МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Магнитогорский государственный технический университет им. Г.И.
Носова»

УТВЕРЖДАЮ
Директор ИЭиАС
_____ В.Р. Храмшин

04.02.2025 г.

**РАБОЧАЯ ПРОГРАММА ДИСЦИПЛИНЫ (МОДУЛЯ)**

# *ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ*

Направление подготовки (специальность)
09.03.01 Информатика и вычислительная техника

Направленность (профиль/специализация) программы
Программное обеспечение средств вычислительной техники и автоматизированных систем

Уровень высшего образования - бакалавриат

Форма обучения
очная

| | |
|---|---|
| Институт/ факультет | Институт энергетики и автоматизированных систем |
| Кафедра | Вычислительной техники и программирования |
| Курс | 4 |
| Семестр | 8 |

Магнитогорск
2025 год

Рабочая программа составлена на основе ФГОС ВО - бакалавриат по направлению подготовки 09.03.01 Информатика и вычислительная техника (приказ Минобрнауки России от 19.09.2017 г. № 929)

Рабочая программа рассмотрена и одобрена на заседании кафедры Вычислительной техники и программирования
03.02.2025 г, протокол № 5

Зав. кафедрой _____ О.С. Логунова

Рабочая программа одобрена методической комиссией ИЭиАС
04.02.2025 г. протокол № 3

Председатель _____ В.Р. Храмшин

Рабочая программа составлена:
доцент кафедры ВТиП, канд. техн. наук _____ Ю.В. Кочержинская

Рецензент:
директор НИИ П«ромбезопасность», д-р. техн. наук _____ М.Ю.Наркевич

**Лист актуализации рабочей программы**

---

Рабочая программа пересмотрена, обсуждена и одобрена для реализации в 2026 - 2027 учебном году на заседании кафедры Вычислительной техники и программирования

Протокол от __ _____ 20__ г. № __
Зав. кафедрой _____ О.С. Логунова

---

Рабочая программа пересмотрена, обсуждена и одобрена для реализации в 2027 - 2028 учебном году на заседании кафедры Вычислительной техники и программирования

Протокол от __ _____ 20__ г. № __
Зав. кафедрой _____ О.С. Логунова

---

Рабочая программа пересмотрена, обсуждена и одобрена для реализации в 2028 - 2029 учебном году на заседании кафедры Вычислительной техники и программирования

Протокол от __ _____ 20__ г. № __
Зав. кафедрой _____ О.С. Логунова

---

Рабочая программа пересмотрена, обсуждена и одобрена для реализации в 2029 - 2030 учебном году на заседании кафедры Вычислительной техники и программирования

Протокол от __ _____ 20__ г. № __
Зав. кафедрой _____ О.С. Логунова

**1 Цели освоения дисциплины (модуля)**

Целями освоения дисциплины «Теория вычислительных процессов» является ознакомление студентов с понятием, видами и моделями вычислительных процессов, методами их взаимодействия; изучение протоколов и интерфейсов работы с вычислительными процессами; овладение методами формального представления взаимодействия процессов при помощи сетей Петри; формирование навыков программной реализации алгоритмов синхронизации процессов.

Для достижения поставленной цели в курсе «Теория вычислительных процессов» решаются задачи:

- формирования понятий о вычислительном процессе, организации вычислительных процессов (параллельных, последовательных, альтернативных).
- изучения видов адресации и структуре адресного пространства в различных ОС.
- освоения алгоритмов диспетчеризации и методов планирования вычислений в мультипрограммных системах.

формирования навыков проектирования и реализации программ в мультипрограммных ОС.

**2 Место дисциплины (модуля) в структуре образовательной программы**

Дисциплина Теория вычислительных процессов входит в часть учебного плана формируемую участниками образовательных отношений образовательной программы.

Для изучения дисциплины необходимы знания (умения, владения), сформированные в результате изучения дисциплин/ практик:

Многопоточное программирование на языке Java

Операционные системы семейства *nix

Знания (умения, владения), полученные при изучении данной дисциплины будут необходимы для изучения дисциплин/практик:

Выполнение и защита выпускной квалификационной работы

Практические аспекты разработки компиляторов

Подготовка к сдаче и сдача государственного экзамена

Проектная деятельность

**3 Компетенции обучающегося, формируемые в результате освоения дисциплины (модуля) и планируемые результаты обучения**

В результате освоения дисциплины (модуля) «Теория вычислительных процессов» обучающийся должен обладать следующими компетенциями:

| Код индикатора | Индикатор достижения компетенции |
|---|---|
| ПК-6 Способность к формализации и алгоритмизации поставленных задач, к написанию программного кода с использованием языков программирования, определения и манипулирования данными и оформлению программного кода в соответствии установленными требованиями | |
| ПК-6.1 | Оценивает качество математической модели при формализации задачи предметной области |
| ПК-6.2 | Оценивает качество разработанных алгоритмов для последующего кодирования |
| ПК-6.3 | Оценивает выбор программных средств для программирования и манипулирования данными в соответствии установленными требованиями |

**4. Структура, объём и содержание дисциплины (модуля)**

Общая трудоемкость дисциплины составляет 4 зачетных единиц 144 акад. часов, в том числе:

– контактная работа – 59,5 акад. часов:
– аудиторная – 56 акад. часов;
– внеаудиторная – 3,5 акад. часов;
– самостоятельная работа – 48,8 акад. часов;
– в форме практической подготовки – 0 акад. час;
– подготовка к экзамену – 35,7 акад. час

Форма аттестации - экзамен

| Раздел/ тема дисциплины | Семестр | Аудиторная контактная работа (в акад. часах) | | | Самостоятельная работа студента | Вид самостоятельной работы | Форма текущего контроля успеваемости и промежуточной аттестации | Код компетенции |
|---|---|---|---|---|---|---|---|---|
| | | Лек. | лаб. зан. | практ. зан. | | | | |
| 1. Теория вычислений | | | | | | | | |
| 1.1 Определения вычислительного процесса, методы представления и формализации. | 8 | 2 | 6 | | 2 | 1. Самостоятельное изучение учебной и научной литературы. 2. Работа с электронным учебником, выполнение лабораторных работ | 1. Беседа – обсуждение 2. Проверка индивидуальных заданий 3. Устный опрос. | ПК-6.1, ПК-6.2, ПК-6.3 |
| 1.2 Запуск, исполнение и режимы исполнения процесса на вычислительной машине; механизмы памяти, схемы программ | | 6 | 6 | | 1 | 1. Самостоятельное изучение учебной и научной литературы. 2. Работа с электронным учебником, выполнение лабораторных работ | 1. Беседа - обсуждение 2. Проверка индивидуальных заданий 3. Устный/тестовый опрос. | ПК-6.1, ПК-6.2, ПК-6.3 |
| Итого по разделу | | 8 | 12 | | 3 | | | |
| 2. Механизмы и алгоритмы реализации процесса на вычислительной машине | | | | | | | | |
| 2.1 Модели вычислительных процессов; процессы и потоки в ОС; теория параллельных вычислений; алгоритмы | 8 | 4 | 6 | | 12 | 1. Самостоятельное изучение учебной и научной литературы. | 1. Беседа - обсуждение 2. Проверка индивидуальных заданий 3. Устный опрос. | ПК-6.1, ПК-6.2, ПК-6.3 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| диспетчеризации | | | | | | 2. Работа с электронным учебником, выполнение лабораторных работ | | |
| 2.2 Мультипрограммные системы и методы планирования в них; приоритеты процессов и потоков; системы реального времени и диспетчеризация в них | 8 | 4 | | | 12 | 1. Самостоятельное изучение учебной и научной литературы. 2. Работа с электронным учебником | 1. Беседа - обсуждение 2. Устный опрос. | ПК-6.1, ПК-6.2, ПК-6.3 |
| 2.3 Синхронизация процессов; блокировки и механизмы их разрешения | | 4 | 6 | | 12 | 1. Самостоятельное изучение учебной и научной литературы. 2. Работа с электронным учебником, выполнение лабораторных работ | 1. Беседа - обсуждение 2. Проверка индивидуальных заданий 3. Устный/тестовый опрос. | ПК-6.1, ПК-6.2, ПК-6.3 |
| Итого по разделу | | 12 | 12 | | 36 | | | |
| 3. Методы представления и технологии организации вычислений | | | | | | | | |
| 3.1 Принципы построения, алгоритмы поведения, способы реализации, области применения; классические задачи синхронизации | 8 | 2 | 4 | | 8 | 1. Самостоятельное изучение учебной и научной литературы. 2. Работа с электронным учебником, выполнение лабораторных работ | 1. Беседа – обсуждение. 2. Проверка индивидуальных заданий. 3. Устный опрос | ПК-6.1, ПК-6.2, ПК-6.3 |
| 3.2 Объекты ядра ОС, предназначенные для решения задач синхронизации; средства межпрограммного обмена | | 2 | 4 | | 1,8 | 1. Самостоятельное изучение учебной и научной литературы. 2. Работа с электронным учебником, выполнение лабораторных работ. | 1. Беседа – обсуждение. 2. Проверка индивидуальных заданий. 3. Устный/тестовый опрос | ПК-6.1, ПК-6.2, ПК-6.3 |
| Итого по разделу | | 4 | 8 | | 9,8 | | | |
| 4. Экзамен | | | | | | | | |
| 4.1 Экзамен | 8 | | | | | | Экзамен | ПК-6.1, ПК-6.2, ПК-6.3 |
| Итого по разделу | | | | | | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Итого за семестр | 24 | 32 | | 48,8 | | экзамен | |
| Итого по дисциплине | 24 | 32 | | 48,8 | | экзамен | |

## 5 Образовательные технологии

1. Традиционные образовательные технологии, ориентированные на организацию образовательного процесса и предполагающую прямую трансляцию знаний от преподавателя к студенту.

Формы учебных занятий с использованием традиционных технологий:

Информационная лекция – последовательное изложение материала в дисциплинарной логике, осуществляемое преимущественно вербальными средствами (монолог преподавателя).

Лабораторная работа – организация учебной работы с реальными материальными и информационными объектами, экспериментальная работа с аналоговыми моделями реальных объектов.

2. Технологии проблемного обучения – организация образовательного процесса, которая предполагает постановку проблемных вопросов, создание учебных проблемных ситуаций для стимулирования активной познавательной деятельности студентов.

3. Интерактивные технологии – организация образовательного процесса, которая предполагает активное и нелинейное взаимодействие всех участников, достижение на этой основе личностно значимого для них образовательного результата.

Формы учебных занятий с использованием специализированных интерактивных технологий:

Лекция «обратной связи» – лекция–провокация (изложение материала с заранее запланированными ошибками), лекция-беседа, лекция-дискуссия, лекция-пресс-конференция.

4. Информационно-коммуникационные образовательные технологии – организация образовательного процесса, основанная на применении программных сред и технических средств работы с знаниями в различных предметных областях.

## 6 Учебно-методическое обеспечение самостоятельной работы обучающихся

Представлено в приложении 1.

## 7 Оценочные средства для проведения промежуточной аттестации

Представлены в приложении 2.

## 8 Учебно-методическое и информационное обеспечение дисциплины

**а) Основная литература:**

1.Беспалов, Д. А. Операционные системы реального времени и технологии разработки кроссплатформенного программного обеспечения. Часть 1 : учебное пособие / Д. А. Беспалов, С. М. Гушанский, Н. М. Коробейникова ; Южный федеральный университет. - Ростов-на-Дону ; Таганрог : Издательство Южного федерального университета, 2019. - 139 с. - ISBN 978-5-9275-3367-1. - Текст : электронный. - URL: https://znanium.com/catalog/product/1088203 (дата обращения: 21.04.2024). – Режим доступа: по подписке..

2. Беспалов, Д. А. Операционные системы реального времени и технологии разработки кроссплатформенного программного обеспечения. Часть 2 : учебное пособие / Д. А. Беспалов, С. М. Гушанский, Н. М. Коробейникова ; Южный федеральный университет. - Ростов-на-Дону ; Таганрог : Издательство Южного федерального университета, 2019. - 168 с. - ISBN 978-5-9275-3368-8. - Текст : электронный. - URL: https://znanium.com/catalog/product/1088205 (дата обращения: 21.04.2024). – Режим доступа: по подписке.

**б) Дополнительная литература:**

1. Кузнецов, А.С. Системное программирование : учеб. пособие / А.С. Кузнецов, И.А. Якимов, П.В. Пересунько. - Красноярск : Сиб. федер. ун-т 2018. - 170с. - ISBN 978-5-7638-3885-5. - Текст : электронный. - URL: https://znanium.com/catalog/product/1032183 (дата обращения: 21.04.2024). – Режим доступа: по подписке.

2. Карепова, Е. Д. Основы многопоточного и параллельного программирования: Учебное пособие / Карепова Е.Д. - Краснояр.:СФУ, 2016. - 356 с.: ISBN 978-5-7638-3385-0. - Текст : электронный. - URL: https://znanium.com/catalog/product/966962 (дата обращения: 21.04.2024). – Режим доступа: по подписке.

**в) Методические указания:**

Методические указания приведены в Приложении 1

**г) Программное обеспечение и Интернет-ресурсы:**

**Программное обеспечение**

| Наименование ПО | № договора | Срок действия лицензии |
|---|---|---|
| MS Office 2007 Professional | № 135 от 17.09.2007 | бессрочно |
| MS Visual Studio 2017 Community Edition | свободно распространяемое ПО | бессрочно |

**Профессиональные базы данных и информационные справочные системы**

| Название курса | Ссылка |
|---|---|
| Национальная информационно-аналитическая система – Российский индекс научного цитирования (РИНЦ) | URL: https://elibrary.ru/project_risc.asp |

**9 Материально-техническое обеспечение дисциплины (модуля)**

Материально-техническое обеспечение дисциплины включает:

1. Лекционная аудитория ауд. 282. Мультимедийные средства хранения, передачи и представления информации.

2. Компьютерные классы Центра информационных технологий ФГБОУ ВО «МГТУ». Персональные компьютеры, объединенные в локальные сети с выходом в Internet, оснащенные современными программно-методическими комплексами для решения задач в области информатики и вычислительной техники.

3. Аудитории для самостоятельной работы: компьютерные классы; читальные залы библиотеки. Все классы УИТ и АСУ с персональными компьютерами, выходом в Интернет и с доступом в электронную информационно-образовательную среду университета.

4. Аудиторий для групповых и индивидуальных консультаций, текущего контроля и промежуточной аттестации. Ауд. 282 и классы УИТ и АСУ.

5. Помещения для самостоятельной работы обучающихся, оснащенных компьютерной техникой с возможностью подключения к сети «Интернет» и наличием доступа в электронную информационно-образовательную среду организации. Классы УИТ и АСУ.

