Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра электронно-вычислительных средств

Д. С. Лихачёв

РАЗРАБОТКА МНОГОПОТОЧНЫХ ПРИЛОЖЕНИЙ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники для специальности 1-40 02 02 «Электронные вычислительные средства» в качестве пособия

Минск БГУИР 2014

УДК 004.45(076.5)
ББК 32.973.26-018.2я7
Л65

Рецензенты:

кафедра информатики учреждения образования
«Минский государственный высший радиотехнический колледж»
(протокол №12 от 28.06.2012);

заведующий лабораторией государственного научного учреждения
«Объединенный институт проблем информатики
Национальной академии наук Беларуси»,
доктор технических наук, профессор Г. И. Алексеев

Лихачёв, Д. С.

Л65 Разработка многопоточных приложений. Лабораторный практикум : пособие / Д. С. Лихачёв. – Минск : БГУИР, 2014. – 48 с. : ил.

ISBN 978-985-488-901-6.

Приведены описания двух лабораторных работ по дисциплине «Системное программирование», посвященных разработке многопоточных приложений в ОС Windows, для студентов специальности 1-40 02 02 «Электронные вычислительные средства».

Автор выражает искреннюю благодарность студентке гр. 010902 Елене Орешко за помощь в разработке пособия.

УДК 004.45(076.5)
ББК 32.973.26-018.2я7

ISBN 978-985-488-901-6

Содержание

ЛАБОРАТОРНАЯ РАБОТА
«Создание многопоточных приложений»

Цель работы: ознакомиться с механизмом многозадачности в современных операционных системах, получить навыки по разработке многопоточных приложений в ОС Windows.

1.1 Теоретические сведения

1.1.1 Понятия мультипрограммирования и многозадачности

Важнейшей функцией операционной системы является организация рационального использования всех ее аппаратных и информационных ресурсов. К основным ресурсам могут быть отнесены процессоры, память, внешние устройства, данные и программы. Главные сложности возникают в мультипрограммных операционных системах (ОС), в которых за ресурсы конкурируют сразу несколько приложений. Именно поэтому большая часть всех проблем относится к мультипрограммным системам.

Мультипрограммирование, или *многозадачность (multitasking)* – это способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются сразу несколько программ [1]. Эти программы совместно используют не только процессор, но и другие ресурсы компьютера: оперативную и внешнюю память, устройства ввода-вывода, данные.

1.1.2 Понятия «процесс» и «поток»

Чтобы поддерживать мультипрограммирование, ОС должна определить и оформить для себя те внутренние единицы работы, между которыми будет разделяться процессор и другие ресурсы компьютера. В настоящее время в большинстве операционных систем определены два типа единиц работы. Более крупная единица работы, обычно носящая название *процесса,* или *задачи,* требует предварительного выполнения нескольких более мелких работ, для обозначения которых используют термины *«поток»,* или *«нить»* [1].

Таким образом, процесс – это объект, с которым сопоставляются все ресурсы, в том числе и адресное пространство, т. е. он является структурой в памяти. В адресном пространстве процесса, кроме кодов и данных, находятся также потоки.

Работа вычислительной системы заключается в выполнении некоторой программы. Поэтому и с процессом, и с потоком связывается определенный программный код, который для этих целей оформляется в виде исполняемого модуля. Чтобы этот программный код мог быть выполнен, его необходимо загрузить в оперативную память, возможно, выделить некоторое место на диске для хранения данных,

предоставить доступ к устройствам ввода-вывода, например, к последовательному порту для получения данных по подключенному к этому порту модему и т. д. Кроме того, необходимо предоставление программе процессорного времени, то есть времени, в течение которого процессор выполняет коды данной программы.

Для того чтобы процессы не могли вмешаться в распределение ресурсов, а также не могли повредить коды и данные друг друга, важнейшей задачей ОС является *изоляция* одного процесса от другого. Для этого операционная система обеспечивает каждый процесс отдельным виртуальным адресным пространством, так что ни один процесс не может получить прямого доступа к командам и данным другого процесса.

Виртуальное адресное пространство процесса – это совокупность адресов, которыми может манипулировать программный модуль процесса [1]. Операционная система отображает виртуальное адресное пространство процесса в отведенной процессу физической памяти.

При необходимости взаимодействия процессы обращаются к операционной системе, которая, выполняя функции посредника, предоставляет им средства межпроцессной связи – конвейеры, почтовые ящики, разделяемые секции памяти и некоторые другие.

Потоки возникли в операционных системах как средство распараллеливания вычислений. Конечно, задача распараллеливания вычислений в рамках одного приложения может быть решена и традиционными способами.

Во-первых, программист может взять на себя сложную задачу организации параллелизма, выделив в приложении некоторую подпрограмму-диспетчер, которая периодически передает управление той или иной ветви вычислений. При этом программа получается логически весьма запутанной, с многочисленными передачами управления, что существенно затрудняет ее отладку и модификацию.

Во-вторых, решением является создание для одного приложения нескольких процессов для каждой из параллельных работ. Однако использование для создания процессов стандартных средств ОС не позволяет учесть тот факт, что эти процессы решают единую задачу, а значит, имеют много общего между собой. А операционная система при таком подходе будет рассматривать эти процессы наравне со всеми остальными процессами и с помощью универсальных механизмов обеспечивать их изоляцию друг от друга. Кроме того, на создание каждого процесса ОС тратит определенные системные ресурсы, которые в данном случае неоправданно дублируются – каждому процессу выделяются собственное виртуальное адресное пространство, физическая память, закрепляются устройства ввода-вывода и т. п.

Таким образом, для эффективной обработки задач в любой операционной системе наряду с процессами необходим специальный механизм распараллеливания вычислений, который учитывал бы тесные связи между отдельными ветвями вычислений одного и того же приложения. Для этих целей современные ОС используют механизм *многопоточной обработки (multithreading)* [1]. При этом вводится новая единица работы – поток выполнения, а понятие «процесс» в значительной степени меняет смысл. Понятию «поток» соответствует последовательный переход процессора от одной команды программы к другой. ОС распределяет процессорное время между потоками. Процессу ОС назначает адресное пространство и набор ресурсов, которые совместно используются всеми его потоками.

Создание потоков требует от ОС меньших накладных расходов, чем процессов. В отличие от процессов, которые принадлежат разным или конкурирующим приложениям, все потоки одного процесса всегда принадлежат одному приложению, поэтому все потоки одного процесса используют общие файлы, таймеры, устройства, одну и ту же область оперативной памяти, одно и то же адресное пространство. Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к любому виртуальному адресу процесса, один поток может использовать стек другого потока. Между потоками одного процесса нет полной защиты, потому что, во-первых, это невозможно, а во-вторых, не нужно. С другой стороны, потоки разных процессов по-прежнему хорошо защищены друг от друга.

Таким образом, мультипрограммирование более эффективно на уровне потоков, а не процессов [1].

Использование потоков связано не только со стремлением повысить производительность системы за счет параллельных вычислений, но и с целью создания более читабельных, логичных программ. Введение нескольких потоков выполнения упрощает программирование.

Наибольший эффект от введения многопоточной обработки достигается в мультипроцессорных системах, в которых потоки, в том числе и принадлежащие одному процессу, могут выполняться на разных процессорах действительно параллельно (а не псевдопараллельно).

1.1.3 Особенности реализации механизма мультипрограммирования для повышения эффективности использования вычислительной системы

Мультипрограммирование по своей сути призвано повысить эффективность использования вычислительной системы, однако эффективность может пониматься по-разному. Наиболее характерными критериями эффективности вычислительных систем являются [1]:

1) *пропускная способность* – количество задач, выполняемых вычислительной системой в единицу времени;

2) *удобство работы пользователей*;

3) *реактивность системы* – способность системы выдерживать заранее заданные (возможно, очень короткие) интервалы времени между запуском программы и получением результата.

В зависимости от выбранного критерия эффективности ОС делятся на системы пакетной обработки, системы разделения времени и системы реального времени [1]. Каждый тип ОС имеет специфические внутренние механизмы и особые области применения. Некоторые операционные системы могут поддерживать одновременно несколько режимов, например, часть задач может выполняться в режиме пакетной обработки, а часть – в режиме реального времени или в режиме разделения времени.

Системы пакетной обработки

При использовании мультипрограммирования для повышения пропускной способности компьютера главной целью является минимизация простоев всех устройств компьютера и прежде всего центрального процессора. Такие простои могут возникать из-за приостановки задачи по ее внутренним причинам, связанным, например, с ожиданием ввода данных для обработки. При возникновении такого рода блокировки выполняемой задачи выполняется переключение процессора на выполнение другой задачи, у которой есть данные для обработки. Такая концепция мультипрограммирования положена в основу пакетных систем.

Системы пакетной обработки предназначались для решения задач в основном вычислительного характера, не требующих быстрого получения результатов. Главной целью и критерием эффективности систем пакетной обработки является максимальная пропускная способность, то есть решение максимального числа задач в единицу времени.

Для достижения этой цели в системах пакетной обработки используется следующая схема функционирования: в начале работы формируется пакет заданий, из этого пакета заданий формируется мультипрограммная смесь, т. е. множество одновременно выполняемых задач. Для одновременного выполнения выбираются задачи так, чтобы обеспечивалась сбалансированная загрузка всех устройств вычислительной машины. Таким образом, выбирается «выгодное» задание. В вычислительных системах, работающих под управлением пакетных ОС, невозможно гарантировать выполнение того или иного задания в течение определенного периода времени.

Совмещение во времени операций ввода-вывода и вычислений может достигаться разными способами. Один из них – использование компьютеров, имеющих специализированный процессор ввода-вывода (канал). Обычно канал имеет

систему команд, которые предназначены для управления внешними устройствами, например, «проверить состояние устройства», «установить магнитную головку», «установить начало листа», «напечатать строку». В системе команд центрального процессора предусматривается специальная инструкция, с помощью которой каналу передаются параметры и указания на то, какую программу ввода-вывода он должен выполнить. Начиная с этого момента центральный процессор и канал могут работать параллельно (рисунок 1, а).

