МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)

*С.В. ВОСТОКИН*

# УПРАВЛЕНИЕ ПРОЦЕССАМИ И ПАМЯТЬЮ В ОПЕРАЦИОННЫХ СИСТЕМАХ

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева» в качестве учебного пособия для обучающихся по основной образовательной программе высшего образования по направлению подготовки 02.03.02 Фундаментальная информатика и информационные технологии

С А М А Р А

Издательство Самарского университета

2023

УДК 004.451(075)
ББК 3973я7
В780

Рецензенты: д-р тех. наук, проф. С.П. О р л о в,
д-р физ.-мат. наук, проф. Д.Л. Г о л о в а ш к и н

***Востокин, Сергей Владимирович***

**В780 Управление процессами и памятью в операционных системах:** учебное пособие / *С.В. Востокин*. – Самара: Издательство Самарского университета, 2023. – 120 с.

**ISBN 978-5-7883-1877-6**

Рассмотрены основные механизмы операционных систем: процессы и методы планирования, методы синхронизации, тупики и защита от них, методы управления памятью, алгоритмы замещения страниц, организация внешней памяти и ввода-вывода. Приведен список экзаменационных вопросов, варианты задач для самоконтроля, проведения лабораторных занятий и экзамена.

Учебное пособие предназначено для изучающих дисциплину «Операционные системы» по образовательным программам бакалавриата в области информационных технологий.

УДК 004.451(075)
ББК 3973я7

**ISBN 978-5-7883-1877-6**

## ОГЛАВЛЕНИЕ

## ВВЕДЕНИЕ

Предлагаемое вашему вниманию учебное пособие «Управление процессами и памятью в операционных системах» – это сборник материалов для проведения лекционных, лабораторных и практических занятий по курсу «Операционные системы», который предназначен студентам, обучающимся на бакалавриате ИТ-направлений института Информатики и кибернетики Самарского национального исследовательского университета имени академика С.П. Королева.

Базовый теоретический материал, включающий обзор предмета курса, вопросы проектирования и классификации операционных систем, а также теорию языка программирования Си, в частности, указатели и адресную арифметику, работу с динамической памятью с использованием стандартной библиотеки Си рассмотрен в пособии «Архитектура операционных систем». В данном учебном пособии представлен основной материал теоретической части курса «Операционные системы», в котором изучаются системы управления процессами, памятью и вводом-выводом. Практическая часть учебного пособия посвящена освоению работы с кучами процессов, реализации явной и неявной динамической компоновки, управлению процессами и потоками на примере ОС Windows.

Рекомендуется следующая методика освоения предлагаемого материала. Теоретическая и практическая части курса осваиваются на параллельно идущих (чередующихся) занятиях. Три раздела «Управление процессами», «Синхронизация» и «Управление памятью» содержат материалы для проведения лекционных и практических занятий по десяти темам, основными из которых являются: «Понятие процесса, методы планирования»; «Синхронизация в разделяемой памяти»; «Процедурные методы синхронизации»; «Тупики и защита от них»; «Методы управления памятью»; «Обзор алгоритмов замещения страниц»; «Внешняя память и ввод-вывод».

Данный материал разбирается на интерактивных лекциях в очном либо онлайн формате на университетской платформе для проведения телеконференций.

В качестве контроля освоения теоретического материала при проведении промежуточной аттестации обучающимся на экзаменационных занятиях предлагается самостоятельно ответить на два случайно выбранных вопроса и две случайно выбранных задачи из числа вопросов и задач, приведенных в параграфах «Вопросы». Самостоятельная подготовка по теоретической части курса включает написание рукописного конспекта ответов на экзаменационные вопросы. При ответе на экзаменационные вопросы теоретической части курса, в том числе в случае онлайн формата промежуточной аттестации, допускается использование лично подготовленных рукописных конспектов.

Углубленное изложение предлагаемого теоретического материала можно найти, в частности, в учебниках Э. Таненбаума и других авторов, приведенных в списке литературы. При самостоятельной подготовке обучающимся рекомендуется ознакомиться с соответствующими главами дополнительной литературы, обзору которой обычно посвящается вводная лекция курса.

Практическая часть освоения курса «Операционные системы» по предлагаемому пособию содержится в разделе «Практика синхронизации и управление памятью» и включает изучение теории и практики системного программирования с использованием интерфейса программирования приложений ОС Windows (Windows API) на языке Си или C++. Предполагается, что студенты самостоятельно изучают теорию языка Си по литературе, в частности по монографии авторов языка Си Б. Кернигана и Д. Ритчи во вводной части курса. Технический материал по работе с Windows API можно найти в монографии «Windows via C/C++» Дж. Рихтера и К. Назара. Он рассматривается на практических занятиях.

Лабораторный практикум включает пять лабораторных работ, варианты заданий к которым приведены в четвертом разделе данного пособия. Для задания рекомендуется выполнять в профессиональной интегрированной среде разработки на языках Си и C++, например, Microsoft Visual Studio или JetBrains CLion. Правильность выполнения задач лабораторного практикума проверяется преподавателем путем аудита кода и проверки работоспособности программ. В качестве итогового отчета по результатам освоения курса обучающийся оформляет написанный на лабораторных занятиях и проверенный преподавателем код в виде электронного отчета. Отчет служит допуском для прохождения промежуточной аттестации по дисциплине «Операционные системы».

Задания к лабораторному практикуму рассчитаны на индивидуальное выполнение. Задания реализуются в виде консольных приложений. Приложение может быть выполнено как интерактивное, запрашивающее входные данные с консоли в диалоге с пользователем, так и с набором заранее подготовленных в коде входных данных, результат обработки которых выводится в консоль.

———

# 1. УПРАВЛЕНИЕ ПРОЦЕССАМИ

## 1.1. Понятие процесса, методы планирования

Определение процесса. Диаграмма состояний процесса. Информационные структуры процесса: контекст и дескриптор. Виды алгоритмов планирования. Виды многозадачности. Потоки исполнения.

**Определение процесса.** Процесс - это выполнение последовательной программы на процессоре компьютера. Компьютерная программа является пассивной совокупностью инструкций, в то время как процесс представляет собой непосредственное выполнение этих инструкций.

С точки зрения операционной системы процесс является, с одной стороны, единицей исполнения (для процесса требуется выделять время процессора). С другой стороны, процесс рассматривается как заявка на потребление системных ресурсов (с ним связана область памяти, открытые файлы и другие ресурсы).

Подсистема управления процессами многозадачной операционной системы планирует выполнение процессов, то есть распределяет процессорное время между несколькими процессами, а также занимается созданием и уничтожением процессов, обеспечивает процессы необходимыми системными ресурсами, поддерживает взаимодействие между процессами.

**Диаграмма состояний процесса.** В многозадачных операционных системах процесс может находиться в одном из трех основных состояний. Это «выполнение», «ожидание», «готовность».

«Выполнение» - активное состояние процесса. В данном состоянии процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором.

«Ожидание» - пассивное состояние процесса. Процесс заблокирован и не может выполняться по своим внутренним причинам. Такими причинами могут являться: ожидание завершения операции

ввода-вывода; получение сообщения от другого процесса; освобождение необходимого для продолжения вычислений ресурса.

«Готовность» - также пассивное состояние процесса. В этом состоянии процесс заблокирован в связи с внешними причинами, по инициативе операционной системы. Процесс имеет все требуемые для выполнения ресурсы, однако процессор занят выполнением другого процесса.

В ходе своего выполнения каждый процесс переходит из одного состояния в другое в соответствии с алгоритмом планирования процессов, реализуемым в данной операционной системе (рис. 1).

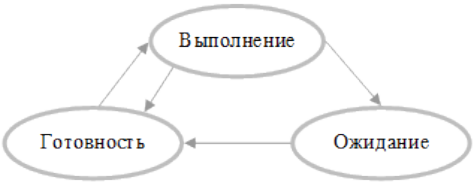



Рис. 1. Диаграмма состояния процесса

В состоянии «выполнение» в однопроцессорной системе может находиться только один процесс. В каждом из состояний «ожидание» и «готовность» - несколько процессов. Эти процессы образуют очереди. Исполнение процесса начинается с состояния «готовность». В данном состоянии он находится в очереди планировщика процессов операционной системы. При активизации процесс переходит в состояние «выполнение» и находится в нем до тех пор, пока ему не потребуется ждать некоторого события; он сам не освободит процессор; он будет принудительно вытеснен планировщиком.

Переходя в состояние «ожидания», процесс помещается в очередь, связанную с конкретным событием, которое он ожидает. Например, процесс может попасть в очередь процессов, ожидающих завершения ввода-вывода.

Если процесс вытесняется или добровольно отдает управление планировщику, он попадает в состояние «готовность» и помещается в очередь планировщика. В это же состояние процесс переходит из состояния «ожидание» после того, как произойдет ожидаемое событие.

**Информационные структуры процесса.** Информационные структуры, которые используются для управления исполнением процессов, называются *контекст* и *дескриптор*. Программный код только тогда начнет выполняться, когда для него операционной системой будет создан процесс. Создание процесса состоит из трех этапов: создания дескриптора и контекста процесса; включения дескриптора нового процесса в очередь готовых процессов; загрузки кодового сегмента процесса в оперативную память.

На протяжении существования процесса его выполнение может быть многократно прервано и продолжено. Для того, чтобы возобновить выполнение процесса, необходимо восстановить состояние его операционной среды. Состояние операционной среды состоит из значений регистров и программного счетчика, режима работы процессора, указателей на открытые файлы, информации о незавершенных операциях ввода-вывода, кодов ошибок, выполняемых данным процессом системных вызовов. Эта информация называется *контекстом процесса*. Контекст является зависимой от аппаратуры структурой данных.

Операционной системе для реализации планирования процессов требуется дополнительная информация: идентификатор процесса, состояние процесса, данные о степени привилегированности процесса, данные о нахождении процесса в очередях, указатель на контекст процесса. Эта информация называется *дескриптором процесса*. Содержание информационных полей дескриптора определяется разработчиком операционной системы, зависит от особенностей алгоритма планирования и не зависит от аппаратуры.

Очереди процессов представляют собой дескрипторы процессов, объединенные в списки. Поэтому каждый дескриптор содержит, по крайней мере, один указатель на другой дескриптор, соседствующий с ним в очереди. Такая организация очередей позволяет легко их переупорядочивать, включать и исключать процессы, переводить процессы из одного состояния в другое.

**Виды алгоритмов планирования.** Управление процессами включает в себя решение следующих задач: определение момента времени для смены выполняемого процесса; выбор процесса на выполнение из очереди готовых процессов; переключение контекстов между вновь запускаемым и снимаемым с исполнения процессом. Последняя задача решается на аппаратном уровне. То, каким образом решаются первые две задачи, определяется алгоритмом планирования.

Существует множество различных алгоритмов планирования процессов, по-разному решающих вышеперечисленные задачи, преследующих различные цели и обеспечивающих различное качество мультипрограммирования. Однако на планирование можно повлиять двумя способами. Можно управлять длительностью исполнения процессов, воздействуя на переход между состоянием «выполнение» - «готовность». Алгоритмы, использующие данный подход – это алгоритмы, основанные на квантовании. Можно управлять выбором готового процесса на исполнение, воздействуя на переход «готовность» - «выполнение». Этот подход используют алгоритмы, основанные на приоритетах.

В соответствии с алгоритмами, основанными на квантовании, смена активного процесса происходит, если процесс завершился и покинул систему; произошла ошибка; процесс перешел в состояние «ожидание»; исчерпан квант процессорного времени для данного процесса. Длительность отводимых на исполнение промежутков

времени (квантов времени) определяется системным таймером, который периодически шлет запросы прерывания центральному процессору. Обработчик прерывания передает управление планировщику операционной системы, который и выполняет переключение контекстов между процессами.

Процесс, который исчерпал свой квант, переводится в состояние «готовность» до тех пор, пока ему будет предоставлен новый квант процессорного времени. На выполнение в соответствии с определенным правилом выбирается новый процесс из очереди готовых процессов. Таким образом, ни один процесс не занимает процессор надолго, поэтому квантование широко используется в системах разделения времени.

Кванты, выделяемые процессам, могут быть одинаковыми для всех процессов или различными. Кванты, выделяемые одному процессу, могут быть фиксированной величины или изменяться пока процесс исполняется. Процессы, которые не полностью использовали выделенный им квант (например, из-за ухода на выполнение операций ввода-вывода), могут получить или не получить компенсацию в виде привилегий при последующем обслуживании. По-разному может быть организована очередь готовых процессов: циклически, по правилу «первый пришел – первый обслужен» (FIFO) или по правилу «последний пришел – первый обслужен» (LIFO).

Другая группа алгоритмов использует понятие приоритет. Приоритет – это число, характеризующее степень привилегированности процесса. Приоритет может выражаться натуральными, целыми или вещественными числами. Чем выше привилегии процесса, тем меньше времени он будет проводить в очередях. Приоритет может назначаться директивно администратором системы в зависимости от важности работы или внесенной платы, либо вычисляться самой операционной системой по определенным правилам. Он может

оставаться фиксированным либо изменяться во времени в соответствии с некоторым законом. В последнем случае приоритеты называются динамическими.

Существует две разновидности приоритетных алгоритмов: алгоритмы, использующие относительные приоритеты, и алгоритмы, использующие абсолютные приоритеты. В обоих случаях выбор процесса на выполнение из очереди готовых осуществляется одинаково: выбирается процесс, имеющий наивысший приоритет. По-разному решается проблема определения момента смены активного процесса. В системах с относительными приоритетами активный процесс выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние «ожидание» (произойдет ошибка или процесс завершится). В системах с абсолютными приоритетами выполнение активного процесса прерывается, если в очереди готовых процессов появился процесс, приоритет которого выше приоритета активного процесса. В этом случае прерванный процесс переходит в состояние готовности.

Во многих операционных системах алгоритмы планирования построены с использованием, как квантования, так и приоритетов. Например, в основе планирования лежит квантование, но величина кванта и порядок выбора процесса из очереди готовых определяется приоритетами процессов.

**Виды многозадачности.** В зависимости от того, кто является инициатором перехода из состояния «выполнение» в состояние «готовность» различают три вида многозадачности.

*Вытесняющая многозадачность* (preemptive multitasking) - это такой способ, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком операционной системы, а не самой активной задачей.

*Невытесняющая многозадачность* (non-preemptive multitasking) - это способ планирования процессов, при котором операционная система не является инициатором переключения контекста с текущего процесса на новый процесс. Такое переключение может осуществляться по инициативе пользователя с использованием программы-переключателя на фоновый процесс или по инициативе исполняющегося процесса.

Вид многозадачности называется *кооперативной* или *совместной многозадачностью* (cooperative multitasking), когда текущий процесс самостоятельно отдает управление планировщику операционной системы для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс.

Основным различием между вытесняющей и кооперативной многозадачностью является степень централизации механизма планирования задач. При вытесняющей многозадачности механизм планирования задач целиком сосредоточен в операционной системе. Программист пишет свое приложение, не заботясь о том, что оно будет выполняться параллельно с другими задачами. Операционная система выполняет следующие функции: определяет момент снятия с выполнения активной задачи; запоминает ее контекст; выбирает из очереди готовых задач следующую и запускает ее на выполнение, загружая ее контекст.

При кооперативной многозадачности механизм планирования распределен между системой и прикладными программами. Прикладная программа, получив управление от операционной системы, сама определяет момент завершения своей очередной итерации и передает управление операционной системе с помощью какого-либо системного вызова, а операционная система формирует очереди задач и выбирает следующую задачу на выполнение.

Для пользователей это означает, что управление системой теряется на произвольный период времени. Если приложение тратит

слишком много времени на выполнение какой-либо работы, например, на форматирование диска, пользователь не может переключиться с этой задачи на другую задачу. Эта ситуация нежелательна, так как пользователи обычно не хотят долго ждать, когда машина завершит свою задачу. Поэтому разработчики приложений для кооперативной операционной системы, возлагая на себя функции планировщика, должны создавать приложения так, чтобы они выполняли свои задачи небольшими частями. Например, программа форматирования может отформатировать одну дорожку диска и вернуть управление системе. После выполнения других задач система возвратит управление программе форматирования, чтобы та отформатировала следующую дорожку. Подобный метод разделения времени между задачами затрудняет разработку программ и предъявляет повышенные требования к квалификации программиста. Если произойдет зацикливание потока управления внутри задачи, это приведет к общему краху системы. В системах с вытесняющей многозадачностью такие ситуации, как правило, исключены, так как центральный планирующий механизм снимет зависшую задачу с выполнения.

Однако распределение функций планировщика между системой и приложениями не всегда является недостатком, а при определенных условиях может быть и преимуществом. Кооперативная многозадачность дает возможность разработчику приложений самому проектировать алгоритм планирования, наиболее подходящий для решаемой задачи. Так как разработчик сам определяет в программе момент отдачи управления, то исключаются нерациональные прерывания программ. Кроме того, легко разрешаются проблемы совместного использования данных: задача во время каждой итерации использует их монопольно. На протяжении итерации никакой другой процесс не изменит данные. Существенным

преимуществом систем с кооперативной многозадачностью является более высокая скорость переключения с задачи на задачу.

Кооперативная многозадачность реализована в Windows до версии 3.11 включительно, в Mac OS (macOS) до версии Mac OS X, а также во многих ранних UNIX-подобных операционных системах.

**Потоки исполнения.** Важным аспектом реализации современных многозадачных операционных систем является наличие потоков выполнения или нитей (thread). Как отмечалось, процессы - это сущности, являющиеся единицами планирования и единицами выделения ресурсов. Концепция потоков позволяет определить не одну, а несколько единиц планирования внутри пула ресурсов, выделенных процессу.

Каждый поток исполнения имеет собственный программный счетчик, стек, регистры, состояние. Потоки разделяют адресное пространство процесса, в котором они исполняются; глобальные переменные; открытые файлы; объекты, адресуемые через таблицу описателей объектов ядра процесса; статистическую информацию.

Существует несколько моделей реализации многопоточности в зависимости от того, как системные потоки, реализуемые в ядре операционной системы, соотносятся с пользовательскими потоками внутри процесса.

Потоки выполнения, созданные пользователем в *модели потоков* ядра (1:1), *соответствуют* потокам ядра. Это простейший возможный вариант реализации многопоточности.

В *модели пользовательских потоков* (N:1) предполагается, что все потоки выполнения уровня пользователя отображаются на поток уровня ядра, и ядро ничего не знает о составе прикладных потоков выполнения. При таком подходе переключение контекста может быть сделано очень быстро. Однако главным недостатком является отсутствие ускорения на многопроцессорных компьютерах,

потому что только один поток выполнения ядра может быть запланирован на процессор.

В *гибридной модели* (M:N) некоторое число M прикладных потоков выполнения отображаются на некоторое число N сущностей ядра или «виртуальных процессоров». Модель является компромиссной между моделью уровня ядра (1:1) и моделью уровня пользователя.

Специальные сущности, используемые для реализации моделей многопоточности (N:1) – это файберы (fiber - волокно). В Windows NT файберы организуются на базе потока. Особенностью является принудительное, выполняемое программистом, переключение контекста между файберами одного потока. Таким образом, на основе файберов могут быть реализованы преимущества кооперативной многозадачности внутри одного процесса.

Есть много различных, несовместимых друг с другом реализаций потоков. К ним относятся как реализации на уровне ядра, так и реализации на пользовательском уровне. Чаще всего они придерживаются более или менее близко стандарта интерфейса POSIX Threads.

### 1.2. Многозадачность в операционной системе Windows

Многозадачность на уровне процессов и на уровне потоков: группы процессов, планирование в режиме пользователя, пулы потоков, файберы. Планировщик: классы и уровни приоритета, структуры данных планировщика, переключение контекста, выбор готового потока на исполнение. Инверсия приоритета, способы преодоления. Многопроцессорная обработка.

**Многозадачность.** Рассмотрим некоторые особенности реализации многозадачности на примере операционной системы Windows. В Windows используются все современные технологии

управления многозадачностью для однопроцессорных и многопроцессорных систем. Подробную информацию можно найти в библиотеке разработчика Microsoft Docs (Microsoft Learn) в разделе About Processes and Threads.

В Windows имеются два основных способа использования многозадачности: многозадачность на уровне процессов и многозадачность на уровне потоков.

Многозадачность на уровне процессов используется, если требуется изоляция адресного пространства и ресурсов. Примером является реализация системных служб Windows в виде автономных процессов. Такая реализация предполагает наличие управляющего процесса, который запускает рабочие процессы, выполняет диагностику их состояния и при необходимости перезапускает. Крах системной службы не приводит к краху других процессов за счет изоляции ресурсов.

Для управления приложением, состоящим из нескольких процессов, в Windows предусмотрены специальные объекты – работы (job objects). Работа позволяет рассматривать группу процессов как целое. Операция, применяемая к работе, влияет на все процессы, связанные с ней. Например, можно установить групповой приоритет, размер рабочей области, или одновременно удалить все процессы, ассоциированные с работой.

Многозадачность на уровне потоков исполнения может использоваться при выполнении следующих задач.

1. Управление вводом из нескольких окон. Примером является оконный менеджер, который для каждой папки создаёт отдельный поток.

2. Управление вводом из нескольких коммуникационных устройств. Данный подход применяется в многопоточных серверах в интернете. Каждого подключенного клиента обслуживает отдель-

ный поток. Использование потоков позволяет упростить архитектуру программы сервера за счет применения блокирующих операций ввода-вывода с сохранением эффективности приложений, построенных на асинхронных операциях ввода-вывода.

3. Разделение задач различного приоритета. Может потребоваться для выполнения высокоприоритетным потоком критичных по длительности счета задач. Например, такой поток может заниматься сохранением телеметрической информации. Основной поток может осуществлять графическое отображение этой информации на терминале.

4. Сохранение интерактивности приложения при выполнении фоновой задачи. При выполнении длительных расчетов возникает необходимость в прерывании вычислений, ввода или корректировки параметров. Для этого вычисления организуют в фоновом низкоприоритетном потоке. Когда пользователь активирует элементы графического интерфейса, более приоритетный поток GUI немедленно прерывает вычисления и обрабатывает команды пользователя.

5. Использование преимуществ многопроцессорных систем для ускорения вычислений. Требуется разделить программу на несколько потоков, если разрабатывается ресурсоемкое приложения, которому необходимо использовать все вычислительные ресурсы современных многоядерных и многопроцессорных систем. Обычная однопоточная программа не сможет выполняться на нескольких ядрах и эффективно использовать возможности современной аппаратуры.