6. Помещения для хранения и профилактического обслуживания учебного оборудования. Центр информационных технологий – ауд. 372

**Приложение 1**

**Учебно-методическое обеспечение самостоятельной работы обучающихся**

По дисциплине «Теория вычислительных процессов» предусмотрена аудиторная и внеаудиторная самостоятельная работа обучающихся.

Аудиторная самостоятельная работа студентов предполагает выполнение лабораторных работ по разделам.

## ЛАБОРАТОРНАЯ РАБОТА №1. ЧТЕНИЕ КАРТЫ ПРОЦЕССОВ И ПОТОКОВ

***Цель работы***

Изучить функции и процедуры семейства *ToolHelp32*, составляющих подмножество *Win32 API*, которые позволяют получить сведения о некоторых низкоуровневых аспектах работы ОС. В частности, получить информацию обо всех процессах, выполняющихся в системе в данный момент, а также потоках, модулях, принадлежащих каждому процессу.

***Информация***

Большинство данных, получаемых от функций *ToolHelp32*, используется, главным образом, приложениями, которые должны заглядывать «внутрь» ОС [2].

*Моментальные снимки*

Благодаря многозадачной природе ОС, такие объекты, как процессы, потоки, модули и т.п., постоянно создаются, разрушаются и модифицируются. И поскольку состояние компьютера непрерывно изменяется, системная информация, которая, возможно, будет иметь значение в данный момент, через секунду уже никого не заинтересует. Например, предположим, что необходимо написать программу для регистрации всех модулей, загруженных в систему. Поскольку операционная система в любое время может прервать выполнение потока, отрабатывающего программу, чтобы предоставить какие-то кванты времени другому потоку в системе, модули теоретически могут создаваться и разрушаться даже в момент выборки информации о них.

В этой динамической среде имеет смысл сделать «снимок» системы в заданный момент времени. Данный снимок делается с помощью функции *CreateToolhelp32Snapshot*.

```
HANDLEWINAPICreateToolhelp32Snapshot(
DWORDdwFlags,
DWORDth32ProcessID);
```

Параметр *dwFlags* означает тип информации, подлежащий включению в моментальный снимок. Этот параметр может иметь одно из значений, перечисленных в табл. 1.

Второй параметр, *th32ProcessID*, задает идентификатор процесса. Для текущего процесса данный параметр принимает значение *0*. Этот параметр используется в том случае, если параметр *dwFlags* принимает значения TH32CS_SNAPHEAPLIST или TH32CS_SNAPMODULE. В остальных случаях игнорируется (принимает значение *0*).

Таблица 1

Значение параметра *dwFlags*

| Значение | Описание |
|---|---|
| TH32CS_INHERIT | Означает, что дескриптор снимка будет наследуемым |
| TH32CS_SNAPALL | Эквивалентно заданию значений:<br>TH32CS_SNAPHEAPLIST,<br>TH32CS_SNAPMODULE,<br>TH32CS_SNAPPROCESS,<br>TH32CS_SNAPTHREAD |
| TH32CS_SNAPHEAPLIST | Включает в снимок список куч заданного процесса |
| TH32CS_SNAPMODULE | Включает в снимок список модулей заданного процесса |

| Значение | Описание |
|---|---|
| TH32CS_SNAPPROCESS | Включает в снимок список процессов |
| TH32CS_SNAPTHREAD | Включает в снимок список потоков |

Функция *CreateToolhelp32Snapshot* возвращает дескриптор созданного снимка или *-1* в случае ошибки. Возвращаемый дескриптор работает подобно другим дескрипторам относительно процессов и потоков, для которых он действителен. По завершении работы с созданным функцией *CreateToolhelp32Snapshot* дескриптором, для освобождения связанных с ним ресурсов используйте функцию *CloseHandle*.

*Обработка информации о процессах*

Имея дескриптор снимка, содержащий информацию о процессах, можно воспользоваться двумя функциями, которые позволяют последовательно просмотреть сведения обо всех процессах в системе. Функции*Process32First*и*Process32Next*определеныследующимобразом:

```
BOOL WINAPI Process32First(
HANDLE hSnapshot,
LPPROCESSENTRY32 lppe);

BOOL WINAPI Process32Next(
HANDLE hSnapshot,
LPPROCESSENTRY32 lppe);
```

Первыйпараметр, *hSnapshot*, уобеихфункцийявляетсядескрипторомснимка, возвращаемымфункцией*CreateToolhelp32Snapshot*.

Второй параметр, *lppe*, представляет собой структуру PROCESSENTRY32, которая передается по ссылке. По мере прохождения по элементам перечисления функции будут заполнять эту структуру информацией о следующем процессе. Запись PROCESSENTRY32 определяетсятак:

```
typedef struct tagPROCESSENTRY32 {
DWORD dwSize;
DWORD cntUsage;
DWORD th32ProcessID;
DWORD th32DefaultHeapID;
DWORD th32ModuleID;
DWORD cntThreads;
DWORD th32ParentProcessID;
LONG   pcPriClassBase;
DWORD dwFlags;
char szExeFile[MAX_PATH];
} PROCESSENTRY32;
typedef PROCESSENTRY32 *   PPROCESSENTRY32;
typedef PROCESSENTRY32 *   LPPROCESSENTRY32;
```

Поля структуры:

dwSize – размер структуры PROCESSENTRY32. До использования этой записи поле *dwSize* должно быть инициализировано значением *sizeof (*PROCESSENTRY32*)*;

cntUsage – значение счетчика ссылок процесса. Когда это значение станет равным нулю, операционная система выгрузит процесс;

th32ProcessID – идентификационный номер процесса.

th32DefaultHeapID – идентификатор *ID* для кучи процесса, действующей по умолчанию. Этот *ID* имеет значение только для функций *ToolHelp32*, и его нельзя использовать с другими функциями *Win32*;

th32ModuleID – идентифицирует модуль, связанный с процессом. Это поле имеет значение только для функций *ToolHelp32*;

cntThreads – количество потоков начало выполняться в данном процессе;
th32ParentProcessID – идентифицирует родительский процесс для данного процесса;
pcPriClassBase – базовый приоритет процесса. Операционная система использует это значение для управления работой потоков;
dwFlags – зарезервировано (не используется);
szExeFile содержит строку с ограничивающим нуль-символом, которая представляет собой путь и имя файла EXE-программы или драйвера, связанного с данным процессом.

После создания снимка, содержащего информацию о процессах, для опроса данных по каждому процессу следует вызвать сначала функцию *Process32First*, а затем вызывать функцию *Process32Next* до тех пор, пока она не вернет значение *false*.

*Обработка информации о потоках*

Для составления списка потоков некоторого процесса в *ToolHelp32* предусмотрены две функции, которые аналогичны функциям, предназначенным для регистрации процессов: *Thread32First* и *Thread32Next*, и объявляются следующим образом:

```
BOOL WINAPI Thread32First(
HANDLE hSnapshot,
LPTHREADENTRY32 lpte);

BOOL WINAPI Thread32Next(
HANDLE hSnapshot,
LPTHREADENTRY32 lpte);
```

Помимообычногопараметра*hSnapshot* (дескрипторснимка, возвращаемымфункцией*CreateToolhelp32Snapshot*), этимфункциямтакжепередаетсяпоссылкепараметртипа THREADENTRY32. Как и в случае функций, работающих с процессами, каждая из них заполняет запись THREADENTRY32, объявление которой имеет вид:

```
typedef struct tagTHREADENTRY32{
DWORD   dwSize;
DWORD   cntUsage;
DWORD   th32ThreadID;
DWORD   th32OwnerProcessID;
LONG    tpBasePri;
LONG    tpDeltaPri;
DWORD   dwFlags;
} THREADENTRY32;
typedef THREADENTRY32 *   PTHREADENTRY32;
typedef THREADENTRY32 *   LPTHREADENTRY32;
```

Поля структуры:
dwSize – размер структуры, и поэтому оно должно быть инициализировано значением *sizeof(*THREADENTRY32*)* до использования этой структуры;
cntUsage – счетчик ссылок данного потока. При обнулении этого счетчика поток выгружается операционной системой;
th32ThreadID – идентификационный номер потока, который имеет значение только для функций *ToolHelp32*;
th32OwnerProcessID – идентификатор *ID* процесса, которому принадлежит данный поток. Этот *ID* можно использовать с другими функциями *Win32*;
tpBasePri – базовый класс приоритета потока. Это значение одинаково для всех потоков данного процесса. Описания этих значений приведены в табл. 2.
dwFlags – зарезервировано (не используется).

Таблица 2

Значение констант приоритетов (параметр *tpBasePri*)

| Константа | Значение |
|---|---|
| THREAD_PRIORITY_IDLE | –15 |
| THREAD_PRIORITY_LOWEST | -2 |
| THREAD_PRIORITY_BELOW_NORMAL | -1 |
| THREAD_PRIORITY_NORMAL | 0 |
| THREAD_PRIORITY_ABOVE_NORMAL | 1 |
| THREAD_PRIORITY_HIGHEST | 2 |
| THREAD_PRIORITY_TIME_CRITICAL | 15 |

Списки потоков, полученные с помощью функций *ToolHelp32*, не связываются с определенным потоком. Поэтому при сканировании потоков нужно обязательно проверять результат так, чтобы потоки были связаны с интересующим вас потоком.

*Обработка информации о модулях*

Опрос модулей выполняется практически так же, как опрос процессов или потоков. Для этого в *ToolHe1p32* предусмотрены функции *Module32First* и *Module32Next*, которые определяются следующим образом:

```
BOOL WINAPI Module32First(
HANDLE hSnapshot,
LPMODULEENTRY32 lpme);

BOOL WINAPI Module32Next(
HANDLE hSnapshot,
LPMODULEENTRY32 lpme);
```

*Пример.*С помощью функций и процедур семейства *ToolHelp32* считать информацию о процессах и потоках, запущенных в системе в определенный момент времени (выполнить «моментальный снимок - *snapshot*»).

```
#include <vcl.h>
#include <tlhelp32.h>

int main(){
//   Читаемкартупроцессов
HANDLE WINAPI SnapShot_Pr;
SnapShot_Pr=CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0);
if ((int)SnapShot_Pr==-1)
  ShowMessage("Не могу прочитать карту процессов");
tagPROCESSENTRY32 ProcEntry;
ProcEntry.dwSize=sizeof(ProcEntry);
// Считываем первый процесс из списка
Process32First(SnapShot_Pr,&ProcEntry);
// Читаемегоидентификатор
DWORD ProcID=ProcEntry.th32ProcessID;
// Имяисполняемогофайла
AnsiString ProcName=ProcEntry.szExeFile;
// Считываемколичествопотоков
DWORD ProcThreadcount=ProcEntry.cntThreads;
inti=1;
...
// Пока не опустеет список, читаем процессы
while (Process32Next(SnapShot_Pr,&ProcEntry))
 {
    i++;
    ProcID=ProcEntry.th32ProcessID;
```

```
    ProcName=ProcEntry.szExeFile;
    ProcThreadcount=ProcEntry.cntThreads;
    ...
} // С потоками поступаем так же
return 0;
}
```

Задание для самостоятельного решения

*Требуется создать программу, позволяющую прочитать список запущенных в системе процессов и потоков. На рис. 1 приведен результат работы программы.*

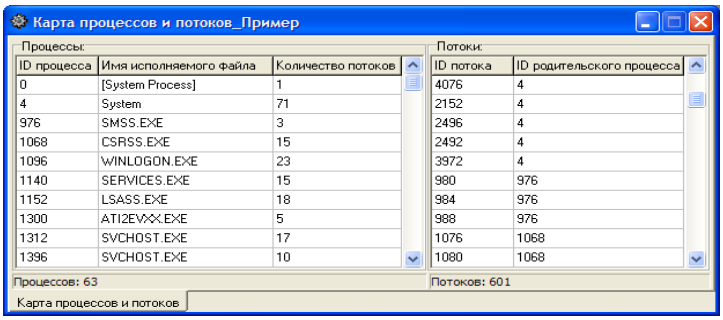



Рис. 1. Карта процессов и потоков

*Контрольные вопросы*

1. Приведите понятие «моментального снимка». С помощью какой функции его можно получить?
2. Информацию о каких объектах ядра операционной системы содержит «моментальный снимок»?
3. Возможно, ли создать снимок, содержащий только информацию о процессах, запущенных в системе?
4. Какую информацию можно узнать о процессе на основе созданного снимка?
5. Какую информацию можно узнать о потоке на основе созданного снимка?
6. Какую функцию необходимо применить к созданному снимку для получения информации о всех процессах, запущенных в системе?
7. Какую функцию необходимо применить к созданному снимку для получения информации о всех потоках, запущенных в системе?
8. Как определить для любого потока, запущенного в системе, его родительский процесс?
9. Какие действия выполнит операционная система, когда счетчик ссылок процесса станет равным нулю?
10. Перечислите основные базовые классы приоритетов потоков?

## ЛАБОРАТОРНАЯ РАБОТА №2. ЧТЕНИЕ КАРТЫ ПАМЯТИ

***Цель работы***

Получение практических навыков по использованию *Win32API* для исследования памяти *Windows*.

***Информация***

*Виртуальное адресное пространство процесса*

Каждому процессу выделяется собственное виртуальное адресное пространство [3]. Для *32*-разрядных процессов его размер составляет *4 Гб*. Соответственно *32*-битный указатель может быть любым числом от *0x00000000* до *0xFFFFFFFF*. Всего, таким образом, указатель может принимать *4 294 967 296* значений, что как раз и перекрывает четырехгигабайтовый диапазон. Для *64*-разрядных процессов размер адресного пространства равен *16* экзабайтам, поскольку *64*-битный указатель может быть любым числом от *0x00000000 00000000* до *0xFFFFFFFF FFFFFFFF*.

Поскольку каждому процессу отводится закрытое адресное пространство, то когда в

процессе выполняется какой-нибудь поток, он получает доступ только к той памяти, которая принадлежит его процессу. Память, отведенная другим процессам, скрыта от этого потока и недоступна ему.

Виртуальное адресное пространство каждого процесса разбивается на разделы. Их размер и назначение в какой-то мере зависят от конкретного ядра *Windows* (табл. 3).

В разделе «Для кода и данных пользовательского режима» располагается закрытая (неразделяемая) часть адресного пространства процесса. Ни один процесс не может получить доступ к данным другого процесса, размещенным в этом разделе. Основной объем данных, принадлежащих процессу, хранится именно в этом разделе (это касается всех приложений). Поэтому приложения менее зависимы от взаимных «капризов», и вся система функционирует устойчивее.

