Электронное учебное издание

Мандрик А.В.
Ухов В.И.

Выполнение практических работ по дисциплине
«Параллельные и распределенные вычисления»

ЭЛЕКТРОННОЕ МЕТОДИЧЕСКОЕ ПОСОБИЕ

Филиал «Протвино»
государственного университета «Дубна
142281 г. Протвино Московской обл.,

Северный проезд, 9

**Государственное бюджетное образовательное учреждение
высшего образования Московской области
«Университет «Дубна»
Филиал «Протвино»
Кафедра «Информационные технологии»**

А.В. Мандрик, В.И. Ухов

Выполнение практических работ по дисциплине «Параллельные и распределенные вычисления»

Электронное методическое пособие

Рекомендовано
кафедрой информационных технологий
филиала «Протвино» государственного университета
«Дубна»
в качестве методического пособия для студентов,
обучающихся по направлению
«Информатика и вычислительная техника»

Протвино, 2017

ББК 32.973.26-02
М 23

Рецензент:
кандидат физико-математических наук,
главный специалист ООО «Систел»
В.А. Мухин

Мандрик, А.В.
М23 Выполнение практических работ по дисциплине «Параллельные и распределенные вычисления»: электронное методическое пособие / А.В. Мандрик, В.И. Ухов. — Протвино: 2017. — 71с.

Предназначено для студентов очного и заочного отделений направления «Информатика и вычислительная техника».

Методическое пособие устанавливает требования к уровню практического освоения студентами основных методов и технологий параллельных и распределенных вычислений. Рассматриваются аспекты параллельного программирования на платформах операционных систем Windows и Linux, а также методика программирования сетевых приложений на базе платформы Windows.

ББК 32.973.26-02

Оглавление

Библиографический список

1. Таненбаум, Э. Распределенные системы. Принципы и парадигмы / Э. Таненбаум, М. Танненбаум. — СПб. : Питер, 2003. — 877 с.

2. Рихтер, Д. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows / Д. Рихтер. — СПб. : Питер, 2001. — 752 с.

3. Эндрюс, Г.Р. Основы многопоточного, параллельного и распределенного программирования / Г.Р. Эндрюс. — М. : «Вильямс», 2003. — 512 с.

4. Гергель, В.П. Теория и практика параллельных вычислений / В.П. Гер-гель. — М. : ИНТУИР.РУ Интернет-Университет Информационных Техноло-гий, 2007.

5. Богачев, К.Ю. Основы параллельного программирования / К.Ю. Богачев. — М. : БИНОМ. Лаборатория знаний, 2003.

6. Воеводин, В.В. Параллельные вычисления / В.В Воеводин., Вл.В. Воеводин. — СПб. : БХВ-Петербург, 2002.

7. Хованский, Е.П. «Лабораторные работы по курсу Параллельные и распределенные вычисления» / Е.П. Хованский. — http://ps.margtu.ru/wiki/index.php?wakka=HomePage/20092010/4kurs/RV&v=14p0.

9. Stevens, W.R. Advanced Programming in the UNIX® Environment: Second Edition / W. R. Stevens, S.A. Rago — Addison Wesley Professional, 2005.

10. Bovet, D.P. Understanding the Linux Kernel, 3rd Edition / D. P. Bovet, M. Cesati. — O'Reilly, 2005.

11. Боровский, А. Потоки / А. Боровский. — http://www.citforum.ru/programming/unix/threads.

Общие положения

Методические указания предназначены для изучения и практического освоения студентами основных методов и технологий параллель-ных и распределенных вычислений.

Возрастающий уровень использования данных технологий в современной практической деятельности можно объяснить следующим:

– быстрый рост сложности моделируемых объектов (переход от простых систем к сложным системам);

– решение задач, требующих исследований. Например, для проведения тщательного анализа сложного поведения (исследование усло-вий перехода к так называемому детерминированному хаосу);

– необходимость управления сложными промышленными и технологическими программно-аппаратными комплексами в режиме реального времени и в условиях неопределенности;

– рост числа задач, для решения которых необходимо обрабатывать гигантские объемы информации.

Очевидно, что «простой перенос» идеологии программирования, созданной для последовательных задач небольшой размерности, не мо-жет гарантировать резкого повышения эффективности вычислитель-ных экспериментов только за счет использования высокопроизводи-тельных вычислительных систем. Таким образом, обретение навыков создания параллельных и/или распределенных программных приложе-ний является первым шагом к адекватному использованию современ-ных вычислительных комплексов и средств.

Цель и задачи курса практических работ

Целью изучения дисциплины является подготовка специалистов в области программного обеспечения вычислительной техники и авто-матизированных систем.

Задачей практического курса является приобретение навыков раз-работки программного обеспечения с применением методов парал-лельной обработки данных и разнесения функциональности ПО между различными ПК в сети.

Курс состоит из теоретической и практической частей. В теорети-ческой части даются основные определения, рассматриваются базовые подходы к решению типовых задач.

Практическая часть состоит из работ трех взаимосвязанных направлений: «Реализация многопоточности», «Синхронизация доступа к данным», «Программирование сокетов». В рамках курса реализация задач осуществляется на платформах ОС Windows и Linux.

Порядок выполнения практических работ

В соответствии с графиком студенты перед выполнением практического задания обязаны ознакомиться с методическими указаниями по ее выполнению и рекомендованной литературой. Во время занятий каждый студент получает индивидуальный вариант зада-ния.

Программа курса рассчитана на 15 занятий (30 акад.часа). Отдель-ное практическое занятие из курса исполняется в пределах от 2 до 4 аудиторных занятий.

Для получения зачета по каждой работе студент предоставляет преподавателю реализации предложенных в работе примеров и задания по вариантам.

Во время собеседования студент обязан проявить знания по цели работы, теоретическому материалу, методам выполнения каждого эта-па работы.

Студент обязан уметь правильно анализировать полученные результаты и представлять области применения освоенных в рамках работы технологий и методик.

Порядок защиты практических работ

Защита выполненных практических заданий осуществляется в устной форме. Для успешной защиты практической работы студенту требуется:

– предоставить преподавателю код приложений, указанных в практическом занятии (включая примеры);


```
return EXIT_FAILURE;
}
result = pthread_join(thread2, NULL);
if (result != 0) {
perror(«Joining the fi rst thread»);
return EXIT_FAILURE;
}
sem_destroy(&sem);
printf(«Done\n»);
return EXIT_SUCCESS;
}
```

Присвоив семафору значение 0, программа создает первый поток и вызывает функцию *sem_wait()*. Эта функция приостановит выполнение функции *main()* до тех пор, пока функция потока не вызовет функ-цию *sem_post()*, а это случится только после того как функция потока обработает значение переменной *id*. Таким образом, мы можем быть уверены, что в момент создания второго потока первый поток уже за-кончит работу с переменной *id*, и мы сможем использовать эту пере-менную для передачи данных второму потоку.

Варианты заданий

Перевести приложение, разработанное в теме № 2 на платформу ОС *Linux*. Для задачи синхронизации примените семафоры или мьютексы (по вашему выбору).

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
#include <semaphore.h>
sem_t sem;
void * thread_func(void * arg)
{
int i;
int loc_id = * (int *) arg;
sem_post(&sem);
for (i = 0; i < 4; i++) {
printf(«Thread %i is running\n», loc_id);
sleep(1);
}
}
int main(int argc, char * argv[])
{
 nt id, result;
pthread_t thread1, thread2;
id = 1;
sem_init(&sem, 0, 0);
result = pthread_create(&thread1, NULL, thread_func, &id);
if (result != 0) {
perror(«Creating the fi rst thread»);
return EXIT_FAILURE;
}
sem_wait(&sem);
id = 2;
result = pthread_create(&thread2, NULL, thread_func, &id);
if (result != 0) {
perror(«Creating the fi rst thread»);
return EXIT_FAILURE;
}
result = pthread_join(thread1, NULL);
if (result != 0) {
perror(«Joining the fi rst thread»);
```


– продемонстрировать работоспособность указанных приложений;

– ответить на 3 вопроса преподавателя по теме практического занятия. Вопросы касаются алгоритма исполнения фрагментов представляемого студентом кода.

Практическое занятие № 1. Создание многопоточных приложений в ОС Windows

Цель работы: научиться создавать простые многопоточные приложения на базе операционной системы Windows.

Порядок выполнения практических заданий

1. Рассмотреть представленные примеры, и разработать приложе-ния на их основе.

2. Разработать алгоритм решения третьего задания, с учетом разделения вычислений между несколькими потоками. Избегать ситуаций изменения одних и тех же общих данных несколькими потоками.

3. Реализовать алгоритм с применением функций WinAPI и протестировать его на нескольких примерах.

Длительность — 6 акад. часов.

Литературные источники

1. Рихтер, Дж. Windows для профессионалов : создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер. — СПб. : Питер, 2001. — 752 с.

2. Эндрюс, Г.Р. Основы многопоточного, параллельного и распределенного программирования / Г.Р. Эндрюс. — М. : «Вильямс», 2003. — 512 с.

3. Хованский, Е.П. Лабораторные работы по курсу «Параллельные и распределенные вычисления» / Е.П. Хованский.

Теоретическая часть

Во многих задачах можно выделить ряд подзадач, каждую из которых возможно решить или независимо от других подзадач, или с их минимальной кооперацией. При этом подзадачи выполняются конкурентно (в однопроцессорной системе) или параллельно в многопроцессорной системе. В многопоточной модели каждая такая подзадача существует

текса. Вторым аргументом функции *pthread_mutex_init()* должен быть указатель на переменную типа *pthread_mutexattr_t*. Эта переменная позволяет установить дополнительные атрибуты мьютекса. Если нам нужен обычный мьютекс, мы можем передать во втором параметре значение *NULL*. Для того чтобы получить исключительный доступ к некоему глобальному ресурсу, поток вызывает функцию *pthread_mutex_lock(3)*, (в этом случае говорят, что «поток захватывает мьютекс»). Единственным параметром функции *pthread_mutex_lock()* должен быть идентификатор мьютекса. Закончив работу с глобальным ресурсом, поток высвобождает мьютекс с помощью функции *pthread_mutex_unlock(3)*, которой также передается идентификатор мьютекса. Если поток вызовет функцию *pthread_mutex_lock()* для мьютекса, уже захваченного другим потоком, эта функция не вернет управление до тех пор, пока другой поток не высвободит мьютекс с помощью вызова *pthread_mutex_unlock()* (после этого мьютекс, естественно, перейдет во владение нового потока). Удаление мьютекса выполняется с помощью функции *pthread_mutex_destroy(3)*. Стоит отметить, что в отличие от многих других функций, приостанавливающих работу потока, вызов *pthread_mutex_lock()* не является точкой останова. Иначе говоря, поток, находящийся в режиме отложенного досрочного завершения, не может быть завершен в тот момент, когда он ожидает выхода из *pthread_mutex_lock()*.

Практическая часть

Пример 1

Вернемся к примеру 1 из предыдущего практического занятия. Напомню, что в том примере мы создавали два потока, используя одну и ту же функцию потока. В процессе создания каждого потока этой функции передавалось целочисленное значение (номер потока). При этом для передачи значения каждому потоку использовалась своя пере-менная. В тот момент объяснялось почему мы не можем использовать одну переменную для передачи значения. Проблема заключалась в том, что мы не могли знать, когда именно новый поток начнет выполнять-ся. С помощью средств синхронизации потоков мы можем решить эту проблему и использовать одну переменную для передачи значений обоим потокам.

P(S): ***пока S == 0 процесс блокируется;***
S = S – 1;
P(S): ***S = S + 1.***

Эта запись означает следующее: при выполнении операции ***P*** над семафором ***S*** сначала проверяется его значение. Если оно больше **0**, то из ***S*** вычитается 1. Если оно меньше или равно **0**, то процесс блокируется до тех пор, пока ***S*** не станет больше **0**, после чего из ***S*** вычитается **1**. При выполнении операции ***V*** над семафором ***S*** к его значению просто прибавляется 1.

В примере мы рассмотрим семафоры *POSIX*, которые специально предназначены для работы с потоками. Все объявления функций и типов, относящиеся к этим семафорам, можно найти в файле */usr/include/nptl/semaphore.h.* Семафоры *POSIX* создаются (инициализируются) с помощью функции *sem_init(3)*. Первый параметр функции *sem_init()* — указатель на переменную типа *sem_t*, которая служит идентификатором семафора. Второй параметр — pshared — указывает что данный семафор будет разграничивать потоки внутри одного процесса или процессы. В наших примерах не используется, и мы оставим его равным нулю. В третьем параметре функции *sem_init()* передается значение, которым инициализируется семафор. Дальнейшая работа с семафором осуществляется с помощью функций *sem_wait(3)* и *sem_post(3)*. Единственным аргументом функции *sem_wait()* служит указатель на идентификатор семафора. Функция *sem_wait()* приостанавливает выполнение вызвавшего ее потока до тех пор, пока значение семафора не станет большим нуля, после чего функция уменьшает значение семафора на единицу и возвращает управление. Функция *sem_post()* увеличивает значение семафора, идентификатор которого был передан ей в каче-стве параметра, на единицу. Когда приложение больше не нуждается в семафорах — мы вызываем функцию *sem_destroy(3)* для удаления семафора и высвобождения его ресурсов.

Семафоры — не единственное средство синхронизации пото-ков. Для разграничения доступа к глобальным объектам потоки мо-гут использовать мьютексы. Все функции и типы данных, имеющие отношение к мьютексам, определены в файле *pthread.h.* Мьютекс создается вызовом функции *pthread_mutex_init(3)*. В качестве первого аргумента этой функции передается указатель на переменную *pthread_mutex_t*, которая играет роль идентификатора нового мью-

как индивидуальный поток выполнения внутри одного и того же процесса. При этом процесс делится на две части. Одна часть содержит ресурсы, используемые через всю программу, такие как программный код и глобальные данные. Другая содержит информацию, относящуюся к состоянию выполнения, например, программный счетчик и стек. Таким образом, мы вплотную подходим к определению потока.

Кратко поток (thread, нить выполнения) можно определить как ис-полняемую сущность процесса.

Выделим основные причины, которые подталкивают программистов создавать многопоточные приложения:

Повышение надежности программы. Зацикливание основного потока приложения полностью блокирует его работу, при этом приложение может быть завершено лишь при помощи диспетчера задач (Task Manager), что, как правило, сопровождается потерей несохраненных данных. Поэтому «неблагонадежные» вычислительные фрагменты рекомендуется переносить из основного потока в отдельные дополнительные потоки, предусмотрев возможность их досрочного завершения.

Повышение быстродействия, экономия ресурсов и расшире-ние функциональных возможностей программы. Многопоточность позволяет параллельно выполнять отдельные участки программы на ЭВМ с несколькими процессорами, или выполнять их на одном процессоре «псевдопараллельно», используя механизм вытесняющей многозадачности Windows. Например, различные потоки в Microsoft Word одновременно принимают пользовательский ввод, проверяют орфографию в фоновом режиме и печатают документ. Microsoft Excel строит диаграммы и выполняет математические расчеты в фоновом режиме. Сервер баз данных для ответа на каждый запрос клиента запускает отдельный поток, в противном случае пришлось бы либо запускать отдельную копию сервера, напрасно расходуя ресурсы , либо чрезвычайно усложнять логику его работы . Интерфейс прикладных программ разнообразят анимация, воспроизведение звука и т. п., выполняемые отдельными потоками .

Напомним, что при вытесняющей многозадачности потоки выполняются попеременно, время процессора выделяется потокам квантами (около 19 мс). ОС вытесняет поток, когда истечет его квант или ког-да на очереди поток с большим приоритетом. Приоритеты постоянно

пересчитываются, чтобы избежать монополизации процессора одним потоком.

Создание и работа с потоками

Каждый поток начинает свое выполнение с некоторой входной функции. У функции должен быть следующий прототип:

DWORD WINAPI ThreadProc(*PVOID pPararn*);

Функция потока может выполнять абсолютно любые задачи. Ниже приведена пустая функция, которая ничего не делает.

