Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ

Кафедра автоматизированных систем управления

В.Т. Калайда, В.В. Романенко

ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ

Методические указания по выполнению лабораторных работ

2013

Корректор: Осипова Е.А.

Калайда В.Т., Романенко В.В.
Теория вычислительных процессов: методические указания по выполнению лабораторных работ. — Томск: Факультет дистанционного обучения, ТУСУР, 2013. — 87 с.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Учебной программной в рамках изучения дисциплины «Теория вычислительных процессов» («ТВП») для студентов, обучающихся с применением дистанционных образовательных технологий, предусмотрено выполнение двух лабораторных работ:

1. Реализация алгоритмов планирования использования процессорного времени.

2. Реализация многопоточной обработки данных.

В данном методическом пособии изложены задания для лабораторных работ, в приложении — пример оформления титульного листа отчета по лабораторным работам.

Работы выполняются по вариантам. Выбор варианта лабораторных работ осуществляется по общим правилам с использованием следующей формулы:

$$V = (N*K) \text{ div } 100,$$

где V — искомый номер варианта,

N — общее количество вариантов,

div — целочисленное деление,

при V= 0 выбирается максимальный вариант,

K — значение 2-х последних цифр пароля.

Предполагается, что учащиеся хорошо владеют хотя бы одним высокоуровневым объектно-ориентированным языком программирования (C++, C# и т.д.) и имеют навыки работы хотя бы в одной современной визуальной среде разработки (C++ Builder, Microsoft Visual Studio, MonoDevelop и т.п.), а также имеют хорошую математическую подготовку.

Выполненная лабораторная работа должна включать:

- Архив (RAR или ZIP) с проектом для используемой среды разработки. В архиве должны находиться все исходные файлы, файлы проекта, а также файлы, необходимые для запуска проекта (исполняемый файл, файлы с входными данными и т.п.). Исходные файлы должны быть подробно комментированы. Временные файлы, создаваемые компилятором или отладчиком, в архив включать не нужно.
- Отчет по проделанной работе.
- Рецензия на предыдущий вариант работы, если она отправляется на проверку повторно после исправления замечаний.

1 ЛАБОРАТОРНАЯ РАБОТА № 1

Цель выполнения лабораторной работы № 1 — освоить реализацию алгоритмов планирования использования ресурсов с вытесняющей и невытесняющей многозадачностью, с абсолютным и относительным приоритетом. Освоить реализацию механизмов безопасности и синхронизации потоков, а также механизмов исключения тупиковых ситуаций.

1.1 Задание

В работе необходимо реализовать ряд алгоритмов распределения ресурсов между конкурирующими процессами. Каждый процесс характеризуется:

- уникальным идентификатором;
- приоритетом;
- временем CPU burst;
- списком требуемых ресурсов;
- дополнительными атрибутами (по вариантам).

Характеристики ресурса:

- уникальный идентификатор;
- наименование ресурса;
- дополнительные атрибуты (по вариантам).

Для имитации времени CPU burst (заданного в миллисекундах) процесс при получении кванта времени на доступ к ресурсу должен делать паузу на указанное количество миллисекунд.

Входной файл должен иметь имя «input.txt» или «input.xml». Формат входного файла представлен в табл. 1.1.

Таблица 1.1 — Формат входного файла для лабораторной работы № 1

Поле	Значение
PA	Выбранный способ планирования
QT	Продолжительность кванта времени (мс)
MaxT	Максимальное время CPU burst. Минимальное — 1 мс
MaxP	Максимальный приоритет потока. Минимальный — 1
NR	Количество ресурсов
…	Характеристики каждого ресурса (наименование и дополнительные атрибуты). Если какие-то характеристики не заданы (пустая

Окончание табл. 1.1

Поле	Значение
	строка), то генерируются программой случайным образом
NP	Количество процессов
…	Характеристики каждого процесса (приоритет, время выполнения, список требуемых ресурсов и дополнительные атрибуты). Если какие-то характеристики не заданы (пустая строка), то генерируются программой случайным образом

Выходной файл должен иметь имя «output.txt». Формат выходного файла представлен в табл. 1.2.

Таблица 1.2 — Формат выходного файла для лабораторной работы № 1

Поле	Значение
NR	Количество ресурсов
…	Характеристики каждого ресурса, если они были сгенерированы случайным образом
NP	Количество процессов
…	Характеристики каждого процесса, если они были сгенерированы случайным образом
T	Общее время выполнения всех потоков. В случае возникновения тупиковой ситуации это будет слово «deadlock»
0…00	Строка, соответствующая состоянию системы после завершения нулевого кванта времени. Для каждого ресурса выводится либо идентификатор владеющего им процесса, либо указание, что ресурс свободен. Для каждого потока выводится его состояние (не инициализирован, ожидает в очереди, работает, завершил работу). Ведущие нули добавляются для того, чтобы выровнять значения в строках (соответствующие значения должны располагаться в виде таблицы друг под другом). Допускается для этой цели использовать пробелы
0…01	Аналогично — после завершения следующего кванта
…	И т.д. для всех оставшихся квантов. Если система зашла в тупик, то следует остановиться на последнем кванте, когда состояние системы претерпело изменения

Для ввода и вывода данных допускается использование в программе визуального интерфейса вместо файлового ввода/вывода.

1.2 Варианты заданий на лабораторную работу № 1

Вариант № 1. Ресурс — преподаватель на экзамене. Атрибуты — ФИО, дисциплина, а также количество студентов N ($N \geq 1$), у которых он может принимать экзамен одновременно. Количество преподавателей — P ($P \geq 1$). Атрибуты студента — ФИО, номер группы и список дисциплин, по которым ему нужно сдать экзамен. Алгоритмы планирования:

1. FCFS, nonpreemptive.
2. Round Robin с очередью типа FCFS, абсолютный приоритет.

Для блокировки доступа к преподавателю использовать сеть Петри.

Вариант № 2. Ресурсы — преподаватель, принимающий лабораторную работу у студентов, а также лабораторное оборудование. Атрибут преподавателя — ФИО, атрибут оборудования — название и количество D ($D \geq 1$). Атрибуты студента — ФИО, номер группы и список оборудования, которое ему необходимо для сдачи лабораторной работы. Алгоритмы планирования:

1. LCFS, nonpreemptive.
2. Round Robin с очередью типа LCFS, абсолютный приоритет.

Для блокировки доступа к преподавателю использовать сеть Петри.

Вариант № 3. Ресурс — оборудование (станки) на заводе. Атрибуты — наименование оборудования, а также количество деталей P ($P \geq 1$), которое оно может обрабатывать одновременно. Количество станков — S ($S \geq 1$). Атрибуты деталей — наименование, количество, а также список оборудования (причем заданный в требуемом порядке обработки). Алгоритмы планирования:

1. SJF, nonpreemptive.
2. SJF, preemptive, абсолютный приоритет.

Для блокировки доступа к преподавателю использовать сеть Петри.

Вариант № 4. Ресурс — преподаватель на экзамене. Атрибуты — ФИО, дисциплина, а также количество студентов N ($N \geq 1$),

у которых он может принимать экзамен одновременно. Количество преподавателей — P ($P \geq 1$). Атрибуты студента — ФИО, номер группы и список дисциплин, по которым ему нужно сдать экзамен. Алгоритмы планирования:

1. FCFS, nonpreemptive.
2. Round Robin с очередью типа FCFS, относительный приоритет.

Для блокировки доступа к преподавателю использовать семафор.

Вариант № 5. Ресурсы — преподаватель, принимающий лабораторную работу у студентов, а также лабораторное оборудование. Атрибут преподавателя — ФИО, атрибут оборудования — название и количество D ($D \geq 1$). Атрибуты студента — ФИО, номер группы и список оборудования, которое ему необходимо для сдачи лабораторной работы. Алгоритмы планирования:

1. SJF, nonpreemptive.
2. SJF, preemptive, относительный приоритет.

Для блокировки доступа к преподавателю использовать семафор.

Вариант № 6. Ресурс — оборудование (станки) на заводе. Атрибуты — наименование оборудования, а также количество деталей P ($P \geq 1$), которое оно может обрабатывать одновременно. Количество станков — S ($S \geq 1$). Атрибуты деталей — наименование, количество, а также список оборудования (причем заданный в требуемом порядке обработки). Алгоритмы планирования:

1. LCFS, nonpreemptive.
2. MLQ, абсолютный приоритет.

Для блокировки доступа к преподавателю использовать сеть Петри.

Вариант № 7. Ресурс — преподаватель на экзамене. Атрибуты — ФИО, дисциплина, а также количество студентов N ($N \geq 1$), у которых он может принимать экзамен одновременно. Количество преподавателей — P ($P \geq 1$). Атрибуты студента — ФИО, номер группы и список дисциплин, по которым ему нужно сдать экзамен. Алгоритмы планирования:

1. LCFS, nonpreemptive.
2. Round Robin с очередью типа LCFS, абсолютный приоритет.

Для блокировки доступа к преподавателю использовать сеть Петри.

Вариант № 8. Ресурсы — преподаватель, принимающий лабораторную работу у студентов, а также лабораторное оборудование. Атрибут преподавателя — ФИО, атрибут оборудования — название и количество D ($D \geq 1$). Атрибуты студента — ФИО, номер группы и список оборудования, которое ему необходимо для сдачи лабораторной работы. Алгоритмы планирования:

1. SJF, nonpreemptive.
2. SJF, preemptive, абсолютный приоритет.

Для блокировки доступа к преподавателю использовать сеть Петри.

Вариант № 9. Ресурс — оборудование (станки) на заводе. Атрибуты — наименование оборудования, а также количество деталей P ($P \geq 1$), которое оно может обрабатывать одновременно. Количество станков — S ($S \geq 1$). Атрибуты деталей — наименование, количество, а также список оборудования (причем заданный в требуемом порядке обработки). Алгоритмы планирования:

1. LCFS, nonpreemptive.
2. MLQ, относительный приоритет.

Для блокировки доступа к преподавателю использовать семафор.

Вариант № 10. Ресурс — преподаватель на экзамене. Атрибуты — ФИО, дисциплина, а также количество студентов N ($N \geq 1$), у которых он может принимать экзамен одновременно. Количество преподавателей — P ($P \geq 1$). Атрибуты студента — ФИО, номер группы и список дисциплин, по которым ему нужно сдать экзамен. Алгоритмы планирования:

1. LCFS, nonpreemptive.
2. Round Robin с очередью типа LCFS, относительный приоритет.

Для блокировки доступа к преподавателю использовать семафор.

Вариант № 11. Ресурсы — преподаватель, принимающий лабораторную работу у студентов, а также лабораторное оборудование. Атрибут преподавателя — ФИО, атрибут оборудования — название и количество D ($D \geq 1$). Атрибуты студента — ФИО, номер группы и список оборудования, которое ему необходимо для сдачи лабораторной работы. Алгоритмы планирования:

1. FCFS, nonpreemptive.
2. Round Robin с очередью типа FCFS, абсолютный приоритет.

Для блокировки доступа к преподавателю использовать сеть Петри.

Вариант № 12. Ресурс — преподаватель на экзамене. Атрибуты — ФИО, дисциплина, а также количество студентов N ($N \geq 1$), у которых он может принимать экзамен одновременно. Количество преподавателей — P ($P \geq 1$). Атрибуты студента — ФИО, номер группы и список дисциплин, по которым ему нужно сдать экзамен. Алгоритмы планирования:

1. SJF, nonpreemptive.
2. SJF, preemptive, абсолютный приоритет.

Для блокировки доступа к преподавателю использовать сеть Петри.

Вариант № 13. Ресурсы — преподаватель, принимающий лабораторную работу у студентов, а также лабораторное оборудование. Атрибут преподавателя — ФИО, атрибут оборудования — название и количество D ($D \geq 1$). Атрибуты студента — ФИО, номер группы и список оборудования, которое ему необходимо для сдачи лабораторной работы. Алгоритмы планирования:

1. FCFS, nonpreemptive.
2. Round Robin с очередью типа FCFS, относительный приоритет.

Для блокировки доступа к преподавателю использовать семафор.

Вариант № 14. Ресурс — оборудование (станки) на заводе. Атрибуты — наименование оборудования, а также количество деталей P ($P \geq 1$), которое оно может обрабатывать одновременно. Количество станков — S ($S \geq 1$). Атрибуты деталей — наименова-

ние, количество, а также список оборудования (причем заданный в требуемом порядке обработки). Алгоритмы планирования:

1. LCFS, nonpreemptive.
2. Round Robin с очередью типа LCFS, относительный приоритет.

Для блокировки доступа к преподавателю использовать семафор.

Вариант № 15. Ресурс — преподаватель на экзамене. Атрибуты — ФИО, дисциплина, а также количество студентов N ($N \geq 1$), у которых он может принимать экзамен одновременно. Количество преподавателей — P ($P \geq 1$). Атрибуты студента — ФИО, номер группы и список дисциплин, по которым ему нужно сдать экзамен. Алгоритмы планирования:

1. SJF, nonpreemptive.
2. SJF, preemptive, относительный приоритет.

Для блокировки доступа к преподавателю использовать семафор.

Вариант № 16. Ресурсы — преподаватель, принимающий лабораторную работу у студентов, а также лабораторное оборудование. Атрибут преподавателя — ФИО, атрибут оборудования — название и количество D ($D \geq 1$). Атрибуты студента — ФИО, номер группы и список оборудования, которое ему необходимо для сдачи лабораторной работы. Алгоритмы планирования:

1. LCFS, nonpreemptive.
2. MLQ, абсолютный приоритет.

Для блокировки доступа к преподавателю использовать сеть Петри.

Вариант № 17. Ресурс — оборудование (станки) на заводе. Атрибуты — наименование оборудования, а также количество деталей P ($P \geq 1$), которое оно может обрабатывать одновременно. Количество станков — S ($S \geq 1$). Атрибуты деталей — наименование, количество, а также список оборудования (причем заданный в требуемом порядке обработки). Алгоритмы планирования:

1. LCFS, nonpreemptive.
2. Round Robin с очередью типа LCFS, абсолютный приоритет.

Для блокировки доступа к преподавателю использовать сеть Петри.

Вариант № 18. Ресурс — преподаватель на экзамене. Атрибуты — ФИО, дисциплина, а также количество студентов N ($N \geq 1$), у которых он может принимать экзамен одновременно. Количество преподавателей — P ($P \geq 1$). Атрибуты студента — ФИО, номер группы и список дисциплин, по которым ему нужно сдать экзамен. Алгоритмы планирования:

1. FSFS, nonpreemptive.
2. MLQ, абсолютный приоритет.

Для блокировки доступа к преподавателю использовать сеть Петри.

Вариант № 19. Ресурсы — преподаватель, принимающий лабораторную работу у студентов, а также лабораторное оборудование. Атрибут преподавателя — ФИО, атрибут оборудования — название и количество D ($D \geq 1$). Атрибуты студента — ФИО, номер группы и список оборудования, которое ему необходимо для сдачи лабораторной работы. Алгоритмы планирования:

1. LCFS, nonpreemptive.
2. Round Robin с очередью типа LCFS, относительный приоритет.

Для блокировки доступа к преподавателю использовать семафор.

Вариант № 20. Ресурс — преподаватель на экзамене. Атрибуты — ФИО, дисциплина, а также количество студентов N ($N \geq 1$), у которых он может принимать экзамен одновременно. Количество преподавателей — P ($P \geq 1$). Атрибуты студента — ФИО, номер группы и список дисциплин, по которым ему нужно сдать экзамен. Алгоритмы планирования:

1. LSFS, nonpreemptive.
2. MLQ, относительный приоритет.

Для блокировки доступа к преподавателю использовать семафор.

Вариант № 21. Ресурсы — преподаватель, принимающий лабораторную работу у студентов, а также лабораторное оборудование. Атрибут преподавателя — ФИО, атрибут оборудования — название и количество D ($D \geq 1$). Атрибуты студента — ФИО, номер группы и список оборудования, которое ему необходимо для сдачи лабораторной работы. Алгоритмы планирования:

1. LCFS, nonpreemptive.
2. MLQ, относительный приоритет.

Для блокировки доступа к преподавателю использовать семафор.

Вариант № 22. Ресурс — оборудование (станки) на заводе. Атрибуты — наименование оборудования, а также количество деталей P ($P \geq 1$), которое оно может обрабатывать одновременно. Количество станков — S ($S \geq 1$). Атрибуты деталей — наименование, количество, а также список оборудования (причем заданный в требуемом порядке обработки). Алгоритмы планирования:

1. FCFS, nonpreemptive.
2. Round Robin с очередью типа FCFS, относительный приоритет.

Для блокировки доступа к преподавателю использовать семафор.

Вариант № 23. Ресурс — оборудование (станки) на заводе. Атрибуты — наименование оборудования, а также количество деталей P ($P \geq 1$), которое оно может обрабатывать одновременно. Количество станков — S ($S \geq 1$). Атрибуты деталей — наименование, количество, а также список оборудования (причем заданный в требуемом порядке обработки). Алгоритмы планирования:

1. SJF, nonpreemptive.
2. SJF, preemptive, относительный приоритет.

Для блокировки доступа к преподавателю использовать семафор.