Реализация многозадачности с использованием одного процесса и нескольких потоков, если не требуется изоляции ресурсов, предпочтительна по следующим соображениям: происходит более быстрое переключение контекста между потоками одного процесса, по сравнению с переключением контекстов между потоками разных

процессов; потоки одного процесса разделяют глобальные переменные; потоки одного процесса разделяют описатели системных ресурсов.

Операционная система Windows обеспечивает альтернативные потокам методы многозадачности, например, асинхронный ввод-вывод, асинхронные вызовы процедур (APC) и другие.

Как отмечалось ранее, в Windows используется модель потоков ядра (1:1). Поэтому при использовании потоков рекомендуют использовать как можно меньше потоков для одного приложения, так как требуются ресурсы для хранения контекста потока, для отслеживания нескольких активных потоков и, обычно, более сложная синхронизация.

В современных версиях системы Windows NT поддерживается гибридная потоковая модель (M:N). В Windows она называется планированием в режиме пользователя (UMS - user mode scheduling). Модель следует использовать при необходимости создавать большое число потоков одновременно. При этом сочетаются преимущества быстрого переключения между такими потоками и отсутствие блокировки всех UMS-потоков, когда один из них выполняет блокирующую операцию ядра. В Windows 11 UMS не поддерживается.

Модель пользовательских потоков (N:1) реализуется при помощи файберов (fiber). Данная модель может использоваться, если требуется реализовать самостоятельно планировщик или для переноса в Windows приложений с кооперативной многозадачностью без значительной переделки кода.

В Windows также имеется встроенная поддержка парадигмы параллельного программирования «управляющий - рабочие». Она называется потоковые пулы (thread pools). Потоковый пул – это группа потоков, которые совместно обрабатывают очередь из заданий, сообщений таймера, запланированных асинхронных событий (например, завершение ввода-вывода). При этом достигается

уменьшение числа необходимых потоков и не требуется ручное управление очередью заданий.

**Планировщик.** В документации разработчика не описывается конкретный алгоритм планирования, так как он разный в разных версиях операционной системы Windows. Однако имеется возможность адаптировать конкретный планировщик под нужды приложения. Повлиять на алгоритм планирования можно путем управления классом и уровнем приоритета (priority class, priority level).

Класс указывается при создании процесса с использованием функции CreateProcess. Существует 6 классов приоритета:

IDLE_PRIORITY_CLASS – самый низкий класс приоритета, его имеют хранители экрана, средства сбора диагностики, средства индексирования и другие процессы фонового режима;

BELOW_NORMAL_PRIORITY_CLASS – приоритет ниже, чем приоритет по умолчанию;

NORMAL_PRIORITY_CLASS – приоритет по умолчанию;

ABOVE_NORMAL_PRIORITY_CLASS – приоритет выше, чем по умолчанию;

HIGH_PRIORITY_CLASS – приоритет процессов, непосредственно работающих с оборудованием, является приоритетом реального времени;

REALTIME_PRIORITY_CLASS – приоритет реального времени, который более приоритетен, чем системные потоки, работающие с диском, клавиатурой и мышью.

Класс приоритета определяется с помощью функции GetPriorityClass () и задается с помощью функции SetPriorityClass().

Типичная ситуация изменения приоритета возникает, когда процессу нужно гарантировать непрерывное выполнение некоторой операции. Для этого кратковременно приоритет повышается, затем понижается.

Внутри процесса устанавливаются относительные приоритеты для его потоков. Они называются уровнями приоритета или приоритетами потока:

THREAD_PRIORITY_IDLE – минимальный приоритет для фоновых потоков простоя;

THREAD_PRIORITY_LOWEST – более высокий приоритет для потоков простоя;

THREAD_PRIORITY_BELOW_NORMAL – приоритет для менее приоритетных рабочих потоков;

THREAD_PRIORITY_NORMAL – приоритет по умолчанию;

THREAD_PRIORITY_ABOVE _NORMAL – приоритет для более приоритетных рабочих потоков;

THREAD_PRIORITY_HIGHEST – приоритет реального времени;

THREAD_PRIORITY_TIME_CRITICAL – наивысший приоритет реального времени.

Уровень приоритета определяется с помощью функции GetThreadPriority() и задается с помощью функции SetThreadPriority(). Всем потокам по умолчанию назначается нормальный уровень приоритета. Если возникает необходимость задать уровень приоритета при создании потока, то используется следующая последовательность вызовов. С использованием CreateThread() создается поток в приостановленном состоянии с флагом CREATE_SUSPENDED. Далее устанавливается нужный уровень приоритета вызовом SetThreadPriority(). Затем поток переводится в готовое состояние вызовом ResumeThread().

Единицами планирования для диспетчера являются именно потоки, а не процессы. Для вычисления приоритета планирования, который называется базовым приоритетом (base priority) по специальным правилам выполняется комбинирование класса и уровня.

Операционная система Windows использует 32 базовых приоритета (от 0 до 31). Старшие приоритеты от 16 до 31 относятся к приоритетам реального времени. Младшие приоритеты относятся к приоритетам разделения времени. Приоритет 0 не назначается потокам пользователя, он зарезервирован за потоком обнуления страниц. Этот поток отвечает за очистку страниц в оперативной памяти, которые переходят от одного процесса к другому. Это обеспечивает защиту объектов согласно требованию класса безопасности С2.

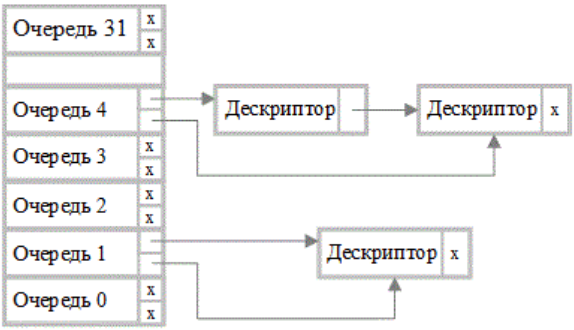



Рис. 2. Схема очередей планировщика Windows

В планировщике имеется 32 очереди потоков, в которые они попадают согласно своим приоритетам (рис. 2). Обслуживание, то есть назначение квантов процессорного времени внутри каждой очереди, осуществляется по принципу карусели (round-robin). Кванты времени получают потоки, находящиеся в непустой очереди наивысшего приоритета. Для переключения контекста между потоками Windows использует следующую последовательность шагов:

- сохранить контекст только что завершившегося потока;
- поместить этот поток в очередь соответствующего приоритета;
- найти очередь наибольшего приоритета, содержащую готовые потоки;

- удалить дескриптор потока из головы этой очереди, загрузить контекст, приступить к исполнению.

Некоторые потоки не находятся в структурах данных планировщика, то есть не являются готовыми. Такими потоками являются потоки, созданные флагом CREATE_SUSPENDED; остановленные командой SuspendThread(); ожидающие события синхронизации (например, в функции WaitForSingleObject() ) или завершения ввода-вывода.

Причины вытеснения текущего потока в планировщике Windows – истечение кванта времени; появление более приоритетного готового потока; переход исполняющегося потока к ожиданию события или завершения ввода-вывода.

Согласно описанной выше процедуре планирования низкоприоритетные потоки не должны получать обслуживание при наличии более приоритетных потоков. Чтобы предотвратить данную нежелательную ситуацию для потоков разделения времени вводится динамический приоритет. Динамический приоритет используется для продвижения потоков при наличии более приоритетных. Планировщик Windows кратковременно повышает приоритеты простаивающих готовых потоков. Процедура повышения приоритета (priority boost) выполняется в случае, когда процесс, содержащий поток, переходит на передний план; окно процесса получает событие от мыши и клавиатуры; наступило событие, которое ожидал поток или завершился ввод-вывод. В любом случае динамический приоритет не может быть меньше базового приоритета. По истечению каждого кванта динамический приоритет уменьшается на 1 до тех пор, пока не достигнет базового приоритета. Повышенный приоритет (priority boost) получают потоки с базовым приоритетом, не превышающим 15. Также в Windows имеется возможность изменения длительности квантов времени, назначаемых потокам.

**Инверсия приоритетов.** Наличие приоритетов может привести к неявной блокировке, когда более высокоприоритетный поток зависит от менее приоритетного потока. Например, ждет освобождения мьютекса, захваченного потоком.

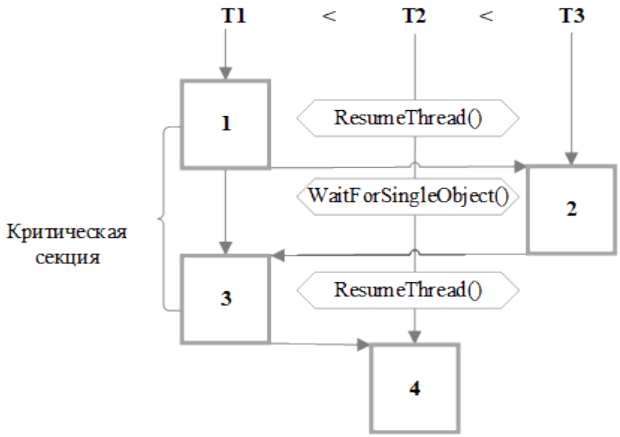



Рис. 3. Возникновение инверсии приоритетов

На рис. 3 показан сценарий возникновения инверсии приоритетов на примере трех потоков. Поток T1 имеет наименьший приоритет, поток T3 наибольший приоритет, поток T2 имеет промежуточный приоритет. Опишем последовательность возникновения инверсии.

Шаг 1. Поток T1 выполняет код в критической секции.

Шаг 2. Появляется поток T3 с наивысшим приоритетом в готовом состоянии. Следовательно, поток T1 вытесняется.

Шаг 3. Поток T3, ожидая событие освобождения мьютекса, отдает управление потоку T1, потому что в данный момент поток T1 наиболее приоритетный поток в готовом состоянии.

Шаг 4. Если в системе появляется готовый поток T2 в то время, как T1 не успел выйти из критической секции и освободить

мьютекс, то поток Т2 блокирует более приоритетный поток Т3. Таким образом, происходит инверсия приоритета.

В Windows 9x диспетчер обнаруживает зависимость более приоритетного потока от менее приоритетного потока, если эта зависимость возникает через объект ядра, и повышает приоритет менее приоритетного потока до уровня приоритета более приоритетного потока. Для предотвращения инверсии в современных версиях Windows планировщик учитывает время простоя готовых потоков и случайным образом повышает их динамический приоритет.

**Многопроцессорная обработка.** В операционной системе Windows имеется возможность управлять назначением потоков на конкретный процессор. Для управления таким назначением используются два атрибута.

Атрибут thread affinity определяет привязку потока к определенной группе процессоров. Этот атрибут представляет собой битовую маску, его указание вынуждает поток исполняться на указанном подмножестве процессоров. Для установки битовых масок привязки к процессорам используются функции SetThreadAffinityMask() и SetProcessAffinityMask(). Для считывания значения битовых масок привязки к процессорам используются функции GetProcessAffinityMask() и GetThreadAffinityMask().

Настройка привязки потока к процессорам может использоваться для отладки в обычных SMP (симметричных мультипроцессорных системах) при наблюдении за активностью потоков средствами диспетчера задач. Основным применением является повышение производительности многопоточных приложений при исполнении в архитектурах с неоднородным доступом к памяти (NUMA).

В таких архитектурах доступ по некоторым адресам памяти из заданного процессора может происходить быстро, а по некоторым

медленнее. Точно также выделяются группы более и менее связанных между собой процессоров (рис. 4). Зная топологию процессоров на компьютере и топологию задач приложения, удается оптимизировать приложение. Для получения данных о топологии имеются специальные функции. Если зависимые потоки поместить

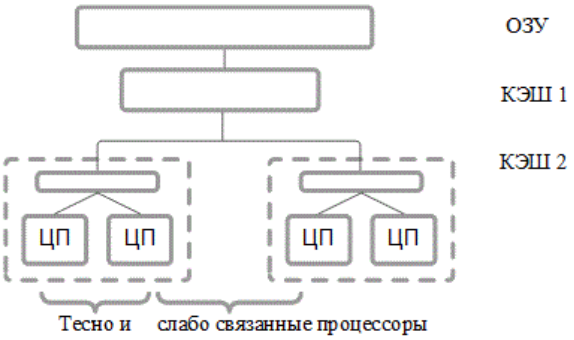



Рис. 4. Архитектура с неоднородным доступом к памяти (NUMA)

на один процессор, то они будут выполняться быстрее, так как будут обрабатываться через один внутренний кэш процессора, а не внешний, как в противном случае (рис. 4). Однако жесткая привязка потоков к процессорам может снизить производительность. Для рекомендации планировщику назначать, по возможности, поток на процессор имеется еще один атрибут – идеальный процессор (ideal processor). Чтение и установка этого атрибута выполняется функциями GetThreadIdealProcessorEx() и SetThreadIdealProcessor().

———

### 1.3. Вопросы

1. Абстракция процесса, управление процессами в многозадачной операционной системе. Определение процесса. Диаграмма состояния, контекст, дескриптор процесса. Квантование и приоритетное планирование. Нити (потоки исполнения).
2. Функциональные возможности многозадачности в ОС Windows. Способы использования многозадачности в приложениях.
3. Планировщик ОС Windows. Класс и уровень приоритета. Переключение контекста. Потоки, не являющиеся готовыми. Динамический приоритет.
4. Эффект инверсии приоритетов. Пример возникновения инверсии. Способы преодоления.
5. Мультипроцессорная обработка в ОС Windows. Термины, вызовы API, их назначение.

## 2. СИНХРОНИЗАЦИЯ

### 2.1. Синхронизация в разделяемой памяти

Необходимость синхронизации на примере возникновения состояния состязания. Аппаратная реализация синхронизации. Задача о критической секции. Решение задачи для двух процессов в алгоритме Петерсона. Проблемы использования разделяемых переменных: агрессивная оптимизация, голодание, ложное разделение.

**Необходимость синхронизации.** Для того, чтобы параллельно или псевдопараллельно выполняющиеся процессы могли решать общую задачу, их исполнение необходимо синхронизировать. Существуют два типа задач синхронизации: конкурентные и кооперативные.

Синхронизация при совместном использовании разделяемого ресурса. Например, имеется очередь заданий на печать, в которую добавляют свои задания независимо работающие процессы. Такой вид взаимодействия (тип синхронизации) называется *конкурентным*. Его отличительной особенностью является то, что остановка одного из процессов-участников вне протокола взаимодействия не влияет на возможность других процессов продолжать работу.

Если речь идет об уведомлении одного процесса о завершении какой-либо операции другим процессом, выполняется *кооперативное* взаимодействие. Здесь, напротив, остановка любого участника со временем приведет к остановке всей системы процессов. Типичный пример такой синхронизации – буферизация данных, передаваемых по цепочке из процессов.

При отсутствии синхронизации процессов при совместном использовании разделяемого ресурса возникает искажение данных, называемое «состоянием состязания» (race condition). Пусть разделяемый ресурс – это глобальная переменная, используемая как счет-

чик числа обращений к некоторому ресурсу. При обращении к ресурсу процесс выполняет инкремент счетчика. Рассмотрим пример, иллюстрирующий искажение данных даже для такого простого ресурса.

Разделяемый ресурс – глобальная переменная типа long

```
long  g_x = 0;
```

Процессы представлены идентичными функциями потоков Thr1 и Thr2.

```
DWORD  WINAPI  Thr1(PVOID)    {
      /*обращение к ресурсу*/
      g_x++; /*увеличение счетчика обращений*/
      return 0;
}

DWORD  WINAPI  Thr2(PVOID)    {
      /*обращение к ресурсу*/
      g_x++; /*увеличение счетчика обращений*/
      return 0;
}
```

Для управления потоками объявляются массив указателей на функции потока thr_arr и массив описателей потоков thr_hnd.

```
LPTHREAD_START_ROUTINE thr_arr [2] = {Thr1, Thr2}
HANDLE  thr_hnd [2];
```

В функции main() вначале запускаем наши потоки на исполнение вызовом CreateThread(), проверяя и обрабатывая ошибки вызовом функции завершения процесса ExitProcess(). Затем, при помощи функции WaitForMultipleObjects ожидаем завершения всех запущенных потоков.

```
int main() {
      DWORD id;
      for (inti=0; i<2; i++) {
        thr_hnd [i] = CreateThread(
          NULL,0,thr_arr[i],NULL,0,&id);
        if (thr_hnd [i] = = NULL) ExitProcess(-1);
      }
      WaitForMultipleObjects(
        2, thr_hnd, TRUE, INFINITE);
      return 0;
}
```

Очевидным кажется состояние переменной g_x=2 после завершения запущенных потоков. Однако истинное постусловие рассмотренной программы - g_x=1 $\vee$ g_x=2. Для того, чтобы понять почему значение переменной g_x может также оказаться равным 1, рассмотрим как представляется оператор g_x++ при компиляции программы. Столбцы Thr1 и Thr2 показывают ассемблерные инструкции потоков, а левый столбец – порядок исполнения этих инструкций на однопроцессорной машине. Показанный порядок выполнения инструкций возможен при переключении контекста в момент выполнения инкремента. С учетом того, что каждый из потоков имеет индивидуальную копию регистра EAX, действия потока Thr2 будут потеряны. Итоговое значение g_x оказывается равным 1.

```
      Thr1                      Thr2
/*     g_x++      */     /*     g_x++     */
(1) MOV EAX, [g_x]
(2)                      MOV EAX, [g_x]
(3)                      ADD EAX, 1
(4)                      MOV [g_x], EAX
(5) ADD EAX, 1
(6) MOV [g_x], EAX
```

Из-за «расщепления» команды инкремента возникает состояние состязания (race condition). Для предотвращения этого эффекта необходимо, чтобы эти три ассемблерные команды выполнялись как единое целое.

Для того, чтобы достичь неделимости инкремента и некоторых других арифметических операций в программном интерфейсе Windows имеется группа функций с префиксом interlocked. При использовании функции InterlockedExchangeAdd() правильный код подсчета обращений к ресурсу выглядит следующим образом.

```
DWORD WinAPI Thr1(PVOID) {
    /*обращение к ресурсу*/
      InterlockedExchangeAdd(&g_x,1);/*увеличение
       счетчика обращений*/
        return 0;
}
```

**Аппаратная реализация синхронизации.** Для реализации неделимых операций в системе команд компьютеров имеется инструкция «проверить и установить блокировку» (test and set lock). Данная инструкция реализует базовую атомарную операцию, используя которую легко построить более сложные операции. Команда неделимым образом записывает ненулевое значение по адресу в памяти, одновременно сохраняя старое значение по этому адресу в регистре.

```
enter:
   TSL Reg, [Lock]
   CMP Reg, #0;0 значит, что текущий поток
   ; выполнил блокировку
   JNE enter; и ему можно войти в критическую секцию
      ;выполняем неделимую последовательность команд
leave:
   MOV [Lock], #0
```

Такой метод синхронизации называется спин-блокировка потому, что в цикле происходит постоянный опрос значения переменной по адресу Lock. При использовании interlocked-функции InterlockedExchange() реализация спин-блокировки выглядит следующим образом.

```
BOOL g_ResInUse = FALSE;
//перед неделимой последовательностью
while(InterlockedExchange(&g_ResInUse,TRUE)==TRUE)
     Sleep(0);
// после неделимой последовательности команд
InterlockedExchange(&g_ResInUse,FALSE);
```

Функция InterlockedExchange() присваивает значение, переданное во втором параметре, переменной, адрес которой указан в первом, и возвращает значение до модификации.

**Задача о критической секции.** Помимо аппаратной реализации возможна и программная реализация неделимой последовательности операций с использованием разделяемых переменных. Э. Дейкстра в 1965 году сформулировал постановку задачи. Т. Деккер описал первое корректное решение.

Задача получила название задача о критической секции. Она формулируется следующим образом. Каждый из процессов, участвующих во взаимодействии в цикле, последовательно выполняет четыре секции кода.

```
white (1)      {
          < протокол входа - enter( )>;
          < код в критической секции >;
          < протокол выхода - leave( )>;
          < код вне критической секции >;
}
```

Для решения задачи требуется выполнение четырех условий.

1. Процессы не должны находиться одновременно в критических секциях.

2. Не должно быть предположений о скорости выполнения процессов.

3. Процесс вне критической секции не должен блокировать другие процессы.

4. Если процесс начал выполнять протокол входа, то он рано или поздно должен войти в критическую секцию.

**Решение задачи для двух процессов в алгоритме Петерсона.** С момента постановки задачи было предложено несколько решений. В 1981 году Петерсон предложил наиболее компактный из известных вариантов решений задачи о критической секции для двух процессов. Он носит название алгоритм разрыва узла. Ниже показана последовательность синтеза этого алгоритма.

Для понимания сложности проблемы рассмотрим «наивный» алгоритм реализации протоколов входа и выхода из критической секции.

```
int lock = 0
void enter( )   {
   while (lock != 0) /*ждем*/;
   lock = 1;
}
void leave( ) { lock = 0; }
```

Очевидно, что такие протоколы не гарантируют выполнения условия взаимного исключения. Дело в том, что, если два процесса одновременно будут выполнять проверку значения флага lock, то они оба увидят разрешенное значение 0 и одновременно войдут в критическую секцию.

Поступим по-другому. Введем переменную turn, которая определяет очередь входа в критическую секцию. Если имеются два процесса, то пусть первый процесс входит в критическую секцию, если turn=0, а второй – если turn =1.

```
int turn = 0;
```

```
while(TRUE){
 while(turn!=0)/*ждем*/;
 < критическая секция >
 turn = 1;
 <вне критической
 секции>;
}
```

```
while(TRUE){
 while(turn!=1)/*ждем*/;
 < критическая секция >
 turn = 0;
 <вне критической
   секции>;
}
```

Этот способ обеспечивает выполнение условия взаимного исключения. Однако не выполняется третье условие: остановившись вне критической секции, первый процесс заблокирует второй и наоборот.

Правильное решение совмещает оба подхода и выглядит следующим образом.

```
int turn = 0;
int interested[2] = { FALSE, FALSE };

void enter( int process){
     int other = 1-process;
     interested[process] = TRUE;
     turn = process;
     while(turn==process&&
      interested[other]==TRUE)/*ждем*/;
}
void leave(int process){interested[process]=FALSE;}
```

Работа алгоритма основана на следующих заключениях. Модификацию условий нужно выполнять до проверки, а не после. Для каждого из процессов нужно завести по флагу (массив interested). Ненулевое значение флага свидетельствует о том, что соответствующий процесс готов войти в критический участок, поэтому каждый из процессов перед входом в критический участок должен выпол-

нять проверку while (interested = = TRUE). Если два процесса одновременно начинают выполнять проверку, то возникает тупик. Для разрешения тупика (разрыва узла) вводится вспомогательная переменная turn, которая в случае конфликта разрешает вход в критическую секцию только одному процессу.