```
DWORD WINAPI ThreadProc(PVOID
pPararn); {
        return 0;
}
```

Когда эта функция закончит выполнение — поток автоматически завершится. В этот момент система выполняет следующие действия:

Останавливает поток.

Освобождает стек.

Счетчик пользователей для объекта ядра потока уменьшится на 1.

Когда счетчик объекта ядра обнуляется — система его удаляет. Получается, что объект ядра может жить дольше, чем сам поток. Это сделано для того, чтобы остальные части программы могли получать доступ к информации о потоке, даже если его уже не существует. Например, если надо узнать код завершения потока.

Функция потока всегда должна возвращать значение. Именно оно будет использоваться как код завершения потока.

При разработке потоковой функции надо всегда стараться обходиться своими локальными переменными, либо параметрами. Никто не запрещает использовать доступ к глобальным переменным, вызывать статические методы классов или пользоваться указателями/ссылками на внешние объекты. Однако в этом случае надо выполнять дополнительные действия по синхронизации потоков. Об этом вы сможете прочитать в одной из следующих статей.

Итак, у нас есть потоковая функция. Давайте заставим систему создать для нас поток, который выполнит эту функцию.

Практическое занятие № 6. Синхронизация потоков в ОС Linux

Цель работы: изучить механизмы синхронизации в ОС *Linux*.

Порядок выполнения практических заданий

1. Рассмотреть представленный пример, и разработать приложение на его основе.

2. Разработать и реализовать алгоритм решения второго задания, с учетом разделения вычислений между несколькими потоками. Определить критические фрагменты алгоритма и защитить их мьютексами.

Длительность — 4 акад. часа.

Литературные источники

1. W.R. Stevens, S.A. Rago, Advanced Programming in the UNIX® Environment: Second Edition, Addison Wesley Professional, 2005

2. D.P. Bovet, M. Cesati, Understanding the Linux Kernel, 3rd Edition, O'Reilly, 2005

3. А. Боровский. «Потоки». http://www.citforum.ru/programming/unix/threads/.

Теоретическая часть

Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали семафоры, концепцию которых описал Дейкстра (Dijkstra) в 1965 г.

Семафор представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: ***P*** (от датского слова proberen — прове-рять) и ***V*** (от verhogen — увеличивать). Классическое определение этих операций выглядит следующим образом:

потоков одновременно, должна обладать свойством реентерабельности (этим же свойством должны обладать функции, допускающие рекур-сию). Реентерабельная функция, это функция, которая может быть вы-звана повторно, в то время, когда она уже вызвана (отсюда и проис-ходит ее название). Реентерабельные функции используют локальные переменные (и локально выделенную память) в тех случаях, когда их не-реентерабельные аналоги могут воспользоваться глобальными пе-ременными.

Мы вызываем последовательно две функции *pthread_join()* для того, чтобы дождаться завершения обоих потоков. Если мы хотим дождаться завершения всех потоков, порядок вызова функций *pthread_join()* для разных потоков, очевидно, не имеет значения.

Варианты заданий

Перевести приложение, разработанное в теме № 1 на платформу ОС *Linux*.

Создание потока

Создание потока в Windows происходит с помощью вызова *API* фукнции:

```
HANDLE CreateThread(PSECURITY_ATTRIBUTES psa,
DWORD cbStack,
                    PTHREAD_START_ROUTINE pfnStartAddr,
PVOID pvParam, DWORD tdwCreate, PDWORD pdwThreadID);
```

Вызов этой функции создает объект ядра «поток» и возвращает его дескриптор. Система выделяет память под стек нового потока из адресного пространства процесса, инициализирует структуры данных потока и передает управление потоковой функции. Новый поток выполняется в контексте того же процесса, что и родительский поток. Поэтому он имеет доступ ко всем дескрипторам процесса, адресному пространству. Поэтому все потоки могут легко взаимодействовать друг с другом.

Параметры функции *CreateThread* следующие:

psa — указатель на структуру *SECURITY_ATTRIBUTES*. Если вы хотите, чтобы потоку были присвоены параметры защиты по умолчанию — передайте сюда *NULL*.

cbStack — размер стека потока. Если параметр равен нулю — используется размер по умолчанию. Если вы передаете не нулевое значение, система выберет большее между настройками текущего исполняемого файла и вашим значением. Этот параметр резервирует только адресное пространство, а физическая память выделяется по мере необходимости.

pfnStartAddr — это указатель на потоковую функцию. Прототип функции мы рассмотрели выше.

pParam — произвольное значение. Этот параметр идентичен параметру потоковой функции. *CreateThread* передаст этот параметр в потоковую функцию. Это может быть число, либо указатель на структуру данных. Можно создавать несколько потоков с одной и той же потоковой функцией. Каждому потоку можно передавать свое значение.

Внимание, не передавайте сюда указатель на локальные переменные! Так как родительский поток работает одновременно с новым — локальные переменные могут выйти из области видимости и раз-

рушиться компилятором. В то время, как новый поток будет пытаться получить к ним доступ.

tdwCreate — дополнительные параметры создания потока. Может принимать значение 0 если надо начать выполнение потока немедленно, либо *CREATE_SlJSPENDED*. В последнем случае система выполняет всю инициализацию, но не запускает выполнение потока. Поток мож-но запустить в любой момент, вызвав *WinAPI* функцию *ResumeThread.*

pdwThreadID — указатель на переменную, которая на выходе бу-дет содержать идентификатор потока. Windows 2k+ позволяет переда-вать сюда *NULL*, если Вам не нужно это значение. Однако, рекоменду-ется всегда передавать адрес переменной для совместимости с более ранними ОС.

Завершение потока

Поток может завершиться в следующих случаях:

1. Поток самоуничтожается с помощью вызова ***ExitThread*** (не ре-комендуется).
2. Функция потока возвращает управление (рекомендуемый спо-соб).
3. Один из потоков данного или стороннего процесса вызывает функцию ***TerminateThread*** (нежелательный способ).
4. Завершается процесс, содержащий данный поток (тоже неже-лательно).

Функцию потока следует проектировать так, чтобы поток завер-шался только после того, как она возвращает управление. Это един-ственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших Вашему потоку. При этом:

– любые C++-объекты, созданные данным потоком, уничтожают-ся соответствующими деструкторами;

– система корректно освобождает память, которую занимал стек потока;

– система устанавливает код завершения данного потока (поддер-живаемый объектом ядра «поток») — его и возвращает Ваша функция потока;

– счетчик пользователей данного объекта ядра «поток» уменьша-ется на 1.


```
result = pthread_create(&thread2, NULL, thread_func,
&size2); if (result != 0) {
perror(«Creating the second thread»);
return EXIT_FAILURE;
}
result = pthread_join(thread1, NULL);
if (result != 0) {
perror(«Joining the fi rst thread»);
return EXIT_FAILURE;
}
result = pthread_join(thread2, NULL);
if (result != 0) {
perror(«Joining the second thread»);
return EXIT_FAILURE;
}
printf(«Done\n»);
return EXIT_SUCCESS;
}
```

Строка для компиляции проекта будет выглядеть приблизительно следующим образом:

gcc threads.c -o thread –lpthread

Рассмотрим сначала функцию *thread_func()*. Как вы, конечно, дога-дались, это и есть функция потока. Наша функция потока очень проста. В качестве аргумента ей передается указатель на переменную типа int, в которой содержится номер потока. Функция потока распечатывает этот номер несколько раз с интервалом в одну секунду и завершает свою ра-боту. В функции *main()* вы видите две переменных типа *pthread_t*. Мы собираемся создать два потока и у каждого из них должен быть свой идентификатор. Вы также видите две переменные типа *int*, *id*1 и *id*2, которые используются для передачи функциям потоков их номеров. Сами потоки создаются с помощью функции *pthread_create()*.В этом примере мы не модифицируем атрибуты потоков, поэтому во втором параметре в обоих случаях передаем *NULL*. Вызывая *pthread_create()* дважды, мы оба раза передаем в качестве третьего параметра адрес функции *thread_func*, в результате чего два созданных потока будут вы-полнять одну и ту же функцию. Функция, вызываемая из нескольких

```
reverse(&v[i], n-i);
return 1;
}
return 0;
}
void print_vect(int * v, int n)
{
int i;
for (i = 0; i < n — 1; i++)
printf(«%i «, v[i]);
printf(«%i\n», v[n-1]);
}
void * thread_func(void *arg)
{
int i;
int * v;
int size = * (int *) arg;
v = malloc(sizeof(int)*size);
for(i = 0; i < size; i++) v[i] = i+1;
print_vect(v, size);
while(next_permutation(v, size)) {
print_vect(v, size);
sync();
}
free(v);
}
int main(int argc, char * argv[])
{
int size1, size2, result;
pthread_t thread1, thread2;
size1 = 4;
result = pthread_create(&thread1, NULL, thread_func, &size1);
if (result != 0) {
perror(«Creating the fi rst thread»);
return EXIT_FAILURE;
}
size2 = 3;
```


Вызов ***ExitThread*** выполняет аналогичные действия, за исключением первого пункта. Поэтому могут быть проблемы.

Завершение потока принудительным образом извне (***TerminateThread***, завершение процесса) может вызвать проблемы не только с корректным освобождением ресурсов, но и с логикой работы программы. Например, «убиенный» поток не освободит доступ к занятым ресурсам и объектам синхронизации. В результате остальная часть программы может повести себя непредсказуемым образом.

Практическая часть

Пример 1.

В первом примере разработаем диалоговое многопоточное приложение.

Сначала напишем функцию, которая будет сигнализировать системным динамиком. Ее будем вызывать при создании потока.

