлять его состоянием: приоритетом Priority; приостановить и продолжить его исполнения, а так же досрочно завершить выполнение потока.

Для вызова методов VCL необходимо синхронизировать дочерний поток с главным. Для этого служит процедура Synchronize(Method:TThreadMethod);

```
unit Unit1;
interface
uses
  Classes;
type
```

```
  TSamples = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;

implementation

{ Подсказка Delphi по поводу Synchronize,

Important: Methods and properties of objects in VCL can only be used in a
  method called using Synchronize, for example,

      Synchronize(UpdateCaption);

  and UpdateCaption could look like,

    procedure Samples.UpdateCaption;
    begin
      Form1.Caption := 'Updated in a thread';
    end; }

{ TSamples }

procedure TSamples.Execute;
begin
  { Здесь должен быть размещен код потока }
end;

end.
```

Список используемой литературы

1. Turbo Pascal for Windows в 2-х томах. Нейл Рубенкинг Пер. с англ. – М.:Мир, 1993, 536 с., ил.
2. Теория и практика C++. Герберт Шилдт. пер. с англ. – СПб.: BHV – Санкт-Петербург, 1996. 416 с., ил.
3. Программирование для Windows 95; в 2-х томах. Чарльз Петзолд. пер. с англ. – СПб.: BHV – Санкт-Петербург, 1997.– 752 с., ил.
4. Микропроцессоры 80x86 Архитектура, функционирование. В.М.Михальчук А.А.Ровдо С.В.Рыжиков Мн.: Беларкас, 1994. – 408с.

В моей работе были рассмотрены основные принципы многозадачности и многопоточности. Приведены примеры оформления многопоточных приложений на Delphi. В дальнейшем многозадачная технология получит дополнительное развитие. В последнем выходящем релизе Windows под названием Windows2000 для поддержки многозадачности используются так называемые объекты-задания симметрично распределяющиеся между процессорами. Для ускорения работы мощных серверных систем используют машины на базе 2,4 и даже 8 процессорах. Такое серверное программное обеспечение как Windows NT4.0, Unix, Linux так же поддерживает SMP – симметричную мультипроцессорную обработку данных. Работа таких систем была бы невозможна без использования алгоритмов многозадачности и многопоточности.