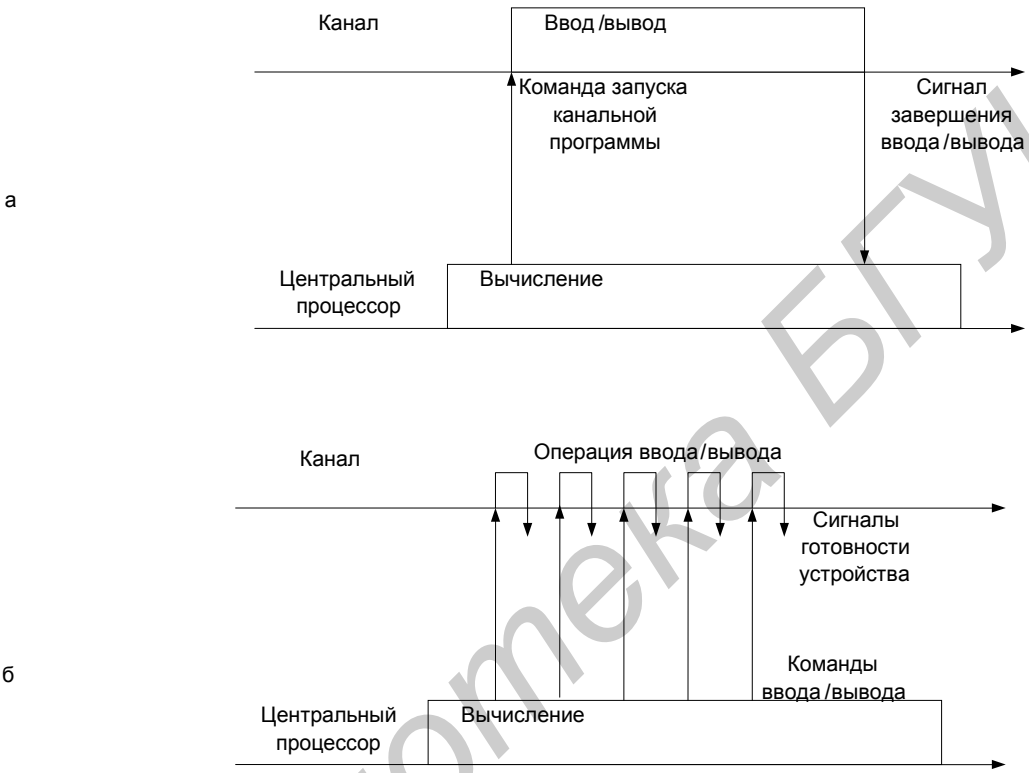


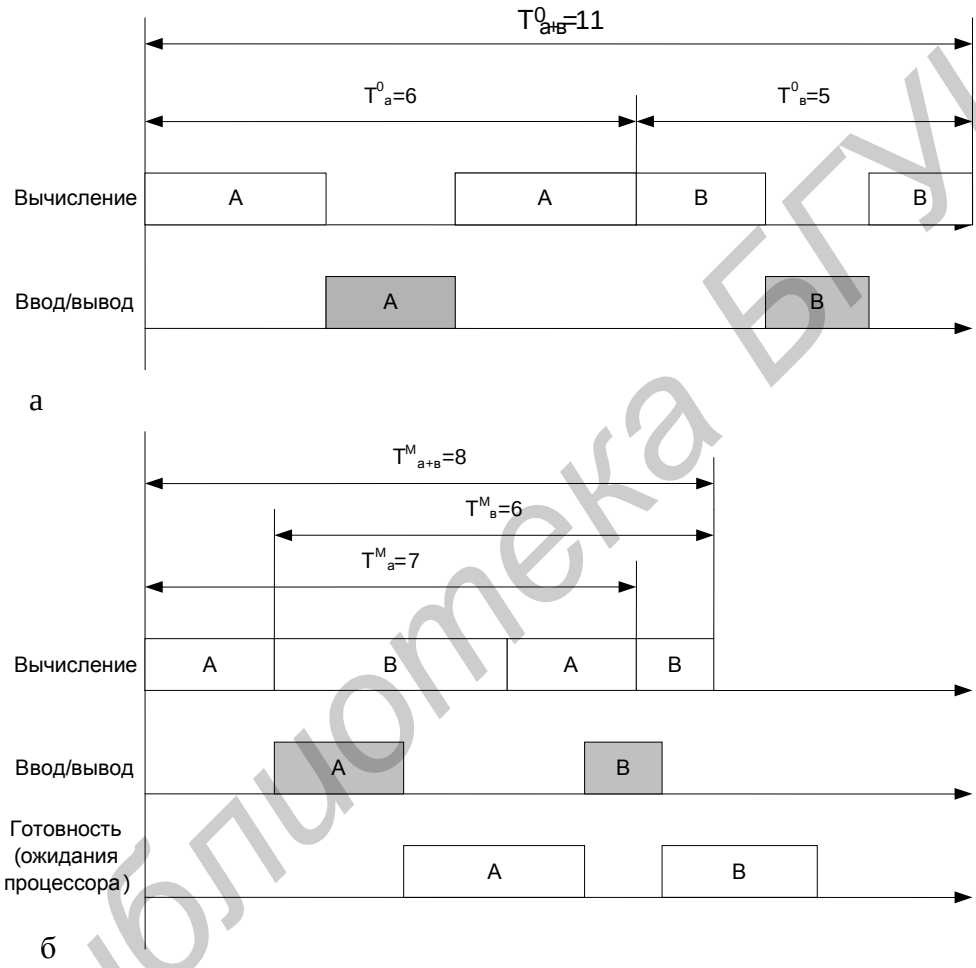
Рисунок 1 – Параллельное выполнение вычислений и операций ввода-вывода [1]

Другой способ совмещения вычислений с операциями ввода-вывода реализуется в компьютерах, в которых внешние устройства управляются не процессором ввода-вывода, а контроллерами. Каждое внешнее устройство (или группа внешних устройств одного типа) имеет свой собственный контроллер, который автономно отрабатывает команды, поступающие от центрального процессора. При этом контроллер и центральный процессор работают асинхронно. Контроллер выполняет свои команды управления устройствами существенно медленнее, чем центральный процессор – свои. Это обстоятельство используется для организации параллельного выполнения вычислений и операций ввода-вывода: в промежутке между передачей команд контроллеру центральный процессор может выполнять

вычисления (рисунок. 1, б). Контроллер может сообщить центральному процессору о том, что он готов принять следующую команду, сигналом прерывания либо центральный процессор узнает об этом, периодически опрашивая состояние контроллера.

Если же в системе выполняются одновременно несколько задач, появляется возможность совмещения вычислений одной задачи с вводом-выводом другой.

Общее время выполнения смеси задач часто оказывается меньше, чем их суммарное время последовательного выполнения (рисунок 2, а).



а – в однопрограммной системе; б – в мультипрограммной системе [1]

Рисунок 2 – Время выполнения двух задач

Однако при совместном использовании процессора в системе могут возникать ситуации, когда задача готова выполняться, но процессор занят выполнением

другой задачи. В таких случаях задача, завершившая ввод-вывод, вынуждена ждать освобождения процессора, и это удлиняет срок ее выполнения.

В системах пакетной обработки переключение процессора с выполнения одной задачи на выполнение другой происходит по инициативе самой активной задачи. При этом одна задача может надолго занять процессор и выполнение интерактивных задач станет невозможным. Взаимодействие пользователя с вычислительной машиной, на которой установлена система пакетной обработки, сводится к тому, что он приносит задание, отдает его диспетчеру-оператору, а в конце дня после выполнения всего пакета заданий получает результат. Очевидно, что такой порядок повышает эффективность функционирования аппаратуры, но снижает эффективность работы пользователя.

Особенности реализации механизма мультипрограммирования в системах разделения времени

Повышение удобства и эффективности работы пользователя – цель исследования систем разделения времени. В них пользователям (или одному пользователю) предоставляется возможность интерактивной работы сразу с несколькими приложениями. Для этого каждое приложение должно регулярно получать возможность «общения» с пользователем. При этом возможности диалога пользователя с приложением ограничены.

В системах разделения времени ОС всем приложениям попеременно выделяется квант процессорного времени, в течение которого программа обрабатывает поступившие в ее адрес сообщения, после чего вне зависимости от состояния программы система забирает управление от программы и передает ее другой программе.

Таким образом, пользователи, запустившие программы на выполнение, получают возможность поддерживать с ними диалог.

Системы разделения времени призваны исправить основной недостаток систем пакетной обработки – изоляцию пользователя-программиста от процесса выполнения его задач.

Системы разделения времени обладают меньшей пропускной способностью, чем системы пакетной обработки, так как на выполнение принимается каждая запущенная пользователем задача, а не та, которая «выгодна» системе. Это вполне соответствует тому, что критерием эффективности систем разделения времени является не максимальная пропускная способность, а удобство и эффективность работы пользователя. Вместе с тем мультипрограммное выполнение интерактивных приложений повышает и пропускную способность компьютера (пусть и не в такой степени, как пакетные системы). Аппаратура загружается лучше, поскольку в то время, пока одно приложение ждет сообщения пользователя, другие приложения могут обрабатываться процессором.

Мультипрограммирование в системах реального времени

Еще одна разновидность мультипрограммирования используется в *системах реального времени,* предназначенных для управления от компьютера различными техническими объектами или технологическими процессами [1]. Критерием эффективности этих систем является способность выдерживать заранее заданные интервалы времени между запуском программы и получением результата (т. е. время реакции системы). Это свойство системы называется реактивностью.

В системах реального времени мультипрограммная смесь представляет собой фиксированный набор заранее разработанных программ, а выбор программы на выполнение осуществляется по прерываниям (исходя из текущего состояния объекта) или в соответствии с расписанием плановых работ.

Способность аппаратуры компьютера и ОС к быстрому ответу зависит в основном от скорости переключения с одной задачи на другую и, в частности, от скорости обработки сигналов прерывания.

В системах реального времени не стремятся максимально загружать все устройства, наоборот, обычно закладывается некоторый «запас» вычислительной мощности на случай пиковой нагрузки.

Если система реального времени не спроектирована для поддержки пиковой нагрузки, то может случиться так, что система не справится с работой именно тогда, когда она нужна в наибольшей степени.

1.1.4 Планирование процессов и потоков

Одной из основных подсистем мультипрограммной ОС, непосредственно влияющей на функционирование вычислительной машины, является подсистема управления процессами и потоками, которая занимается их созданием и уничтожением, поддерживает взаимодействие между ними, а также распределяет процессорное время между несколькими одновременно существующими в системе процессами и потоками.

Подсистема управления процессами и потоками ответственна за *обеспечение процессов необходимыми ресурсами* [1]. ОС поддерживает в памяти специальные информационные структуры, в которые записывает, какие ресурсы выделены каждому процессу. Она может назначить процессу ресурсы в единоличное пользование или в совместное пользование с другими процессами. Подсистема управления процессами взаимодействует с другими подсистемами ОС, ответственными за управление ресурсами, такими, как подсистема управления памятью, подсистема ввода-вывода, файловая система.

Когда в системе одновременно выполняется несколько независимых задач, то возникают дополнительные проблемы. Согласование скоростей потоков также очень важно для предотвращения эффекта «гонок» (когда несколько потоков пы-

таются изменить один и тот же файл), взаимных блокировок или других коллизий, которые возникают при совместном использовании ресурсов. *Синхронизация* потоков является одной из важных функций подсистемы управления процессами и потоками.

Каждый раз, когда процесс завершается, ОС предпринимает шаги, чтобы «зачистить следы» его пребывания в системе. Подсистема управления процессами закрывает все файлы, с которыми работал процесс, освобождает области оперативной памяти, отведенные под коды, данные и системные информационные структуры процесса. Выполняется коррекция всевозможных очередей ОС и списков ресурсов, в которых имелись ссылки на завершаемый процесс.

1.1.5 Создание процессов и потоков

Создать процесс – это прежде всего означает создать *описатель процесса,* в качестве которого выступает одна или несколько информационных структур, содержащих все сведения о процессе, необходимые операционной системе для управления им [1]. В число таких сведений могут входить, например, идентификатор процесса, данные о расположении в памяти исполняемого модуля, степень привилегированности процесса (приоритет и права доступа) и т. п.

Создание процесса включает загрузку кодов и данных исполняемой программы данного процесса с диска в оперативную память. Для этого ОС должна обнаружить местоположение такой программы на диске, перераспределить оперативную память и выделить память исполняемой программе нового процесса. Затем необходимо считать программу в выделенные для нее участки памяти и, возможно, изменить параметры программы в зависимости от размещения в памяти. В системах с виртуальной памятью в начальный момент может загружаться только часть кодов и данных процесса, чтобы «подкачивать» остальные по мере необходимости.

В многопоточной системе при создании процесса ОС создает для каждого процесса как минимум один поток выполнения. При создании потока так же, как и при создании процесса, операционная система генерирует специальную информационную структуру – описатель потока, который содержит идентификатор потока, данные о правах доступа и приоритете, о состоянии потока и другую информацию. В исходном состоянии поток (или процесс, если речь идет о системе, в которой понятие «поток» не определяется) находится в приостановленном состоянии. Момент выборки потока на выполнение осуществляется в соответствии с принятым в данной системе правилом предоставления процессорного времени и с учетом всех существующих в данный момент потоков и процессов. Необходимым условием активизации потока процесса является также наличие места в оперативной памяти для загрузки его исполняемого модуля.

1.1.6 Состояния потока

ОС выполняет планирование потоков, принимая во внимание их состояние. В мультипрограммной системе поток может находиться в одном из трех основных состояний:

1) *выполнение* – активное состояние потока, во время которого поток обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;

2) *ожидание* – пассивное состояние потока, находясь в котором, поток заблокирован по своим внутренним причинам (ждет осуществления некоторого события, например, завершения операции ввода-вывода, получения сообщения от другого потока или освобождения какого-либо необходимого ему ресурса);

3) *готовность* – также пассивное состояние потока, но в этом случае поток заблокирован в связи с внешним по отношению к нему обстоятельством (имеет все требуемые для него ресурсы, готов выполняться, однако процессор занят выполнением другого потока).

В течение своей жизни каждый поток переходит из одного состояния в другое в соответствии с алгоритмом планирования потоков, принятым в данной операционной системе.

Рассмотрим типичный граф состояния потока (рисунок 3). Только что созданный поток находится в состоянии готовности, он готов к выполнению и стоит в очереди к процессору. Когда в результате планирования подсистема управления потоками принимает решение об активизации данного потока, он переходит в состояние выполнения и находится в нем до тех пор, пока либо он сам освободит процессор, перейдя в состояние ожидания какого-нибудь события, либо будет принудительно «вытеснен» из процессора, например, вследствие исчерпания отведенного данному потоку кванта процессорного времени. В последнем случае поток возвращается в состояние готовности. В это же состояние поток переходит из состояния ожидания, после того как ожидаемое событие произойдет.

В состоянии выполнения в однопроцессорной системе может находиться не более одного потока, а в каждом из состояний ожидания и готовности – несколько потоков. Эти потоки образуют очереди соответственно ожидающих и готовых потоков. Очереди потоков организуются путем объединения в списки описателей отдельных потоков. Таким образом, каждый описатель потока, кроме всего прочего, содержит по крайней мере один указатель на другой описатель, соседствующий с ним в очереди. Такая организация очередей позволяет легко их переупорядочивать, включать и исключать потоки, переводить потоки из одного состояния в другое. Если предположить, что на рисунке 4 показана очередь готовых потоков, то запланированный порядок выполнения выглядит так: A, B, E, D, C.