**Агрессивная оптимизация.** Реализация синхронизации с использованием разделяемых переменных и примитива test and set lock имеет максимальное быстродействие. Однако является достаточно сложной в силу ряда эффектов.

Эффект агрессивной оптимизации возникает, когда компилятор на основе предположения о неизменяемости переменной вне текущего потока сохраняет ее значение в регистре, тем самым нарушает логику работы программы. Например, в коде, показанном ниже, используется флаг g_fFinished для ожидания завершения операции в потоке с функцией CalcFunc.

```
volatile BOOL g_fFinished = FALSE;
/*volatile – загружаем значение из памяти при каждом
обращении к переменной (отключает оптимизацию кода
компилятором).*/

int main( )   {
     CreateThread (…, CalcFunc,…);
     while (g_fFinished = = FALSE);
}
DWORD WINAPI CalcFunc (PVOID)   {
     …
     g_fFinished = TRUE;
}
```

Если не использовать ключевое слово volatile, то возникнет риск того, что значения флага загрузятся в регистр процессора перед вычислением while, и в дальнейшем проверяться будет значение из регистра. То есть, ждущий процесс никогда не увидит окончания вычислений.

**Голодание.** При реализации спин-блокировки возможна ситуация, когда поток длительное время опрашивает условие входа в критическую секцию. Периодически это условие оказывается истинным. Тем не менее, поток не может войти в свою критическую секцию потому, что во время истинного значения условия входа поток не получает время процессора. Проблема решается введением задержек перед повторными попытками проверки условия входа. Например, пусть два потока выполняют идентичный код и в начальный момент времени остановились на строке (1).

```
(1)   while(1){
(2)         while(InterlockedExchange(&x,1));
(3)         /*критическая секция*/
(4)         InterlockedExchange(&x,0)
(5)   }
```

Когда первому потоку будет отведен квант времени, он в цикле выполняет строки (1)-(5). Так как строка (3) выполняется значительно дольше, чем остальные строки, первый поток по истечении кванта остановится на ней. Далее квант времени выделяется второму потоку. Однако он в течение отводимого ему кванта сможет только выполнять цикл (2). Ситуация повторяется при каждом следующем обслуживании: второй поток не сможет войти в свою критическую секцию.

**Ложное разделение.** Еще один эффект связан с понижением быстродействия программы при неправильном объявлении переменных. Например, в объявлении, показанном ниже, две глобальные переменные объявлены вместе. Одна их них используется исключительно в функции потока Thread1, а другая в функции потока Thread2.

```
volatile int x = 0;  // используется в Thread1( )
volatile int y = 0;  // используется в Thread2( )
```

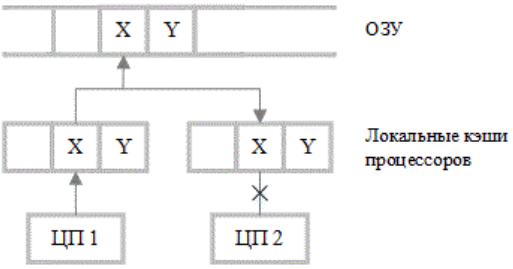



Рис. 5. Переменные X и Y находятся в одной линии кэша, к ним невозможен одновременный доступ из разных процессоров

Потоки кажутся независимыми, однако это не так. Если поток Thread1 выполняет операцию с переменной x, поток Thread2 вынужден приостанавливать операцию с переменной y и наоборот, из-за чего снижается быстродействие программы (рис. 5). Это происходит потому, что для ускорения работы с памятью каждый из процессоров использует локальный кэш, однако загрузка значений в кэш производится не побайтно, а блоком. Обычно это участок памяти, выровненный по 32-байтной границе. Он называется кэш-линия. Если две переменные попадают в одну кэш-линию, то эффект от использования кэш-памяти пропадает. Поэтому следует выравнивать переменные по границам кэш-линий или вводить фиктивные переменные, чтобы гарантировать, что переменные не попадут в одну кэш-линию.

Правильные объявления будут выглядеть следующим образом.

```
#define CACHE_LINE  32
#define CACHE_ALIGN __declspec(align(CACHE_LINE))
// используется в Thread1( )
volatile CACHE_ALIGN int x = 0;
// используется в Thread2( )
volatile CACHE_ALIGN int y = 0;
```

Для того, чтобы избежать проблем, сопутствующих синхронизации с разделяемыми переменными при сохранении высокой скорости, в Windows реализована группа функций для организации критической секции. Функция InitializeCriticalSection() используется для настройки структуры данных, служащей для управления критической секцией перед первым входом. Функция DeleteCriticalSection() служит для перевода структуры управления критической секцией в исходное состояние. Функции EnterCriticalSection() вместе с TryEnterCriticalSection() реализуют протокол входа в критическую секцию. Функция LeaveCriticalSection() реализует протокол выхода из критической секции. Дополнительно имеются функции InitializeCriticalSectionAndSpinCount() и SetCriticalSectionSpinCount() для настройки числа попыток входа в критическую секцию перед переходом в режим ядра (или возврата в случае TryEnterCriticalSection() ).

Достоинством функций является высокое быстродействие. Однако в них нельзя указать тайм-аут (аварийный выход через определенное время), их нельзя применять для организации взаимодействия между процессами, только для взаимодействия между потоками одного процесса. От этих недостатков свободны процедурные методы синхронизации.

### 2.2. Процедурные методы синхронизации в ОС Windows

Объекты синхронизации, функции ожидания. Управление объектами ядра. События. Семафоры. Мьютексы. Пример задачи об ограниченном буфере.

**Объекты синхронизации, функции ожидания.** Процедурные методы синхронизации в Windows используются по отношению к объектам, реализованным в ядре операционной системы. Для такой синхронизации применяются только объекты ядра, которые могут

находиться в сигнальном (свободном) и несигнальном (занятом) состоянии. К ним относятся рассмотренные ранее объекты: задания, процессы, потоки. Эти объекты переходят в сигнальное состояние при завершении исполнения. Имеется группа объектов, используемых специально для синхронизации потоков и процессов, – это события, мьютексы, семафоры и таймеры. Отдельную группу образуют объекты, предназначенные для синхронизации ввода-вывода.

С точки зрения программиста все перечисленные объекты имеют общее свойство: их описатели можно использовать в функциях ожидания WaitForSingleObject(), WaitForMultipleObjects() и некоторых других. Если объект находится в несигнальном состоянии, то вызов функции ожидания с описателем объекта блокируется. Дескриптор потока, который вызвал функцию, помещается в очередь этого объекта ядра, а сам поток переходит в состояние ожидания. Когда объект ядра переходит в сигнальное состояние, выполняются действия, специфичные для данного типа объекта ядра. Например, может быть разблокирован один или все потоки, ожидающие на данном объекте ядра.

В функцию WaitForSingleObject() передаются два параметра: описатель объекта и значение тайм-аута. Тайм-аут определяет предельное время нахождения потока в блокированном состоянии. В функцию WaitForMultipleObjects() передается массив описателей объектов ядра. Для этого указывается адрес начала массива и количество описателей в нем. Также указывается параметр, определяющий семантику ожидания на группе описателей: можно ждать перехода всех объектов ядра в сигнальное состояние или какого-то одного объекта. Указывается также тайм-аут.

При возврате из функции ожидания обычно требуется определить причину завершения функции. В случае ошибки функция возвращает значение WAIT_FAILED. Для уточнения состояния

ошибки далее используют GetLastError() и другие функции. В случае завершения по тайм-ауту возвращается значение WAIT_TIMEOUT. Если причиной возврата является переход в сигнальное состояние, возвращается значение WAIT_OBJECT_0.

Если выполняется ожидание на функции WaitForMultipleObjects(), то, чтобы узнать, какой именно объект перешел в сигнальное состояние, нужно проверить, что значение, возвращенное функцией, не равно WAIT_FAILED и WAIT_TIMEOUT и вычесть из него константу WAIT_OBJECT_0. В результате мы получим индекс описателя объекта, перешедшего в сигнальное состояние, в массиве описателей, переданном в функцию WaitForMultipleObjects().

**Управление объектами ядра.** Рассмотрим операции, относящиеся как к объектам, использующимся в функциях ожидания, так и к другим объектам ядра в операционной системе Windows.

Для *создания* объектов ядра используются индивидуальные для каждого объекта функции, имеющие префикс Create. Например, мьютекс может быть создан вызовом

HANDLE mutex=CreateMutex(NULL, FALSE, NULL).

Обычно первым параметром всех функций, создающих объекты ядра, является структура атрибутов безопасности, а последним – имя объекта.

*Закрытие* описателя любого объекта ядра выполняется вызовом функции CloseHandle().

Чтобы понять принцип работы функции, рассмотрим более детально, что представляет собой описатель объекта ядра (рис. 6).

Для каждого процесса в ядре операционной системы создается таблица описателей. Запись в таблице содержит некоторые атрибуты описателя и указатель на объект ядра. На один и тот же объект ядра могут указывать несколько описателей, возможно из таблиц описателей разных процессов. В любом объекте ядра имеется спе-

циальный атрибут для подсчета ссылок на объект. Значение описателя, используемое в адресном пространстве процесса, – это смещение на запись в таблице описателей процесса.

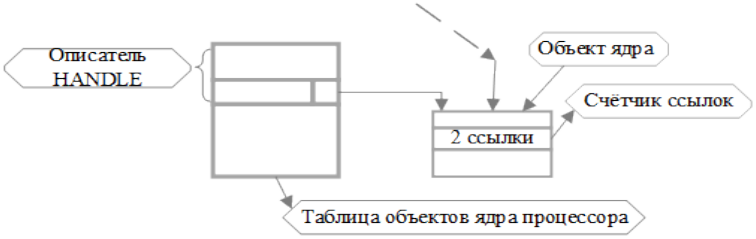



Рис. 6.Описатель объекта ядра в Windows

При закрытии описателя происходит модификация атрибута, указывающего на то, что запись теперь недействительна. Далее происходит декремент счетчика ссылок у объекта. Если значение счетчика ссылок достигает нуля, происходит удаление объекта из памяти ядра операционной системы.

Важное преимущество объектов ядра – возможность их использования для синхронизации потоков, принадлежащих разным процессам. Для этой цели нужно уметь передавать описатели объектов между процессами. Их можно передавать путем наследования, именования и дублирования.

При *наследовании* вначале необходимо модифицировать атрибут наследования в таблице описателей процесса. Это можно сделать непосредственно при создании описателя, как показано ниже.

```
SECURITY_ATTRIBUTES sa;
sa.nlength = sizeof(sa);
sa.lpSecurityDescriptor=NULL; /*NULL – защита по умолчанию; определяет, кто может пользоваться объектом (при NULL – создатель процесса и администраторы) */
sa.bInheritHandle=TRUE; /*наследовать описатель*/
HANDLE hMutex = CreateMutex (&sa, FALSE, NULL);
```

Можно воспользоваться функциями GetHandleInformation() и SetHandleInformation() для изменения атрибутов описателя уже после создания объекта ядра.

Наследование описателей происходит следующим образом (рис. 7). При создании процесса функцией CreateProcess() для него создается новая таблица описателей. В эту таблицу копируются все записи родительского процесса, имеющие атрибут наследования. Записи помещаются по тем же смещениям, по которым размещались исходные записи родительского процесса. Для каждой скопированной записи производится инкремент счетчика ссылок у связанного с ней объекта ядра.

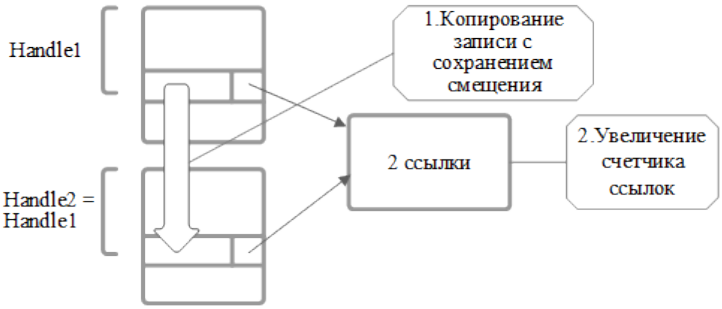



Рис. 7. Схема наследования описателя в Windows

Дочерний процесс может использовать ссылки на унаследованные объекты ядра. Для передачи значений унаследованных описателей в дочерний процесс обычно используется командная строка или переменные окружения. Если дочерний процесс уже создан, то нельзя воспользоваться механизмом наследования.

При создании объекта ядра может быть указано его имя:

```
HANDLE mutex = CreateMutex(NULL, FALSE, “SomeMutex”);.
```

В данном случае любой процесс может получить *доступ к объекту ядра по имени*, если такой доступ разрешен настройками безопасности. При повторном вызове функции Create*() проводится проверка: имеется ли объект ядра с данным именем и типом. Если объект с таким именем и типом уже существует, то функция не создает новый объект, а возвращает описатель существующего объекта (при этом остальные параметры в вызове игнорируются). При необходимости можно проверить, создан ли объект ядра или получен описатель ранее созданного объекта:

```
if(GetLastError()==ERROR_ALREADY_EXISTS){
   /*если создан ранее*/
}.
```

Механизм ссылки на объекты ядра подходит для всех случаев взаимодействия, когда можно определить соглашения на имена создаваемых объектов. Этот механизм также часто используется для контроля количества запущенных экземпляров приложения.

Универсальным способом передачи описателей объектов ядра между процессами является *дублирование описателей*. Дублирование выполняется функцией DuplicateHandle():

```
BOOL DuplicateHandle(
HANDLE hSourceProcessHandle,/*описатель исходного
                              процесса*/
HANDLE hSourceHandle, /*дублируемый описатель
                       процесса-источника*/
HANDLE hTargetProcessHandle, /*процесс-приемник*/
PHANDLE phTargetHandle, /*описатель в процессе-
                          приемнике*/
DWORD dwDesiredAccess, /*настройка атрибутов
                        дублируемого описателя*/
DWORD dwQptions.
```

В дублировании принимают участие (в общем случае) три процесса: процесс, выполняющий дублирование, процесс – источник и процесс – приемник. Если источником или приемником является

управляющий процесс, то вместо соответствующего описателя помещается вызов функции GetCurrentProcess(). Особенностью дублирования является то, что необходимо использовать механизм межпроцессного взаимодействия для передачи продублированного описателя в процесс – приемник. Функция DuplicateHandle() его сформирует, но нужно также поставить в известность сам процесс – приемник о том, что в его таблицу описателей добавлен новый описатель.

Далее рассмотрим объекты синхронизации и специфические для каждого объекта функции процедурной синхронизации.

**События** используются при реализации кооперативной синхронизации, когда один поток ждет поступления данных от другого. При наступлении события объект переходит в сигнальное состояние. Если событие не наступило – находится в несигнальном состоянии. В зависимости от того, каким образом осуществляется перевод события в несигнальное состояние, существуют два типа событий: событие со сбросом вручную и событие с автоматическим сбросом. Оба типа объектов создаются функцией CreateEvent():

```
HANDLE CreateEvent(
LPSECURITY_ATTRIBUTES sa,/*атрибуты безопасности*/
BOOL fManualReset,/*тип объекта TRUE - со сбросом
вручную*/
BOOL fInitialState,/*начальное состояние события*/
LPCTSTR name);/*имя события*/.
```

Для управления событием используются следующие функции: SetEvent() – устанавливает событие в сигнальное состояние; ResetEvent() – устанавливает событие в несигнальном состояние; PulseEvent() – устанавливает в сигнальное состояние и сбрасывает в несигнальное.

Функция PulseEvent() выполняет разблокирование потоков, если таковые имеются в момент ее вызова. События с автоматиче-

ским сбросом целесообразно использовать, если происходит периодическое ожидание завершения события. В данном случае вместо комбинации вызовов WaitForSingleObject()/ResetEvent() достаточно использовать один вызов WaitForSingleObject().

**Семафоры** – универсальные объекты процедурной синхронизации, предложены Э. Дейкстра. Семафоры выполняют роль счетчика числа доступных ресурсов. С семафорами определены две операции. Для возврата ресурсов в пул доступных ресурсов используется операция увеличения счетчика (up-операция или V-операция). Для захвата ресурсов используется операция уменьшения счетчика (down-операция или P-операция). Если счетчик ресурсов принимает значение ноль, то поток блокируется, до тех пор, пока ресурс не будет возвращен в пул другим потоком с использованием up или V-операции. В любой момент счетчик не может быть больше максимального числа ресурсов и меньше нуля.

Семафор создается при помощи функции CreateSemaphore:

```
HANDLE CreateSemaphore(
LPSECURITY_ATTRIBUTES sa, /*атрибуты безопасности*/
LONG lInitialCount,/*начальное значение счетчика
                    (ресурсов)*/
LONG lMaxCount,/*предельное значение счетчика
                (ресурсов)*/
LPCTSTR name);/*имя семафора*/.
```

Инкремент счетчика семафора выполняется при помощи вызова функции BOOL ReleaseSemaphore (HANDLE, LONG ReleaseCount, LPLONG lpPreviousCount). В ней, в отличие от абстрактных операций up или V, можно указать, насколько следует увеличить счетчик (параметр ReleaseCount) и узнать его предыдущее значение (параметр lpPreviousCount). Декремент счетчика выполняется при помощи любой функции ожидания, примененной к семафору, например, WaitForSingleObject().

**Мьютекс** (MUTEX – MUTualEXclusion, взаимное исключение) – бинарный семафор, специально предназначенный для решения задачи взаимного исключения, защиты критической секции. Для создания мьютекса используется функция CreateMutex():

```
HANDLE CreateMutex(
LPSECURITY_ATTRIBUTES sa,/*атрибуты безопасности*/
BOOL  fInitialOwner,/*признак,  является  ли  поток,
создающий мьютекс его владельцем*/
LPCTSTR name);/*имя мьютекса*/.
```

Операция освобождения мьютекса (up или V операция) выполняется вызовом функции BOOL ReleaseMutex (HANDLE). Операция захвата мьютекса выполняется при помощи любой функции ожидания, например, WaitForSingleObject().

Мьютекс можно рассматривать как бинарный семафор. Также мьютекс похож на критическую секцию. От семафора мьютекс отличается тем, что он имеет атрибут владения (как критическая секция). Отличие от критических секций состоит в том, что при помощи мьютекса можно синхронизировать потоки в разных процессах и указывать тайм-аут.

Если поток владеет мьютексом, то вызов функции ожидания с этим мьютексом завершится успешно, при этом увеличится внутренний счетчик рекурсий. Функция ReleaseMutex() выполняет декремент счетчика рекурсий и освобождает мьютекс, если счетчик достигает значения ноль. Также ведет себя и критическая секция.

Если попытаться освободить мьютекс из протока, который им не владеет, то возникнет ошибка и функция GetLastError() вернет значение ERROR_NOT_OWNER.

Если поток, владеющий мьютексом, завершается, то мьютекс переходит в несигнальное состояние и может быть использован другими потоками. Вызов функции ожидания с таким мьютексом будет выполнен успешно, но функция ожидания, WaitForSingleObject() вернет код WAIT_ABANDONED.

**Пример задачи об ограниченном буфере.** Проиллюстрируем применение семафоров и мьютексов для решения одной из классических задач синхронизации – задаче об ограниченном буфере, также известной как проблема производителя и потребителя.

Два процесса совместно используют буфер ограниченного размера. Один из них (производитель) помещает данные в этот буфер, а другой (потребитель) считывает их оттуда. Требуется обеспечить ожидание процесса-производителя, когда буфер заполнен и ожидание процесса-потребителя, когда буфер пуст. В других случаях процессы могут добавлять (производитель) и извлекать (потребитель) данные из буфера.

Приведем программу на псевдокоде с условными операциями для семафоров, мьютексов и буфера, иллюстрирующую решение проблемы для одного производителя и одного потребителя.

```
semaphore mutex = 1;// защита критического
                    // ресурса - буфера
semaphore empty = 100; //число пустых сегментов бу-
фера
semaphore full = 0; // число полных сегментов буфера

// код процесса - производителя
void producer(){
      int item;
      while(1){
            item=produce_item();// создать данные,
            //помещаемые в буфер
            down(&empty); // уменьшить счетчик пустых
            // сегментов буфера
            down(&mutex);// вход в критическую секцию
            insert_item(item);// поместить в буфер
            // новый элемент
            up(&mutex); //выход из критической секции
            up(&full); //увеличить счетчик
}}                      // полных сегментов буфера
// код процесса - потребителя
void consumer(void){
      int item;
```

```
        while(1){
              down(&full); // уменьшить счетчик
              // полных сегментов буфера
              down(&mutex);// вход в критическую секцию
              item = remove_item(;// извлечь элемент
              // из буфера
              up(&mutex); //выход из критической секции
              up(&empty); //увеличить счетчик
              // пустых сегментов буфера
              consume_item(item);// обработать элемент
}}
```

Заметим, что семафоры в приведенном примере используются по-разному. Семафор mutex служит для реализации взаимного исключения, то есть конкурентной синхронизации. В реальной программе для операционной системы Windows его необходимо реализовывать при помощи мьютекса или критической секции. Семафоры full и empty используются для задания определенной последовательности событий при кооперативной синхронизации. В реализации для Windows уместно использовать семафоры. Таким образом, проблема об ограниченном буфере – пример гибридной проблемы, где встречается как кооперативная, так конкурентная синхронизация.

### 2.3. Тупики и защита от них

Определение тупика. Условие возникновения тупика. Моделирование блокировок. Стратегии борьбы с блокировками: игнорирование проблемы, обнаружение и восстановление, динамическое избегание тупиковых ситуаций, предотвращение тупиков при помощи устранения одного из условий их возникновения.

**Определение тупика.** Процедурные методы синхронизации более удобны в применении, чем методы синхронизации на основе

разделяемых переменных. Однако, при некорректном их использовании возможно возникновение еще одного типа ошибок синхронизации, известного как проблема тупиков (deadlock).

Группа процессов находится в тупиковой ситуации, если каждый процесс из группы ожидает событие, которое может вызвать только другой процесс из этой же группы.

**Условия возникновения тупика.** В большинстве случаев событием, которого ждет каждый процесс, является освобождение ресурса. В 1971 году Коффман с соавторами доказали, что для возникновения ситуации блокировки в системе процессов и ресурсов должны выполняться четыре условия.