```
DWORD WINAPI OurFunction (PVOID pParam)
{
Beep(200, 1000); //первый параметр–частота, второй — длительность
return (0);
}
```

Вызов создания потока можно интегрировать в функцию нажатия кнопки или на любое другое событие по вашему выбору. Для вызова потока по кнопке достаточно в код вставить следующий фрагмент:

```
DWORD dwID;
CreateThread(NULL, 0, OurFunction, NULL, NULL, &dwID);
```

После запуска собранного приложения и нажатия на требуемую кнопку мы можем услышать характерное «пищание» системного динамика (в случае если динамик подключен и активирован).

Все способы, за исключением рекомендуемого, являются нежелательными и должны использоваться только в форс-мажорных обстоятельствах.

Функция потока, возвращая управление, гарантирует корректную очистку всех ресурсов, принадлежащих данному потоку. При этом:

– любые C++-объекты, созданные данным потоком, уничтожают-ся соответствующими деструкторами;

– система корректно освобождает память, которую занимал стек потока;

– система устанавливает код завершения данного потока. Его функция и возвращает;

– счетчик пользователей данного объекта ядра (поток) уменьшается на 1.

При желании немедленно завершить поток изнутри используют функцию *ExitThread(DWORD dwExitCode)*.

При этом освобождаются все ресурсы ОС, выделенные данному потоку, но СС++-ресурсы (например, объекты классов C++) не очища-ются. Именно поэтому не рекомендовано завершать поток, используя эту функцию.

Если появилась необходимость уничтожить поток снаружи, то это может сделать функция *TeminateThread*.

Пример 2.

Дана последовательность натуральных чисел *a*0, …, *a*99. Создать многопоточное приложение для поиска суммы квадратов Σai Вычисле-ния должны независимо выполнять четыре потока.

Обсуждение. Разобьем последовательность чисел на четыре части и создадим четыре потока, каждый из которых будет вычислять суммы квадратов элементов в отдельной части последовательности. Главный поток создаст дочерние потоки, соберет данные и вычислит окончательный результат, после того, как отработают четыре дочерних потока (рис. 1.1). Приложение сделаем консольным.

момент вызова *pthread_join()* ожидаемый поток уже завершился, функция вернет управление немедленно. Функцию *pthread_join()* можно рассматривать как эквивалент *waitpid(2)* для потоков. Эта функция позволяет вызвавшему ее потоку дождаться завершения работы другого потока. Попытка выполнить более одного вызова *pthread_join()* (из разных потоков) для одного и того же потока приведет к ошибке.

Практическая часть

Пример 1

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>

void inline swap(int *i, int *j)
{
int t;
t = *i;
*i = *j;
*j = t;
}
void reverse(int * v, int n)
{
int i;
for (i = 0; i < (n/2); i++)
swap(&v[i], &v[n-1-i]);
}
int next_permutation(int * v, int n)
{
int i, j;
i = n — 1;
while ((i > 1) && (v[i] < v[i-1])) i--;
if (v[i] > v[i-1]) {
j = n — 1;
while (v[j] < v[i-1]) j--;
swap(&v[j], &v[i-1]);
```

чи значения, возвращаемого функцией потока. Вскоре после вызова *pthread_create()* функция потока будет запущена на выполнение парал-лельно с другими потоками программы. Таким образом, собственно, и создается новый поток. Я говорю, что новый поток запускается «вско-ре» после вызова *pthread_create()* потому, что перед тем как запустить новую функцию потока, нужно выполнить некоторые подготовитель-ные действия, а поток-родитель между тем продолжает выполняться. Непонимание этого факта может привести вас к ошибкам, которые трудно будет обнаружить. Если в ходе создания потока возникла ошиб-ка, функция *pthread_create()* возвращает ненулевое значение, соответ-ствующее номеру ошибки.

Функция потока должна иметь заголовок вида:

*void * func_name(void * arg)*

Имя функции, естественно, может быть любым. Аргумент *arg*, — это тот самый указатель, который передается в последнем параметре функции *pthread_create()*. Функция потока может вернуть значение, которое затем будет проанализировано заинтересованным потоком, но это не обязательно. Завершение функции потока происходит если:

– функция потока вызвала функцию *pthread_exit(3)*;

– функция потока достигла точки выхода;

– поток был досрочно завершен другим потоком.

Функция *pthread_exit()* представляет собой потоковый аналог функции *_exit()*. Аргумент функции *pthread_exit()*, значение типа *void **, становится возвращаемым значением функции потока. Как (и кому?) функция потока может вернуть значение, если она не вызыва-ется из программы явным образом? Для того, чтобы получить значе-ние, возвращенное функцией потока, нужно воспользоваться функци-ей *pthread_join(3)*. У этой функции два параметра. Первый параметр *pthread_join()*, — это идентификатор потока, второй параметр имеет тип «указатель на нетипизированный указатель». В этом параметре функция *pthread_join()* возвращает значение, возвращенное функцией потока. Конечно, в многопоточном приложении есть и более простые способы организовать передачу данных между потоками. Основная задача функции *pthread_join()* заключается, однако, в синхронизации потоков. Вызов функции *pthread_join()* приостанавливает выполнение вызвавшего ее потока до тех пор, пока поток, чей идентификатор пере-дан функции в качестве аргумента, не завершит свою работу. Если в

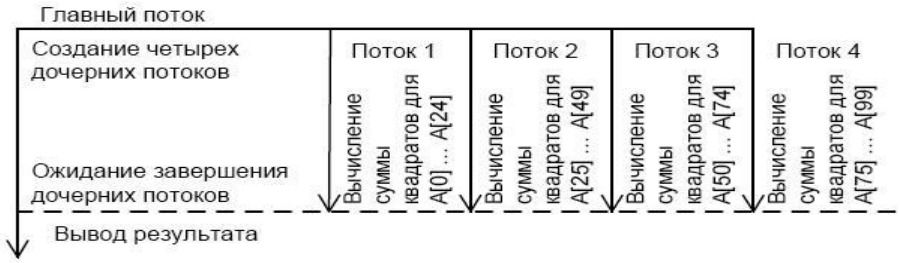

Рис. 1.1. Схема потоков для примера 2

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>
const int n = 4;
int a[100];
DWORD WINAPI ThreadFunc(PVOID pvParam)
{
        int num,sum = 0,i;
        num = 25*(*((int *)pvParam));
        for(i=num;i<num+25;i++) sum += a[i]*a[i];
        *(int*)pvParam = sum;
        DWORD dwResult = num;
        return dwResult;
}
int main(int argc, char** argv)
{
        int x[n];
        int i,rez = 0;
        DWORD dwThreadId[n],dw,dwResult[n];
        HANDLE hThread[n];
        for (i=0;i<100;i++) a[i] = i;
        //создание n дочерних потоков
        for (i=0;i<n;i++)
        {
                x[i] = i;
```

```
            hThread[i] = CreateThread(NULL,0,ThreadFunc,(PVOID)&
            x[i], 0, &dwThreadId[i]);
            if(!hThread) printf(«main process: thread %d not
            execute!»,i);
        }
        // ожидание завершения n потоков
        dw = WaitForMultipleObjects(n,hThread,TRUE,INFINITE);
        // получение значений, переданных потоком в
        return for (i=0;i<n;i++)
        {
            GetExitCodeThread(hThread[i],&dwResult[i]);
            printf(«%d\n»,(int)dwResult[i]);
        }
        for(i=0;i<n;i++) rez+=x[i];
        printf(«\nСумма квадратов =
        %d»,rez); getch();
        return 0;
}
```

Варианты заданий

1. Даны последовательности символов $A = \{a0 \ldots an - 1\}$ и $C = \{c0 \ldots cк - 1\}$. В общем случае $n \neq k$. Создать многопоточное приложение, определяющее, совпадают ли посимвольно строки A и C. Количество потоков является входным параметром программы, количество символов в строках может быть не кратно количеству потоков.

2. Дана последовательность символов $C = \{c0 \ldots cn - 1\}$ и символ b. Создать многопоточное приложение для определения количество вхождений символа b в строку C. Количество потоков является входным параметром программы, количество символов в строке может быть не кратно количеству потоков.

3. Дана последовательность натуральных чисел $\{a0 \ldots an - 1\}$. Соз-дать многопоточное приложение для поиска суммы Σai. Количество потоков является входным параметром программы, потоки проводят вычисления независимо друг от друга, количество символов в строке может быть не кратно количеству потоков.

разделена между независимыми процессами, то доступом к их общим ресурсам управляет операционная система, и вероятность ошибок из-за конфликтов доступа снижается. Впрочем, разделение задачи между несколькими независимыми процессами само по себе не защитит вас от других разновидностей ошибок синхронизации. В пользу потоков можно указать то, что накладные расходы на создание нового потока в многопоточном приложении ниже, чем накладные расходы на создание нового самостоятельного процесса. Уровень контроля над потоками в многопоточном приложении выше, чем уровень контроля приложения над дочерними процессами. Кроме того, многопоточные программы не склонны оставлять за собой вереницы зомби или «осиротевших» неза-висимых процессов.

Первая подсистема потоков в *Linux* появилась около 1996 года и называлась, без лишних затей, — *LinuxThreads*. Рудимент этой подсистемы, который вы найдете в любой современной системе *Linux*, — файл /usr/include/pthread.h, указывает год релиза — 1996 и имя разработчика — Ксавье Лерой (Xavier Leroy). Библиотека *LinuxThreads* была попыткой организовать поддержку потоков в Linux в то время, когда ядро системы еще не предоставляло никаких специальных меха-низмов для работы с потоками. Позднее разработку потоков для *Linux* вели сразу две конкурирующие группы — *NGPT* и *NPTL*. В 2002 году группа *NGPT* фактически присоединилась к *NPTL* и теперь реализация потоков *NPTL* является стандартом *Linux*. Подсистема потоков Linux стремится соответствовать требованиям стандартов POSIX, так что но-вые многопоточные приложения *Linux* должны без проблем компили-роваться на новых *POSIX*-совместимых системах.

Потоки создаются функцией pthread_create, определенной в заголовочном файле <pthread.h>.

int pthread_create(&thread1, NULL, thread_func, &size1);

Первый параметр этой функции представляет собой указатель на переменную типа *pthread_t*, которая служит идентификатором создаваемого потока. Второй параметр, указатель на переменную типа *pthread_attr_t*, используется для передачи атрибутов потока. Третьим параметром функции *pthread_create()* должен быть адрес функции потока. Эта функция играет для потока ту же роль, что функция *main()* — для главной программы. Четвертый параметр функции *pthread_create()* имеет тип *void* *. Этот параметр может использоваться для переда-

Практическое занятие № 5. Создание многопоточных приложений в ОС Linux

Цель работы: научиться создавать простые многопоточные приложения на базе операционной системы Linux.

Порядок выполнения практических заданий

1. Рассмотреть представленный пример, и разработать приложение на его основе.

2. Разработать и реализовать алгоритм решения второго задания, с учетом разделения вычислений между несколькими потоками. Избегать ситуаций изменения одних и тех же общих данных несколькими потоками.

Длительность — 2 акад. часа.

Литературные источники

1. W.R. Stevens, S.A. Rago, Advanced Programming in the UNIX® Environment: Second Edition, Addison Wesley Professional, 2005.

2. D.P. Bovet, M. Cesati, Understanding the Linux Kernel, 3rd Edition, O'Reilly, 2005.

3. Боровский А. «Потоки». http://www.citforum.ru/programming/unix/threads/.

Теоретическая часть

Прежде чем приступать к программированию потоков, следует ответить на вопрос, а нужны ли они вам. С помощью управления процессами в *Linux* можно решить многие задачи, которые в других ОС решаются только с помощью потоков. Потоки часто становятся источниками программных ошибок особого рода. Эти ошибки возникают при использовании потоками разделяемых ресурсов системы (напри-мер, общего адресного пространства) и являются частным случаем бо-лее широкого класса ошибок — ошибок синхронизации. Если задача

4. Дана последовательность натуральных чисел $\{a0 \ldots an - 1\}$. Создать многопоточное приложение для поиска произведения чисел $a0 \times a1 \times \ldots \times an - 1$. Количество потоков является входным параметром программы, потоки проводят вычисления независимо друг от друга, количество символов в строке может быть не кратно количеству потоков.

5. Дана последовательность натуральных чисел $\{a0 \ldots an - 1\}$. Соз-дать многопоточное приложение для поиска максимального ai. Коли-чество потоков является входным параметром программы, потоки про-водят вычисления независимо друг от друга, количество символов в строке может быть не кратно количеству потоков.

6. Дана последовательность натуральных чисел $\{a0 \ldots an - 1\}$. Соз-дать многопоточное приложение для поиска минимального ai. Количе-ство потоков является входным параметром программы, потоки про-водят вычисления независимо друг от друга, количество символов в строке может быть не кратно количеству потоков.

7. Дана последовательность натуральных чисел $\{a0 \ldots an - 1\}$. Создать многопоточное приложение для поиска всех ai, являющихся простыми числами. Количество потоков является входным параметром программы, потоки проводят вычисления независимо друг от друга, количество символов в строке может быть не кратно количеству по-токов.

8. Дана последовательность натуральных чисел $\{a0 \ldots an - 1\}$. Соз-дать многопоточное приложение для поиска всех ai, являющихся ква-дратами, любого натурального числа.

9. Дана последовательность натуральных чисел $\{a0 \ldots an - 1\}$. Соз-дать многопоточное приложение для вычисления выражения $a0 - a1 + a2 - a3 + a4 - a5 + \ldots$.

10. Дана последовательность натуральных чисел $\{a0 \ldots an - 1\}$. Создать многопоточное приложение для поиска суммы Σai, где ai — четные числа.

11. Изготовление знаменитого самурайского меча — катаны происходит в три этапа. Сначала младший ученик мастера выковывает заготовку будущего меча. Затем старший ученик мастера закаливает меч в трех водах — кипящей, студеной и теплой. И в конце мастер собственноручно изготавливает рукоять меча и наносит узоры. Требуется создать многопоточное приложение, в котором мастер и его ученики

представлены разными потоками. Изготовление меча представить в виде разных арифметических операций над глобальной переменной.

12. Командиру N-ской ВЧ полковнику Кузнецову требуется перемножить два секретных числа. Полковник Кузнецов вызывает дежурного по части лейтенанта Смирнова и требует в течение получаса предоставить ему ответ. Лейтенант Смирнов будит старшего по караулу сержанта Петрова и приказывает ему в 15 минут предоставить ответ. Сержант Петров вызывает к себе рядового Иванова, бывшего студента, и поручает ему ответственное задание по определению произведения. Рядовой Иванов успешно справляется с поставленной задачей и ответ по цепочке передается полковнику Кузнецову. Требуется создать многопоточное приложение, в котором все военнослужащие от полковника до рядового моделируются потоками одного вида.

13. Даны результаты сдачи экзамена по курсу «Параллельные и распределенные вычисления» по группам. Требуется создать многопоточное приложение, вычисляющее средний балл. Потоки должны осуществлять вычисления параллельно по группам. Количество потоков является входным параметром программы, потоки проводят вычисления независимо друг от друга, количество групп может быть не кратно количеству потоков.

14. Охранное агентство разработало новую систему управления электронными замками. Для открытия двери клиент обязан произнести произвольную фразу из 25 слов. В этой фразе должно встречаться заранее оговоренное слово, причем только один раз. Требуется создать многопоточное приложение, управляющее замком. Потоки должны осуществлять сравнение параллельно по словам.

15. Среди студентов нашего университета проведен опрос с целью определения процента студентов, знающих точную формулировку правила Буравчика. В результате собраны данные о количестве знатоков на каждом направлении по группам. Известно, что всего в филиале обучается 500 студентов. Требуется создать многопоточное приложение для определения процента знающих правило Буравчика студентов. Потоки должны осуществлять поиск количества знатоков по факультету. Искомый процент определяет главный поток. Количество потоков является входным параметром программы, потоки проводят вычисления независимо друг от друга, количество направлений может быть не кратно количеству потоков.