*Windows-функции, сообщающие о состоянии системной памяти и виртуального адресного пространства в процессах*

Многие параметры операционной системы (размер страницы, гранулярность выделения памяти и др.) зависят от используемого в компьютере процессора. Поэтому нельзя жестко «зашивать» их значения в исходный код программ. Эту информацию необходимо считывать в момент инициализации процесса с помощью функции *GetSystemInfo*:

*VOID GetSystemInfo(LPSYSTEM_INFO lpSystemInfo);*

Таблица 3

Разделы адресного пространства процесса

| Раздел | 32-разрядная Windows 2000 (на x86 и Alpha) | 64-разрядная Windows 2000 (на Alpha и IA-64) | Windows 98 |
|---|---|---|---|
| Для выявления нулевых указателей | 0x00000000 0x0000FFFF | 0x00000000 00000000 0x00000000 0000FFFF | 0x00000000 0x00000FFF |
| Для совместимости с программами DOS и 16-разрядной Windows | Нет | Нет | 0x00001000 0x003FFFFF |
| Для кода и данных пользовательского режима | 0x00010000 0x7FFEFFFF | 0x00000000 00010000 0x000003FF FFFEFFFF | 0x00400000 0x7FFFFFFF |
| Закрытый, размером 64 Кб | 0x7FFF0000 0x7FFFFFFF | 0x000003FF FFFF0000 0x000003FF FFFFFFFF | Нет |
| Для общих MMF (файлов, проецируемых в память) | Нет | Нет | 0x80000000 0xBFFFFFFF |
| Для кода и данных режима ядра | 0x80000000 0xFFFFFFFF | 0x00000400 00000000 0xFFFFFFFF FFFFFFFF | 0xC0000000 0xFFFFFFFF |

В функцию *GetSystemInfo* передается адрес структуры SYSTEM_INFO, и функция инициализирует элементы этой структуры:

```
typedef struct _SYSTEM_INFO {
union {
    DWORD   dwOemId;
    struct {
        WORD wProcessorArchitecture;
        WORD wReserved;
    };
};
DWORD   dwPageSize;
```

*LPVOID lpMinimumApplicationAddress;*
*LPVOID lpMaximumApplicationAddress;*
*DWORD dwActiveProcessorMask;*
*DWORD dwNumberOfProcessors;*
*DWORD dwProcessorType;*
*DWORD dwAllocationGranularity;*
*WORD wProcessorLevel;*
*WORDwProcessorRevision;*

*} SYSTEM_INFO;*

При загрузке система определяет значения элементов этой структуры; для конкретной системы их значения постоянны. Функция *GetSystemInfo* предусмотрена специально для того, чтобы и приложения могли получать эту информацию (рис. 2). Из всех элементов структуры SYSTEM_INFO лишь четыре имеют отношение к памяти (табл. 4).

Таблица 4

Элементы структуры SYSTEM_INFO

| Элемент | Описание |
|---|---|
| dwPageSize | Размер страницы памяти. На процессорах x86 это значение равно 4096, а на процессорах Alpha – 8 192 байтам |
| lpMinimumApplicationAddress | Минимальный адрес памяти доступного адресного пространства для каждого процесса. В Windows 98 это значение равно 4 194 304, или 0x00400000, поскольку нижние 4 Мб адресного пространства каждого процесса недоступны. В Windows 2000 это значение равно 65536, или 0x00010000, так как в этой системе резервируются лишь первые 64 Кб адресного пространства каждого процесса |
| lpMaximumApplicationAddress | Максимальный адрес памяти доступного адресного пространства, отведенного в «личное пользование» каждому процессу. В Windows 98 этот адрес равен 2 147 483 647, или 0x7FFFFFFF, так как верхние 2 Гб занимают общие файлы, проецируемые в память, и разделяемый код операционной системы. В Windows 2000 этот адрес соответствует началу раздела для кода и данных режима ядра за вычетом 64 Кб |
| dwAllocationGranularity | Гранулярность резервирования регионов адресного пространства. |

Остальные элементы структуры SYSTEM_INFOприведены в табл. 5.

Таблица 5

Элементы структуры SYSTEM_INFO

| Элемент | Описание |
|---|---|
| dwOemId | Устарел; больше не используется |
| wReserved | Зарезервирован на будущее; пока не используется |
| dwNumberOfProcessors | Число процессоров в компьютере |
| dwActiveProcessorMask | Битовая маска, которая сообщает, какие процессоры активны (выполняют потоки) |
| dwProcessorType | Используется только в Windows 98; сообщает тип процессора, например Intel 386, 486 или Pentium |
| wProcessorArchitecture | Используется только в Windows 2000; сообщает тип архитектуры процессора, например Intel, Alpha, 64-разрядный Intel или 64-разрядная Alpha |

| Элемент | Описание |
|---|---|
| wProcessorLevel | Используется только в Windows 2000; сообщает дополнительные подробности об архитектуре процессора, например IntelPentiumPro или PentiumII |
| wProcessorRevision | Используется только в Windows 2000; сообщает дополнительные подробности об уровне данной архитектуры процессора |

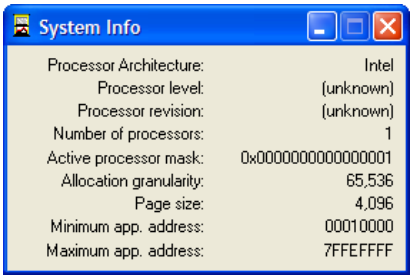



Рис. 2. Системная информация [3],
полученная при вызове функции *GetSystemInfo*

В Windows имеется функция, позволяющая запрашивать определенную информацию об участке памяти по заданному адресу (в пределах адресного пространства вызывающего процесса): размер, тип памяти и атрибуты защиты. Описаниефункции*VirtualQuery*:

*DWORD VirtualQuery(*
*LPCVOID lpAddress,*
*PMEMORY_BASIC_INFORMATION lpBuffer,*
*DWORDdwLength);*

Парная ей функция, *VirtualQueryEx*,сообщает ту же информацию о памяти, но в другом процессе. Описаниефункции*VirtualQueryEx*:

*DWORD VirtualQueryEx(*
*HANDLE hProcess,*
*LPCVOID lpAddress,*
*PMEMORY_BASIC_INFORMATION lpBuffer,*
*DWORDdwLength);*

Эти функции идентичны с тем исключением, что *VirtualQueryEx* принимает описатель процесса, об адресном пространстве которого необходимо получить информацию. Чаще всего функцией *VirtualQueryEx* пользуются отладчики и системные утилиты – остальные приложения обращаются к *VirtualQuery*. При вызове *VirtualQuery(Ex)* параметр *lpAddress*должен содержать адрес виртуальной памяти, о которой необходимо получить информацию. Параметр *lpBuffer*– это адрес структуры MEMORY_BASIC_INFORMATION, которую надо создать перед вызовом функции. Данная структура определена в следующем виде:

*typedefstruct _MEMORY_BASIC_INFORMATION { // mbi*
*PVOID BaseAddress;*
*PVOID AllocationBase;*
*DWORD AllocationProtect;*
*DWORD RegionSize;*
*DWORD State;*
*DWORD Protect;*
*DWORD Type;*
*} MEMORY_BASIC_INFORMATION;*
*typedef*
*MEMORY_BASIC_INFORMATION *PMEMORY_BASIC_INFORMATION;*

Параметр*dwLength*задаетразмерструктуры MEMORY_BASIC_INFORMATION. Функция*VirtualQuery(Ex)*возвращаетчислобайтов, скопированныхвбуфер.

Используя адрес, указанный в параметре *lpAddress*, функция *VirtualQuery(Ex)* заполняет структуру информацией о диапазоне смежных страниц, имеющих одинаковые состояние, атрибуты защиты и тип. Описание элементов структуры приведено в табл. 6.

Отдельным страницам физической памяти можно присвоить свои атрибуты защиты,

представленные в табл. 7.

Так как функция *VirtualQueryEx* принимает описатель процесса, возникает проблема получения описателя (дескриптора) процесса по известному идентификатору процесса. По идентификатору можно определить дескриптор любого процесса с помощью функции *OpenProcess*:

*HANDLE OpenProcess(*
*DWORD dwDesiredAccess,*
*BOOL bInheritHandle,*
*DWORD dwProcessId);*

Таблица 6

Элементыструктуры MEMORY_BASIC_INFORMATION

| Элемент | Описание |
|---|---|
| BaseAddress | Сообщает то же значение, что и параметр *lpAddress*, но округленное до ближайшего меньшего размера, кратного размеру страницы |
| AllocationBase | Идентифицирует базовый адрес региона, включающего в себя адрес, указанный в параметре *lpAddress* |
| AllocationProtect | Идентифицирует атрибут защиты, присвоенный региону при его резервировании (табл. 7) |
| RegionSize | Сообщает суммарный размер (в байтах) группы страниц, которые начинаются с базового адреса *BaseAddress* и имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре *lpAddress* |
| State | Сообщает состояние (MEM_FREE, MEM_RESERVE или MEM_COMMIT) всех смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре *lpAddress.* При MEM_FREE элементы *AllocationBase*, *AllocationProtect*, *Protect* и *Type* содержат неопределенные значения, а при MEM_RESERVE неопределенное значение содержит элемент *Protect* |
| Protect | Идентифицирует атрибут защиты (PAGE_*) всех смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре *lpAddress* (табл. 7) |
| Type | Идентифицирует тип физической памяти (MEM_IMAGE, MEM_MAPPED или MEM_PRIVATE) (табл. 8), связанной с группой смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре *lpAddress.* |

Таблица 7

Атрибуты защиты

| Атрибут защиты | Описание |
|---|---|
| PAGE_NOACCESS | Попытки чтения, записи или исполнения содержимого памяти на этой странице вызывают нарушение доступа |
| PAGE_READONLY | Попытки записи или исполнения содержимого памяти на этой странице вызывают нарушение доступа |
| PAGE_READWRITE | Попытки исполнения содержимого памяти на этой странице вызывают нарушение доступа |
| PAGE_EXECUTE | Попытки чтения или записи на этой странице вызывают нарушение доступа |
| PAGE_EXECUTE_READ | Попытки записи на этой странице вызывают нарушение доступа |
| PAGE_EXECUTE_READWRITE | На этой странице возможны любые операции |

| Атрибут защиты | Описание |
|---|---|
| PAGE_WRITECOPY | Попытки исполнения содержимого памяти на этой странице вызывают нарушение доступа; попытка записи приводит к тому, что процессу предоставляется «личная» копия данной страницы |
| PAGE_EXECUTE_WRITECOPY | На этой странице возможны любые операции; попытка записи приводит к тому, что процессу предоставляется «личная» копия данной страницы |
| Специальные флаги атрибутов защиты | |
| PAGE_NOCACHE | Отключает кэширование переданных страниц. Данный флаг предусмотрен главным образом для разработчиков драйверов устройств при манипулировании буферами памяти |
| PAGE_GUARD | Позволяет приложениям получать уведомление (через механизм исключений) в тот момент, когда на страницу записывается какой-нибудь байт |
| PAGE_WRITECOMBINE | Предназначен для разработчиков драйверов устройств. Позволяет объединять несколько операций записи на устройство в один пакет, что увеличивает скорость передачи данных |

Типы регионов памяти приведены в табл. 8.

Таблица 8

Типы регионов памяти

| Тип | Описание |
|---|---|
| Free | Этот диапазон виртуальных адресов не сопоставлен ни с каким типом физической памяти. Его адресное пространство не зарезервировано; приложение может зарезервировать регион по указанному базовому адресу или в любом месте в границах свободного региона |
| Private | Этот диапазон виртуальных адресов сопоставлен со страничным файлом |
| Image | Этот диапазон виртуальных адресов изначально был сопоставлен с образом EXE- или DLL-файла, проецируемого в память, но теперь, возможно, уже нет. Например, при записи в глобальную переменную из образа модуля механизм поддержки «копирования при записи» выделяет соответствующую страницу памяти из страничного файла, а не исходного образа файла |
| Mapped | Этот диапазон виртуальных адресов изначально был сопоставлен с файлом данных, проецируемым в память, но теперь, возможно, уже нет. Например, файл данных мог быть спроецирован с использованием механизма поддержки «копирование при записи». Любые операции записи в этот файл приведут к тому, что соответствующие страницы памяти будут выделены из страничного файла, а не из исходного файла данных |

Параметр *dwDesiredAccess* имеет отношение к правам доступа и может принимать различные значения (табл. 9).

Таблица 9

Значения параметра

| Значение | Описание |
|---|---|
| PROCESS_ALL_ACCESS | Эквивалентно установке флагов полного доступа |
| PROCESS_CREATE_PROCESS | Для внутреннего использования |
| PROCESS_CREATE_THREAD | Позволяет использовать дескриптор процесса в функции *CreateRemote-Thread* для создания потоков в процессе |
| PROCESS_DUP_HANDLE | Использует дескриптор, как исходного процесса, так и принимающего в функции *DuplicateHandle* для копирования (дублирования) дескриптора |

| Значение | Описание |
|---|---|
| PROCESS_QUERY_INFORMATION | Задействует дескриптор процесса для чтения информации из объекта *Process* |
| PROCESS_SET_INFORMATION | Позволяет использовать дескриптор процесса в *SetPriorityClass* функцию, чтобы установить класс приоритета процесса |
| PROCESS_TERMINATE | Работает для завершения процесса с его дескриптором в функции *TerminateProcess* |
| PROCESS_VM_OPERATION | Использует дескриптор процесса для модификации виртуальной памяти процесса |
| PROCESS_VM_READ | Применяет для чтения из виртуальной памяти процесса его дескриптора в функции *ReadProcessMemory* |
| PROCESS_VM_WRITE | Использует для записи в виртуальную память процесса его дескриптора в функции *WriteProcessMernory* |
| SYNCHRONIZE | Windows NT: работает с дескриптором процесса в любой из функций ожидания, таких как *WaitForSingleObject*, для ожидания завершения процесса |

Параметр *bInheritHandle* – установлен в значение TRUE, для того чтобы позволить порожденным процессам наследовать дескриптор. Иначе говоря, порожденный процесс получает дескриптор родительского процесса. Отметим, что значение дескриптора может изменяться.

Параметр *dwProcessID* – должен иметь значение идентификатора того процесса, дескриптор которого нужно узнать.

Функция *OpenProcess* возвращает дескриптор указанного процесса.

*Пример.* Определить информацию о первом регионе памяти адресного пространства *(суммарный размер (в байтах) группы страниц, которые начинаются с базового адреса (минимальный адрес памяти доступного адресного пространства) и имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по данному адресу)* вызываемого процесса.

```
{
  _SYSTEM_INFO sysinfo;
  GetSystemInfo(&sysinfo);
  LPVOID minAddress=sysinfo.lpMinimumApplicationAddress,
         maxAddress=sysinfo.lpMaximumApplicationAddress;
  LabelEdit1->Text=IntToHex((int)minAddress,8);
  LabelEdit2->Text=IntToHex((int)maxAddress,8);
  _MEMORY_BASIC_INFORMATION meminfo;
  VirtualQuery(minAddress,&meminfo,sizeof(meminfo));
  LabelEdit3->Text=IntToStr(meminfo.RegionSize);
}
```

Результат работы программы представлен на рис. 3.