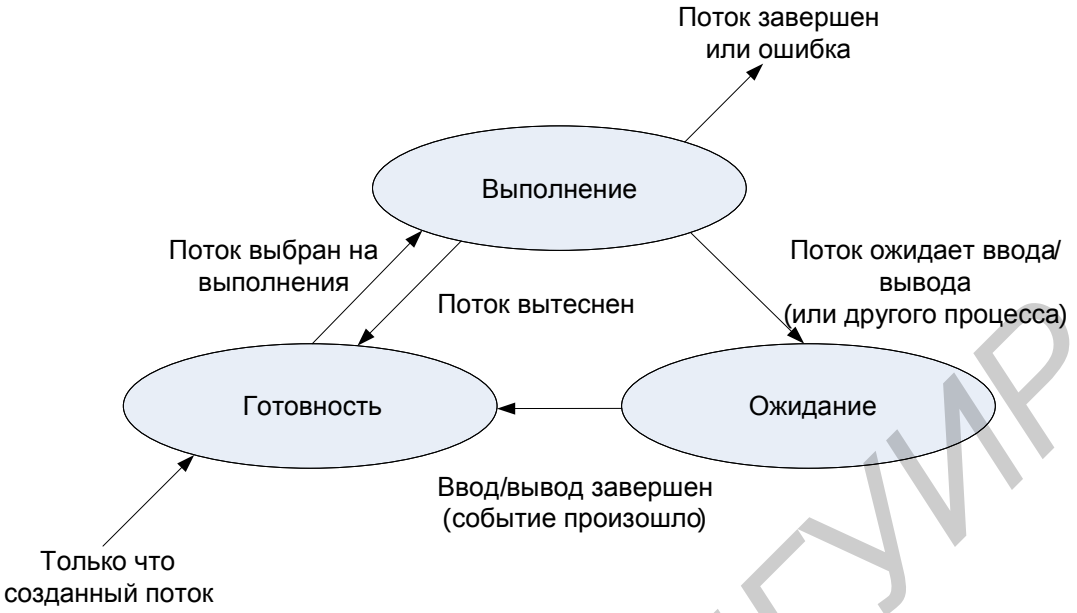


Рисунок 3 – Граф состояний потока в многозадачной среде [1]

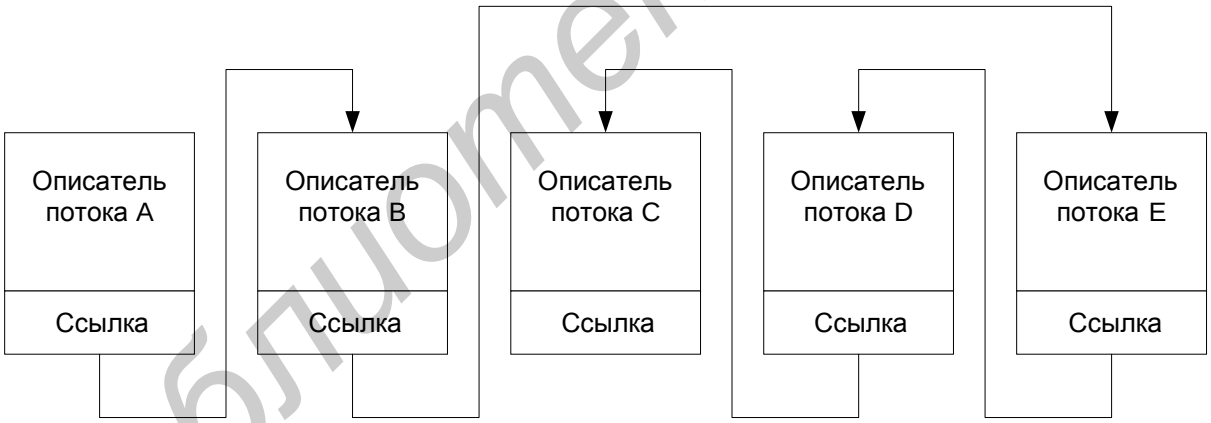


Рисунок 4 – Очередь потоков [1]

1.1.7 Вытесняющие и невытесняющие алгоритмы планирования

Все множество алгоритмов планирования можно разделить на два класса: вытесняющие и невытесняющие алгоритмы планирования.

1) *Невытесняющие (non-preemptive)* алгоритмы основаны на том, что активному потоку позволяется выполняться, пока он сам, по собственной инициативе, не отдаст управление операционной системе для того, чтобы та выбрала из очереди очередной готовый к выполнению поток.

2) *Вытесняющие (preemptive)* алгоритмы – это такие способы планирования потоков, в которых решение о переключении процессора с выполнения одного потока на выполнение другого потока принимается операционной системой, а не активной задачей.

Основным различием между вытесняющими и невытесняющими алгоритмами является степень централизации механизма планирования потоков. При вытесняющем мультипрограммировании функции планирования потоков целиком сосредоточены в операционной системе. При этом операционная система выполняет следующие функции: определяет момент снятия с выполнения активного потока, запоминает его контекст, выбирает из очереди готовых потоков следующий, запускает новый поток на выполнение, загружая его контекст.

При невытесняющем мультипрограммировании механизм планирования распределен между операционной системой и прикладными программами. Прикладная программа, получив управление от операционной системы, сама определяет момент завершения очередного цикла своего выполнения и только затем передает управление ОС с помощью какого-либо системного вызова. ОС формирует очереди потоков и выбирает в соответствии с некоторым правилом (например, с учетом приоритетов) следующий поток на выполнение. Такой механизм создает проблемы как для пользователей, так и для разработчиков приложений.

Для пользователей это означает, что управление системой теряется на произвольный период времени, который определяется приложением (а не пользователем). Поэтому разработчики приложений для операционной среды с невытесняющей многозадачностью вынуждены, возлагая на себя часть функций планировщика, создавать приложения так, чтобы они выполняли свои задачи небольшими частями. Подобный метод разделения времени между задачами существенно затрудняет разработку программ и предъявляет повышенные требования к квалификации программиста. Программист должен обеспечить «дружественное» отношение своей программы к другим выполняемым одновременно с ней программам. Для этого в программе должны быть предусмотрены частые передачи управления операционной системе. Крайним проявлением «не дружественности» приложения является его зависание, которое приводит к общему краху системы. В си-

стемах с вытесняющей многозадачностью такие ситуации, как правило, исключены, так как центральный планирующий механизм имеет возможность снять зависшую задачу с выполнения.

Однако распределение функций планирования потоков между системой и приложениями при определенных условиях может быть и преимуществом, потому что дает возможность разработчику приложений самому проектировать алгоритм планирования, наиболее подходящий для данного фиксированного набора задач. Существенным преимуществом невытесняющего планирования является более высокая скорость переключения с потока на поток.

Почти во всех современных операционных системах, ориентированных на высокопроизводительное выполнение приложений, реализованы вытесняющие алгоритмы планирования потоков (процессов) [1, 2].

1.1.8 Планирование и диспетчеризация потоков

На протяжении существования процесса выполнение его потоков может быть многократно прервано и продолжено. Переход от выполнения одного потока к другому осуществляется в результате планирования и диспетчеризации [1]. Работа по определению того, в какой момент необходимо прервать выполнение текущего активного потока и какому потоку предоставить возможность выполняться, называется *планированием* [1]. Планирование потоков осуществляется на основе информации, хранящейся в описателях процессов и потоков. При планировании могут приниматься во внимание приоритет потоков, время их ожидания в очереди, накопленное время выполнения, интенсивность обращений к вводу-выводу и другие факторы.

Планирование потоков, по существу, включает в себя решение двух задач:

1) определение момента времени для смены текущего активного потока;

2) выбор для выполнения потока из очереди готовых потоков.

Существует множество различных алгоритмов планирования потоков, по-своему решающих каждую из приведенных выше задач. Алгоритмы планирования могут преследовать различные цели и обеспечивать разное качество мультипрограммирования.

В большинстве операционных систем универсального назначения планирование осуществляется *динамически* (on-line), т. е. решения принимаются во время работы системы на основе анализа текущей ситуации. ОС работает в условиях неопределенности – потоки и процессы появляются в случайные моменты времени и также непредсказуемо завершаются. Динамические планировщики могут гибко приспосабливаться к изменяющейся ситуации и не используют никаких предположений о мультипрограммной смеси. Для того чтобы оперативно найти в условиях такой неопределенности оптимальный в некотором смысле порядок выполнения задач, операционная система должна затрачивать значительные усилия.

Другой тип планирования – статический – может быть использован в специализированных системах, в которых весь набор одновременно выполняемых задач определен заранее, например, в системах реального времени. *Планировщик* называется *статическим* (или предварительным), если он принимает решения о планировании не во время работы системы, а заранее (off-line).

Результатом работы статического планировщика является таблица, называемая расписанием, в которой указывается, какому потоку/процессу, когда и на какое время должен быть предоставлен процессор. Для построения расписания планировщику нужны как можно более полные предварительные знания о характеристиках набора задач.

Готовое расписание может использоваться операционной системой для переключения потоков и процессов. При этом накладные расходы ОС на исполнение расписания сводятся лишь к диспетчеризации потоков/процессов.

Диспетчеризация заключается в реализации найденного в результате планирования (динамического или статистического) решения, то есть в переключении процессора с одного потока на другой. Прежде чем прервать выполнение потока, ОС запоминает его контекст, с тем чтобы впоследствии использовать эту информацию для последующего возобновления выполнения данного потока. Контекст отражает, во-первых, состояние аппаратуры компьютера в момент прерывания потока: значение счетчика команд, содержимое регистров общего назначения, режим работы процессора, флаги, маски прерываний и другие параметры. Во-вторых, контекст включает параметры операционной среды, а именно ссылки на открытые файлы, данные о незавершенных операциях ввода-вывода, коды ошибок выполняемых данным потоком системных вызовов и т. д.

Диспетчеризация сводится к следующему [1]:

1) сохранение контекста текущего потока, который требуется сменить;

2) загрузка контекста нового потока, выбранного в результате планирования;

3) запуск нового потока на выполнение.

Поскольку операция переключения контекстов существенно влияет на производительность вычислительной системы, программные модули ОС выполняют диспетчеризацию потоков совместно с аппаратными средствами процессора.

Для реализации алгоритма планирования ОС должна получать управление всякий раз, когда в системе происходит событие, требующее перераспределения процессорного времени. К таким событиям могут быть отнесены следующие [1]:

1) прерывание от таймера, сигнализирующее, что время, отведенное активной задаче на выполнение, закончилось – планировщик переводит задачу в состояние готовности и выполняет перепланирование;

2) активная задача выполнила системный вызов, связанный с запросом на ввод-вывод или на доступ к ресурсу, который в настоящий момент занят, – планировщик переводит задачу в состояние ожидания и выполняет перепланирование;

3) активная задача выполнила системный вызов, связанный с освобождением ресурса, – планировщик проверяет, не ожидает ли этот ресурс какая-либо задача;

4) внешнее (аппаратное) прерывание, которое сигнализирует о завершении периферийным устройством операции ввода-вывода, переводит соответствующую задачу в очередь готовых, и выполняется планирование;

5) внутреннее прерывание сигнализирует об ошибке, которая произошла в результате выполнения активной задачи – планировщик снимает задачу и выполняет перепланирование.

При возникновении каждого из этих событий планировщик выполняет просмотр очередей и решает вопрос о том, какая задача будет выполняться следующей. Помимо указанных, существует и ряд других событий (часто связанных с системными вызовами), требующих перепланировки. Например, запросы приложений и пользователей на создание новой задачи или повышение приоритета уже существующей задачи создают новую ситуацию, которая требует пересмотра очередей и, возможно, переключения процессора.

В системах реального времени для отработки статического расписания планировщик активизируется по прерываниям от таймера. Эти прерывания пронизывают всю временную ось, возникая через короткие постоянные интервалы времени. После каждого прерывания планировщик просматривает расписание и проверяет, не пора ли переключить задачи. Кроме прерываний от таймера в системах реального времени перепланирование задач может происходить по прерываниям от внешних устройств - различного вида датчиков и исполнительных механизмов.

1.1.9 Алгоритмы планирования, основанные на квантовании

В основе многих вытесняющих алгоритмов планирования лежит концепция *квантования* [1]. В соответствии с этой концепцией каждому потоку поочередно для выполнения предоставляется ограниченный непрерывный период процессорного времени – *квант*. Смена активного потока происходит, если:

1) поток завершился и покинул систему;

2) произошла ошибка;

3) поток перешел в состояние ожидания;

4) исчерпан квант процессорного времени, отведенный данному потоку.

Поток, который исчерпал свой квант, переводится в состояние готовности и ожидает, когда ему будет предоставлен новый квант процессорного времени, а на выполнение в соответствии с определенным правилом выбирается новый поток из

очереди готовых. Граф состояний потока, изображенный на рисунке 5, соответствует алгоритму планирования, основанному на квантовании.