```
    int nsize;
    while ((nsize = recv(my_sock, &buff[0], sizeof(buff) — 1, 0)) != SOCK-
ET_ERROR)
    {
        // ставим завершающий ноль в конце строки
        buff[nsize] = 0;
        // выводим на экран
        printf(«S=>C:%s», buff);
        // читаем пользовательский ввод с клавиатуры
        printf(«S<=C:»); fgets(&buff[0], sizeof(buff) — 1, stdin);
        // проверка на «quit»
        if (!strcmp(&buff[0], «quit\n»))
        {
            // Корректный выход
            printf(«Exit...»);
            closesocket(my_sock);
            WSACleanup();
            return 0;
        }
        // передаем строку клиента серверу
        send(my_sock, &buff[0], strlen(&buff[0]), 0);
    }
    printf(«Recv error %d\n», WSAGetLastError());
    closesocket(my_sock);
    WSACleanup();
    return -1;
}
```

Варианты заданий

Доработать программу задания темы № 1.

Клиент должен сформировать пакет данных для расчетов на сервере и передать их по каналу связи. Формат сообщений разработать самостоятельно.

```
    return -1;
  }
  // Шаг 3 — установка соединения
  // заполнение структуры sockaddr_in — указание адреса и порта
сервера
  sockaddr_in dest_addr;
  dest_addr.sin_family = AF_INET;
  dest_addr.sin_port = htons(PORT);
  HOSTENT *hst;
  // преобразование IP адреса из символьного в сетевой формат
  if (inet_addr(SERVERADDR) != INADDR_NONE)
    dest_addr.sin_addr.s_addr = inet_addr(SERVERADDR);
  else
  {
    // попытка получить IP адрес по доменному имени сервера
    if (hst = gethostbyname(SERVERADDR))
      // hst->h_addr_list содержит не массив адресов,
      // а массив указателей на адреса
      ((unsigned long *)&dest_addr.sin_addr)[0] =
      ((unsigned long **)hst->h_addr_list)[0][0];
    else
    {
      printf(«Invalid address %s\n», SERVERADDR);
      closesocket(my_sock);
      WSACleanup();
      return -1;
    }
  }
  // адрес сервера получен — пытаемся установить соединение
  if (connect(my_sock, (sockaddr *)&dest_addr, sizeof(dest_addr)))
  {
    printf(«Connect error %d\n», WSAGetLastError());
    return -1;
  }
  printf(«Соединение с %s успешно установлено\n \
      Type quit for quit\n\n», SERVERADDR);
  // Шаг 4 — чтение и передача сообщений
```


16. Руководство заготовительной компании «Рога и Копыта» проводит соревнование по заготовке рогов среди своих региональных отделений. Все данные по результатам заготовки рогов (заготовитель, его результат) хранятся в общей базе данных по отделениям. Требуется создать многопоточное приложение для поиска лучшего заготовителя. Потоки должны осуществлять поиск победителя параллельно по отделениям. Главный поток определит победителя. Количество потоков является входным параметром программы, потоки проводят вычисления независимо друг от друга, количество отделений может быть не кратно количеству потоков.

Практическое занятие № 2. Синхронизация потоков в ОС Windows

Цель работы: изучить работу с критическими секциями. Научиться выделять «проблемные» фрагменты алгоритма и защищать их с помощью критических секций.

Порядок выполнения практических заданий

1. Рассмотреть представленные примеры, и разработать приложе-ния на их основе.
2. Разработать алгоритм решения третьего задания, с учетом разделения вычислений между несколькими потоками. Определить критические фрагменты алгоритма и защитить их критическими секциями.
3. Реализовать алгоритм с применением функций *WinAPI* и протестировать его на нескольких примерах.

Длительность — 6 акад. часов.

Литературные источники

1. Рихтер, Дж. Windows для профессионалов : создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер. — СПб. : Питер, 2001. — 752 с.
2. Эндрюс, Г.Р. Основы многопоточного, параллельного и распределенного программирования / Г.Р. Эндрюс. — М. : «Вильямс», 2003. — 512 с.
3. Хованский, Е.П. Лабораторные работы по курсу «Параллельные и распределенные вычисления» / Е.П. Хованский.
4. Гергель, В.П. Теория и практика параллельных вычислений / В.П. Гергель. — М. : ИНТУИР.РУ Интернет-Университет Информаци-онных Технологий, 2007.


```
    int bytes_recv;
    while ((bytes_recv = recv(my_sock, &buff[0], sizeof(buff), 0)) &&
    bytes_recv != SOCKET_ERROR) send(my_sock, &buff[0],
    bytes_recv, 0);
    // если мы здесь, то произошел выход из цикла по причине
    // возращения функцией recv ошибки — соединение с
клиентом разорвано
    nclients--; // уменьшаем счетчик активных
    клиентов printf(«-disconnect\n»); PRINTNUSERS
    // закрываем сокет
    closesocket(my_sock);
    return 0;
}
```

Пример реализации TCP-клиента.

```
// Пример простого TCP-
клиента #include <stdio.h>
#include <string.h>
#include <winsock2.h>
#include <windows.h>
#defi ne PORT 666
#defi ne SERVERADDR «127.0.0.1»
int main(int argc, char* argv[])
{
    char buff[1024];
    printf(«TCP DEMO CLIENT\n»);
    // Шаг 1 — инициализация библиотеки Winsock if
    (WSAStartup(0x202, (WSADATA *)&buff[0]))
    {
        printf(«WSAStart error %d\n»,
        WSAGetLastError()); return -1;
    }
    // Шаг 2 — создание сокета
    SOCKET my_sock;
    my_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (my_sock < 0)
    {
        printf(«Socket() error %d\n», WSAGetLastError());
```

```
   // цикл извлечения запросов на подключение из очереди
   while ((client_socket = accept(mysocket, (sockaddr *)&client_addr, \
         &client_addr_size)))
   {
      nclients++; // увеличиваем счетчик подключившихся клиентов
      // пытаемся получить имя хоста
      HOSTENT *hst;
      hst = gethostbyaddr((char *)&client_addr. sin_addr.s_addr, 4, AF_
INET);
      // вывод сведений о клиенте
      printf(«+%s [%s] new connect!\n»,
      (hst) ? hst->h_name : «», inet_ntoa(client_addr.sin_addr));
      PRINTNUSERS
      // Вызов нового потока для обслуживания клиента
      // Да, для этого рекомендуется использовать _beginthreadex
      // но, поскольку никаких вызовов функций стандартной Си би-
блиотеки
      // поток не делает, можно обойтись и CreateThread
      DWORD thID;
      CreateThread(NULL, NULL, SexToClient, &client_socket, NULL,
&thID);
   }
   return 0;
}
// Эта функция создается в отдельном потоке
// и обсуживает очередного подключившегося клиента независимо
от остальных
DWORD WINAPI SexToClient(LPVOID client_socket)
{
   SOCKET my_sock;
   my_sock = ((SOCKET *)client_socket)[0];
   char buff[20 * 1024];
   #define sHELLO «Hello, Sailor\r\n»
   // отправляем клиенту приветствие
   send(my_sock, sHELLO, sizeof(sHELLO), 0);
   // цикл эхо-сервера: прием строки от клиента и возвращение ее
клиенту
```


Теоретическая часть

Критические секции

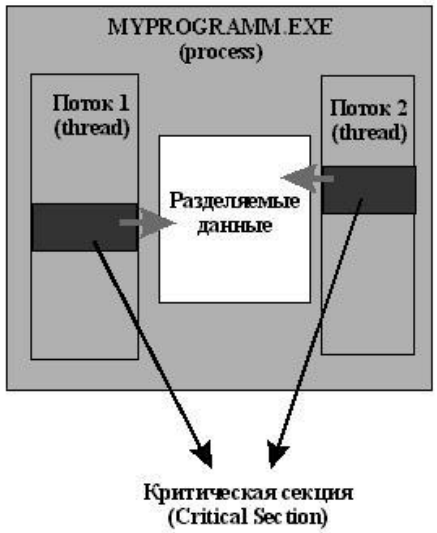

Рис. 2.1. Критическая секция

Критическая секция (critical section) (рис. 2.1) — это небольшой участок кода, требующий монопольного доступа к каким-то общим данным. Она позволяет сделать так, чтобы единовременно только один поток получал доступ к определенному ресурсу. Естественно, система может в любой момент вытеснить Ваш поток и подключить к процессору другой, но ни один из потоков, которым нужен занятый Вами ресурс, не получит процессорное время до тех пор, пока Ваш поток не выйдет за границы критической секции.

Ниже приведен фрагмент кода, который демонстрирует, что может произойти без критической секции:

```
const int MAX_TIMES = 1000,
int g_nIndex — 0,
DWORD g_dwTimes[MAX_TIMES];
DWORD WINAPI FirstThread(PVOID pvParam)
```

```
{
        while (g_nIndex < MAX_TIMES)
        {
            g_dwTimes[g__nIndex] = GetTickCount();
            g_nIndex++;
  }
return(0),
}

  DWORD WINAPI SecondThread(PVOID pvParam)
  {
          while Cg_nIndex < MAX_TIMES)
          {
                  g_nIndex++;
                  g_dwTimes[g_nIndex — 1] = GetTickCount();
          }
          return(0);
  }
```

Здесь предполагается, что функции обоих потоков дают одинаковый результат, хоть они и закодированы с небольшими различиями. Если бы исполнялась только функция *FirstThread*, она заполнила бы массив *g_dwTimes* набором чисел с возрастающими значениями. Это верно и в отношении *SecondThread* — если бы она тоже исполнялась независимо. В идеале обе функции даже при одновременном выполне-нии должны бы по-прежнему заполнять массив тем же набором чисел. Но в нашем коде возникает проблема: масив *g_dwTimes* не будет за-полнен, как надо, потому что функции обоих потоков одновременно обращаются к одним и тем же глобальным переменным. Вот как это может произойти.

Допустим, мы только что начали исполнение обоих потоков в системе с одним процессором. Первым включился в работу второй по-ток, то есть функция *SecondThread* (что вполне вероятно), и только она успела увеличить счетчик *g_nIndex* 1, как система вытеснила ее по-ток и перешла к исполнению *FtrstThread*. Та заносит в *g_dwTimes[1]* показания системного времени, и процессор вновь переключается на исполнение второго потока. *SecondThread* теперь присваивает элемен-


```
    // Ошибка!
    printf(«Error socket %d\n», WSAGetLastError());
    WSACleanup(); // Деинициализация библиотеки
    Winsock return -1;
  }
  // Шаг 3 — связывание сокета с локальным
  адресом sockaddr_in local_addr;
  local_addr.sin_family = AF_INET;
  local_addr.sin_port = htons(MY_PORT); // не забываем о сетевом
порядке!!!
  local_addr.sin_addr.s_addr = 0; // сервер принимает подключения
                        // на все свои IP-адреса
  // вызываем bind для связывания
  if (bind(mysocket, (sockaddr *)&local_addr, sizeof(local_addr)))
  {
    // Ошибка
    printf(«Error bind %d\n», WSAGetLastError());
    closesocket(mysocket); // закрываем сокет!
    WSACleanup();
    return -1;
  }
  // Шаг 4 — ожидание подключений
  // размер очереди — 0x100
  if (listen(mysocket, 0x100))
  {
    // Ошибка
    printf(«Error listen %d\n», WSAGetLastError());
    closesocket(mysocket);
    WSACleanup();
    return -1;
  }
  printf(«Ожидание подключений...\n»);
  // Шаг 5 — извлекаем сообщение из очереди
  SOCKET client_socket; // сокет для клиента
  sockaddr_in client_addr; // адрес клиента (заполняется системой)
  // функции accept необходимо передать размер структуры
  int client_addr_size = sizeof(client_addr);
```

