**30. Опишите процесс синхронизации потоков.**

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

Во многих операционных системах эти средства называются средствами межпро­цессного взаимодействия — Inter Process Communications (IPC), что отражает историческую первичность понятия «процесс» по отношению к понятию «поток». Обычно к средствам IPC относят не только средства межпроцессной синхрони­зации, но и средства межпроцессного обмена данными.

Выполнение потока в мультипрограммной среде всегда имеет асинхронный ха­рактер. Очень сложно с полной определенностью сказать, на каком этапе выпол­нения будет находиться процесс в определенный момент времени. Даже в однопрограммном режиме не всегда можно точно оценить время выполнения задачи. Это время во многих случаях существенно зависит от значения исходных дан­ных, которые влияют на количество циклов, направления разветвления програм­мы, время выполнения операций ввода-вывода и т. п. Так как исходные данные в разные моменты запуска задачи могут быть разными, то и время выполнения от­дельных этапов и задачи в целом является весьма неопределенной величиной.

Еще более неопределенным является время выполнения программы в мульти­программной системе. Моменты прерывания потоков, время нахождения их в очередях к разделяемым ресурсам, порядок выбора потоков для выполнения — все эти события являются результатом стечения многих обстоятельств и могут быть интерпретированы как случайные. В лучшем случае можно оценить веро­ятностные характеристики вычислительного процесса, например, вероятность его завершения за данный период времени.

Таким образом, потоки в общем случае (когда программист не предпринял спе­циальных мер по их синхронизации) протекают независимо, асинхронно друг другу. Это справедливо как по отношению к потокам одного процесса, выпол­няющим общий программный код, так и по отношению к потокам разных про­цессов, каждый из которых выполняет собственную программу.

Любое взаимодействие процессов или потоков связано с их синхронизацией*,*ко­торая заключается в согласовании их скоростей путем приостановки потока до наступления некоторого события и последующей его активизации при наступ­лении этого события. Синхронизация лежит в основе любого взаимодействия потоков, связано ли это взаимодействие с разделением ресурсов или с обменом данными. Например, поток-получатель должен обращаться за данными только после того, как они помещены в буфер потоком-отправителем. Если же поток-получатель обратился к данным до момента их поступления в буфер, то он дол­жен быть приостановлен.

При совместном использовании аппаратных ресурсов синхронизация также со­вершенно необходима. Когда, например, активному потоку требуется доступ к последовательному порту, а с этим портом в монопольном режиме работает дру­гой поток, находящийся в данный момент в состоянии ожидания, то ОС приос­танавливает активный поток и не активизирует его до тех пор, пока нужный ему порт не освободится. Часто нужна также синхронизация с событиями, внешними по отношению к вычислительной системе, например реакции на нажатие комби­нации клавиш Ctrl+C.

Ежесекундно в системе происходят сотни событий, связанных с распределением и освобождением ресурсов, и ОС должна иметь надежные и производительные средства, которые бы позволяли ей синхронизировать потоки с происходящими в системе событиями.

Для синхронизации потоков прикладных программ исполь­зуются как собственные средства и приемы синхронизации, так и средства опера­ционной системы. Например, два потока одного прикладного процесса могут ко­ординировать свою работу с помощью доступной для них обоих глобальной логической переменной, которая устанавливается в единицу при осуществлении некоторого события, например, выработки одним потоком данных, нужных для продолжения работы другого. Однако, во многих случаях более эффективными или даже единственно возможными являются средства синхронизации, предос­тавляемые операционной системой в форме системных вызовов. Так, потоки, принадлежащие разным процессам, не имеют возможности вмешиваться каким-либо образом в работу друг друга. Без посредничества операционной системы они не могут приостановить друг друга или оповестить о произошедшем собы­тии. Средства синхронизации используются операционной системой не только для синхронизации прикладных процессов, но и для ее внутренних нужд.

Обычно разработчики операционных систем предоставляют в распоряжение при­кладных и системных программистов широкий спектр средств синхронизации. Эти средства могут образовывать иерархию, когда на основе более простых средств строятся более сложные, а также быть функционально специализированными, например средства для синхронизации потоков одного процесса, средства для синхронизации потоков разных процессов при обмене данными и т. д. Часто функциональные возможности разных системных вызовов синхронизации пере­крываются, так что для решения одной задачи программист может воспользо­ваться несколькими вызовами в зависимости от своих личных предпочтений.

Пренебрежение вопросами синхронизации в многопоточной системе может при­вести к неправильному решению задачи или даже к краху системы. Рассмотрим, например (рис. 1), задачу ведения базы данных клиентов некоторого предпри­ятия. Каждому клиенту отводится отдельная запись в базе данных, в которой среди прочих полей имеются поля Заказ и Оплата. Программа, ведущая базу дан­ных, оформлена как единый процесс, имеющий несколько потоков, в том числе поток А, который заносит в базу данных информацию о заказах, поступивших от клиентов, и поток В, который фиксирует в базе данных сведения об оплате кли­ентами выставленных счетов. Оба эти потока совместно работают над общим файлом базы данных, используя однотипные алгоритмы, включающие три шага.

1. Считать из файла базы данных в буфер запись о клиенте с заданным иденти­фикатором.

2. Внести новое значение в поле Заказ (для потока А) или Оплата (для потока В).

3. Вернуть модифицированную запись в файл базы данных.

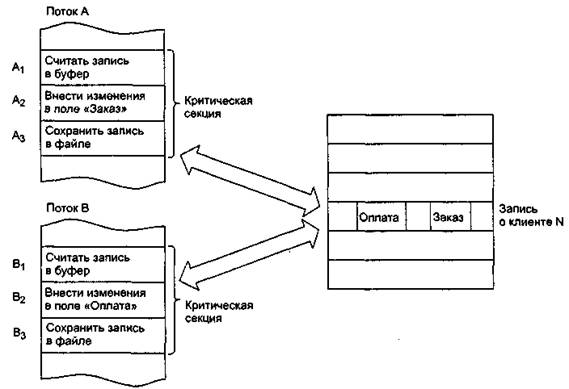


Рис. 1.Возникновение гонок при доступе к разделяемым данным

Обозначим соответствующие шаги для потока А как Al, A2 и A3, а для потока В как В1, В2 и ВЗ. Предположим, что в некоторый момент поток А обновляет поле Заказ записи о клиенте N. Для этого он считывает эту запись в свой буфер (шаг А1), модифицирует значение поля Заказ (шаг А2), но внести запись в базу данных (шаг A3) не успевает, так как его выполнение прерывается, например, вследствие завершения кванта времени.

Предположим также, что потоку В также потребовалось внести сведения об оп­лате относительно того же клиента N. Когда подходит очередь потока В, он успе­вает считать запись в свой буфер (шаг В1) и выполнить обновление поля Оплата (шаг В2), а затем прерывается. Заметим, что в буфере у потока В находится за­пись о клиенте N, в которой поле Заказ имеет прежнее, не измененное значение.