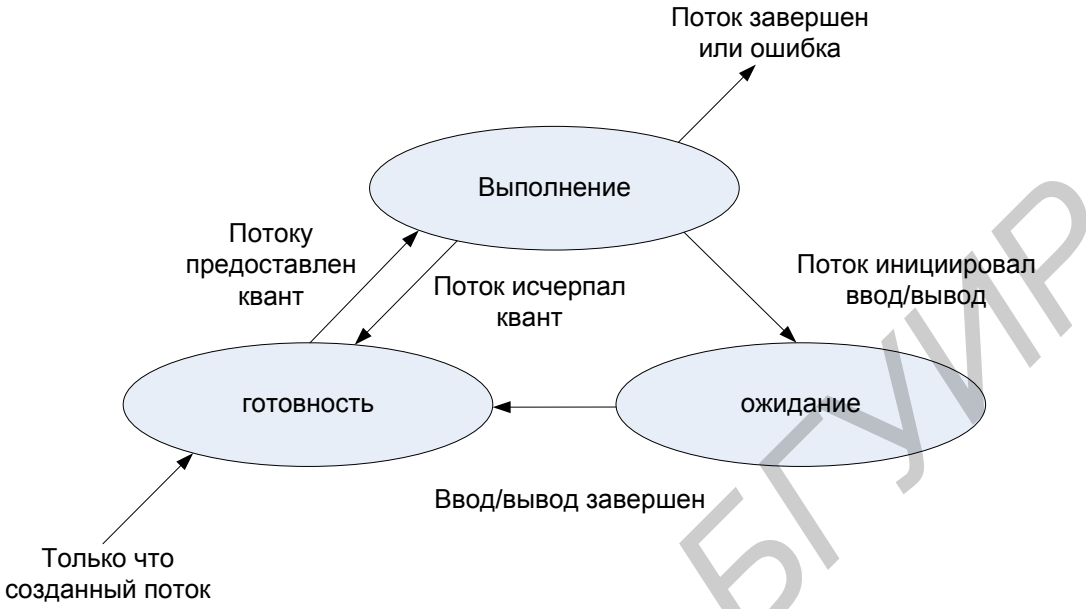


Рисунок 5 – Граф состояний потока в системе с квантованием [1]

Кванты, выделяемые потокам, могут быть одинаковыми для всех потоков или различными. Рассмотрим, например, случай, когда всем потокам предоставляются кванты одинаковой длины q (рисунок 6). Если в системе имеется **n** потоков, то время, которое поток проводит в ожидании следующего кванта, можно грубо оценить как q(n – l). Чем больше потоков в системе, тем больше время ожидания и тем меньше возможности вести одновременную интерактивную работу нескольким пользователям. Но если величина кванта выбрана очень небольшой, то значение произведения q(n – l) все равно будет достаточно мало для того, чтобы пользователь не ощущал дискомфорта от присутствия в системе других пользователей. Типичное значение кванта в системах разделения времени составляет десятки миллисекунд.

Если квант короткий, то суммарное время, которое проводит поток в ожидании процессора, прямо пропорционально времени, требуемому для его выполнения (т. е. времени, которое потребовалось бы для выполнения этого потока при монопольном использовании вычислительной системы). Чем больше квант, тем выше вероятность того, что потоки завершатся в результате первого же цикла выполнения, и тем менее явной становится зависимость времени ожидания потоков от их времени выполнения. При достаточно большом кванте алгоритм квантования вырождается в алгоритм последовательной обработки, присущий однопрограммным системам, при котором время ожидания задачи в очереди вообще никак не зависит от ее длительности.

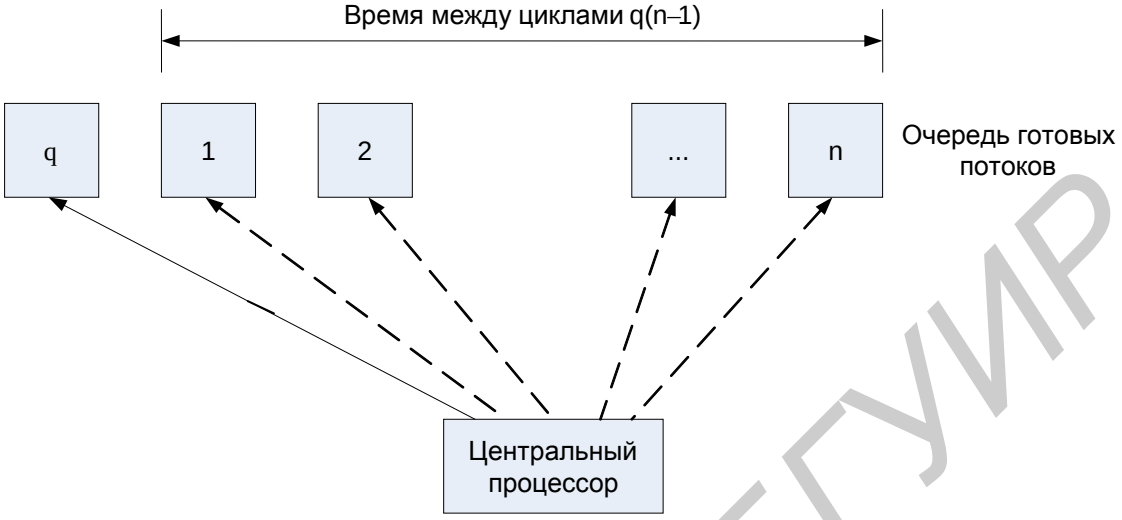


Рисунок 6 – Иллюстрация расчета времени ожидания в очереди

Кванты, выделяемые одному потоку, могут быть фиксированной величины, а могут и изменяться в разные периоды жизни потока. Такой подход позволяет уменьшить накладные расходы на переключение задач в том случае, когда сразу несколько задач выполняют длительные вычисления.

Потоки получают для выполнения квант времени, но некоторые из них используют его не полностью, например, из-за необходимости выполнить ввод или вывод данных. Алгоритм планирования может исправить эту «несправедливость». В качестве компенсации за неиспользованные полностью кванты потоки получают привилегии при последующем обслуживании. Для этого планировщик создает две очереди готовых потоков (рисунок 7).

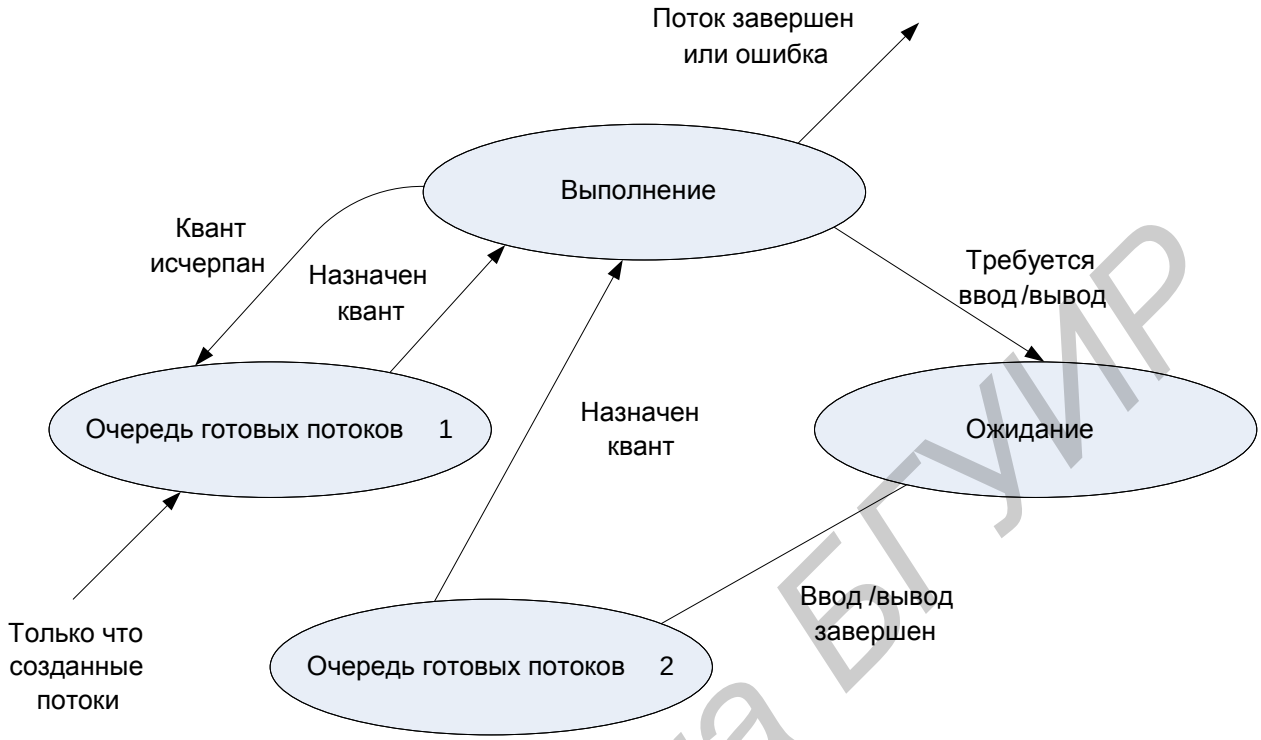


Рисунок 7 – Квантование с предпочтением потоков, интенсивно обращающихся к вводу-выводу [1]

Очередь 1 образована потоками, которые пришли в состояние готовности в результате исчерпания кванта времени, а очередь 2 – потоками, у которых завершилась операция ввода-вывода. При выборе потока для выполнения на процессоре прежде всего просматривается вторая очередь, и только если она пуста, квант выделяется потоку из первой очереди.

Многозадачные ОС теряют некоторое количество процессорного времени для выполнения вспомогательных работ во время переключения контекстов задач. При этом запоминаются и восстанавливаются регистры, флаги и указатели стека, а также проверяется статус задач для передачи управления. Затраты на эти вспомогательные действия не зависят от величины кванта времени, поэтому чем больше квант, тем меньше суммарные накладные расходы, связанные с переключением потоков [1].

1.1.10 Алгоритмы планирования, основанные на приоритетах

Другой важной концепцией, лежащей в основе многих вытесняющих алгоритмов планирования, является *приоритетное обслуживание* [1]. Приоритетное

обслуживание предполагает наличие у потоков приоритета, на основании которого определяется порядок их выполнения. *Приоритет* – это число, характеризующее степень привилегированности потока при использовании ресурсов вычислительной машины, в частности процессорного времени: чем выше приоритет, тем выше привилегии и тем меньше времени будет проводить поток в очередях [1].

Приоритет может выражаться целым или дробным, положительным или отрицательным значением. В некоторых ОС принято, что приоритет потока тем выше, чем больше (в арифметическом смысле) число, обозначающее приоритет. В других системах, наоборот, чем меньше число, тем выше приоритет.

В большинстве операционных систем, поддерживающих потоки, приоритет потока непосредственно связан с приоритетом процесса, в рамках которого выполняется данный поток. Приоритет процесса назначается операционной системой при его создании. Значение приоритета включается в описатель процесса и используется при назначении приоритета потокам этого процесса. При назначении приоритета вновь созданному процессу ОС учитывает, является этот процесс системным или прикладным, каков статус пользователя, запустившего процесс, было ли явное указание пользователя на присвоение процессу определенного уровня приоритета.

Во многих ОС предусматривается возможность изменения приоритетов в течение жизни потока. Изменение приоритета могут происходить по инициативе самого потока или по инициативе пользователя. Кроме того, ОС сама может изменять приоритеты потоков в зависимости от ситуации, складывающейся в системе. В последнем случае приоритеты называются *динамическими* в отличие от неизменяемых, *фиксированных,* приоритетов [1].

От того, какие приоритеты назначены потокам, существенно зависит эффективность работы всей вычислительной системы. В большинстве же случаев ОС присваивает приоритеты потокам по умолчанию.

В качестве примера рассмотрим схему назначения приоритетов потокам, принятую в операционной системе Windows NT (рисунок 8). В системе определено 32 уровня приоритетов и два класса потоков – потоки реального времени и потоки с переменными приоритетами. Диапазон от 1 до 15 включительно отведен для потоков с переменными приоритетами, а от 16 до 31 – для более критичных ко времени потоков реального времени (приоритет 0 зарезервирован для системных целей).

При создании процесса поток в зависимости от класса получает по умолчанию базовый приоритет в верхней или нижней части диапазона. Базовый приоритет процесса в дальнейшем может быть повышен или понижен операционной системой.

В Windows NT с течением времени приоритет потока, относящегося к классу потоков с переменными приоритетами, может отклоняться от базового приори-

тета потока. ОС может повышать приоритет потока (который в этом случае называется динамическим) в тех случаях, когда поток не полностью использовал отведенный ему квант, или понижать приоритет, если квант был использован полностью. ОС наращивает приоритет дифференцированно в зависимости от того, какого типа событие не дало потоку полностью использовать квант. В частности, ОС повышает приоритет в большей степени **q** потоков, которые ожидают ввода с клавиатуры (интерактивным приложениям) и в меньшей степени – потокам, выполняющим дисковые операции. Именно на основе динамических приоритетов осуществляется планирование потоков.