```
#include <windows.h>
#defi ne MY_PORT 666 // Порт, который слушает сервер 666 //
макрос для печати количества активных пользователей
#defi ne PRINTNUSERS if (nclients) printf(«%d user on-line\n», nclients);
      \ else printf(«No User on line\n»);
// прототип функции, обслуживающий подключившихся
пользовате-лей
DWORD WINAPI SexToClient(LPVOID client_socket);
// глобальная переменная — количество активных
пользователей int nclients = 0;
int main(int argc, char* argv[])
{
   char buff[1024]; // Буфер для различных
   нужд printf(«TCP SERVER DEMO\n»);
   // Шаг 1 — Инициализация Библиотеки Сокетов
   // так как возвращенная функцией информация не используется
   // ей передается указатель на рабочий буфер, преобразуемый
к указателю
   // на структуру WSADATA.
   // Такой прием позволяет сэкономить одну переменную,
однако, буфер
   // должен быть не менее полкилобайта размером (структура
WSADATA
   // занимает 400 байт)
   if (WSAStartup(0x0202, (WSADATA *)&buff[0]))
   {
      // Ошибка!
      printf(«Error WSAStartup %d\n», WSAGetLastError());
      return -1;
   }
   // Шаг 2 — создание
   сокета SOCKET mysocket;
   // AF_INET — сокет Интернета
   // SOCK_STREAM — потоковый сокет (с установкой соединения)
   // 0 — по умолчанию выбирается TCP протокол
   if ((mysocket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   {
```


ту *g_dwTtmes[1 — 1]* новые показания системного времени. Поскольку эта операция выполняется позже, новые показания, естественно, выше, чем записанные в элемент *g_dwTimes[1]функ* цией *FirstThread.* Отметьте также, что сначала заполняется первый элемент массива и только потом нулевой. Таким образом, данные в массиве оказываются ошибочными.

Пример приведенный выше в значительной степени надуман, но прост. Рассмотрим пример с управлением связанным списком объектов. Если доступ к связанному списку не синхронизирован, один поток может добавить элемент в список в тот момент, когда другой поток пытается найти в нем какой-то элемент. Ситуация станет еще более угрожающей, если оба потока одновременно добавят в список новые элементы. Так что, используя критические секции, можно и нужно ко-ординировать доступ потоков к структурам данных.

Важно отметить, что критические секции — это механизм для синхронизации потоков внутри одного процесса. Для работы с критическими секциями есть ряд функций ***API*** и тип данных ***CRITICAL_SECTION***.

Для использования критической секции нужно создать переменную данного типа, и проинициализировать ее перед использованием с помощью функции ***InitializeCriticalSection()***.

```
CRITICAL_SECTION g_cs;
InitializeCriticalSection(&g_cs);
```

Для того, чтобы войти в секцию нужно вызвать функ-цию ***EnterCriticalSection()***, а после завершения работы ***LeaveCriticalSection()***.

```
EnterCriticalSection(&g_cs);
LeaveCriticalSection(&g_cs);
```

Что будет, если поток обратится к секции, в которой сейчас другой поток? Тот поток, который обратится будет блокирован пока критиче-ская секция не будет освобождена. Саму критическую секцию можно удалить функцией ***DeleteCriticalSection()***.

```
DeleteCriticalSection(&g_cs);
```

Для того, чтобы обойти блокировку потока при обращении к заня-той секции есть функция ***TryEnterCriticalSection()***, которая позволяет проверить критическую секцию на занятость.

Практическая часть

Пример 1.

Рассмотрим использование критических секций и полезность такого решения. Для начала посмотрим, что случится, если мы попробуем обратиться к одному ресурсу двумя потоками. Создадим два потока, которые одновременно захотят «пищать» системным динамиком. Сначала нарисуем две функции, которые будут выполняться потоками, а потом и сами потоки.

```
DWORD WINAPI firstFunc(LPVOID lpParam)
{
        Beep(100, 500);
        return (0);
}

DWORD WINAPI secondFunc(LPVOID lpParam)
{
        Beep(400, 500);
        return (0);
}
```

Вызовы *CreateThread* можно интегрировать, например, в функцию нажатия на кнопку.

```
DWORD dwFirst;
HANDLE fi rstThread = CreateThread(NULL, 0, fi rstFunc, NULL,
NULL, dwFirst);
DWORD dwSecond;
HANDLE secondThread = CreateThread(NULL, 0, secondFunc,
NULL, NULL, dwSecond);
/* Когда описатели нам больше не требуются, удалим их */
CloseHandle(fi rstThread);
CloseHandle(secondThread);
```


поменьше. Добавим большому полю переменную *m_strSimpleDisplay* — здесь будет выдаваться полученная информация. А маленькому полю добавим переменную *m_strReturned* — тут будем выдавать коли-чество полученных байт.

```
m_strSimpleDisplay = «»;
m_strReturned = «»;
UpdateData(FALSE);
```

Начнем принимать информацию

```
while (1)
{
        nret = recv(theSocket, bufferRecv, sizeof(bufferRecv),
0); //заполняем поле m_strReturned
        char charRet[128] = »»;
        itoa(nret, charRet, 10);
        strcat(charRet, « bytes\r\n»);

        m_strReturned += ret;

//заполняем поле m_strSimpleDisplay
        m_strSimpleDisplay += bufferRecv;

        UpdateData(FALSE);
        if (nret == 0 || nret == -1)
                break;
}
```

Компилируем, запускаем и нажимаем кнопку SendRequest. Смотрим, каким образом присылаются данные.

Пример 2

Задача — реализовать *TCP* эхо-сервер.

Пример простого TCP-эхо-сервера

```
#include <stdio.h>
#include <winsock2.h> // Wincosk2.h должен быть раньше windows!
```

Мы заполнили все нужные структуры, знаем порт и IP-адрес, теперь пора связываться с сервером. Проверка того же типа, что и раньше.

```
nret = connect(theSocket, (LPSOCKADDR) &serverInfo, sizeof(struct
sock-addr));
if (nret != 0)
{
        hr =
        HRESULT_FROM_WIN32(WSAGetLastError());
        ReportError(hr, «connect()»); WSACleanup();
        return;
}
```

Начнем процедуру общения с сервером. Вопросы он понимает не все, а только правильно сформулированные. Мы попросим дать нам страницу *index.htm*.

```
//в этом буфере будет запрос серверу
char bufferSend[128] = «»;
/*Сформируем запрос. Большинство серверов воспринимают
пустой запрос GET / как запрос на страничку index.htm. В конце
запроса ОБЯ-ЗАТЕЛЬНО должна идти пустая строка, иначе сервер
не прекратит нас слушать.*/
strcpy(bufferSend, «GET / HTTP/1.0\r\n»);
strcat(bufferSend, «\r\n»);
//спрашиваем
nret = send(theSocket, bufferSend, strlen(bufferSend),
0); //проверяем на ошибки
if (nret == SOCKET_ERROR)
{
        hr = HRESULT_FROM_WIN32(WSAGetLastError());
        ReportError(hr, «send()»);
        WSACleanup();
        return;
}
```

Мы задали вопрос и должны ждать и принимать ответ. Сначала создадим на диалоговом окне два поля *EditBox* — одно побольше, другое

Как вы думаете, что произойдет? Правильно, ничего того, что мы ожидали. Потоки будут драться и так как приоритеты у них одинаковы, то «пищать» будет только один — какой, не известно, но с большей вероятностью первый вызванный. Можете сами прослушать.

Разрешить данную ситуацию можно с помощью критической секции. Обычно структуры *CRITICAL_SECTION* создаются как глобаль-ные переменные, доступные всем потокам процесса. Однако вы може-те сузить область видимости данной структуры, исходя из специфики задач. Главное, чтобы структура *CRITICAL_SECTION* была в области видимости тех потоков, которые будут обращаться к разделяемому ресурсу. Следует выполнять два условия. Во-первых, все потоки, кото-рым может понадобиться данный ресурс, должны знать *адрес* струк-туры *CRITICAL_SECTION*, которая защищает этот ресурс. Во-вторых, элементы структуры *CRITICAL_SECTION* следует инициализировать до обращения какого-либо потока к защищенному ресурсу.

CRITICAL_SECTION csOurSection;//это объявление структуры Критическая секция

Помня про условия использования критической секции, мы ее инициализируем. Эту операцию провернем в функции OnInitDialog. Не забываем, что критическую секцию следует инициализировать до какого-либо к ней обращения.

InitializeCriticalSection(&csOurSection);

Теперь перепишем наши функции, учитывая наличие критической секции.

```
    DWORD WINAPI firstFunc(LPVOID lpParam)
{
        EnterCriticalSection(&csOurSection);
        Beep(100, 500);
        LeaveCriticalSection(&csOurSection);
return (0);
}

DWORD WINAPI SecondFunc(LPVOID lpParam)
{
        EnterCriticalSection(&csOurSection);
        Beep(400, 500);
```

```
        LeaveCriticalSection(&csOurSection);
return (0);
}
```

Вызовы создания потоков, естественно, такие же.

Когда мы понимаем, что критическая секция нам больше не нуж-на, мы должны ее удалить. Сделать это можно, например, в функции OnClose. (Создать ее можно с помощью MFC Class Wizard, по сообщению WM_CLOSE нашего диалогового окна.). Для этого надо сделать вызов:

```
DeleteCriticalSection(&csOurSection);
```

Теперь первым выполняется поток, который первым вошел в кри-тическую секцию. Второй ждет, а затем и сам выполняется. Сигнали-зируют потоки по очереди.

Пример 2.

Задача о кольцевом буфере. Потоки производители и потребители разделяют кольцевой буфер, состоящий из 100 ячеек. Производители передают сообщение потребителям, помещая его в конец очереди буфера. Потребители сообщение извлекают из начала очереди буфера. Создать многопоточное приложение с потоками писателями и читателями. Предотвратить такие ситуации как, изъятие сообщения из пустой очереди или помещение сообщения в полный буфер. При решении задачи использовать критические секции.

Пусть для определенности буфер — это целочисленный массив из 100 элементов. Задача обладает двумя проблемными участками алгоритма. Первый из них связан с операциями чтения-записи нескольких потоков в общий буфер. Второй проблемный фрагмент определяет-ся тем, что буфер являются конечным, запись должна производиться только в те ячейки, которые являются свободными или уже прочитаны потоками-читателями (условная взаимная синхронизация).

Для защиты обоих проблемных участков воспользуемся критической секцией. Она сделает возможным запись в буфер только одного потока-писателя. И она же сделает возможным чтение из буфера толь-ко одного потока-читателя. Операция чтения должна быть защище-на, потому что она является и операцией записи тоже, так как поток, прочитавший ячейку буфера, обязан ее как-то пометить. Иначе, через определенное время выполнения программы, операция записи может


```
if (hostEntry == NULL)
{
        hr = HRESULT_FROM_WIN32(WSAGetLastError());
        ReportError(hr, «gethostbyname()»);
        WSACleanup();
        return;
}
```

Здесь мы получаем код ошибки (в случае, если ошибка есть) с по-мощью конструкции HRESULT_FROM_WIN32(WSAGetLastError()). Функция *FormatMessage* требует для работы переменной типа *HRESULT* — для этого код, полученный с помощью функции *WSAGetLastError*, мы преобразуем с помощью макроса *HRESULT_FROM_WIN32*.

Затем создаем сокет и проверяем на правильность создания.

```
SOCKET theSocket;
theSocket = socket(AF_INET,    //go over TCP/IP
                   SOCK_STREAM,        //stream-oriented socket
                   IPPROTO_TCP);       //TCP

if (theSocket == INVALID_SOCKET)
{
        hr =
        HRESULT_FROM_WIN32(WSAGetLastError());
        ReportError(hr, «socket()»); WSACleanup();
        return;
}
```

Для дальнейшей работы заполним структуру *SOCKADDR_IN*:

```
SOCKADDR_IN serverInfo;
serverInfo.sin_family = AF_INET;
serverInfo.sin_addr = *((LPIN_ADDR)*hostEntry->h_addr_list);
serverInfo.sin_port = htons(80);
```

Остается добавить, что функция htons() переводит прямой поря-док байт в порядок, используемый в сети.

```
        return;
}
```

Сама функция *ReportMessage()* принимает в качестве первого параметра код ошибки, в качестве второго параметра она принимает имя функции, из которой сигнализируется об ошибке.

Начнем описывать реакцию программы на нажатие кнопки *SendRequest*. Реализация начинается с инициализации Winsock'а.

```
WORD sockVersion;
WSADATA wsaData;
sockVersion = MAKEWORD(1, 1);        //используем версию 1.1
WSAStartup(sockVersion, &wsaData);   //инициализируем Winsock
```

Еще нам понадобится переменная для хранения результата опера-ций *int nret;* и переменная для кода ошибки *HRESULT hr;*

Теперь заполним структуру *HOSTENT*, которая говорит сокету с каким компом и портом связываться. Эта структура обычно фигурирует как переменные типа *LPHOSTENT*, которые являются попросту указателями на *HOSTENT*.