Когда в очередной раз управление будет передано потоку А, то он, продолжая свою работу, запишет запись о клиенте N с модифицированным полем Заказ в базу данных (шаг A3). После прерывания потока А и активизации потока В по­следний запишет в базу данных поверх только что обновленной записи о клиен­те N свой вариант записи, в которой обновлено значение поля Оплата. Таким об­разом, в базе данных будут зафиксированы сведения о том, что клиент N произвел оплату, но информация о его заказе окажется потерянной (рис. 2, а).

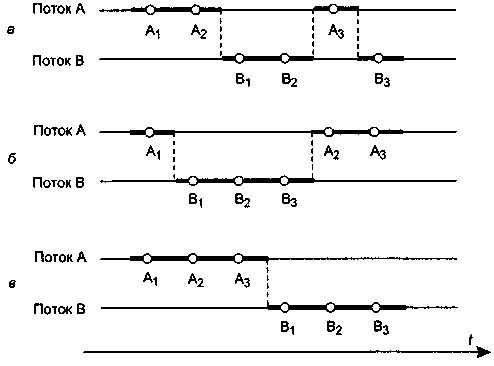


Рис. 2.Влияние относительных скоростей потоков на результат решения задачи

Сложность проблемы синхронизации кроется в нерегулярности возникающих ситуаций. Так, в предыдущем примере можно представить и другое развитие со­бытий: могла быть потеряна информация не о заказе, а об оплате (рис. 2, *6)* или, напротив, все исправления были успешно внесены (рис. 2, *в).*Все опреде­ляется взаимными скоростями потоков и моментами их прерывания. Поэтому отладка взаимодействующих потоков является сложной задачей. Ситуации, по­добные той, когда два или более потоков обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей потоков, называются гонками*.*

Важным понятием синхронизации потоков является понятие «критической сек­ции» программы. Критическая секция- это часть программы, результат выпол­нения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программы, изменяются другими потоками в то время, когда вы­полнение этой части еще не завершено. Критическая секция всегда определяется по отношению к определенным критическим данным,при несогласованном из­менении которых могут возникнуть нежелательные эффекты. В предыдущем при­мере такими критическими данными являлись записи файла базы данных. Во всех потоках, работающих с критическими данными, должна быть определена критическая секция. Заметим, что в разных потоках критическая секция состоит в общем случае из разных последовательностей команд.

Чтобы исключить эффект гонок по отношению к критическим данным, необхо­димо обеспечить, чтобы в каждый момент времени в критической секции, свя­занной с этими данными, находился только один поток. При этом неважно, нахо­дится этот поток в активном или в приостановленном состоянии. Этот прием называют взаимным исключением.Операционная система использует разные способы реализации взаимного исключения. Некоторые способы пригодны для взаимного исключения при вхождении в критическую секцию только потоков одного процесса, в то время как другие могут обеспечить взаимное исключение и для потоков разных процессов.

Самый простой и в то же время самый неэффективный способ обеспечения вза­имного исключения состоит в том, что операционная система позволяет потоку запрещать любые прерывания на время его нахождения в критической секции. Однако этот способ практически не применяется, так как опасно доверять управ­ление системой пользовательскому потоку — он может надолго занять процес­сор, а при крахе потока в критической секции крах потерпит вся система, потому что прерывания никогда не будут разрешены.

**Механизмы синхронизации**

Для синхронизации потоков одного процесса можно использовать глобальные ***блокирующие переменные.***С этими переменными, к ко­торым все потоки процесса имеют прямой доступ, программист работает, не об­ращаясь к системным вызовам ОС.

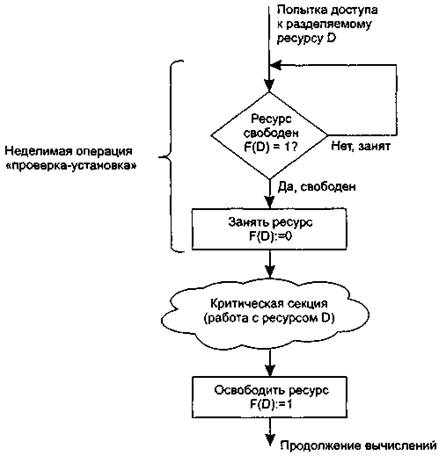


Рис. 3.Реализация критических секций с использованием блокирующих переменных

Каждому набору критических данных ставится в соответствие двоичная пере­менная, которой поток присваивает значение 0, когда он входит в критическую секцию, и значение 1, когда он ее покидает. На рис. 3 показан фрагмент алго­ритма потока, использующего для реализации взаимного исключения доступа к критическим данным D блокирующую переменную F(D). Перед входом в критическую секцию поток проверяет, не работает ли уже какой-нибудь поток с дан­ными D. Если переменная F(D) установлена в 0, то данные заняты и проверка циклически повторяется. Если же данные свободны (F(D) = 1), то значение пере­менной F(D) устанавливается в 0 и поток входит в критическую секцию. После того как поток выполнит все действия с данными D, значение переменной F(D) снова устанавливается равным 1.

Блокирующие переменные могут использоваться не только при доступе к разде­ляемым данным, но и при доступе к разделяемым ресурсам любого вида.

Если все потоки написаны с учетом вышеописанных соглашений, то взаимное исключение гарантируется. При этом потоки могут быть прерваны операци­онной системой в любой момент и в любом месте, в том числе в критической секции.

Однако следует заметить, что одно ограничение на прерывания все же имеется. Нельзя прерывать поток между выполнением операций проверки и установки блокирующей переменной. Поясним это.

Пусть в результате проверки перемен­ной поток определил, что ресурс свободен, но сразу после этого, не успев уста­новить переменную в 0, был прерван. За время его приостановки другой поток занял ресурс, вошел в свою критическую секцию, но также был прерван, не за­вершив работы с разделяемым ресурсом. Когда управление было возвращено первому потоку, он, считая ресурс свободным, установил признак занятости и начал выполнять свою критическую секцию. Таким образом, был нарушен прин­цип взаимного исключения, что потенциально может привести к нежелательным последствиям. Во избежание таких ситуаций в системе команд компью­теров предусмотрена единая, неделимая команда анализа и присвоения значения логической переменной (например, команды ВТС, BTR и BTS процессора). При отсутствии такой команды в процессоре соответствующие действия должны реализовываться специальными системными примитивами, которые бы запре­щали прерывания на протяжении всей операции проверки и установки.

Реализация взаимного исключения описанным выше способом имеет существен­ный недостаток: в течение времени, когда один поток находится в критической секции, другой поток, которому требуется тот же ресурс, получив доступ к про­цессору, будет непрерывно опрашивать блокирующую переменную, бесполезно тратя выделяемое ему процессорное время, которое могло бы быть использовано для выполнения какого-нибудь другого потока. Для устранения этого недостат­ка во многих ОС предусматриваются специальные системные вызовы для рабо­ты с критическими секциями.