Рисунок 8 – Схема назначения приоритетов в Windows NT [1]

Существуют две разновидности приоритетного планирования: обслуживание с относительными приоритетами и обслуживание с абсолютными приоритетами.

В обоих случаях выбор потока на выполнение из очереди готовых осуществляется одинаково: выбирается поток, имеющий наивысший приоритет. Однако проблема определения момента смены активного потока решается поразному. В системах с относительными приоритетами активный поток выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ожидания (или же произойдет ошибка, или поток завершится). На рисунке 9 показан граф состояний потока в системе с относительными приоритетами.

В системах с абсолютными приоритетами выполнение активного потока прерывается, кроме указанных выше причин, еще при одном условии: если в очереди готовых потоков появился поток, приоритет которого выше приоритета активного потока. В этом случае прерванный поток переходит в состояние готовности (рисунок 10).

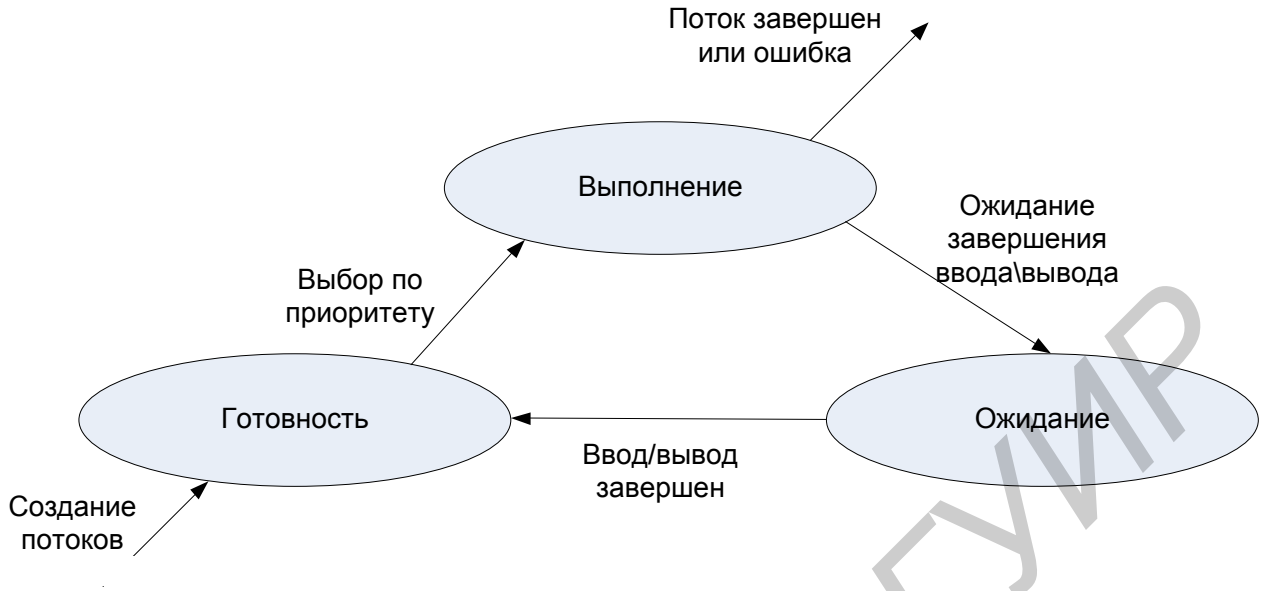


Рисунок 9 – Графы состояний потоков в системах с относительными приоритетами [1]

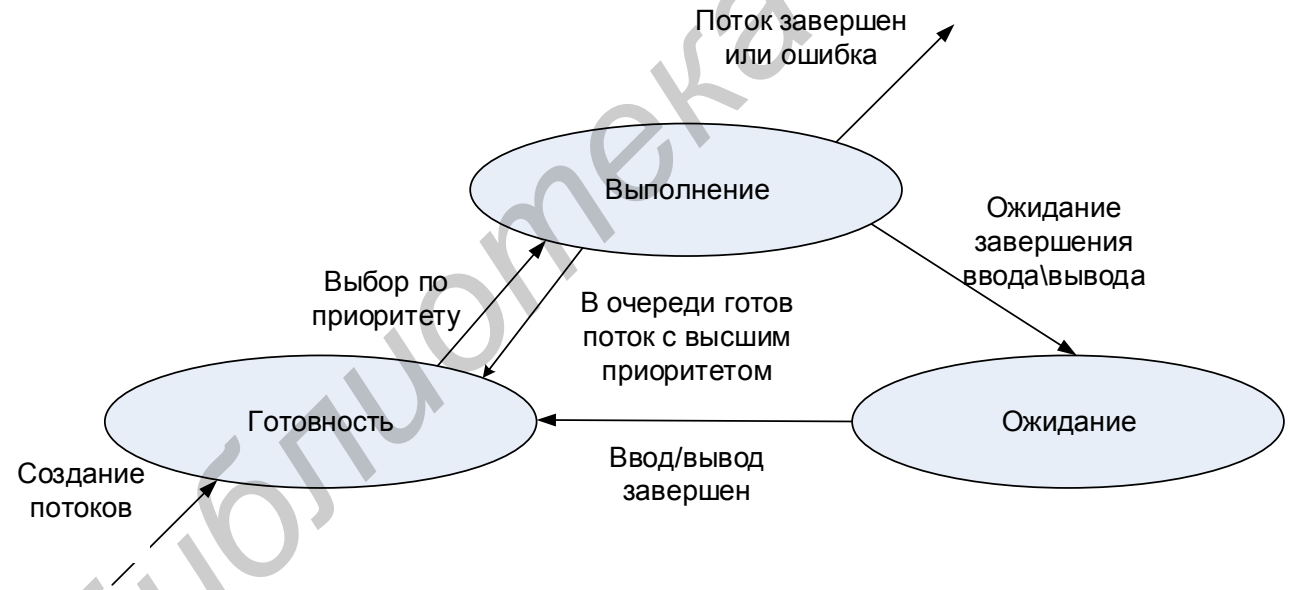


Рисунок 10 – Графы состояний потоков в системах с абсолютными приоритетами [1]

В системах, в которых планирование осуществляется на основе относительных приоритетов, минимизируются затраты на переключения процессора с одной работы на другую. С другой стороны, здесь могут возникать ситуации, когда одна задача занимает процессор долгое время. Ясно, что для систем разделения времени и реального времени такая дисциплина обслуживания не подходит: интер-

активное приложение может ждать своей очереди часами, пока вычислительной задаче не потребуется ввод-вывод.

В системах с абсолютными приоритетами время ожидания потока в очередях может быть сведено к минимуму, если ему назначить самый высокий приоритет. Такой поток будет вытеснять из процессора все остальные потоки (кроме потоков, имеющих такой же наивысший приоритет). Это делает планирование на основе абсолютных приоритетов подходящим для систем управления объектами, в которых важна быстрая реакция на событие.

1.1.11 Смешанные алгоритмы планирования

Во многих операционных системах алгоритмы планирования построены с использованием как концепции квантования, так и приоритетов [1]. Например, в основе планирования лежит квантование, но величина кванта и/или порядок выбора потока из очереди готовых определяется приоритетами потоков. Именно так реализовано планирование в системе Windows NT, в которой квантование сочетается с динамическими абсолютными приоритетами. На выполнение выбирается готовый поток с наивысшим приоритетом. Ему выделяется квант времени. Если во время выполнения в очереди готовых появляется поток с более высоким приоритетом, то он вытесняет выполняемый поток. Вытесненный поток возвращается в очередь готовых, причем он становится впереди всех остальных потоков имеющих такой же приоритет.

1.1.12 Задача 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) [1]. Обычно к средствам IPC относят не только средства межпроцессной синхронизации, но и средства межпроцессного обмена данными.

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.

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 программист может использовать как собственные средства и приемы синхронизации, так и средства операционной системы. 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. 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, задачу ведения базы данных клиентов некоторого предприятия (рисунок 11). 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.

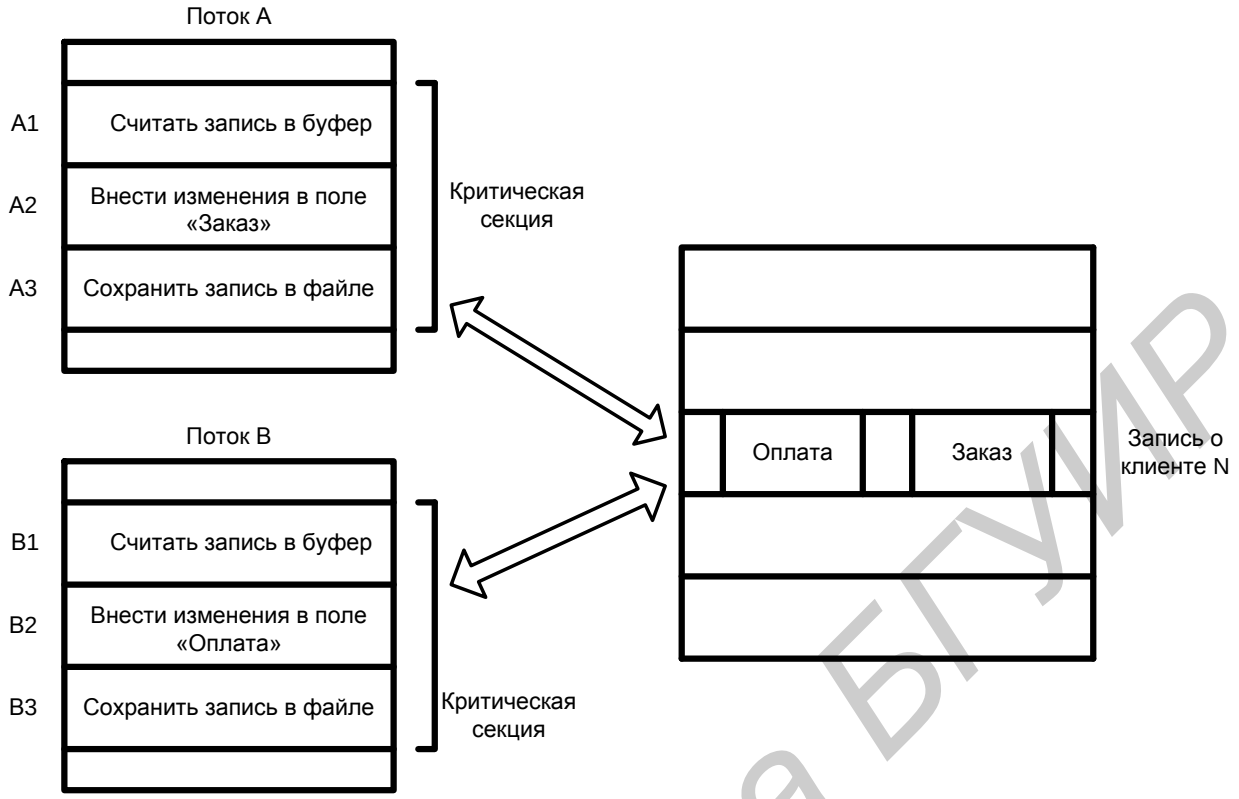


Рисунок 11 – Возникновение гонок при доступе к разделяемым данным

Оба эти потока совместно работают над общим файлом базы данных, используя однотипные алгоритмы, включающие три шага [1]:

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.

Обозначим соответствующие шаги для потока А как Al, А2 и А3, а для потока В – как Bl, В2 и В3. 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 А3) 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.

Когда в очередной раз управление будет передано потоку A, то он, продолжая свою работу, запишет запись о клиенте N с модифицированным полем Заказ в базу данных (шаг A3). После прерывания потока A и активизации потока B 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 (рисунок 12, а).

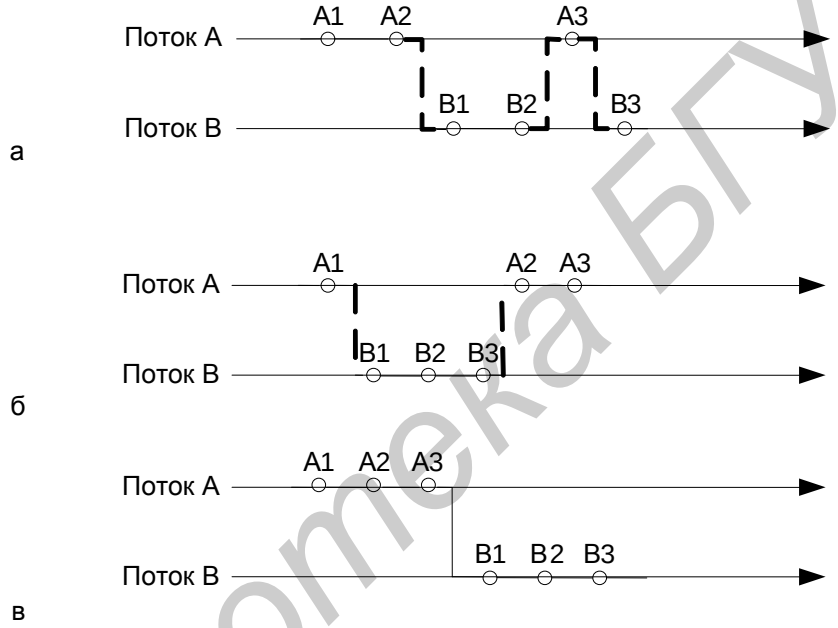


Рисунок 12 – Влияние относительных скоростей потоков на результат решения задачи

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 (рисунок 12, б) или, напротив, все исправления были успешно внесены (рисунок 12, в). Все определяется взаимными скоростями потоков и моментами их прерывания. Поэтому отладка взаимодействующих потоков является сложной задачей. Ситуации, когда два или более потоков обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей потоков, называются *гонками.*

1.1.13 Понятие критической секции

Важным понятием синхронизации потоков являются понятия «*критическая секция*» и «*блокирующие переменные*». *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 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. Этот прием называют *взаимным исключением.* Операционная система использует разные способы реализации взаимного исключения.