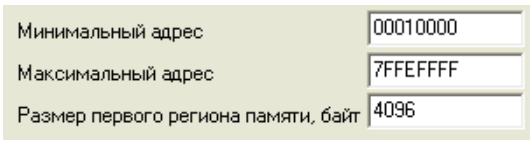



Рис. 3. Информация об адресном пространстве и первом регионе памяти

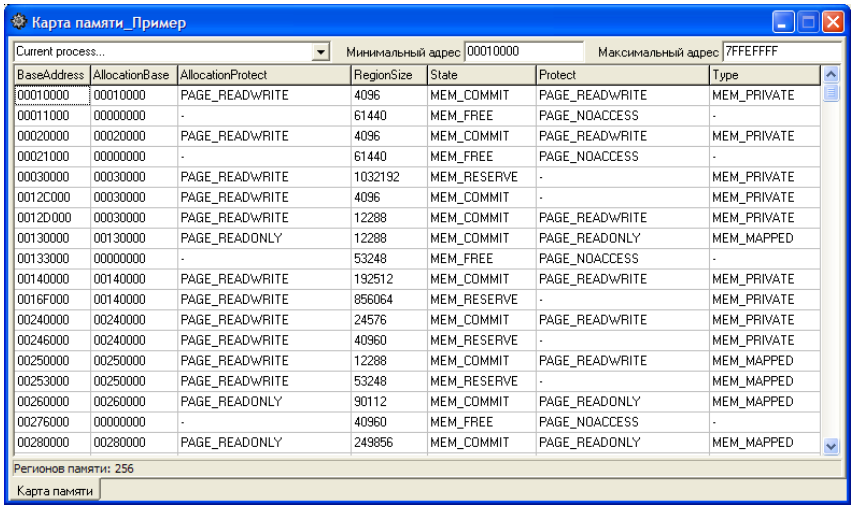
Адрес следующего региона получаем так:

*<Текущий адрес региона> + <объем текущего региона>*

*Далее продолжаем итерации, пока не доберемся до максимального из доступных адресов.*

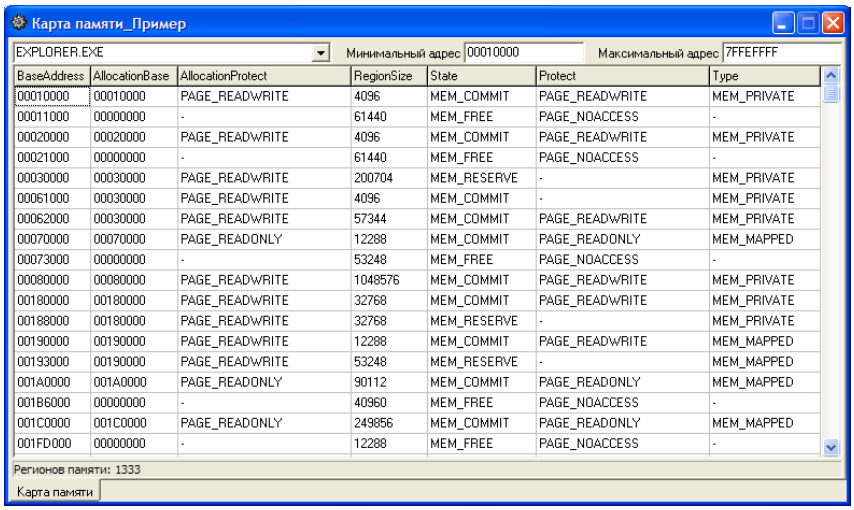
## Задание для самостоятельного решения

Требуется создать программу, позволяющую прочитать информацию (размер, тип памяти и атрибуты защиты) о регионах памяти адресного пространства процесса, как для вызываемого процесса (рис. 4), так и для любого процесса, запущенного в системе (рис. 5).




| BaseAddress | AllocationBase | AllocationProtect | RegionSize | State | Protect | Type |
|---|---|---|---|---|---|---|
| 00010000 | 00010000 | PAGE_READWRITE | 4096 | MEM_COMMIT | PAGE_READWRITE | MEM_PRIVATE |
| 00011000 | 00000000 | - | 61440 | MEM_FREE | PAGE_NOACCESS | - |
| 00020000 | 00020000 | PAGE_READWRITE | 4096 | MEM_COMMIT | PAGE_READWRITE | MEM_PRIVATE |
| 00021000 | 00000000 | - | 61440 | MEM_FREE | PAGE_NOACCESS | - |
| 00030000 | 00030000 | PAGE_READWRITE | 1032192 | MEM_RESERVE | - | MEM_PRIVATE |
| 0012C000 | 00030000 | PAGE_READWRITE | 4096 | MEM_COMMIT | - | MEM_PRIVATE |
| 0012D000 | 00030000 | PAGE_READWRITE | 12288 | MEM_COMMIT | PAGE_READWRITE | MEM_PRIVATE |
| 00130000 | 00130000 | PAGE_READONLY | 12288 | MEM_COMMIT | PAGE_READONLY | MEM_MAPPED |
| 00133000 | 00000000 | - | 53248 | MEM_FREE | PAGE_NOACCESS | - |
| 00140000 | 00140000 | PAGE_READWRITE | 192512 | MEM_COMMIT | PAGE_READWRITE | MEM_PRIVATE |
| 0016F000 | 00140000 | PAGE_READWRITE | 856064 | MEM_RESERVE | - | MEM_PRIVATE |
| 00240000 | 00240000 | PAGE_READWRITE | 24576 | MEM_COMMIT | PAGE_READWRITE | MEM_PRIVATE |
| 00246000 | 00240000 | PAGE_READWRITE | 40960 | MEM_RESERVE | - | MEM_PRIVATE |
| 00250000 | 00250000 | PAGE_READWRITE | 12288 | MEM_COMMIT | PAGE_READWRITE | MEM_MAPPED |
| 00253000 | 00250000 | PAGE_READWRITE | 53248 | MEM_RESERVE | - | MEM_MAPPED |
| 00260000 | 00260000 | PAGE_READONLY | 90112 | MEM_COMMIT | PAGE_READONLY | MEM_MAPPED |
| 00276000 | 00000000 | - | 40960 | MEM_FREE | PAGE_NOACCESS | - |
| 00280000 | 00280000 | PAGE_READONLY | 249856 | MEM_COMMIT | PAGE_READONLY | MEM_MAPPED |

Рис. 4. Карта памяти вызываемого процесса




| BaseAddress | AllocationBase | AllocationProtect | RegionSize | State | Protect | Type |
|---|---|---|---|---|---|---|
| 00010000 | 00010000 | PAGE_READWRITE | 4096 | MEM_COMMIT | PAGE_READWRITE | MEM_PRIVATE |
| 00011000 | 00000000 | - | 61440 | MEM_FREE | PAGE_NOACCESS | - |
| 00020000 | 00020000 | PAGE_READWRITE | 4096 | MEM_COMMIT | PAGE_READWRITE | MEM_PRIVATE |
| 00021000 | 00000000 | - | 61440 | MEM_FREE | PAGE_NOACCESS | - |
| 00030000 | 00030000 | PAGE_READWRITE | 200704 | MEM_RESERVE | - | MEM_PRIVATE |
| 00061000 | 00030000 | PAGE_READWRITE | 4096 | MEM_COMMIT | - | MEM_PRIVATE |
| 00062000 | 00030000 | PAGE_READWRITE | 57344 | MEM_COMMIT | PAGE_READWRITE | MEM_PRIVATE |
| 00070000 | 00070000 | PAGE_READONLY | 12288 | MEM_COMMIT | PAGE_READONLY | MEM_MAPPED |
| 00073000 | 00000000 | - | 53248 | MEM_FREE | PAGE_NOACCESS | - |
| 00080000 | 00080000 | PAGE_READWRITE | 1048576 | MEM_COMMIT | PAGE_READWRITE | MEM_PRIVATE |
| 00180000 | 00180000 | PAGE_READWRITE | 32768 | MEM_COMMIT | PAGE_READWRITE | MEM_PRIVATE |
| 00188000 | 00180000 | PAGE_READWRITE | 32768 | MEM_RESERVE | - | MEM_PRIVATE |
| 00190000 | 00190000 | PAGE_READWRITE | 12288 | MEM_COMMIT | PAGE_READWRITE | MEM_MAPPED |
| 00193000 | 00190000 | PAGE_READWRITE | 53248 | MEM_RESERVE | - | MEM_MAPPED |
| 001A0000 | 001A0000 | PAGE_READONLY | 90112 | MEM_COMMIT | PAGE_READONLY | MEM_MAPPED |
| 001B6000 | 00000000 | - | 40960 | MEM_FREE | PAGE_NOACCESS | - |
| 001C0000 | 001C0000 | PAGE_READONLY | 249856 | MEM_COMMIT | PAGE_READONLY | MEM_MAPPED |
| 001FD000 | 00000000 | - | 12288 | MEM_FREE | PAGE_NOACCESS | - |

Рис. 5. Карта памяти процесса (explorer.exe),
запущенного в системе

## Контрольные вопросы

1. Что такое адресное пространство процесса?
2. Перечислите области, которые присутствуют в адресном пространстве ОС *Windows*.
3. Какую функцию необходимо применить для получения информации об адресном пространстве (диапазон адресов, размер страницы, гранулярность выделения памяти и др.)?
4. Размер адресного пространства для 32-разрядных процессов?
5. Назовите раздел адресного пространства и его размер для 32-разрядных процессов, к которому пользователь имеет доступ?
6. Размер адресного пространства для 64-разрядных процессов?
7. Какую функцию необходимо использовать для получения информации об участке памяти (размер, тип памяти и атрибуты защиты) по заданному адресу (в пределах адресного пространства вызывающего процесса)?
8. Для каких процессов разрешено чтение информации об адресном пространстве?
9. В чем заключаются отличия между функциями *VirtualQuery* и *VirtualQueryEx*?
10. Какую функцию необходимо применить для получения информации о процессоре?

# ЛАБОРАТОРНАЯ РАБОТА №3.
# МНОГОПОТОКОВАЯ ОБРАБОТКА

***Цель работы***

Изучить функции, предназначенные для создания (порождения) дополнительных потоков в системе, принципы использования системой объектов ядра «поток» для управления потоками.

***Информация***

Любой поток состоит из двух компонентов:

объекта ядра, через который операционная система управляет потоком. Там же хранится статистическая информация о потоке;

стека потока, который содержит параметры всех функций и локальные переменные, необходимые потоку для выполнения кода.

Процесс ничего не исполняет, он просто служит контейнером потоков. Потоки всегда создаются в контексте какого-либо процесса, и вся их жизнь проходит только в его границах. На практике это означает, что потоки исполняют код и манипулируют данными в адресном пространстве процесса. Поэтому, если два и более потока выполняется в контексте одного процесса, все они делят одно адресное пространство. Потоки могут исполнять один и тот же код и манипулировать одними и теми же данными, а также совместно использовать описатели объектов ядра, поскольку таблица описателей создается не в отдельных потоках, а в процессах [3].

Поток (*thread*) определяет последовательность исполнения кода в процессе. При инициализации процесса система всегда создает первичный поток. Большинство приложений обходится единственным, первичным потоком. Однако процессы могут создавать дополнительные потоки, что позволяет им эффективнее выполнять свою работу.

Каждый поток начинает выполнение с некоей входной функции. В первичном потоке таковым является *main, wmain, WinMain* или *wWinMain*. Если необходимо создать вторичный поток, в нем должна быть входная функция, которая выглядит примерно так:

```
DWORD WINAPI ThreadFunc(PVOID lpParam)   {
DWORDdwResult=0;
...
return(dwResult)
}
```

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

В основе реализации функции потока заложены следующие требования.

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

Поскольку входным функциям первичного потока передаются строковые параметры, они существуют в ANSI- и Unicode- версиях: *main – wmain* и *WinMain – wWinMain.* Но функциям потока передается единственный параметр, смысл которого определяется программистом, а не операционной системой. Поэтому проблем с ANSI/Unicode нет.

Функция потока должна возвращать значение, которое будет использоваться как код завершения потока. Полная аналогия с библиотекой C/C++: код завершения первичного потока становится кодом завершения процесса.

Функции потоков (да и все функции) должны по мере возможности обходиться своими параметрами и локальными переменными. Так как к статической или глобальной переменной могут одновременно обратиться несколько потоков, есть повредить ее

содержимое. Однако параметры и локальные переменные создаются в стеке потока, поэтому они в гораздо меньшей степени подвержены влиянию другого потока.

Реализовав функцию потока, необходимо, чтобы бы операционная система создала поток, который выполнит эту функцию.

*Создание потока*

Для создания дополнительных потоков необходимо вызвать из первичного потока функцию *CreateThread*:

*HANDLE CreateThread(*
*LPSECURITY_ATTRIBUTES lpThreadAttributes,*
*DWORD dwStackSize,*
*LPTHREAD_START_ROUTINE lpStartAddress,*
*LPVOID lpParameter,*
*DWORD dwCreationFlags,*
*LPDWORD lpThreadId);*

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

---

**ПРИМЕЧАНИЕ***CreateThread*– это Windows-функция, создающая поток. Если Вы пишите код на *C*/*C++* эффективнее использовать функцию *_beginthreadex* из библиотеки *VisualC++*.

---

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

Параметр *dwStackSize* определяет, какую часть адресного пространства поток сможет использовать под свой стек. Каждому потоку выделяется отдельный стек. Если при обращении к функции *CreateThread,* передается в параметре *dwStackSize* ненулевое значение, функция резервирует всю указанную память. Ее объем определяется либо значением параметра *dwStackSize,* либо значением, заданным в ключе /STACK (/STACK:[*reserve*][, *commit*] – аргумент *reserve* определяет объем адресного пространства, который система должна зарезервировать под стек потока (по умолчанию – 1Мб); аргумент *commit* задает объем физической памяти, который изначально передается области, зарезервированной под стек (по умолчанию – 1 страница)) компоновщика (выбирается большее из них). Но передается стеку лишь тот объем памяти, который соответствует значению в *dwStackSize*. Если же в параметре *dwStackSize* передается нулевое значение, *CreateThread* создает стек для нового потока, используя информацию, встроенную компоновщиком в EXE-файл.

Параметр *lpStartAddress* определяет адрес функции потока, с которой должен будет начать работу создаваемый поток, а параметр *lpParameter* идентичен параметру *lpParameter* функции потока. *CreateThread* лишь передает этот параметр по эстафете той функции, с которой начинается выполнение создаваемого потока. Таким образом, данный параметр позволяет передавать функции потока какое-либо инициализирующее значение. Оно может быть или просто числовым значением, или указателем на структуру данных с дополнительной информацией. Вполне допустимо и даже полезно создавать несколько потоков, у которых в качестве входной точки используется адрес одной и той же функции. Например, можно реализовать *Web*-сервер, который обрабатывает каждый клиентский запрос в отдельном потоке. При создании каждому потоку передается свое значение *lpParameter*. Так как Windows – операционная система с вытесняющей многозадачностью, следовательно, новый поток и поток, вызвавший *CreateThread,* могут выполняться одновременно, что может привести к определенным проблемам.