На рис. 4 показано, как с помощью этих функций реализовано взаимное ис­ключение в операционной системе Windows.

Перед тем как начать измене­ние критических данных, поток выполняет системный вызов EnterCriticalSection(). В рамках этого вызова сначала выполняется, как и в предыдущем случае, проверка блокирующей переменной, отражающей состояние критического ресур­са. Если системный вызов определил, что ресурс занят (F(D) = 0), он в отличие от предыдущего случая не выполняет циклический опрос, а переводит поток в состояние ожидания (D) и делает отметку о том, что данный поток должен быть активизирован, когда соответствующий ресурс освободится. Поток, который в это время использует данный ресурс, после выхода из критической секции дол­жен выполнить системную функцию LeaveCriticalSection(), в результате чего блокирующая переменная принимает значение, соответствующее свободному состоянию ресурса (F(D) = 1), а операционная система просматривает очередь ожидающих этот ресурс потоков и переводит первый поток из очереди в состоя­ние готовности.

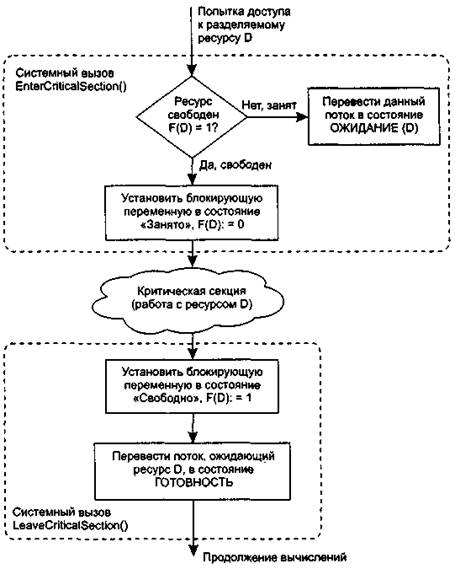


Рис. 4.Реализация взаимного исключения с использованием системных функций входа в критическую секцию и выхода из нее.

Таким образом, исключается непроизводительная потеря процессорного времени на циклическую проверку освобождения занятого ресурса. Однако в тех случаях, когда объем работы в критической секции небольшой и существует высокая ве­роятность в очень скором доступе к разделяемому ресурсу, более предпочтитель­ным может оказаться использование блокирующих переменных. Действительно, в такой ситуации накладные расходы ОС по реализации функции входа в крити­ческую секцию и выхода из нее могут превысить полученную экономию.

***Семафоры***

Обобщением блокирующих переменных являются так называемые *семафоры Дийкстры.*Вместо двоичных переменных Дийкстра (Dijkstra) предложил исполь­зовать переменные, которые могут принимать целые неотрицательные значения. Такие переменные, используемые для синхронизации вычислительных процес­сов, получили название семафоров.

Для работы с семафорами вводятся два примитива, традиционно обозначаемых Р и V. Пусть переменная S представляет собой семафор. Тогда действия V(S) и P(S) определяются следующим образом.

\* V(S): переменная S увеличивается на 1 единым действием. Выборка, наращи­вание и запоминание не могут быть прерваны. К переменной S нет доступа другим потокам во время выполнения этой операции.

\* P(S): уменьшение S на 1, если это возможно. Если S=0 и невозможно умень­шить S, оставаясь в области целых неотрицательных значений, то в этом случае поток, вызывающий операцию Р, ждет, пока это уменьшение станет возможным. Успешная проверка и уменьшение также являются неделимой операцией.

Никакие прерывания во время выполнения примитивов V и Р недопустимы.

В частном случае, когда семафор S может принимать только значения 0 и 1, он превращается в блокирующую переменную, которую по этой причине часто на­зывают двоичным семафором. Операция Р заключает в себе потенциальную воз­можность перехода потока, который ее выполняет, в состояние ожидания, в то время как операция V может при некоторых обстоятельствах активизировать дру­гой поток, приостановленный операцией Р.

Рассмотрим использование семафоров на классическом примере взаимодействия двух выполняющихся в режиме мультипрограммирования потоков, один из ко­торых пишет данные в буферный пул, а другой считывает их из буферного пула.

Пусть буферный пул состоит из N буферов, каждый из которых может содержать одну запись. В общем случае поток-писатель и поток-читатель могут иметь раз­личные скорости и обращаться к буферному пулу с переменой интенсивностью. В один период скорость записи может превышать скорость чтения, в другой - наоборот. Для правильной совместной работы поток-писатель должен приоста­навливаться, когда все буферы оказываются занятыми, и активизироваться при освобождении хотя бы одного буфера. Напротив, поток-читатель должен приос­танавливаться, когда все буферы пусты, и активизироваться при появлении хотя бы одной записи.

Введем два семафора: е — число пустых буферов, и f — число заполненных буфе­ров, причем в исходном состоянии е = N, a f = 0. Тогда работа потоков с общим буферным пулом может быть описана следующим образом (рис. 5).

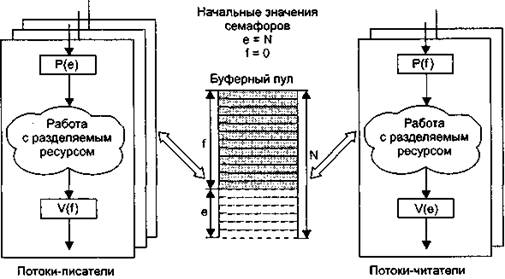


Рис. 5.Использование семафоров для синхронизации потоков

Поток-писатель прежде всего выполняет операцию Р(е), с помощью которой он проверяет, имеются ли в буферном пуле незаполненные буферы. В соответствии с семантикой операции Р, если семафор е равен 0 (то есть свободных буферов в данный момент нет), то поток-писатель переходит в состояние ожидания. Если же значением е является положительное число, то он уменьшает число свободных буферов, записывает данные в очередной свободный буфер и после этого наращи­вает число занятых буферов операцией V(f). Поток-читатель действует анало­гичным образом, с той разницей, что он начинает работу с проверки наличия заполненных буферов, а после чтения данных наращивает количество свободных буферов.

В данном случае предпочтительнее использовать семафоры вместо блокирующих переменных. Действительно, критическим ресурсом здесь является буферный пул, который может быть представлен как набор идентичных ресурсов — отдель­ных буферов, а значит, с буферным пулом могут работать сразу несколько пото­ков, и именно столько, сколько буферов в нем содержится. Использование дво­ичной переменной не позволяет организовать доступ к критическому ресурсу более чем одному потоку. Семафор же решает задачу синхронизации более гиб­ко, допуская к разделяемому пулу ресурсов заданное количество потоков. Так, в нашем примере с буферным пулом могут работать максимум N потоков, часть из которых может быть «писателями», а часть — «читателями».