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.

Блокирующие переменные – переменные, к которым все потоки процесса имеют прямой доступ. С ними программист работает, не обращаясь к системным вызовам ОС.

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 рисунке 13 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.

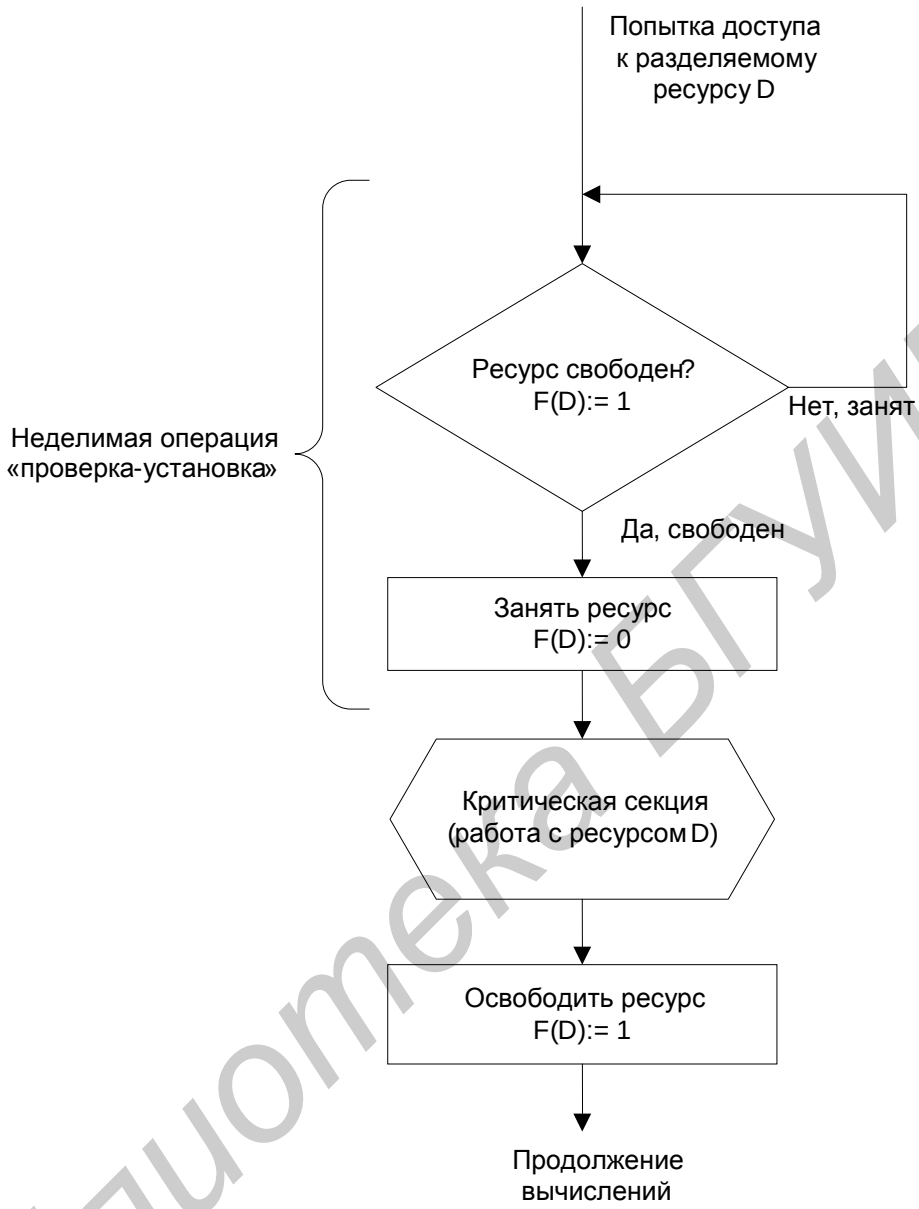


Рисунок 13 – Реализация критических секций с использованием блокирующих переменных [1]

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

Потоки могут быть прерваны операционной системой в любой момент и в любом месте, в том числе в критической секции.

Однако нельзя прерывать поток между выполнением операций проверки и установки блокирующей переменной. Этот способ ограничения прерываний 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.

Итак, на рисунке 14 показано, как с помощью функций для работы с критической секцией реализовано взаимное исключение в операционной системе Windows NT.

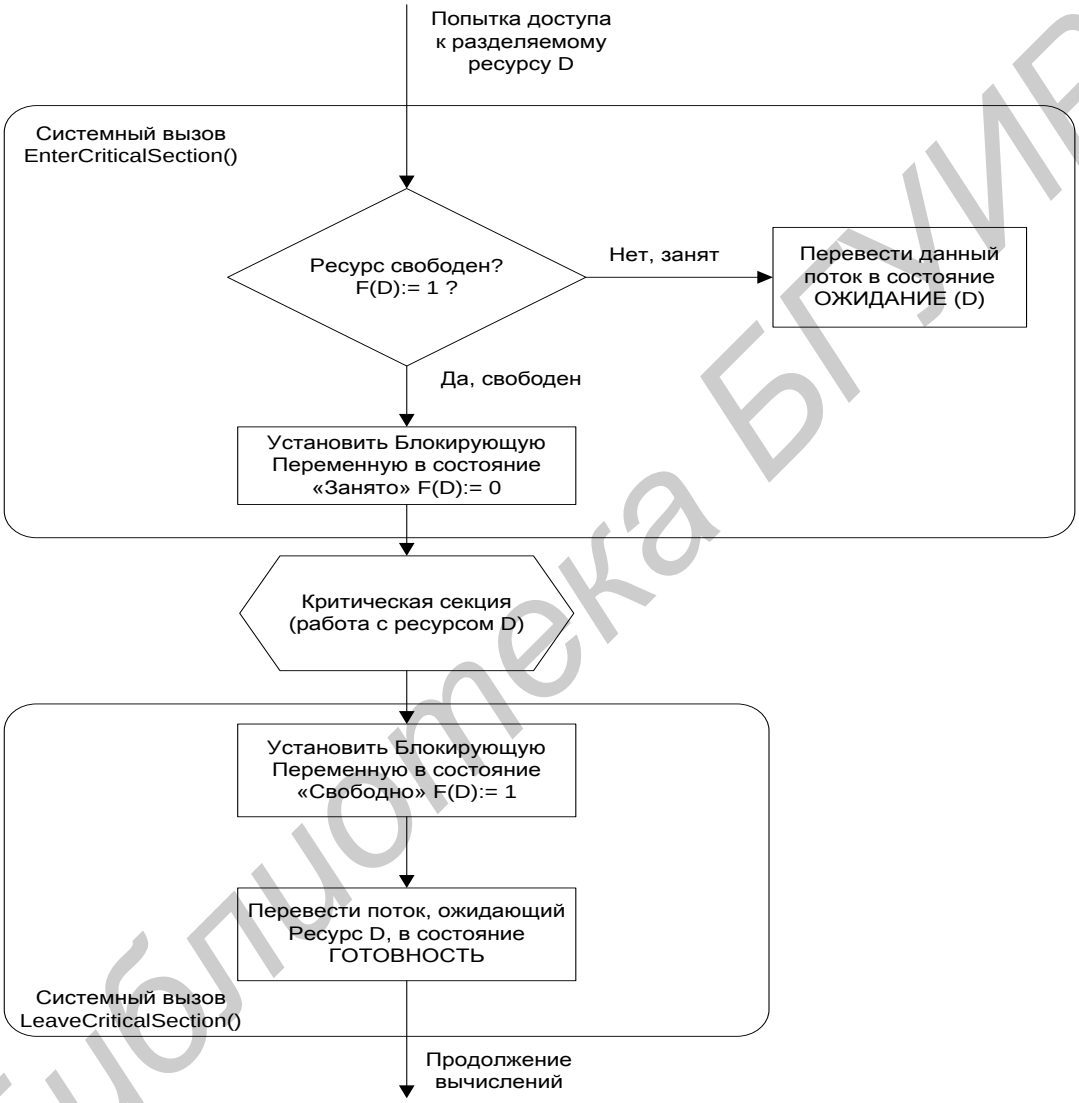


Рисунок 14 – Реализация взаимного исключения с использованием системных функций входа в критическую секцию и выхода из нее [1]

Перед тем как начать изменение критических данных, поток выполняет системный вызов EnterCriticalSection(). В рамках этого вызова сначала выполняется проверка блокирующей переменной, отражающей состояние критического ресурса. Если системный вызов определил, что ресурс занят (F(D) = 0), он 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.

Таким образом, исключается непроизводительная потеря процессорного времени на циклическую проверку освобождения занятого ресурса.

1.1.14 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 др.

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 NT.

Поток может ожидать установки сигнального состояния не одного объекта, а нескольких. 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.

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.

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), мьютекс (mutex), системный семафор* и др.

Семафоры – переменные, которые могут принимать целые неотрицательные значения.

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

1) 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;

2) 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 P, ждет, пока это уменьшение станет возможным. Успешная проверка и уменьшение также являются неделимой операцией.

Никакие прерывания во время выполнения примитивов V и P недопусмы.

В частном случае, когда семафор S может принимать только значения 0 и 1, он превращается в блокирующую переменную, которую называют двоичным семафором.

Рассмотрим использование семафоров на 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. 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: e – число пустых буферов и f – число заполненных буферов, причем в исходном состоянии e = N, а f = 0. Тогда работа потоков с общим буферным пулом может быть описана следующим образом (рисунок 15).

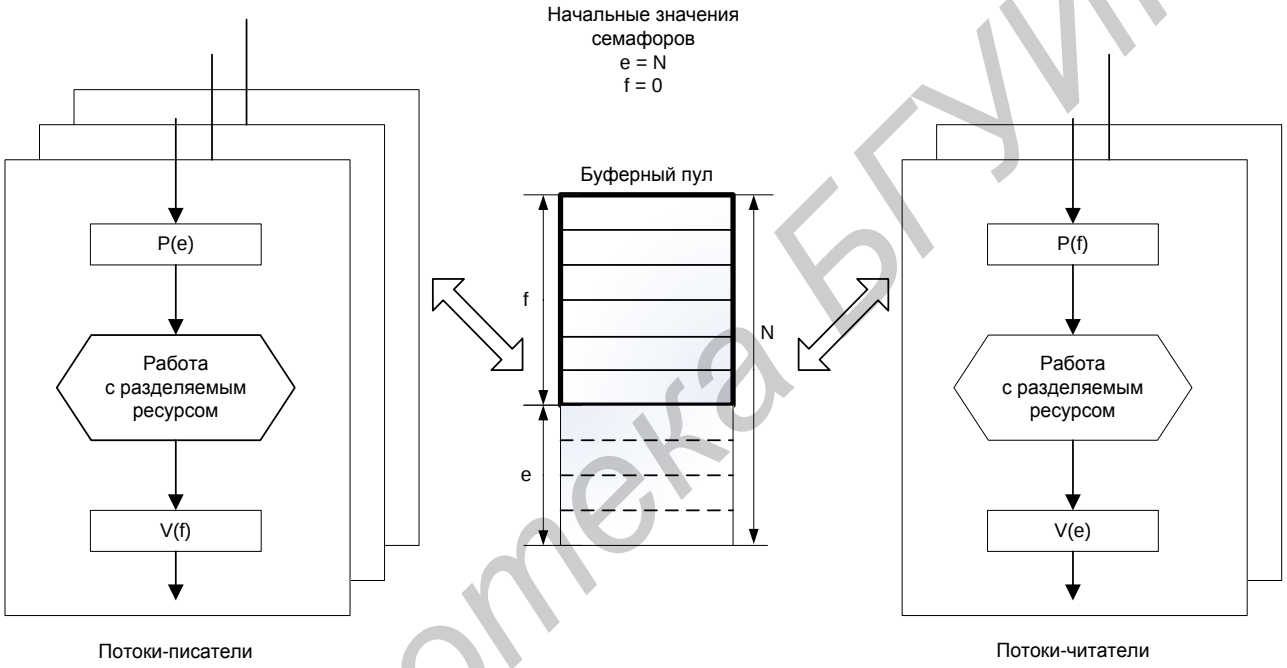


Рисунок 15 – Использование семафоров для синхронизации потоков [1]

Поток-писатель прежде всего выполняет операцию P(e), с помощью которой он проверяет, имеются ли в буферном пуле незаполненные буферы. В соответствии с семантикой операции P, если семафор e равен 0 (т. е. свободных буферов в данный момент нет), то поток-писатель переходит в состояние ожидания. Если же значением e 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 бу-

ферный пул, который может быть представлен как набор идентичных ресурсов – отдельных буферов. Семафор допускает к разделяемому пулу ресурсов заданное количество потоков. Так, в 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.