```
LPHOSTENT hostEntry;
in_addr iaHost;
iaHost.s_addr=inet_addr(«192.168.0.253»);//это адрес университетского сервера Debian
hostEntry = gethostbyaddr((const char*) &iaHost, sizeof(struct in_addr), AF_INET);
```

Функция *gethostbyaddr* заполняет *HOSTENT* пригодными для дальнейшего использования значениями в случае, когда известен IP-адрес сервера. Иначе

```
hostEntry = gethostbyname(«www.uni-protvino.ru»); /*адрес того же университетского сервера */
```

и, соответственно, переменная *iaHost* нам не нужна.

Можно выбрать функцию на свой вкус. Так как мы знаем IP-адрес сервера, то удобнее пользоваться *gethostbyaddr*.

Далее проверим, что мы получили.

стать невозможной или некорректной, в силу того, что буфер конечен. Операции чтения и записи могут проходить параллельно, так как всег-да происходят в разных ячейках.

За условную синхронизацию будет отвечать та же самая критическая секция. Также, должна присутствовать переменная, отображаю-щая, сколько ячеек в буфере свободно. Ячейка свободна, когда в нее еще не осуществлялась запись или ячейка была прочитана. Вторая переменная должна показывать, сколько ячеек в буфере занято. Есте-ственно, операция записи не может быть выполнена, пока количество занятых ячеек равно 100 (или количество свободных ячеек равно 0), и операция чтения не может быть выполнена, пока количество свобод-ных ячеек равно 100 (или количество занятых ячеек равно 0).

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <windows.h>
const int n = 100, // блина буфера
m = 7, // количество производителей
k = 3; // количество потребителей
int buf[n], front = 0, rear = 0;//кольцевой буфер его голова и хвост CRITICAL_SECTION ArraySection;//секция на доступ к буферу //процесс, пополняющий буфер
DWORD WINAPI Producer(PVOID pvParam)
{
        int num;
        num = *((int *)pvParam);
        printf(«thread %d (producer): start!\n»,num);
        while(true)
        {
                EnterCriticalSection( &ArraySection );
                buf[rear] = rand()%n;
                printf(«\nproducer %d: data = %d to %d», num, buf[rear], rear);
                rear = (rear+1)%n;
                LeaveCriticalSection( &ArraySection );
                Sleep(1000);
```

```
	}
	return 0;
}
//процесс, берущий данные из буфера
DWORD WINAPI Consumer(PVOID pvParam)
{
	int num=0;
	int data=0;
	long prev=0;
	num = *((int *)pvParam);
	printf(«thread %d (consumer): start!\n»,num);
	while(true)
	{
		EnterCriticalSection( &ArraySection );
		data = buf[front];
		printf(«\nconsumer %d: data = %d from %d», num, data,
front);
		front = (front+1)%n;
		LeaveCriticalSection( &ArraySection );
		Sleep(1000);
	}
	return 0;
}

int main(int argc, char* argv)
{
	int i, x[k+m];
	DWORD dwThreadId[k+m];
	HANDLE hThread[k+m];
	InitializeCriticalSection(&ArraySection);
	for(i=0;i<k;i++)
	{
		x[i] = i;
		hThread[i] =
CreateThread(NULL,0,Producer,(PVOID)&x [i], 0, &dwThreadId[i]);
		if(!hThread) printf(«main process: thread %d not ex-
ecute!», i);
```


вращает указатель на строку, если преобразование выполнено успешно и ноль в противном случае.

Практическая часть

Задача 1

Наше сетевое программирование начнем с написания программы-клиента, использующей сокеты (*sockets*).

Итак, начнем. Создадим диалоговое приложение, не забывая поставить галочку около поддержки сокетов во вкладке *Advanced*. Созда-дим кнопку *Send Request*, по нажатию на которую будут происходить все действия.

Напишем функцию, сообщающую нам об ошибках. Раньше мы не писали их за отсутствием острой потребности (хотя это неправильно). Для сокетов же проверки критичны и помогут избежать многих трудностей и неполадок. Чтобы нам было понятно, в чем состоит ошибка, мы ее расшифруем на нормальный человеческий язык, заставив систе-му саму расшифровывать нам коды ошибок. Делается это с помощью функции *FormatMessage*. В-общем, она принимает код ошибки в третьем параметре и записывает ответ в переменную типа *char*, передан-ную ей в качестве пятого параметра. Также ей надо знать размер этой переменной — он передается шестым параметром.

```
void ReportError(HRESULT errorCode, const char *whichFunc)
{
  //расшифровываем ошибку
	char chErrMsg[1024];
	FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, NULL,
errorCode, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
chErrMsg, sizeof(chErrMsg), NULL);
//форматим для вывода пользователю
	CString strErrorMsg;
	strErrorMsg.Format(«Вызов функции %s вернул следующую
ошибку (код%d): \n%s», (char*)whichFunc, errorCode, chErrMsg);
	MessageBox(NULL, strErrorMsg, «socketIndication», MB_OK);
//возврат к диалоговому окну :-)
```

В общем-то, ничего не изменилось — замена беззнакового короткого целого на знаковое короткое целое для представления семейства протоколов ничего не дает. Зато теперь адрес узла представлен в виде трех полей — *sin_port* (номера порта), *sin_addr* (IP-адреса узла) и «хвоста» из восьми нулевых байт, который остался от 14-символьного массива *sa_data*. Для чего он нужен? Дело в том, что структура *sockaddr* не привязана именно к Интернет и может работать и с другими сетями. Адреса же некоторых сетей требуют для своего представления гораздо больше четырех байт — вот и приходится брать с запасом!

Структура *in_addr* определяется следующим образом:

```
struct in_addr {
   union {
      struct {u_char s_b1, s_b2, s_b3, s_b4;} S_un_b; // IP-адрес
      struct {u_short s_w1, s_w2;} S_un_w; // IP-адрес u_long
      S_addr; // IP-адрес
   } S_un;
};
```

Структура hostent выглядит следующим образом:

```
struct hostent
{
   char FAR * h_name;          // официальное имя узла
   char FAR * FAR * h_aliases; // альтернативные имена узла (массив
строк)
   short h_addrtype;           // тип адреса
   short h_length;             // длина адреса (как правило AF_INET)
   char FAR * FAR * h_addr_list; // список указателей на IP-
                        адреса // ноль — конец списка
};
```

Определение имени узла по его адресу бывает полезным для серверов, желающих «в лицо» знать своих клиентов.

Для преобразования IP-адреса, записанного в сетевом формате в символьную строку, предусмотрена функция *«char FAR * **inet_ntoa** (struct in_addr)»*, которая принимает на вход структуру *in_addr*, а воз-


```
        }
        for(i=k;i<k+m;i++)
        {
                x[i] = i;
                hThread[i] =
CreateThread(NULL,0,Consumer,(PVOID)&x [i], 0, &dwThreadId[i]);
                if(!hThread) printf(«main process: thread %d not ex-
ecute!», i);
        }
        WaitForMultipleObjects(k+m,hThread,TRUE,INFINITE);

        // закрытие критической секции
        DeleteCriticalSection(&ArraySection);
        return 0;
}
```

Варианты заданий

1. *Задача о парикмахере*. В тихом городке есть парикмахерская. Салон парикмахерской мал ходить там может только парикмахер и один посетитель. Парикмахер всю жизнь обслуживает посетителей. Когда в салоне никого нет, он спит в кресле. Когда посетитель приходит и видит спящего парикмахера, он будет его, садится в кресло и спит, пока парикмахер занят стрижкой. Если посетитель приходит, а парикмахер занят, то он встает в очередь и засыпает. После стрижки парикмахер сам провожает посетителя. Если есть ожидающие посетители, то парикмахер будит одного из них, и ждет пока тот сядет в кресло парикмахера и начинает стрижку. Если никого нет, он снова садится в свое кресло и засыпает до прихода посетителя. Создать многопоточное приложение, моделирующее рабочий день парикмахерской. Условную синхронизацию потоков выполнить с помощью критических секций или событий.

2. *Задача о Винни-Пухе или правильные пчелы.* В одном лесу жи-вут n пчел и один медведь, которые используют один горшок меда, вместимостью H глотков. Сначала горшок пустой. Пока горшок не наполнится, медведь спит. Как только горшок заполняется, медведь просыпается и съедает весь мед, после чего снова засыпает. Каждая пчела

многократно собирает по одному глотку меда и кладет его в горшок. Пчела, которая приносит последнюю порцию меда, будит медведя. Создать многопоточное приложение, моделирующее поведение пчел и медведя. Условную синхронизацию потоков выполнить с помощью критических секций.

3. *Задача о читателях и писателях.* Базу данных разделяют два типа процессов — читатели и писатели. Читатели выполняют транзакции, которые просматривают записи базы данных, транзакции писателей и просматривают и изменяют записи. Предполагается, что в начале БД находится в непротиворечивом состоянии (то есть отношения меж-ду данными имеют смысл). Каждая отдельная транзакция переводит БД из одного непротиворечивого состояния в другое. Для предотвра-щения взаимного влияния транзакций процесс-писатель должен иметь исключительный доступ к БД. Если к БД не обращается ни один из процессов-писателей, то выполнять транзакции могут одновременно сколько угодно читателей. Создать многопоточное приложение с потоками-писателями и потоками-читателями. Реализовать решение, используя критические секции.

4. *Задача об обедающих философах.* Пять философов сидят возле круглого стола. Они проводят жизнь, чередуя приемы пищи и размышления. В центре стола находится большое блюдо спагетти. Спагетти длинные и запутанные, философам тяжело управляться с ними, поэто-му каждый из них. чтобы сьесть порцию, должен пользоваться дву-мя вилками. *К* несчастью, философам дали только пять вилок. Между каждой парой философов лежит одна вилка, поэтому эти высококультурные и предельно вежливые люди договорились, что каждый будет пользоваться только теми вилками, которые лежат рядом с ним (слева и справа). Написать многопоточную программу, моделирующую пове-дение философов с помощью критических секций. Программа должна избегать фатальной ситуации, в которой все философы голодны, но ни один из них не может взять обе вилки (например, каждый из филосо-фов держит по одной вилки н не хочет отдавать ее). Решение должно быть симметричным, то есть все потоки-философы должны выполнять один и тот же код.

5. *Задача о каннибалах.* Племя из *n* дикарей ест вместе из боль-шого горшка, который вмещает *m* кусков тушеного миссионера. Когда дикарь хочет обедать, он ест из горшка один кусок, если только горшок

сторон, оставляя другую сторону активной. Например, клиент может сообщить серверу, что не будет больше передавать ему никаких дан-ных и закрывает соединение «клиент -> сервер», однако готов продол-жать принимать от него данные до тех пор, пока сервер будет их посы-лать, то есть хочет оставить соединение «сервер -> клиент» открытым. Дляэтогонеобходимовызватьфункцию*«intshutdown(SOCKETs,inthow)»*, передав в аргументе how одно из следующих значений: *SD_RECEIVE* для закрытия канала «сервер -> клиент», *SD_SEND* для закрытия канала «клиент -> сервер», и, наконец, *SD_BOTH* для закрытия обоих каналов. Последний вариант выгодно отличается от *closesocket* «мягким» за-крытием соединения — удаленному узлу будет послано уведомление о желании разорвать связь, но это желание не будет воплощено в действительность, пока тот узел не возвратит свое подтверждение. Таким образом, можно не волноваться, что соединение будет закрыто в самый неподходящий момент.

Внимание: вызов *shutdown* не освобождает от необходимости за-крытия сокета функцией *closesocket*!

Адрес.

С адресами как раз и наблюдается наибольшая путаница. Прежде всего, структура *sockaddr* определенная так:

```
struct sockaddr
{
   u_short sa_family; // семейство протоколов (как правило, AF_INET)
   char sa_data[14]; // IP-адрес узла и порт
};
```

Однако, теперь уже считается устаревшей, и в Winsock 2.x на смену ей пришла структура *sockaddr_in*, определенная следующим образом:

```
struct sockaddr_in
{
   short sin_family; // семейство протоколов (как правило, AF_INET)
   u_short sin_port; // порт
   struct in_addr sin_addr; // IP-адрес
   char sin_zero[8]; // хвост
};
```

пришло сообщение. Поскольку функция recvfrom заносит IP-адрес и номер порта клиента после получения от него сообщения, программис-ту фактически ничего не нужно делать — только передать *sendto* тот же самый указатель на структуру *sockaddr*, который был ранее передан функции recvfrem, получившей сообщение от клиента.

Еще одна деталь *— транспортный протокол UDP, на который опираются дейтаграммные сокеты, не гарантирует успешной до-ставки сообщений и эта задача ложиться на плечи самого разработ-чика. Решить ее можно, например, посылкой клиентом подтвержде-ния об успешности получения данных. Правда, клиент тоже не может быть уверен, что подтверждение дойдет до сервера, а не потеряется где-нибудь в дороге. Подтверждать же получение подтверждения — бессмысленно, так как это рекурсивно неразрешимо. Лучше вообще не использовать дейтаграммные сокеты на ненадежных каналах.*

Во всем остальном обе пары функций полностью идентичны и ра-ботают с теми самыми флагами — *MSG_PEEK* и *MSG_OOB*.

Все четыре функции при возникновении ошибки возвращают зна-чение *SOCKET_ERROR* (== -1).

Примечание: *в UNIX с сокетами можно обращаться точно так-же, как и с обычными файлами, в частности писать и читать в них функциями write и read. ОС Windows 3.1 не поддерживала такой воз-можности, поэтому при переносе приложений их UNIX в Windows все вызовы write и read должны были быть заменены на send и recv соот-ветственно. В Windows 95 с установленным Windows 2.x это упущение исправлено, теперь дескрипторы сокетов можно передавать функци-ям ReadFil, WriteFile, DuplicateHandle и др.*

Шаг последний.

Для закрытия соединения и уничтожения сокета предназначена функция *«int* ***closesocket*** *(SOCKET s)»*, которая в случае удачного за-вершения операции возвращает нулевое значение.

Перед выходом из программы необходимо вызвать функцию *«int* ***WSACleanup*** *(void)»* для деинициализации библиотеки *WINSOCK* и освобождения используемых этим приложением ресурсов.

Внимание: завершение процесса функцией *ExitProcess* автомати-чески не освобождает ресурсы сокетов!

Примечание: более сложные приемы закрытия соединения — протокол *TCP* позволяет выборочно закрывать соединение любой из

не пуст, иначе дикарь будит повара и ждет, пока тот не наполнит гор-шок. Повар, сварив обед, засыпает. Создать многопоточное приложе-ние, моделирующее обед дикарей. При решении задачи пользоваться критическими секциями.