Параметр *dwCreationFlags* определяет дополнительные флаги, управляющие

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

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

---

**ПРИМЕЧАНИЕ** В *Windows 2000* и *Windows NT* в этом параметре можно передавать NULL. Тем самым сообщается функции, что программиста не интересует идентификатор потока. Но в *Windows 95/98* это приведет к ошибке, так как функция попытается записать идентификатор потока по нулевому адресу, что недопустимо. И поток не будет создан.

---

*Завершение потока*

Поток можно завершить четырьмя способами [3]:

функция потока возвращает управление (рекомендуемый способ);

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

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

завершается процесс, содержащий данный поток (нежелательный способ).

*Возврат управления функцией потока*

Функцию потока следует проектировать так, чтобы поток завершался только после того, как она возвращает управление. Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежащих потоку. При этом:

любые *C++* – объекты, созданные данным потоком, уничтожаются соответствующими деструкторами;

система корректно освобождает память, которую занимал стек потока;

система устанавливает код завершения данного потока (поддерживаемый объектом ядра «поток») – его и возвращает функция потока;

счетчик пользователей данного объекта ядра «поток» уменьшается на 1.

*Функция ExitThread*

Поток можно завершить принудительно, вызвав:

*VOIDExitThread(DWORDdwExitCode);*

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

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

---

**ПРИМЕЧАНИЕ***ExitThread*– это Windows-функция, которая уничтожает поток. Если Вы пишите код на *C/C++* эффективнее использовать функцию *_endthreadex* из библиотеки *VisualC++.*

---

*Функция TerminateThread*

Вызов этой функции также завершает поток.

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

В отличие от *ExitThread,* которая уничтожает только вызывающий поток, эта функция завершает поток, указанный в параметре *hThread*. В параметре *dwExitCode* указывается

значение, которое система рассматривает как код завершения потока. После того как поток будет уничтожен, счетчик пользователей его объекта ядра «поток» уменьшится на 1. Корректно написанное приложение не должно вызывать эту функцию, поскольку поток не получает никакого уведомления о завершении; из-за этого он не может выполнить должную очистку ресурсов.

---

**ПРИМЕЧАНИЕ** Уничтожение потока при вызове *ExitThread*или возврате управления из функции потока приводит к разрушению стека. Но если он завершен функцией *TerminateThread,* система не уничтожает стек, пока он не завершится и процесс, которому принадлежал этот поток. Так сделано потому, что другие потоки могут использовать указатели, ссылающиеся на данные в стеке завершенного потока. Если бы они обратились к несуществующему стеку, произошло бы нарушение доступа. Кроме того, при завершении потока система уведомляет об этом все DLL, подключенные к процессу – владельцу завершенного потока. Но при вызове *TerminateThread* такого не происходит, и процесс может быть завершен некорректно.

---

*Завершение процесса, содержащего данный поток*

Функции *ExitProcess* и *TerminateProcess* тоже завершают потоки. Единственное отличие в том, что они прекращают выполнение всех потоков, принадлежащих завершенному процессу. При этом гарантируется высвобождение любых выделенных процессу ресурсов, в том числе стеков потоков. Однако эти две функции уничтожают потоки принудительно – так, будто для каждого из них вызывается функция *TerminateThread*. А это означает, что очистка проводится некорректно: деструкторы *C++*-объектов не вызываются, данные на диск не сбрасываются и т.д.

При завершении потока сопоставленный с ним объект ядра «поток» не освобождается до тех пор, пока не будут закрыты все внешние ссылки на этот объект.

Для проверки завершен ли поток, идентифицируемый описателем *hThread,* из других потоков, запущенных в системе, используется функция *GetExitCodeThread.*

```
BOOL GetExitCodeThread(
HANDLE hThread,
LPDWORD lpExitCode);
```

Код завершения возвращается в переменной типа DWORD, на которую указывает *lpExitCode*. Если поток не завершен на момент вызова *GetExitCodeThread,* функция записывает в эту переменную идентификатор STILL_ACTIVE (0x103). При успешном вызове функция возвращает TRUE.

*Планирование потоков*

Операционная система с вытесняющей многозадачностью должна использовать тот или иной алгоритм, позволяющий ей распределять процессорное время между потоками. Каждые 20 мс (или около того) *Windows* просматривает все существующие объекты ядра «поток» и отмечает те из них, которые могут получать процессорное время. Далее она выбирает один из таких объектов и загружает в регистры процессора значения из его контекста. Эта операция называется *переключением контекста* (*contextswitching*). По каждому потоку *Windows* ведет учет того, сколько раз он подключался к процессору. Поток выполняет код и манипулирует данными в адресном пространстве своего процесса. Примерно через 20 мс *Windows* сохранит значения регистров процессора в контексте потока и приостановит его выполнение. Далее система просмотрит остальные объекты ядра «поток», подлежащие выполнению, выберет один из них, загрузит его контекст в регистры процессора, и все повторится. Этот цикл операций – выбор потока, загрузка его контекста, выполнение и сохранение контекста – начинается с момента запуска системы и продолжается до ее выключения. Таков вкратце механизм планирования работы множества потоков.

*Приостановка и возобновление потоков*

В объекте ядра «поток» имеется переменная – счетчик числа простоев данного

потока. При вызове *CreateProcess* или *CreateThread* он инициализируется значением, равным *1*, которое запрещает системе выделять новому потоку процессорное время. Такая схема весьма разумна: сразу после создания поток не готов к выполнению, ему нужно время для инициализации.

После того как поток полностью инициализирован, *CreateProcess* или *CreateThread* проверяет, не передан ли ей флаг CREATE_SUSPENDED, и, если да, возвращает управление, оставив поток в приостановленном состоянии. В ином случае счетчик простоев обнуляется, и поток включается в число планируемых – если только он не ждет какого-то события (например, ввода с клавиатуры).

Создав поток в приостановленном состоянии, можно настроить некоторые его свойства (например, приоритет). Закончив настройку, необходимо разрешить выполнение потока. Для этого вызывается функция *ResumeThread* и передается описатель потока *hThread*, возвращенный функцией *CreateThread* (описатель можно взять и из структуры, на которую указывает параметр *lpProcessInformation*, передаваемый в *CreateProcess*).

*DWORDResumeThread(HANDLEhThread);*

Если вызов *ResumeThread*прошел успешно, она возвращает предыдущее значение счетчика простоев данного потока; в противном случае – 0xFFFFFFFF.

Выполнение отдельного потока можно приостанавливать несколько раз. Если поток приостановлен 3 раза, то и возобновлен он должен быть тоже 3 раза – лишь тогда система выделит ему процессорное время. Выполнение потока можно приостановить не только при его создании с флагом CREATE_SUSPENDED, но и вызовом функции *SuspendThread*:

*DWORDSuspendThread(HANDLEhThread);*

Любой поток может вызвать эту функцию и приостановить выполнение другого потока (если его описатель известен). Приостановить свое выполнение поток способен сам, а возобновить себя – нет. Как и функция *ResumeThread*, функция *SuspendThread* возвращает предыдущее значение счетчика простоев данного потока. Поток можно приостанавливать не более чем MAXIMUM_SUSPEND_COUNT раз. Функция *SuspendThread* в режиме ядра работает асинхронно, но в пользовательском режиме не выполняется, пока поток остается в приостановленном состоянии.

*Дополнительные функции для работы с потоками*

*Функция Sleep*

Поток может сообщить системе о невыделении ему процессорного времени на определенный период, вызвав:

*VOIDSleep(DWORDdwMilliseconds);*

Эта функция приостанавливает поток на *dwMilliseconds* миллисекунд.

*Функция SwitchToThread*

Функция *SwitchToThread* позволяет подключить к процессору другой поток (если он есть):

*BOOL SwitchToThread(VOID)*

*SwitchToThread*позволяет потоку, которому не хватает процессорного времени, отнять этот ресурс у потока с более низким приоритетом. Она возвращает FALSE, если на момент ее вызова в системе нет ни одного потока, готового к исполнению; в ином случае – ненулевое значение.

Вызов функции *SwitchToThread*аналогичен вызову функции *Sleep* с передачей в *dwMilliseconds* нулевого значения. Разница лишь в том, что функция *SwitchToThread* дает возможность выполнять потоки с более низким приоритетом, которым не хватает процессорного времени, а функция *Sleep* действует без оглядки на «голодающие» потоки.

*Функции изменения приоритетов потоков*

*Windows* поддерживает шесть классов приоритета: *idle* (простаивающий), *belownormal*(ниже обычного), *normal* (обычный), *abovenormal* (выше обычного), *high* (высокий) и *realtime* (реального времени). Самый распространенный класс приоритета – *normal*; его используют 99% приложений. Для потоков *Windows* поддерживает семь относительных приоритетов: *idle* (простаивающий), *lowest* (низший), *belownormal* (ниже обычного), *normal* (обычный), *abovenormal*(выше обычного), *highest* (высший) и *time-critical* (критический по времени). Эти приоритеты относительны классу приоритета процесса. Большинство потоков использует обычный приоритет. Поэтому только что созданный поток получает относительный приоритет *normal*. При этом функция *CreateThread* не позволяет задать относительный приоритет. Операция изменения относительного приоритета потока осуществляется вызовом функции:

```
BOOLSetThreadPriority(
HANDLEhThread,
intnPriority);
```

Параметр *hThread* указывает на поток, чей приоритет необходимо изменить, а через параметр *nPriority* передается один из идентификаторов, соответствующий определенному приоритету потока (табл. 10).

Таблица 10

Значения параметра *nPriority* в соответствии с приоритетом потока

| Относительный приоритет потока | Идентификатор |
|---|---|
| Time-critical | THREAD_PRIORITY_TIME_CRITICAL |
| Highest | THREAD_PRIORITY_HIGHEST |
| Above normal | THREAD_PRIORITY_ABOVE_NORMAL |
| Normal | THREAD_PRIORITY_NORMAL |
| Below normal | THREAD_PRIORITY_BELOW_NORMAL |
| Lowest | THREAD_PRIORITY_LOWEST |
| Idle | HREAD_PRIORITY_IDLE |

Функция *GetThreadPriority*, парная *SetThreadPriority*, позволяет узнать относительный приоритет потока:

```
intGetThreadPriority(HANDLEhThread);
```

Она возвращает один из идентификаторов, приведенных в табл. 10.

*Пример.* Создать поток с относительным приоритетом *idle* и функцию потока, с которой должен будет начать работу создаваемый поток. В качестве параметра из первичного (главного) потока процесса в функцию потока передается текущее время, а функция создаваемого потока выводит переданную информацию на экран.

```
#include<windows.h>
#include<stdio.h>
#include<dos.h>

DWORD WINAPI ThreadFunc(PVOID lpParam)
{
struct time tt=*((struct time*)lpParam);
printf("\n The current time is: %2d:%02d:%02d.%02d\n",
        tt.ti_hour, tt.ti_min, tt.ti_sec, tt.ti_hund);
return 0;
}

int main()
```

```
    {
struct   time t;
gettime(&t);
DWORD dwThreadID;
HANDLE hThread = CreateThread(NULL,0,ThreadFunc,&t,
                    CREATE_SUSPENDED,&dwThreadID);
SetThreadPriority(hThread,THREAD_PRIORITY_IDLE);
ResumeThread(hThread);
CloseHandle(hThread);
getchar();
return 0;
    }
```

Результат работы программы представлен на рис. 6.

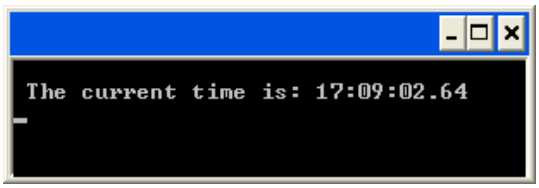
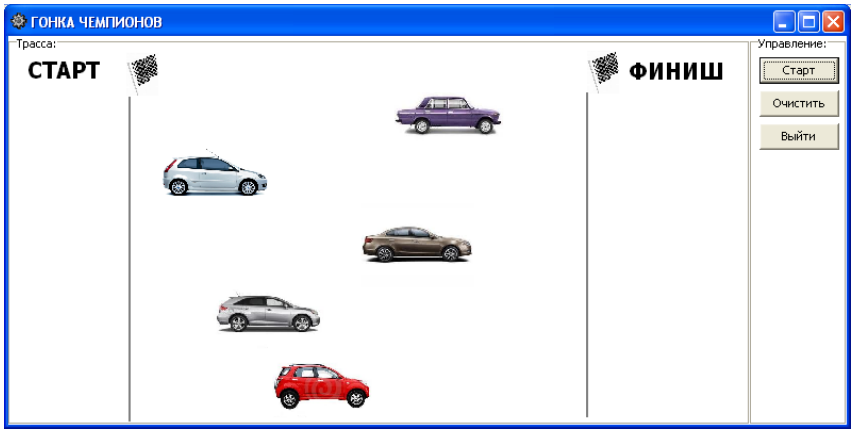



Рис. 6. Результат работы функции потока

Задание для самостоятельного решения

Выполнить имитатор гонок (в просторечии эта задача известна как «тараканьи бега») при помощи создания нескольких потоков (рис. 7). Каждый поток обслуживает свою «беговую дорожку». На исполнение все потоки запускаются одновременно, после чего потоки произвольным образом приостанавливаются и запускаются вновь. На исполнение каждому потоку выделяется квант времени (например, 500 мс или 1 с). За этот период поток производит выполнение задачи, например, увеличивает позицию гонщика на некоторую величину. После истечения кванта времени поток приостанавливается на произвольный период времени, определяемый при помощи генератора случайных чисел. После завершения гонки производится выдача результатов (очередность завершения).

Использование класса *TThread*, включенного в поставку *Borland Developer Studio*, допускается только в ознакомительных целях.