Таким образом, семафоры позволяют эффективно решать задачу синхронизации доступа к ресурсным пулам, таким, например, как набор идентичных в функцио­нальном назначении внешних устройств (модемов, принтеров, портов), или на­бор областей памяти одинаковой величины, или информационных структур. Во всех этих и подобных им случаях с помощью семафоров можно организовать доступ к разделяемым ресурсам сразу нескольких потоков.

Семафор может использоваться и в качестве блокирующей переменной. В рас­смотренном выше примере, для того чтобы исключить коллизии при работе с разделяемой областью памяти, будем считать, что запись в буфер и считывание из буфера являются критическими секциями. Взаимное исключение будем обес­печивать с помощью двоичного семафора b (рис. 6).

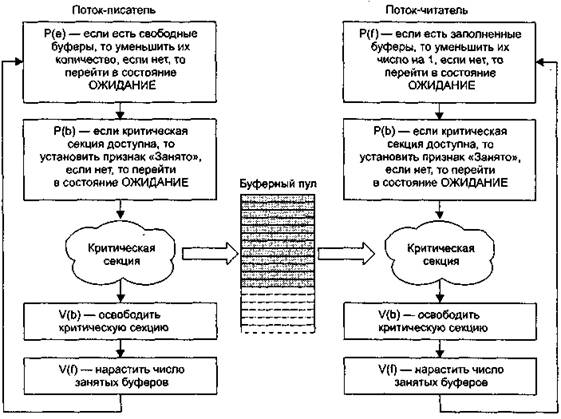


Рис. 6.Использование двоичного семафора.

Оба потока после провер­ки доступности буферов должны выполнить проверку доступности критической секции.

***Тупики***

Приведенный выше пример позволяет также проиллюстрировать еще одну проблему синхронизации — *взаимные блокировки,*называемые также *дедлоками (deadlocks), клинчами (clinch),*или *тупиками.*Покажем, что если переставить мес­тами операции Р(е) и Р(b) в потоке-писателе, то при некотором стечении обстоя­тельств эти два потока могут взаимно блокировать друг друга.

Итак, пусть поток-писатель начинает свою работу с проверки доступности кри­тической секции — операции Р(b), и пусть он первым войдет в критическую сек­цию. Выполняя операцию Р(е), он может обнаружить отсутствие свободных бу­феров и перейти в состояние ожидания. Как уже было показано, из этого состоя­ния его может вывести только поток-читатель, который возьмет очередную за­пись из буфера. Но поток-читатель не сможет этого сделать, так как для этого ему потребуется войти в критическую секцию, вход в которую заблокирован по­током-писателем. Таким образом, ни один из этих потоков не может завершить начатую работу и возникнет тупиковая ситуация, которая не может разрешиться без внешнего воздействия.

Рассмотрим еще один пример тупика. Пусть двум потокам, принадлежащим разным процессам и выполняющимся в режиме мультипрограммирования, для выполнения их работы нужно два ресурса, например принтер и последовательный порт. Такая ситуация может возникнуть, например, во время работы приложения, задачей кото­рого является распечатка информации, поступающей по модемной связи.

На рис. 7, *а*показаны фрагменты соответствующих программ. Поток А запра­шивает сначала принтер, а затем порт, а поток В запрашивает устройства в обрат­ном порядке. Предположим, что после того, как ОС назначила принтер потоку А и установила связанную с этим ресурсом блокирующую переменную, поток А был прерван. Управление получил поток В, который сначала выполнил запрос на получение СОМ-порта, затем при выполнении следующей команды был забло­кирован, так как принтер оказался уже занятым потоком А. Управление снова по­лучил поток А, который в соответствии со своей программой сделал попытку за­нять порт и был заблокирован, поскольку порт уже выделен потоку В. В таком положении потоки А и В могут находиться сколь угодно долго.

В зависимости от соотношения скоростей потоков они могут либо взаимно бло­кировать друг друга (рис. 7, *б),*либо образовывать очереди к разделяемым ре­сурсам (рис. 7, б), либо совершенно независимо использовать разделяемые ре­сурсы (рис. 7, *г).*

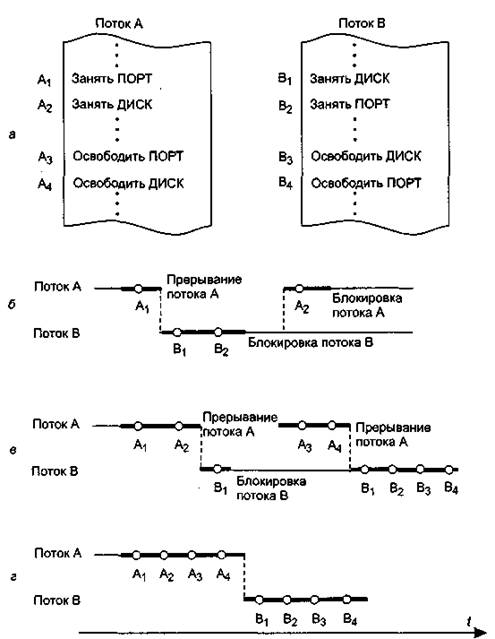


Рис. 7.Возникновение взаимных блокировок при выполнении программы

Примечание:тупиковые ситуации надо отличать от простых очередей, хотя те и другие возникают при совместном использовании ресурсов и внешне выглядят похоже: поток приостанавлива­ется и ждет освобождения ресурса. Однако очередь — это нормальное явление, неотъемле­мый признак высокого коэффициента использования ресурсов при случайном поступле­нии запросов. Очередь появляется тогда, когда ресурс недоступен в данный момент, но освободится через некоторое время, позволив потоку продолжить выполнение. Тупик же, что видно из его названия, является в некотором роде неразрешимой ситуацией. Необхо­димым условием возникновения тупика является потребность потока сразу в нескольких ресурсах.

В рассмотренных примерах тупик был образован двумя потоками, но взаимно блокировать друг друга может и большее число потоков. На рис. 8 показано такое распределение ресурсов Ri между несколькими потоками Tj, которое при­вело к возникновению взаимных блокировок. Стрелки обозначают потребность потока в ресурсах. Сплошная стрелка означает, что соответствующий ресурс был выделен потоку, а пунктирная стрелка соединяет поток с тем ресурсом, который необходим, но не может быть пока выделен, поскольку занят другим потоком. Например, потоку Т1 для выполнения работы необходимы ресурсы R1 и R2, из ко­торых выделен только один — R1, а ресурс R2 удерживается потоком Т2. Ни один из четырех показанных на рисунке потоков не может продолжить свою работу, так как не имеет всех необходимых для этого ресурсов.

Невозможность потоков завершить начатую работу из-за возникновения вза­имных блокировок снижает производительность вычислительной системы. По­этому проблеме предотвращения тупиков уделяется большое внимание. На тот случай, когда взаимная блокировка все же возникает, система должна предоста­вить администратору-оператору средства, с помощью которых он смог бы распо­знать тупик, отличить его от обычной блокировки из-за временной недоступности ресурсов. И наконец, если тупик диагностирован, то нужны средства для снятия взаимных блокировок и восстановления нормального вычислительного процесса.