Вариант № 24. Ресурс — оборудование (станки) на заводе. Атрибуты — наименование оборудования, а также количество деталей P ($P \geq 1$), которое оно может обрабатывать одновременно. Количество станков — S ($S \geq 1$). Атрибуты деталей — наименова-

ние, количество, а также список оборудования (причем заданный в требуемом порядке обработки). Алгоритмы планирования:

1. FCFS, nonpreemptive.
2. Round Robin с очередью типа FCFS, абсолютный приоритет.

Для блокировки доступа к преподавателю использовать сеть Петри.

Вариант № 25. Ресурс — преподаватель на экзамене. Атрибуты — ФИО, дисциплина, а также количество студентов N ($N \geq 1$), у которых он может принимать экзамен одновременно. Количество преподавателей — P ($P \geq 1$). Атрибуты студента — ФИО, номер группы и список дисциплин, по которым ему нужно сдать экзамен. Алгоритмы планирования:

1. Round Robin с очередью типа FCFS, абсолютный приоритет.
2. Round Robin с очередью типа LCFS, абсолютный приоритет.

Для блокировки доступа к преподавателю использовать сеть Петри.

2 ЛАБОРАТОРНАЯ РАБОТА № 2

Цель выполнения лабораторной работы № 2 — освоить реализацию многопоточной обработки данных, а также пула потоков и механизма асинхронного ввода/вывода.

2.1 Задание

В работе необходимо реализовать многопоточную обработку массива структур данных (из N элементов) четырьмя способами:

1. При помощи массива из M потоков ($M \leq N$), используя для синхронизации объект ядра — семафор.

2. При помощи массива из M потоков ($M \leq N$), используя для синхронизации сеть Петри, моделирующую семафор.

3. При помощи пула из M потоков ($M \leq N$), используя системный пул потоков или асинхронные потоки ввода/вывода.

4. При помощи пула из M потоков ($M \leq N$), моделируя его при помощи сети Петри.

При обработке массива данных массивом потоков каждый поток либо заранее выбирает диапазон элементов массива данных, которые он будет обрабатывать, либо просто берет первый необработанный элемент. Завершив обработку одного элемента, поток переходит к обработке следующего.

При обработке массива данных пулом потоков, завершив обработку одного элемента массива данных, освободившийся в пуле поток переходит к обработке следующего необработанного элемента.

Чтобы не требовалось создавать слишком большие массивы (для которых эффект от параллельной обработки будет наиболее очевидным), можно имитировать ситуацию, когда обработка одного элемента массива требует больше процессорного времени, чем в действительности. Для этого после обработки очередного элемента массива поток может делать паузу на указанное количество миллисекунд.

Входной файл должен иметь имя «input.txt» или «input.xml». Формат входного файла представлен в табл. 2.1.

Таблица 2.1 — Формат входного файла для лабораторной работы № 2

Поле	Значение
PA	Выбранный способ обработки массива
N	Размер массива структур данных (значения полей каждой структуры генерируются программой случайным образом)
M	Количество параллельных потоков (если 0, то принимается равным числу процессорных ядер в системе)
PT	Пауза после обработки каждого элемента массива (мс)
…	Дополнительные входные данные (зависят от варианта)

Выходной файл должен иметь имя «output.txt». Формат выходного файла представлен в табл. 2.2.

Таблица 2.2 — Формат выходного файла для лабораторной работы № 2

Поле	Значение
T1	Время, требующееся на линейную обработку массива (без распараллеливания вычислений)
TP	Время, требующееся на параллельную обработку массива выбранным способом
…	Результаты обработки (зависят от варианта)

Для ввода и вывода данных допускается использование в программе визуального интерфейса вместо файлового ввода/вывода.

2.2 Варианты заданий на лабораторную работу № 2

Вариант № 1. Структура содержит корректное описание даты (день, месяц, год). Требуется определить, какие даты выпадают на определенный день недели *W*, и вывести их в выходной файл.

Вариант № 2. Структура содержит описание времени дня (часы, минуты, секунды). Требуется определить, какие из указанных значений времени лежат в диапазоне от T_1 до T_2, и вывести их в выходной файл.

Вариант № 3. Структура содержит запись телефонного справочника (ФИО абонента, номер телефона, адрес). Требуется по фамилии найти номер телефона и адрес абонента и вывести их в выходной файл.

Вариант № 4. Структура содержит данные о точке на плоскости (координаты и тип координат — полярные или декартовые). Требуется преобразовать все декартовые координаты в полярные, и наоборот. Также требуется определить точку, наиболее удаленную от точки с номером *P*. Вывести в выходной файл координаты этих точек и расстояние между ними.

Вариант № 5. Структура содержит данные о жителях городов и их адресах (ФИО, город, улица, номер дома, номер квартиры). Требуется определить всех жителей, живущих в разных городах, но по одинаковому адресу, и вывести информацию о таких жителях в выходной файл.

Вариант № 6. Структура содержит анкетные данные студентов (ФИО, группа, дата рождения, номер комнаты в общежитии). Требуется вывести в выходной файл данные о студентах, которые родились в заданном месяце *M*.

Вариант № 7. Структура содержит библиографическое описание книги (авторы, название, издательство и год издания одной строкой, число страниц). Требуется определить книги, изданные в издательстве *P*, и вывести их список в выходной файл.

Вариант № 8. Структура содержит сведения о товарах в магазине (буквенно-цифровой код товара, наименование, цена, количество на складе). Требуется изменить в товарах с кодом, удовлетворяющим маске *M*, количество на величину *V* (но количество при этом не должно оказаться отрицательным). Длина маски равна длине кода товаров, для указания произвольного символа используется знак вопроса. В выходной файл вывести список товаров, количество которых на складе изменилось, а также стоимость всех товаров на складе до и после изменения.

Вариант № 9. Структура содержит сведения о вещах в багаже пассажиров (ФИО пассажира, название вещи, количество единиц, вес одной единицы). Требуется найти багаж, число вещей в котором не меньше, чем в любом другом, а вес не больше, чем в любом другом с тем же числом вещей, и вывести данные о нем в выходной файл.

Вариант № 10. Структура содержит анкетные данные студентов (ФИО, группа, номер зачетной книжки, результаты пройденных сессий). Результаты сессий представлены названием дисциплины и отметкой — «зачтено/не зачтено» для зачета и числом от 0 до 5 для экзамена. Требуется определить и вывести в выходной файл среднюю успеваемость студентов группы *G*, а также общую среднюю успеваемость в семестре с номером *S*.

Вариант № 11. Структура описывает кость домино (количество точек — две цифры в диапазоне от 1 до 6, а также ориентация кости). Требуется определить — образует ли последовательность костей домино правильную последовательность. Если да — вывести в выходной файл всю последовательность. Если нет — вывести номера костей, нарушающих последовательность.

Вариант № 12. Структура содержит библиографическое описание книги (один или несколько авторов одной строкой, название, издательство, год издания, число страниц). Требуется определить книги, в написании которых принимал участие автор *A*, и вывести их список в выходной файл.

Вариант № 13. Структура содержит анкетные данные некоторой группы лиц (ФИО, дата рождения, пол, рост, вес). Требуется вывести в выходной файл средний рост мужчин и женщин, рожденных начиная с года *A* и заканчивая *B*.

Вариант № 14. Структура содержит сведения о странице телефонной записной книжки — это буква, с которой начинаются фамилии на данной странице, а также список фамилий и номеров телефонов. Программа должна по маске номера телефона *T* определить подходящие фамилии абонентов и вывести сведения о них

в выходной файл. В маске вместо неизвестной цифры используется знак вопроса.

Вариант № 15. Структура содержит анкетные данные студентов (ФИО, группа, номер зачетной книжки, дата рождения). Требуется определить самых старших студентов в каждой группе. Список таких студентов вывести в выходной файл.

Вариант № 16. Структура описывает игральную карту (масть, достоинство). Карты могут повторяться, т.е. разные структуры могут описывать одну и ту же карту. Необходимо определить карты, которых не хватает в колоде, и вывести их список в выходной файл.

Вариант № 17. Структура содержит сведения о вещах в багаже пассажиров (ФИО пассажира, название вещи, количество единиц, вес одной единицы). Требуется найти общий вес вещей с названием *X*. В выходной файл необходимо вывести список пассажиров, имеющих в багаже вещь *X*, а также найденный вес.

Вариант № 18. Структура содержит описание даты (день, месяц, год). Требуется проверить правильность каждой даты (т.е. чтобы не было 31 июня и т.п.). В выходной файл необходимо вывести количество неправильных дат и их список.

Вариант № 19. Структура содержит сведения о товарах в магазине (вид товара из двух символов, код товара из четырех символов, наименование, цена, количество на складе). Требуется подсчитать стоимость товара вида *X*, а также общую стоимость товаров, чьи коды лежат в диапазоне от *A* до *B*. Определенные стоимости вывести в выходной файл вместе со списками соответствующих товаров.

Вариант № 20. Структура содержит анкетные данные студентов (ФИО, группа, номер зачетной книжки, результаты пройденных сессий). Результаты сессий представлены названием дисциплины и отметкой — «зачтено/не зачтено» для зачета и числом от 0 до 5 для экзамена. Требуется определить, сколько студентов

группы *G* получали стипендию в семестре с номером *S*. В первом семестре все студенты получает стипендию, а далее — по результатам предыдущей сессии (для получения стипендии необходимо получить все зачеты и сдать все экзамены на оценку 4 или 5). Список таких студентов вывести в выходной файл.

3 КРАТКАЯ ТЕОРИЯ

3.1 Потоки. Основные понятия

Для разделения различных выполняемых приложений в операционных системах используются *процессы*. *Потоки* являются основными элементами, для которых операционная система выделяет время процессора; внутри процесса может выполняться более одного потока. Каждый поток поддерживает обработчик исключений, планируемый *приоритет* и набор структур, используемых системой для сохранения *контекста потока* во время его планирования. Контекст потока содержит все необходимые данные для возобновления выполнения (включая набор регистров процессора и стек) в адресном пространстве ведущего процесса.

3.1.1 Основы организации потоков

По умолчанию, все разрабатываемые приложения являются *однопоточными*. *Многопоточность* позволяет приложениям разделять задачи и работать над каждой независимо, чтобы максимально эффективно задействовать процессор и пользовательское время. Однако чрезмерное злоупотребление многопоточностью может снизить эффективность программы. Разделять процесс на потоки следует только в том случае, если это оправданно.

3.1.1.1 Потоки и многозадачность

Поток является единицей обработки данных, а *многозадачность* — это одновременное исполнение нескольких потоков. Существует два вида многозадачности — совместная (cooperative) и вытесняющая (preemptive). Самые ранние версии Microsoft Windows поддерживали совместную многозадачность. Это означало, что каждый поток отвечал за возврат управления процессору, чтобы тот смог обработать другие потоки. То есть если какой-либо поток не возвращал управление (из-за ошибки в программе или по другой причине), другие потоки не могли продолжить выполнение. И если этот поток «зависал», то «зависала» вся система.

Однако, начиная с Windows NT, стала поддерживаться вытесняющая многозадачность. При этом процессор отвечает за выдачу каждому потоку определенного количества времени, в течение которого поток может выполняться — *кванта времени* (timeslice). Далее процессор переключается между разными потоками, выдавая каждому потоку его квант времени, а программист может не заботиться о том, как и когда возвращать управление, в результате чего могут работать и другие потоки.

Даже в случае вытесняющей многозадачности, если в системе установлен только один процессор (вернее будет сказать — одно процессорное ядро, т.к. современные процессоры содержат несколько ядер), то все равно в любой момент времени реально будет исполняться только один поток. Поскольку интервалы между переключениями процессора от процесса к процессу измеряются миллисекундами, возникает иллюзия многозадачности. Чтобы несколько потоков на самом деле работали одновременно, необходимо работать на многопроцессорной или многоядерной машине (или использовать процессоры, поддерживающие технологию Intel HyperThreading, позволяющую двум потокам работать на одном процессорном ядре параллельно), а также использовать специальные механизмы, позволяющие назначить каждому потоку выполнение на соответствующем ядре процессора.

3.1.1.2 Переключение контекста

Неотъемлемый атрибут потоков — *переключение контекста* (context switching). Процессор с помощью аппаратного таймера определяет момент окончания кванта, выделенного для данного потока. Когда аппаратный таймер генерирует прерывание, процессор сохраняет в стеке содержимое всех регистров для данного потока. Затем процессор перемещает содержимое этих же регистров в специальную структуру данных контекста. При необходимости переключения обратно на поток, выполнявшийся прежде, процессор выполняет обратную процедуру и восстанавливает содержимое регистров из структуры контекста, ассоциированной с потоком. Весь этот процесс называется переключением контекста.

3.1.1.3 Правила использования потоков

После создания многопоточной программы может неожиданно оказаться, что издержки, связанные с созданием и диспетчеризацией потоков, привели к тому, что однопоточное приложение работает быстрее.

Например, если требуется считать с диска три файла, создание для этого трех потоков не принесет пользы, поскольку все они будут обращаться к одному и тому же жесткому диску. Поэтому всегда нужно стараться тестировать несколько версий программы и выбирать оптимальное решение.

Потоки следует использовать тогда, когда целью является достижение повышенного параллелизма, упрощение структуры программы и эффективность использования процессорного времени.

1) Повышенный параллелизм. Если написан сложный алгоритм обработки каких-либо данных (сортировка больших массивов данных, обработка больших блоков текста, сканирование файловой системы и т.п.), время выполнения которого может превышать 5—10 секунд, то логично будет выделить работу этого алгоритма в отдельный поток. Иначе во время работы программы диспетчер сообщений, поступающих от ОС, не будет получать управления, и, таким образом, пользовательский ввод в программе окажется полностью блокированным. Хотя известно, что некоторые файловые менеджеры (FAR Manager, Total Commander и т.п.), архиваторы (WinRAR и т.п.), менеджеры закачек, браузеры, программы для обслуживания жестких дисков, программы редактирования мультимедиа и т.д. допускают выполнение трудоемких задач в фоновом режиме. В этом случае, при необходимости, такую задачу можно прервать или запустить параллельно другую задачу.

Кроме того, выполнение самой задачи, если оно подразумевает обработку однородных данных, может быть разделено между несколькими потоками.

2) Упрощенная структура. Например, приложение получает асинхронные сообщения (т.е. которые могут приходить в любой момент, в т.ч. во время обработки других сообщений) от других приложений или потоков. Популярный способ упрощения струк-

туры таких систем — использование очередей и асинхронной обработки. Вместо прямого вызова методов создаются специальные объекты сообщений и помещаются в очереди, в которых производится их обработка. На другом конце этих очередей работает несколько потоков, настроенных на отслеживание приходящих сообщений.

3) Эффективное использование процессорного времени. Часто приложение реально не выполняет никакой работы, в то же время продолжая использовать свой квант процессорного времени. Например, ожидает поступления какого-либо события (от устройства ввода команд пользователя, об окончании печати документа, об окончании обработки файла и т.д.). Эти случаи являются кандидатами на перевод в потоки, работающие в фоновом режиме.

3.1.2 Планирование использования процессора

3.1.2.1 Невытесняющая многозадачность

Планирование использования процессорного времени выступает в качестве краткосрочного планирования. Выбор конкретного алгоритма определяется классом задач, решаемых вычислительной системой, и целями, которых стремятся достичь, используя планирование.

Деятельность любого процесса можно представить как последовательность циклов использования процессора и ожидания завершения операций ввода-вывода. Далее будем использовать следующие обозначения:

– CPU burst — промежуток времени непрерывного использования процессора;

– I/O burst — промежуток времени непрерывного ожидания ввода-вывода.

Планировщик может принимать решения о выборе для исполнения нового процесса, из числа находящихся в состоянии готовности, в следующих четырех случаях:

1) когда процесс переводится из состояния «исполнение» в состояние «завершение»;

2) когда процесс переводится из состояния «исполнение» в состояние «ожидание»;

3) когда процесс переводится из состояния «исполнение» в состояние «готовность» (например, после прерывания от таймера);

4) когда процесс переводится из состояния «ожидание» в состояние «готовность» (завершилась операция ввода-вывода или произошло другое событие).

В случаях 1 и 2 процесс, находившийся в состоянии «исполнение», не может дальше исполняться, и для выполнения всегда необходимо выбрать новый процесс. Если планирование осуществляется только в случаях 1 и 2, то говорят, что имеет место невытесняющее (nonpreemptive) планирование. В противном случае говорят о вытесняющем (preemptive) планировании.

1) Простейшим алгоритмом планирования является алгоритм, который принято обозначать аббревиатурой FCFS по первым буквам его английского названия — First Come, First Served (первым пришел, первым обслужен). Представим себе, что процессы, находящиеся в состоянии «готовность», организованы в очередь. Когда процесс переходит в состояние «готовность», он, а точнее ссылка на его PCB (process control block), помещается в конец этой очереди. Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки на его PCB. Очередь подобного типа имеет в программировании специальное наименование FIFO — сокращение от First In, First Out (первым вошел, первым вышел).