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 (рисунок 16).

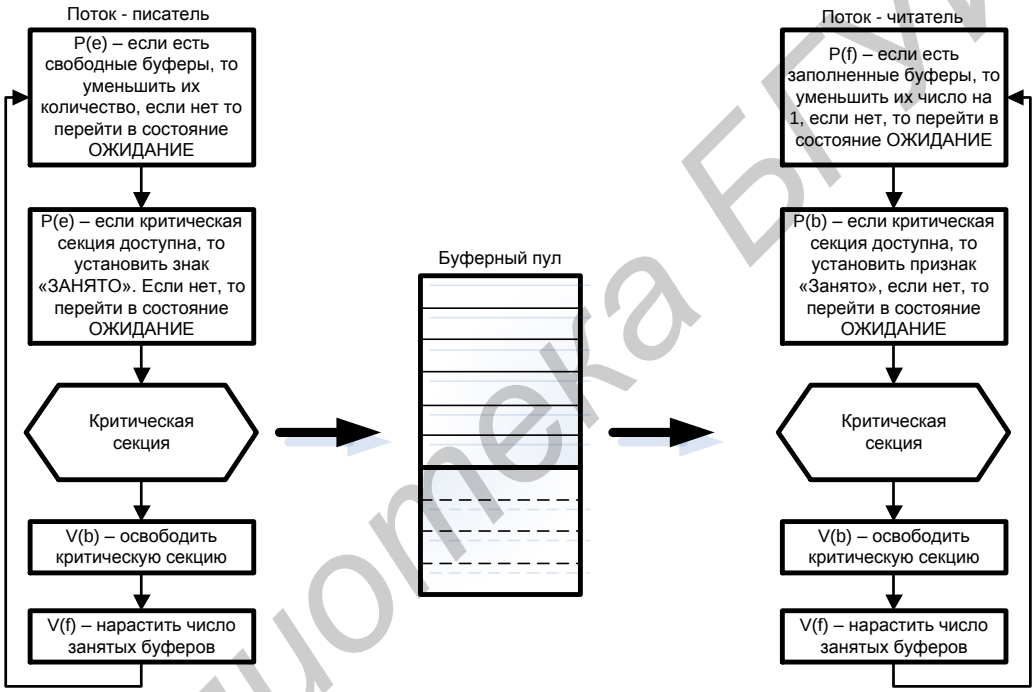


Рисунок 16 – Использование двоичного семафора [1]

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

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, пытаясь получить доступ к критическим данным, выполнил системный вызов Wait(X), где X – указатель на мьютекс. Если 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.

1.1.15 Потоки в Windows. Функция потока. Создание потока

Функция потока [2]

В первичном потоке функцией потока является *main, wmain, WinMain* или *wWinMain.* Если создать вторичный поток, в нем тоже будет входная функция примерно следующего вида:

```
DWORD WINAPI ThreadFunc(PVOID pvParam)
{
DWORD rtwResult = 0;
....
return(dwResult);
}
```

Функция потока может выполнять любые задачи. Рано или поздно она закончит свою работу и вернет управление. В этот момент поток остановится и память, отведенная под его стек, будет освобождена, а счетчик пользователей его объекта ядра «поток» уменьшится на 1. Когда счетчик обнулится, этот объект ядра будет разрушен. Но, как и объект ядра «процесс», он может жить гораздо дольше, чем сопоставленный с ним поток.

Создание потока [2]

При вызове функции **CreateProcess** появляется первичный поток процесса. Для создания дополнительных потоков нужно вызывать из первичного потока функцию CreateThread:

```
HANDLE CreateThread(
LPSECURITY_ATTRIBUTES psa,
DWORD cbStack,
LPTHREAD_START_ROUTINE pfnStartAddr,
LPVOID pvParam,
DWORD tdwCreate,
LPDWORD pdwThreadID);
```

При каждом вызове этой функции система создает объект ядра «поток». Это не сам поток, а компактная структура данных, которая используется операционной системой для управления потоком и хранит статистическую информацию о потоке.

Система выделяет память под стек потока из адресного пространства процесса. Новый поток выполняется в контексте того же процесса, что и родительский поток. Поэтому он получает доступ ко всем описателям объектов ядра, всей памяти и стекам всех потоков в процессе. За счет этого потоки в рамках одного процесса могут легко взаимодействовать друг с другом.

Параметр psa является указателем на структуру SECURITY_ATTRIBUTES. Если необходимо, чтобы объекту ядра «поток» были присвоены атрибуты защиты по умолчанию, в этом параметре передается NULL. А чтобы дочерние процессы смогли наследовать описатель этого объекта, необходимо определить структуру SECURITY_ATTRIBUTES и инициализировать ее элемент *hInheritHandle* значением TRUE.

Параметр cbStack. Этот параметр определяет, какую часть адресного пространства поток сможет использовать под свой стек. Каждому потоку выделяется отдельный стек Функция CreateProcess, запуская приложение, вызывает CreateThread, и эта функция инициализирует первичный поток процесса. При этом CreateProcess заносит в параметр *cbStack* значение, хранящееся в самом исполняемом файле.

Параметры pfnStartAddr и pvParam. Параметр *pfnStartAddr* определяет адрес функции потока, с которой должен будет начать работу создаваемый поток, а параметр *pvParam* идентичен параметру *pvParam* функции потока. CreateThread лишь передает этот параметр по эстафете той функции, с которой начинается выполнение создаваемого потока. Таким образом, данный параметр позволяет передавать функции потока какое-либо инициализирующее значение. Оно может быть или просто числовым значением, или указателем на структуру данных с дополнительной информацией.

Параметр fdwCreate. Этот параметр определяет дополнительные флаги, управляющие созданием потока. Он принимает одно из двух значений. 0 (исполнение потока начинается немедленно) или CREATE_SUSPENDED. В последнем случае система создает поток, инициализирует его и приостанавливает до последующих указаний. Флаг CREATE_SUSPENDED позволяет программе изменить какие-либо свойства потока перед тем, как он начнет выполнять код.

Параметр pdwThreadID. Последний параметр функции – это адрес переменной типа DWORD, в которой функция возвращает идентификатор, приписанный системой новому потоку.

Завершение потока [2]

Поток можно завершить четырьмя способами:

1) функция потока возвращает управление (рекомендуемый способ);

2) поток самоуничтожается вызовом функции ExitThread (нежелательный способ);

3) один из потоков данного или стороннего процесса вызывает функцию TerminateThread (нежелательный способ);

4) завершается процесс, содержащий данный поток (нежелательно).

Возврат управления функцией потока. Функцию рекомендуется проектировать так, чтобы поток завершался только после того, как она возвращает управление. Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших потоку.

Функция *ExitThread.* Поток можно завершить принудительно, вызвав:

```
VOID ExitThread(DWORD dwExitCode);
```

При этом освобождаются все ресурсы операционной системы, выделенные данному потоку, но C/C++ – ресурсы (например, объекты, созданные из C++-классов) не очищаются. Именно поэтому лучше возвращать управление из функции потока, чем самому вызывать функцию ExitThread.

В параметр *dwExitCode* помещается значение, которое система рассматривает как код завершения потока. Возвращаемого значения у этой функции нет, потому что после ее вызова поток перестает существовать.

Функция *TerminateThread.* Вызов этой функции также завершает поток:

```
BOOL TerminateThread( HANDLE hThread, DWORD dwExitCode);
```

В отличие от ExitThread, которая уничтожает только вызывающий поток, эта функция завершает поток, указанный в параметре *hThread.* В параметр *dwExitCode* помещается значение, которое система рассматривает как код завершения потока. После того как поток будет уничтожен, счетчик пользователей уменьшится на 1.

Приостановка и возобновление потоков [2]

В объекте ядра «поток» имеется переменная – счетчик числа простоев данного потока. При вызове **CreateProcess** или **CreateThread** он инициализируется значением, равным 1, которое запрещает системе выделять новому потоку процессорное время. Сразу после создания поток не готов к выполнению, ему нужно время для инициализации.

После того как поток полностью инициализирован, CreateProcess или CreateThread проверяет, не передан ли ей флаг CREATE_SUSPENDED, и, если да, возвращает управление, оставив поток в приостановленном состоянии. В ином случае счетчик простоев обнуляется.

Создав поток в приостановленном состоянии, можно настроить некоторые его свойства. Закончив настройку, необходимо разрешить выполнение потока. Для этого используется функция ResumeThread:

```
DWORD ResumeThread(HANDLE hThread);
```

Если вызов ResumeThread прошел успешно, эта функция возвращает предыдущее значение счетчика простоев данного потока; в ином случае – 0xFFFFFFFF.

Выполнение отдельного потока можно приостанавливать несколько раз. Если поток приостановлен три раза, то и возобновлен он должен быть тоже три раза. Выполнение потока можно приостановить не только при его создании с флагом CREATE_SUSPENDED, но и вызовом SuspendThread:

```
DWORD SuspendThread(HANDLE hThread);
```

Любой поток может вызвать эту функцию и приостановить выполнение другого потока.

Функция Sleep [2]

Поток может приказать системе не выделять ему процессорное время на определенный период, вызвав:

```
VOID Sleep(DWORD dwMilliseconds);
```

Эта функция приостанавливает поток на *dwMilliseconds*.

Особенности:

1) вызывая Sleep, поток добровольно отказывается от остатка выделенного ему кванта времени;

2) система прекращает выделять потоку процессорное время на период, примерно равный заданному;

3) можно вызвать Sleep и передать в *dwMilliseconds* значение INFINITE, вообще запретив планировать поток;

4) можно вызвать Sleep и передать в *dwMilliseconds* нулевое значение – т. е. отказаться от остатка своего кванта времени и заставить систему подключить к процессору другой поток.

При загрузке системы создается особый поток – поток обнуления страниц (zero page thread), которому присваивается нулевой уровень приоритета. Ни один поток, кроме этого, не может иметь нулевой уровень приоритета. Он обнуляет свободные страницы в оперативной памяти при отсутствии других потоков, требующих внимания со стороны системы.

1.1.16 Синхронизация потоков в WINDOWS

Синхронизация с помощью критических секций [2–4]

В Windows для реализации критических секций используется структура CRITICAL_SECTION, работа с которой осуществляется исключительно через функции Windows, передавая им адрес соответствующего экземпляра этой структуры. Структура инициализируется с помощью следующей функции:

```
VOID InitializeCriticalSection(
                              LPCRITICAL_SECTION pcs);
```

Эта функция инициализирует элементы структуры CRITICAL_SECTION, на которую указывает параметр *pcs*. Поскольку вся работа данной функции заключается в инициализации нескольких переменных-членов, она не дает сбоев и поэтому ничего не возвращает (void). InitializeCriticalSection должна быть вызвана до того, как один из потоков обратится к EnterCriticalSection. Участок кода, работающий с разделяемым ресурсом, начинается вызовом:

```
VOID EnterCriticalSection(LPCRITICAL_SECTION pcs);
```

EnterCriticalSection – исследует значения элементов структуры CRITICAL_SECTION. Если ресурс занят, в них содержатся сведения о том, какой поток пользуется ресурсом. EnterCriticalSection выполняет следующие действия.

Если ресурс свободен, EnterCriticalSection модифицирует элементы структуры, указывая, что вызывающий поток занимает ресурс, после чего немедленно возвращает управление, и поток продолжает свою работу.

Если ресурс уже захвачен вызывающим потоком, EnterCriticalSection обновляет их, отмечая тем самым, сколько раз подряд этот поток захватил ресурс, и немедленно возвращает управление. Такая ситуация бывает нечасто – лишь тогда, когда поток два раза подряд вызывает EnterCriticalSection без промежуточного вызова LeaveCriticalSection.

Если значения элементов структуры указывают на то, что ресурс занят другим потоком, EnterCriticalSection переводит вызывающий поток в режим ожидания. Система запоминает, что данный поток хочет получить доступ к ресурсу, и – как только поток, занимавший этот ресурс, вызывает LeaveCriticalSection – вновь начинает выделять данному потоку процессорное время.