1. Условие взаимного исключения. Каждый ресурс в любой момент отдан ровно одному процессу или доступен.

2. Условие удержания и ожидания. Процессы, удерживающие полученные ранее ресурсы, могут запрашивать новые.

3. Условие отсутствия принудительной выгрузки. У процесса нельзя забрать ранее полученный ресурс. Процесс, владеющий ресурсом, должен сам освободить его.

4. Условие циклического ожидания. Должна существовать круговая последовательность из двух и более процессов, каждый из которых ждет доступа к ресурсу, удерживаемому следующим членом последовательности.

Для возникновения блокировки должны выполняться все четыре условия. Если хотя бы одно условие отсутствует, блокировка невозможна.

**Моделирование блокировок.** В 1972 году Холт предложил метод моделирования условий возникновения тупика, используя ориентированные графы. Графические обозначения диаграмм Холта (рис. 8). Графы имеют два типа узлов. Кругом обозначают процесс, квадратом – ресурс. Ребро, направленное от ресурса (квадрат) к процессу (круг) обозначает, что ресурс был ранее запрошен процессом, получен и в данный момент используется. Ребро,

направленное от процесса к ресурсу, означает, что процесс в данный момент блокирован и находится в состоянии ожидания доступа к данному ресурсу.

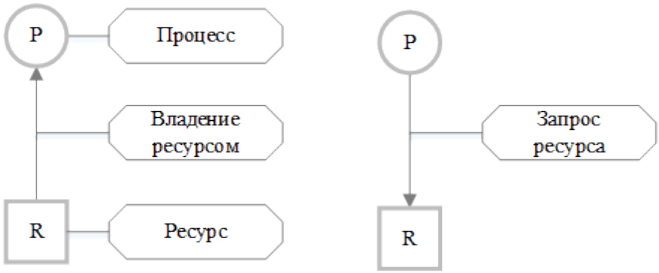



Рис. 8. Условные обозначения на диаграммах Холта

Простейший тупик в системе из двух процессов и двух ресурсов представлен графом Холта (рис. 9).

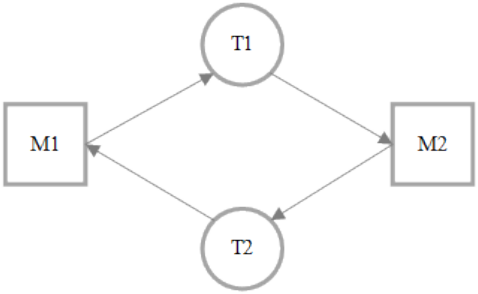



Рис. 9. Простейший тупик

Рассмотрим, каким образом может возникнуть показанная на рис. 9 ситуация. Пусть имеются функции потоков T1 и T2, код которых показан ниже. В качестве ресурсов выступают мьютексы M1 и M2. В начальный момент нет запросов и захваченных ресурсов.

```
DWORD T1(PVOID){        DWORD T2(PVOID){
(1)WaitforSingleObject(M1,-1);
(2)                     WaitforSingleObject(M2,-1);
(3)                     WaitforSingleObject(M1,-1);
(4)WaitforSingleObject(M2,-1);
(5)-------------тупик----------------------------
//критическая секция    //критическая секция
ReleaseMutex(M1);       ReleaseMutex(M1);
ReleaseMutex(M2);       ReleaseMutex(M2);
}                       }
```

В момент времени (1) поток T1 выполняет захват мьютекса M1. Так как мьютекс свободен, происходит успешный захват. Это показано ребром M1→T1. Далее в момент (2) планировщик может передать управление потоку T2, который выполняет захват свободного мьютекса M2. На графе (рис. 9) появляется ребро M2→T2. Продолжая вычисления, поток T2 выполняет запрос мьютекса M1, но так как мьютекс M1 уже заблокирован, поток переходит в состояние «ожидание» на мьютексе M1. На графе это показано ребром T2→M1. В момент (4) управление передается потоку T1, который выполняет попытку захватить мьютекс M2, уже принадлежащий потоку T2, и тоже переходит в состояние ожидания. Цикл на графе Холта, обозначающий состояние тупика, замыкается ребром T1→M2.

Рассматривая данный простейший сценарий возникновения тупика, можно сделать следующие выводы. Во-первых, тупики возникают не при каждом исполнении. Например, если бы на шаге (2) планировщик не выполнил переключение контекста на поток T2, а дал возможность потоку T1 выполнить захват мьютекса M2, тупик бы не возник. Во-вторых, аккуратным планированием можно избежать блокировок. То есть, планировщик, обладая некоторой информацией о необходимых потокам ресурсах, мог бы исключить переключение контекста, вызвавшее блокировку. В-третьих, блоки-

ровки можно обнаружить. При переходе процесса в состояние ожидания при запросе ресурса, можно выполнить анализ графа ресурсов и процессов на наличие цикла, то есть тупика.

**Стратегии борьбы с блокировками.** Рассмотрим стратегии предотвращения блокировок, вытекающие их моделей Холта и Коффмана.

*«Страусовый алгоритм».* Допущение, что блокировки в системе можно игнорировать. Усилия, затраченные на преодоление блокировок, могут себя не оправдать. Например, область памяти, в которой исполняется ядро операционной системы, ограничена. Также имеют предел таблицы процессов, таблицы открытых файлов и таблицы других системных ресурсов. Допустим, в операционной системе можно открыть n файлов, и каждый из N запущенных процессов открыл по n/N файлов. Далее, если каждый из процессов будет периодически повторять попытки открыть еще по одному файлу, возникнет тупик. Большая часть операционных систем, включая UNIX и Windows, игнорируют эту проблему. Полное ее решение включало бы неприемлемые с инженерной точки зрения ограничения на использование ресурсов процессами.

*Обнаружение и восстановление.* Данная стратегия состоит в том, чтобы дать возможность произойти блокировке, обнаружить ее и предпринять действия для ее исправления. Существует несколько способов восстановления из тупика.

Принудительная выгрузка заключается в том, чтобы взять ресурс у одного процесса, отдать другому процессу и затем возвратить назад. Такая возможность зависит от свойств ресурса. Например, в случае необходимости можно временно прервать процесс печати и предоставить принтер другому процессу.

При восстановлении через откат процесс периодически сохраняет свое состояние, создавая контрольные точки. Это означает, что его состояние записывается в файл, и впоследствии процесс может

быть возобновлен из этого файла. Чтобы выйти из тупика процесс, занимающий необходимый ресурс, откатывает к тому моменту времени, перед которым он получил данный ресурс. Освободившийся ресурс отдается другому процессу, попавшему в тупик.

Простейшим способом выхода из блокировки является уничтожение одного или нескольких процессов, находящихся в цикле взаимоблокировки. В некоторых случаях имеются процессы без побочных эффектов, которые можно перезапустить, не нарушая работу системы. Например, процедуру компиляции всегда можно повторить заново.

Рассмотрим простейший случай обнаружения тупика, когда каждый тип ресурсов системы представлен единственным ресурсом. Например, такая система могла бы иметь один сканер, одно устройство записи дисков, один принтер и так далее.

Обнаружение тупика в случае единственности ресурса каждого типа состоит из двух этапов: в текущем состоянии необходимо построить граф Холта, далее найти на нем цикл по какому-либо алгоритму.

Рассмотрим пример системы из семи процессов, состояние которых описывает таблица:

| Процесс | Захватил ресурс | Ожидает ресурс |
|---|---|---|
| A | R | S |
| B | - | T |
| C | - | S |
| D | U | S, T |
| E | T | V |
| F | W | S |
| G | V | U |

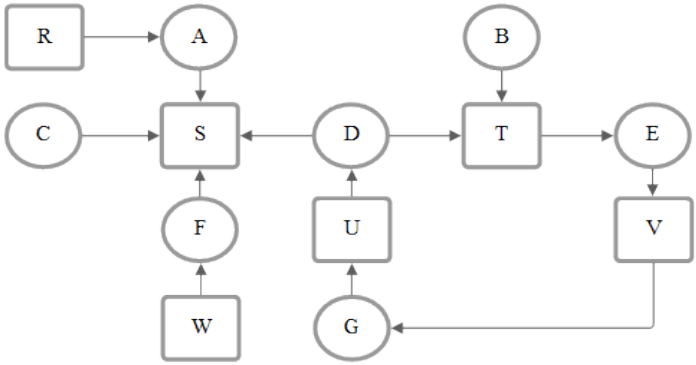



Рис. 10. Граф Холта, показывающий состояние тупика процессов D,V и G

Граф Холта, построенный по таблице, показан на рис. 10. По визуальной модели состояния системы процессов видно, что процессы D, E, G (вместе с ними и процесс B) находятся в тупике. Процессы A, C и F могут завершить исполнение и высвободить занимаемые ресурсы, поскольку любому из них можно предоставить ресурс S.

Для анализа графа большей размерности может применяться следующий алгоритм. Используем список L для хранения посещенных узлов и пометки для пройденных ребер. Алгоритм применяется для каждого узла графа.

Шаг 1. Начальные условия: имеется текущий узел, список L пустой, ребра не маркированы.

Шаг 2. Текущий узел добавляем в конец списка L. Если текущий узел присутствует в списке два раза, то граф содержит цикл (записанный в L). Алгоритм завершается.

Шаг 3. Если из текущего узла выходит немаркированное ребро, переходим к шагу 4, иначе переходим к шагу 5.

Шаг 4. Выбираем любое немаркированное ребро, маркируем его, по нему переходим к новому текущему узлу. Переходим к шагу 2.

Шаг 5. Удаляем последний узел из списка. Предпоследний узел объявляем текущим. Возвращаемся к шагу 3. Если перед шагом 5 в списке был только начальный узел, то граф не содержит циклов. Алгоритм завершается.

Теперь рассмотрим общий случай обнаружения тупика, если имеется один или несколько типов ресурсов, представленных не одним, а сразу несколькими ресурсами.

В системе имеется m типов ресурсов. $E = (e_1, e_2, \ldots, e_m)$ – вектор существующих ресурсов, в котором $e_j$- количество ресурсов типа $j$ $(1 \le j \le m)$. A = $(a_1, a_2, \ldots, a_m)$ – вектор доступных ресурсов. В системе имеется $n$ процессов. Матрица текущего распределения ресурсов $C$ имеет вид:

$$C = \begin{pmatrix} c_{11} c_{12} \ldots\ldots\ldots\ldots\ldots c_{1m} \\ c_{21} \ldots\ldots\ldots\ldots\ldots\ldots c_{2m} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ c_{n1} \ldots\ldots\ldots\ldots\ldots\ldots c_{nm} \end{pmatrix}$$, где $c_{ij}$ – количество ресурсов типа $j$, полученных процессом $i$.

Матрица запросов ресурсов $R$ имеет вид:

$$R = \begin{pmatrix} r_{11} r_{12} \ldots\ldots\ldots\ldots\ldots r_{1m} \\ r_{21} \ldots\ldots\ldots\ldots\ldots\ldots r_{2m} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ r_{n1} \ldots\ldots\ldots\ldots\ldots\ldots r_{nm} \end{pmatrix}$$, где $r_{ij}$- количество ресурсов типа $j$, запрашиваемых процессом $i$.

С учетом условия взаимного исключения для ресурсов выполняется соотношение $\sum_{i=1}^{n} c_{ij} + a_j = e_j$.

Алгоритм обнаружения тупиков состоит из следующих шагов.

Все процессы не маркированы.

Шаг 1. В матрице *R* ищем строку, соответствующую немаркированному процессу, которая меньше либо равна *A* ($r_j \le a_j$).

Шаг 2. Если такая строка *i* найдена, то маркируем соответствующий процесс, прибавляем строку номером *i* матрицы *C* к вектору *A*, возвращаемся к шагу 1.

Шаг 3. Если немаркированных процессов нет, то алгоритм завершается.

Если, после завершения алгоритма остаются немаркированные процессы, то они находятся в тупике.

*Пример*. Требуется определить, находится ли система процесс-ресурс в тупике.

*E*=(4, 2, 3, 1) *A*=(2, 1, 0, 0)

$$C = \begin{pmatrix} 0010 \\ 2001 \\ 0120 \end{pmatrix} \quad R = \begin{pmatrix} 2001 \\ 1010 \\ 2100 \end{pmatrix}$$

Убедимся в выполнении условия взаимного исключения для ресурсов: сложим строки матрицы *C* и вектор *A*, получим вектор *E*.

$A^1 = (2,2,2,0)$ *маркируем* $3-$*й процесс*

$A^2 = (4,2,2,1)$ *маркируем* $2-$*й процесс*

$A^3 = (4,2,3,1)$ *маркируем* $1-$*й процесс*

Выполняя последовательность шагов алгоритма, маркируем все процессы. Следовательно, нет процессов, находящихся в тупике.

*Избегание взаимных блокировок.* В первом примере показано, что, выбирая определенный порядок планирования среди нескольких возможных, планировщик избегает тупика. Стратегию поведения планировщика в случае двух процессов и произвольного количества ресурсов (в каждом типе имеется 1 ресурс) представляет диаграмма траектории ресурсов (рис. 11).

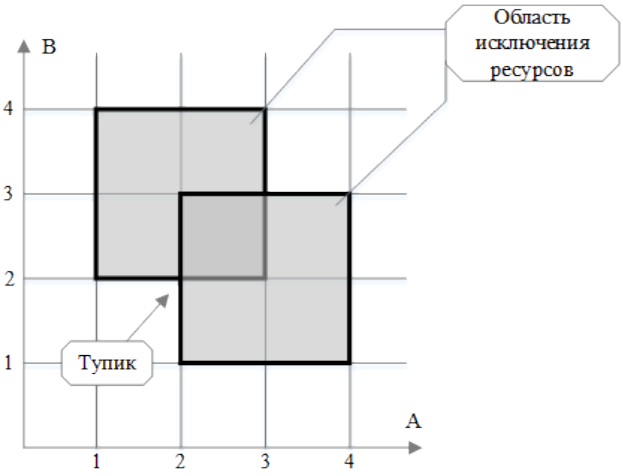



Рис. 11. Диаграмма траектории ресурсов

Диаграмма представляет собой первый квадрант координатной плоскости. Оси являются временными, представляя события процесса A и процесса B. Для определенности свяжем отсчеты времени процесса A со следующими событиями: 1 – захват принтера, 2 – захват плоттера, 3 – освобождение принтера, 4 – освобождение плоттера. На временной оси процесса B отмечаются следующие события: 1 – захват плоттера, 2 – захват принтера, 3 – освобождение плоттера, 4 – освобождение принтера.

С учетом введенных обозначений, точка на диаграмме траектории ресурсов представляет состояние системы. Она может дви-

гаться только вверх и вправо. В некоторых областях диаграммы система не может находиться по правилу исключения для ресурса. Области называются, соответственно, области исключения ресурса (рис. 11).

Если во время планирования точка, обозначающая состояние системы, попадет в область X, то со временем в системе возникнет блокировка. Так как точка не может двигаться вниз и влево. Любое состояние в области серого цвета является безопасным состоянием. В таком состоянии планировщик путем аккуратного планирования ресурсов может избежать возникновения тупика (рис. 11).

*Алгоритм банкира для одного типа ресурсов.* Анализ диаграммы траектории ресурсов показал, что цель алгоритма планирования - проверить приводит ли удовлетворение запроса на ресурс к выходу из безопасного состояния системы.

Пусть имеется один ресурс некоторого типа. Воспользуемся аналогией работы планировщика с поведением банкира, кредитующего клиентов. Запишем состояние системы в виде таблицы распределения ресурсов, в которой A, B, C – имена клиентов банка (процессы); второй столбец – выданные клиенту кредиты (ресурсы); третий столбец – число кредитов, необходимых клиентам. Внизу подписано число свободных кредитов у банкира.

Вначале проверим, является ли текущее состояние безопасным. То есть, сможем ли мы предложить стратегию обслуживания клиентов, удовлетворяющую все запросы на кредиты. Для этого будем давать ссуду только тем клиентам, запросы которых в кредитах мы можем покрыть полностью при текущем количестве свободных кредитов у банка. Ниже показана такая стратегия обслуживания.

| | | |
|---|---|---|
| A | 3 | 9 |
| B | 2 | 4 |
| C | 2 | 7 |
| Свободно 3 | | |

→

| | | |
|---|---|---|
| A | 3 | 9 |
| B | 4 | 4 |
| C | 2 | 7 |
| Свободно 1 | | |

→

| | | |
|---|---|---|
| A | 3 | 9 |
| B | 0 | - |
| C | 2 | 7 |
| Свободно 5 | | |

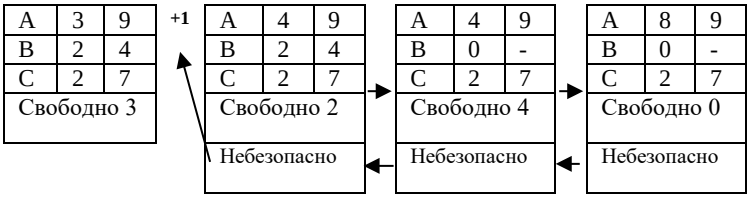
→

| | | |
|---|---|---|
| A | 3 | 9 |
| B | 0 | - |
| C | 0 | - |
| Свободно 7 | | |

→

| | | |
|---|---|---|
| A | 0 | - |
| B | 0 | - |
| C | 0 | - |
| Свободно 10 | | |

Теперь допустим, что в рассмотренном безопасном состоянии клиент А выставил запрос на один кредит. Рассмотрим гипотетическое состояние, возникающее при удовлетворении такого запроса.




Мы видим, что возникает состояние, в котором не хватает одного кредита для удовлетворения запроса клиента А (показано выше), так и клиента С (определяется аналогично). Оказавшись в таком состоянии, банкир признает банкротство, так как вынужден отказать всем не обслуженным клиентам. То есть система оказывается в тупике, следовательно, запрос процессом А одного ресурса приводит к выходу из безопасного состояния и должен быть отклонен.

*Предотвращение при помощи устранения одного из условий возникновения тупика.* Если удастся предложить стратегию работы с ресурсами, в которой исключено хотя бы одно из условий Хоффмана, тупик не возникнет.

1. Атака взаимного исключения. Пример использования очереди печати показывает, что можно избежать захвата принтера.

2. Атака условия удержания и ожидания. Если все ресурсы запрашиваются одновременно, то тупиков не возникнет. Например, можно использовать функцию WaitForMultipleObjects().

3. Атака условия отсутствия принудительной выгрузки. Реализуется при обработке транзакций. Каждый процесс готов освободить захваченные ресурсы по запросу менеджера транзакций.

4. Атака условия циклического ожидания. Если все ресурсы пронумерованы, и каждый процесс имеет право захватывать ресурс только с большим порядковым номером, чем номера уже захваченных ресурсов, то тупика не возникнет.

### 2.4. Вопросы

1. Состояние состязания. Пример возникновения и способ преодоления.
2. Средства синхронизации в режиме пользователя в ОС Windows. Функции, реализующие атомарные операции, объект «критическая секция».
3. Задача о критической секции. Алгоритм Питерсона для двух процессов. Условия задачи. Объяснение принципа работы алгоритма.
4. Предотвращение агрессивной оптимизации кода с использованием модификатора volatile. Эффект голодания, пример возникновения.
5. Эффект ложного разделения переменных. Пример влияния кэш-линий на скорость исполнения многопоточных программ.
6. Управление объектами ядра в ОС Windows. Описатель объекта. Таблица описателей объектов процесса. Создание, наследование, именование, дублирование описателей.
7. Средства синхронизации в режиме ядра в ОС Windows. События, семафоры, мьютексы.
8. Эффект взаимоблокировки или возникновения тупика. Определение, условия возникновения, моделирование графами Холта.
9. Стратегия «обнаружение-устранение» для борьбы с взаимоблокировками. Применение графов Холта и матриц распределения ресурсов.
10. Стратегия избегания блокировок. Диаграмма траектории ресурсов. Алгоритм банкира для одного вида ресурсов.
11. Предотвращение блокировок путем исключения условий их возникновения.

# 3. УПРАВЛЕНИЕ ПАМЯТЬЮ

## 3.1. Методы управления памятью

Типы адресов и их преобразований. Классификация методов управления памятью. Методы управления памятью, использующие разделы. Страничный, сегментный и сегментно-страничный методы управления виртуальной памятью. Механизм свопинга. Принцип работы кэш-памяти.

В программе можно выделить три типа адресов: символьные имена, виртуальные адреса и физические адреса (рис. 12). Символьные имена являются идентификаторами переменных в программе. Виртуальные адреса – это условные адреса, вырабатываемые транслятором исходного кода в объектный код. В простейшем случае транслятор может выполнять преобразование символьных имен непосредственно в физические адреса. Физические адреса – это номера ячеек в памяти. Данные номера выставляются центральным процессором на адресную шину при доступе к оперативной памяти.

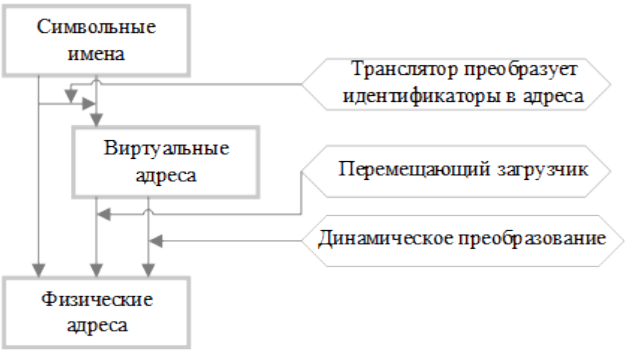



Рис. 12. Типы адресов и их преобразований

Преобразование виртуального адреса в физический адрес может выполняться двумя способами. При загрузке программы в память виртуальные адреса отображаются с использованием перемещающего загрузчика. Его работа состоит из загрузки программы в последовательные ячейки, начиная с некоторого базового адреса, и настройки смещений внутри программы относительно этого адреса (рис. 13).

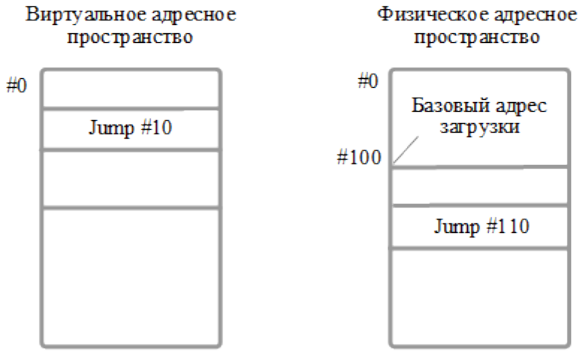



Рис. 13. Схема работы перемещающего загрузчика

Динамическое преобразование происходит при каждом обращении по виртуальному адресу в темпе обращений. Для такого преобразования используется аппаратура компьютера и операционная система.