6. *Задача о курильщиках.* Есть три процесса-курильщика и один процесс-посредник. Курильщик непрерывно скручивает сигареты и курит их. Чтобы скрутить сигарету, нужны табак, бумага и спички.
У одного процесса-курильщика есть табак, у второго — бумага, а у третьего — спички. Посредник кладет на стол по два разных случай-ных компонента. Тот процесс-курильщик, у которого есть третий ком-понент, забирает компоненты со стола, скручивает сигарету и курит. Посредник дожидается, пока курильщик закончит, затем процесс по-вторяется. Создать многопоточное приложение, моделируюшее пове-дение курильщиков и посредника. При решении задачи использовать критические секции.

7. *Задача о картинной галерее.* Вахтер следит за тем, чтобы в кар-тинной галерее было не более 50 посетителей. Для обозрения пред-ставлены 5 картин. Посетитель ходит от картины к картине. Посети-тель может покинуть галерею. Создать многопоточное приложение, моделирую шее работу картинной галереи.

8. *Задача о Винни-Пухе — 2 пли неправильные пчелы. N* пчел живет в улье, каждая пчела может собирать мед и сторожить улей (*N* > 3). Ни одна пчела не покинет улей, если кроме нее в нем нет других пчел. Каждая пчела приносит за раз одну порцию меда. Все-го в улей может войти тридцать порций меда. Винни-Пух спит пока меда в улье меньше половины, но как только его становится доста-точно, он просыпается и пытается достать весь мед из улья. Если в улье находится менее чем три пчелы, Винни-Пух забирает мед убегает, съедает мед и снова засыпает. Если в улье пчел больше, они кусают Винни-Пуха, он убегает, лечит укус, и снова бежит за медом. Создать многопоточное приложение, моделирующее поведе-ние пчел и медведя.

9. *Задача о нелюдимых садовниках.* Имеется пустой участок земли (двумерный массив) и план сада, который необходимо реализовать. Эту задачу выполняют два садовника, которые не хотят встречаться друг с другом. Первый садовник начинает работу с верхнего левого угла сада и перемешается слева направо, сделав ряд, он спускается вниз. Второй

садовник начинает работу с нижнего правого угла сада и перемещается снизу вверх, сделав ряд, он перемещается влево. Если садовник видит, что участок сада уже выполнен другим садовником, он идет дальше. Садовники должны работать параллельно. Создать многопоточное приложение, моделирую шее работу садовников. При решении задачи использовать критические секции.

10. *Задача о супермаркете.* В супермаркете работают два кассира, покупатели заходят в супермаркет, делают покупки и становятся в очередь к случайному кассиру. Пока очередь пуста, кассир спит, как только появляется покупатель, кассир просыпается. Покупатель спит в очереди, пока не подойдет к кассиру. Создать многопоточное приложе-ние, моделирующее рабочий день супермаркета.

11. *Задача о магазине.* В магазине работают три отдела, каждый отдел обслуживает один продавец. Покупатель, зайдя в магазин, делает покупки в произвольных отделах, и если в выбранном отделе продавец не свободен, покупатель становится в очередь и засыпает, пока прода-вец не освободится. Создать многопоточное приложение, моделирую-щее рабочий день магазина.

12. *Задача о больнице.* В больнице два врача принимают пациентов, выслушивают их жалобы и отправляют их или к стоматологу или к хирургу или к терапевту. Стоматолог, хирург и терапевт лечат пациента. Каждый врач может принять только одного пациента за раз. Пациенты стоят в очереди к врачам и никогда их не покидают. Создать многопоточное приложение, моделирующее рабочий день клиники.

13. *Задача о гостинице.* В гостинице 30 номеров, клиенты гостиницы снимают номер на одну ночь, если в гостинице нет свободных номеров, клиенты устраиваются на ночлег рядом с гостиницей и ждут, пока любой номер не освободится. Создать многопоточное приложе-ние, моделирующее работу гостиницы.

14. *Задача о гостинице — 2 (умные клиенты).* В гостинице 10 номеров с ценой 200 рублей, 10 номеров с ценой 400 рублей и 5 номеров с ценой 600 руб. Клиент, зашедший в гостиницу, обладает некоторой суммой и получает номер по своим финансовым возможностям, если тот свободен. Если среди доступных клиенту номеров нет свободных, клиент уходит искать ночлег в другое место. Создать многопоточное приложение, моделирующее работу гостиницы.

взведенным флагом *MSG_PEEK* возвращает количество уже переданных байт (вызов send не блокирует управления). На самом деле функция send игнорирует этот флаг!

Флаг *MSG_OOB* предназначен для передачи и приема срочных (Out Of Band) данных. Срочные данные не имеют преимущества перед другими при пересылке по сети, а всего лишь позволяют оторвать кли-ента от нормальной обработки потока обычных данных и сообщить ему «срочную» информацию. Если данные передавались функцией *send* с установленным флагом *MSG_OOB*, для их чтения флаг *MSG_ OOB* функции *recv* также должен быть установлен.

Замечание: *настоятельно рекомендуется воздержаться от использования срочных данных в своих приложениях. Во-первых, они совершенно необязательны — гораздо проще, надежнее и элегантнее вместо них создать отдельное TCP-соединение. Во-вторых, по поводу их реализации нет единого мнения и интерпретации различных производителей очень сильно отличаются друг от друга. Так, разработчики до сих пор не пришли к окончательному соглашению по поводу того, куда должен указывать указатель срочности: на последний байт срочных данных или на байт, следующий за последним байтом срочных данных. В результате отправитель никогда не имеет уверен-ности, что получатель сможет правильно интерпретировать его за-прос.*

Еще существует флаг *MSG_DONTROUTE*, предписывающий передавать данные без маршрутизации, но он не поддерживаться Winsock и поэтому здесь не рассматривается.

Дейтаграммный сокет также может пользоваться функциями send и *recv*, если предварительно вызовет connect (см. «Клиент: шаг третий»), но у него есть и свои, «персональные», функции: *«int* ***sendto*** *(SOCKET s, const char FAR * buf, int len, int fl ags, const struct sockaddr FAR * to, int tolen)»* и *«int* ***recvfrom*** *(SOCKET s, char FAR* buf, int len, int fl ags, struct sockaddr FAR* from, int FAR* fromlen)»*.

Они очень похожи на *send* и *recv* — разница лишь в том, что *sendto* и *recvfrom* требуют явного указания адреса узла, принимаемого или передаваемого данные. Вызов *recvfrom* не требует предварительного задания адреса передающего узла — функция принимает все пакеты, приходящие на заданный UDP-порт со всех IP-адресов и портов. Напротив, отвечать отправителю следует на тот же самый порт откуда

*(SOCKET s, const char FAR * buf, int len,int fl ags)»* и *«int* ***recv*** *(SOCKET s, char FAR* buf, int len, int fl ags)»* для посылки и приема данных соот-ветственно.

Функция *send* возвращает управление сразу же после ее выполнения, независимо от того, получила ли принимающая сторона наши данные или нет. При успешном завершении функция возвращает количество *передаваемых (не переданных!)* данных — то есть успешное завершение еще не свидетельствует от успешной доставке! В общем-то, протокол *TCP* (на который опираются потоковые сокеты) гарантирует успешную доставку данных получателю, но лишь при условии, что соединение не будет преждевременно разорвано. Если связь прервется до окончания пересылки, данные останутся непереданными, но вызывающий код не получит об этом никакого уведомления! А ошибка возвращается лишь в том случае, если соединение разорвано **до** вызова функции *send*!

Функция же *recv* возвращает управление только после того, как получит хотя бы один байт. Точнее говоря, она ожидает прихода целой *дейтаграммы*. Дейтаграмма — это совокупность одного или несколь-ких IP пакетов, посланных вызовом send. Упрощенно говоря, каждый вызов recv за один раз получает столько байтов, сколько их было посла-но функцией send. При этом подразумевается, что функции recv предо-ставлен буфер достаточных размеров, в противном случае ее придется вызвать несколько раз. Однако, при всех последующих обращениях данные будут браться из локального буфера, а не приниматься из сети, так как *TCP*-провайдер не может получить «кусочек» дейтаграммы, а только ее всю целиком.

Работой обоих функций можно управлять с помощью *флагов*, передаваемых в одной переменной типа int третьим слева аргументом. Эта переменная может принимать одно из двух значений: *MSG_PEEK* и *MSG_OOB*.

Флаг *MSG_PEEK* заставляет функцию recv просматривать дан-ные вместо их чтения. Просмотр в отличие от чтения не уничтожает просматриваемые данные. Некоторые источники утверждают, что при взведенном флаге *MSG_PEEK* функция recv не задерживает управле-ния, если в локальном буфере нет данных, доступных для немедлен-ного получения. Это неверно! Аналогично, иногда приходится встре-чать откровенно ложное утверждение о том, что якобы функция *send* со

15. *Задача о клумбе.* На клумбе растет 40 цветов, за ними непрерывно следят два садовника и поливают увядшие цветы, при этом оба садовника очень боятся полить одни и тот же цветок. Создать многопоточное приложение, моделирующее состояния клумбы и действия садовников. Для изменения состояния цветов создать отдельный поток.

Практическое занятие № 3.

Синхронизация процессов

Цель работы: изучить работу с мьютексами. Закрепить полученные в практической работе № 2 навыки по выделению критический секций кода до уровня разделяемых системных ресурсов.

Порядок выполнения практических заданий

1. Рассмотреть представленный пример, и разработать приложения на его основе.
2. Разработать алгоритм решения второго задания. Определить критические фрагменты алгоритма и возможные разделяемые ресурсы и защитить их мьютексами.
3. Реализовать алгоритм с применением функций *WinAPI* и протестировать его на нескольких примерах.

Длительность — 4 акад. часа.

Литературные источники

1. Таненбаум, Э. Распределенные системы. Принципы и парадигмы / Э.Таненбаум, Танненбаум, М. ванн Стесн. — СПб. : Питер, 2003. — 877 с.
2. Рихтер, Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер. — СПб. : Питер, 2001. — 752 с.
3. Эндрюс, Г.Р. Основы многопоточного, параллельного и распределенного программирования / Г.Р. Эндрюс. — М. : «Вильямс», 2003. — 512 с.
4. Гергель, В.П. Теория и практика параллельных вычислений / В.П. Гергель. — М. : ИНТУИР.РУ Интернет-Университет Информационных Технологий, 2007.

При успешном выполнении функция возвращает нулевое значение и ненулевое в противном случае.

Сервер: шаг четвертый.

Выполнив связывание, потоковый сервер переходит в режим ожидания подключений, вызывая функцию *«int **listen** (SOCKET s, int backlog)»*, где *s* — дескриптор сокета, а *backlog* — максимально допустимый размер очереди сообщений.

Размер очереди ограничивает количество одновременно обрабатываемых соединений, поэтому к его выбору следует подходить «с умом». Если очередь полностью заполнена, очередной клиент при попытке установить соединение получит отказ (*TCP* пакет с установленным флагом *RST*). В то же время максимально разумное количество подключений определяется производительностью сервера, объемом оперативной памяти и т. д.

Датаграммные серверы не вызывают функцию listen, так как работают без установки соединения и сразу же после выполнения связывания могут вызывать recvfrom для чтения входящих сообщений, минуя четвертый и пятый шаги.

Сервер: шаг пятый.

Извлечение запросов на соединение из очереди осуществляется функцией *«SOCKET accept (SOCKET s, struct sockaddr FAR* addr, int FAR* addrlen)»*, которая автоматически создает новый сокет, выполняет связывание и возвращает его дескриптор, а в структуру *sockaddr* заносит сведения о подключившемся клиенте (IP-адрес и порт). Если в момент вызова accept очередь пуста, функция не возвращает управление до тех пор, пока с сервером не будет установлено хотя бы одно соединение. В случае возникновения ошибки функция возвращает отрицательное значение.

Для параллельной работы с несколькими клиентами следует сразу же после извлечения запроса из очереди порождать новый поток (процесс), передавая ему дескриптор созданного функцией *accept* сокета, затем вновь извлекать из очереди очередной запрос и т. д. В противном случае, пока не завершит работу один клиент, сервер не сможет обслуживать всех остальных.

Все вместе.

После того как соединение установлено, потоковые сокеты могут обмениваться с удаленным узлом данными, вызывая функции *«int **send***

***Примечание*:** *за словом «обычно» стоит один хитрый примем программирования — вызов connect позволяет дейтаграмному сокету обмениваться данными с узлом не только функциями sendto, recvfrom, но* и *более удобными и компактными send и recv. Эта тонкость описана в Winsocket SDK и широко используется как самой Microsoft, так и сторонними разработчиками. Поэтому ее использование вполне безопасно.*

Первый слева аргумент — дескриптор сокета, возвращенный функ-цией *socket*; второй — указатель на структуру *«sockaddr»*, содержащую в себе адрес и порт удаленного узла, с которым устанавливается соеди-нение. Структура sockaddr используется множеством функций, поэто-му ее описание вынесено в отдельный раздел «Адрес» теоретической части данной практической работы. Последний аргумент сообщает функции размер структуры *sockaddr*.

После вызова connect, система предпринимает попытку установить соединение с указанным узлом. Если по каким-то причинам это сделать не удастся (адрес задан неправильно, узел не существует или «висит», компьютер находится не в сети), функция возвратит ненулевое значение.

Сервер: шаг третий.