В конце участка кода, использующего разделяемый ресурс, должен присутствовать вызов:

```
VOID LeaveCriticalSection(LPCRITICAL_SECTION pcs);
```

Эта функция просматривает элементы структуры CRITICAL_SECTION и уменьшает счетчик числа захватов ресурса вызывающим потоком на 1. Если его значение больше 0, LeaveCriticalSection ничего не делает и просто возвращает управление.

Если значение счетчика достигло 0, LeaveCriticalSection сначала выясняет, есть ли в системе другие потоки, ждущие данный ресурс в вызове EnterCriticalSection. Если есть хотя бы один такой поток, функция настраивает значения элементов структуры, чтобы они сигнализировали о занятости ресурса, и отдает его одному из ждущих потоков (поток выбирается «по справедливости»). Если же ресурс никому не нужен, LeaveCriticalSection сбрасывает элементы структуры.

Пример реализации критической секции:

```
HANDLE ht1;
CRITICAL_SECTION cs;
int x;
PVOID pvParam =(PVOID) &x;
```

```
//--Инициализация критической секции-------------------------
InitializeCriticalSection((LPCRITICAL_SECTION)& cs);

//--Функция потока ThreadFunc1-------------------------------
DWORD WINAPI ThreadFunc1(PVOID pvParam){
while (true){
EnterCriticalSection((LPCRITICAL_SECTION)& cs);
…
// --Код, который работает с общим ресурсом
…
//--Выход из критической секции
LeaveCriticalSection((LPCRITICAL_SECTION)& cs);
Sleep(100);
}
}
…
//---удаление критической секции-----------------------------
DeleteCriticalSection(&cs);
```

Синхронизация потоков с помощью объектов ядра. Функции ожидания объектов ядра [5,6]

В Windows есть специальные функции ожидания синхронизирующих объектов ядра (wait-функции), которые позволяют потоку в любой момент приостановиться и ждать освобождения какого-либо объекта ядра. Из всего семейства этих функций чаще всего используется WaitForSingleObject:

```
DWORD WaitForSingleObject( HANDLE hHandle, DWORD dwMilliseconds);
```

Когда поток вызывает эту функцию, параметр *hHandle* идентифицирует объект ядра, поддерживающий состояние «свободен-занят». Второй параметр, *dwMilliseconds*, указывает сколько времени (в миллисекундах) поток готов ждать освобождения объекта.

Следующий вызов сообщает системе, что поток будет ждать до тех пор, пока не завершится процесс, идентифицируемый описателем *hProcess*.

```
WaitForSingleObject(hProcess, INFINITE);
```

В данном случае константа INFINITE подсказывает системе, что вызывающий поток готов ждать этого события хоть целую вечность. Именно эта константа

обычно и передается функции WaitForSingleObject, но можно указать любое значение в миллисекундах.

Функция WaitForMultipleObjects позволяет ждать освобождения сразу нескольких объектов или какого-то одного из списка объектов:

```
DWORD WaitForMultipleObjects( DWORD dwCount,
CONST HANDLE* phHandles, BOOL bWaitAll, DWORD dwMilli-
seconds);
```

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

WaitForMultipleObjects приостанавливает поток и заставляет его ждать освобождения либо всех заданных объектов ядра, либо одного из них. Параметр *bWaitAll* определяет, что именно требуется от функции. Если он равен TRUE, функция не даст потоку возобновить свою работу, пока не освободятся все объекты.

Параметр *dwMilliseconds* идентичен одноименному параметру функции WaitForSingleObject.

Возвращаемое значение функции WaitForMultipleObjects сообщает, почему возобновилось выполнение вызвавшего ее потока.

Синхронизация потоков с помощью мьютексов [4]

Для использования объекта-мьютекса один из процессов должен сначала создать его вызовом CreateMutex:

```
HANDLE CreateMutex( LPSECURITY_ATTRIBUTES psa, BOOL
bInitialOwner, LPCTSTR pszName);
```

Любой процесс может получить свой («процессозависимый») описатель существующего объекта «мьютекс», вызвав OpenMutex:

```
HANDLE OpenMutex( DWORD fdwAccess, BOOL bInheritHandle,
PCTSTR pszName);
```

Параметр *fInitialOwner* определяет начальное состояние мьютекса. Если в нем передается FALSE (что обычно и бывает), объект-мьютекс не принадлежит ни одному из потоков и поэтому находится в свободном состоянии. При этом его идентификатор потока и счетчик рекурсии равны 0. Если же в нем передается TRUE, идентификатор потока, принадлежащий мьютексу, приравнивается иден-

тификатору вызывающего потока, а счетчик рекурсии получает значение 1. Поскольку теперь идентификатор потока отличен от 0, мьютекс изначально находится в занятом состоянии.

Когда поток, занимающий ресурс, заканчивает с ним работать, он должен освободить мьютекс вызовом функции ReleaseMutex:

```
BOOL ReleaseMutex(HANDLE hMutex);
```

Эта функция уменьшает счетчик рекурсии в объекте-мьютексе на 1. Если данный объект передавался во владение потоку неоднократно, поток обязан вызвать ReleaseMutex столько раз, сколько необходимо для обнуления счетчика рекурсии. Как только счетчик станет равен 0, переменная, хранящая идентификатор потока, тоже обнулится, и объект-мьютекс освободится. После этого система проверит, ожидают ли освобождения мьютекса какие-нибудь другие потоки. Если да, система «по-честному» выберет один из ждущих потоков и передаст ему во владение объект-мьютекс.

Остальные объекты ядра отличаются от объекта-мьютекса тем, что способны запоминать, какому потоку они принадлежат. Если какой-то посторонний поток попытается освободить мьютекс вызовом функции ReleaseMutex, то она, проверив идентификаторы потоков и обнаружив их несовпадение, ничего делать не станет, а просто вернет FALSE.

Пример использования мьютекса приведен ниже.

```
HANDLE ht1,hMutex;
int x;
BOOL TermFlag1=FALSE;

//--Создание мьютекса с помощью функции CreateMutex ----
hMutex=CreateMutex(
        NULL,
        FALSE,
        NULL);

//--Функция потока ThreadFunc1----------------------------------
DWORD WINAPI ThreadFunc1() {
while (!TermFlag1) {
WaitForSingleObject(
        hMutex,
        INFINITE);
...
// --Код, который работает с общим ресурсом
...
```

```
ReleaseMutex(hMutex);
Sleep(1500);
}
return(0);
}
}

//--- Создание потока ----------------------------------------
DWORD dwThread;
TermFlag1=FALSE;
    ht1=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)
ThreadFunc1, (PVOID)&x, 0, &dwThread);
…

//---Закрытие мьютекса-------------------------------------------
CloseHandle(hMutex);
```

Синхронизация потоков с помощью семафоров [4]

Объекты ядра «семафор» используются для учета ресурсов. Как и все объекты ядра, они содержат счетчик числа пользователей, но, кроме того, поддерживают два 32-битных значения со знаком: одно определяет максимальное число ресурсов (контролируемое семафором), другое используется как счетчик текущего числа ресурсов.

Для семафоров определены следующие правила:

1) когда счетчик текущего числа ресурсов становится больше 0, семафор переходит в свободное состояние;

2) если этот счетчик равен 0, семафор занят;

3) система не допускает присвоения отрицательных значений счетчику текущего числа ресурсов;

4) счетчик текущего числа ресурсов не может быть больше максимального числа ресурсов.

Объект ядра «семафор» создается вызовом CreateSemaphore:

```
HANDLE CreateSemaphore( LPSECURITY_ATTRIBUTE psa,
LONG lInitialCount, LONG lMAXimumCount, LPCTRTR
pszName);
```

Любой процесс может получить свой («процессозависимый») описатель существующего объекта «семафор», вызвав OpenSemaphore.

```
HANDLE OpenSemaphore( DWORD fdwAccess,
BOOL bInhentHandle, LPCTSTR pszName);
```

Параметр *lMAXimumCount* сообщает системе максимальное число ресурсов, обрабатываемое приложением. Поскольку это 32-битное значение со знаком, предельное число ресурсов может достигать 2 147 483 647. Параметр *lInitia1Count* указывает, сколько из этих ресурсов доступны изначально (на данный момент).

Поток получает доступ к ресурсу, вызывая одну из Wait-функций и передавая ей описатель семафора, который охраняет этот ресурс. Wait-функция проверяет у семафора счетчик текущего числа ресурсов, если его значение больше 0 (семафор свободен), уменьшает значение этого счетчика на 1, и вызывающий поток остается планируемым. Только после того как счетчик ресурсов будет уменьшен на 1, доступ к ресурсу сможет запросить другой поток.

Если Wait-функция определяет, что счетчик текущего числа ресурсов равен 0 (семафор занят), система переводит вызывающий поток в состояние ожидания. Когда другой поток увеличит значение этого счетчика, система вспомнит о ждущем потоке и снова начнет выделять ему процессорное время.

Поток увеличивает значение счетчика текущего числа ресурсов, вызывая функцию ReleaseSemaphore:

```
BOOL ReleaseSemaphore( HANDLE hSemaphore, LONG lRe-
leaseCount, LPLONG lpPreviousCount);
```

ReleaseSemaphore просто складывает величину *lReleaseCount* со значением счетчика текущего числа ресурсов. Обычно в параметре *lReleaseCount* передают 1, но это вовсе не обязательно. Функция возвращает исходное значение счетчика ресурсов в **lpPreviousCount*.

Пример реализации семафоров:

```
HANDLE ht1,hSemaphore;
BOOL TermFlag1=FALSE;

//--Создание семафора с помощью функции CreateSemaphore
hSemaphore=CreateSemaphore(
                NULL,
                1,
                1,
                NULL);
…

//--Функция потока ThreadFunc1---------------------------------
DWORD WINAPI ThreadFunc1() {
```

```
while (!TermFlag1) {
WaitForSingleObject(
            hSemaphore,
            INFINITE);
...
// --Код, который работает с общим ресурсом
...
ReleaseSemaphore(hSemaphore,1,NULL);
Sleep(1500);
}
return(0);
}
    …

//--- Создание потока ----------------------------------------
    DWORD dwThread;
int x;
    TermFlag1=FALSE;
    ht1=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)
ThreadFunc1, (PVOID)&x, 0, &dwThread);

//---Закрытие семафора-----------------------------------
CloseHandle(hSemaphore);
```

1.2 Порядок выполнения работы

1 Изучить теоретические основы реализации механизма многопоточности в операционных системах и набор системных функций для работы с потоками в Windows.

2 Получить задание у преподавателя.

3 В среде Builder C++ или MS Visual Studio создать приложение и продемонстрировать правильность его работы преподавателю.

4 Оформить отчет по лабораторной работе.

1.3 Содержание отчета

1 Название работы и цель работы.

2 Исходное задание.

3 Листинг разработанной программы.

4 Выводы.

1.4 Контрольные вопросы

1 Как приостановить/возобновить выполнение потока системными средствами Windows?

2 Какие системные ресурсы сопоставляются с объектом «поток»?

3 В чем отличие объекта «мьютекс» от объекта «семафор» применительно к задаче синхронизации потоков?

4 Можно ли, используя критическую секцию, синхронизировать потоки для разных процессов ?

5 Как принудительно завершить поток системными средствами Windows?

6 Какие существуют способы синхронизации потоков в современных ОС?

7 В чем функциональное отличие объектов «поток» и «процесс»?

8 В каких трех основных состояниях может находиться поток?

9 Можно ли влиять на время, которое отдельно взятый поток находится в в очереди за ресурсами?

Литература

1 Олифер, В. Г. Сетевые операционные системы / В. Г. Олифер, Н. А. Олифер. – СПб. : Питер, 2002.

2 Побегайло, А. Системное программирование в Windows / А. Побегайло. – СПб. : БХВ-Петербург, 2006.

3 Харт, Д. Системное программирование в среде Windows / Д. Харт. – СПб. : Издат. дом «Вильямс», 2005.

4 Рихтер, Дж. Windows для профессионалов. Создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер ; пер. с англ. – 4-е изд. – СПб. : Питер ; М. : Издат.-торг. дом «Русская Редакция», 2004.

5 Гальченко, В. Г. Системное программирование в среде WIN32. Создание Windows-приложений / В. Г. Гальченко. – Томск : ТПУ, 2009.

6 Щупак, Ю. Win32 API. Эффективная разработка приложений / Ю. Щупак. – СПб. : Питер, 2007.

Св. план 2012, поз. 11

Учебное издание

Лихачёв Денис Сергеевич

РАЗРАБОТКА МНОГОПОТОЧНЫХ ПРИЛОЖЕНИЙ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

ПОСОБИЕ

Редакторы *Т. П. Андрейченко*, *Е. И. Герман*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *В. М. Задоля*

Подписано в печать 08.01.2014. Формат 60x84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 3,02. Уч.-изд. л. 3,0. Тираж 100 экз. Заказ 349.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0494371 от 16.03.2009. ЛП №02330/0494175 от 03.04.2009.
220013, Минск, П. Бровки, 6