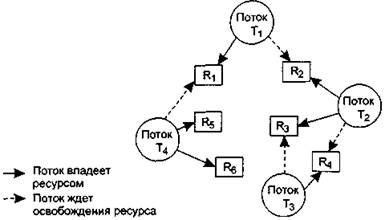


Рис. 8.Взаимная блокировка нескольких потоков

Тупики могут быть предотвращены на стадии написания программ, то есть про­граммы должны быть написаны таким образом, чтобы тупик не мог возникнуть при любом соотношении взаимных скоростей потоков. Так, если бы в примере, показанном на рис. 7, поток А и поток В запрашивали ресурсы в одинако­вой последовательности, то тупик был бы в принципе невозможен. Другой, бо­лее гибкий подход к предотвращению тупиков заключается в том, что ОС каж­дый раз при запуске задач анализирует их потребности в ресурсах и определяет, может ли в данной мультипрограммной смеси возникнуть тупик. Если да, то за­пуск новой задачи временно откладывается. ОС может также использовать опре­деленные правила при назначении ресурсов потокам, например, ресурсы могут выделяться операционной системой в определенной последовательности, общей для всех потоков.

В тех же случаях, когда тупиковую ситуацию не удалось предотвратить, важно быстро и точно ее распознать, поскольку блокированные потоки не выполняют никакой полезной работы. Если тупиковая ситуация образована множеством по­токов, занимающих массу ресурсов, распознавание тупика является нетривиаль­ной задачей. Существуют формальные, программно-реализованные методы рас­познавания тупиков, основанные на ведении таблиц распределения ресурсов и таблиц запросов к занятым ресурсам. Анализ этих таблиц позволяет обнаружить взаимные блокировки.

Если же тупиковая ситуация возникла, то не обязательно снимать с выполнения все заблокированные потоки. Можно снять только часть из них, освободив ресурсы, ожидаемые остальными потоками, можно вернуть некоторые потоки в область под­качки, можно совершить «откат» некоторых потоков до так называемой контроль­ной точки, в которой запоминается вся информация, необходимая для восстановле­ния выполнения программы с данного места. Контрольные точки расставляются в программе в тех местах, после которых возможно возникновение тупика.

**Синхронизирующие объекты ОС**

Рассмотренные выше механизмы синхронизации, основанные на использовании глобальных переменных процесса, обладают существенным недостатком — они не подходят для синхронизации потоков разных процессов. В таких случаях опе­рационная система должна предоставлять потокам системные объекты синхро­низации, которые были бы видны для всех потоков, даже если они принадлежат разным процессам и работают в разных адресных пространствах. Примерами таких синхронизирующих объектов ОС являются системные сема­форы, мьютексы, события, таймеры и другие — их набор зависит от конкретной ОС, которая создает эти объекты по запросам процессов. Чтобы процессы могли разделять синхронизирующие объекты, в разных ОС используются разные методы. Некоторые ОС возвращают указатель на объект. Этот указатель может быть до­ступен всем родственным процессам, наследующим характеристики общего ро­дительского процесса. В других ОС процессы в запросах на создание объектов синхронизации указывают имена, которые должны быть им присвоены. Далее эти имена используются разными процессами для манипуляций объектами син­хронизации. В таком случае работа с синхронизирующими объектами подобна ра­боте с файлами. Их можно создавать, открывать, закрывать, уничтожать.

Кроме того, для синхронизации могут быть использованы такие «обычные» объ­екты ОС, как файлы, процессы и потоки. Все эти объекты могут находиться в двух состояниях: сигнальном и несигнальном — свободном. Для каждого объекта смысл, вкладываемый в понятие «сигнальное состояние», зависит от типа объек­та. Так, например, поток переходит в сигнальное состояние тогда, когда он завер­шается. Процесс переходит в сигнальное состояние тогда, когда завершаются все его потоки. Файл переходит в сигнальное состояние в том случае, когда заверша­ется операция ввода-вывода для этого файла. Для остальных объектов сигнальное состояние устанавливается в результате выполнения специальных системных вы­зовов. Приостановка и активизация потоков осуществляются в зависимости от состояния синхронизирующих объектов ОС.

Потоки с помощью специального системного вызова сообщают операционной сис­теме о том, что они хотят синхронизировать свое выполнение с состоянием неко­торого объекта. Будем далее называть этот системный вызов Wait(X), где X — ука­затель на объект синхронизации. Системный вызов, с помощью которого поток может перевести объект синхронизации в сигнальное состояние, назовем Set(X).

Поток, выполнивший системный вызов Wait(X), переводится операционной сис­темой в состояние ожидания до тех пор, пока объект X не перейдет в сигнальное состояние. Примерами системных вызовов типа Wait() и Set() являются вызовы WaitForSingleObject() и SetEvent() в Windows, DosSemWait() и DosSemSet() в OS/2, sleep() и wakeup() в UNIX.

Поток может ожидать установки сигнального состояния не одного объекта, а не­скольких. При этом поток может попросить ОС активизировать его при установ­ке либо одного из указанных объектов, либо всех объектов. Поток может в каче­стве аргумента системного вызова Wait() указать также максимальное время, которое он будет ожидать перехода объекта в сигнальное состояние, после чего ОС должна его активизировать в любом случае. Может случиться, что установки некоторого объекта в сигнальное состояние ожидают сразу несколько потоков. В зависимости от объекта синхронизации в состояние готовности могут перево­диться либо все ожидающие это событие потоки, либо один из них.

Синхронизация тесно связана с планированием потоков. Во-первых, любое об­ращение потока с системным вызовом Wait(X) влечет за собой действия в под­системе планирования — этот поток снимается с выполнения и помещается в очередь ожидающих потоков, а из очереди готовых потоков выбирается и акти­визируется новый поток. Во-вторых, при переходе объекта в сигнальное состоя­ние (в результате выполнения некоторого потока — либо системного, либо при­кладного) ожидающий этот объект поток (или потоки) переводится в очередь готовых к выполнению потоков. В обоих случаях осуществляется перепланиро­вание потоков, при этом если в ОС предусмотрены изменяемые приоритеты и/или кванты времени, то они пересчитываются по правилам, принятым в этой опера­ционной системе.

Рассмотрим несколько примеров, когда в качестве синхронизирующих объектов используются файлы, потоки и процессы.

Пусть программа приложения построена так, что для выполнения запросов, по­ступающих из сети, основной поток создает вспомогательные серверные потоки.