**Методы управления памятью.** Существует два основных подхода к реализации процедур управления памятью: с использованием дискового пространства и без использования дискового пространства. Методы фиксированных разделов, динамических разделов и перемещаемых разделов не используют дисковое пространство. Страничный, сегментный и сегментно-страничный метод используют дисковую память. Специальными приемами управления памятью являются свопинг и кэширование.

Реализацию *метода фиксированных разделов* иллюстрирует рис. 14. Перед началом работы оператор разделяет физическую память на разделы заданного размера. Поступающие в систему задачи либо занимают свободный раздел подходящего размера, либо попадают в очередь. Очередь может быть общей для всех разделов или индивидуальной для каждого раздела.

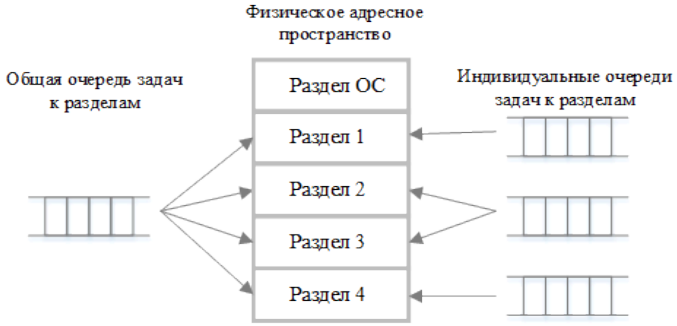



Рис. 14. Метод фиксированных разделов

Достоинством данного метода является простота реализации. Однако присутствует недостаточная гибкость. Например, уровень мультипрограммирования жестко ограничен числом выделенных разделов. В современных операционных системах пользователь также может вручную управлять разделами внутри виртуального адресного пространства процесса. Пример такого приема описан в п. 3.4.

*Динамические разделы.* При использовании динамических разделов распределение памяти по разделам заранее неизвестно. Операционная система ведет таблицы занятых и свободных разделов. При поступлении новой задачи для ее загрузки выбирается свобод-

ный раздел подходящего размера. Возможны разные стратегии выбора свободного раздела: первый по порядку подходящий, наименьший по размеру подходящий, наибольший по размеру подходящий раздел.

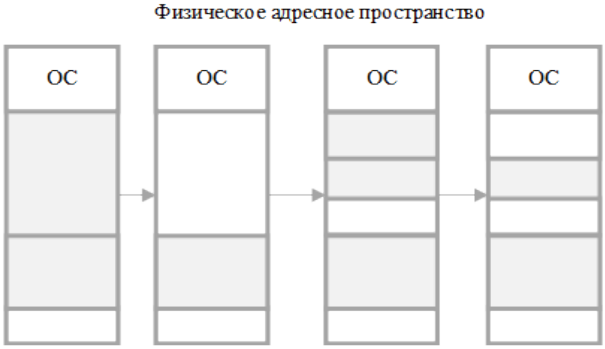



Рис. 15. Метод динамических разделов, возникновение фрагментации

Достоинства способа – это большая гибкость по сравнению с методом фиксированных разделов и отсутствие зависимости уровня мультипрограммирования от начального разбиения на разделы. Недостатком является эффект фрагментации памяти (рис. 15). Он заключается в появлении с течением времени большого числа небольших несмежных сегментов. Суммарный объем свободной памяти, содержащийся в таких сегментах, может быть большим, однако малый размер каждого отдельного сегмента не позволяет загрузить программу. Варьирование стратегий выбора свободного раздела может уменьшить фрагментацию. В современных операционных системах такой способ используется для управления кучей (динамической памятью) процесса.

*Перемещаемые разделы.* Данный способ расширяет управление динамическими разделами путем добавления процедуры сжатия: перемещения занятых разделов в одну последовательную об-

ласть старших или младших адресов. В результате свободная память размещается в последовательно расположенных ячейках памяти. Достоинством такой организации является отсутствие фрагментации памяти. Однако в отличие от предыдущих способов нельзя использовать перемещающий загрузчик. Процедура сжатия может быть затратной по времени. Поэтому обычно сжатие выполняется, когда не удается выполнить загрузку программы или во время простоя системы.

*Страничный способ.* Мотивом использования дискового адресного пространства является предоставление виртуальной памяти. Виртуальная память - это совокупность программно-аппаратных средств, позволяющих исполнять программы, размер которых превосходит доступную оперативную память компьютера. Для прикладного программиста механизм виртуальной памяти является прозрачным. То есть он не требует от него дополнительных усилий по управлению памятью.

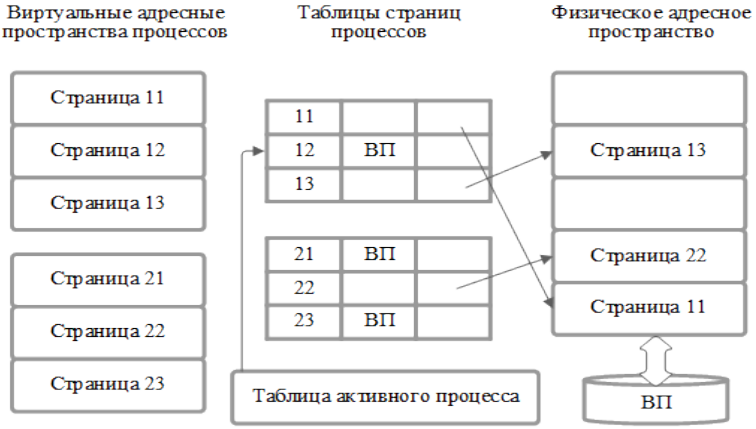



Рис. 16. Схема страничной организации памяти

При страничной организации памяти виртуальное пространство процессов делится на страницы равного размера. Размер страницы кратен степени двойки. Физическое адресное пространство делится на страницы такого же размера. Схема организации страничной памяти показана на рис. 16.

Страницы процесса при таком разбиении размещаются в физическом адресном пространстве произвольным образом: необязательно по непрерывным адресам. Часть адресного пространства процесса может быть выгружена во внешнюю память.

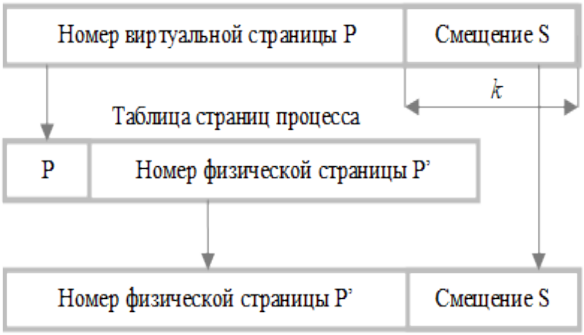



Рис. 17. Схема преобразования адреса при страничной организации памяти

Обращение к памяти выполняется по следующей схеме. В случае если виртуальная страница присутствует в физической памяти, старшие биты виртуального адреса (P) содержат номер виртуальной страницы. По таблице страниц устанавливается соответствующий адрес физической страницы. Он конкатенируется с младшими k разрядами виртуального адреса. Получается физический адрес. Схема преобразования виртуального адреса в физический адрес показана на рис. 17.

В случае если виртуальный адрес отсутствует в физической памяти (клетка ВП, содержимое располагается во внешней памяти)

происходит страничное прерывание. Процесс переходит к ожиданию на системном объекте ядра, запускается процедура загрузки страниц, а контекст переключается на следующий готовый процесс.

Для загрузки страниц определяется место в физической памяти. Если оно есть, то выполняется загрузка, коррекция таблицы страниц, перевод процесса в готовое состояние. В противном случае, выбирается страница для вытеснения. Если страница была модифицирована, то предварительно необходимо записать страницу на диск, иначе содержание страницы переписывается загружаемой страницей. Кандидаты на выгрузку определяются управляющей информацией операционной системы.

*Сегментный способ.* При страничном распределении памяти адресное пространство делится механически на страницы равного размера. При сегментном распределении памяти размеры сегментов произвольны, сегменты снабжены атрибутами, определяющими способ их использования. Разделение программы на сегменты определяется программистом или компилятором. Атрибуты сегмента определяют способ его использования: для исполнения, чтения, записи. Также сегментное распределение позволяет разделять сегменты между процессами.

В таблице сегментов указывается базовый адрес сегмента в физической памяти, а не номер, как в случае страничного способа. Размер сегмента не фиксирован, в отличие от страничного способа адресации. Он хранится в таблице сегментов и используется для контроля выхода за границы сегмента. Из-за разницы в размерах сегментов, как и в случае с динамическими разделами, для сегментного способа управления памятью характерно явление фрагментации памяти.

*Сегментно-страничный способ.* Цель сегментно-страничной организации памяти: решить проблему фрагментации физической памяти, сохраняя преимущество сегментного распределения, которое заключается в осмысленном распределении памяти.

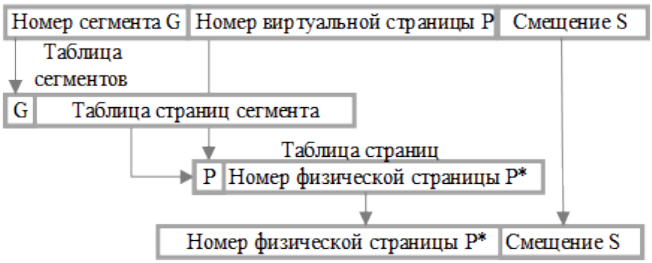



Рис. 18. Схема преобразования адреса при сегментно-страничной организации памяти

При сегментно-страничной организации виртуальный адрес состоит из 3 частей: номера сегмента g, номера страницы в сегменте p и смещения внутри страницы S. Номер сегмента преобразуется с использованием таблицы сегментов, а номер страницы – с использованием таблицы страниц (рис. 18). В результате линейный физический адрес состоит из 2 частей: номера физической страницы n и смещения S.

**Свопинг.** Виртуализация памяти в ранних операционных системах осуществлялась на основе свопинга. Свопинг (swapping) – способ управления памятью, при котором образы процессов выгружаются на диск и возвращаются в оперативную память целиком. Свопинг представляет собой частный, более простой в реализации, вариант виртуальной памяти, позволяющий совместно использовать оперативную память и диск.

Цель применения свопинга в ранних многозадачных операционных системах иллюстрирует рис. 19. Если на компьютере одновременно выполняется большое число вычислительных задач и задач с интенсивным вводом/выводом, то удается добиться высокой эффективности использования центрального процессора. Это происходит за счет того, что в такой конфигурации на время выполнения ввода/вывода одной задачи процессор можно переключить на

другую задачу. Так как положение участка насыщения на графике эффективности (рис. 19) обычно находилось за пределами числа задач, которое можно было разместить в оперативной памяти, ожидающие завершения ввода/вывода задачи было целесообразно сбрасывать на диск.

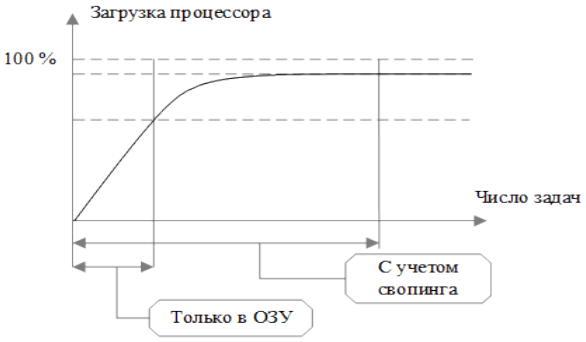



Рис. 19. Повышение загрузки процессора при использовании свопинга

**Кэш-память** – способ организации совместной работы устройств хранения, различающихся стоимостью хранения единицы информации и скоростью доступа рис. 20.


Рис. 20. К определению кэш-памяти

Цель применения кэш-памяти: увеличить скорость доступа и понизить стоимость хранения за счет размещения часто используемой информации в более быстром запоминающем устройстве. Механизм кэша, аналогично страничному и сегментному способам управления памятью, прозрачен для пользователя.

Принцип работы кэша оперативной памяти заключается в следующем (рис. 21). Кэш одновременно с памятью «прослушивает» запрос центрального процессора, но если данные есть в самом кэше, ответить успевает быстрее. Хранение информации в кэш-памяти организуется по ассоциативному принципу. Ключом для доступа (тегом) является адрес нескольких последовательно расположенных ячеек. Последствия такой организации памяти для программиста: память разбивается на кэш-линии (несколько последовательных ячеек, адресуемых одним тегом). Это приводит к тому, что в некоторых случаях для оптимизации скорости работы программ необходимо выполнять выравнивание данных по границе кэш-линии. Такая необходимость возникает, например, в случае возникновения эффекта ложного разделения см. п. 2.1.

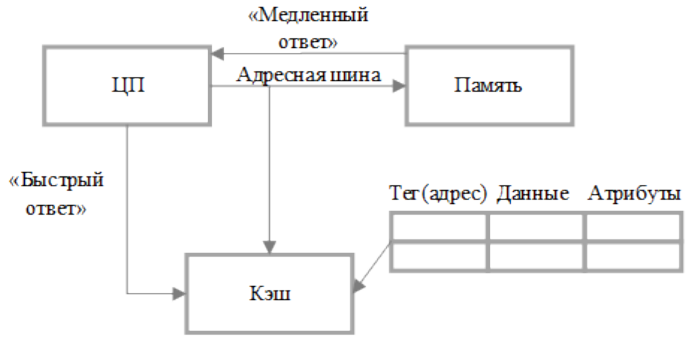



Рис. 21. Принцип работы кэш-памяти

Работа кэша основана на принципах пространственной и временной локальности, обеспечивающих высокий процент «попадания» в кэш p. Это приводит к снижению среднего времени доступа к памяти: $t_{доступа}=p\cdot t_{кэш} + (1-p)\cdot t_{памяти}$; p~90%.

Пространственная локальность - если в некоторый момент времени t происходит обращение по адресу A, то с большой долей вероятности в момент времени t+1 происходит обращение по адресу A±1.

Временная локальность – если в некоторый момент времени t произошло обращение по адресу A, то существует большая вероятность того, что в течение некоторого временного интервала (t, t+τ] снова произойдет обращение по адресу A.

### 3.2. Средства аппаратной поддержки управления памятью в архитектуре x86_32

Режимы работы микропроцессора x86_32. Сегментный механизм, структуры данных, схема преобразования виртуального адреса. Страничный механизм, структуры данных, схемы преобразования адреса. Средства вызова подпрограмм и задач.

**Режимы работы микропроцессора i386.** Общие принципы управления памятью с использованием внешней памяти представлены в п. 3.1. Рассмотрим конкретный пример управления памятью в 32 разрядной архитектуре микропроцессоров семейства Intel x86. Изложение ведется на примере микропроцессора i386. Старшие модели 32 разрядных процессоров Intel имеют сходную архитектуру.

Микропроцессор i386 имеет два режима работы: реальный (real mode) и защищенный (protected mode). В защищенном режиме доступны сегментный и страничный механизмы виртуальной памяти. В реальном режиме процессор работает как 16 разрядный процессор 8086, но с расширенным набором команд. Размер адресуемого

пространства в данном режиме составляет 1 МБ. Для работы в адресном пространстве 1МБ надо сформировать адрес определенным образом, как показано на рис. 22.

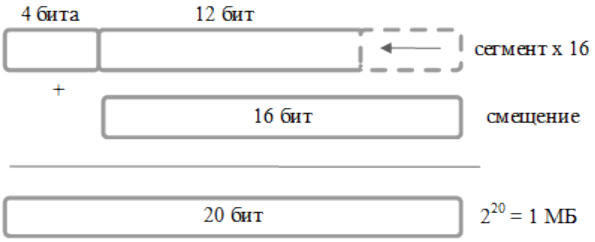



Рис. 22. Формирование адреса в реальном режиме

Реальный режим устанавливается по сбросу и используется для начальной инициализации системы. Путем записи управляющих битов в регистры командой MOV процессор переводится в защищенный режим. В защищенном режиме всегда включен сегментный механизм, и может быть установлен режим v86 (virtual 86), при котором процессор работает как несколько процессоров 8086 с общей памятью. Кроме того, в защищенном режиме может быть включен режим страничной адресации.

**Сегментный механизм.** В сегментном механизме управления памятью используются следующие структуры данных. Селектор – это структура данных, которая служит для выбора записи в глобальной или локальной дескрипторной таблице (рис. 23). Селекторы хранятся в сегментных регистрах и в некоторых типах записей внутри самих дескрипторных таблиц. Дескрипторные таблицы – это аналог таблицы сегментов (см. п. 3.1). Однако на практике, кроме описания сегментов, в них содержится дополнительная информация.

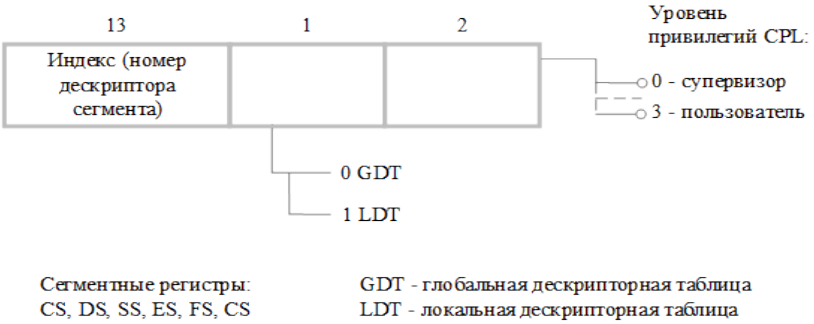



Рис. 23. Селектор

Регистр GDTR описывает положение глобальной дескрипторной таблицы в физической памяти (рис. 24). В нем также указывается размер таблицы в байтах. Поскольку в сегментном регистре на указание индекса дескриптора сегмента отводится 13 бит, всего сегментов $2^{13} = 8192$.

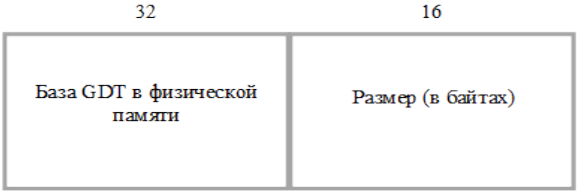



Рис. 24. Регистр GDTR

Формат дескриптора сегмента данных и кода показан на рис. 25. Размер дескриптора 8 байт; с учетом 8192 возможных сегментов, размер дескрипторной таблицы составит $8 \cdot 8192 = 2^{16} = 64$ КБ. Бит G определяет способ изменения размера сегмента (в байтах – G=0 или в страницах – G=1). Если G=0, то размер сегмента $2^{20} = 1$ МБ (совместимость с 8086, используется в режиме v86). Если G=1, то при 4КБ страницах размер сегмента равен $4КБ \cdot 2^{20} = 4$ ГБ.

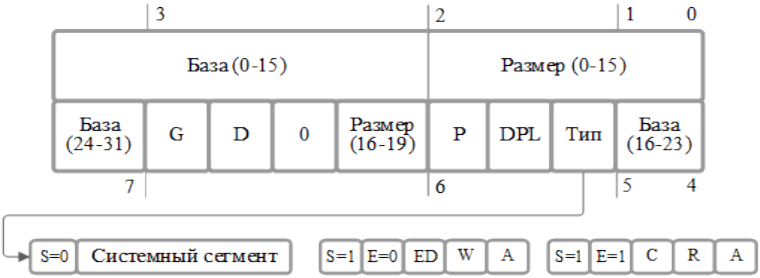



Рис. 25. Дескриптор сегмента данных и кода

Бит D определяет выравнивание сегментов по 32-битной (D=1) или 16-битной границе (D=0).

Бит P – бит присутствия сегмента в физической памяти. Если он не установлен, то сегмент выгружен на диск. Обращение к сегменту невозможно и приводит к прерыванию.

Биты DPL определяют уровень привилегий дескриптора, то есть возможность обращения к сегменту с данным уровнем привилегий задачи. Если CPL≤DPL – доступ разрешен, иначе возникает прерывание.

Тип сегментов определяет бит S. Если S=0, то данный дескриптор описывает системный сегмент или имеет специальное назначение. К системным сегментам, например, относятся LDT – локальная дескрипторная таблица, TSS – сегмент состояния задачи, в который отображается содержимое контекста при переключении задач. Дескрипторы специального назначения - ловушки (шлюзы) вызова, предназначенные для межсегментного вызова с повышением уровня привилегий.

Пользовательские сегменты имеют бит S=1. К пользовательским сегментам относятся сегменты данных E=0 и сегменты кода E=1. В пользовательских сегментах имеются следующие управляющие биты. Бит ED – бит распространения сегмента. Если ED=0 сегмент распространяется в сторону старших адресов, если ED=1 – в

сторону младших адресов. В сторону младших адресов распространяются сегменты стека. Бит W – бит разрешения записи в сегмент. Бит A – бит доступа к сегменту. Если A=1 произошел доступ к сегменту. Анализ этого бита позволяет оценить интенсивность обращений к сегменту для выбора наименее используемого сегмента с целью вытеснения его на диск. Бит C – бит подчинения. Если C=1, то проверка CPL≤DPL игнорируется, и можно вызвать более привилегированный сегмент кода, чем тот сегмент кода, из которого выполняется вызов. Наконец, бит R используется для указания на возможность чтения кодового сегмента.