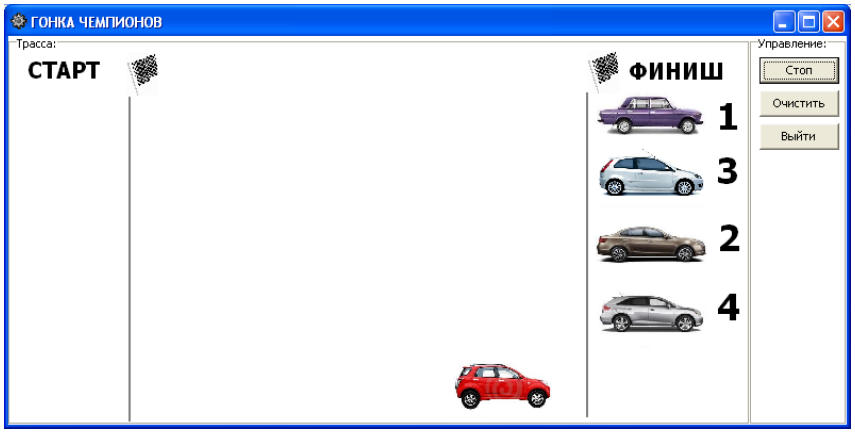



Рис. 7. Фрагменты работы программы «Имитатор гонки»

Контрольные вопросы

1. Перечислите основные функции для работы с потоками?
2. Возможно, ли создать поток в приостановленном состоянии (не запускается на исполнение)?
3. Перечислите способы завершения потока. Какой из них является наиболее безопасным?
4. Какой относительный приоритет получает поток при создании?
5. Позволяет ли функция *CreateThread* задать относительный приоритет потока?
6. Перечислите способы приостановки выполнения функции потока?
7. Перечислите основные базовые классы приоритетов потоков?
8. Приведите пример функции изменения приоритетов потоков?

## ЛАБОРАТОРНАЯ РАБОТА №4.
## СРЕДСТВА МЕЖПРОЦЕССНОГО ВЗАИМОДЕЙСТВИЯ – КАНАЛЫ (PIPE)

***Цель работы***

Изучение механизмов межпроцессного взаимодействия (*InterProcess Communication*) в *Windows*, получение практических навыков по использованию *Win32API* для программирования механизмов *IPC*.

***Информация***

К механизмам межпроцессного обмена относятся [2]:
файлы, проецируемые в память (*file mapping*);
почтовые ящики (*mailslot*).

Почтовые ящики обеспечивают только однонаправленные соединения. Каждый процесс, который создает почтовый ящик, является «сервером почтовых ящиков» (*mailslot server*). Другие процессы, называемые «клиентами почтовых ящиков» (*mailslot clients*), посылают сообщения серверу, записывая их в почтовый ящик. Входящие сообщения всегда дописываются в почтовый ящик и сохраняются до тех пор, пока сервер их не прочтет. Каждый процесс может одновременно быть и сервером, и клиентом почтовых ящиков, создавая, таким образом, двунаправленные коммуникации между процессами.

Клиент может посылать сообщения на почтовый ящик, расположенный на том же компьютере, на компьютере в сети, или на все почтовые ящики с одним именем всем компьютерам выбранного домена. При этом широковещательное сообщение, транслируемое по домену, не может быть более 400 байт. В остальных случаях размер сообщения ограничивается только при создании почтового ящика сервером.

Почтовые ящики предлагают легкий путь для обмена короткими сообщениями, позволяя при этом вести передачу и по локальной сети, в том числе и по всему домену.

*Mailslot* является псевдофайлом, находящимся в памяти, и следует использовать стандартные функции для работы с файлами, чтобы получить доступ к нему. Данные в почтовом ящике могут быть в любой форме – их интерпретацией занимается прикладная программа, но их общий объем не должен превышать *64 Кб*. Однако, в отличие от дисковых файлов, *mailslot* являются временными – когда все дескрипторы почтового ящика закрыты, он и все его данные удаляются. Все почтовые ящики являются локальными по отношению к

создавшему их процессу; процесс не может создать удаленный *mailslot*.

Сообщения меньше чем *425 байт* передаются с использованием дейтаграмм. Сообщения, больше чем *426 байт*, используют передачу с установлением логического соединения на основе *SMB*-сеансов. Передачи с установлением соединения допускают только индивидуальную передачу от одного клиента к одному серверу. Следовательно, теряется возможность широковещательной трансляции сообщений от одного клиента ко многим серверам. Windows не поддерживает сообщения размером в *425* или *426 байт*.

Когда процесс создает почтовый ящик, имя последнего должно иметь следующую форму:

\\.*mailslot**[path]name*

Например:

\\.*mailslot**taxes**bobs_comments*
\\.*mailslot**taxes**petes_comments*
\\.*mailslot**taxes**sues_comments*

Если необходимо отправить сообщение в почтовый ящик на удаленный компьютер, то следует воспользоваться NETBIOS-именем:

*ComputerName**mailslot**[path]name*

Чтобы передать сообщение всем mailslot с указанным именем внутри домена, понадобится NETBIOS-имя домена:

*DomainName**mailslot**[path]name*

Для главного домена операционной системы (домен, в котором находится рабочая станция):

**mailslot**[path]name*

Клиенты и серверы, использующие почтовые ящики, при работе с ними должны пользоваться функциями, представленными в табл. 11.

Таблица 11

Функции почтовых ящиков

| Функция | Описание |
|---|---|
| *Серверов* | |
| CreateMailslot | Создает почтовый ящик и возвращает его дескриптор |
| GetMailslotInfo | Извлекает максимальный размер сообщения, размер почтового ящика, размер следующего сообщения в ящике, количество сообщений и время ожидания сообщения при выполнении операции чтения |
| SetMailslotInfo | Изменение таймаута при чтении из почтового ящика |
| DuplicateHandle | Дублирование дескриптора почтового ящика |
| ReadFile<br>ReadFileEx | Считывание сообщений из почтового ящика |
| GetFileTime | Получение даты и времени создания почтового ящика |
| SetFileTime | Установка даты и времени создания, модификации почтового ящика |
| GetHandleInformation | Получение свойств дескриптора почтового ящика |
| SetHandleInformation | Установка свойств дескриптора почтового ящика |

| Функция | Описание |
|---|---|
| *Клиентов* | |
| CreateFile | Создает дескриптор почтового ящика для клиентского процесса |
| DuplicateHandle | Дублирование дескриптора почтового ящика |
| WriteFile<br>WriteFileEx | Запись сообщений в почтовый ящик |
| CloseHandle | Закрывает дескриптор почтового ящика для клиентского процесса |

Рассмотрим последовательно все операции, необходимые для корректной работы с почтовыми ящиками.

1. Создание почтового ящика. Операция выполняется процессом сервера с использованием функции *CreateMailslot*:

```
HANDLE CreateMailslot(
LPCTSTR lpName,             // Имяпочтовогоящика
DWORD nMaxMessageSize, // Максимальныйразмерсообщения
DWORD lReadTimeout,       // Таймаутоперациичтения
LPSECURITY_ATTRIBUTES   // Опции наследования и
lpSecurityAttributes     // безопасности
);
```

2. Запись сообщений в почтовый ящик производится аналогично записи в стандартный дисковый файл с помощью функции *WriteFile*.

3. Чтение сообщений из почтового ящика. Создавший почтовый ящик процесс получает право считывания сообщений, из него используя дескриптор *mailslot* в вызове функции *ReadFile*.

Почтовый ящик существует до тех пор, пока не вызвана функция *CloseHandle* на сервере или пока существует сам процесс сервера. В обоих случаях все непрочитанные сообщения удаляются из почтового ящика, уничтожаются все клиентские дескрипторы, и *mailslot* удаляется из памяти.

Функция считывает параметры почтового ящика:

```
BOOLGetMailslotInfo(
HANDLEhMailslot,            // Дескриптор почтового ящика.
LPDWORDlpMaxMessageSize,// Максимальный размер сообщения.
LPDWORDlpNextSize,          // Размер следующего
                            // непрочитанного сообщения.
LPDWORDlpMessageCount,   // Количество сообщений.
LPDWORDlpReadTimeout       // Таймаут операции чтения.
);
```

Функция устанавливает таймаут операции чтения:

```
BOOLSetMailslotInfo(
HANDLEhMailslot,          // Дескриптор почтового ящика.
DWORDlReadTimeout         // Новый таймаут операции чтения.
);
```

*Каналы (pipe)*

Существует два способа организовать двунаправленное соединение с помощью следующих типов каналов [4]:

1. *Безымянные* (*анонимные*) *каналы* позволяют связанным процессам передавать информацию друг другу. Обычно, безымянные каналы используются для перенаправления стандартного ввода/вывода дочернего процесса так, чтобы он мог обмениваться данными с

родительским процессом. Чтобы производить обмен данными в обоих направлениях, следует создать два безымянных канала. Родительский процесс записывает данные в первый канал, используя его дескриптор записи, в то время как дочерний процесс считывает данные из канала, используя дескриптор чтения. Аналогично, дочерний процесс записывает данные во второй канал и родительский процесс считывает из него данные. Безымянные каналы не могут быть использованы для передачи данных по сети и для обмена между несвязанными процессами.

2. *Именованныеканалы* используются для передачи данных между независимыми процессами или между процессами, работающими на разных компьютерах. Обычно, процесс сервера именованных каналов создает именованный канал с известным именем или с именем, которое будет передано клиентам. Процесс клиента именованных каналов, зная имя созданного канала, открывает его на своей стороне с учетом ограничений, указанных процессом сервера. После этого между сервером и клиентом создается соединение, по которому может производиться обмен данными в обоих направлениях. В организации межпроцессного обмена наибольший интерес представляют именованные каналы.

Общие принципы работы именованных и неименованных каналов:

1. При чтении меньшего числа байт, чем находится в канале, возвращается требуемое число байт, остаток сохраняется для последующих чтений.
2. При чтении большего числа байт, чем находится в канале, возвращается доступное число байт. Процесс, читающий из канала, должен эту ситуацию отработать.
3. Если канал пуст и ни один процесс не открыл его на запись, при чтении из канала будет получено *0* байт. Если один или более процессов открыли канал для записи, вызов на чтение будет заблокирован до появления данных в канале.
4. Запись числа байт, меньшего емкости канала, гарантирована атомарно. В случае, когда несколько процессов одновременно записывают в канал, порции данных от них не перемешиваются.
5. При записи большего числа байт, чем это позволяет канал, вызов на запись блокируется до освобождения требуемого места в канале. Атомарность при этом не гарантируется.

При создании и получении доступа к существующему каналу необходимо придерживаться следующего стандарта имен каналов:

\\.*pipe**pipename*

Если канал находится на удаленном компьютере, то потребуется NETBIOS-имя компьютера:

*ComputerName**pipe**pipename*

Клиентам и серверам для работы с каналами допускается использовать функции, представленные в табл. 12.

Кроме того, для работы с каналами используется функция *CreateFile* (для подключения к каналу со стороны клиента) и функции *WriteFile* и *ReadFile* для записи и чтения данных в/из канала соответственно.

Таблица 12

Функции работы с каналами

| Функция | Описание |
|---|---|
| CallNamedPipe | Выполняет подключение к каналу, записывает в канал сообщение, считывает из канала сообщение и затем закрывает канал |
| ConnectNamedPipe | Позволяет серверу именованных каналов ожидать подключения одного или нескольких клиентских процессов к экземпляру именованного канала |
| CreateNamedPipe | Создает экземпляр именованного канала и возвращает |

| Функция | Описание |
|---|---|
| | дескриптор для последующих операций с каналом |
| CreatePipe | Создает безымянный канал |
| DisconnectNamedPipe | Отсоединяет серверную сторону экземпляра именованного канала от клиентского процесса |
| GetNamedPipeHandleState | Получает информацию о работе указанного именованного канала |
| GetNamedPipeInfo | Извлекает свойства указанного именованного канала |
| PeekNamedPipe | Копирует данные их именованного или безымянного канала в буфер без удаления их из канала |
| SetNamedPipeHandleState | Устанавливает режим чтения и режим блокировки вызова функций (синхронный или асинхронный) для указанного именованного канала |
| TransactNamedPipe | Комбинирует операции записи сообщения в канал и чтения сообщения из канала в одну сетевую транзакцию |
| WaitNamedPipe | Ожидает, пока истечет время ожидания или пока экземпляр указанного именованного канала не будет доступен для подключения к нему |

Пример реализации межпроцессного взаимодействия (клиент-серверного приложения) [5] представлен в приложении. Результат работы программ межпроцессного взаимодействия приведен на рис. 8.

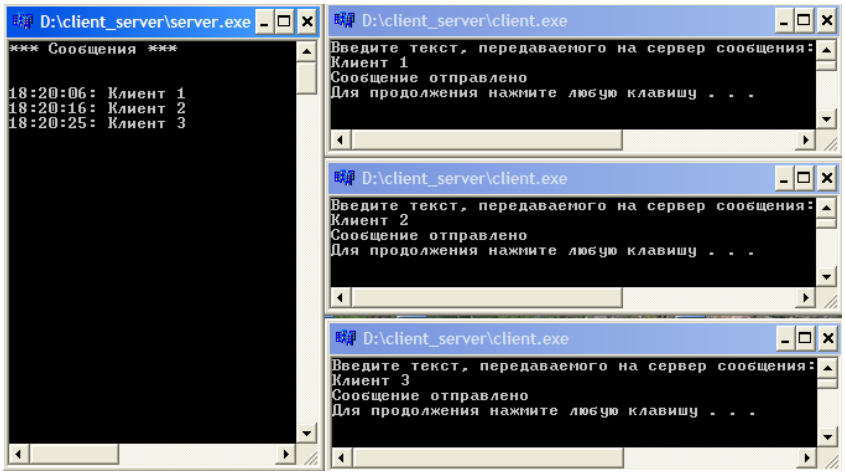



Рис. 8. Результат работы программ межпроцессного взаимодействия

Задание для самостоятельного решения

Организовать работу программы-сервера и нескольких программ-клиентов следующим образом.

Сервер предоставляет клиентам какой-либо из своих ресурсов (например, собственное окно), причем сервер может быть запущен только один.

Клиенты подключаются к серверу и начинают запись в окно, причем первый клиент пишет только «*1*», второй – только «*2*», и т.д. в каждый момент времени. Клиентов может быть произвольное количество, но не менее пяти. Предусмотреть возможность отправки на сервер произвольного сообщения.

Если клиент подключается к серверу в монопольном режиме, он получает исключительные права на использование ресурса сервера. Все остальные клиенты, пытающиеся подключиться в данный момент, не должны получить доступ к ресурсу сервера и должны оказаться в очереди на обслуживание.

В разделяемом режиме каждому из подключенных клиентов предоставляется квант времени на исполнение (например, *1с*). Если клиент записывает символы в окно сервера с частотой *1* символ в секунду, то в случае, когда к серверу подключены пять клиентов, окно сервера должно содержать примерно следующую информацию, представленную на рис. 9.

Обмен данными между клиентами и сервером организовать при помощи именованных каналов.