При поступлении от пользователя команды завершения приложения основной поток должен дождаться завершения всех серверных потоков и только после этого завершиться сам. Следовательно, процедура завершения должна включать вызов Wait(Xl, Х2, ...), где XI, Х2 — указатели на серверные потоки. В результате выпол­нения данного системного вызова основной поток будет переведен в состояние ожидания и останется в нем до тех пор, пока все серверные потоки не перейдут в сигнальное состояние, то есть завершатся. После этого ОС переведет основной поток в состояние готовности. При получении доступа к процессору основной поток завершится.

Другой пример. Пусть выполнение некоторого приложения требует последова­тельных работ-этапов. Для каждого этапа имеется свой отдельный процесс. Сиг­налом для начала работы каждого следующего процесса является завершение предыдущего. Для реализации такой логики работы необходимо в каждом про­цессе, кроме первого, предусмотреть выполнение системного вызова Wait(X), в котором синхронизирующим объектом является предшествующий поток.

Объект-файл, переход которого в сигнальное состояние соответствует заверше­нию операции ввода-вывода с этим файлом, используется в тех случаях, когда поток, инициировавший эту операцию, решает дождаться ее завершения, прежде чем продолжить свои вычисления.

Однако круг событий, с которыми потоку может потребоваться синхронизиро­вать свое выполнение, отнюдь не исчерпывается завершением потока, процесса или операции ввода-вывода. Поэтому в ОС, как правило, имеются и другие, бо­лее универсальные объекты синхронизации, такие как *событие (event), мъютекс (mutex), системный семафор*и другие.

Мьютекс, как и семафор, обычно используется для управления доступом к дан­ным.

В отличие от объектов-потоков, объектов-процессов и объектов-файлов, которые при переходе в сигнальное состояние переводят в состояние готовности все по­токи, ожидающие этого события, объект-мьютекс «освобождает» из очереди ожи­дающих только один поток.

Работа мьютекса хорошо поясняется в терминах «владения». Пусть поток, кото­рый, пытаясь получить доступ к критическим данным, выполнил системный вы­зов Wait(X), где X — указатель на мьютекс. Предположим, что мьютекс находится в сигнальном состоянии, в этом случае поток тут же становится его владельцем, устанавливая его в несигнальное состояние, и входит в критическую секцию. После того как поток выполнил работу с критическими данными, он «отдает» мьютекс, устанавливая его в сигнальное состояние. В этот момент мьютекс сво­боден и не принадлежит ни одному потоку. Если какой-либо поток ожидает его освобождения, то он становится следующим владельцем этого мьютекса, одно­временно мьютекс переходит в несигнальное состояние.

Объект-событие (в данном случае слово «событие» используется в узком смыс­ле, как обозначение конкретного вида объектов синхронизации) обычно исполь­зуется не для доступа к данным, а для того, чтобы оповестить другие потоки о том, что некоторые действия завершены. Пусть, например, в некотором прило­жении работа организована таким образом, что один поток читает данные из файла в буфер памяти, а другие потоки обрабатывают эти данные, затем первый поток считывает новую порцию данных, а другие потоки снова ее обрабатывают и так далее. В начале работы первый поток устанавливает объект-событие в не­сигнальное состояние. Все остальные потоки выполнили вызов Wait(X), где X — указатель события, и находятся в приостановленном состоянии, ожидая наступ­ления этого события. Как только буфер заполняется, первый поток сообщает об этом операционной системе, выполняя вызов Set(X). Операционная система просматривает очередь ожидающих потоков и активизирует все потоки, которые ждут этого события.

*Сигнал*дает возможность задаче реагировать на событие, источником которого может быть операционная система или другая задача. Сигналы выполняют пре­рывание задачи и выполнение заранее предусмотренных действий. Сигналы мо­гут вырабатываться синхронно, то есть как результат работы самого процесса, а могут быть направлены процессу другим процессом, то есть вырабатываться асинхронно. Синхронные сигналы чаще всего приходят от системы прерываний процессора и свидетельствуют о действиях процесса, блокируемых аппаратурой, например деление на нуль, ошибка адресации, нарушение защиты памяти и т. д.

Примером асинхронного сигнала является сигнал с терминала. Во многих ОС предусматривается оперативное снятие процесса с выполнения. Для этого поль­зователь может нажать некоторую комбинацию клавиш (Ctrl+C, Ctrl+Break), в ре­зультате чего ОС вырабатывает сигнал и направляет его активному процессу. Сигнал может поступить в любой момент выполнения процесса (то есть он явля­ется асинхронным), требуя от процесса немедленного завершения работы. В дан­ном случае реакцией на сигнал является безусловное завершение процесса.

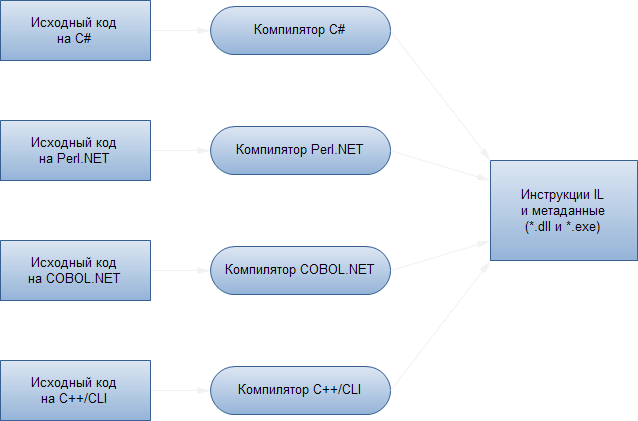
В системе может быть определен набор сигналов. Программный код процесса, которому поступил сигнал, может либо проигнорировать его, либо прореагиро­вать на него стандартным действием (например, завершиться), либо выполнить специфические действия, определенные прикладным программистом. В послед­нем случае в программном коде необходимо предусмотреть специальные систем­ные вызовы, с помощью которых операционная система информируется, какую процедуру надо выполнить в ответ на поступление того или иного сигнала.

Сигналы обеспечивают логическую связь между процессами, а также между про­цессами и пользователями (терминалами). Поскольку посылка сигнала преду­сматривает знание идентификатора процесса, то взаимодействие посредством сигналов возможно только между родственными процессами, которые могут по­лучить данные об идентификаторах друг друга.

В распределенных системах, состоящих из нескольких процессоров, каждый из которых имеет собственную оперативную память, блокирующие переменные, се­мафоры, сигналы и другие аналогичные средства, основанные на разделяемой памяти, оказываются непригодными. В таких системах синхронизация может быть реализована только посредством обмена сообщениями.