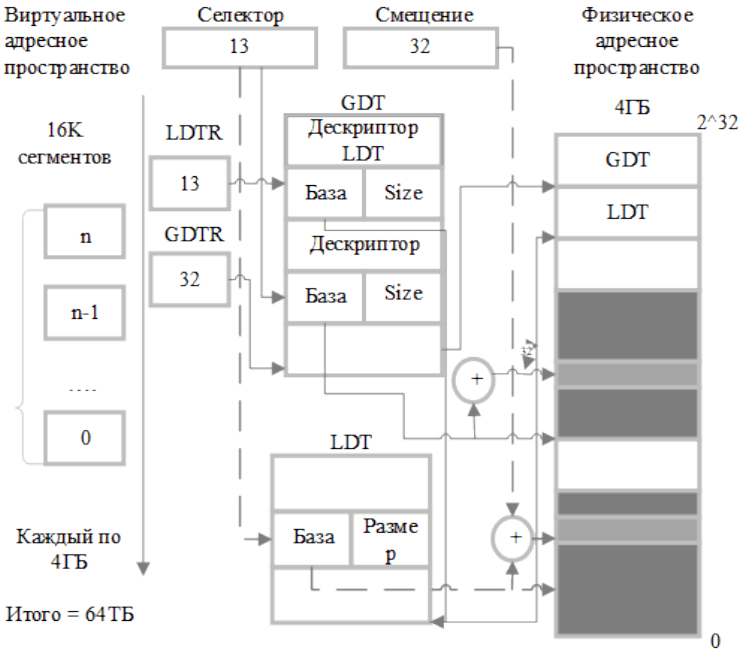



Рис. 26. Схема обращения к сегментам кода и данных

Рассмотрим схему обращения к сегментам кода и данных при использовании глобальных и локальных дескрипторных таблиц (рис. 26). Если используется адресация из таблицы GDT, то индекс сдвигается влево на 3 разряда, так как дескриптор имеет размер 8 Б = $2^3$. Получается смещение в таблице GDT. Далее проверяется, не выходит ли смещение за границы таблицы. Для этого анализируются младшие 16 разрядов GDTR, хранящих ее размер. Далее берется 32-битный адрес GDT и складывается с этим смещением. В итоге получается 32-битный адрес дескриптора в физической памяти. При этом определяется, присутствует ли сегмент в физической памяти и разрешен ли доступ к нему. Если доступ разрешен, то берется базовый адрес сегмента из дескриптора (база) и складывается со смещением из команды. Также на этой стадии производится контроль выхода за границу сегмента. Результатом является 32-битный адрес в физической памяти.

Если обращение идет через локальную дескрипторную таблицу LDT (см. соответствующий признак в селекторе), то используется регистр LDTR, который указывает на дескриптор LDT в глобальной дескрипторной таблице GDT. С использованием этого дескриптора вычисляется базовый адрес LDT в физической памяти. Далее преобразование происходит аналогичным способом.

Для увеличения скорости преобразования адреса каждому из шести сегментных регистров соответствует теневой 8-байтный регистр, хранящий дескриптор, соответствующий значению сегментного регистра.

Таким образом, для инициализации системы в защищенном режиме необходимо как минимум создать дескрипторную таблицу с одним входом и проинициализировать GDTR.

Если включен страничный механизм, то вычисленный 32-битный адрес подвергается дальнейшему преобразованию блоком управления страничной памятью.

**Страничный механизм.** Структура данных, используемая для страничного механизма управления памятью, – это дескриптор страниц (рис. 27).

| 20 | 3 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| Номер физической страницы | AVL | O | D | A | PCD | PWT | U | W | P |

Рис. 27. Дескриптор страниц

Общее число страниц, адресуемых через дескриптор страниц, составляет $2^{20}$ = 1 М. Часть линейного адреса, отводимая под смещение внутри страницы, составляет 32-20 = 12 бит. Размер страницы составляет $2^{12}$ = 4 К.

Поля в структуре дескриптора станиц имеют следующие назначения: AVL – резерв ОС; D – признак модификации; A – признак того, был ли доступ; PCD, PWT – биты управления кэшированием страницы; U – пользователь/супервизор; W – разрешение записи в страницу; P – бит присутствия страницы в физической памяти.

Так как дескриптор занимает в памяти 4 байта, а всего дескрипторов 1 М, для хранения таблицы страниц в физической памяти потребуется 4 МБ оперативной памяти. Хранение таблицы страниц целиком привело бы к нерациональному использованию физической памяти. Поэтому в архитектуре i386 используется двухуровневый механизм преобразования линейного виртуального адреса в физический. Схема такого преобразования адреса показана на рис. 28.

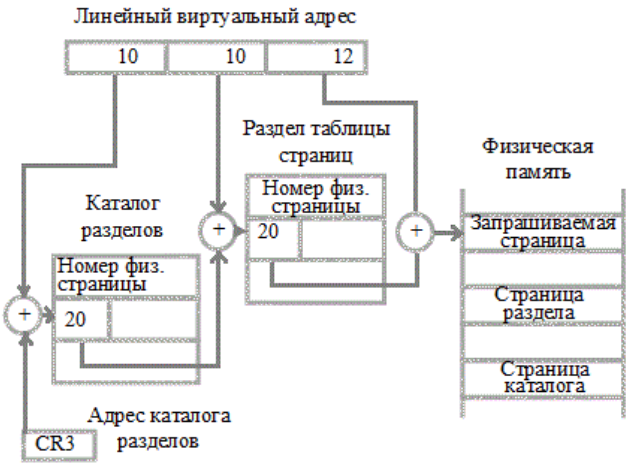



Рис. 28. Схема преобразования линейного виртуального адреса в физический

Идея механизма состоит в том, чтобы использовать страничный механизм для управления самой таблицей страниц. То есть, части таблицы страниц могут храниться во внешней памяти и подгружаться в оперативную память по мере необходимости.

Таблица страниц состоит из разделов и каталога разделов. Старшие 10 бит линейного виртуального адреса используются для адресации раздела внутри каталога разделов. Следующие 10 бит используются для адресации страницы внутри раздела. Таким образом, каталог разделов, так же, как и раздел, может включать в себя 1024 ($2^{10}$) дескриптора. Размер дескриптора равен 4 Б и каталог разделов целиком умещается в физическую страницу (4 КБ). Каталог разделов всегда присутствует в физической памяти, его адрес содержится в управляющем регистре CR3. Старшие 10 бит умножаются на 4 (т.к. дескриптор имеет размер 4 Б = $2^2$) и складываются с адресом в CR3, получается адрес дескриптора раздела в физической памяти. В нем содержится номер физической страницы раздела. С

его использованием вычисляется адрес дескриптора запрашиваемой страницы внутри раздела. Наконец, этот дескриптор содержит адрес запрашиваемой физической страницы.

Для ускорения описанного преобразования линейного адреса в блоке управления страницами кэшируется 20 последних комбинаций номеров виртуальной и физической страницы.

**Средства вызова подпрограмм и задач.** Существуют внутрисегментные и межсегментные вызовы, которые осуществляются командами CALL и JMP. Внутрисегментные вызовы не имеют особенностей. Различаются межсегментные вызовы с использованием дескриптора сегмента кода, шлюза вызова и вызова с переключением задачи через TSS (сегмент состояния задачи). Схема непосредственного вызова через дескриптор кода показана на рис. 29.

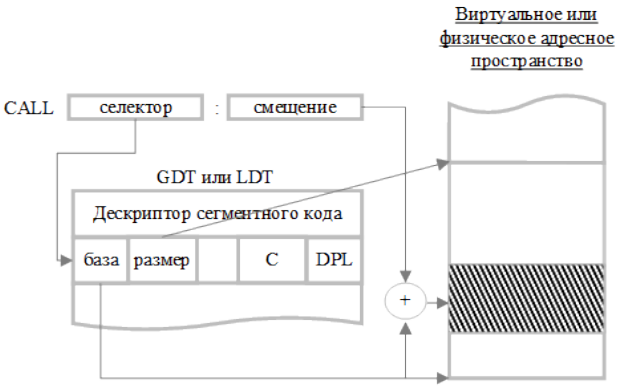



Рис. 29. Вызов через дескриптор кода

При вызове может указываться дескриптор сегмента кода. В этом случае возможны подчиненный и неподчиненный вызовы. При подчиненном вызове (C=1) можно вызвать сегмент кода с более высоким уровнем привилегий, чем CPL, при этом текущий уровень привилегий не изменяется. При неподчиненном вызове (C=0) вызов

осуществляется только в случае CPL≤DPL. Недостаток этого механизма в том, что он не обеспечивает защиту операционной системы: можно задать произвольную точку входа в кодовый сегмент операционной системы; отсутствует защита ее стека вызовов.

Защищенный вызов с повышением уровня привилегий обеспечивает шлюз вызовов (рис. 30).

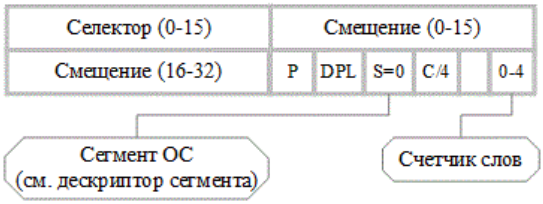



Рис. 30. Формат дескриптора шлюза

На рис. 30 используются следующие обозначения. Селектор – селектор сегмента кода, в котором находится вызываемая процедура или TSS. Смещение – точка входа в процедуру. Счетчик слов показывает, сколько слов требуется скопировать из стека текущего уровня привилегий в стек уровня привилегий вызываемой подпрограммы. Вызов разрешен, если CPL≤DPL, но DPL сегмента, на который указывает селектор, может быть любым. При удачном вызове процедура выполняется именно с указанным в требуемом сегменте уровнем привилегий DPL. Схема вызова через шлюз вызова показана на рис. 31.

В команде CALL также может быть непосредственно указан селектор сегмента TSS, хранящий полное состояние задачи, на которую выполняется переключение. Состояние текущей задачи запоминается в сегменте TSS, на который указывает регистр TR.

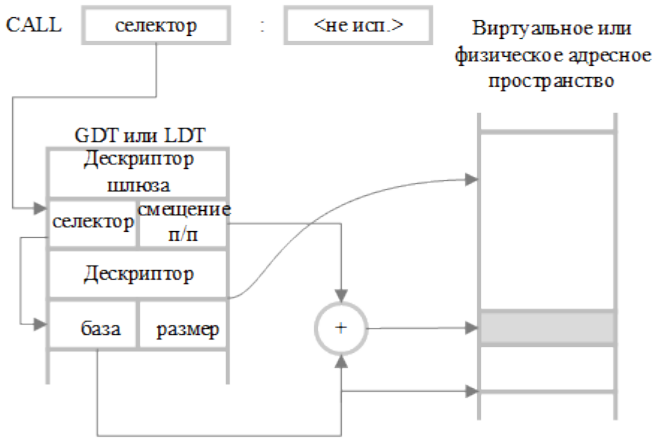



Рис. 31. Схема вызова через шлюз

### 3.3. Обзор алгоритмов замещения страниц

Оптимальный алгоритм; простые алгоритмы замещения, основанные на статистике использования страниц: алгоритм NRU, алгоритм FIFO, алгоритм «вторая попытка», алгоритм «часы»; алгоритмы выгрузки страницы, не использовавшейся дольше всех LRU: аппаратные реализации LRU, алгоритм NFU, алгоритм старения; алгоритмы, использующие понятие «рабочий набор».

Алгоритмы замещения страниц преследуют цель сделать осмысленный выбор удаляемой страницы для повышения производительности работы операционной системы. Когда происходит страничное прерывание и в памяти не оказывается свободного места (свободного страничного блока) для размещения запрашиваемой страницы, требуется освободить место для её загрузки, удалив некоторую страницу из памяти. Хотя каждый раз для удаления можно выбирать случайную страницу, производительность си-

стемы заметно повысится, если удалить редко используемую страницу. Если же удалить страницу, обращение к которой происходят часто, велика вероятность, что вскоре потребуется вернуть её в память, и возникнут дополнительные издержки.

Проблемы, похожие на проблему замещения страниц при управлении страничной памятью, возникают и в смежных областях. Например, сходные проблемы возникают при проектировании аппаратных кэшей процессоров или кэшировании web-страниц в прокси- серверах.

**Оптимальный алгоритм**. Допустим, что нам известно поведение программы в любой момент времени в будущем, то есть в произвольный момент исполнения программы нам известна будущая последовательность выполняемых программой команд и обращений к памяти. Если так, то в любой момент времени для любой страницы *P* программы мы можем определить число команд *K*(*P*), после выполнения которых наступит обращение к данной странице. Очевидно, что выгрузить следует страницу с наибольшим значением признака *K*(*P*).

Данный алгоритм иллюстрирует принцип выбора замещаемой страницы и имеет только теоретическое значение. Действительно, информацию, необходимую для работы оптимального алгоритма, можно получить лишь при повторном запуске программы, притом с теми же исходными данными. Этот алгоритм может использоваться для оценки эффективности работы реализуемых на практике алгоритмов.

**Алгоритм определения не использовавшейся в последнее время страницы (NRU)**.

Алгоритм NRU (Not Recently Used) предполагает наличие таймера и аппаратно управляемых статусных битов страницы *R* и *M*. Бит *R* (Referenced) устанавливается аппаратно всякий раз при обращении к странице. Бит *M* (Modified) устанавливается аппаратно, если обращение выполняется для записи в страницу. Те же статусные биты могут называться A (Access – признак доступа)

и D (Dirty – признак загрязнения). Заметим, что при отсутствии таких битов их можно смоделировать путем обработки прерываний по ошибке доступа к странице.

Алгоритм NRU работает следующим образом. Периодически по прерыванию от таймера сбрасывается в 0 значение бита *R*. В момент возникновения страничного прерывания все страницы, в зависимости от состояния битов *R* и *M,* относят к 4-м классам: класс 0 – *R*=0 и *M*=0; класс 1 – *R*=0 и *M*=1; класс 2: *R*=1 и *M*=0; класс 3 – *R*=1 и *M*=1. Для выгрузки выбирается произвольная страница из не пустого класса с минимальным номером.

Алгоритм предполагает, что лучше выгрузить изменённую страницу, к которой не было обращений в течении одного тика таймера (из класса 1), чем стереть часто используемую страницу (из класса 2). Достоинством алгоритма является лёгкость для понимания и реализации. Его производительность не оптимальна, но может оказаться достаточной для некоторых приложений.

**Алгоритм FIFO**. Это подход к проектированию класса алгоритмов с небольшими накладными расходами на функционирование, основанный на использовании очереди с дисциплиной обслуживания FIFO (First-In-First-Out). Операционная система поддерживает список всех страниц в списке FIFO. Страницы попадают в конец очереди при загрузке в память. При страничном прерывании для выгрузки берётся страница из головы очереди. Проблемой данной упрощённой реализации является то, что выгруженной может оказаться часто используемая страница.

**Алгоритм «вторая попытка»**. Простейшим усовершенствованием алгоритма FIFO, позволяющим избежать выгрузки часто используемой страницы, является использование бита обращения к странице *R*. Так же как и в алгоритме NRU, данный бит устанавливается аппаратно при доступе к странице и периодически сбрасывается по прерыванию от таймера. При страничном прерывании берётся страница из головы очереди. Однако она выгружается только в случае, если *R*=0. Если *R*=1, то

значение бита этой страницы устанавливается в *R*:=0, и она помещается в хвост очереди. Для анализа извлекается следующая страница из головы очереди. Алгоритм получил своё название «вторая попытка» (second chance) потому, что если все страницы имеют установленный бит *R*=1, то происходит «вторая попытка» выгрузки первой страницы по исходному алгоритму FIFO.

**Алгоритм «часы»**. Можно заметить, что алгоритм «вторая попытка» эффективнее реализуется на основе кольцевого списка с указателем, перемещающимся по элементам данного списка. Такую структуру можно мысленно представить в виде циферблата часов с движущейся стрелкой. Когда происходит страничное прерывание, проверяется та страница, на которую указывает стрелка воображаемых часов. Если требуется выгрузка (*R*=0), то новая страница замещает выгружаемую страницу в памяти (и позиции списка, указываемой стрелкой), а сама стрелка перемещается на следующую запись в списке. На этом обработка страничного прерывания завершается. Если выгрузка страницы не требуется, то выполняется модификация бита (*R:*=0); стрелка перемещается на следующую запись в списке и снова выполняется проверка условия выгрузки *R*=0.

**Дольше всех не использовавшаяся страница (LRU)**. В основе LRU (the Least Recently Used) аппроксимации оптимального алгоритма лежит статистическая закономерность, верная для большинства программ: страницы, к которым были многократные обращения в нескольких последних командах, также часто будут использоваться в последующих командах. Эта идея приводит к следующему реализуемому алгоритму: когда происходит страничное прерывание, выгружается страница, которая не использовалась дольше всего.

Для эффективной реализации алгоритма LRU требуется аппаратная поддержка в архитектуре компьютера. Например, можно использовать аппаратный счетчик, который в начале исполнения программы равен нулю и автоматически возрастает после каждой

команды. Каждая запись в таблице страниц должна содержать специальное поле для хранения значения счетчика. После каждого обращения к памяти значение счетчика записывается в поле страницы, к которой произошло обращение. Если произошло страничное прерывание, операционная система проверяет все значения счетчиков, сохранённые в таблице страниц. Страница с наименьшим значением поля счетчика удаляется, если нет свободных страничных блоков для размещения в памяти запрашиваемой страницы.

Ещё один вариант оборудования LRU – это специальное устройство, которое хранит и управляет изменением значений битовой матрицы размера $N \times N$, где $N$ – количество страничных блоков в компьютере. Всякий раз, когда происходит обращение к страничному блоку $k$, вначале производится заполнение всех элементов строки $k$ матрицы единицами, а затем заполнение всех элементов столбца $k$ матрицы нулями. Оказывается, что строка матрицы, имеющая наименьшее значение (если её представить как двоичную запись числа), будет соответствовать наиболее давно использовавшейся станице памяти, как показано на рисунке рис. 32.


Рис. 32. Работа алгоритма LRU, использующего матрицу

**Алгоритм NFU**. Ввиду редкости аппаратной поддержки метода LRU интерес представляют его программные реализации, использующие стандартную аппаратуру большинства компьютеров. Одна из разновидностей программных реализаций LRU называется алгоритмом NFU (Not Frequently Used – редко использовавшаяся страница). Для него необходим программный счетчик, связанный с каждой страницей в памяти, изначально равный нулю. Во время прерывания от таймера операционная система исследует все записи в таблице страниц. Бит $R$ каждой страницы прибавляется к значению счетчика страницы, затем сбрасывается в ноль. При страничном прерывании для замещения выбирается страница с наименьшим значением счетчика.

Проблемой алгоритма является то, что он не забывает значения счетчика обращений с течением времени. Работа многих программ состоит из фаз. Например, такие фазы присутствуют в многопроходном компиляторе. После выполнения первой фазы страницы кода этой фазы будут иметь большие значения счетчика обращений. Во второй фазе страницы с кодом первой фазы использоваться не будут, но будут иметь низкую вероятность выгрузки, провоцируя выгрузку страниц с кодом второй фазы с малыми значениями счетчика обращений.

**Алгоритм старения**. Необходимые изменения алгоритма NFU при обработке прерываний от таймера состоят из двух частей. Во-первых, каждый счетчик вначале сдвигается вправо на один разряд. Во-вторых, значение бита $R$ не прибавляется, а записывается в крайний левый бит счетчика. Работа видоизменённого алгоритма, известного под названием «старение» (aging), показана на рис. 33.

| R | 101011 | 110010 | 110101 | 100010 | 011000 |
|---|---|---|---|---|---|
| № страницы | | | | | |
| 0 | 10000000 | 11000000 | 11100000 | 11110000 | 01111000 |
| 1 | 00000000 | 10000000 | 11000000 | 01100000 | 10110000 |
| 2 | 10000000 | 01000000 | 00100000 | 00010000 | 10001000 |
| 3 | 00000000 | 00000000 | 10000000 | 01000000 | 00100000 |
| 4 | 10000000 | 11000000 | 01100000 | 10110000 | 01011000 |
| 5 | 10000000 | 01000000 | 10100000 | 01010000 | 00101000 |
| | *а)* | *б)* | *в)* | *г)* | *д)* |

Рис. 33. Работа алгоритма старения после пяти тиков таймера от (а) до (д)

Когда происходит страничное прерывание, удаляется страница с наименьшим значением счетчика. Например, счетчик страницы, к которой не было обращений 4 тика, будет иметь 4 нуля справа. То есть он будет иметь меньшее значение, чем счетчик страницы, к которой не было обращений 3 тика таймера с 3 нулями справа.

Алгоритм может принять неправильное решение о времени последнего использования страницы в двух случаях. Рассмотрим момент времени *д)* на рисунке рис. 33. К страницам 3 и 5 не было обращений 2 тика таймера подряд. Согласно алгоритму старения, для вытеснения следует выбрать страницу 3 с меньшим значением счётчика. Однако мы не знаем к третьей или пятой странице 3 тика назад произошло обращение позже, ввиду недостаточности разрешения таймера. Согласно правилу LRU, для вытеснения следует выбрать страницу 5, если к ней 3 тика назад произошло обращение позже, что противоречит алгоритму старения.

Другой случай возникает, когда у двух или более счетчиков значения равны нулю. Алгоритм старения выбирает произвольную страницу, хотя время последнего доступа у таких страниц может значительно отличаться. На практике, при использовании времени тика 20 мс и 8 битов счетчика отсутствие обращений в течении 160 мс означает, что с большой вероятностью страница не важна, и её

можно выгрузить без потери эффективности.

**Понятие «рабочий набор»**. Множество страниц, которое процесс использует в данный момент, называется рабочим набором. Если рабочий набор целиком находится в оперативной памяти, то процесс не вызывает страничных прерываний, пока выполнение не перейдёт к следующей фазе (например, к следующему проходу компиляции), и рабочий набор не изменится. Знать состав рабочего набора процесса важно в момент загрузки процесса для того, чтобы сразу загрузить все страницы рабочего набора, выполнив опережающую подкачку (prepaging). Если этого не делать и положиться на загрузку по запросу (demand paging), то вначале работы процесса эффективность выполнения окажется низкой из-за частых страничных прерываний. Поэтому актуально отслеживать состав рабочего набора. Из приведённых рассуждений также следует, что при возникновении страничного прерывания нужно в первую очередь выгружать из памяти страницы, не входящие в рабочий набор.

Для использования в алгоритме замещения страниц требуется точное определение рабочего набора. Формально рабочий набор в момент времени $t$ – это множество страниц $w(k,t)$, в которое входят все страницы, использовавшиеся за $k$ последних обращений к памяти. Обычно в качестве аппроксимации количества обращений к памяти удобно использовать время вычисления. При этом учитывается только время, в течение которого процесс выполнялся. Оно называется виртуальным временем процесса.

**Алгоритм WSClock**. В алгоритме WSClock используются значение текущего виртуального времени процесса, биты $R$ и $M$. Записи о страничных блоках процесса организованы в виде кольцевого списка с указателем. Определено также предельное время нахождения страницы в рабочем наборе $k$. Время последнего обращения к странице фиксируется обычным образом: по прерыванию от таймера проводится сканирование записей страничных блоков, у блоков с $R$=1 обновляется время последнего

использования текущим значением виртуального времени процесса, и бит *R* сбрасывается в ноль.

При возникновении страничного прерывания начинается обход кольцевого списка, начиная с текущей позиции указателя. Во время обхода выполняются следующие действия. Если *R*=1, то выполняется присваивание *R*:=0 и переход к следующей записи. Если *R*=0 и возраст страницы больше *k*, то страница не входит в рабочий набор и может быть замещена. Если у этой страницы *M*=0, то страница переписывается замещаемой страницей, и обработка страничного прерывания завершается. Иначе у этой страницы *M*=1, поэтому запускается асинхронный сброс содержимого страницы на диск, а сканирование кольцевого списка продолжается.