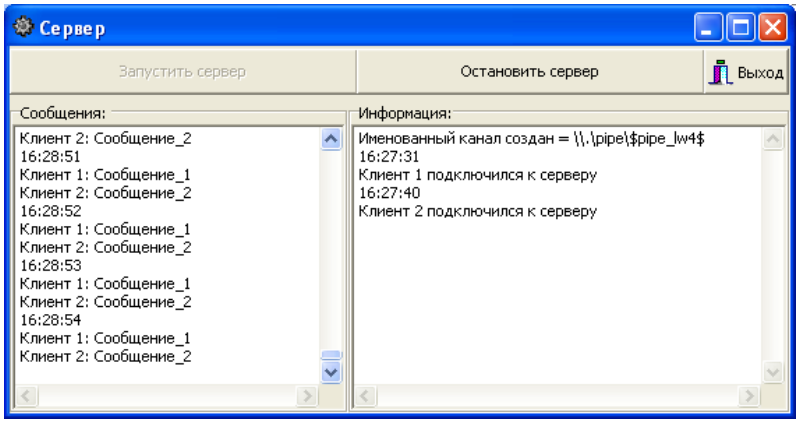



Рис. 9а. Результат работы программы-сервера

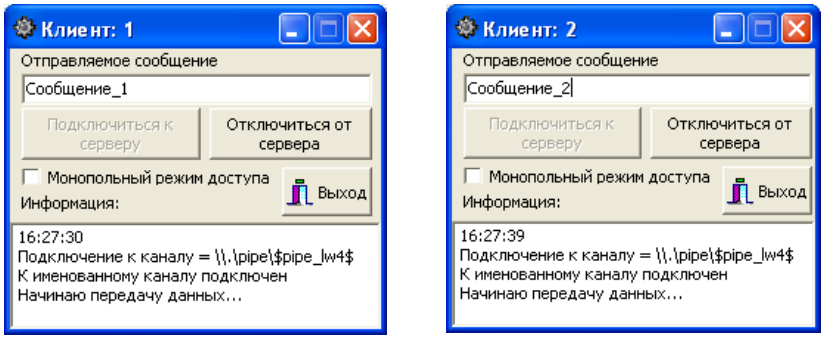



Рис. 9б. Результат работы программ-клиентов

Контрольные вопросы

1. Перечислите средства межпроцессного взаимодействия?
2. Для решения каких задач могут применяться почтовые ящики (*mailslot*)?
3. Возможно, ли на основе почтовых ящиков организовать двунаправленные коммуникации между процессами?
4. Какому стандарту имен почтовых ящиков необходимо придерживаться при создании и получении доступа к существующему почтовому ящику?
5. Перечислите основные функции используемые процессом – «сервером почтовых ящиков»?
6. Перечислите основные функции используемые процессами – «клиентами почтовых ящиков»?
7. Приведите классификацию каналов (*pipe*)?
8. Как можно организовать параллельные вычисления при помощи именованных каналов?
9. Какому стандарту имен каналов необходимо придерживаться при создании и получении доступа к существующему каналу?
10. Приведите основные функции для работы с каналами?

## ЛАБОРАТОРНАЯ РАБОТА №5.
## ФАЙЛЫ ДАННЫХ, ПРОЕЦИРУЕМЫЕ В ПАМЯТЬ

***Цель работы***

Изучить принципы работы с файлами, проецируемыми в память.

***Информация***

Как и виртуальная память, проецируемые файлы позволяют резервировать регион адресного пространства и передавать ему физическую память.

Операционная система позволяет проецировать на адресное пространство процесса и файл данных. Это очень удобно при манипуляциях с большими потоками данных [3].

Проецируемые файлы применяются для:

загрузки и выполнения EXE- и DLL-файлов. Это позволяет существенно экономить как на размере страничного файла, так и на времени, необходимом для подготовки приложения к выполнению;

доступа к файлу данных, размещенному на диске. Это позволяет обойтись без операций файлового ввода-вывода и буферизации его содержимого;

разделения данных между несколькими процессами, выполняемыми на одной машине.

Чтобы представить всю мощь такого применения механизма проецирования файлов, рассмотрим четыре возможных метода реализации программы, меняющей порядок следования всех байтов в файле на обратный.

*Метод 1: один файл, один буфер*

Первый (и теоретически простейший) метод – выделение блока памяти, достаточного для размещения всего файла. Открываем файл, считываем его содержимое в блок памяти, закрываем. Располагая в памяти содержимым файла, можно поменять первый байт с последним, второй – с предпоследним и т.д. Этот процесс будет продолжаться, пока мы не поменяем местами два смежных байта, находящихся в середине файла. Закончив эту операцию, вновь открываем файл и перезаписываем его содержимое.

*Метод 2: два файла, один буфер*

Открываем существующий файл и создаем на диске новый – нулевой длины. Затем выделяем небольшой внутренний буфер размером, скажем, *8 Кб*. Устанавливаем указатель файла в позицию *8 Кб* от конца, считываем в буфер последние *8 Кб* содержимого файла, меняем в нем порядок следования байтов на обратный и переписываем буфер в только что созданный файл. Повторяем эти операции, пока не дойдем до начала исходного файла. Конечно, если длина файла не будет кратна *8 Кб*, операции придется немного усложнить. Закончив обработку, закрываем оба файла и удаляем исходный файл.

*Метод 3: один файл, два буфера*

Программа инициализирует два раздельных буфера, допустим, по *8 Кб* и считываем первые *8 Кб* файла в один буфер, а последние *8 Кб* – в другой. Далее содержимое обоих буферов обмениваются в обратном порядке, и первый буфер записывается в конец, а второй – в начало того же файла. На каждой итерации программа перемещает восьмикилобитные блоки из одной половины файла в другую. В данном методе следует предусмотреть обработку на случай, если длина файла не кратна *16 Кб*.

*Метод 4: один файл и никаких буферов*

Открывается файл с указанием системе зарезервировать регион виртуального адресного пространства. Далее, первый байт файла проецируется на первый байт этого региона и производится обращение к региону так, будто он на самом деле содержит файл. Если в конце файла есть отдельный нулевой байт, можно вызвать библиотечную функцию *_strrev* и поменять порядок следования байтов на обратный.

*Использование проецируемых в память файлов*

Для этого нужно выполнить три операции:

1. Создать или открыть объект ядра «файл», идентифицирующий дисковый файл, который используется как проецируемый в память.
2. Создать объект ядра «проекция файла», сообщив системе размер файла и способ доступа к нему.
3. Указать системе, как спроецировать в адресное пространство процесса объект «проекция файла» - целиком или частично.

Закончив работу с проецируемым в память файлом, следует выполнить следующие операции:

1. Сообщить системе об отмене проецирования на адресное пространство процесса объекта ядра «проекция файла».
2. Закрыть этот объект.
3. Закрыть объект ядра «файл».

*Этап 1: создание или открытие объекта ядра «файл»*

Для создания и открытия объекта ядра «файл» используется функция *CreateFile:*

*HANDLE CreateFile(*

*PCSTR lpFileName,*
*DWORD dwDesiredAccess,*
*DWORD dwShareMode,*
*PSECURITY_ATTRIBUTES lpSecurityAttributes,*
*DWORD dwCreationDisposition,*
*DWORD dwFlagsAndAttributes,*
*HANDLE hTemplateFile);*

Рассмотримтрипервыхпараметра*lpFileName, dwDesiredAccess*и*dwShareMode*.

Параметр *lpFileName* идентифицирует имя создаваемого или открываемого файла (при необходимости вместе с путем). Второй параметр, *dwDesiredAccess*, указывает способ доступа к содержимому файла. Здесь задается одно из четырех значений (табл. 13):

Таблица 13

Значение параметра *dwDesiredAccess*

| Значение | Описание |
|---|---|
| 0 | Содержимое файла нельзя считывать или записывать; указывается это значение, если требуется получить атрибуты файла |
| GENERIC_READ | Чтение файла разрешено |
| GENERIC_WRITE | Запись в файл разрешена |
| GENERIC_READ\| GENERIC_WRITE | Разрешено и чтение и запись |

Создавая или открывая файл данных, используемый в качестве проецируемого в память, устанавливается флаг GENERIC_READ (только для чтения), либо комбинированный флаг GENERIC_READ | GENERIC_WRITE (чтение/запись).

Третий параметр, *dwShareMode*, указывает тип совместного доступа к данному файлу (табл. 14).

Таблица 14

Значение параметра *dwShareMode*

| Значение | Описание |
|---|---|
| 0 | Другие попытки открыть файл закончатся неудачно |
| FILE_SHARE_READ | Попытка постороннего процесса открыть файл с флагом GENERIC_WRITE не удастся |
| FILE_SHARE_WRITE | Попытка постороннего процесса открыть файл с флагом GENERIC_READ не удастся |
| FILE_SHARE_READ \| FILE_SHARE_WRITE | Посторонний процесс может открыть файл без ограничений |

Создав или открыв указанный файл, *CreateFile* возвращает его описатель, в ином случае – идентификатор INVALID_HANDLE_VALUE, определенный как ((HANDLE) -1).

*Этап 2: создание объекта ядра «проекция файла»*

Указав операционной системе, где находится физическая память для проекции файла: на жестком диске, в сети, на CD-ROM или в другом месте (вызов функции *CreateFile*), необходимо указать системе какой объем физической памяти нужен проекции файла. Дляэтогонеобходимовызватьфункцию*CreateFileMapping*:

*HANDLE CreateFileMapping(*
*HANDLE hFile,*
*LPSECURITY_ATTRIBUTES lpFileMappingAttributes,*
*DWORD flProtect,*
*DWORD dwMaximumSizeHigh,*
*DWORD dwMaximumSizeLow,*
*LPCTSTR lpName);*

Первый параметр, *hFile,* идентифицирует описатель файла, проецируемого на адресное пространство процесса. Этот описатель получили после вызова функции *CreateFile*. Параметр *lpFileMappingAttributes* – указатель на структуру SECURITY_ATTRIBUTES, которая относится к объекту ядра «проекция файла»; для установки защиты по умолчанию ему присваивается NULL.

Создание файла, проецируемого в память, аналогично резервированию региона адресного пространства с последующей передачей ему физической памяти. Разница лишь в том, что физическая память для проецируемого файла – сам файл на диске, и для него не нужно выделять пространство в страничном файле. При создании объекта «проекция файла» система не резервирует регион адресного пространства и не увязывает его с физической памятью из файла. Но, как только дело дойдет до отображения физической памяти на адресное пространство процесса, системе понадобится точно знать атрибут защиты, присваиваемый страницам физической памяти. Поэтому в параметре *flProtect* надо указать желательные атрибуты защиты (табл. 15).

Таблица 15

Значение параметра *flProtect*

| Атрибут защиты | Описание |
|---|---|
| PAGE_READONLY | Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла. При этом необходимо было передать в функцию *CreateFile*флаг GENERIC_READ. |
| PAGE_READWRITE | Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла и записывать их. При этом необходимо было передать в функцию *CreateFile*комбинациюфлагов GENERIC_READ \| GENERIC_WRITE. |
| PAGE_WRITECOPY | Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла и записывать их. Запись приведет к созданию закрытой копии страницы. При этом необходимо было передать в функцию *CreateFile*либо GENERIC_READ, либо GENERIC_READ \| GENERIC_WRITE. |

Кроме рассмотренных выше атрибутов защиты страницы, существует еще четыре атрибута раздела: SEC_NOCACHE, SEC_IMAGE, SEC_RESERVE и SEC_COMMIT.

Следующие два параметра этой функции (*dwMaximumSizeHigh* и *dwMaximumSizeLow*) самые важные. Основное назначение *CreateFileMapping* – гарантировать, что объекту «проекция файла» доступен нужный объем физической памяти. Через эти параметры сообщается системе максимальный размер файла в байтах. Так как Windows позволяет работать с файлами, размеры которых выражаются 64-разрядными числами, в параметре *dwMaximumSizeHigh* указываются старшие *32бита*, а в*dwMaximumSizeLow* – младшие *32бита* этого значения. Для файлов размером менее *4 Гбdw*MaximumSizeHigh всегда равен *0.* Наличие 64-разрядного значения подразумевает, что Windows способна обрабатывать файлы длиной до *16* экзабайтов.

Для создания объекта «проекция файла» таким, чтобы он отражал текущий размер файла, необходимо передавать в обоих параметрах нули. Так же следует поступить, если необходимо ограничиться считыванием или как-то обработать файл, не меняя его размер. Для дозаписи данных в файл выбирается его размер максимальным, чтобы оставить пространство для «маневра». Если в данный момент файл на диске имеет нулевую длину, в параметрах *dwMaximumSizeHigh* и *dwMaximumSizeLow* нельзя передавать нули. Иначе система решит, что необходима проекция файла с объемом памяти, равным *0*. А это ошибка, и *CreateFileMapping* вернет NULL.

Последний параметр функции *CreateFileMapping – lpName* – строка с нулевым байтов в конце; в ней указывается имя объекта «проекция файла», которое используется для

доступа к данному объекту из другого процесса. В случае если совместное использование проецируемого в память файла не требуется, в данном параметре передают NULL.

Чтобы получить доступ к существующему объекту ядра «проекция файла» необходимо вызвать функцию *OpenFileMapping* с указанием операций, которые будут проводиться над объектом:

*HANDLEOpenFileMapping(*
*DWORD dwDesiredAccess,*
*BOOL bInheritHandle,*
*LPCTSTR lpName);*

Первый параметр, *dwDesiredAccess,* идентифицирует вид доступа к данным. Необходимо указать, как именно мы хотим обращаться к файловым данным, задавая одно из четырех значений (табл. 16).

Таблица 16

Значение параметра *dwDesiredAccess*

| Значение | Описание |
|---|---|
| FILE_MAP_WRITE | Файловые данные можно считывать и записывать; при этом в функцию *CreateFileMapping* должен быть передан атрибут PAGE_READWRITE. |
| FILE_MAP_READ | Файловые данные можно только считывать; при этом в функцию *CreateFileMapping* должен быть передан любой из следующих атрибутов PAGE_READONLY, PAGE_READWRITE или PAGE_WRITECOPY. |
| FILE_MAP_ALL_ACCESS | То же, что и FILE_MAP_WRITE. |
| FILE_MAP_COPY | Файловые данные можно считывать и записывать, но запись приводит к созданию закрытой копии страницы; при этом в функцию *CreateFileMapping* должен быть передан любой из следующих атрибутов PAGE_READONLY, PAGE_READWRITE или PAGE_WRITECOPY. |

Второй параметр, bInheritHandle, указывает, наследовал ли новый процесс дескрипторы от процесса запроса. Если TRUE, каждый наследственный открытый дескриптор в процессе запроса унаследован новым процессом.

Последний параметр функции *OpenFileMapping – lpName –* строка с нулевым байтом в конце; в ней указывается имя объекта «проекция файла», которое используется для доступа к объекту из данного процесса.

Функция *OpenFileMapping,* прежде чем вернуть действительный описатель, проверяет тип защиты объекта. Если есть доступ к существующему объекту ядра «проекция файла», функция *OpenFileMapping* возвращает действительный описатель. Но если отказано в доступе, функция *OpenFileMapping* возвращает NULL, а вызов *GetLastError* дает код ошибки 5 (или ERROR_ACCESS_DENIED).