**6. Опишите сборки из одного и нескольких файлов.**

Какой бы язык .NET не выбирался для программирования, важно понимать, что хотя двоичные .NET-единицы имеют такое же файловое расширение, как и двоичные единицы СОМ-серверов и неуправляемых программ Win32 (\* . dll или \* . ехе), внутренне они устроены абсолютно по-другому. Например, двоичные .NET-единицы \* .dll не экспортируют методы для упрощения взаимодействия с исполняющей средой СОМ (поскольку .NET — это не СОМ). Более того, они не описываются с помощью библиотек СОМ-типов и не регистрируются в системном реестре. Пожалуй, самым важным является то, что они содержат не специфические, а наоборот, не зависящие от платформы инструкции на промежуточном языке (Intermediate Language — IL), а также метаданные типов. На следующей схеме показано, как все это выглядит:



Отсюда следует, что **сборка (assembly)** — это логическая единица, содержащая скомпилированный код для .NET Framework, т.е. это полностью самодостаточный и скорее логический, нежели физический элемент. Это значит, что он может быть сохранен в более чем одном файле (хотя динамические сборки хранятся в памяти, а вовсе не в файлах). Если сборка хранится в более чем одном файле, то должен существовать один главный файл, содержащий точку входа и описывающий остальные файлы.

Следует отметить, что одна и та же структура сборки используется как для исполняемого кода, так и для кода библиотек. Единственное реальное отличие исполняемой сборки заключается в том, что она содержит главную точку входа программы, тогда как библиотечная сборка — нет.

Важное свойство сборок состоит в том, что они содержат метаданные, которые описывают типы и методы, определенные в ее коде. Помимо этого, сборка хранит в себе метаданные, описывающие ее саму. Эти метаданные, хранимые в области манифеста, позволяют выполнять проверку номера версии сборки и ее целостность.

Тот факт, что сборка содержит метаданные программы, означает, что приложение или другие сборки, которые вызывают код данной, не нуждаются в обращении к реестру или любому другому источнику данных, чтобы узнать, как конкретную сборку следует использовать. Это существенный прорыв по сравнению со старым способом работы СОМ, когда GUID-идентификаторы компонентов и интерфейсов необходимо было извлекать из реестра, а подробности методов и свойств в некоторых случаях читать из библиотеки типов.

Хранение данных в трех различных местах приводило к очевидному риску получить несинхронизированные части, что исключало возможность использования данного компонента другим программным обеспечением. Что касается сборок, то здесь риск подобного рода исключен, потому что все метаданные хранятся вместе с выполняемыми инструкциями программы*.* Следует отметить, что несмотря на то, что сборки могут храниться в нескольких файлах, проблема синхронизации не возникает, поскольку в том же файле, где находится точка входа, хранится и информация о содержимом остальных файлов. То есть даже если один из них подменить или испортить, то это немедленно будет обнаружено, и всей сборке будет запрещено загружаться на выполнение.

Сборки бывают двух видов: разделяемые и приватные.

Приватные сборки

Это простейший тип сборок. Обычно они поставляются с определенным программным обеспечением и предназначены для применения только в его составе. Обычный сценарий получения приватной сборки — это когда приложение поставляется в виде исполняемой программы и множества библиотек, код которых может быть использован только этим приложением.

Система гарантирует, что приватные сборки не будут применяться другим программным обеспечением, потому что приложение может загружать только приватные сборки, находящиеся в той же папке, где хранится и главная исполняемая программа, которая их загружает, либо во вложенных папках.

Поскольку обычно можно рассчитывать на то, что коммерческое программное обеспечение всегда устанавливается в собственный каталог, значит, нет опасности того, что один программный пакет перепишет, модифицирует или непреднамеренно загрузит приватные сборки, принадлежащие другому пакету. Поскольку приватные сборки могут быть использованы только тем пакетом программного обеспечения, для которого они предназначены, имеется возможность управлять тем программным обеспечением, которое их использует. Отсюда уменьшается необходимость в некоторых предосторожностях для обеспечения безопасности, поскольку нет риска, например, того, что какое-то коммерческое программное обеспечение перепишет сборки их новыми версиями (за исключением случаев, когда программное обеспечение специально разработано для нанесения определенного вреда). Не существует также проблемы коллизии имен. Если окажется, что классы в приватной сборке имеют те же имена, что и классы в чьей-то чужой приватной сборке, то это не имеет значения, поскольку каждое приложение будет в состоянии видеть только один набор собственных приватных сборок.

Поскольку приватная сборка полностью самодостаточна, процесс ее развертывания весьма прост. Вы просто помещаете соответствующий файл (или файлы) в соответствующую папку системы (никаких записей вносить в реестр не потребуется). Этот процесс известен как установка с нулевым воздействием или установка с помощью хсору (zero impact (хсору) installation).

Разделяемые сборки

Назначение разделяемых сборок — служить библиотеками общего применения, которые могут использоваться любым другим приложением. Поскольку любое другое приложение может получить доступ к разделяемой сборке, возникает необходимость в некоторых предосторожностях, чтобы исключить описанные ниже риски.

Коллизия имен, когда разделяемая сборка, поставленная другой компанией, реализует типы с теми же именами, что используются в вашей сборке. Поскольку клиентский код теоретически может иметь доступ к двум таким сборкам одновременно, это может представлять серьезную проблему.

Риск того, что данная сборка будет перезаписана другой версией той же сборки, и новая версия окажется несовместимой с некоторым существующим клиентским кодом.

Решение этих проблем включает размещение разделенных сборок в специальном поддереве каталогов файловой системы, известном под названием глобальный кэш сборок (global assembly cache — GAC). В отличие от приватных сборок, это не может быть сделано простым копированием сборки в определенную папку — сборку понадобится специальным образом установить в кэше GAC. Данный процесс может быть реализован с помощью множества утилит .NET и включает в себя выполнение необходимых проверок устанавливаемой сборки, а также создание небольшой иерархии папок в пределах GAC, используемых для обеспечения целостности сборок.

Чтобы исключить коллизии имен, разделяемым сборкам назначаются имена, основанные на шифровании индивидуальным ключом (в отличие от приватных сборок, которым дается имя, совпадающее с именем главного файла). Это имя, известное как строгое имя, гарантированно является уникальным, и оно должно указываться приложением, ссылающимся на данную разделенную сборку.

Проблемы, связанные с риском перезаписи сборки, решаются указанием информации о версии в манифесте сборки, а также возможностью параллельных установок различных версий данной сборки.

Однофайловые и многофайловые сборки

В большом количестве случаев между сборками .NET и файлами двоичного кода (\*.dll или \*.ехе) соблюдается простое соответствие "один к одному". Следовательно, получается, что при построении \*.dll-библиотеки .NET, можно спокойно полагать, что файл двоичного кода и сборка представляют собой одно и то же, и что, аналогичным образом, при построении исполняемого приложения для настольной системы на файл \*.ехе можно ссылаться как на саму сборку. Однако это не совсем так. С технической точки зрения, сборка, состоящая из одного единственного модуля \*.dll или \*.ехе, называется однофайловой сборкой. В однофайловых сборках все необходимые CIL-инструкции, метаданные и манифесты содержатся в одном автономном четко определенном пакете.