Если при обходе кольцевого списка происходит возврат к исходной позиции указателя, то возможны два состояния:

1) запланирована, по крайней мере, одна операция сброса страницы на диск;

2) не запланировано ни одной операции сброса.

В первом случае, так как запланированные операции сброса со временем завершатся и установят бит *M*:=0 у сбрасываемых на диск страниц, требуется продолжать сканирование. Сканирование, в конце концов, приведет к замещению страничного блока содержимым запрашиваемой страницы и выходу из процедуры обработки страничного прерывания. Во втором случае все страницы находятся в рабочем наборе. Если есть чистая страница (*M*=0), то для выгрузки выбирается она, иначе выбирается произвольная грязная страница (с *M*=1), и обработка страничного прерывания завершается.

Положение последней по порядку обхода чистой страницы должно отслеживаться в цикле обхода. Число планируемых асинхронных операций выгрузки грязных страниц ограничивают некоторым разумным пределом, по достижении которого планирование не происходит.

Рассмотрены основные алгоритмы замещения страниц,

имеющие теоретическое и практическое значение. Из приведённых алгоритмов, благодаря простой реализации и хорошей производительности, на практике чаще всего используются алгоритм старения и алгоритм WSClock.

### 3.4. Управление виртуальной памятью в ОС Windows

Структура виртуального адресного пространства процесса. Обзор методов управления памятью в ОС Windows. Применение функции VirtualAlloc. Обработка страничных прерываний с использованием структурированной обработки исключений. Файлы, отображаемые в память.

**Структура виртуального адресного пространства процесса.** Виртуальное адресное пространство процесса 32 разрядных версий Windows устроено следующим образом. В «десктопных» 32 разрядных версиях (например, Windows 7) пользователю выделяется младшие 2 ГБ адресного пространства (рис. 34).

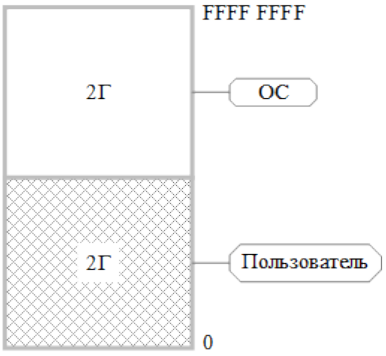



Рис. 34. Распределение памяти в системе Windows 7

Таким образом, в 32 разрядных операционных системах семейства Windows пользователь может выделить блок памяти, располагающийся по непрерывным адресам размером 2 ГБ, так как код и данные операционной системы помещаются (проецируются) в адресное пространство каждого процесса.

Средствами операционной системы поддерживаются несколько механизмов управления памятью. Простейшим является механизм динамической памяти. Процесс может иметь один или несколько разделов динамической памяти внутри пользовательской части виртуального адресного пространства (несколько куч процессов). К кучам может быть организован синхронизированный доступ из нескольких потоков. Кучи можно создавать динамически и удалять целиком. Они предназначены для создания большого количества относительно малых по размеру блоков памяти. Функции для работы с кучами имеют префикс Heap в своих названиях, например, HeapAlloc().

Существуют некоторые специфические ситуации в практике программирования, когда использование возможностей библиотеки времени исполнения и куч операционной системы оказывается недостаточным. Например, при обработке массивов данных большого размера, не умещающихся в оперативной памяти целиком, в численном моделировании, при обработке большеформатных изображений. В данном случае необходимо воспользоваться средствами управления виртуальной памятью. В Windows такое управление реализуется функцией VirtualAlloc() и другими вспомогательными функциями API. Совместно с функцией VirtualAlloc() обычно используется механизм структурированной обработки исключений.

Для управления большими объемами данных, организации межпроцессного взаимодействия через общую (разделяемую) память используется механизм отображения файлов на виртуальную память процессов. На данном механизме основана работа самой

операционной системы при загрузке исполняемых файлов и динамической компоновке библиотек.

Дополнительно 32 разрядным приложениям Windows доступен программный интерфейс Address Windowing Extensions (AWE), позволяющий получить доступ ко всему физическому адресному пространству компьютера (если его размер превышает 4 ГБ) через адресное окно в пределах пользовательских 2-3 ГБ в виртуальном адресном пространстве процесса. В современных 64 разрядных версиях операционных систем объем пользовательского адресного пространства, как правило, значительно превышает объем оперативной памяти, и в использовании механизма AWE не возникает необходимости.

В качестве примера работы с виртуальной памятью рассмотрим реализацию поиска простых чисел методом решета Эратосфена. В алгоритме используется массив флагов arr размером MAXNUM, где MAXNUM граница числового диапазона внутри которого ищутся все простые числа. Метод заключается в постепенном вычеркивании всех составных чисел установкой флагов arr[num]=1. Все не помеченные флагами числа будут являться простыми. Для нас в данном алгоритме представляет интерес организация сканирования массива размером около 1 ГБ.

Объявления и инициализация.

```
#include "stdafx.h"

DWORD i,j,num;
DWORD MAXNUM=1024*1024;
LPSTR arr;
DWORD dwPageSize;

VOID ErrorExit(LPTSTR);
INT PageFaultExceptionFilter(DWORD);

int main(intargc, char* argv[])
{
      BOOL bSuccess;
```

```
SYSTEM_INFO sSysInfo;

int num_MB=1;
printf("Amount of memory in MB:");
scanf("%d",&num_MB);
printf("Getting %d MB...\n",num_MB);

MAXNUM*=num_MB;

GetSystemInfo(&sSysInfo);
dwPageSize = sSysInfo.dwPageSize;

arr =(LPSTR)VirtualAlloc(NULL,MAXNUM,
            MEM_RESERVE,PAGE_NOACCESS);
if (arr == NULL) ErrorExit(
      "VirtualAlloc reserve    failed");
else printf("VirtualAlloc successful\n");
```

Для работы программы нам потребуется подключить некоторые заголовочные файлы, среди которых <windows.h>. На массив флагов указывает переменная arr, размер массива хранится в переменной MAXNUM. При ее инициализации пользователю программы предлагается ввести размер массива в мегабайтах. Также в начале работы вычисляется размер страницы с использованием функции GetSystemInfo() и выполняется резервирование памяти размером MAXNUM в произвольной области виртуальной памяти процесса под массив флагов.

Резервирование необходимо по следующей причине. Фактически запущенной программе разрешен доступ только к незначительной части адресов из ее пространства в 2 ГБ. Это области, где находится код программы, ее ресурсы, статические переменные, стеки и кучи. Обращения к остальной части адресов пространства пользователя вызовет прерывания. Страницы, соответствующие этим адресам, отсутствуют в оперативной памяти. Более того, операционная система автоматически не поддерживает структуры для управ-

ления такими страницами. Управление всем 2 ГБ адресным пространством у каждого запущенного процесса было бы крайне накладно с точки зрения используемых ресурсов.

С точки зрения системного программиста виртуальная память процесса разделена на страницы, которые могут находиться в трех состояниях.

*Свободная страница (free).* Доступ к таким страницам запрещен. Операционная система не поддерживает структуры для управления свободными страницами, при обращении к ним возникает страничное прерывание.

*Зарезервированная страница (reserve).* Операционная система подготавливает необходимые управляющие структуры для этих страниц, но физическая память не выделена. При обращении также возникает страничное прерывание.

*Выделенная страница (committed).* Страница присутствует в физической памяти, при обращении прерывание не происходит.

Последняя часть инициализирующего кода выполняет резервирование блока памяти из последовательно расположенных страниц по произвольному начальному адресу в виртуальном адресном пространстве процесса. Адрес начала блока присваивается переменной arr.

```
__try
    {
            for(i=0;i<MAXNUM;i++){num=i;arr[num]=0;}

            for(i=2;i<MAXNUM;i++)
            {
                  num=i;if(arr[num])continue;
                  printf("%d\n",i);

      for(j=i*2;j<MAXNUM;j+=i){num=j;arr[num]=1;}
            }
      }
```

```
    __except ( PageFaultExceptionFilter(
              GetExceptionCode() ) )
    {
            ExitProcess(GetLastError() );
    } bSuccess = VirtualFree(arr,0,MEM_RELEASE);

      printf ("Release was %s.\n", bSuccess ?
         "successful" : "unsuccessful" );

      return 0;
}
```

Далее сам алгоритм реализуется обычным способом, однако он помещается внутри блока структурированной обработки исключений. В блоке выполняется обращение к массиву arr, как будто память действительно выделена. Цель такой организации кода: перехватить прерывания, возникающие при обращении к странице памяти, принадлежащей массиву arr; вручную выполнить загрузку страницы в оперативную память; передать управление в точку возникновения прерывания.

Стандартная конструкция C++ для перехвата исключений catch(ExceptionType C) {…}. В отличие от нее, фильтр исключения __except(…) содержит выражение, вычисляемое в момент исключения. Значение этого выражения управляет передачей управления при возникновении исключения. Особенностью структурированной обработки является наличие семантики продолжения, которую мы используем в примере.

Еще одной особенностью фильтра исключений и тела обработчика исключений является то, что в них используются специальные встроенные псевдо-функции (intrinsic function), обрабатываемые непосредственно компилятором. В примере такой функцией является функция GetExceptionCode(), которая возвращает код исключения. Ее использование бессмысленно вне контекста исключения и вызывает ошибку компиляции.

Код обработчика исключений выглядит следующим образом.

```
INT PageFaultExceptionFilter(DWORD dwCode)
{
  LPVOID lpvResult;
  DWORD  comSize;
  if (dwCode != EXCEPTION_ACCESS_VIOLATION)
  {
   printf("Exception code = %d\n", dwCode);
   return EXCEPTION_EXECUTE_HANDLER;
  }

  //printf("Exception is a page fault\n");

  if(MAXNUM-num>dwPageSize)comSize=dwPageSize;
  else comSize=MAXNUM-num;

  lpvResult = VirtualAlloc(
   &arr[num],comSize,MEM_COMMIT,PAGE_READWRITE);

  if (lpvResult == NULL )
    {
       printf("VirtualAlloc failed\n");
       return EXCEPTION_EXECUTE_HANDLER;
    } else {
      //printf ("Allocating another page.\n");
    }

  return EXCEPTION_CONTINUE_EXECUTION;
}
```

Обработчик получает код исключения и возвращает значение, определяющее передачу управления после обработки. Мы используем следующие возвращаемые значения: EXCEPTION_EXECUTE_HANDLER для возврата управления в тело обработчика исключений в секции __except и аварийному завершению; EXCEPTION_CONTINUE_EXECUTION для возврата к месту возникновения страничного прерывания и повторной попытки обращения к памяти. Для передачи управления следующему обработчику в иерархии может использоваться значение EXCEPTION_CONTINUE_SEARCH.

Загрузку страницы в физическую память выполняет вызов VirtualAlloc с параметром MEM_COMMIT. Код загрузки имеет следующие особенности. Страница, в которой произошло исключение, вычисляется с использованием глобальной переменной num, выполняющей роль индекса по массиву arr. Доступ к массиву выполняется только с использованием данной переменной. Способ, не требующей глобальной переменной num, состоит в получении адреса, по которому возникло исключение при помощи еще одной встроенной псевдо-функции GetExceptionInformation().

Другой особенностью является коррекция запрашиваемого к загрузке размера памяти comSize. Она необходима при сканировании последней страницы массива, чтобы не выйти за пределы зарезервированной под массив arr области.

Наконец, после использования, аналогично кучам, виртуальная память требует очистки, которая выполняется вызовом VirtualFree(arr,0,MEM_RELEASE).

Кратко рассмотрим работу с файлами, отображаемыми в память процесса. Работа состоит из следующих этапов. Вначале нужно получить описатель файла, куда при работе механизма отображения будут сбрасываться страницы памяти. Данный этап можно пропустить, если используется файл подкачки.

Далее создается объект отображения при помощи вызова

```
hMapFile = CreateFileMapping(hFile,// описатель файла
NULL, // безопасность
PAGE_READWRITE, // желаемый доступ
0, 0, // размер объекта и файла
“name”); // имя объекта
```

Сконструированный объект ядра с описателем hMapFile может использоваться для организации взаимодействия процессов через разделяемую память. Для этого нужно передать описатель другому процессу рассмотренными ранее способами.

Далее выполняется собственно проецирование при помощи вызова вида

```
lpMapAddress = MapViewOfFile(hMapFile,
FILE_MAP_ALL_ACCESS, 0, 0, 0).
```

В приведенном примере отображается весь файл целиком. Работа с файлами большого размера осуществляется через адресное окно, о чем говорит название функции (map view of file). Функция возвращает адрес начала области памяти, в которую было выполнено отображение файла. Косвенное обращение по этому адресу, например

```
*((LPDWORD) pMapAddress) = 1000;
```

позволяет, как считывать, так и записывать информацию непосредственно в файл без использования функций ввода-вывода. Запись-чтение выполняется механизмом управления страничной памятью (пп. 3.1, 3.2).

После окончания работы с отображением выполняется очистка при помощи вызова функции UnMapViewOfFile() и закрытия описателей созданных объектов ядра вызовом CloseHandle().

### 3.5. Введение во взаимодействие с внешними устройствами

Типы внешних устройств. Способы программного взаимодействия с внешними устройствами: циклический опрос, прерывания, прямой доступ к памяти. Архитектура программного обеспечения ввода-вывода. Файловая система. Логическая организация файлов. Физическая организация файлов.

**Типы внешних устройств.** Устройства ввода-вывода можно разделить на два класса: блок-ориентированные, байт-ориентированные. Типичным блок-ориентированным устройством является диск. Обмен информацией с дисками и адресация информации вы-

полняется блоками. Для байт-ориентированных устройств характерна потоковая передача информации и отсутствие адресации. Клавиатура, манипулятор-мышь, сетевой адаптер представляют данную разновидность внешних устройств. Имеются внешние устройства, которые сложно отнести к какому-либо из перечисленных классов. Например, программируемый таймер и другие устройства, информирующие компьютер о наступлении внешних событий.

Какой программный механизм используется для передачи информации из внешних устройств в операционную систему? У внешних устройств имеются специальные регистры, в которые можно записывать и считывать значения.

Обращение к регистрам контроллера может выполняться двумя способами. Регистры внешних устройств могут отображаться на память компьютера. Программное взаимодействие при этом происходит с использованием команды MOVE и других аналогичных команд пересылки. Достоинством данного способа является удобство программирования. В языках программирования высокого уровня, в которых предусмотрены механизмы обращение к памяти (например, C, C++, Pascal), не требуется никаких специальных механизмов для взаимодействия внешними устройствами с регистрами, отображенными на память. Также этот способ взаимодействия подразумевает высокую скорость передачи данных. Типичный пример такого взаимодействия - видеокарты. Недостатком является необходимость усложнения аппаратуры компьютера по следующим причинам. Требуется согласование скорости с внешними устройствами, так как обычно устройства ввода-вывода обладают значительно меньшей пропускной способностью, чем оперативная память. Необходимо определять границы кэшируемых областей памяти, так как отображенные на внешние устройства участки памяти кэшировать нельзя. Возникающую неоднородность памяти необходимо поддерживать на уровне микросхем вспомогательной логики:

контроллеров-концентраторов памяти и контроллеров-концентраторов ввода-вывода (т.н. северный и южный мосты в архитектуре Intel). Кроме этого сужается диапазон адресуемой физической памяти.

Другой возможностью программного взаимодействия является использование специального пространства ввода-вывода, не пересекающегося с оперативной памятью. Доступ к регистрам внешних устройств в нем осуществляется специальными командами чтения IN и записи OUT регистров внешних устройств. В данном случае упрощается аппаратная организация взаимодействия с внешними устройствами и несколько усложняется программное взаимодействие. Необходимы ассемблерные вставки в код на языке высокого уровня, поддержка компилятором или специальные функции в библиотеке времени исполнения.

Процедура организации взаимодействия с внешними устройствами может организовываться тремя способами.

**Циклический опрос.** Команда на выполнение некоторой операции записывается во входной регистр внешнего устройства. Далее в цикле считывается и анализируется значение выходного регистра до тех пор, пока не будет считан признак завершения операции. Недостатком данного способа является неэффективное использование процессора и шины памяти (данных).

**Взаимодействие по прерыванию.** Команда на выполнение некоторой операции записывается во входной регистр внешнего устройства. Далее процессор может выполнять другие полезные действия одновременно с работой внешнего устройства или остановиться, при этом не блокировать системную шину. Когда операция внешнего устройства завершена, внешнее устройство самостоятельно извещает процессор, инициируя прерывание. В ответ на прерывание процессор запоминает адрес текущей команды и переходит к выполнению специальной процедуры обработки прерывания. В ней организуется считывание уже готовой информации из выходных регистров внешнего устройства. Далее происходит возврат управления по сохраненному адресу прерванной команды.

Этот способ является более эффективным с точки зрения использования аппаратных ресурсов. Однако он требует сложной программной реализации: специальной настройки таблицы векторов обработки прерывания; нелинейной организации потока управления в программе; некоторого механизма планирования действий процессора, чтобы исключить простой процессора в течение ввода-вывода.

**Взаимодействие с использованием прямого доступа к памяти.** В данном способе передача информации из внешнего устройства в оперативную память производится не с использованием центрального процессора, а напрямую специальным контроллером прямого доступа (DMA-контроллер). Контроллер самостоятельно реализует описанную во втором способе процедуру взаимодействия с внешним устройством, а по окончании передачи данных прерыванием извещает центральный процессор о завершении взаимодействия. Преимуществом способа является еще большая эффективность, особенно при передаче пакетов данных. Способ позволяет контроллеру обращаться к системной шине, пока сам процессор выполняет команду. Очевидно, что способ требует дополнительных усилий по программированию ввода-вывода.

**Архитектура программного обеспечения ввода-вывода.** Для автоматизации рутинных низкоуровневых процедур в операционных системах реализуется специальная подсистема ввода-вывода, примерная архитектура которой показана на рис. 35.

Рассмотрим компоненты, показанные на рисунке. Система обработки прерываний находится на нижнем уровне иерархии. Управление прерываниями выполняет сама операционная система. При возникновении прерываний происходит вызов драйверов внешних устройств. Драйверы выполняют обработку запросов с уровня пользователей или прерываний и имеют доступ к регистрам контроллеров внешних устройств. Имеется возможность синхронизации работы драйверов с использованием специальных мьютексов режима ядра.


Рис. 35. Архитектура подсистемы ввода-вывода

В операционных системах имеется независимый от устройств уровень «системы буферизации», типичными функциями которого являются обеспечение независимого размера блоков данных, защита, общий интерфейс к драйверам устройств и их именование, распределение/освобождение устройств, уведомление об ошибках. Этот уровень единообразно подготавливает блоки для передачи запросов драйверам от вышележащих слоев и получения ответа.

Обращение к функциям ввода-вывода, как и к другим сервисам операционной системы, осуществляется посредством системных вызовов. Иногда они задействуются непосредственно или, как в ОС Windows, через функции интерфейса прикладного программирования (OpenFile(), CreateFile(), WriteFile() ), служащего для унификации различных версий операционной системы.

Выше по иерархии располагается библиотека поддержки времени исполнения (RTL), тоже содержащая функции ввода-вывода (printf(), scanf() и т.д.). Функции библиотеки RTL используют функ-

ции вызовов интерфейсов прикладного программирования API, часто являясь просто их обертками. В C++ функции потокового вывода <<, >> основаны на printf(), scanf(). Зная о такой организации ввода-вывода в некоторых языках (C, C++), можно сократить размер исполняемого файла, работая напрямую с вызовами API операционной системы. Для этого можно использовать функции ввода-вывода API, не компонуя соответствующие функции из RTL.

**Файловая система.** Важнейшими типами внешних устройств являются разнообразные накопители данных. Необходимые удобные абстракции для работы с ними реализуются файловыми системами.

Файловая система – часть операционной системы, назначение которой – обеспечить удобный интерфейс для работы с данными, хранящимися на дисках. Файлы – именованный набор данных. Файл может идентифицироваться по имени. Обычно файлы могут иметь одинаковые имена. Уникальность файла определяется составным именем, включающим символьные имена каталогов. Файл может идентифицироваться уникальным дескриптором. Если файловая система представлена в виде сети, файл может находиться в нескольких каталогах сразу.

Кроме обычных файлов с данными имеются файлы, ассоциированные с устройствами ввода-вывода. Это позволяет выполнять ввод-вывод посредством программного интерфейса доступа к файлам. Каталоги с файлами также являются специальным видом файлов. Каталоги файлов хранят атрибуты файла. Для организации защиты файлов от несанкционированного доступа, аналогично другим объектам операционной системы, используется избирательный или мандатный механизм безопасности.

Внешние устройства хранения имеют следующий простой интерфейс доступа к данным: по указанному адресу можно записать

или считать блок данных фиксированного размера (в несколько килобайт). Например, для доступа к блоку на диске требуется указать его физическое положение на поверхности носителя (номер поверхности, номер цилиндра (дорожки), номер сектора). Поэтому файловая система должна организовывать некоторый произвольный доступ, обеспечивая отличные от блочной модели способы работы с данными. Это логическая организация файлов. Также требуется поддерживать некоторую организацию физических блоков внутри файлов. Это физическая организация файлов.

**Логическая организация файлов.** Приведем некоторые распространенные модели файлов.

*Последовательность байт.* Это типовая модель доступа, используется как произвольный, так и последовательный доступ к хранимым данным.


Рис. 36. Последовательность блоков переменной длины

*Последовательность блоков переменной длины.* Эта модель (рис. 36) позволяет по логическому номеру блока адресовать его содержимое. В некоторых файловых системах используется гибридная модель, когда в файле содержится несколько независимых последовательностей байт (потоков данных). Это может оказаться удобным, например, для хранения видео, звука и субтитров в отдельных потоках средствами файловой системы.

*Упорядоченная или неупорядоченная последовательность ключей.* По ключу адресуется ассоциированный с ним блок данных.

Если последовательность ключей упорядочена, то имеется возможность выбрать следующий блок по порядку относительно текущего блока. Данная логическая организация удобна для управления и организации доступа к файлам очень больших размеров.

**Физическая организация файлов.** Рассмотрим некоторые распространенные приемы физической организации файлов.

*Последовательность блоков* (непрерывное размещение). Блоки, принадлежащие файлу, следуют по порядку их физических адресов. В каталоге указывается адрес первого блока, количество блоков или последний блок. Достоинство этого способа – простота, а основная проблема – фрагментация блоков. Этот способ, например, находит применение в файловых системах длительного хранения, в частности в ленточных запоминающих устройствах.