Как вариант, алгоритм может иметь вид LCFS (Last Come, First Served — последним пришел, первым обслужен). Тогда очередь будет называться LIFO — Last In, First Out (последним вошел, первым вышел).

Такой алгоритм выбора процесса осуществляет невытесняющее планирование. Процесс, получивший в свое распоряжение процессор, занимает его до истечения своего текущего CPU burst. После этого для выполнения выбирается новый процесс из начала очереди.

2) В том случае, когда известно время следующих CPU burst для процессов, находящихся в состоянии «готовность», можно выбрать для исполнения процесс с минимальной длительностью CPU burst. Если же таких процессов два или больше, то для вы-

бора одного из них можно использовать уже известный нам алгоритм FCFS или LCFS. Квантование времени при этом не применяется. Описанный алгоритм получил название «кратчайшая работа первой», или Shortest Job First (SJF).

Алгоритм краткосрочного планирования может быть как вытесняющим, так и невытесняющим. При невытесняющем SJF планировании процессор предоставляется избранному процессу на все требующееся ему время, независимо от событий, происходящих в вычислительной системе.

При приоритетном планировании каждому процессу присваивается определенное числовое значение — приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми приоритетами планируются в порядке FCFS или LCFS.

Планирование с использованием приоритетов может быть как вытесняющим, так и невытесняющим. Если приоритеты процессов не изменялись с течением времени, то такие приоритеты принято называть статическими. Более гибкими являются динамические приоритеты процессов, изменяющие свои значения по ходу исполнения процессов.

3.1.2.2 Вытесняющая многозадачность

Когда процесс переводится из состояния «исполнение» в состояние «готовность» (например, после прерывания от таймера), а также когда процесс переводится из состояния «ожидание» в состояние «готовность» (завершилась операция ввода-вывода или произошло другое событие), то говорят, что имеет место вытесняющее (preemptive) планирование. Термин «вытесняющее планирование» возник потому, что исполняющийся процесс помимо своей воли может быть вытеснен из состояния исполнения другим процессом.

1) Самый простой и часто используемый алгоритм планирования является модификацией алгоритма FCFS/LCFS и реализован в режиме вытесняющего планирования. Каждому процессу предоставляется квант времени процессора. Когда квант заканчивается, процесс переводится планировщиком в конец очереди. При блокировке процесс выпадает из очереди.

Этот алгоритм получил название Round Robin (Round Robin — это вид детской карусели в США), или сокращенно RR. Можно представить себе все множество готовых процессов организованным циклически — процессы сидят на карусели. Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный квант времени. Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.

Планировщик выбирает для очередного исполнения процесс, расположенный в начале очереди, и устанавливает таймер для генерации прерывания по истечении определенного кванта времени.

При выполнении процесса возможны два варианта:

а) время непрерывного использования процессора, требующееся процессу (остаток текущего CPU burst), меньше или равно продолжительности кванта времени. Тогда процесс по своей воле освобождает процессор до истечения кванта времени, на исполнение выбирается новый процесс из начала очереди, и таймер начинает отсчет кванта заново;

б) продолжительность остатка текущего CPU burst процесса больше, чем квант времени. Тогда по истечении этого кванта процесс прерывается таймером и помещается в конец очереди процессов, готовых к исполнению, а процессор выделяется для использования процессу, находящемуся в ее начале.

2) Алгоритм краткосрочного планирования SJF может быть так же и вытесняющим. При вытесняющем SJF-планировании учитывается появление новых процессов в очереди, готовых к исполнению (из числа вновь родившихся или разблокированных) во время работы выбранного процесса. Если CPU burst нового процесса меньше, чем остаток CPU burst у исполняющегося процесса, то исполняющийся процесс вытесняется новым.

Основную сложность при реализации алгоритма SJF представляет невозможность точного знания времени очередного CPU burst для исполняющихся процессов. При краткосрочном планировании можно делать только прогноз длительности следующего CPU burst исходя из предыстории работы процесса.

3) Многоочередное вытесняющее планирование (MLQ — Multilevel Queue). Здесь приоритет определяется очередью (ее

номером). Первый запрос PCB из очереди с номером *i* поступает лишь тогда, когда все очереди с номерами от 0 до *i*–1 пустые. Выбранному из очереди процессу выделяется квант процессорного времени. Если за это время CPU burst завершается полностью, то процесс покидает очередь. В противном случае он поступает в конец очереди с номером *i*+1.

После завершения очереди *i* (т.е. когда в ней не остается PCB процессов) система выбирает для обслуживания PCB из непустой очереди с самым младшим номером.

3.1.2.3 Планирование с абсолютным и относительным приоритетом

При вытесняющей многозадачности планирование процессорного времени может происходить с абсолютным или относительным приоритетом.

В вытесняющей многозадачности, где поступающий в очередь процесс имеет определенный приоритет, первыми получают квант процессорного времени процессы с наивысшим приоритетом. Если используется планирование с **абсолютным приоритетом**, то при поступлении в очередь процесса с более высоким приоритетом обслуживание текущего процесса прерывается, и начинает работать вновь поступивший процесс. Только после этого будет дообслужен прерванный процесс.

При планировании с **относительным приоритетом** процесс, поступающий в очередь, не вызывает прерывания выполняемого в настоящее время процесса, даже если последний и менее приоритетный. В данном случае только после окончания CPU burst выполняемого процесса будет выделен квант процессорного времени более приоритетному процессу.

Все вышеизложенное касается планирования распределения только одного ресурса — процессорного времени, причем без учета взаимосвязи процессов между собой. Процессы, как правило, используют различные ресурсы, и этот факт оказывает влияние на планирование их распределения. На практике должна быть построена стратегия планирования, удовлетворительная не только в отношении некоторого конкретного ресурса, но и согласованная со стратегиями планирования доступа к другим ресурсам.

Реализация планирования распределения ресурсов на практике также усложняется из-за необходимости анализа возможности возникновения *тупиковых ситуаций*, проверки анализа полномочий на использование каждым процессом распределяемого ресурса.

3.1.2.4 Тупиковые ситуации

При параллельном исполнении процессов могут возникнуть такие ситуации, при которых два или более процессов все время находятся в заблокированном состоянии. Для возникновения тупиковой ситуации необходимо, чтобы одновременно выполнялись четыре условия:

1) *взаимного исключения*, при котором процессы осуществляют монопольный доступ к ресурсам;

2) *ожидания*, при котором процесс, запросивший ресурс, ждет до тех пор, пока запрос не будет удовлетворен, при этом удерживая ранее полученные ресурсы;

3) *отсутствия перераспределения*, при котором ресурсы нельзя отбирать у процесса, если они уже ему выделены;

4) *кругового ожидания*, при котором существует замкнутая цепь процессов, каждый из которых ждет ресурс, удерживаемый его предшественником в этой цепи.

Для решения проблемы тупиковой ситуации, можно выбрать одну из трех стратегий:

1) стратегия предотвращения deadlock (запрет существования опасных состояний) — тупиковые ситуации настолько дорогостоящи, что лучше потратить дополнительные ресурсы системы для сведения к нулю вероятности возникновения тупиковых ситуаций при любых обстоятельствах;

2) стратегия обхода deadlock (запрет входа в опасное состояние) гарантирует, что тупиковая ситуация хотя, в принципе, и возможна, но не возникает для конкретного набора процессов и запросов, выполняющихся в данный момент;

3) стратегия распознавания deadlock и последующего восстановления (запрет постоянного пребывания в опасном состоянии) базируется на том, что тупиковая ситуация возникает достаточно редко, и поэтому предпочтительнее просто распознать ее и

произвести восстановление, чем применять стратегии предотвращения или обхода тупика.

Дисциплина, предотвращающая deadlock, должна гарантировать, что хотя бы одно из четырех условий, необходимых для его возникновения, не наступит. Поэтому для предотвращения deadlock следует подавить хотя бы одно из следующих условий:

– Условие взаимного исключения подавляется путем разрешения неограниченного разделения ресурсов.

– Условие ожидания подавляется предварительным выделением ресурсов. Процесс может потребовать все ресурсы заранее, и он не может начать выполнение до тех пор, пока они ему не будут выделены. Следовательно, общее число ресурсов, необходимое параллельным процессам, не может превышать возможности системы. Каждый процесс должен ждать до тех пор, пока не получит все необходимые ресурсы, даже если один из них используется только в конце исполнения процесса.

– Условие отсутствия перераспределения подавляется разрешением операционной системе отнимать у процесса ресурсы. Это возможно, если можно запомнить состояние процесса для его последующего восстановления.

– Условие кругового ожидания подавляется предотвращением образования цепи запросов, что можно обеспечить иерархическим выделением ресурсов. Все ресурсы образуют некоторую иерархию. Процесс, затребовавший ресурс на одном уровне, может затем потребовать ресурсы на более высоком уровне. Он может освободить ресурсы на данном уровне только после освобождения всех ресурсов на всех более высоких уровнях.

Когда последовательность запросов, связанных с каждым процессом, неизвестна заранее, но известен общий запрос на ресурсы каждого типа, то выделение ресурсов можно контролировать: для каждого требования, предполагая, что оно удовлетворено, надо определять, существует ли среди общих запросов некоторая последовательность требований, которая может привести к опасному состоянию.

Вход в опасное состояние можно предотвратить, если у системы есть информация о последовательности запросов, связанных с каждым параллельным процессом.

Если вычисления находятся в любом неопасном состоянии, то существует, по крайней мере, одна последовательность состояний, которая обходит опасное состояние. Следовательно, достаточно проверить, не приведет ли выделение затребованного ресурса сразу же к опасному состоянию. Если да, запрос отклоняется. Если нет, его можно выполнить. Проверка того, является состояние опасным или нет, требует анализа последующих запросов процессов. Существуют методы эффективного выполнения такого просмотра. Данный подход является примером контролируемого выделения ресурса. Классическое решение этой задачи известно как «*алгоритм банкира*».

Алгоритм банкира. Банкир ссужает денежные суммы (в одной валюте) некоторому числу клиентов, каждый из которых заранее сообщает банкиру максимальную сумму, которая ему будет нужна. Клиент может занимать эту сумму по частям, и нет никакой гарантии, что он возвратит часть денег до того, как сделает весь заем. Весь капитал банкира обычно меньше, чем суммарные требования клиентов, так как банкир не предполагает, что все в некоторый момент сделают максимальные заемы одновременно. Если в некоторый момент клиент запросит денежную сумму, банкир должен знать, сможет ли он ссудить ее без риска попасть в ситуацию, когда не будет достаточного количества денег, чтобы обеспечить дальнейшие заемы, а именно это, в конце концов, позволяет клиентам возвратить долг. Чтобы решить проблему:

– банкир предполагает, что выполнил запрос и оценивает сложившуюся ситуацию;

– определяет клиента, чей текущий заем наиболее близок к максимальному;

– если банкир не может ссудить оставшуюся сумму этому клиенту, то он отклоняет первоначальный запрос;

– если же банкир может ссудить оставшуюся сумму, то он предполагает, что этот клиент полностью рассчитался, и обращает свое внимание на того клиента из оставшихся, чей запрос ближе всего к своему лимиту;

– просматривая, таким образом, всех клиентов, банкир каждый раз проверяет, будет ли достаточно денег, чтобы удовле-

творить минимальный заем клиента и предоставить последнему полную сумму. В случае если это так, банкир удовлетворяет первоначальный заем.

3.2 Многопоточное программирование

3.2.1 Работа с потоками в ОС Windows

В языках C++ и Pascal создание и удаление потоков, а также управление ими осуществляется вызовом функций API Windows. Языки, поддерживающие платформу .NET (C++ CLI, Pascal.NET, C#), для этих целей используют класс System.Threading.Thread.

3.2.1.1 Многопоточное приложение

Прежде чем изучать способы использования потоков, посмотрим, как создать простейший вторичный поток на различных языках.

В языке C# с использованием .NET:

```
using System;
using System.Threading;

namespace SimpleThreadSample
{
      class Program
      {
            static void ThreadMethod()
            {
                  Console.WriteLine("Поток работает");
            }

            static int Main()
            {
                  ThreadStart ts = new ThreadStart(ThreadMethod);
                  Thread th = new Thread(ts);

                  Console.WriteLine("Main: запускаем поток");
                  th.Start();
                  Console.WriteLine("Main: поток запущен");
                  Console.ReadKey();
                  return 0;
            }
      }
}
```

В языке C++ CLI с использованием .NET:

```
using namespace System;
using namespace System::Threading;

void ThreadMethod()
{
	Console::WriteLine(L"Поток работает");
}

int main()
{
	ThreadStart ^ts = gcnew ThreadStart(ThreadMethod);
	Thread ^th = gcnew Thread(ts);

	Console::WriteLine(L"Main: запускаем поток");
	th->Start();
	Console::WriteLine(L"Main: поток запущен");
	Console::ReadKey();
	return 0;
}
```

В языке C# с использованием API Windows:

```
using System;
using System.Runtime.InteropServices;

namespace SimpleThreadSample
{
	class Program
	{
		[DllImport("kernel32.dll")] static extern IntPtr
CreateThread([In] ref SECURITY_ATTRIBUTES lpSecurityAttributes, uint
dwStackSize, THREAD_START_ROUTINE lpStartAddress, IntPtr lpParameter, uint
dwCreationFlags, out uint lpThreadId);
		[DllImport("kernel32.dll")] static extern uint
ResumeThread(IntPtr hThread);
		const uint CREATE_SUSPENDED = 0x00000004;
		[StructLayout(LayoutKind.Sequential)] public struct
SECURITY_ATTRIBUTES
		{
			public int nLength;
			public IntPtr lpSecurityDescriptor;
			public int bInheritHandle;
		}
		delegate uint THREAD_START_ROUTINE(IntPtr lpParameter);

		static uint ThreadMethod(IntPtr lpParameter)
		{
			Console.WriteLine("Поток работает");
			return 0;
		}

		static void Main()
		{
			uint id;
			SECURITY_ATTRIBUTES attr = new
SECURITY_ATTRIBUTES();
			IntPtr th = CreateThread(ref attr, 0, ThreadMethod,
IntPtr.Zero, CREATE_SUSPENDED, out id);
			Console.WriteLine("Main: запускаем поток");
			ResumeThread(th);
			Console.WriteLine("Main: поток запущен");
			Console.ReadKey();
		}
	}
}
```

В языке C++ с использованием API Windows:

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

DWORD __stdcall ThreadMethod(LPVOID)
{
      printf("Поток работает\n");
      return 0;
}

int main()
{
      DWORD id;
      HANDLE th = CreateThread(NULL, 0, ThreadMethod, NULL,
CREATE_SUSPENDED, &id);
      printf("Main: запускаем поток\n");
      ResumeThread(th);
      printf("Main: поток запущен\n");
      getch();
      return 0;
}
```

Вывод на консоль для всех примеров:

```
Main: запускаем поток
Main: поток запущен
Поток работает
```

Как видно, сообщения метода Main выводится перед сообщением потока. Это доказывает, что поток действительно работает асинхронно. Проанализируем происходящие здесь события.

В пространстве имен System.Threading описаны типы, необходимые для организации потоков в среде .NET. Тип ThreadStart — это делегат, который содержит адрес метода, вызываемого при запуске потока. Его необходимо указать в конструкторе класса Thread, собственно инкапсулирующего поток. Описание этого делегата следующее:

```
delegate void ThreadStart();
```

Итак, требуемый метод должен ничего не возвращать и не иметь параметров. Можно использовать сокращенную форму записи, без явного создания экземпляра делегата ThreadStart:

```
Thread th = new Thread(ThreadMethod);
```

Есть и другой делегат ParameterizedThreadStart для методов с пользовательским параметром:

```
delegate void ParameterizedThreadStart(object obj);
```

Его также можно использовать при конструировании экземпляра класса Thread. В качестве параметра можно указать ссылку на любые данные, которая будет передана в метод потока. Метод Start объекта Thread запускает поток на выполнение, т.к. создается поток в приостановленном режиме.

При использовании функции API Windows CreateThread происходят те же события. Данная функция, согласно MSDN, имеет следующий прототип:

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

Рассмотрим ее аргументы:

- lpThreadAttributes — дескриптор, определяющий наследование прав доступа в иерархии потоков. Если он не нужен, то в языке C++ передаем нулевой указатель NULL, а в языке C# можно первый параметр функции CreateThread заменить указателем типа IntPtr и передавать нулевой указатель IntPtr.Zero:

```
[DllImport("kernel32.dll")] static extern IntPtr CreateThread(IntPtr
lpSecurityAttributes, uint dwStackSize, THREAD_START_ROUTINE
lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, out uint
lpThreadId);
...
IntPtr th = CreateThread(IntPtr.Zero, 0, ThreadMethod, IntPtr.Zero,
CREATE_SUSPENDED, out id);
```

- dwStackSize — размер стека, при передаче значения 0 используется размер по умолчанию.
- lpStartAddress — адрес функции, запускаемой при старте потока. Имеет тот же смысл, что и делегат, передаваемый в конструктор класса Thread. Согласно MSDN данная функция должна иметь следующий вид:

```
DWORD WINAPI ThreadFunc(LPVOID lpParameter);
```

- lpParameter — пользовательский параметр, передаваемый в функцию потока (имеет тот же смысл, что и параметр obj делегата ParameterizedThreadStart).
- dwCreationFlags — дополнительные флаги. По умолчанию поток, созданный функцией CreateThread, сразу запускается

на выполнение. Чтобы провести аналогию с классом Thread, в данных примерах он создается приостановленным (флаг CREATE_SUSPENDED). В этом случае продолжить выполнение потока позволяет функция ResumeThread. Но поток можно запустить сразу при создании, например:

```
printf("Main: запускаем поток\n");
CreateThread(NULL, 0, ThreadMethod, NULL, 0, &id);
printf("Main: поток запущен\n");
```

• lpThreadId — уникальный идентификатор, присваиваемый потоку. Приблизительно соответствует свойству ManagedThreadId класса Thread.

При успешном завершении функция CreateThread возвращает дескриптор потока, который затем используется для работы с ним. Например, как аргумент функции ResumeThread и других функций.

Можно сделать вывод, что проще всего работать с потоками, используя классы .NET. Поэтому в дальнейшем большая часть примеров будет дана на языке C# с использованием .NET. Хотя синтаксис для разных .NET-совместимых языков (C++ CLI, Pascal.NET, C# и т.д.) будет отличаться, переделать примеры для нужного языка или среды разработки будет несложно. Также будут приводиться функции API Windows с аналогичной функциональностью. Для них будет использован синтаксис языка C++. Эти функции являются стандартными для ОС Windows, поэтому их синтаксис для других языков программирования (Pascal и т.д.) и дополнительные сведения можно получить в справочной системе используемой среды разработки.

3.2.1.2 Работа с потоками

Практически все операции с потоками в .NET инкапсулированы в классе Thread:

1) Получение потоков и информации о них. Мы можем получить новый поток, создав экземпляр класса Thread как в примере выше. Другой способ получить объект Thread для потока, исполняемого в данный момент, — вызов статического метода Thread.CurrentThread:

```
      Thread th = Thread.CurrentThread;

      Console.WriteLine("Текущий поток: ");
      Console.WriteLine("  Язык: {0}", th.CurrentCulture);
      Console.WriteLine("  Идентификатор: {0}", th.ManagedThreadId);
      Console.WriteLine("  Приоритет: {0}", th.Priority);
      Console.WriteLine("  Состояние: {0}", th.ThreadState);
```

Информацию о потоке можно получить и с помощью API Windows, хотя возможностей для этого меньше, а усилий требуется больше:

```
            HANDLE th = GetCurrentThread();
            LCID lcid = GetThreadLocale();
            int ln = GetLocaleInfo(lcid, LOCALE_SISO639LANGNAME, NULL, 0);
            int cn = GetLocaleInfo(lcid, LOCALE_SISO3166CTRYNAME, NULL, 0);
            char *lstr = new char [ln + 1];
            char *cstr = new char [cn + 1];
            DWORD code;
            GetLocaleInfo(lcid, LOCALE_SISO639LANGNAME, lstr, ln);
            GetLocaleInfo(lcid, LOCALE_SISO3166CTRYNAME, cstr, cn);
            GetExitCodeThread(th, &code);
            printf("Текущий поток:\n");
            printf("  Язык: %s-%s\n", lstr, cstr);
            printf("  Идентификатор: %d\n", GetCurrentThreadId());
            printf("  Приоритет: %d\n", GetThreadPriority(th));
            printf("  Состояние: %s\n", code == STILL_ACTIVE ? "running" :
"finished");
            delete [] lstr;
            delete [] cstr;
```

Если этих возможностей мало, можно использовать недокументированные функции (типа NtQuerySystemInformation с первым параметром в виде флага SYSTEM_THREAD_INFORMATION) либо класс Win32_Thread библиотеки WMI (Windows Management Instrumentation).

2) Уничтожение потоков. Уничтожить поток можно вызовом метода Thread.Abort. Исполняющая среда принудительно завершает выполнение потока, генерируя исключение ThreadAbortException. Даже если исполняемый метод потока попытается уловить ThreadAbortException, исполняющая среда этого не допустит. Однако она исполнит код из блока **finally** потока, выполнение которого прервано, если этот блок присутствует.

Необходимо отдавать себе отчет в том, что при вызове метода Thread.Abort выполнение потока не может остановиться сразу. Исполняющая среда ожидает, пока поток не достигнет *безопасной точки* (safe point). Поэтому, если наша программа зависит от некоторых действий, которые происходят после прерывания потока, и надо быть уверенными в том, что поток остановлен, не-

обходимо использовать метод Thread.Join. Это синхронный вызов, т.е. он не вернет управление, пока поток не будет остановлен.

После прерывания поток нельзя перезапустить. В этом случае, несмотря на то, что у нас есть экземпляр класса Thread, от него нет никакой пользы в плане выполнения кода.

Аналогом метода Thread.Abort в API Windows является функция TerminateThread, но использовать ее нужно лишь в крайних случаях, т.к. это приводит к утечкам ресурсов (не все ресурсы, ассоциированные с потоком, освобождаются при его уничтожении). Метод Thread.Abort в этом плане более надежен, но также может в определенных ситуациях приводить к утечкам. Аналогом метода Thread.Join является следующий вызов функции WaitForSingleObject, где «th» — дескриптор потока:

```
WaitForSingleObject(th, INFINITE);
```

3) Управление временем существования потоков. Для того чтобы приостановить («усыпить») текущий поток, используется метод Thread.Sleep, который принимает единственный аргумент, представляющий собой время (в миллисекундах) «сна» потока.

Есть еще два способа вызова метода Thread.Sleep. Первый — вызов Thread.Sleep со значением 0. При этом мы заставляем текущий поток освободить неиспользованный остаток своего кванта процессорного времени. При передаче значения Timeout.Infinite поток будет «усыплен» на неопределенно долгий срок, пока это состояние потока не будет прервано другим потоком, вызвавшим метод приостановленного потока Thread.Interrupt. В этом случае поток получит исключение ThreadInterruptedException.

Пример:

```
static void ThreadMethod1(object id)
{
      try
      {
            Console.WriteLine(id);
            Thread.Sleep(Timeout.Infinite);
      }
      catch (ThreadInterruptedException)
      {
      }
}

static int Main()
{
      Thread th1 = new Thread(new
ParameterizedThreadStart(ThreadMethod1));
```

```
        Console.WriteLine("Запуск вторичного потока th1");
        th1.Start(1);
        Thread.Sleep(5000);
        Console.WriteLine("Состояние th1: {0}", th1.ThreadState);
        Console.WriteLine("Прерываем вторичный поток th1");
        th1.Interrupt();
        Console.WriteLine("Состояние th1: {0}", th1.ThreadState);
        th1.Join();
        Console.WriteLine("Состояние th1: {0}", th1.ThreadState);
        return 0;
}
```

Вывод на консоль:

```
1
Состояние th1: WaitSleepJoin
Прерываем вторичный поток th1
Состояние th1: WaitSleepJoin
Состояние th1: Stopped
```

В этом примере поток выводит на экран переданное ему значение (1) и «засыпает» на неограниченное время. Его состояние — WaitSleepJoin. В этом состоянии находится поток, вызвавший метод Sleep или Join. При прерывании потока извне методом Interrupt генерируется исключение ThreadInterruptedException, таким образом, поток может отреагировать на прерывание. Если его прервать методом Abort, данное исключение не будет генерироваться. Затем выводим на экран состояние потока. Это все еще WaitSleepJoin — поток не достиг безопасной точки и пока не прерван (хотя на других компьютерах может оказаться, что поток уже достиг безопасной точки и успел прерваться). Поэтому вызываем метод Join. После того, как он вернет управление первичному потоку, видим, что вторичный поток действительно остановлен.

В API Windows также есть функция Sleep с возможностями, аналогичными методу Thread.Sleep. Однако для того чтобы иметь возможность прервать «спящее» состояние потока, используется либо метод SleepEx (если сигналом к пробуждению будет асинхронная операция или операция ввода-вывода), либо функции WaitFor…/MsgWaitFor…/SignalObjectAndWait (см. табл. 3.5, п. 3.2.3).

Второй способ приостановить исполнение потока — вызов метода Thread.Suspend. Между этими методами есть несколько важных отличий. Во-первых, можно вызвать метод Thread.Suspend для любого потока, а не только текущего. Во-вторых, если таким

образом приостановить выполнение потока, любой другой поток способен возобновить его выполнение с помощью метода Thread.Resume. Единственный вызов Thread.Resume возобновит исполнение данного потока независимо от числа вызовов метода Thread.Suspend, выполненных ранее:

```
      static void ThreadMethod2(object id)
      {
            while (true)
            {
                  Console.Write(id);
                  Thread.Sleep(500);
            }
      }

      static int Main()
      {
            Thread th2 = new Thread(new
ParameterizedThreadStart(ThreadMethod2));

            Console.WriteLine("Запуск вторичного потока th2");
            th2.Start(2);
            Thread.Sleep(5000);
            Console.WriteLine("\nСостояние th2: {0}", th2.ThreadState);
            Console.WriteLine("Приостанавливаем вторичный поток th2");
            th2.Suspend();
            Console.WriteLine("Состояние th2: {0}", th2.ThreadState);
            Thread.Sleep(2000);
            Console.WriteLine("Возобновляем вторичный поток th2");
            th2.Resume();
            Thread.Sleep(5000);
            Console.WriteLine("\nУничтожаем вторичный поток th2");
            th2.Abort();
            th2.Join();
            return 0;
      }
```

Вывод на консоль:

```
      Запуск вторичного потока th2
      22222222222
      Состояние th2: Running
      Приостанавливаем вторичный поток th2
      Состояние th2: SuspendRequested, WaitSleepJoin
      Возобновляем вторичный поток th2
      22222222222
      Уничтожаем вторичный поток th2
```

Результаты работы на другой машине также могут отличаться. Например, при первом выводе состояния потока можем получить не Running, а WaitSleepJoin, если в это время поток будет «спать». Также до вызова метода «th2.Suspend()» поток может еще раз успеть вывести на консоль «2» в какой-либо строке. Это лишний раз доказывает, что синхронизация и планирование потоков — достаточно сложные задачи.

Более того, компиляторы .NET, начиная с версии 3.5, уже считают методы Suspend и Resume устаревшими. Для управления потоками рекомендуется использовать такие объекты, как мониторы, семафоры и т.п.

Аналогами методов Thread.Suspend и Thread.Resume в API Windows являются функции ResumeThread и SuspendThread.

3.2.1.3 Планирование потоков

Переключение процессора на следующий поток выполняется не произвольным образом. У каждого потока есть приоритет, указывающий процессору, как должно планироваться выполнение этого потока по отношению к другим потокам системы. Для потоков, создаваемых в период выполнения, уровень приоритета по умолчанию равен Normal. Для просмотра и установки этого значения служит свойство Thread.Priority. Установщик свойства Thread.Priority принимает аргумент типа Thread.ThreadPriority, представляющего собой перечисление. Допустимые значения — Highest, AboveNormal, Normal, BelowNormal и Lowest (в порядке убывания приоритета).

Пример:

```
const int Counter = 10000;

static void ThreadMethod(object id)
{
      while (true)
      {
            for (int j = 0; j <= Counter; j++)
            {
                  for (int k = 0; k <= Counter; k++)
                  {
                        if (j == Counter && k == Counter)
                        {
                              Console.Write(id);
                        }
                  }
            }
      }
}

static int Main()
{
      Thread[] th;
      ThreadPriority[] priority = {
            ThreadPriority.Highest,
            ThreadPriority.Lowest,
      };
```

```
            th = new Thread[priority.Length];

            for (int i = 0; i < th.Length; i++)
            {
                  th[i] = new Thread(ThreadMethod);
                  th[i].Priority = priority[i];
                  th[i].Start(i + 1);
            }

            return 0;
      }
```

В данном примере количество запускаемых потоков можно регулировать, добавляя или удаляя элементы из массива priority, а количество итераций циклов — изменяя значение константы Counter. Зачем это сделано? Результаты работы программы зависят от частоты процессоров и количества процессорных ядер, установленных в системе. Если вывод данных происходит слишком медленно, необходимо уменьшить значение счетчика Counter, если слишком быстро — увеличить его. Бесполезное, в принципе, условие «**if** (j == Counter && k == Counter)» (можно ведь было вызов метода WriteLine просто разместить после циклов **for**) требуется для того, чтобы процессор или компилятор не оптимизировали пустые циклы.

Например, создадим два потока с приоритетами Highest и Lowest. Смотрим на результаты работы (прервать выполнение программы можно нажатием комбинации клавиш Ctrl+C):

```
12121212121212121212121212121212121212121212121212121212121212...
```

Несмотря на разницу в приоритетах, потоки получают одинаковое количество процессорного времени! Дело в том, что программа была запущена на ПК, имеющем один процессор с двумя ядрами, технология HyperThreading была отключена. В итоге каждый процесс мог получить в свое распоряжение по одному ядру, и конкуренции между ними не было (рис. 3.1).

Каждый поток полностью нагрузил свое ядро, в результате чего загрузка процессора достигла 100 %.

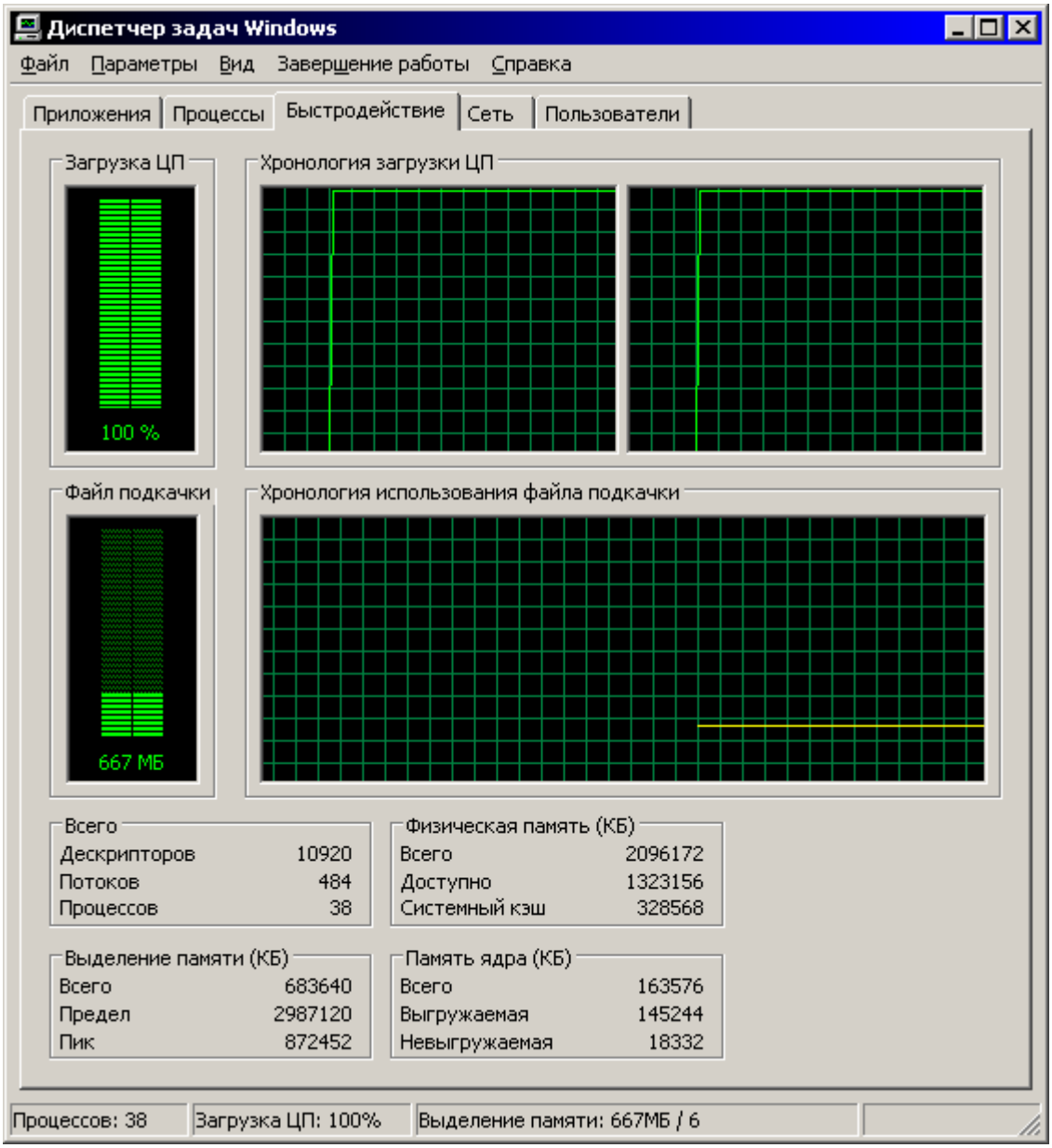

Рис. 3.1 — Диспетчер задач Windows

По умолчанию, операционная система позволяет приложению использовать все имеющиеся процессорные ядра. Однако можно вручную задать соответствие между запущенными процессами и доступными для них ядрами. Для этого необходимо вызвать контекстное меню для интересующего процесса в диспетчере задач, выбрать в нем пункт «Задать соответствие…» и в появившемся диалоге (рис. 3.2) пометить только разрешенные для процесса ядра.

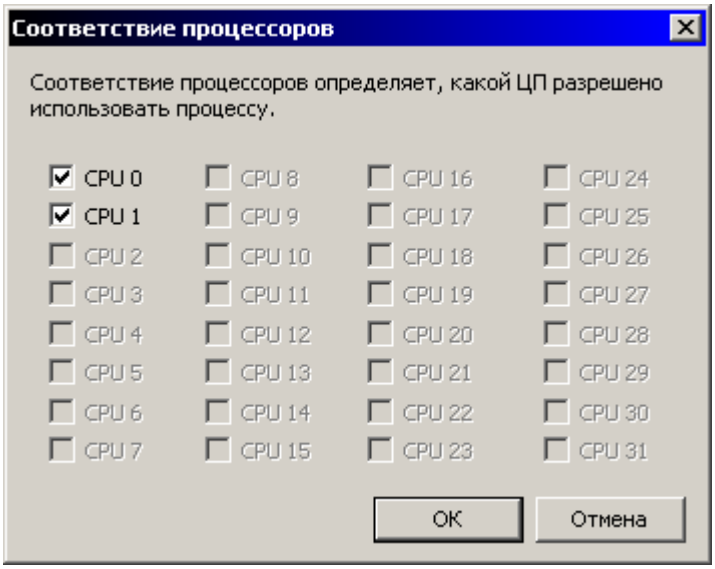

Рис. 3.2 — Соответствие процесса и процессоров

Программно задать соответствие ядер ЦП выполняемому процессу позволяет класс System.Diagnostics.Process (свойство ProcessorAffinity). В API Windows также есть функции, выполняющие аналогичные действия для процессов и потоков (см. табл. 3.4, п. 3.2.3).

Изменим условия задачи, добавив еще два потока:

```
ThreadPriority[] priority = {
      ThreadPriority.Highest,
      ThreadPriority.Lowest,
      ThreadPriority.Normal,
      ThreadPriority.Normal,
};
```

Результаты работы программы:

```
131134114311431141314131411341411341134131411231413141314141314131413...
```

В данном случае второй поток процессорного времени практически не получает, третий и четвертый получают его примерно поровну (имея одинаковый приоритет). Поэтому необходимо регулировать количество потоков и их приоритет исходя из имеющейся аппаратной конфигурации.

Изменим условие задачи еще раз. Вместо бесконечного цикла **while** сделаем цикл

```
for (int i = 0; i < 15; i++)
```

Результаты работы программы:

```
131411431143131141314131414134343434343434343222222222222222222
```

Первым свою работу завершил, как и ожидалось, первый поток, а последним — второй. Третий и четвертый потоки расположены между ними — в зависимости от ситуации, первым завершит свою работу либо один из них, либо второй.

Когда процессору указывается приоритет для потока, то его значение используется ОС как часть алгоритма планирования распределения процессорного времени. В .NET этот алгоритм основан на уровнях приоритета Thread.Priority, а также на *классе приоритета* (priority class) процесса и значении *динамического повышения приоритета* (dynamic boost). С помощью всех этих значений создается численное значение (для процессоров архитектуры x86 — от 0 до 31), представляющее реальный приоритет потока.

Класс приоритета процесса задается свойством PriorityClass, а динамическое повышение приоритета — свойством PriorityBoostEnabled класса System.Diagnostics.Process. Например:

```
using System;
using System.Diagnostics;

class Program
{
      static int Main()
      {
            Process proc = Process.GetCurrentProcess();

            proc.PriorityClass = ProcessPriorityClass.AboveNormal;
            proc.PriorityBoostEnabled = true;
            Console.WriteLine(proc.BasePriority);
            return 0;
      }
}
```

Класс приоритета процесса определяет *базовый приоритет процесса* (в табл. 3.1 перечислены в порядке возрастания приоритета). Базовый приоритет процесса можно узнать, используя доступное только для чтения свойство BasePriority класса Process.

Таблица 3.1 — Класс приоритета и базовый приоритет процесса

Класс приоритета	Базовый приоритет
`Idle`	4
`BelowNormal`	6
`Normal`	8
`AboveNormal`	10
`High`	13
`RealTime`	24

Реальный приоритет потока получается сложением базового приоритета процесса и собственного приоритета потока (табл. 3.2).

Таблица 3.2 — Приоритеты потоков

Класс приоритета процесса / Приоритет потока	Idle	BelowNormal	Normal	AboveNormal	High	RealTime
TimeCritical	15	15	15	15	15	31
Highest	6	8	10	12	15	26
AboveNormal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
BelowNormal	3	5	7	9	12	23
Lowest	2	4	6	8	11	22
Idle	1	1	1	1	1	16

В табл. 3.2 не показано, как задать уровень приоритета 0. Это связано с тем, что нулевой приоритет зарезервирован для потока обнуления страниц, и никакой другой поток не может иметь такой приоритет. Кроме того, уровни 17—21 и 27—30 в обычном приложении тоже недоступны. Ими может пользоваться только драйвер устройства, работающий в режиме ядра. Уровень приоритета потока в процессе с классом real-time не может опускаться ниже 16, а потока в процессе с любым другим классом — подниматься выше 15.

Свойство PriorityBoostEnabled используется для временного увеличения уровня приоритета потоков, взятых из состояния ожидания. Приоритет сбрасывается при возвращении процесса в состояние ожидания.

Функции для работы с приоритетом и динамическим повышением приоритета процессов и потоков перечислены в п. 3.2.3 (табл. 3.4).

Несколько потоков с одинаковым приоритетом получают равное количество процессорного времени. Это называется *циклическим планированием* (round robin scheduling).

3.2.1.4 Пул потоков

Если имеются небольшие задачи, которые нуждаются в фоновой обработке, *пул управляемых потоков* — это самый простой способ воспользоваться преимуществами нескольких потоков. Статический класс ThreadPool обеспечивает приложение пулом рабочих потоков, управляемых системой, позволяя пользователю сосредоточиться на выполнении задач приложения, а не на управлении потоками.

Потоки из пула потоков являются фоновыми потоками. Для каждого потока используется размер стека и приоритет по умолчанию. Для каждого процесса можно использовать только один пул потоков.

Количество операций, которое может быть помещено в очередь пула потоков, ограничено только объемом памяти, однако пул потоков ограничивает количество потоков, которое может быть одновременно активно в процессе. По умолчанию это ограничение составляет 500 рабочих потоков на ЦП и 1000 потоков асинхронного ввода/вывода (зависит от версии .NET Framework). Можно управлять максимальным количеством потоков с помощью методов GetMaxThreads и SetMaxThreads. Также можно задавать минимальное количество потоков с помощью методов GetMinThreads и SetMinThreads. Пользоваться этими методами следует с осторожностью, т.к. если задать большое минимальное число потоков и часть из них не будет использована, это приведет к ненужному расходу системных ресурсов.

Добавление в пул рабочих потоков производится путем вызова метода QueueUserWorkItem и передачи делегата WaitCallback, представляющего метод, который выполняет задачу:

```
static bool QueueUserWorkItem(WaitCallback callBack);
static bool QueueUserWorkItem(WaitCallback callBack, object state);
delegate void WaitCallback(object state);
```

Пример:

```
const int Counter = 10000;

static void ThreadMethod(object id)
{
      for (int i = 0; i < 15; i++)
      {
            for (int j = 0; j <= Counter; j++)
            {
                  for (int k = 0; k <= Counter; k++)
                  {
```

```
                              if (j == Counter && k == Counter)
                              {
                                    Console.Write(id);
                              }
                        }
                  }
            }
      }

      static int Main()
      {
            int min_cpu, min_io, max_cpu, max_io;

            ThreadPool.GetMinThreads(out min_cpu, out min_io);
            ThreadPool.GetMaxThreads(out max_cpu, out max_io);
            Console.WriteLine("Потоки ЦП: {0}..{1}", min_cpu, max_cpu);
            Console.WriteLine("Асинхронные потоки: {0}..{1}", min_io,
max_io);

            for (int i = 0; i < 4; i++)
            {
                  ThreadPool.QueueUserWorkItem(ThreadMethod, i + 1);
            }

            Console.ReadKey(true);
            return 0;
      }
```

Вывод на консоль:

```
Потоки ЦП: 2..500
Асинхронные потоки: 2..1000
12121212112313121312132413241342341234124342434243424343434343
```

Результаты аналогичны предыдущему примеру, только приоритеты всех потоков одинаковы (Normal). Здесь в пуле запускаются четыре потока, но из-за ограничений ЦП (2 ядра) первыми выполняются 1-й и 2-й потоки, а уже затем 3-й и 4-й. Окончание работы главного потока приведет к прерыванию работы всех потоков в пуле. Поэтому нажатие на любую клавишу прервет работу. В принципе, вместо вызова метода ReadKey можно «усыпить» основной поток, например вызвать метод Thread.Sleep(5000). Тогда на выполнение потоков в пуле будет отведено 5 секунд. Если же мы хотим гарантированно дождаться выполнения работы всеми потоками, используем метод, возвращающий количество доступных потоков (GetAvailableThreads):

```
      int cur_cpu, cur_io;

      do
      {
            Thread.Sleep(100);
            ThreadPool.GetAvailableThreads(out cur_cpu, out cur_io);
      } while (cur_cpu != max_cpu);

      Console.WriteLine("\nВсе потоки завершили свою работу");
```

Заметим, что в цикле основной поток «усыпляется» на 100 мс. Это сделано для того, чтобы дать поработать другим потокам. Иначе постоянный запрос количества доступных потоков в цикле будет приводить к трате процессорного времени.

Либо для этих целей можно использовать объекты типа семафора:

```
      const int Counter = 10000;
      static Semaphore SemaPool;

      static void ThreadMethod(object id)
      {
            for (int i = 0; i < 15; i++)
            {
                  for (int j = 0; j <= Counter; j++)
                  {
                        for (int k = 0; k <= Counter; k++)
                        {
                              if (j == Counter && k == Counter)
                              {
                                    Console.Write(id);
                              }
                        }
                  }
            }
            SemaPool.Release();
      }

      static int Main()
      {
            int min_cpu, min_io, max_cpu, max_io;

            ThreadPool.GetMinThreads(out min_cpu, out min_io);
            ThreadPool.GetMaxThreads(out max_cpu, out max_io);
            Console.WriteLine("Потоки ЦП: {0}..{1}", min_cpu, max_cpu);
            Console.WriteLine("Асинхронные потоки: {0}..{1}", min_io,
max_io);
            SemaPool = new Semaphore(4, 4);

            for (int i = 0; i < 4; i++)
            {
                  SemaPool.WaitOne();
                  ThreadPool.QueueUserWorkItem(ThreadMethod, i + 1);
            }

            for (int i = 0; i < 4; i++)
            {
                  SemaPool.WaitOne();
            }

            Console.WriteLine("\nВсе потоки завершили свою работу");
            Console.ReadKey(true);
            return 0;
      }
```

Добавление в пул асинхронных потоков производится путем вызова метода RegisterWaitForSingleObject и передачи ему объекта синхронизации WaitHandle. Этот объект ждет наступления некоторого события, и при его наступлении или при истечении времени ожидания вызывает метод, представленный делегатом WaitOrTimerCallback:

```
static RegisteredWaitHandle RegisterWaitForSingleObject(
      WaitHandle waitObject, WaitOrTimerCallback callBack,
      object state, int timeOutInterval, bool executeOnlyOnce);
delegate void WaitOrTimerCallback(object state, bool timedOut);
```

Объект синхронизации — это экземпляр класса WaitHandle или его потомка:

```
System.Threading.WaitHandle
├── System.Threading.EventWaitHandle
│   ├── System.Threading.AutoResetEvent
│   └── System.Threading.ManualResetEvent
├── System.Threading.Mutex
└── System.Threading.Semaphore
```

Пример:

```
const int Counter = 10000;

class WaitObject
{
      public bool TimeOut = false;
      public int ID = 1;
      public RegisteredWaitHandle Handle;

      public static void CallbackMethod(object state, bool timeout)
      {
            WaitObject obj = (WaitObject)state;
            int id = obj.ID++;

            if (!timeout)
            {
                  Console.WriteLine("\nПоток #{0} получил сигнал о
запуске", id);
                  for (int i = 0; i < 15; i++)
                  {
                        for (int j = 0; j <= Counter; j++)
                        {
                              for (int k = 0; k <= Counter; k++)
                              {
                                    if (j == Counter && k == Counter)
                                    {
                                          Console.Write(id);
                                    }
                              }
                        }
                  }
            }
            else
            {
```

```
                        Console.WriteLine("\nВремя ожидания закончилось");
                        obj.TimeOut = true;
                        obj.Handle.Unregister(null);
                  }
            }
      }

      static int Main()
      {
            int max_cpu, max_io, cur_cpu, cur_io;
            WaitObject obj = new WaitObject();
            AutoResetEvent are = new AutoResetEvent(false);
            const int wait = 10;
            char key;

            Console.WriteLine("Для запуска потока в пуле нажмите S");
            Console.WriteLine("Для отмены ожидания новых потоков - U");
            Console.Write("Через {0} сек ожидание будет окончено", wait);
            obj.Handle = ThreadPool.RegisterWaitForSingleObject(are,
                  WaitObject.CallbackMethod, obj, wait * 1000, false);

            do
            {
                  if (Console.KeyAvailable)
                  {
                        key = Console.ReadKey(true).KeyChar;
                        if (key == 'S' || key == 's')
                        {
                              are.Set();
                        }
                        else if (key == 'U' || key == 'u')
                        {
                              obj.Handle.Unregister(null);
                              Console.WriteLine("\nОжидание отменено");
                              break;
                        }
                  }
                  else
                  {
                        Thread.Sleep(100);
                  }
            } while (!obj.TimeOut);

            ThreadPool.GetMaxThreads(out max_cpu, out max_io);
            ThreadPool.GetAvailableThreads(out cur_cpu, out cur_io);
            if (cur_io != max_io)
            {
                  Console.WriteLine("\nПодождите, пока все потоки завершат
свою работу");
                  do
                  {
                        Thread.Sleep(100);
                        ThreadPool.GetAvailableThreads(out cur_cpu, out
cur_io);
                  } while (cur_io != max_io);
                  Console.WriteLine("\nВсе потоки завершили свою работу");
            }

            Console.ReadKey(true);
            return 0;
      }
```

В данном примере потоки в пуле запускаются при поступлении сигнала. Сигнал эмулируется вызовом метода AutoResetEvent.Set() при нажатии на клавишу «S». При нажатии клавиши «U» или если в течение 10 секунд (значение константы wait) не поступают сигналы, регистрация новых сигналов прекращается (вызовом метода RegisteredWaitHandle.Unregister). Иначе основной поток «засыпает» на 100 мс, чтобы, как уже было сказано, не потреблять системные ресурсы и дать поработать асинхронным потокам в пуле. Затем опрос клавиатуры повторяется.

Если время ожидания новых потоков истекло, то регистрация в методе CallbackMethod также отменяется, хотя это делать не обязательно. Если регистрацию при значении timeout = **false** не отменять, то новые сообщения будут продолжать приниматься. Спустя 10 секунд, если новых сообщений о регистрации не будет, CallbackMethod с параметром timeout = **false** будет вызван повторно и т.д.

Для завершения всех потоков в пуле снова используем значения, возвращаемые методом GetAvailableThreads. Но проверяем не количество рабочих потоков, как в предыдущем примере, а количество асинхронных потоков ввода-вывода.

Пример вывода на консоль (зависит от нажатых клавиш):

```
Для запуска потока в пуле нажмите S
Для отмены ожидания новых потоков - U
Через 10 сек ожидание будет окончено
Поток #1 получил сигнал о запуске
11
Поток #2 получил сигнал о запуске
Поток #3 получил сигнал о запуске
1233132312331
Поток #4 получил сигнал о запуске
2343321
Поток #5 получил сигнал о запуске
3452135432153124352351455125455152455
Время ожидания закончилось
Подождите, пока все потоки завершат свою работу
2514521424244444
Все потоки завершили свою работу
```

Подробную информацию о данных классах и их членах смотрите в библиотеке MSDN.

В API Windows пула потоков, как такового, нет. Для создания пула рабочих потоков можно вручную создавать массив потоков вызовом функции CreateThread, а затем ожидать окончания их выполнения вызовом функции WaitForMultipleObjects(Ex) или

MsgWaitForMultipleObjects(Ex). Для создания очереди асинхронных операций используется функция QueueUserAPC. В качестве сигнальных объектов могут выступать как сами процессы и потоки, так и семафоры, мьютексы, таймеры и события (табл. 3.5, п. 3.2.3). Также для организации пула потоков можно использовать порты завершения ввода/вывода (п. 3.2.3.5).

3.2.2 Безопасность и синхронизация потоков

Что произойдет при попытке одновременного доступа к объекту нескольких потоков? Проверим:

```
using System;
using System.Text;
using System.Threading;

namespace ThreadSynchronizeSample
{
      class Program
      {
            static StringBuilder sb;

            static void ThreadMethod(object id)
            {
                  for (int i = 0; i < sb.Length; i++)
                  {
                        sb[i] = (char)id;
                        Thread.Sleep(100);
                  }
            }

            static int Main()
            {
                  Thread th1 = new Thread(ThreadMethod);
                  Thread th2 = new Thread(ThreadMethod);

                  sb = new StringBuilder("--------------------");
                  th1.Start(1);
                  th2.Start(2);
                  th1.Join();
                  th2.Join();
                  Console.WriteLine(sb);
                  return 0;
            }
      }
}
```

Вывод на консоль:

```
21111121121212121121
```

Два потока модифицировали содержимое одного и того же объекта, в результате чего его состояние стало неопределенным.

Как избежать подобных непредсказуемых состояний? Существует стандартный способ решения этой проблемы — *синхронизация*. Синхронизация позволяет создавать *критические секции* (critical sections) кода, в которые в каждый отдельный момент может входить только один поток, гарантируя, что любые временные недействительные состояния вашего объекта будут невидимы его клиентам. То есть выполнение действий внутри критической секции является *атомарной операцией* с точки зрения других потоков.

Например, для асинхронных потоков в пуле мы использовали временную переменную «id = obj.ID++» в теле метода CallbackMethod. Если бы вместо id использовалось поле obj.ID, то не было бы гарантии, что во время работы этого метода для какого-либо потока значение поля не изменилось бы другим потоком. Но даже операция инкремента не является атомарной, т.к. состоит из трех шагов:

1) загрузить значение из экземпляра переменной в регистр;

2) увеличить значение регистра на 1;

3) сохранить значение в экземпляре переменной.

Другой поток может вклиниться между любой из этих операций.

3.2.2.1 Защита кода с помощью класса Monitor

Статический класс System.Threading.Monitor контролирует доступ к объектам, предоставляя блокировку объекта одному потоку. Блокировки объектов предоставляют возможность ограничения доступа к критической секции. Пока поток владеет блокировкой для объекта, никакой другой поток не может ею завладеть. Для управления блокировкой чаще всего используются два метода:

```
static void Enter(object obj);
static void Exit(object obj);
```

Первый метод пытается получить блокировку монитора для указанного объекта. Если у другого потока уже есть эта блокировка, текущий поток блокируется до тех пор, пока блокировка не будет освобождена. Объект должен являться экземпляром ссылочного типа. Второй метод снимает блокировку.

Изменим метод ThreadMethod в примере выше:

```
Monitor.Enter(sb);
for (int i = 0; i < sb.Length; i++)
{
      sb[i] = (char)id;
      Thread.Sleep(100);
}
Console.WriteLine(sb);
Monitor.Exit(sb);
```

Теперь два потока не смогут одновременно изменять строку:

```
11111111111111111111
22222222222222222222
22222222222222222222
```

Поток с идентификатором «2» был запущен вторым, поэтому в итоге в строке остались двойки. В другой ситуации вторым может запуститься поток с идентификатором «1».

Для того, чтобы синхронизировать коллекции, следует в методы Enter и Exit передавать ссылку не на саму коллекцию, а на ее свойство SyncRoot, наследуемое от интерфейса ICollection. Например, класс Array реализует интерфейс ICollection, поэтому все массивы синхронизируются следующим образом:

```
int[] mas = new int[100];
Monitor.Enter(mas.SyncRoot);
// безопасная обработка массива
Monitor.Exit(mas.SyncRoot);
```

В API Windows для этих целей используются критические секции. Сначала критическую секцию необходимо создать (InitializeCriticalSection), затем, для установки блокировки, войти в нее (EnterCriticalSection или TryEnterCriticalSection). Для снятия блокировки — выйти из критической секции (LeaveCriticalSection) и затем удалить ее (DeleteCriticalSection).

3.2.2.2 Применение блокировок монитора оператором lock

Блокировать объект также позволяет оператор языка C# **lock**:

```
<оператор блокировки> :: lock (<выражение>) <внедряемый оператор>
```

Фактически, это синтаксическое сокращение для вызовов методов Monitor.Enter и Monitor.Exit в рамках блока **try-finally**:

```
Monitor.Enter(<выражение>);
try
{
      <внедряемый оператор>
}
finally
{
      Monitor.Exit(<выражение>);
}
```

Следующий пример даст результат, аналогичный предыдущему:

```
static StringBuilder sb;

static void ThreadMethod(object id)
{
      lock (sb)
      {
            for (int i = 0; i < sb.Length; i++)
            {
                  sb[i] = (char)id;
                  Thread.Sleep(100);
            }
            Console.WriteLine(sb);
      }
}

static int Main()
{
      Thread th1 = new Thread(ThreadMethod);
      Thread th2 = new Thread(ThreadMethod);

      sb = new StringBuilder("--------------------");
      th1.Start('1');
      th2.Start('2');
      th1.Join();
      th2.Join();
      Console.WriteLine(sb);

      int[] mas = new int[100];
      lock (mas.SyncRoot)
      {
            // безопасная обработка массива
      }

      return 0;
}
```

Как правило, рекомендуется избегать блокировки членов типа **public** или экземпляров, которыми код не управляет. Например:

- **lock** (**this**) может привести к проблеме, если к экземпляру допускается открытый доступ извне потока;
- **lock** (**typeof** (MyType)) может привести к проблеме, если к MyType допускается открытый доступ извне потока;

- **lock** ("myLock") может привести к проблеме, поскольку любой код в процессе, используя ту же строку, будет совместно использовать ту же блокировку.

Поток может неоднократно блокировать один и тот же объект многократными вызовами метода Monitor.Enter или вложенными операторами **lock**. Объект будет освобожден, когда соответствующее количество раз будет вызван метод Monitor.Exit или произойдет выход из самой внешней конструкции **lock**.

3.2.2.3 Синхронизация кода с помощью классов Mutex и Semaphore

Также синхронизацию кода можно выполнить, используя функциональность мьютексов (класс System.Threading.Mutex) и семафоров (класс System.Threading.Semaphore). Они работают медленнее, зато более универсальны — в частности, доступны из других процессов, поэтому блокировка осуществляется на уровне системы, а не отдельного процесса. Отличия этих двух классов в том, что мьютекс (от англ. mutual exclusion, взаимное исключение) гарантирует использование блокированного ресурса только одним потоком, а семафор позволяет нескольким потокам получить доступом к пулу ресурсов. Количество потоков, которые могут войти в семафор, ограничено. Счетчик на семафоре уменьшается на единицу каждый раз, когда в семафор входит поток, и увеличивается на единицу, когда поток освобождает семафор. Когда счетчик равен нулю, последующие запросы блокируются, пока другие потоки не освободят семафор. Когда семафор освобожден всеми потоками, счетчик имеет максимальное значение, заданное при создании семафора. В принципе, семафор со счетчиком, равным 1, соответствует мьютексу, только не имеет потока-хозяина.

Программисты Win32, которые занимались блокировкой ресурсов при написании программ на C++ (в частности, запрет запуска второй копии приложений и т.п.), должны помнить основы работы с мьютексами и семафорами. В API Win32 это были не классы, а набор функций, поэтому создавались они вызовом функции CreateMutex и CreateSemaphore, а удалялись — ReleaseMutex и ReleaseSemaphore (см. табл. 3.5, п. 3.2.3).

Перепишем предыдущий пример с использованием мьютекса. Дополнительно встроим защиту от повторного запуска программы:

```
      static StringBuilder sb;
      static Mutex mux;

      static void ThreadMethod(object id)
      {
            mux.WaitOne();
            for (int i = 0; i < sb.Length; i++)
            {
                  sb[i] = (char)id;
                  Thread.Sleep(100);
            }
            Console.WriteLine(sb);
            mux.ReleaseMutex();
      }

      static int Main()
      {
            Mutex mutex = new Mutex(false, "c#_sample_mutex");

            if (!mutex.WaitOne(1000, false))
            {
                  Console.WriteLine("В системе запущен другой экземпляр
программы!");
                  return 1;
            }

            Thread th1 = new Thread(ThreadMethod);
            Thread th2 = new Thread(ThreadMethod);

            sb = new StringBuilder("--------------------");
            mux = new Mutex();
            th1.Start('1');
            th2.Start('2');
            th1.Join();
            th2.Join();
            Console.WriteLine(sb);
            return 0;
      }
```

Для демонстрации работы семафора перепишем класс, использующий пул потоков, чтобы одновременно запускалось не больше потоков, чем имеется процессорных ядер в системе:

```
      const int Counter = 10000;
      static Semaphore Sema;

      static void ThreadMethod(object id)
      {
            Sema.WaitOne();
            for (int i = 0; i < 15; i++)
            {
                  for (int j = 0; j <= Counter; j++)
                  {
                        for (int k = 0; k <= Counter; k++)
                        {
```

```
                              if (j == Counter && k == Counter)
                              {
                                    Console.Write(id);
                                    Thread.Sleep(100);
                              }
                        }
                  }
            }
            Sema.Release();
      }

      static int Main()
      {
            int max_cpu, max_io, cur_cpu, cur_io;

            ThreadPool.GetMaxThreads(out max_cpu, out max_io);
            Sema = new Semaphore(Environment.ProcessorCount,
Environment.ProcessorCount);

            for (int i = 0; i < 4; i++)
            {
                  ThreadPool.QueueUserWorkItem(ThreadMethod, i + 1);
            }

            do
            {
                  Thread.Sleep(100);
                  ThreadPool.GetAvailableThreads(out cur_cpu, out cur_io);
            } while (cur_cpu != max_cpu);

            Console.WriteLine("\nВсе потоки завершили свою работу");
            return 0;
      }
```

Если семафор убрать, то потоки будут работать вперемешку. С семафором сначала отработает столько потоков, сколько имеется процессорных ядер в системе, а потом, по мере их завершения, запустятся остальные. При необходимости количество потоков в пуле можно увеличить.

3.2.2.4 Атомарные операции с классом Interlocked

Как уже было сказано выше, даже простейшие с точки зрения программиста операции (инкремент, декремент и т.д.) не являются атомарными с точки зрения потоков. А блокировка оператором **lock** или классом Monitor работает только для ссылочных объектов. Для того, чтобы сделать выполнение некоторых арифметических операций с числами атомарными, используется класс System.Threading.Interlocked (табл. 3.3).

Таблица 3.3 — Методы класса Interlocked

Метод	Описание
static int Add(**ref int** location, **int** value) **static long** Add(**ref long** location, **long** value)	Сложение двух чисел и помещение результата в первый аргумент как атомарная операция
static <тип> CompareExchange(**ref** <тип> location, <тип> value, <тип> comparand)	Если location = comparand, то произойдет замена value на location как атомарная операция(*)
static int Decrement(**ref int** location) **static long** Decrement(**ref long** location)	Уменьшение значения заданной переменной и сохранение результата как атомарная операция
static <тип> Exchange(**ref** <тип> location, <тип> value)	Значение value заменяет location как атомарная операция(*)
static int Increment(**ref int** location) **static long** Increment(**ref long** location)	Увеличение значения заданной переменной и сохранение результата как атомарная операция
static long Read(**ref long** location)	Возвращает 64-битное значение, загруженное в виде атомарной операции

Примечание: (*) Есть версии для различных типов данных и универсальный метод.

Функции, выполняющие аналогичные действия, есть и в API Windows (см. табл. 3.5, п. 3.2.3).

3.2.2.5 Использование модификатора volatile

Ключевое слово **volatile** указывает, что поле может быть изменено несколькими потоками, выполняющимися одновременно. Поля, объявленные как **volatile**, не проходят оптимизацию компилятором, которая предусматривает доступ посредством отдельного потока. Это гарантирует наличие наиболее актуального значения в поле в любое время.

Как правило, модификатор **volatile** используется для поля, обращение к которому выполняется из нескольких потоков без использования оператора **lock**.

Ключевое слово **volatile** в языке C# можно применять к полям следующих типов:

- ссылочным типам;
- типам указателей (в небезопасном контексте);
- типам **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **char**, **float** и **bool**;
- типу перечисления с одним из следующих базовых типов: **byte**, **sbyte**, **short**, **ushort**, **int** или **uint**;
- параметрам универсальных типов, являющихся ссылочными типами;
- IntPtr и UIntPtr.

Ключевое слово **volatile** в языке C# можно применить только к полям класса или структуры. Локальные переменные не могут быть объявлены как **volatile**. Подобных ограничений нет в языке C++.

3.2.2.6 Потокобезопасность классов

Почти все типы .NET Framework, не являющиеся примитивными, также не являются и потокобезопасными, и все же они могут использоваться в многопоточном коде, если доступ к любому объекту защищен блокировкой.

Перечисление коллекций также не является потокобезопасной операцией, так как если другой поток меняет список в процессе перечисления, генерируется исключение. Однако даже если бы коллекции были полностью потокобезопасными, это изменило бы немногое. Для примера рассмотрим добавление элемента к гипотетической потокобезопасной коллекции:

```
if (!myCollection.Contains(newItem)) myCollection.Add(newItem);
```

Независимо от потокобезопасности собственно коллекции, данная конструкция в целом определенно не потокобезопасна. Заблокирован должен быть весь этот код целиком, чтобы предотвратить вытеснение потока между проверкой и добавлением но-

вого элемента. Также блокировка должна быть использована везде, где изменяется список. К примеру, следующая конструкция должна быть обернута в блокировку для гарантии, что ее исполнение не будет прервано:

```
myCollection.Clear();
```

Другими словами, блокировки пришлось бы использовать точно так же, как с существующими потоконебезопасными классами.

Хуже всего дела обстоят со статическими полями с модификатором **public**. Для примера представим, что какое-либо статическое свойство структуры DateTime (например, DateTime.Now) потоконебезопасное и два параллельных запроса могут привести к неправильным результатам или исключению. Единственная возможность исправить положение с использованием внешней блокировки — использовать конструкцию **lock** (**typeof** (DateTime)) при каждом обращении к DateTime.Now. Но мы не можем заставить делать это каждого программиста. Кроме того, как мы уже говорили, рекомендуется избегать блокировки типов. По этой причине статические поля структуры DateTime гарантированно потокобезопасны. Это обычное поведение типов в .NET Framework — статические члены потокобезопасны, нестатические — нет. Так же следует проектировать и наши собственные типы.

3.2.3 Использование потоков в API Windows

Перечислим основные функции для работы с процессами и потоками в API Windows (табл. 3.4), а также функции для синхронизации процессов и потоков (табл. 3.5).

Таблица 3.4 — Основные функции для работы с процессами и потоками в API Windows

Функция	Описание
CloseHandle	Удаление дескриптора процесса или потока
CreateThread	Создание нового потока
ExitThread	Завершение потока
GetCurrentProcess	Получение дескриптора текущего процесса

Окончание табл. 3.4

Функция	Описание
GetCurrentThread	Получение дескриптора текущего потока
GetCurrentThreadId	Получение идентификатора текущего потока
GetExitCodeThread	Информация о текущем статусе потока
GetPriorityClass	Получение класса приоритета процесса
GetProcessAffinityMask	Получение информации о процессорных ядрах, на которых может выполняться процесс
GetProcessPriorityBoost	Получение информации о динамическом повышении приоритета процесса
GetProcessTimes	Получение информации о времени выполнения процесса
GetThreadPriority	Получение приоритета потока
GetThreadPriorityBoost	Получение информации о динамическом повышении приоритета потока
GetThreadTimes	Получение информации о времени выполнения потока
ResumeThread	Продолжение выполнения приостановленного потока
SetPriorityClass	Изменение класса приоритета процесса
SetProcessAffinityMask	Изменение соответствия процессорных ядер, на которых может выполняться процесс
SetProcessPriorityBoost	Изменение параметра динамического повышения приоритета процесса
SetThreadAffinityMask	Изменение соответствия процессорных ядер, на которых может выполняться поток
SetThreadIdealProcessor	Установка предпочитаемого процессорного ядра для работы потока
SetThreadPriority	Изменение приоритета потока
SetThreadPriorityBoost	Изменение параметра динамического повышения приоритета потока
Sleep	Приостановление работы потока на определенное время
SleepEx	Приостановление работы потока с возможностью выйти из этого состояния по сигналу
SuspendThread	Приостановление работы потока
SwitchToThread	Переключение на следующий поток, ожидающий выполнения на данном процессорном ядре
TerminateThread	Прерывание выполнения потока

Таблица 3.5 — Основные функции для синхронизации процессов и потоков в API Windows

Функция	Описание
CancelWaitableTimer	Отключение таймера ожидания
CloseHandle	Удаление дескриптора таймера ожидания, события, мьютекса, семафора
CreateEvent	Создание именованного или безымянного события
CreateIoCompletionPort	Создание порта завершения ввода/вывода
CreateMutex	Создание именованного или безымянного мьютекса
CreateSemaphore	Создание именованного или безымянного семафора
CreateWaitableTimer	Создание таймера ожидания
DeleteCriticalSection	Удаление ресурсов критической секции
EnterCriticalSection	Вход в критическую секцию
GetQueuedCompletionStatus	Получение сообщения о завершении асинхронной операции ввода/вывода
InitializeCriticalSection	Инициализация ресурсов критической секции
InterlockedCompareExchange InterlockedDecrement InterlockedExchange InterlockedExchangeAdd InterlockedIncrement	Выполнение атомарных операций (см. п. 3.2.2.4)
LeaveCriticalSection	Выход из критической секции
MsgWaitForMultipleObjects MsgWaitForMultipleObjectsEx WaitForMultipleObjects WaitForMultipleObjectsEx	Ожидание поступления сигнала от одного или всех объектов из множества объектов — событий, мьютексов, процессов, потоков, семафоров, таймеров и т.п.
OpenEvent	Получение дескриптора именованного события
OpenMutex	Получение дескриптора именованного мьютекса
OpenSemaphor	Получение дескриптора именованного семафора
OpenWaitableTimer	Получение дескриптора таймера ожидания
PostQueuedCompletionStatus	Отправление сообщения о завершении асинхронной операции ввода/вывода
PulseEvent	Установка события в сигнальное состояние с последующим сбросом

Окончание табл. 3.5

Функция	Описание
QueueUserAPC	Добавление в очередь асинхронного вызова процедуры
ReleaseMutex	Освобождение ресурсов мьютекса
ReleaseSemaphore	Увеличение счетчика семафора
ResetEvent	Сброс сигнального состояния события
SetEvent	Установка события в сигнальное состояние
SetWaitableTimer	Активация таймера ожидания
SignalObjectAndWait	Подача сигнала другому объекту и ожидание его завершения
TryEnterCriticalSection	Попытка входа в критическую секцию
WaitForSingleObject WaitForSingleObjectEx	Ожидание поступления сигнала от объекта — события, мьютекса, процесса, потока, семафора, таймера и т.п.

Дополнительные сведения можно получить в MSDN или справочной системе используемой среды разработки.

3.2.3.1 Создание и запуск потоков

Потоки создаются с помощью функции CreateThread, куда передается указатель на функцию (функцию потока), которая будет выполняться в созданном потоке. Функция CreateThread возвращает специальное значение типа HANDLE — дескриптор потока, который может быть использован для приостановки, уничтожения потока, синхронизации. Поток считается завершенным, когда выполнится функция потока.

Если же требуется гарантировать завершение потока, то можно воспользоваться функцией TerminateThread, которая «убивает» поток, что не всегда корректно. Функция ExitThread будет вызвана неявно, когда завершится функция потока, или же можно вызвать данную функцию самостоятельно. Главная ее задача — освободить стек потока и его дескриптор, то есть структуры ядра, которые обслуживают данный поток.

Поток может пребывать в состоянии сна (suspend). Чтобы «усыпить» поток (приостановить поток извне или из самого потока), используется функция SuspendThread. «Пробуждение» (продолжение выполнения) потока возможно с помощью вызова

функции ResumeThread. Поток можно перевести в состояние сна при создании. Для этого нужно передать в CreateThread значение флага CREATE_SUSPENDED в предпоследнем аргументе.

Таким образом, в результате выполнения функции CreateThread будет создан новый поток, функция которого начнет выполняться либо сразу же, либо будет приостановлена до вызова ResumeThread. При создании каждому потоку также назначается уникальный идентификатор.

Повторный вызов CreateThread приведет к созданию еще одного потока, выполняющегося одновременно с созданным ранее, и т. д. Таким образом можно создавать неограниченное число потоков, но каждый новый поток тормозит выполнение остальных.

Для ожидания окончания выполнения потока можно использовать функцию WaitForSingleObject.

3.2.3.2 Механизмы синхронизации ОС Windows

В Win32 существуют средства синхронизации двух типов:

– реализованные на уровне пользователя (критические секции — critical sections);

– реализованные на уровне ядра (мьютексы — Mutex, события — Event, семафоры — Semaphore).

Общие черты механизмов синхронизации:

– используют примитивы ядра при выполнении, что сказывается на производительности;

– могут быть именованными и неименованными;

– работают на уровне системы, то есть могут служить механизмом межпроцессного взаимодействия (IPC);

– используют для ожидания и захвата примитива единую группу функций WaitFor…/MsgWaitFor…

Существуют несколько стратегий, которые могут применяться, чтобы разрешать проблемы, связанные с взаимодействием потоков. Наиболее распространенным способом является синхронизация потоков, суть которой состоит в том, чтобы вынудить один поток ждать, пока другой не закончит какую-то определенную заранее операцию. Для этой цели существуют специальные синхронизирующие объекты ядра операционной системы Win-

dows. Они исключают возможность одновременного доступа к тем данным, которые с ними связаны. Их реализация зависит от конкретной ситуации и предпочтений программиста.

Общие положения использования объектов ядра системы:

– однажды созданный объект ядра можно открыть в любом приложении, если оно имеет соответствующие права доступа к нему;

– каждый объект ядра имеет счетчик числа своих пользователей. Как только он станет равным нулю, система уничтожит объект ядра;

– обращаться к объекту ядра надо через дескриптор (HANDLE), который система дает при создании объекта;

– каждый объект может находиться в одном из двух состояний: свободном (signaled) или занятом (nonsignaled).

1) Работа с объектом «критическая секция» (critical section). Это самые простые объекты ядра Windows, которые не снижают общей эффективности приложения. Пометив блок кодов в качестве критической секции, можно синхронизировать доступ к нему от нескольких потоков.

Для работы с критическими секциями есть ряд функций API Windows и структура CRITICAL_SECTION. Алгоритм использования следующий:

1. Объявить глобальную структуру

```
CRITICAL_SECTION cs;
```

2. Инициализировать (обычно это делается один раз, перед тем как начнется работа с разделяемым ресурсом) глобальную структуру вызовом функции

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

3. Поместить охраняемую часть программы внутрь блока, который начинается вызовом функции EnterCriticalSection и заканчивается вызовом LeaveCriticalSection:

```
EnterCriticalSection(&cs);
// охраняемый блок кода
LeaveCriticalSection(&cs);
```

Функция EnterCriticalSection, анализируя поле структуры «cs», которое является счетчиком ссылок, выясняет, вызвана ли она в первый раз. Если да, то функция увеличивает значение

счетчика и разрешает выполнение потока дальше. При этом выполняется блок, модифицирующий критические данные. Допустим, в это время истекает квант времени, отпущенный данному потоку, или он вытесняется более приоритетным потоком, использующим те же данные. Новый поток выполняется, пока не встречает функцию EnterCriticalSection, которая помнит, что объект «cs» уже занят. Новый поток приостанавливается, а остаток процессорного времени передается другому потоку. Функция LeaveCriticalSection уменьшает счетчик ссылок на объект «cs». Как только поток покидает критическую секцию, счетчик ссылок обнуляется и система будит ожидающий поток, снимая защиту секции кодов. Критические секции применяются для синхронизации потоков лишь в пределах одного процесса. Они управляют доступом к данным так, что в каждый конкретный момент времени только один поток может их изменять.

4. Когда надобность в синхронизации потоков отпадает, следует вызвать функцию, освобождающую все ресурсы, включенные в критическую секцию:

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

Примечание. Функция TryEnterCriticalSection позволяет проверить критическую секцию на занятость:

– если критическая секция свободна, поток занимает ее;

– если же нет, поток блокируется до тех пор, пока секция не будет освобождена другим потоком с помощью вызова функции LeaveCriticalSection.

Данные функции — атомарные, то есть целостность данных нарушена не будет.

2) Работа с объектом «Семафор» (semaphore). Семафор представляет собой счетчик, содержащий целое число в диапазоне от 0 до максимальной величины, заданной при его создании. Для работы с объектом Semaphore существует ряд функций:

– функция CreateSemaphore создает семафор с заданным начальным значением счетчика и максимальным значением, которое ограничивает доступ;

– функция OpenSemaphore осуществляет доступ к семафору;

– функция ReleaseSemaphore увеличивает значение счетчика. Счетчик может меняться от 0 до максимального значения;

– после завершения работы необходимо удалить семафор вызовом функции CloseHandle.

3) Работа с объектом «Мьютекс» (mutex). Для работы с этим объектом предусмотрен ряд функций:

– функция создания объекта — CreateMutex;

– функция доступа — OpenMutex;

– для освобождения ресурса — ReleaseMutex;

– для доступа к объекту Mutex используется ожидающая функция WaitForSingleObject;

– после завершения работы необходимо удалить мьютекс вызовом функции CloseHandle.

Каждая программа создает объект Mutex по имени, то есть Mutex — это именованный объект. Если такой объект синхронизации уже создала другая программа, то по вызову CreateMutex можно получить указатель на объект, который уже создала первая программа, то есть у обеих программ будет один и тот же объект, что и позволяет производить синхронизацию. Если имя не задано, мьютекс будет неименованным, и им можно пользоваться только в пределах одного процесса.

4) Работа с объектом «Событие» (event). Для работы с событиями предусмотрены следующие функции:

– функция CreateEvent используется для создания события;

– функция OpenEvent — для доступа к событию;

– две функции SetEvent и PulseEvent — для установки события;

– функция ResetEvent — для сброса события.

– после завершения работы необходимо удалить событие вызовом функции CloseHandle.

Событие является синхронизирующим объектом ядра. Оно позволяет одному потоку уведомить (notify) другой поток о том, что произошло событие, которое тот поток, возможно, ждал. Существует два типа событий: ручные (manual) и автоматические (automatic):

– Ручной объект начинает сигнализировать, когда будет вызван метод SetEvent. Вызов ResetEvent переводит его в противоположное состояние.

– Автоматический объект не нуждается в сбросе. Он сам переходит в состояние nonsignaled, и охраняемый код при этом недоступен, когда хотя бы один поток был уведомлен о наступлении события.

3.2.3.3 Функции ожидания

Win32 API поддерживает целый ряд функций, которые начинаются с префикса Wait или MsgWait:

– WaitForMultipleObjects;
– WaitForMultipleObjectsEx;
– WaitForSingleObject;
– WaitForSingleObjectEx;
– MsgWaitForMultipleObjects;
– WaitForMultipleObjectsEx.

Также существует функция WaitCommEvent, предназначенная для работы с данными в последовательных портах, и функция SignalObjectAndWait для передачи сигнала между объектами с последующим ожиданием.

Функции, у которых в имени есть Single, предназначены для установки одного синхронизирующего объекта. Функции, у которых в имени есть Multiple, используются для установки ожидания сразу нескольким объектам. Функции с префиксами Msg предназначены для ожидания события определенного типа, например ввода с клавиатуры. Функции с окончанием Ex расширены опциями по управлению асинхронным вводом-выводом.

Примечание. При необходимости захвата нескольких ресурсов используется функция WaitForMultipleObject, так как эта функция, ожидая несколько объектов, пока не захватит их все, менять состояние одного из них не будет. Функцию WaitForSingleObject в этом случае использовать нельзя, так как это приведет к deadlock.

В простейшем случае потоки «усыпляют» себя до освобождения какого-либо синхронизирующего объекта с помощью следующих функций:

```
      DWORD WaitForSingleObject(HANDLE hObject, DWORD dwTimeOut);
      DWORD WaitForMultipleObjects(DWORD nCount, CONST HANDLE *lpHandles,
BOOL bWaitAll, DWORD dwTimeOut);
```

Первая функция приостанавливает поток до тех пор, пока:

– заданный параметром hObject синхронизирующий объект не освободится;

– не истечет интервал времени, задаваемый параметром dwTimeOut. Если указанный объект в течение заданного интервала не перейдет в свободное состояние, то система вновь активизирует поток и он продолжит свое выполнение. В качестве параметра dwTimeOut могут выступать два особых значения: 0 — функция только проверяет состояние объекта (занят или свободен) и сразу же возвращается, а также INFINITE — время ожидания бесконечно (если объект так и не освободится, поток останется в неактивном состоянии и никогда не получит процессорного времени).

Функция WaitForSingleObject, в соответствии с причинами, по которым поток продолжает выполнение, может возвращать одно из следующих значений:

– WAIT_TIMEOUT — объект не перешел в свободное состояние, но интервал времени истек;

– WAIT_ABANDONED — ожидаемый объект является Mutex, который не был освобожден владеющим им потоком перед окончанием этого потока. Объект Mutex автоматически переводится системой в состояние «свободен». Такая ситуация называется «отказ от Mutex»;

– WAIT_OBJECT_0 — объект перешел в свободное состояние;

– WAIT_FAILED — произошла ошибка, причину которой можно узнать, вызвав GetLastError.

Функция WaitForMultipleObjects задерживает поток и, в зависимости от значения флага bWaitAll, ждет одного из следующих событий:

– освобождение хотя бы одного синхронизирующего объекта из заданного списка;

– освобождение всех указанных объектов;

– истечение заданного интервала времени.

3.2.3.4 Приоритеты в ОС Windows

1) Приоритеты процессов. Часть ОС, называемая системным планировщиком (system scheduler), управляет переключением заданий, определяя, какому из конкурирующих потоков следует выделить следующий квант времени процессора. Решение принимается с учетом приоритетов конкурирующих потоков. Множество приоритетов, определенных в операционной системе для потоков, занимает диапазон от 0 (низший приоритет) до 31 (высший приоритет). Нулевой уровень приоритета система присваивает особому потоку обнуления свободных страниц. Он работает при отсутствии других потоков, требующих внимания со стороны операционной системы. Ни один поток, кроме него, не может иметь нулевой уровень.

Приоритет каждого потока определяется в два этапа, исходя из:

1) класса приоритета процесса, в контексте которого выполняется поток;

2) уровня приоритета потока внутри класса приоритета потока.

Комбинация этих параметров определяет базовый приоритет (base priority) потока. Существуют шесть классов приоритетов для процессов:

– IDLE_PRIORITY_CLASS;
– BELOW_NORMAL_PRIORITY_CLASS;
– NORMAL__PRIORITY_CLASS;
– ABOVE_NORMAL_PRIORITY_CLASS;
– HIGH_PRIORITY_CLASS;
– REALTIME_PRIORITY_CLASS.

Работа с приоритетами процесса:

1) по умолчанию процесс получает класс приоритета NORMAL;

2) программист может задать класс приоритета создаваемому им процессу, указав его в качестве одного из параметров функции CreateProcess;

3) кроме того, существует возможность динамически, во время выполнения потока, изменять класс приоритета процесса с помощью функции SetPriorityClass;

4) выяснить класс приоритета какого-либо процесса можно с помощью функции GetPriorityClass.

Процессы, осуществляющие мониторинг системы, а также хранители экрана (screen savers) должны иметь низший класс (IDLE), чтобы не мешать другим полезным потокам. Процессы самого высокого класса (REALTIME) способны прервать даже те системные потоки, которые обрабатывают сообщения мыши, ввод с клавиатуры и фоновую работу с диском. Этот класс должны иметь только те процессы, которые выполняют короткие обменные операции с аппаратурой. Для написания драйвера какого-либо устройства, используя API-функции из набора Device Driver Kit (DDK), следует использовать для процесса класс REALTIME. С осторожностью следует использовать класс HIGH, так как если поток процесса этого класса подолгу занимает процессор, то другие потоки не имеют шанса получить свой квант времени. Если несколько потоков имеют высокий приоритет, то эффективность работы каждого из них, а также всей системы резко падает. Этот класс зарезервирован для реакций на события, критичные ко времени их обработки.

2) Приоритеты потоков. Теперь рассмотрим уровни приоритета, которые могут быть присвоены потокам процесса. Внутри каждого процесса, которому присвоен какой-либо класс приоритета, могут существовать потоки, где уровень приоритета принимает одно из семи возможных значений:

– THREAD_PRIORITY_IDLE;
– THREAD_PRIORITY_LOWEST;
– THREAD_PRIORITY_BELOW_NORMAL;
– THREAD_PRIORITY_NORMAL;
– THREAD_PRIORITY_ABOVE_NORMAL;
– THREAD_PRIORITY_HIGHEST;
– THREAD_PRIORITY_TIME_CRITICAL.

Работа с приоритетами потока следующая:

1) все потоки сначала создаются с уровнем NORMAL;

2) программист может изменить этот начальный уровень, вызвав функцию SetThreadPriority;

3) для определения текущего уровня приоритета потока существует функция GetThreadPriority, которая возвращает один из семи рассмотренных уровней.

Типичной стратегией является повышение уровня до ABOVE_NORMAL или HIGHEST для потоков, которые должны быстро реагировать на действия пользователя по вводу информации. Потоки, которые интенсивно используют процессор для вычислений, часто относят к фоновым потокам. Им дают уровень приоритета BELOW_NORMAL или LOWEST, чтобы при необходимости они могли быть вытеснены.

Иногда возникает ситуация, когда поток с более высоким приоритетом должен ждать поток с низким приоритетом, пока тот не закончит какую-либо операцию. В этом случае не следует программировать ожидание завершения операции в виде цикла, так как львиная доля времени процессора уйдет на выполнение команд этого цикла. Возможно даже зацикливание — ситуация, когда поток не имеет шанса получить управление и завершить операцию. Обычной практикой в таких случаях является использование:

– одной из функций ожидания;
– вызов функции Sleep (SleepEx);
– вызов функции SwitchToThread;
– использование объекта типа «Критическая секция».

Базовый приоритет потока является комбинацией класса приоритета процесса и уровня приоритета потока (табл. 3.2).

3) Переключение потоков. Планировщик операционной системы поддерживает для каждого из базовых уровней приоритета функционирование очереди выполняемых или готовых к выполнению потоков (ready threads queue). Когда процессор становится доступным, то планировщик производит переключение контекстов. Алгоритм переключения следующий:

– сохранение контекста потока, завершающего выполнение;
– перемещение этого потока в конец своей очереди;
– поиск очереди с высшим приоритетом, которая содержит потоки, готовые к выполнению;
– выбор первого потока из этой очереди, загрузка его контекста и запуск на выполнение.

Примечание. Если в системе за каждым процессором закреплен хотя бы один поток с приоритетом 31, то остальные потоки с более низким приоритетом не смогут получить доступ к про-

цессору и поэтому не будут выполняться. Такая ситуация называется *starvation*.

Различают потоки, неготовые к выполнению:

– потоки, которые при создании имели флаг CREATE_SUSPENDED;

– потоки, выполнение которых было прервано вызовом функции SuspendThread или SwitchToThread;

– потоки, которые ожидают ввода или синхронизирующего события.

Блокированные или приостановленные потоки не получают кванта времени независимо от величины их приоритета.

Типичные причины переключения контекстов:

– истек квант времени;

– в очереди с более высоким приоритетом появился поток, готовый к выполнению;

– текущий поток вынужден ждать.

В последнем случае система не ждет завершения кванта времени и отбирает управление, как только поток впадает в ожидание. Каждый поток обладает динамическим приоритетом, кроме рассмотренного базового уровня. Под этим понятием скрываются временные колебания уровня приоритета, которые вызваны планировщиком. Он намеренно вызывает такие колебания для того, чтобы убедиться в управляемости и реактивности потока, а также, чтобы дать шанс выполниться потокам с низким приоритетом. Система никогда не подстегивает потоки, приоритет которых и так высок (от 16 до 31).

Иногда система инвертирует приоритеты, чтобы разрешить конфликты типа deadlock. Благодаря динамике изменения приоритетов потоки активного процесса вытесняют потоки фонового процесса, а потоки с низким приоритетом все-таки имеют шанс получить управление.

Программист имеет возможность управлять процессом ускорения потоков с помощью функций SetProcessPriorityBoost (все потоки данного процесса) или SetThreadPriorityBoost (только данный поток). Функции GetProcessPriorityBoost и GetThreadPriorityBoost позволяют выяснить текущее состояние флага.

При наличии нескольких процессоров или процессорных ядер (реальных или виртуальных, например, при использовании

технологии Intel HyperThreading) ОС Windows применяет симметричную модель распределения потоков по процессорам symmetric multiprocessing (SMP). Это означает, что любой поток может быть направлен на любой процессор. Программист может ввести некоторые коррективы в эту модель равноправного распределения. Функции SetProcessAffinityMask и SetThreadAffinityMask позволяют указать предпочтения в смысле выбора процессора для всех потоков процесса или для одного определенного потока. Потоковое предпочтение (thread affinity) вынуждает систему выбирать процессоры только из множества, указанного в маске.

Существует также возможность для каждого потока указать один предпочтительный процессор. Это делается с помощью вызова функции SetThreadIdealProcessor.

3.2.3.5 Использование портов завершения ввода/вывода

В Windows 2000 предусмотрен объект ядра, реализующий функции пула потоков, но в NT4 этот примитив не доступен. Поэтому рассмотрим возможность организации пула с помощью объекта ядра — порта завершения ввода/вывода (ПЗВВ).

Создание порта завершения происходит вызовом функции CreateIoCompletionPort:

```
HANDLE CreateIoCompletionPort (
  HANDLE FileHandle,
  HANDLE ExistingCompletionPort,
  ULONG_PTR CompletionKey,
  DWORD NumberOfConcurrentThreads
);
```

Здесь первый параметр — дескриптор файла, второй — дескриптор существующего ПЗВВ, третий — определенное пользователем значение, уникальный ключ порта, четвертый — максимальное количество конкурирующих потоков.

Создание ПЗВВ происходит в два этапа. На первом этапе функция вызывается со следующими параметрами:

```
HANDLE hcp = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, N);
```

Здесь N — количество потоков в пуле (если задать 0, то принимается равным количеству процессоров в системе).

Далее необходимо создать файл для связи порта с устройством. Необходимо помнить, что для ОС Windows любое внешнее

устройство является файлом (это типично для систем, построенных согласно архитектуре фон Неймана). Файл создается с помощью вызова функции CreateFile (см. MSDN).

При следующем вызове функции CreateIoCompletionPort первым параметром ей передается дескриптор созданного файла, а вторым — дескриптор hcp, полученный в результате предыдущего вызова этой функции, третьим — значение, которое будет передаваться потокам пула при регистрации им завершения операции ввода/вывода. В четвертом параметре передается то же самое значение, что и при предыдущем вызове, то есть максимальное число потоков в пуле.

После создания пула необходимо создать потоки, которые будут находиться в пуле, и ожидать окончания операции ввода/вывода. Это делается вызовом функции CreateThread. При создании потоков следует запоминать их дескрипторы, т.к. после удаления пула потоков потребуется принудительное завершение всех созданных потоков вызовом TerminateThread. Все созданные потоки должны при работе вызывать одну и ту же функцию.

Каждый из созданных потоков должен впадать в бесконечный цикл ожидания и обработки запросов. Для получения нотификации о завершении операции, связанной с устройством, поток должен вызвать функцию

```
BOOL GetQueuedCompletionStatus(
  HANDLE CompletionPort,
  LPDWORD lpNumberOfBytes,
  PULONG_PTR lpCompletionKey,
  LPOVERLAPPED *lpOverlapped,
  DWORD dwMilliseconds
);
```

Здесь первый параметр — дескриптор ПЗВВ, второй — указатель на переменную, куда будет записано количество байтов, переданных в результате операции ввода/вывода, третий — уникальный ключ порта, четвертый — указатель на структуру типа OVERLAPPED, пятый — время в миллисекундах, которое наш поток может находиться в состоянии ожидания завершения операции. Если задать константу INFINITE, то поток будет ожидать бесконечное количество времени.

Структура OVERLAPPED используется при асинхронном вводе/выводе, ее экземпляр необходимо создать до вызова функ-

ции GetQueuedCompletionStatus, и передать указатель на нее. В результате вызова она будет заполнена информацией об операции ввода/вывода (см. MSDN).

В результате вызова этой функции поток впадет в спячку до тех пор, пока другой поток не инициирует завершение операции ввода/вывода и порт завершения не передаст управление этому потоку.

Для эмуляции завершения операции ввода/вывода, то есть именно той операции, которая повлечет за собой пробуждение одного из потоков в пуле, необходимо вызвать функцию PostQueuedCompletionStatus:

```
BOOL PostQueuedCompletionStatus(
  HANDLE CompletionPort,
  DWORD dwNumberOfBytesTransferred,
  ULONG_PTR dwCompletionKey,
  LPOVERLAPPED lpOverlapped
);
```

Здесь первый параметр — дескриптор ПЗВВ, второй — количество передаваемых байтов (должно быть больше 0), третий — уникальный ключ порта, четвертый — указатель на структуру типа OVERLAPPED (если она не нужна, можно передать NULL). Таким способом потоку в пуле передается задание на обработку асинхронной операции ввода/вывода.

3.3 Использование сетей Петри

В данном разделе рассматривается моделирование взаимодействия параллельных процессов с использованием сетей Петри.

3.3.1 Задача взаимного исключения

Задача взаимного исключения является одной из ключевых проблем параллельного программирования. Было предложено много способов решения этой проблемы. Взаимоисключение одновременного доступа процессов к данным необходимо применить при работе нескольких параллельных процессов с общими данными. При этом участки программ процессов для работы с разделяемыми данными образуют так называемые критические области (секции). Операционная система может в любой момент

вытеснить поток *P* и подключить к процессору другой, но ни один из потоков, которым нужен занятый потоком *P* ресурс, не получит процессорное время до тех пор, пока поток *P* не выйдет за границы критической секции.

3.3.2 Задача «производитель-потребитель»

Имеется большое число вариантов постановки и решения такой задачи в рамках конкретных операционных систем. В общем случае взаимодействуют несколько процессов с жестко распределенными между ними функциями. Одни процессы вырабатывают сообщения, предназначенные для восприятия и обработки другими процессами.

Процесс, вырабатывающий сообщения, называют *производителем*, а воспринимающий сообщения — *потребителем*.

Процессы взаимодействуют через некоторую обобщенную область памяти, которая по смыслу является критическим ресурсом. В эту область процесс-производитель должен поместить очередное сообщение, а процесс-потребитель должен считывать очередное сообщение.

Общий вид постановки задачи. Необходимо согласовать работу двух процессов при одностороннем (в простейшем случае) обмене сообщениями по мере развития процессов таким образом, чтобы удовлетворить следующим требованиям:

1. Выполнять требования задачи взаимного исключения по отношению к критическому ресурсу — обобщенной памяти для хранения сообщения.

2. Учитывать состояние обобщенной области памяти, характеризующей возможность или невозможность посылки (принятия) очередного сообщения.

3. Попытка процесса-производителя поместить очередное сообщение в область, из которой не было считано предыдущее сообщение процессом-потребителем, должна быть блокирована. Процесс-производитель должен быть либо оповещен о невозможности помещения сообщения, либо переведен в состояние ожидания возможности поместить очередное сообщение через некоторое время в область памяти, по мере ее освобождения.

4. Попытка процесса-потребителя считать сообщение из области в ситуации, когда процесс-производитель не поместил туда очередного сообщения, должна быть заблокирована. Процесс-потребитель должен быть либо оповещен о невозможности считывания сообщения, либо переведен в состояние ожидания поступления очередного сообщения.

5. Если используется вариант с ожиданием изменения состояния обобщенной области для хранения сообщений, необходимо обеспечить перевод ожидающих процессов в состояние готовности всякий раз, когда изменится состояние области. Либо процесс-производитель поместит очередное сообщение в область, и оно теперь может быть считано ожидающим процессом-потребителем, либо процесс-потребитель считал очередное сообщение из области и обеспечил возможность ожидающему процессу-потребителю послать очередное сообщение.

Примечание. Множественность постановки задачи «производитель-потребитель» определяется следующим:

– процессов-потребителей и процессов-производителей может быть больше одного;

– каждый из таких процессов может устанавливать не только одностороннюю, но и двустороннюю связь через одну и ту же обобщенную область или другие области;

– области могут хранить не только одно, а большое количество сообщений.

3.3.3 Задача «читатели-писатели»

Существует несколько вариантов этой задачи, однако их основная структура остается неизменной. Имеется система параллельных процессов, которые взаимодействуют друг с другом следующим образом:

1. Все процессы делятся на два типа: процессы-читатели и процессы-писатели. Процессы работают с общей областью памяти.

2. Процессы-читатели считывают, а процессы-писатели записывают информацию в общую область памяти.

3. Одновременно может быть несколько активных процессов-читателей.

4. При записи информации область памяти рассматривается как критический ресурс для всех процессов, то есть если работает процесс-писатель, то он должен быть единственным активным процессом.

Задача состоит в определении структуры управления, которая не приведет к тупику и не допустит нарушения критерия взаимного исключения.

Наиболее характерная область использования этой задачи — построение файловых систем.

В отношении некоторой области памяти, являющейся по смыслу критическим ресурсом для параллельных процессов, работающих с ней, выделяется два типа процессов:

– процессы-читатели, которые считывают одновременно информацию из области, если это допускается при работе с конкретным устройством памяти;

– процессы-писатели, которые записывают информацию в область и могут делать это, только исключая как друг друга, и процессы-читатели, то есть запись должна удовлетворяться на основании решения задачи взаимного исключения.

Имеются различные варианты взаимодействия между процессами-писателями и процессами-читателями. Наиболее широко распространены следующие:

1. Устанавливается высшая приоритетность в использовании критического ресурса процессам-читателям (если хотя бы один процесс-читатель пользуется ресурсом, то он закрыт для использования всеми процессами-писателями).

2. Наивысший приоритет у процессов-писателей (при появлении запроса от процесса-писателя необходимо закрыть ресурс для использования процессами-читателями).

3.3.4 Задача об обедающих философах

Название и формулировка этой задачи носят несколько абстрактный характер, но такая задача синхронизации также имеет место при построении систем распределения ресурсов в составе операционной системы. В рамках этой задачи формулируются требования на синхронизацию работы процессов, которые совместно используют пересекающиеся группы ресурсов.

Рассмотрим формулировку задачи «Обедающие философы» в терминологии, предложенной Э. Дейкстрой (Edsser Wybe Dijkstra).

За круглым столом расставлены пять стульев, на каждом из которых сидит определенный философ. В центре стола — большое блюдо спагетти, а на столе лежат пять вилок — каждая между двумя соседними тарелками. Любой философ может находиться только в двух состояниях — либо он размышляет, либо ест спагетти. Начать думать философу ничто не мешает. Но чтобы начать есть, философу нужны две вилки: одна в правой руке, другая в левой руке. Закончив еду, философ кладет вилки слева и справа от своей тарелки и опять начинает размышлять до тех пор, пока снова не проголодается.

Существует множество различных формулировок данной задачи, в одной из которых философы интерпретируются как процессы, а вилки — как ресурсы.

В представленной задаче имеются две опасные ситуации: ситуация голодной смерти и ситуации голодания отдельного философа.

Ситуация голодной смерти возникает в случае, когда философы одновременно проголодаются и одновременно попытаются взять, например, свою левую вилку. В данном случае возникает тупиковая ситуация, так как никто из них не может начать есть, не имея второй вилки. Для исключения ситуации голодной смерти были предложены различные стратегии распределения вилок.

Ситуация голодания возникает в случае заговора двух соседей слева и справа против философа, в отношении которого строятся козни. Заговорщики поочередно забирают вилки то слева, то справа от него. Такие согласованные действия злоумышленников приводят жертву к вынужденному голоданию, так как он никогда не может воспользоваться обеими вилками.

В рассмотренной системе тупиков можно избежать, используя ограничение на пользование вилками, согласно которому философ может взять две вилки, необходимые для еды, только в том случае, если обе они свободны (предполагается, что каждый из философов берет две необходимые для еды вилки одновременно). Фаза ожидания каждого философа должна быть конечной (следовательно, система будет свободна от ситуации голодания).

Другим способом избегания тупиков является введение еще одной задачи, гарантирующей, что в каждый данный момент времени за столом находится не более четырех философов. Каждый философ, прежде чем сесть за стол, должен получить разрешение задачи, а покидая стол, уведомить задачу об этом. Тупиков быть не может, так как за столом будет, по крайней мере, один философ, который может приступить к еде. После еды, занимающей конечное время, он уйдет, и другой философ сможет начать прием пищи.

СПИСОК ЛИТЕРАТУРЫ

1. Калайда В.Т. Теория вычислительных процессов: учеб. пособие / В. Т. Калайда. — Томск : Изд-во ТМЦДО, 2013.

2. Рихтер Дж. Windows для профессионалов / Дж. Рихтер. — СПб. : Питер, 2000. — 752 с.

3. Рихтер Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C# / Дж. Рихтер. — СПб. : Питер, 2007. — 656 с.

4. Троелсен Э. C# и платформа .NET 3.0 / Э. Троелсен. — СПб. : Питер, 2008. — 1456 с.

5. Вебер Дж. Технология Win32 в примерах / Дж. Вебер. — СПб. : БХВ-Петербург, 1998. — 1070 с.

6. Воеводин В.В. Параллельные вычисления. — СПб.: БХВ-Петербург, 2002. — 608 с.

7. Таненбаум Э. Современные операционные системы / Э. Таненбаум. — 3-е изд. — СПб. : Питер, 2010. — 1120 с.

8. Харт Дж. Системное программирование в среде Win32 / Дж. Харт. — 3-е изд. — М. : Вильямс, 2005. — 586 с.

ПРИЛОЖЕНИЕ А
СТРУКТУРА ОТЧЕТА

Федеральное агентство по образованию

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Факультет дистанционного образования (ФДО)

Кафедра автоматизированных систем управления (АСУ)

НАЗВАНИЕ РАБОТЫ

Отчет по лабораторной работе № X по дисциплине

«Теория вычислительных процессов»

Выполнил: _____

_____

«____» _____ 20__ г.

Проверил: _____

_____

«____» _____ 20__ г.

Томск — 20__

Содержание