Многофайловые сборки, в свою очередь, состоят из множества файлов двоичного кода .NET, каждый из которых называется модулем (module). При построении многофайловой сборки в одном из ее модулей (называемом первичным или главным (primary) модулем) содержится манифест всей самой сборки (и, возможно, СIL-инструкции и метаданные по различным типам), а во всех остальных — манифест, CIL-инструкции и метаданные типов, охватывающие уровень только соответствующего модуля. Как нетрудно догадаться, в главном модуле содержится описание набора требуемых дополнительных модулей внутри манифеста сборки.

Таким образом, сборки являются исполняемым файлом или файлом библиотеки динамической компоновки, содержащим инструкции на промежуточном языке (Intermediate Language - IL), и метаданные, предназначенные для исполнения платформой .NET.

**61. Создайте приложение для изменения основных свойств компонента Bevel. Изменения должны отображаться**

using Bevel;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Windows.Forms;

using Bevel;

using BevelStyle = ClassLibrary1.BevelStyle;

using BevelType = ClassLibrary1.BevelType;

namespace BevelWinForm

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

public bool red = false;

public bool black = false;

public bool white = false;

private void bevelControl1\_Click(object sender, EventArgs e)

{

if (red == true)

{

bevelControl1.BackColor = Color.Red;

bevelControl1.Refresh();

}

else if (black == true)

{

bevelControl1.BackColor = Color.Black;

bevelControl1.Refresh();

}

}

private void radioButton1\_Click(object sender, EventArgs e)

{

red = true;

bevelControl1.BackColor = Color.Red;

bevelControl1.Refresh();

}

private void radioButton2\_Click(object sender, EventArgs e)

{

black = true;

bevelControl1.BackColor = Color.Black;

bevelControl1.Refresh();

}

private void rd2\_Click(object sender, EventArgs e)

{

white = true;

bevelControl1.BackColor = Color.White;

bevelControl1.Refresh();

}

private void radioButton1\_CheckedChanged(object sender, EventArgs e)

{

bevelControl1.BevelStyle = BevelStyle.Raised;

bevelControl1.Refresh();

}

private void radioButton2\_CheckedChanged(object sender, EventArgs e)

{

bevelControl1.BevelStyle = BevelStyle.Lowered;

bevelControl1.Refresh();

}

private void radioButton3\_CheckedChanged(object sender, EventArgs e)

{

bevelControl1.BevelType = BevelType.Box;

bevelControl1.Refresh();

}

private void radioButton7\_CheckedChanged(object sender, EventArgs e)

{

bevelControl1.BevelType = BevelType.Frame;

bevelControl1.Refresh();

}

private void radioButton8\_CheckedChanged(object sender, EventArgs e)

{

bevelControl1.BevelType = BevelType.TopLine;

bevelControl1.Refresh();

}

private void radioButton9\_CheckedChanged(object sender, EventArgs e)

{

bevelControl1.BevelType = BevelType.BottomLine;

bevelControl1.Refresh();

}

private void radioButton10\_CheckedChanged(object sender, EventArgs e)

{

bevelControl1.BevelType = BevelType.LeftLine;

bevelControl1.Refresh();

}

private void radioButton11\_CheckedChanged(object sender, EventArgs e)

{

bevelControl1.BevelType = BevelType.RightLine;

bevelControl1.Refresh();

}

private void radioButton12\_CheckedChanged(object sender, EventArgs e)

{

bevelControl1.BevelType = BevelType.Spacer;

bevelControl1.Refresh();

}

private void radioButton6\_CheckedChanged(object sender, EventArgs e)

{

label1.BackColor = Color.Red;

label1.ForeColor = Color.White;

label1.Refresh();

}

private void radioButton5\_CheckedChanged(object sender, EventArgs e)

{

label1.BackColor = Color.Black;

label1.ForeColor = Color.White;

label1.Refresh();

}

private void radioButton4\_CheckedChanged(object sender, EventArgs e)

{

label1.BackColor = Color.White;

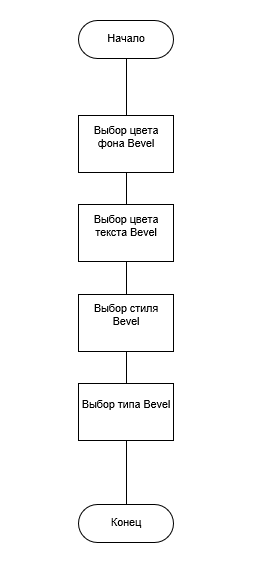
label1.ForeColor = Color.Black;

label1.Refresh();

}

}

}

****

**76. Создайте приложение для построения диаграммы отражающей число читателей в трех библиотеках. Тип диаграммы: круговая или гистограмма выбирается пользователем. Число читателей в каждой библиотеке вводите с формы.**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Windows.Forms;

using System.Windows.Forms.DataVisualization.Charting;

namespace BiblioDiagAPP

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

if (radioButton2.Checked)

{

chart1.Series[0].Points.Clear();

chart1.Series[1].Points.Clear();

chart1.Series[2].Points.Clear();

chart1.Series[0].ChartType = SeriesChartType.Pie;

chart1.Series[1].ChartType = SeriesChartType.Pie;

chart1.Series[2].ChartType = SeriesChartType.Pie;

chart1.Series[1].Color = Color.Gold;

chart1.Series[2].Color = Color.DarkRed;

double number = double.Parse(textBox1.Text);

double num2 = double.Parse(textBox2.Text);

double num3 = double.Parse(textBox3.Text);

chart1.Series[0].Points.AddXY("1", number);

chart1.Series[0].LegendText = "Библиотека 1";

chart1.Series[0].Points.AddXY("2", num2);

chart1.Series[0].LegendText = "Библиотека 2";

chart1.Series[0].Points.AddXY("3", num3);

chart1.Series[0].LegendText = "Библиотека 3";

}

else

{

double number = double.Parse(textBox1.Text);

double num2 = double.Parse(textBox2.Text);

double num3 = double.Parse(textBox3.Text);

chart1.Series[0].Points.AddY(number);

chart1.Series[0].LegendText = "Библиотека 1";

chart1.Series[1].Points.AddY(num2);

chart1.Series[1].LegendText = "Библиотека 2";

chart1.Series[2].Points.AddY(num3);

chart1.Series[2].LegendText = "Библиотека 3";

}

}

private void radioButton2\_CheckedChanged(object sender, EventArgs e)

{

chart1.Series[0].Points.Clear();

chart1.Series[1].Points.Clear();

chart1.Series[2].Points.Clear();

chart1.Series[0].ChartType = SeriesChartType.Pie;

chart1.Series[1].ChartType = SeriesChartType.Pie;

chart1.Series[2].ChartType = SeriesChartType.Pie;

chart1.Series[1].Color = Color.Gold;

chart1.Series[2].Color = Color.DarkRed;

}

private void radioButton1\_CheckedChanged(object sender, EventArgs e)

{

chart1.Series[0].ChartType = SeriesChartType.Column;

chart1.Series[1].ChartType = SeriesChartType.Column;

chart1.Series[2].ChartType = SeriesChartType.Column;

}

}

}