*Этап 3: проецирование файловых данных на адресное пространство процесса*

Когда объект «проекция файла» создан, необходимо, чтобы система, зарезервировав регион адресного пространства под данные файла, передала их как физическую память, отображенную на регион. Этоделаетфункция*MapViewOfFile*:

*LPVOID MapViewOfFile(*
*HANDLE hFileMappingObject,*
*DWORD dwDesiredAccess,*
*DWORD dwFileOffsetHigh,*
*DWORD dwFileOffsetLow,*
*DWORD dwNumberOfBytesToMap);*

Параметр*hFileMappingObject*идентифицируетописательобъекта «проекцияфайла», возвращаемыйпредшествующимвызовомлибофункцией*CreateFileMapping*, либофункцией*OpenFileMapping*. Параметр *dwDesiredAccess* идентифицирует вид доступа к данным (табл. 16).

Остальные три параметра относятся к резервированию региона адресного пространства и к отображению на него физической памяти. При этом необязательно проецировать на адресное пространство весь файл сразу. Можно спроецировать лишь малую его часть, которая в таком случае называется представлением (*view*).

Проецируя на адресное пространство процесса представление файла, необходимо сделать две вещи. Во-первых, сообщить системе, какой байт файла данных считать в представлении первым. Для этого предназначены параметры *dwFileOffsetHigh* и *dwFileOffsetLow*. Во-вторых, потребуется указать размер представления, т.е. сколько байтов файла данных должно быть спроецировано на адресное пространство. Размер указывается в параметре *dwNumberOfBytesToMap*. Если этот параметр равен *0*, система попытается спроецировать представление, начиная с указанного смещения и до конца файла.

Если при вызове *MapViewOfFile* указан флаг FILE_MAP_COPY, система передаст физическую память из страничного файла. Размер передаваемого пространства определяется параметром *dwNumberOfBytesToMap*. Пока данные считываются из представления файла, страницы, переданные из страничного файла, не используются. Но стоит какому-нибудь потоку в процессе совершить попытку записи по адресу, попадающему в границы представления файла, как система тут же берет из страничного файла одну из перечисленных страниц, копирует на нее исходные данные и проецирует ее на адресное пространство процесса. С этого момента потоки процесса начинают обращаться к локальной копии данных и теряют доступ к исходным данным. Создав копию исходной страницы, система меняет ее атрибут защиты с PAGE_WRITECOPY на PAGE_READWRITE.

*Этап 4: отключение файла данных от адресного пространства процесса*

Когда необходимость в данных файла (спроецированного на регион адресного пространства) отпадает, требуется освободить регион вызовом функции *UnmapViewOfFile*:

*BOOLUnmapViewOfFile(LPCVOIDlpBaseAddress);*

Параметр, *lpBaseAddress*, указывает базовый адрес возвращаемого системе региона. Он должен совпадать со значением, полученным после вызова функции *MapViewOfFile*. Если не вызвать функцию *MapViewOfFile* регион не освободится до завершения процесса. Повторный вызов *MapViewOfFile* приводит к резервированию нового региона в пределах адресного пространства процесса, но ранее выделенные регионы *не освобождаются*.

У функции *UnmapViewOfFile* есть одна особенность. Если первоначально представление было спроецировано с флагом FILE_MAP_COPY, любые изменения, внесенные в файловые данные, на самом деле производятся над копией этих данных, хранящихся в страничном файле. Вызванной в этом случае функции *UnmapViewOfFile* нечего обновлять в дисковом файле, и она просто инициирует возврат системе страниц физической памяти, выделенных из страничного файла. Все изменения в данных на этих страницах теряются. Поэтому о сохранении измененных данных придется заботиться самостоятельно.

*Этапы 5 и 6: закрытие объектов «проекция файлов» и «файл»*

Закончив работу с любым открытым объектом ядра, необходимо его закрыть, иначе в процессе начнется утечка ресурсов. Для закрытия объектов «проекция файлов» и «файл» требуется дважды вызвать функцию *CloseHandle*.

При операциях с проецируемыми файлами обычно открывают файл, создают объект «проекция файла» и с его помощью проецируют представление файловых данных на адресное пространство процесса. Поскольку система увеличивает внутренние счетчики объектов «файл» и «проекция файла», их можно закрыть в начале кода, тем самым,

исключив возможность утечки ресурсов.

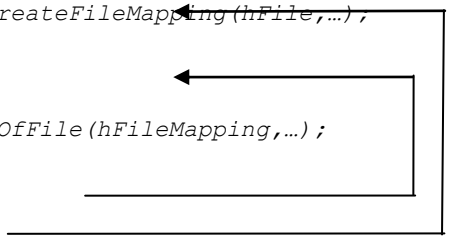
Если требуется создавать из одного файла несколько объектов «проекция файла» или проецировать несколько представлений этого объекта, применить функцию *CloseHandle* в начале кода не удастся – описатели еще понадобятся для дополнительных вызовов *CreateFileMapping* и *MapViewOfFile*.

Рассмотрим это подробнее на фрагменте псевдокода:

```
HANDLE hFile = CreateFile(…);

HANDLE hFileMapping = CreateFileMapping(hFile,…);


LPVOID pvFile = MapViewOfFile(hFileMapping,…);
```



*Пример.* Создаем файл “*File-mapping.txt”*, пишем в него строчку “*The named or unnamed file-mapping object.”*, закрываем, открываем для чтения, проецируем в память и копируем строку из спроецированного региона памяти.

```
#include <windows.h>
#include <iostream.h>

int main()
{
   PDWORD lpNumberOfBytesWritten=new DWORD;
   char str[]="The named or unnamed file-mapping object.";
   HANDLE     MyFile=CreateFile("File-mapping.txt",
                                GENERIC_WRITE,
                                FILE_SHARE_WRITE, NULL,
                                CREATE_ALWAYS,
                                FILE_ATTRIBUTE_NORMAL,0);
   WriteFile(MyFile,str,strlen(str),
         lpNumberOfBytesWritten,NULL);
   SetEndOfFile(MyFile);
   CloseHandle(MyFile);
   MyFile=CreateFile("File-mapping.txt", GENERIC_READ,
                     FILE_SHARE_READ,NULL,OPEN_EXISTING,
                     FILE_ATTRIBUTE_NORMAL,0);
   HANDLE MappedFile=CreateFileMapping(MyFile,NULL,
                                       PAGE_READONLY,0,0,
                                       NULL);
   LPVOID Map=MapViewOfFile(MappedFile,FILE_MAP_READ,
                            0,0,0);
   char * strin =(char *) Map;
   cout << strin << endl;
   UnmapViewOfFile(Map);
   CloseHandle(MappedFile);
   CloseHandle(MyFile);
return 0;
}
```

Задание для самостоятельного решения

*Требуется создать программу «Писатель» и программу «Читатель». Запустить программу «Писатель» (обеспечить запуск только одного экземпляра программы) и несколько экземпляров программы «Читатель». Программа «Писатель» постоянно обновляет содержимое некоторого файла (физический файл на диске) и проецируют его на собственное адресное пространство (рис. 10). Программы «Читатель» получают*

*доступ к объекту ядра «проекция файла», созданному программой «Писатель». При обновлении файла происходит автоматическое обновление содержимого файла в окнах программ «Читатель» (рис. 11).*

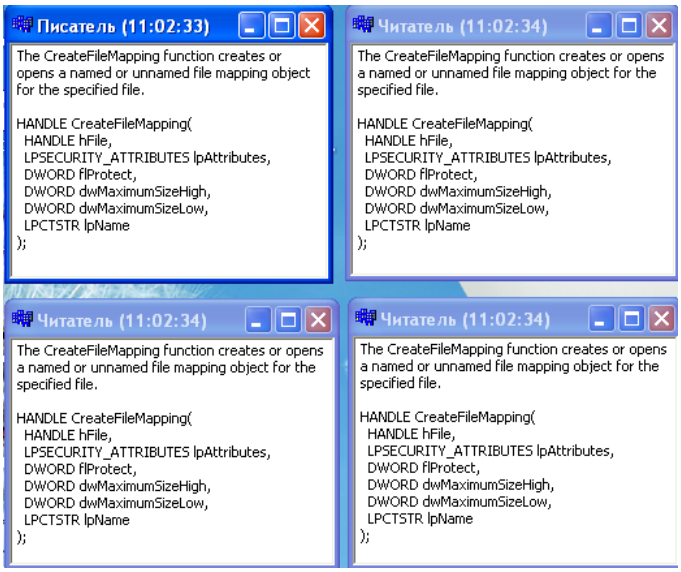



Рис. 10. Результаты работы программ

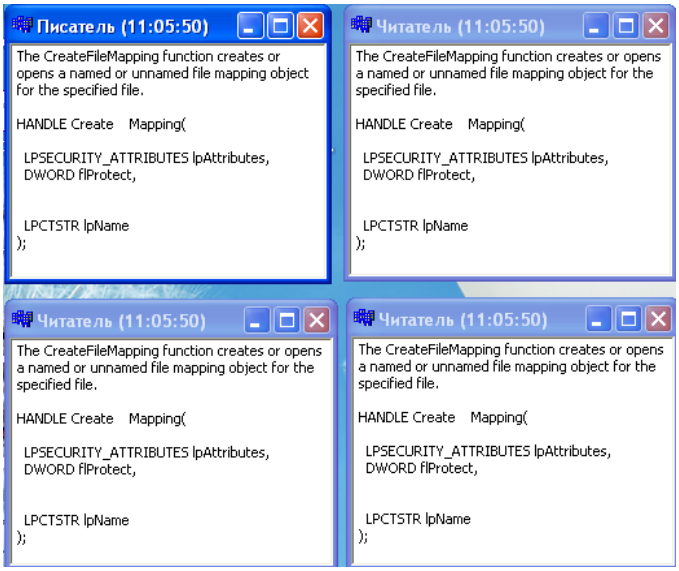



Рис. 11. Результаты работы программ
(при изменении информации писателем)

Контрольные вопросы

1. Перечислите средства межпроцессного взаимодействия?
2. Перечислите области применения файлов, проецируемых в память?
3. Как можно осуществить межпроцессное взаимодействие, используя файлы, проецируемые в память?
4. Перечислите основные операции при работе с файлами, проецируемыми в память?
5. Перечислите основные функции при работе с файлами, проецируемыми в память?
6. Перечислите основные способы доступа к содержимому файла?

**Приложение 2**

**Оценочные средства для проведения промежуточной аттестации**

**а) Планируемые результаты обучения и оценочные средства для проведения промежуточной аттестации:**

| Код индикатора | Индикатор достижения компетенции | Оценочные средства |
|---|---|---|
| ПК-6: Способность к формализации и алгоритмизации поставленных задач, к написанию программного кода с использованием языков программирования, определения и манипулирования данными и оформлению программного кода в соответствии установленными требованиями | | |
| ПК-6.1 | Оценивает качество математической модели при формализации задачи предметной области | Основное различие между долгосрочным и краткосрочным планированием (диспетчеризацией) заключается в …<br>1) длительности выполнения<br>2) скорости выполнения<br>3) очередности выполнения очередности выполнения<br>4) частоте выполнения |
| | | Приоритет, меняющейся во время исполнения процесса, называется _____ приоритетом<br>1) фиксированным<br>2) статическим<br>3) циклическим<br>4) динамическим |
| | | При совместном использовании процессами аппаратных и информационных ресурсов вычислительной системы возникает потребность в …<br>1) адаптации<br>2) синхронизации<br>3) буферизации<br>4) оптимизации |
| ПК-6.2 | Оценивает качество разработанных алгоритмов для последующего кодирования. | Главной целью мультипрограммирования в системах пакетной обработки является …<br>1) обеспечение удобства работы пользователей<br>2) минимизация времени выполнения одной задачи<br>3) минимизация простоев всех устройств компьютера<br>4) обеспечение реактивности системы |
| | | Использование виртуальной памяти в однопрограммном режиме приводит к … процесса, если размер программы существенно больше объема доступной оперативной памяти<br>1) аварийному завершению<br>2) замедлению выполнения<br>3) ускорению<br>4) перезапуску |
| | | При страничной организации памяти таблица страниц может размещаться в …<br>1) только в оперативной памяти<br>2) только в процессоре<br>3) в оперативной памяти и на диске<br>4) в специальной быстрой памяти процессора и в оперативной памяти |
| ПК- 6.3 | Оценивает выбор программных средств для программирования и манипулирования данными в | В многопоточной системе при создании процесса ОС создает для каждого процесса<br>1) как минимум два потока выполнения<br>2) ни одного потока выполнения<br>3) как минимум один поток выполнения<br>4) только один поток выполнения |

| Код индикатора | Индикатор достижения компетенции | Оценочные средства |
|---|---|---|
| | соответствии установленными требованиями | В мультипрограммной смеси желательно одновременное присутствие<br>1) вычислительных задач и задач с интенсивным вводом-выводом<br>2) простых и сложных задач<br>3) задач управления и задач с интенсивным вводом-выводом<br>4) задач управления и вычислительных задач |
| | | Планирование потоков осуществляется на основе информации, хранящейся в …<br>1) описателях процессов и потоков<br>2) контекстах процессов<br>3) идентификаторах процессов<br>4) идентификаторах потоков |

**б) Порядок проведения промежуточной аттестации, показатели и критерии оценивания:**

Промежуточная аттестация по дисциплине «Теория вычислительных процессов» включает теоретические вопросы, позволяющие оценить уровень усвоения обучающимися знаний, и практические задания, выявляющие степень сформированности умений и владений, проводится в форме экзамена.

Экзамен по дисциплине проводится в устной форме.

**Показатели и критерии оценивания экзамена:**

– на оценку **«отлично» –** обучающийся показывает высокий уровень сформированности компетенций, т.е. полно раскрыто содержание материала; чётко и правильно даны определения и раскрыто содержание материала; ответ самостоятельный, при ответе использованы знания, приобретённые ранее;

– на оценку **«хорошо» –** обучающийся показывает средний уровень сформированности компетенций, т.е. раскрыто основное содержание материала в объёме; в основном правильно даны определения, понятия; материал изложен неполно, при ответе допущены неточности, нарушена последовательность изложения; допущены небольшие неточности при выводах и использовании терминов; практические навыки нетвёрдые;

– на оценку **«удовлетворительно» –** обучающийся показывает пороговый уровень сформированности компетенций, т.е. усвоено основное содержание материала, но изложено фрагментарно, не всегда последовательно; определения и понятия даны не чётко; практические навыки слабые;

– на оценку **«неудовлетворительно»** – результат обучения не достигнут, обучающийся не может показать знания на уровне воспроизведения и объяснения информации, не может показать интеллектуальные навыки решения простых задач.