Прежде чем сервер сможет использовать сокет, он должен связать его с локальным адресом. Локальный, как впрочем, и любой другой адрес Интернета, состоит из IP-адреса узла и номера порта. Если сер-вер имеет несколько IP-адресов, то сокет может быть связан как со все-ми ними сразу (для этого вместо IP-адреса следует указать константу *INADDR_ANY*, равную нулю), так и с каким-то конкретным одним.

Связывание осуществляется вызовом функции *«int* ***bind*** *(SOCKET s, const struct sockaddr FAR* name, int namelen)»*. Первым слева аргу-ментом передается дескриптор сокета, возвращенный функций *socket*, за ним следуют указатель на структуру sockaddr и ее длина (см. раздел «Адрес раз, адрес два...»).

Строго говоря, клиент также должен связывать сокет с локальным адресом перед его использованием, однако за него это делает функция connect, ассоциируя сокет с одним из портов, наугад выбранных из диа-пазона 1024—5000. Сервер же должен «садиться» на заранее опреде-ленный порт — например, 21 для FTP, 23 для telnet, 25 для SMTP, 80 для Web, 110 для POP3 и т. д. Поэтому ему приходится осуществлять связывание «вручную».

Теоретическая часть

Мьютексы в Windows

Для решения проблемы взаимного исключения между параллельными потоками, выполняющимися в контексте разных процессов, в операционных системах Windows используется объект ядра мьютекс. Слово мьютекс является переводом английского слова *mutex*, которое в свою очередь является сокращением от выражения *mutual exclusion*, что на русском языке значит взаимное исключение. Мьютекс находит-ся в сигнальном состоянии, если он не принадлежит ни одному пото-ку. В противном случае мьютекс находится в несигнальном состоянии. Одновременно мьютекс может принадлежать только одному потоку.

Создается мьютекс вызовом функции *CreateMutex*, которая имеет следующий прототип:

```
HANDLE CreateMutex(
LPSECURITY_ATTRIBUTES lpMutexAttributes, // атрибуты защиты
BOOL bInitialOwner, // начальный владелец мьютекса
LPCTSTR lpName // имя мьютекса
);
```

Пока значение параметра *LPSECURITY_ATTRIBUTES* будем устанавливать в *NULL*. Это означает, что атрибуты защиты заданы по умолчанию, то есть дескриптор мьютекса не наследуется и доступ к мьютексу имеют все пользователи. Теперь перейдем к другим параметрам.

Если значение параметра *bInitialOwner* равно *TRUE*, то мьютекс сразу переходит во владение потоку, которым он был создан. В противном случае вновь созданный мьютекс свободен. Поток, создавший мьютекс, имеет все права доступа к этому мьютексу.

Значение параметра *lpName* определяет уникальное имя мьютекса для всех процессов, выполняющихся под управлением операционной системы. Это имя позволяет обращаться к мьютексу из дру-гих процессов, запущенных под управлением этой же операционной системы. Длина имени не должна превышать значение *MAX_PATH*. Значением параметра *lpName* может быть пустой указатель *NULL*. В этом случае система создает безымянный мьютекс. Отметим также,

что имена мьютексов являются чувствительными к нижнему и верхнему регистрам.

В случае удачного завершения функция *CreateMutex* возвращает дескриптор созданного мьютекса. В случае неудачи эта функция возвращает значение *NULL*. Если мьютекс с заданным именем уже существует, то функция *CreateMutex* возвращает дескриптор этого мьютекса, а функция *GetLastError*, вызванная после функции *CreateMutex* вернет значение *ERROR_ALREADY_EXISTS*.

Мьютекс захватывается потоком посредством любой функции ожидания, а освобождается функцией *ReleaseMutex*, которая имеет сле-дующий прототип:

```
BOOL ReleaseMutex(
HANDLE hMutex // дескриптор мьютекса
);
```

В случае успешного завершения функция *ReleaseMutex* возвраща-ет значение *TRUE*, в случае неудачи — *FALSE*. Если поток освобождает мьютекс, которым он не владеет, то функция *ReleaseMutex* возвращает значение *FALSE*.

Для доступа к существующему мьютексу поток может использовать одну из функций *CreateMutex* или *OpenMutex*. Функция *CreateMutex* используется в тех случаях, когда поток не знает, создан или нет мьютекс с указанным именем другим потоком. В этом случае значение параметра *bInitialOwner* нужно установить в *FALSE*, так как невозможно определить какой из потоков создает мьютекс. Если поток использует для доступа к уже созданному мьютексу функцию *CreateMutex*, то он получает полный доступ к этому мьютексу. Для того чтобы получить доступ к уже созданному мьютексу, поток может также использовать функцию *OpenMutex*, которая имеет следующий прототип:

```
HANDLE OpenMutex(
DWORD dwDesiredAccess, // доступ к мьютексу
BOOL bInheritHandle // свойство наследования
LPCTSTR lpName // имя мьютекса
);
```


мандной строке линкера указать «ws2_32.lib». В Microsoft Visual Studio для этого достаточно нажать <Alt-F7>, перейти к закладке «Link» и к списку библиотек, перечисленных в строке «Object/Library modules», добавить «ws2_32.lib», отделив ее от остальных символом пробела.

Перед началом использования функций библиотеки *Winsock*, ее необходимо подготовить к работе вызовом функции *«int **WSAStartup** (WORD wVersionRequested, LPWSADATA lpWSAData)»*, передав в стар-шем байта слова *wVersionRequested* номер требуемой версии, а в млад-шем — номер подверсии.

Аргумент *lpWSAData* должен указывать на структуру *WSADATA*, в которую при успешной инициализации будет занесена информация о производителе библиотеки. Никакого особенного интереса она не представляет и прикладное приложение может ее игнорировать. Если инициализация проваливается, функция возвращает ненулевое значение.

Программирование сокета начинается с создания объекта «сокет». Это осуществляется функцией *«SOCKET **socket** (int af, int type, int protocol)»* Первый слева аргумент указывает на семейство используе-мых протоколов. Для Интернет-приложений он должен иметь значение *AF_INET*.

Следующий аргумент задает тип создаваемого сокета — *потоковый (SOCK_STREAM)* или *дейтаграммный (SOCK_DGRAM)* (еще существуют и сырые сокеты, но они не поддерживаются Windows).

Последний аргумент уточняет какой транспортный протокол следует использовать. Нулевое значение соответствует выбору по умолчанию: *TCP* — для потоковых сокетов и *UDP* для дейтаграммных. В большинстве случаев нет никакого смысла задавать протокол вручную и обычно полагаются на автоматический выбор по умолчанию.

Если функция завершилась успешно, она возвращает дескриптор сокета, в противном случае — *INVALID_SOCKET*.

Примечание: *дальнейшие шаги зависят от того, является ли приложение сервером или клиентом. Ниже эти два случая будут опи-саны раздельно.*

Клиент: шаг второй.

Для установки соединения с удаленным узлом потоковый сокет должен вызвать функцию *«int **connect** (SOCKET s, const struct sockaddr FAR* name, int namelen)»*. Датаграмные сокеты работают без установки соединения, поэтому обычно не обращаются к функции *connect*.

Отметим, что реализация сокетов в Unix и Windows значительно отличается, что создает очевидные проблемы.

Основное подспорье в изучении сокетов — *Windows Sockets 2 SDK*. SDK — это документация, набор заголовочных файлов и инстру-ментарий разработчика. Большинство книг, имеющиеся на рынке, явно уступают Microsoft в полноте и продуманности описания. Единствен-ный недостаток *SDK* — он полностью на английском (для некоторых студентов это очень существенно).

Обзор сокетов

Библиотека *Winsock* поддерживает два вида сокетов — ***синхронные*** *(блокируемые)* и ***асинхронные*** *(неблокируемые)*. Синхронные сокеты задерживают управление на время выполнения операции, а асинхронные возвращают его немедленно, продолжая выполнение в фоновом режиме, и, закончив работу, уведомляют об этом вызывающий код.

Сокеты позволяют работать со множеством протоколов и являют-ся удобным средством межпроцессорного взаимодействия, но в данной практической работе речь будет идти только о сокетах семейства про-токолов *TCP/IP*, использующихся для обмена данными между узлами сети Интернет. Все остальные протоколы, такие как *IPX/SPX, NetBIOS* могут быть изучены студентами самостоятельно.

Независимо от вида, сокеты делятся на два типа — ***потоковые*** и ***дейтаграммные***. Потоковые сокеты работают с установкой соединения, обеспечивая надежную идентификацию обоих сторон и гарантируют целостность и успешность доставки данных. Дейтаграмные сокеты работают без установки соединения и не обеспечивают ни идентификации отправителя, ни контроля успешности доставки данных, зато они заметно быстрее потоковых.

Выбор того или иного типа сокетов определяется транспортным протоколом, на котором работает сервер, клиент не может по своему желанию установить с дейтаграммным сервером потоковое соединение.

Замечание: *дейтаграммные сокеты опираются на протокол UDP, а потоковые — на TCP.*

Первый шаг.

Для работы с библиотекой Winsock 2.x в исходный тест программы необходимо включить директиву *«#include <winsock2.h>»*, а в ко-

Параметр *dwDesiredAccess* этой функции может принимать одно из двух значений:

– *MUTEX_ALL_ACCESS*;

– *SYNCHRONIZE*.

В первом случае поток получает полный доступ к мьютексу. Во втором случае поток может использовать мьютекс только в функциях ожидания, чтобы захватить мьютекс, или в функции *ReleaseMutex*, для его освобождения. Параметр *bInheritHandle* определяет свойство на-следования мьютекса. Если значение этого параметра равно *TRUE*, то дескриптор открываемого мьютекса является наследуемым. В против-ном случае — дескриптор не наследуется.

В случае успешного завершения функция *OpenMutex* возвращает дескриптор открытого мьютекса, в случае неудачи эта функция возвра-щает значение *NULL*.

Практическая часть

Пример 1.

Приложение создает три потока, которые сигнализируют системным динамиком с заданной периодичностью и частотой. Написать приложение, которое будет корректно реализовать заданную функциональность даже при запуске нескольких копий приложения.

Разрешение проблемы незнания программы о существовании сво-ей копии.

Мьютексы позволяют потоку получить в монопольное владение определенный ресурс и тем самым обеспечить сохранность данных. С его помощью мы сможем решить проблему незнания программы о существовании своей копии и конфликтов на этой почве.

Для начала объявим глобальные переменные:

– *volatile BOOL Exit*;

– *HANDLE hMutex*.

В функцию, создающую потоки добавим, перед созданием потоков, проверку на открытие мьютекса. То есть, если мьютекс с таким именем уже существует, значит копия программы уже есть и мы просто делаем *OpenMutex*, иначе *CreateMutex*.

Наша функция (фрагмент кода можно интегрировать в функцию нажатия на кнопку в форме):

```
if (OpenMutex(SYNCHRONIZE, FALSE, «myMutex») == NULL)
        hMutex = CreateMutex(NULL, FALSE, «myMutex»);
else
        hMutex = OpenMutex(SYNCHRONIZE, FALSE, «myMutex»);

HANDLE hFirst = CreateThread(NULL, 0, FirstFunc, NULL, NULL,
NULL); HANDLE hSecond = CreateThread(NULL, 0, SecondFunc, NULL,
NULL, NULL);
HANDLE hThird = CreateThread(NULL, 0, ThirdFunc, NULL,
NULL, NULL);

Функция потока:
DWORD WINAPI FirstFunc (LPVOID pParam)
{
        while(!Exit)
        {
                if (WaitForSingleObject(hMutex, INFINITE) ==
        WAIT_OB-JECT_0)
                {
                                for (int i = 0; i<= 4; i++)
                                        Beep(100, 250);
                        ReleaseMutex(hMutex);
                }
        }
        return (0);
}
```

Варианты заданий

В качестве заданий по вариантам Вам предлагается реализовать приложение из практической работы № 2 со смещением номера варианта от выполненного вами на 3 варианта. Синхронизацию потоков в данном приложении рекомендуется производить с помощью мьютексов.

Практическое занятие № 4. Сетевое взаимодействие в Windows

Цель работы: изучить механизм сокетов. Научиться разграничивать функциональность ПО между клиентской и серверной частью.

Порядок выполнения практических заданий

1. Рассмотреть представленные примеры, и разработать приложе-ния на их основе.
2. Разработать алгоритм решения третьего задания, с учетом разделения функциональности между клиентом и сервером. Определить формат обмена информацией между клиентом и сервером.
3. Реализовать алгоритм с применением функций *WinAPI* и протестировать его на нескольких примерах.

Длительность — 8 акад. часов.

Литературные источники

1. Таненбаум, Э. Распределенные системы. Принципы и парадиг-мы / Э. Таненбаум, Танненбаум, М. ванн Стесн. — СПб. : Питер, 2003. — 877 с.
2. Эндрюс, Г.Р. Основы многопоточного, параллельного и распределенного программирования / Г.Р. Эндрюс. — М. : «Вильямс», 2003. — 512 с.
3. Уолтон, Ш. «Создание сетевых приложений в среде Linux» / Ш. Уолтон. — 2001.

Теоретическая часть

Сокеты (*sockets*) представляют собой высокоуровневый унифицированный интерфейс взаимодействия с телекоммуникационными протоколами. В технической литературе встречаются различные переводы этого слова — их называют и гнездами, и соединителями, и патронами, и патрубками, и т. д. По причине отсутствия устоявшегося русскоязычного термина, в настоящем разделе сокеты будет именоваться сокетами и никак иначе.