*Связанный список блоков.* Простейший способ преодолеть фрагментацию (рис. 37).

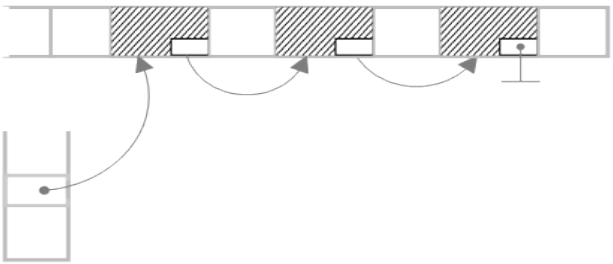


Рис. 37. Связанный список блоков

Проблемой является большое время доступа к информации в конце файла. Все блоки нужно последовательно перебрать. Информация о следующем блоке является частью блока, на это уходит некоторое количество памяти.

*Связанный список индексов.* Используемый на практике способ организации файловых систем относительно небольшого размера

(рис. 38). Он основан на применении таблицы размещения файла (File Allocation Table, FAT). Файловые системы, использующие этот способ: FAT12, FAT16, FAT32, VFAT. Метод предложен Microsoft.

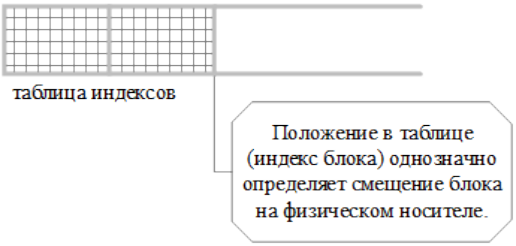



Рис. 38. Связанный список индексов

Идея заключается в переносе информации о следующем блоке в отдельную таблицу в начальных блоках физического носителя. Индекс блока в таблице и его физическое положение на диске связаны линейным преобразованием. Блоки таблицы индексов считываются в оперативную память. Благодаря этому можно быстро перебрать цепочку индексов блоков файла в памяти и запросить нужный блок на диске. Поэтому эффективность произвольного доступа оказывается высокой. Преимуществом также является целостность файловой системы, поддерживаемая несколькими копиями таблицы индексов и хранением информации о состоянии каждого физического блока в таблице индексов. Недостатком является рост размера требуемой служебной информации при увеличении размеров файловой системы (фактически предел 8 тебибайт при 2 КБ кластерах).

*Непосредственное перечисление блоков* и способы, использующие сбалансированные деревья (рис. 39).

Это наиболее распространенная группа способов физической организации файлов. Часть физических блоков в произвольной области дискового пространства отводится под ссылочные структуры

для доступа к блокам с данными. Например, на рисунке показана структура таких блоков. В корневом блоке, адрес которого записывается в каталоге, хранится 13 полей, содержащих адреса других блоков. Первые 10 полей непосредственно адресуют блоки файла. Если их недостаточно, то используется следующее поле, адресующее блок, в котором может быть записано 256 адресов следующих блоков файла. Если размер файла превышает 266 блоков, то используется двенадцатая запись корневого блока, с использованием которой физические блоки адресуются уже с двойным уровнем косвенности и так далее. Общее количество физических блоков в файле, таким образом, может достигать 16.843.018.

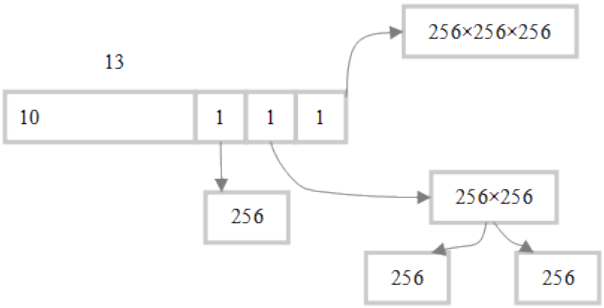



Рис. 39. Непосредственное перечисление блоков

В настоящее время разработано большое количество файловых систем различного назначения и устройства. Помимо логической и физической организации при рассмотрении архитектур файловых систем обсуждаются методы обеспечения надежности (например, журналирование), повышения скорости доступа (экстенты), хранения больших объемов данных (распределенные параллельные файловые системы), защиты данных (шифрованные файловые системы) и другие.

### 3.6. Вопросы

1. Методы управления памятью без использования внешней памяти. Фиксированные, динамические и перемещаемые разделы.
2. Методы управления памятью с использованием внешней памяти. Сегментный, страничный, сегментно-страничный способ.
3. Назначение, принцип работы механизма свопинга.
4. Назначение, принцип работы механизма кэширования.
5. Реализация сегментного механизма управления памятью в процессорах семейства x86_32.
6. Реализация страничного механизма управления памятью в процессорах семейства x86_32. Размер и основные поля структур данных, особенности реализации.
7. Принцип работы алгоритмов замещения страниц, оптимальный алгоритм. Простые аппроксимации оптимального алгоритма: алгоритм NRU, алгоритм FIFO, алгоритм «вторая попытка», алгоритм «часы».
8. Алгоритмы выгрузки дольше всех не использовавшейся страницы LRU: аппаратные реализации LRU, алгоритм NFU, алгоритм старения.
9. Понятие «рабочий набор», алгоритм WSClock.
10. Средства ОС Windows для управления виртуальной памятью процесса. Функция VirtualAlloc. Структурированная обработка исключений. Файлы, отображаемые в память.
11. Архитектура программного обеспечения ввода-вывода. Средства программного взаимодействия с внешними устройствами. Архитектура программного стека, функции слоев.
12. Общие принципы организации файловых систем и файлов. Идентификация файлов в файловых системах, логическая и физическая организация файлов.

———

## 4. ПРАКТИКА СИНХРОНИЗАЦИИ И УПРАВЛЕНИЕ ПАМЯТЬЮ

### 4.1. Работа с кучами процессов в ОС Windows

Цель выполнения лабораторной работы, перечень вариантов заданий, требования к решению, порядок сдачи.

Целью лабораторной работы является изучение методов работы с динамической памятью средствами программного интерфейса Windows API. Требуется реализовать код приложения согласно варианту задания, структурно обозначив в нем (например, с помощью функций) три части: код для выделения в динамической памяти массивов средствами Windows API; код для выполнения обработки содержимого массивов; код для освобождения динамической памяти.

Динамическая память (память в куче) должна выделяться при помощи функции HeapAlloc() и связанных с ней функций, а не путем использования функций или операторов стандартной библиотеки языков C/C++. Для выделения памяти может быть выбрана стратегия работы со стандартной кучей процесса или выделения памяти в специально созданной куче. Можно выделить память непосредственными обращениями к функциям HeapAlloc() или путем переопределения оператора new на языке C++.

Метод освобождения динамической памяти зависит от способа ее выделения и может заключаться в поочередном возврате блоков в кучу путем вызова HeepFree() или в удалении кучи вместе со всеми выделенными в ней блоками памяти.

Информацию о способах работы с кучей средствами Windows API следует самостоятельно изучить с использованием сайта технической документации Microsoft: https://learn.microsoft.com/ и по монографии Дж. Рихтера и К. Назара [12].

При демонстрации работы программы следует знать о перечисленных выше возможностях реализации кода, а также о том, когда

данные возможности уместно применять в реальной разработке. Код сданной программы включается в итоговый отчет по лабораторному практикуму. Отчет оформляется в электронном виде и включает в одном файле листинги всех лабораторных работ практикума. Листингу в отчете должна предшествовать формулировка задания. После листинга нужно привести результаты тестового запуска работы программы.

***Вариант №1.*** Найти наибольший и наименьший элементы в динамическом массиве размерностью N×M.

***Вариант №2.*** Необходимо каждый элемент строки разделить на сумму элементов строки в динамическом массиве размерностью N×M.

***Вариант №3.*** Необходимо каждый элемент строки разделить на наибольший элемент строки в динамическом массиве размерностью N×M.

***Вариант №4.*** По динамическому массиву из M вещественных чисел необходимо рассчитать выборочное среднее и выборочную дисперсию.

***Вариант №5.*** Динамический массив размерности M×N необходимо дополнить (M+1)-й строкой и (N+1)-м столбцом, в которых записать суммы элементов соответствующих строк и столбцов. В элементе M+1,N+1 должна храниться сумма всех элементов массива.

***Вариант №6.*** В динамическом массиве размерности M×N необходимо в каждой строке найти элемент с наименьшим значением, а затем среди этих чисел найти наибольшее. На экран вывести индексы этого элемента.

***Вариант №7.*** Транспонировать матрицу, размещенную в динамическом массиве размерности N×N, не используя дополнительного массива.

***Вариант №8.*** В динамическом массиве размерности M×N необходимо найти номер строки и номер столбца, в которых находится наименьший элемент.

***Вариант №9.*** Создать динамический массив из M строк по N символов каждая. Необходимо вывести только те строки, которые являются палиндромами, то есть читаются одинаково слева направо и справа налево.

***Вариант №10.*** Реализовать код для перемножения двух матриц, размещенных в динамических массивах размерности N.

## 4.2. Разработка приложений с раздельной компиляцией кода, файловый ввод-вывод

Цель выполнения лабораторной работы, перечень вариантов заданий, обзор используемых в вариантах заданий абстрактных типов данных, требования к решению, порядок сдачи.

Целью лабораторной работы является изучения подхода к разработке сложных приложений с раздельной компиляцией кода на языках C/C++, а также изучение программного интерфейса Windows API для работы с файлами.

В работе требуется реализовать в виде отдельной единицы компиляции набор функций и объявлений данных, класс или систему связанных классов, необходимых для работы с указанным в задании типом данных. В отдельном модуле пишется код для тестирования функций данного модуля.

Дополнительно к базовому функционалу типа данных требуется реализовать код для сохранения экземпляра (объекта) реализованного типа данных в файл целиком, а также код для восстановления состояния экземпляра (объекта) из данных, записанных в файл. Для этого разрешено использовать только функции WriteFile() / ReadFile() Windows API и связанные с ними функции. Рекомендуется выполнять сохранение содержимого объекта в двоичном формате. Работу с динамической памятью можно выполнять

как средствами Windows API, так и средствами стандартной библиотеки языков C/C++. Конкретные типы данных, помещаемых в указанные в заданиях контейнеры, можно выбирать произвольно.

Допускается выполнять реализацию в процедурном, объектно-ориентированном или объектно-ориентированном с элементами обобщенного программирования стилях. При реализации раздельной компиляции следует иметь ввиду, что код для методов класса, определенных внутри своего класса, генерируется только при создании объектов данного класса, поэтому тело метода следует вынести из объявления класса. Например, исходный код class A{public: int my_method(){return 42;}}; не вызывает генерации объектного кода для my_method(). В тоже время обработка компилятором кода class A{public: int my_method();}; int A:: my_method(){return 42;} приведет к генерации объектного кода для метода my_method() в текущей единице компиляции.

Также код для шаблонного класса не генерируется, пока не будет известен конкретный тип параметра шаблона. Подстановка конкретного типа в шаблон, приводящая к генерации кода, но не создающая объект, может выглядеть так: template class MyContainerType<ElementType>;.

Данные особенности следует учесть, так как они существенны при оформлении кода в виде библиотеки динамической компоновки в следующей лабораторной работе.

Код программы, согласно полученному варианту задания, показывается для проверки преподавателю. Далее на основе этого кода в следующей лабораторной работе разрабатывается приложение с элементами динамической компоновки. Поэтому включать код этой лабораторной работы в письменный электронный отчет не требуется.

В заданиях лабораторной работы рассматриваются следующие абстрактные типы данных.

***Ассоциативный массив (словарь)*** – абстрактный тип данных (интерфейс к хранилищу данных), позволяющий хранить пары вида (ключ, значение) и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу.

***Очередь*** – структура данных с дисциплиной доступа к элементам «первый пришёл - первый вышел». Добавление элемента возможно лишь в конец очереди, выборка - только из начала очереди, при этом выбранный элемент из очереди удаляется.

***Стек*** – структура данных, в которой доступ к элементам организован по принципу «последним пришёл - первым вышел». Добавление элемента возможно только в вершину стека. Удаление элемента тоже возможно только из вершины стека, при этом второй сверху элемент становится верхним.

***Дек*** – двусвязная (двухсторонняя) очередь, «очередь с двумя концами» - структура данных, в которой элементы можно добавлять и удалять как в начало, так и в конец.

***Связный список*** – структура данных, состоящая из узлов, каждый из которых содержит как собственные данные, так и одну или две ссылки на следующий и/или предыдущий узел списка.

***Хэш-таблица*** – структура данных, реализующая ассоциативный массив. При реализации по методу цепочек представляет собой массив указателей на списки. Индекс массива – это хэш-код ключа из пары (ключ, значение). Список, соответствующий элементу массива, состоит из элементов (ключ, значение), имеющих одинаковый хэш-код, равный значению индекса этого списка в массиве.

Ниже приведены варианты заданий, в которых задается тип структуры данных и способ его реализации для выполнения этой и последующих лабораторных работ

***Вариант №1.*** Требуется реализовать структуру данных «ассоциативный массив» используя динамический массив.

***Вариант №2.*** Требуется реализовать структуру данных «ассоциативный массив» используя связный список.

***Вариант №3.*** Требуется реализовать структуру данных «ассоциативный массив» используя бинарное дерево.

***Вариант №4.*** Требуется реализовать структуру данных «ассоциативный массив» используя хэш-таблицу, построенную по методу цепочек.

***Вариант №5.*** Реализовать очередь на основе связного списка.

***Вариант №6.*** Реализовать очередь на основе динамического массива.

***Вариант №7.*** Реализовать стек на основе связного списка.

***Вариант №8.*** Реализовать стек на основе динамического массива.

***Вариант №9.*** Реализовать дек на основе связного списка.

***Вариант №10.*** Реализовать дек на основе динамического массива.

### 4.3. Разработка библиотек динамической компоновки

Цель выполнения лабораторной работы, требования к решению, порядок сдачи и оформления лабораторной работы.

Целью лабораторной работы является изучение методов работы с динамически подключаемыми библиотеками в программном интерфейсе Windows API. Раздельно компилируемый код для работы с типом данных из лабораторной работы п.4.2 преобразуется в динамически компонуемую библиотеку DLL. Операции для работы с типом данных должны экспортироваться из DLL. Библиотека DLL должна быть снабжена соответствующим заголовочным файлом. Из основного модуля кода приложения с функцией main() динамическая библиотека должна подключаться в явном виде с использованием функций LoadLibrary() и GetProcAddress(), а также в неявном виде путем конфигурирования проекта.

При выполнении задания в MS Visual Studio рекомендуется вначале настроить решение (solution) двумя проектами (project): проект исполняемой программы и проект динамической библиотеки. Далее отладить неявную компоновку. Затем убедиться, что функции библиотеки вызываются явно, путем получения их адресов в библиотеке динамической компоновки.

Сдача лабораторной работы выполняется путем предъявления кода преподавателю и демонстрации работоспособности программы. Код программы вместе с заданием из п.4.3 и п.4.2, примером вывода программы включается в письменный электронный отчет.

### 4.4. Разработка многопоточных приложений

Цель выполнения лабораторной работы, требования к решению, порядок сдачи и оформления лабораторной работы.

Целью лабораторной работы является изучение методов написания многопоточных приложений и синхронизации потоков в программном интерфейсе Windows API. В библиотеку функций для работы со структурой данных, реализованную в заданиях п.4.3 и п.4.2, добавляется следующая функциональность. Вызов функции или метода класса для добавления элемента в структуру выполняется в одном потоке, а обработка вызова с действительным помещением элементов в нее – в другом потоке. Передача аргументов вызова осуществляется через буфер в памяти, доступ к которому синхронизируется. При каждом добавлении элемента в структуру данных происходит ее сохранение на диск целиком средствами, реализованными по заданию п.4.2. Тестирующая программа демонстрирует корректность записи элементов путем чтения файла на диске и печати его содержимого по окончании добавления.

Следует обратить внимание, что предлагаемая к реализации схема взаимодействия потоков реализуется в задаче об ограничен-

ном буфере (задаче поставщик-потребитель). Схема решения рассмотрена в п.2.2 настоящего пособия. Для простоты в решении можно использовать буфер единичного размера. В данном случае не требуется применение мьютекса, а семафоры-счетчики ресурсов заменяются событиями, сигнализирующими потоки о заполненности или пустоте буфера. Далее реализуйте корректное завершение дочернего потока, который в цикле извлекает данные из буфера и переносит их в структуру данных.

Сдача лабораторной проводится путем демонстрации ее работоспособности преподавателю. Допускается реализация задания как на основе лабораторной по заданию п.4.2, так и на основе кода лабораторной по заданию п.4.3. При оформлении отчета в электронном виде можно привести общий итоговый листинг работ п.4.2, п.4.3, п.4.4 соответственно объединив постановку задания и примеры выполнения.

### 4.5. Приложения с несколькими процессами

Цель выполнения лабораторной работы, требования к решению, порядок сдачи и оформления лабораторной работы.

В лабораторной работе выполняется изучение методов работы с процессами в программном интерфейсе Windows API. Задание выполняется по схеме и варианту задания п.4.4 за исключением того, что поток, осуществляющий фактическое добавление элементов в структуру данных, реализуется в дочернем процессе.

Для выполнения задания требуется изучить API взаимодействия процессов через файл, проецируемый в память, изучить и применить передачу описателя объекта ядра (HANDLE) в другой процесс, изучить и применить функцию запуска дочернего процесса CreateProcess().

Возможны следующие подходы к реализации задания. Запуск дочернего процесса может выполняться из отдельного образа: второго исполняемого файла с кодом структуры данных. Можно запустить дочерний процесс из того же образа (исполняемого файла), из которого запущен родительский процесс. В этом случае требуется реализовать код для определения первого или последующего запуска для исключения рекурсии. Также можно применить неявное управление файловыми проекциями при реализации буфера для взаимодействия дочернего и родительского процессов. Информацию о функциях Windows API и необходимых опциях компилятора можно найти на сайте Microsoft Learn: https://learn.microsoft.com/ и в монографии Д. Рихтера, К. Назара [12].

Отчет по заданию лабораторной работы производится путем демонстрации кода и работоспособности программы. Аналогично инструкциям к лабораторным работам по п. 4.3 и п. 4.4, при составлении итогового письменного отчета можно не дублировать код из предыдущих лабораторных работ.

———

## СПИСОК ЛИТЕРАТУРЫ

1. Гордеев, А.В. Операционные системы: учеб. для вузов по направлению подготовки бакалавров и магистров «Информатика и вычислительная техника» и направлению подготовки дипломированных специалистов «Информатика и вычислительная техника» / А. В. Гордеев. – 2-е изд. – СПб: Питер: 2009. – 416 с.

2. Олифер, В.Г. Сетевые операционные системы: учеб. пособие для вузов по направлению подготовки дипломированных специалистов «Информатика и вычислительная техника» / В. Г. Олифер, Н. А. Олифер. – 2-е изд. – СПб: Питер: Питер Пресс, 2009. – 669 с.

3. Столлингс, В. Операционные системы. Внутреннее устройство и принципы проектирования / В. Столлингс; пер. с англ. – 9-е изд. – М: Диалектика, 2020. – 1264 с.

4. Бэкон, Джин. Операционные системы. Параллельные и распределенные системы / Дж. Бэкон, Т. Харрис; пер. с англ. – СПб: Питер, 2004. – 799 с.

5. Таненбаум, Э. Современные операционные системы / Эндрю Таненбаум, Херберт Бос. – 4-е изд. – СПб: Питер: Питер Пресс, 2022. – 1120 с. – (Классика computer science).

6. Карпов, В.Е. Основы операционных систем: курс лекций: учебное пособие: для вузов по специальностям в области информационных технологий / В. Е. Карпов, К. А. Коньков; под ред. В. П. Иванникова; Интернет-университет информационных технологий. – М.: ИНТУИТ. РУ, 2005. – 536 с.

7. Дейтел, Х.М. Операционные системы: [в 2 т.] / Х.М. Дейтел, П.Д. Дейтел, Д.Р. Чофнес; под ред. С.М. Молявко; пер. с англ. – 3-е изд. – М.: Бином, 2016 – Т. 1: Основы и принципы. – 2016. – 1024 с.

8. Дейтел, Х.М. Операционные системы: [в 2 т.] / Х. М. Дейтел, П. Д. Дейтел, Д. Р. Чофнес; пер. с англ. ред. С. М. Молявко. –

3-е изд. – М.: Бином, 2016. – Т. 2: Распределенные системы, сети, безопасность. – 2016. – 704 с.

9. Таненбаум, Э. Операционные системы. Разработка и реализация / Э. С. Таненбаум, А. Вудхалл; пер. с англ. – 3-е изд. – СПб.; М.; Нижний Новгород: Питер, 2007. – 704 с.

10. Руссинович, М. Внутреннее устройство Windows / М. Руссинович, Д. Соломон, А. Ионеску; пер. с англ. – 7-е изд. – СПб.; М.; Нижний Новгород: Питер, 2022. – 944 с. – (Классика computer science).

11. Уорд, Б. Внутреннее устройство Linux / Б. Уорд; пер. с англ. – 3-е изд. – СПб.; М.; Нижний Новгород: Питер, 2023. – 480 с.

12. Рихтер, Дж. Windows via C/C++. Программирование на языке Visual C++ / Дж. Рихтер, К. Назар. – М.: Питер, Русская Редакция, 2009. – 896 с.

13. Керниган, Б.У. Язык программирования C / Б.У. Керниган, Д.М. Ритчи. – 2-е изд. – М.: Диалектика, 2020. – 288 с.

14. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт. – М.: ДМК Пресс, 2016. – 272 с.

Учебное издание

Востокин Сергей Владимирович

**УПРАВЛЕНИЕ ПРОЦЕССАМИ И ПАМЯТЬЮ В ОПЕРАЦИОННЫХ СИСТЕМАХ**

*Учебное пособие*

Редакционно-издательская обработка
издательства Самарского университета

Подписано в печать 18.04.2023. Формат 60×84 1/16.
Бумага офсетная. Печ. л. 7,5. Тираж 120 экз. (1-й з-д 1-27).
Заказ №  . Арт. – 13(Р1УП)/2023.

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА».
(САМАРСКИЙ УНИВЕРСИТЕТ)
443086, Самара, Московское шоссе, 34.

---

Издательство Самарского университета.
443086, Самара, Московское шоссе, 34.